Sprachbasierte Konstruktion sicherer Systeme

Detlef Marek

Institut für Informatik
der Technischen Universität München

Sprachbasierte Konstruktion sicherer Systeme

Detlef Marek

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. U. Baumgarten

Prüfer der Dissertation:

1. Univ.-Prof. Dr. P.P. Spies
2. Univ.-Prof. Dr. C. Eckert, Technische Universität Darmstadt
3. Univ.-Prof. Dr. J. Biskup, Universität Dortmund
   (schriftliche Beurteilung)

Die Dissertation wurde am 27. Juni 2002 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 06. Februar 2003 angenommen.

Kurzfassung

Im Zeitalter der Informationsgesellschaft kommt der Sicherheit der in den informationstechnischen Systemen gepeicherten und verarbeiteten Informationen eine zentrale Bedeutung zu. Zur Förderung der Akzeptanz moderner Informationstechnologien werden somit geeignete Konzepte und Verfahren für die Konstruktion und Realisierung qualitativ hochwertiger sicherer Rechensysteme benötigt.

In dieser Arbeit wird ein sprachbasierter *top–down* Ansatz zur Konstruktion sicherer verteilter Systeme vorgestellt. Dieser Ansatz ist dadurch charakterisiert, daß alle Eigenschaften eines Systems einschließlich der Sicherheitsanforderungen mit Hilfe programmiersprachlicher Konzepte auf hohem Abstraktionsniveau festgelegt werden. Die Realisierung eines mit den Sprachkonzepten konstruierten Systems und der für dieses festgelegten Sicherheitseigenschaften erfolgt automatisiert unter Rückgriff auf ein Spektrum von Sicherheitsmechanismen und –diensten der zugrundeliegenden Ausführungsbasis. Das hohe Abstraktionsniveau der entwickelten Sprachkonzepte liefert zum einen die Basis für das Schließen der Lücke, die in herkömmlichen *bottom–up* orientierten Ansätzen zwischen der Spezifikation von Sicherheitsanforderungen und deren Implementierung besteht. Zum anderen wird der Systementwickler von dem Einsatz realisierungsnaher Konzepte und Mechanismen zur Implementierung von Sicherheitsanforderungen befreit. Sicherheitslücken, die als Folge einer unsachgemäßen Nutzung derartiger Mechanismen auftreten können, werden somit vermieden. Der sprachbasierte Konstruktionsansatz ermöglicht es zudem, bereits zur Übersetzungszeit Aussagen über die Konsistenz der mit den Sprachkonzepten festgelegten Sicherheitseigenschaften zu gewinnen.

Ausgehend von einem Grundlagenteil werden in der Arbeit zunächst Sprachkonzepte zur Konstruktion sicherer verteilter Systeme entwickelt. Die Sprache INSEL$^+$ stellt als Erweiterung der verteilten objekt–basierten Programmiersprache INSEL neben einem Benutzer– und Rollenkonzept Möglichkeiten zur differenzierten Festlegung von Sichtbarkeitsbereichen sowie Konzepte zur Konstruktion zugriffskontrollierter Komponenten zur Verfügung. Für die Nutzung zugriffskontrollierter Komponenten können komplexe Zugriffsbeschränkungen in Form von Zugriffsrestriktionsausdrücken deklarativ festgelegt werden, womit die Implementierung eines breiten Spektrums anwendungsspezifischer Zugriffskontrollpolitiken unterstützt wird.

Im weiteren der Arbeit werden zur Analyse der Konsistenz der mit den INSEL$^+$–Konzepten festgelegten Zugriffsbeschränkungen Konsistenzkriterien und darauf aufbauende statische Analyseverfahren erarbeitet. Aus diesen Kriterien werden konstruktive Regeln abgeleitet, an denen sich ein Systementwickler hinsichtlich der Konstruktion eines INSEL$^+$–Systems mit konsistenten Zugriffsbeschränkungen orientieren kann.

Die mit den INSEL$^+$–Konzepten festgelegten Zugriffsbeschränkungen werden im Zuge einer systematischen *top–down* orientierten Realisierung automatisiert durchgesetzt. Das hierfür benötigte Spektrum an Realisierungskonzepten wird in dieser Arbeit ausgehend von der Sprachebene schrittweise über mehrere Realisierungsebenen hinweg entwickelt und konkretisiert.

Danksagung

Zunächst möchte mich ganz herzlich bei Herrn Prof. Dr. Peter Paul Spies für die langjährige Betreuung und Geduld bis zur Fertigstellung dieser Arbeit bedanken. Neben der inhaltlichen Unterstützung durch zahlreiche Diskussionen und Anregungen möchte ich inbesondere die organisatorische Unterstützung sowohl während meiner Zeit an seinem Lehrstuhl an der Technischen Universität München als auch nach meinem Wechsel in den hohen Norden nach Flensburg erwähnen.

Mein besonderer Dank gilt Frau Prof. Dr. Claudia Eckert für die langjährige und fruchtbare Zusammenarbeit. Während meiner Zeit in München war sie jederzeit bereit, Probleme zu diskutieren und Hilfestellungen zu leisten. Ihre eigenen Arbeiten, ihre stets konstruktive Kritik sowie ihre hilfreichen Anmerkungen zu ersten Fassungen des vorliegenden Textes haben diese Arbeit mit beeinflußt.

Erwähnen möchte ich an dieser Stelle auch Dr. Hans–Michael Windisch, der als Weggefährte am Lehrstuhl von Prof. Dr. Spies in München vor allem zu den praktischen Teilen dieser Arbeit Anregungen gegeben hat. Mein Dank gilt ebenfalls Frau Christina Preuss, die durch ihre freundschaftlichen Aufmunterungen für mich zu einer positiven Arbeitsatmosphäre am Lehrstuhl beigetragen hat und mir bei späteren Besuchen in München stets Unterschlupf gewährt hat.

Ganz besonders bedanken möchte ich mich bei meiner Frau Katharina Marek für ihre stets liebevolle Unterstützung, ihre Ermutigungen und ihre Geduld. Sie hat durch Worte und Taten wesentlich dazu beigetragen, daß ich an der „Dritten“ bzw. zwischenzeitlich „Vierten im Bunde“ auch in Flensburg weitergearbeitet habe und diese Arbeit schließlich erfolgreich vollenden konnte. Zudem hat sie die Mühe auf sich genommen, große Teile des vorliegenden Textes auf orthographische und grammatikalische Fehler sowie besondere Stilblüten Korrektur zu lesen.

Schließlich geht ein Dank an meine Eltern, die immer für mich da waren, nie Druck auf mich ausgeübt haben und dennoch die Hoffnung auf die Fertigstellung dieser Arbeit nicht aufgegeben haben.

Inhaltsverzeichnis

1 Zielsetzung und Lösungsweg **1**

2 Sichere Rechensysteme **7**

2.1 Grundlagen 7

2.1.1 Grundlegende Begriffe 7

2.1.2 Sicherheitsanforderungen und –bedrohungen 8

2.1.3 Sicherheitsdienste und –mechanismen 12

2.1.3.1 Kryptographische Verfahren 12

2.1.3.2 Identifikation und Authentifikation 17

2.1.3.3 Rechteverwaltung und –prüfung 19

2.2 Konstruktion sicherer Rechensysteme 21

2.2.1 Allgemeine Konstruktionsprinzipien 21

2.2.2 Systematischer Konstruktionsprozeß 22

2.3 Programmiersprachliche Konzepte zur Konstruktion sicherer Rechensysteme . 25

2.3.1 Allgemeine Konzepte und Paradigmen 26

2.3.2 Spezielle Konzepte 28

2.3.2.1 Capability–basierte Ansätze 28

2.3.2.2 Zugriffskontrollisten–basierte Ansätze 32

2.3.3 Fazit 33

3 Konzepte der Sprache INSEL **37**

3.1 INSEL–Systeme 37

3.2 Komponenten–Konzepte 39

3.2.1 Generatoren und Inkarnationen 39

3.2.2 DE–Inkarnationen 40

3.2.3 DA–Inkarnationen 40

3.2.4 DA–Generatoren 44

3.2.5 Beispiel 45

3.3 Entwicklungsmöglichkeiten und Systemstrukturen 47

3.3.1 Erzeugung und Auflösung von Komponenten 47
3.3.2 Systemstrukturen 50
3.4 Ausführungsphasen einer kanonischen Operation 55
3.5 Ausführungsumgebung einer DA–Inkarnation 57
3.6 INSEL als Implementierungssprache für sichere Systeme 60

4 Konzepte der Sprache INSEL$^+$ 63
4.1 Einordnung und Übersicht 64
4.2 INSEL$^+$–Systeme 66
4.2.1 Erweiterte Ein– und Ausgabemöglichkeiten 67
4.2.2 Benutzerintegration und Benutzerzugang 67
4.3 Konzepte zur differenzierten Festlegung der Ausführungsumgebung 73
4.3.1 Import– und Exportfestlegungen 73
4.3.2 Operationen–qualifizierte Zeiger 78
4.4 Konzepte zur Konstruktion zugriffskontrollierter Komponenten 82
4.4.1 Zugriffskontrollierte Komponenten 82
4.4.2 Views 86
4.4.3 Zugriffskontrollisten 91
4.4.4 Zugriffsrestriktionsausdrücke 98
4.4.4.1 Zugriffsrestriktionsprädikat `IN_ACL` 104
4.4.4.2 Zugriffsrestriktionsprädikat `ACCESSED` 108
4.4.4.3 Aufruferbeschreibung `Caller` und Kontextrestriktionen . . . 111
4.4.4.4 Bedingungen über lokale und globale DE–Inkarnationen . . . 120
4.5 Konzept zur Behandlung von Zugriffsverboten 123
4.6 Beispiele 131
4.6.1 Kontenverwaltungssystem einer Bank 131
4.6.1.1 Funktionale Anforderungen 131
4.6.1.2 Zugriffskontrollpolitik 133
4.6.1.3 INSEL$^+$–Implementierung 134
4.6.2 System zur Hausaufgabenverwaltung eines Übungskurses 140
4.6.2.1 Lösung unter Verwendung boolescher DE–Inkarnationen . . . 141
4.6.2.2 Lösung unter Verwendung des Zugriffsrestriktionsprädikats `ACCESSED` 142
4.7 Zusammenfassung 143

5 INSEL$^+$–Systeme mit konsistenten Rechtefestlegungen 145

5.1 Literaturüberblick . . . 146

5.1.1 Objektlokaler Konsistenzbegriff: Konsistenz bezogen auf positive und negative Rechte an einem Objekt . . . 146

5.1.2 Konsistenz bezogen auf Schachtelungsabhängigkeiten zwischen Objekten 148

5.1.3 Konsistenz bezogen auf funktionale Abhängigkeiten zwischen Operationen . . . 149

5.1.4 Fazit . . . 150

5.2 Konsistenz des Rechts eines INSEL$^+$–Systems . . . 151

5.2.1 Recht eines INSEL$^+$–Systems . . . 151

5.2.2 Konsistenzbegriff für INSEL$^+$–Systeme . . . 154

5.2.3 Maßnahmen zur Gewährleistung der Konsistenz . . . 158

5.3 Statische Konsistenzanalysen . . . 163

5.3.1 Attributierter statischer Aufrufgraph eines INSEL$^+$–Programms . . . 164

5.3.2 Potentielle Konsistenz . . . 179

5.3.2.1 Kriterien zur Analyse der Erfüllbarkeit eines Zugriffsrestriktionsausdrucks . . . 182

5.3.2.2 Kriterien zur Analyse der Widerspruchsfreiheit zweier Zugriffsrestriktionsausdrücke . . . 190

5.3.3 Absolute Konsistenz . . . 215

5.3.3.1 Kriterien zur Analyse der Konsistenz zweier Zugriffsrestriktionsausdrücke bzgl. eines Aufrufpfades . . . 216

5.3.3.2 Analyse der absoluten Konsistenz . . . 226

5.4 Konstruktion von INSEL$^+$–Systemen mit absolut konsistentem Recht . . . 233

5.5 Zusammenfassung . . . 244

6 Realisierung von INSEL$^+$–Systemen 247

6.1 Grundlagen . . . 248

6.1.1 Realisierungsaufgabe . . . 249

6.1.2 Realisierungsansatz . . . 250

6.1.3 Verwalterarchitektur . . . 252

6.2 Die Realisierungsebene 1 . . . 255

6.2.1 Charakteristika und Konkretisierung der Verwalter . . . 256

6.2.2 Akteursphärenverwalter . . . 260

6.2.2.1 Erzeugung von Inkarnationen . . . 260

6.2.2.2 Terminierung und Auflösung von Inkarnationen . . . 262

6.2.2.3 Realisierung des Operationen–orientierten Rendezvous . . . 265

6.2.2.4 Verwaltung von Zugriffskontrollinformationen . . . 267

6.2.2.5 Durchführung von Zugriffskontrollen . . . 273

6.2.2.6 Zusammenfassung . . . 281
6.2.3 Tickets . . . 283
6.2.3.1 Vergabe von Tickets . . . 284
6.2.3.2 Einsatz von Tickets . . . 286
6.2.3.3 Löschen von Tickets . . . 286
6.2.3.4 Realisierungsalternative für die Ticketverwaltung . . . 292
6.2.3.5 Kriterien für die Vergabe eines Tickets . . . 293
6.2.3.6 Zusammenfassung . . . 299
6.3 Realisierungsebene 2 . . . 300
6.3.1 Charakteristika . . . 300
6.3.2 Konkretisierung der Verwalter . . . 303
6.3.3 Realisierung stellenübergreifender Zugriffe . . . 305
6.3.4 Wechselseitige Authentifizierung von Akteursphärenverwaltern . . . 310
6.3.4.1 Anforderungen und Schlüsselverwalter . . . 310
6.3.4.2 Protokoll zur Registrierung eines öffentlichen Schlüssels . . . 314
6.3.4.3 Wechselseitige Authentifizierung zweier Akteursphärenverwalter . . . 316
6.3.5 Optimierungen . . . 318
6.4 Zusammenfassung . . . 323

7 Zusammenfassung und Ausblick **325**
7.1 Zusammenfassung . . . 325
7.2 Ausblick . . . 329

A INSEL$^+$–Syntax **331**

B INSEL$^+$–Beispielprogramm: Kontenverwaltungssystem einer Bank **336**

Abbildungsverzeichnis **365**

Tabellenverzeichnis **367**

Literaturverzeichnis **369**

Kapitel 1

Zielsetzung und Lösungsweg

Im Zeitalter der Informationsgesellschaft, das durch die Verbreitung der Informationstechnologie in nahezu allen Lebensbereichen gekennzeichnet ist, kommt der Sicherheit der in den informationstechnischen Systemen verarbeiteten und gespeicherten Informationen eine zentrale Bedeutung zu. Die Sicherheitsthematik spielt insbesondere im Kontext der zunehmenden Nutzung des Internet und der Möglichkeiten des elektronischen Handels (*Electronic Commerce*) eine wichtige Rolle. Das Gebiet der Sicherheit von Rechensystemen umfaßt ein breites Spektrum von Fragestellungen und Themenbereichen, die sich von technischen, über organisatorische bis hin zu juristischen Aspekten erstrecken. Diese Arbeit beschäftigt sich mit der Konstruktion sicherer verteilter Rechensysteme. Konzepte und Verfahren zur Konstruktion sicherer Rechensysteme sowie wirksame Maßnahmen, um Bedrohungen der unautorisierten Manipulation und der Vertraulichkeit der in diesen Systemen verarbeiteten und gespeicherten Informationen abzuwehren, sind eine wesentliche Voraussetzung für eine breite Akzeptanz und Nutzung der Möglichkeiten moderner Informationstechnologien.

Die Konstruktion sicherer verteilter Rechensysteme ist eine komplexe Aufgabe, die eine systematische Vorgehensweise erfordert. Aus Sicht des Systementwicklers besteht die Anforderung, daß die systemweit und in der Regel räumlich verteilt zur Verfügung stehenden Hard– und Software–Ressourcen einfach und transparent nutzbar sein sollten und daß anwendungsspezifische Sicherheitsanforderungen einfach festgelegt werden können. Der Anwendungsentwickler soll sich damit auf die Lösung seines Anwendungsproblems und der Formulierung der gestellten Sicherheitsanforderungen konzentrieren können, ohne Realisierungsrandbedingungen beachten zu müssen. Hierfür werden Konzepte auf hohem Abstraktionsniveau benötigt, die es ermöglichen, sichere verteilte Systeme angepaßt an die Erfordernisse des jeweiligen Anwendungsproblems zu konstruieren. Die Realisierung der mit diesen Konzepten konstruierten Systeme hat dann automatisiert und transparent gegenüber dem Anwendungsentwickler durch geeignete Infrastrukturen, wie zum Beispiel Übersetzer und Mechanismen des zugrundeliegenden Betriebssystems, zu erfolgen.

Betrachtet man die herkömmlichen Ansätze zur Konstruktion sicherer verteilter Systeme, so zeigt sich, daß diese die gestellten Anforderungen nicht oder lediglich unzureichend erfüllen. Dies gilt sowohl bezüglich der einfachen und transparenten Nutzung von Systemressourcen als auch hinsichtlich der Festlegung von Sicherheitsanforderungen. Zur Konstruktion einer verteilten Anwendung muß ein Anwendungsentwickler in der Regel neben den Konzepten der verwendeten Programmiersprache Konzepte der zugrundeliegenden Ausführungsumgebung bzw. des Betriebssystems, die im allgemeinen ein sehr niedriges Abstraktionsniveau haben, einsetzen. Da die heutigen Programmiersprachen keine speziellen Konzepte zur Festlegung

von Sicherheitsanforderungen anbieten, besteht für den Systementwickler die Notwendigkeit, die Durchsetzung der Sicherheitsanforderungen unter Nutzung vom *low–level* Sicherheitsmechanismen und –diensten der Ausführungsumgebung oder des Betriebssystems explizit zu programmieren. Ein Beispiel hierfür ist die Implementierung der Vergabe von Zugriffsrechten auf Basis grobgranularer Dateien mit einfachen betriebssystemseitig festgelegten Zugriffskontrollisten. Eine ggf. anwendungsspezifisch erforderliche differenzierte Vergabe von Berechtigungen für feingranulare Objekte läßt sich auf dieser Basis nicht oder nur mit sehr hohem Aufwand realisieren. Die Implementierung von Sicherheitsanforderungen bzw. –eigenschaften unter Nutzung derartiger *low–level* Mechanismen ist ein im allgemeinen mühsamer und damit fehlerträchtiger Vorgang. Hierbei besteht von vornherein die Gefahr, daß aufgrund von Implementierungsfehlern Sicherheitslücken entstehen und die Sicherheitsanforderungen nicht korrekt realisiert werden. Beispiele für derartige durch Implementierungsfehler aufgetretene Sicherheitsprobleme sind in der Literatur zahlreich bekannt und dokumentiert (siehe z.B. [LBMC94]). Um Sicherheitslücken zu schließen, die aus einem unsachgemäßen Einsatz realisierungsnaher *low–level* Konzepte resultieren, muß das Abstraktionsniveau der zur Implementierung sicherer Systeme genutzten Konzepte deutlich angehoben werden.

Im Rahmen eines systematischen Konstruktionsprozesses für sichere Systeme sind die auf Basis einer Bedrohungs– und Risikoanalyse an ein System gestellten Sicherheitsanforderungen ausgehend von einer zunächst verbalen Formulierung formal durch Entwicklung eines Sicherheitsmodells zu spezifizieren. Die so spezifizierten Sicherheitsanforderungen sind dann unter Nutzung einer Programmiersprache zu implementieren. Bei Einsatz der herkömmlichen Programmiersprachen besteht aus den oben genannten Gründen eine große Lücke zwischen der Spezifikation und der Implementierung der Sicherheitsanforderungen. Das Ziel der vorliegenden Arbeit besteht darin, mit einen sprachbasierten *top–down* Ansatz einen Beitrag zum Schließen dieser Lücke zu leisten und somit die Konstruktion qualitativ hochwertiger sicherer verteilter Systeme zu ermöglichen.

Der in dieser Arbeit vorgestellte sprachbasierte *top–down* Ansatz zur Konstruktion und Realisierung sicherer Systeme ist dadurch gekennzeichnet, daß alle Eigenschaften eines Systems einschließlich der Sicherheitsanforderungen mit Hilfe programmiersprachlicher Konzepte auf hohem Abstraktionsniveau festgelegt werden. Das mit den Sprachkonzepten konstruierte System und die festgelegten Sicherheitsanforderungen werden durch den entwickelten Übersetzer unter Rückgriff auf Sicherheitsmechanismen und –dienste der zugrundeliegenden Ausführungsbasis automatisiert so realisiert, daß das realisierte System die Sicherheitsanforderungen durchsetzt. Aufgrund des hohen Abstraktionsniveaus der Sprachkonzepte wird zum einen die bestehende Lücke, die in herkömmlichen *bottom–up* orientierten Ansätzen zwischen der Spezifikation und der programmiersprachlichen Implementierung besteht, erheblich verkleinert. Zum anderen wird der Systementwickler von dem Einsatz realisierungsnaher und an *low–level* Mechanismen orientierten Konzepten zur Implementierung von Sicherheitsanforderungen entlastet. So muß er zum Beispiel die Durchführung von Zugriffskontrollen und die Verwaltung hierfür benötigter Datenstrukturen, wie Zugriffskontrollisten oder Capabilities, nicht explizit programmieren, sondern lediglich deklarativ angeben, welche Bedingungen bei den entsprechenden Kontrollen zu überprüfen sind.

Für die Realisierung der mit den Sprachkonzepten anwendungsspezifisch festgelegten Sicherheitsanforderungen muß die zugrundeliegende Ausführungsbasis ein breites Spektrum von Sicherheitsmechanismen und –diensten bereitstellen. So werden neben Basismechanismen für die Durchführung von Zugriffskontrollen u.a. kryptographische Verfahren und darauf aufbauende Dienste wie zum Beispiel Verschlüsselungs– und Authentifikationsdienste benötigt. Die Auswahl der für die Realisierung der jeweiligen mit den Sprachkonzepten festgelegten Sicher-

heitsanforderungen einzusetzenden Mechanismen und Dienste erfolgt automatisiert durch die entwickelten Transformationswerkzeuge. Die *top–down* Vorgehensweise ermöglicht es, Alternativen für Realisierungen insbesondere im Hinblick auf Effizienzsteigerungen systematisch abzuleiten und durch Bereitstellung eines entsprechenden Spektrums von Realisierungskonzepten und –mechanismen umzusetzen.

In dieser Arbeit wird die im Rahmen des MoDiS–Projekts (*Model–oriented Distributed Systems*) [EW95, Eck96, PE97, EP98] entwickelte verteilte objekt–basierte Programmiersprache INSEL (*Integration and Separation supporting Language*) um Konzepte für die Konstruktion von Systemen mit anwendungsspezifisch festgelegten Zugriffskontrollpolitiken erweitert. Die so erweiterte Sprache wird INSEL$^+$ genannt. In dem MoDiS–Projekt werden Konzepte und Verfahren zur Spezifikation und automatisierten Realisierung paralleler und kooperativer Anwendungssysteme auf Basis vernetzter Hardware–Konfigurationen entwickelt. In MoDiS wird dazu ein sprachbasierter Ansatz verfolgt, in dem gemäß einer *top–down* orientierten Vorgehensweise die Realisierungskonzepte angepaßt an die Sprachkonzepte der zugrundeliegenden objekt–basierten Sprache INSEL entworfen werden. INSEL$^+$ stellt im Vergleich zu INSEL neben erweiterten Möglichkeiten zur differenzierten Festlegung von Sichtbarkeitsbereichen und einem Ausnahmebehandlungskonzept insbesondere Konzepte zur Konstruktion sogenannter zugriffskontrollierter Komponenten, für deren Nutzung komplexe Zugriffsbeschränkungen festgelegt werden können, zur Verfügung. Die INSEL$^+$–Konzepte unterstützen die Implementierung eines breiten Spektrums anwendungsspezifisch festgelegter Zugriffskontrollpolitiken. Obwohl die neuen Konzepte in ihrer konkreten Ausprägung auf die INSEL–Konzepte zugeschnitten sind, lassen sie sich mit geringfügigen Modifikationen auch in anderen geeigneten objekt–basierten bzw. objekt–orientierten Umgebungen einsetzen.

Die mit den INSEL$^+$–Konzepten festgelegten Zugriffsbeschränkungen sind im Zuge der systematischen *top–down* orientierten Realisierung eines INSEL$^+$–Systems automatisiert durchzusetzen. Da die eingeführten Sprachkonzepte unabhängig von Realisierungseigenschaften sind, bleiben für eine automatisierte an die anwendungsspezifischen Anforderungen angepaßte Realisierung genügend Freiheitsgrade offen. Das hierfür benötigte Spektrum an Realisierungskonzepten und –maßnahmen wird in dieser Arbeit ausgehend von der Sprachebene schrittweise über mehrere Realisierungsstufen hinweg angepaßt an die Sprachkonzepte entwickelt und konkretisiert.

Ein wichtiger Aspekt bei der Konstruktion sicherer Rechensysteme, der in herkömmlichen Ansätzen weitgehend unbeachtet bleibt, betrifft die Konsistenz der für ein System getroffenen Rechtefestlegungen. Die Forderung nach Konsistenz der Rechtefestlegungen besagt, daß die Rechte zur Nutzung von Komponenten des Systems widerspruchsfrei vergeben sein müssen, um die mit der Wahrnehmung eines Rechts verbundene Leistungsanforderung (siehe u.a. [Spi85]) erfüllen zu können und somit insbesondere Ausführungsabbrüche begonnener Operationsausführungen aufgrund fehlender Rechte zu vermeiden. Der in dieser Arbeit vorgestellte sprachbasierte Konstruktionsansatz eröffnet die Möglichkeit, bereits zur Übersetzungszeit durch geeignete statische Analysen Aussagen über die Konsistenz der mit den Sprachkonzepten spezifizierten Rechtefestlegungen zu gewinnen. Dabei festgestellte Inkonsistenzen sind in einem iterativen Prozeß von dem Systementwickler durch Modifikation der formulierten Zugriffsbeschränkungen zu beseitigen. Auf das Thema der Konsistenz wird in dieser Arbeit ausführlich eingegangen. Es werden Konsistenzkriterien und darauf aufbauende statische Analyseverfahren erarbeitet. Weiterhin werden aus diesen Kriterien konstruktive Leitlinien abgeleitet, an denen sich ein Systementwickler hinsichtlich der Konstruktion eines INSEL$^+$–Systems mit konsistentem Recht orientieren kann. Damit leistet die Arbeit einen wesentlichen Beitrag zur Konstruktion von Systemen mit konsistenten Rechtefestlegungen.

Zum Abschluß dieses einführenden Kapitels wird ein grober Überblick über den Aufbau und die weiteren Kapitel dieser Arbeit gegeben.

In **Kapitel 2** werden zunächst die für die Arbeit relevanten Begriffe sowie Konzepte und Verfahren aus dem Bereich der sicheren Rechensysteme eingeführt. Anschließend wird genauer auf die systematische Konstruktion sicherer Rechensysteme eingegangen, und es wird die Bedeutung, die den für die Implementierung dieser Systeme eingesetzten Programmiersprachen in diesem Konstruktionsprozeß zukommt, herausgestellt. Ausgehend von grundlegenden Eigenschaften, die eine für die Implementierung sicherer Systeme eingesetzte Programmiersprache haben sollte, werden Anforderungen an spezifische programmiersprachliche Konzepte zur Konstruktion sicherer Systeme erarbeitet.

Die imperative objekt–basierte Programmiersprache INSEL stellt eine geeignete Ausgangsbasis für die Entwicklung einer Sprache zur Konstruktion sicherer verteilter Systeme dar. Die der Sprache INSEL zugrundeliegenden Prinzipien und ihre wesentlichen Konzepte werden in **Kapitel 3** vorgestellt. Zudem werden notwendige Erweiterungen der Sprachkonzepte motiviert, die für die Implementierung von Systemen mit anwendungsspezifisch festgelegten Zugriffskontrollpolitiken benötigt werden.

Die motivierten Ergänzungen werden von der Sprache INSEL$^+$ bereitgestellt, die in **Kapitel 4** erklärt wird. INSEL$^+$ erweitert den Konzeptevorrat von INSEL neben einem Benutzer– und Rollenkonzept sowie zusätzlichen Möglichkeiten zur differenzierten Festlegung von Sichtbarkeitsbereichen bzw. Ausführungsumgebungen insbesondere um Konzepte zur Konstruktion sogenannter zugriffskontrollierter Komponenten. Für die Nutzung zugriffskontrollierter Komponenten können komplexe Zugriffsbeschränkungen auf Basis von Zugriffsrestriktionsausdrücken, die für die äußeren Operationen dieser Komponenten anzugeben sind, festgelegt werden. Beim Zugriff auf eine zugriffskontrollierte Komponente wird zur Laufzeit des Systems eine Zugriffskontrolle durchgeführt, die darin besteht, den für die entsprechende Operation festgelegten Zugriffsrestriktionsausdruck auszuwerten und damit zu überprüfen, ob die aufrufende Komponente das Recht zur Ausführung der Operation hat. Zur Signalisierung und Behandlung gegebenenfalls auftretender Zugriffsverbote stellt INSEL$^+$ ein einfaches Ausnahmebehandlungskonzept zur Verfügung. Die INSEL$^+$–Konzepte werden an einem durchgehenden Beispiel, dem Kontenverwaltungsverwaltungssystem einer Bank, veranschaulicht.

Kapitel 5 behandelt ausführlich das Thema der Konsistenz der in einem INSEL$^+$–System auf Basis der programmiersprachlich festgelegten Zugriffsbeschränkungen vergebenen Rechte. Nach einem kurzen Überblick über in der Literatur vorhandene Konsistenzbegriffe wird der Begriff des Rechts eines INSEL$^+$–Systems eingeführt und es wird definiert, wann das Recht eines INSEL$^+$–Systems konsistent ist. Anschließend werden die Kriterien und statischen Analyseverfahren erarbeitet, die es ermöglichen, bereits zur Übersetzungszeit Aussagen über die Konsistenz des Rechts eines INSEL$^+$–Systems zu gewinnen. Aus den Konsistenzkriterien werden schließlich Richtlinien abgeleitet, die bei der Konstruktion eines INSEL$^+$–Systems im Hinblick auf die Durchsetzung der Forderung nach Konsistenz des Rechts des Systems zu beachten sind.

In **Kapitel 6** wird auf die Realisierung von INSEL$^+$–Systemen auf einer verteilten Hardware–Konfiguration eingegangen. Der Schwerpunkt liegt dabei auf der Erläuterung der für die Durchsetzung des Rechts eines INSEL$^+$–Systems erforderlichen Realisierungskonzepte und –maßnahmen. Diese werden gemäß des verfolgten *top–down* Ansatzes ausgehend von der Sprachebene schrittweise über zwei Realisierungsebenen hinweg entwickelt und konkretisiert. Für die Durchführung der erforderlichen Zugriffskontrollen werden mehrere Realisierungsalternativen angegeben, die eine anwendungsangepaßte Realisierung der Zugriffskontrollen ermöglichen.

Kapitel 7 faßt den Inhalt und die wesentlichen Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf weiterführende Aufgabenstellungen.

Kapitel 2

Sichere Rechensysteme

Das Gebiet der Sicherheit in Rechensystemen überdeckt ein breites Spektrum von Fragestellungen und Themenbereichen, die sich von technischen, über organisatorische bis zu juristischen Aspekten erstrecken. In diesem Kapitel wird ein Überblick über die für diese Arbeit relevanten Themenbereiche sicherer Rechensysteme gegeben. In Abschnitt 2.1 werden zunächst grundlegende Begriffe sowie die wesentlichen Konzepte und Verfahren aus dem Bereich der sicheren Rechensysteme eingeführt. Abschnitt 2.2 geht schwerpunktmäßig auf die systematische Konstruktion sicherer Rechensysteme ein und ordnet die vorliegende Arbeit in den Konstruktionsprozeß ein. Einen wesentlichen Beitrag zur Konstruktion sicherer Rechensysteme können die für die Implementierung dieser Systeme eingesetzten Programmiersprachen leisten. In Abschnitt 2.3 werden zunächst grundlegende Eigenschaften angegeben, die eine Programmiersprache, die für die Implementierung sicherer Systeme eingesetzt wird, erfüllen sollte. Anschließend werden aus der Literatur bekannte spezielle programmiersprachliche Konzepte, die die Implementierung sicherer Rechensysteme unterstützen sollen, vorgestellt und bewertet. Hieraus werden abschließend Anforderungen an geeignete programmiersprachliche Konzepte zur Konstruktion sicherer Rechensysteme erarbeitet.

2.1 Grundlagen

Dieser Abschnitt dient dazu, die grundlegenden Begriffe, Standardkonzepte und –verfahren aus dem Bereich der sicheren Rechensysteme einzuführen.

2.1.1 Grundlegende Begriffe

Obwohl das Gebiet der Sicherheit von Rechensystemen bereits seit vielen Jahren intensiv bearbeitet wird und gerade im Zeitalter von *Internet* und *Electronic Commerce* zunehmend an Bedeutung gewinnt, erfolgt in der Literatur immer noch keine einheitliche Begriffsbildung und –verwendung. Im folgenden werden deshalb die für diese Arbeit wesentlichen Begriffe und Termini eingeführt und gegeneinander abgegrenzt. Die Begriffsbildung orientiert sich dabei an der in der Literatur überwiegend verwendeten Bedeutung.

Rechensystem, Objekte, Subjekte und Zugriffsrechte

Ein **Rechensystem** ist ein geschlossenes (zentrales) oder offenes (physikalisch verteiltes), dynamisches und technisches System mit der Fähigkeit zur Speicherung und Verarbeitung von Information. **Information** ist ein Abstraktum, das in einem Rechensystem in Form von

Daten bzw. Datenobjekten repräsentiert, d.h. konkretisiert wird. Der Zugriff auf die Datenobjekte ist zu beschränken und zu kontrollieren, um die Information, die durch die Daten repräsentiert wird, zu schützen. Es wird unterschieden zwischen **passiven Objekten**, die die Fähigkeit haben, Information zu speichern (z.B. Dateien, Programme), und **aktiven Objekten** mit der Fähigkeit, sowohl Information zu speichern als auch zu verarbeiten (z.B. Prozesse). Die Objekte eines Systems sind somit die zu schützenden Einheiten.

Die Objekte eines Systems sind über wohldefinierte Schnittstellen (Operationen, Methoden) von anderen Objekten des Systems oder seiner Umwelt benutzbar. Zu der Umwelt des Systems gehören insbesondere seine Benutzer. Die Benutzer eines Systems und alle aktiven Objekte, die im Auftrag von Benutzern in dem System aktiv sein können (z.B. Prozesse, Server), werden als **Subjekte** des Systems bezeichnet. Die Subjekte greifen auf Objekte zu. Der Zugriff auf Objekte und damit auf die zu schützende Information ist für Subjekte zu beschränken und zu kontrollieren. Hierzu sind **Zugriffsrechte** für die Nutzung von Objekten festzulegen und an die Subjekte zu vergeben. Die Zugriffe von Subjekten auf Objekte müssen kontrolliert werden, so daß nur autorisierte Zugriffe möglich sind. Ein Zugriff ist **autorisiert**, wenn das Subjekt das entsprechende Zugriffsrecht für das Objekt besitzt. Subjekte und Objekte müssen hierzu eindeutig identifiziert und ihre Identifikation muß verifiziert sein, d.h. Subjekte und Objekte müssen **authentifiziert** sein.

Sicherheitsbegriffe

Unter **Datensicherheit** (engl.: *proctection*) wird die Gesamtheit aller Maßnahmen verstanden, die notwendig sind, um unautorisierte Zugriffe von Subjekten zu verhindern. Mit dem Begriff **Datensicherung** ist demgegenüber im engeren Sinn der Schutz vor dem Verlust von Daten gemeint, z.B. durch Erstellung von Sicherungskopien bzw. Backups. Der Begriff **Datenschutz** (engl.: *privacy*) bezieht sich auf die Fähigkeit eines Benutzers, die Weitergabe von Informationen, die ihn persönlich betreffen, zu kontrollieren (siehe hierzu auch das Bundesdatenschutzgesetz (BDSG) [BDS94]).

Unter der **Rechtssicherheit** bzw. allgemein **Sicherheit** (engl.: *security*) eines Systems wird die Eigenschaft eines Systems verstanden, nur solche Zustände anzunehmen, die zu keiner unautorisierten Informationsveränderung und –gewinnung führen. Die Sicherheit eines Systems ist damit ein Maß für die Fähigkeit des Systems, den Bedrohungen zu begegnen und die spezifizierten Sicherheitsanforderungen (vgl. Abschnitt 2.1.2), wie z.B. die mit den vergebenen Zugriffsrechten festgelegten Zugriffsbeschränkungen, durchzusetzen.

Der Begriff der **Verläßlichkeit** (engl.: *dependability*) geht über den Begriff der Sicherheit hinaus. Ein System ist verläßlich, wenn es keine funktional unzulässigen Zustände annehmen kann, d.h. seine Ist–Funktionalität mit der spezifizierten Soll-Funktionalität übereinstimmt, und gewährleistet ist, daß die spezifizierten Funktionen vollständig erbracht werden.

In dieser Arbeit werden sichere Rechensysteme im Sinne von rechtssicheren Systemen betrachtet. Das bedeutet, daß im Vordergrund Konzepte und Verfahren zur Konstruktion und Realisierung rechtssicherer Systeme stehen.

2.1.2 Sicherheitsanforderungen und –bedrohungen

Die Sicherheitsanforderungen, die an ein Rechensystem gestellt werden, beziehen sich im allgemeinen auf die Datenintegrität, die Vertraulichkeit und die Verfügbarkeit (vgl. z.B. [BSI92]). Weitere Anforderungen betreffen die Verbindlichkeit von ausgeführten Aktionen und die Anonymität von Benutzern bzw. Kommunikationsbeziehungen. Die gestellten Sicherheitsanfor-

derungen sind gegenüber vielfältigen Bedrohungen, denen ein System ausgesetzt ist, durchzusetzen.

Sicherheitsanforderungen

Die Forderung nach Gewährleistung der **Datenintegrität** (engl.: *integrity*) besagt, daß die Objekte des Systems nur über wohldefinierte Operationen genutzt werden, daß die Objekte authentisch sind, d.h. die für sie festgelegte Funktionalität erbringen, und daß die Zugriffsrechte für die Nutzung der Objekte kontrolliert und systematisch an die Subjekte des Systems vergeben werden. Die Gewährleistung der Datenintegrität erfordert Maßnahmen, die die Objekte vor unautorisierten Zugriffen und damit vor unerlaubter Modifikation bzw. Manipulation schützen (vgl. [KH91]).

Die Forderung nach **Vertraulichkeit** besagt, daß in einem System keine unautorisierte Kenntnisnahme von Informationen erfolgen darf. Zur Durchsetzung dieser Forderung sind neben dem Einsatz kryptographischer Verfahren (siehe Abschnitt 2.1.3) Maßnahmen zur Kontrolle des Informationsflusses zwischen den Subjekten des Systems erforderlich, die sicherstellen, daß keine Information zu unautorisierten Subjekten, d.h. Subjekten, die keinen Zugriff auf diese Information haben dürfen, durchsickern kann. Das Problem einer derartigen unautorisierten Informationsgewinnung wurde bereits 1973 als Confinement–Problem [Lam73] bezeichnet. In diesem Zusammenhang spielen die in einem System potentiell auftretenden Kanäle, über die Information fließen kann, eine wichtige Rolle. Diese Kanäle lassen sich in Speicherkanäle, legitime und verdeckte Kanäle klassifizieren. Speicherkanäle ergeben sich aus Objekten, die von Subjekten gemeinsam genutzt werden können (z.B. Dateien). Die legitimen Kanäle sind die Kanäle, die ein Subjekt in der Regel für den Informationsaustausch nutzt, wie z.B. Nachrichten oder Parameter bei Operationsaufrufen. **Verdeckte Kanäle** (engl.: *covert channels*) sind Kanäle, die nicht für einen Informationsaustausch vorgesehen sind, aber dazu mißbraucht werden können. Beispielsweise kann über die Seitenfehlerrate oder die Anzahl versendeter Nachrichten eine verdeckte Information an den Beobachter dieser Merkmale übermittelt werden.

Grundlage für die Durchsetzung von Integritäts– und Vertraulichkeitsanforderungen ist die **Authentizität** der Subjekte und Objekte des Systems. Es ist sicherzustellen, daß Information nur an authentifizierte und berechtigte Subjekte weitergegeben wird bzw. von diesen verändert wird und daß nur authentifizierte Objekte Operationen zur Informationsgewinnung und –verarbeitung anbieten.

Die Forderung nach **Verfügbarkeit** (engl.: *availability*) besagt, daß in einem System keine unautorisierte Beeinträchtigung der Funktionalität des Systems und seiner Objekte erfolgen darf. Die Verfügbarkeitsanforderung soll gewährleisten, daß ein Subjekt, das Zugriffsrechte besitzt, diese auch wahrnehmen kann, und das System die angebotenen Funktionen vollständig und korrekt erbringt. Im Zusammenhang mit der Gewährleistung der Verfügbarkeit sind u.a. Maßnahmen zum Schutz vor sogenannten *Denial–of–Service* Angriffen sowie aus den Bereichen Fehlertoleranz und Fehlervermeidung von Interesse (vgl. [Rei91]).

Die Forderung nach **Verbindlichkeit** (engl.: *liability*) oder auch Nichtabstreitbarkeit (engl.: *non–repudiation*) besagt, daß es einem Subjekt nicht möglich sein darf, eine von ihm durchgeführte Aktion im Nachhinein abzustreiten. Diese Forderung spielt insbesondere im Kontext des *Electronic Commerce* eine wichtige Rolle, um zum Beispiel zu gewährleisten, daß der Abschluß eines elekronischen Kaufvertrags nicht nachträglich von einem der beiden Vertragspartner abgestritten werden kann. Zur Durchsetzung der Verbindlichkeitsanforderung kommen vor allem kryptographische Verfahren zur Erstellung digitaler Signaturen zum Einsatz (siehe hierzu Abschnitt 2.1.3.1).

Die Forderung nach **Anonymität** von Benutzern bzw. Kommunikationsbeziehungen hat zum Ziel, die Erstellung persönlicher Profile einzelner Benutzer, wie Bewegungs–, Kommunikations– oder Zugriffsprofile, durch unautorisierte Dritte durch die Verschleierung von Aufenthaltsorten und Kommunikationsbeziehungen zu vermeiden. Diese Forderung gewinnt in Zusammenhang mit dem *Surfen* im *Internet* und dem elektronischen Bezahlen sowie der elektronischen Geldkarte zunehmend an Bedeutung. Geeignete Maßnahmen zur Gewährleistung der Anonymität werden zum Beispiel in [Cha85] beschrieben.

Sicherheitsbedrohungen

Ein Rechensystem ist vielfältigen Bedrohungen ausgesetzt, gegenüber denen die gestellten Sicherheitsanforderungen durchzusetzen sind. Die in [BSI92] genannten Grundbedrohungen betreffen den Verlust der Datenintegrität, der Vertraulichkeit und der Verfügbarkeit. Die Bedrohungen ergeben sich dabei aus passiven und aktiven Angriffen auf das System. Unter einem **Angriff** (engl.: *attack*) wird der Versuch bzw. die erfolgreiche Durchführung eines nicht autorisierten Zugriffs auf das System verstanden.

Ein **passiver Angriff** liegt vor, wenn jemand zum Beispiel durch das Abhören von Datenleitungen (engl.: *wiretapping*) in vernetzten Systemen oder durch das unautorisierte Lesen von Dateien, Informationen erlangt, bezüglich der er nicht autorisiert ist. Passive Angriffe betreffen somit die unautorisierte Informationsgewinnung und stellen damit eine Bedrohung der Vertraulichkeit dar.

Aktive Angriffe (engl.: *tampering*) nehmen demgegenüber Einfluß auf die Objekte eines Systems. Sie umfassen zum Beispiel die unautorisierte Modifikation von Daten, das Verändern, Entfernen oder Einfügen von Datenpaketen eines Nachrichtenstroms und die Maskierung eines Benutzers als berechtigtes Subjekt. Aktive Angriffe bedrohen somit die Datenintegrität. Zu den aktiven Angriffen gehören auch die sogenannten *Denial–of–Service* Attacken, die zum Ziel haben, die Funktionalität eines Systems bzw. einzelner Objekte zu beeinträchtigen und somit eine Bedrohung der Verfügbarkeit darstellen. Beispiele für *Denial–of–Service* Angriffe sind das Überfluten eines Kommunikationsnetzes mit einer großen Anzahl von Nachrichten, so daß eine Überlastung des Netzes eintritt und es nicht mehr verfügbar ist, oder die sogenannte *Sync–floading* Attacke, bei der eine große Anzahl spezieller Nachrichtenpakete an einen Server geschickt wird, die letztlich zum Absturz des Servers führen (*ping of death*).

Gegen die Bedrohungen der Datenintegrität und der Vertraulichkeit, die sich aus passiven und aktiven Angriffen ergeben, bieten kryptographische Verfahren einen geeigneten Schutz. Passive Angriffe lassen sich durch den Einsatz kryptographischer Verfahren zwar nicht verhindern, sie werden durch deren Anwendung jedoch wirkungslos gemacht. Zur Abwehr aktiver Angriffe sind kryptographische Verfahren allein nicht ausreichend. Hierfür werden zusätzliche Maßnahmen benötigt, wie die Berechnung von Prüfsummen auf Basis von Hash–Verfahren zur Erkennung von Modifikationen, die Einführung von Sequenznummern zur Kontrolle der Nachrichtenreihenfolge oder Zeitstempel zur Erkennung von Wiedereinspielungen von Nachrichten. Auf kryptographische Verfahren wird in Abschnitt 2.1.3.1 noch näher eingegangen.

Abhängig von der Funktionalität und der Einsatzumgebung eines zu konstruierenden Rechensystems besitzen die angegebenen Sicherheitsanforderungen und Bedrohungen ein unterschiedliches Gewicht für das System. Für eine Bücher–Datenbank eines elektronischen Buchhandels, in der über das *Internet* öffentlich recherchiert werden kann, bestehen zum Beispiel keine Anforderungen hinsichtlich der Vertraulichkeit. Es ist jedoch zu gewährleisten, daß keine unautorisierte Modifikation der in dieser Datenbank gespeicherten Informationen, wie zum Beispiel die Preise der Bücher, möglich ist. Im Gegensatz dazu bestehen zum Beispiel für die

Datenbank des Verkehrszentralregisters beim Kraftfahrt–Bundesamt, in der personenbezogene Daten über im Straßenverkehr auffällige Personen gespeichert sind, hohe Vertraulichkeits– und Integritätsanforderungen, die durch den Einsatz aufwendiger Maßnahmen gewährleistet werden. Für ein zu konstruierendes sicheres System ist somit zunächst eine Bedrohungs– und Risikoanalyse durchzuführen, um die tatsächlichen Bedrohungen, denen das System ausgesetzt ist, und deren Relevanz für das System zu ermitteln (vgl. hierzu auch [BSI92]). Auf Basis der Ergebnisse der Bedrohungsanalyse sind dann die spezifischen Sicherheitsanforderungen des Systems festzulegen und durch eine Sicherheitspolitik zu formulieren.

Sicherheitspolitiken

Die **Sicherheitspolitik** eines Systems legt die Menge der Regeln und Maßnahmen fest, die zum Schutz der sensiblen Information vor den für das System relevanten Bedrohungen einzusetzen sind. Die Sicherheitspolitik legt dabei insbesondere fest, welche Subjekte welche Zugriffsrechte an welchen Objekten haben dürfen und welche Information einem Subjekt zugänglich sein darf. Das Spektrum möglicher Sicherheitspolitiken ist breit und variiert mit den jeweiligen Anwendungssystemen. In der Literatur werden traditionell zwei große Klassen von Sicherheitspolitiken unterschieden.

Die Klasse der **benutzerbestimmbaren** Politiken (engl.: *discretionary policy* oder *discretionary access control*, kurz DAC) ist dadurch charakterisiert, daß einzelne Benutzer die Zugriffsrechte an den Objekten, die sie besitzen, das sind in der Regel die Objekte, die sie erzeugt haben, individuell vergeben können. Die benutzerbestimmbaren Politiken basieren auf dem Owner–Prinzip. Dies besagt, daß der Erzeuger eines Objekts alle Zugriffsrechte an dem erzeugten Objekt erhält inklusive des sogenannten Owner–Rechts, das zur Vergabe von Rechten an diesem Objekt berechtigt.

Für die Klasse der **systembestimmten** bzw. systemglobalen Politiken (engl.: *mandatory policy* oder *mandatory access control*, kurz MAC) gilt, daß die Zugriffsrechte auf der Basis systemglobaler Regeln und Festlegungen vergeben und kontrolliert werden. Ein bekanntes Beispiel für eine derartige Festlegung besteht darin, eine geordnete Menge von Sicherheitsklassen einzuführen (z.B. 'öffentlich', 'vertraulich', 'streng vertraulich') und jedes Subjekt und jedes Objekt einer dieser Sicherheitsklassen zuzuordnen. Die systembestimmte Politik legt dann fest, daß Information nur gemäß der festgelegten Ordnung aufwärts fließen darf, z.B. von 'öffentlich' nach 'vertraulich'. Für jedes klassifizierte Subjekt ist damit festgelegt, welche Sicherheitsklassen diesem Subjekt maximal zugänglich sein können. So darf zum Beispiel ein Subjekt, das als 'vertraulich' eingestuft ist, lediglich auf Objekte lesend zugreifen, die entweder als 'öffentlich' oder 'vertraulich' klassifiziert sind. Der Zugriff auf 'streng vertraulich' eingestufte Objekte ist für dieses Subjekt verboten. Politiken, die den Informationsfluß auf Basis klassifizierter Subjekte und Objekte beschränken, werden auch Multi–Level Sicherheitspolitiken genannt.

Eine Sicherheitspolitik kann auch aus diesen beiden Klassen kombiniert werden, so daß Zugriffe, die gemäß einer benutzerbestimmbaren Politik erlaubt sind, durch die Hinzunahme systembestimmter Regeln eingeschränkt werden.

Neben der Klassifikation in benutzerbestimmbare und systembestimmte Politiken lassen sich Sicherheitspolitiken dahingehend unterscheiden, ob es sich um eine Zugriffskontrollpolitik oder eine Informationsflußpolitik handelt. Eine **Zugriffskontrollpolitik** ist dadurch charakterisiert, daß die Zugriffe auf Objekte durch die Festlegung von Zugriffsbeschränkungen bzw. die Vergabe von Zugriffsrechten beschränkt und kontrolliert werden. Benutzerbestimmbare Politiken sind ein Beispiel für Zugriffskontrollpolitiken. **Informationsflußpolitiken** konzentrieren sich demgegenüber auf die Festlegung von Informationsflußbeschränkungen, d.h. sie

spezifizieren zulässige und verbotene Informationsflüsse zwischen Objekten bzw. Subjekten. Ein Beispiel hiefür sind die oben erwähnten Multi–Level Sicherheitspolitiken. Zugriffs– und Informationsflußbeschränkungen lassen sich ebenfalls in einer Sicherheitspolitik kombinieren. Wichtig ist, daß eine Sicherheitspolitik anwendungsspezifisch formuliert wird, um den Sicherheitsanforderungen des jeweiligen Anwendungssystems gerecht zu werden. Zur formalen Spezifikation von Sicherheitspolitiken dienen Sicherheitsmodelle. Auf diese wird in Abschnitt 2.2 noch näher eingegangen.

Die mit der Sicherheitspolitik eines Systems festgelegten Sicherheitsanforderungen sind unter Nutzung von Sicherheitsdiensten und –mechanismen zu realisieren und zu gewährleisten. In dem folgenden Abschnitt werden die wichtigsten Verfahren und Mechanismen zur Realisierung sicherer Systeme kurz vorgestellt.

2.1.3 Sicherheitsdienste und –mechanismen

Zur Realisierung sicherer Systeme werden eine Reihe von Basissicherheitsdiensten, die auch Sicherheitsgrundfunktionen genannt werden, benötigt. Die wesentlichen Basissicherheitsdienste, die ausreichend sind, um ein breites Spektrum von Sicherheitsanforderungen zu realisieren, umfassen Maßnahmen zur Identifikation und Authentifikation insbesondere im Rahmen der Zugangskontrolle, zur Rechteverwaltung und –prüfung sowie zur Gewährleistung der Kommunikationssicherheit. Ein Großteil dieser Dienste und Maßnahmen basiert auf dem Einsatz kryptographischer Verfahren. Im folgenden wird deshalb zunächst eine kurze Einführung in kryptographische Verfahren gegeben, um anschließend die darauf aufbauenden Dienste näher erläutern zu können.

2.1.3.1 Kryptographische Verfahren

Kryptographische Verfahren (auch kryptographische Systeme oder kurz Kryptosysteme genannt) legen fest, wie Daten oder Nachrichten, die als Klartext vorliegen, in Kryptotext (auch Geheimtext genannt) transformiert, d.h. verschlüsselt werden und wie der entsprechende Kryptotext wieder in Klartext zurück tranformiert, d.h. entschlüsselt wird. Die ursprüngliche Entwicklung und Anwendung kryptographischer Verfahren läßt sich auf den Wunsch zurückführen, die in Daten bzw. Nachrichten codierten Informationen gegenüber unautorisierten Dritten – Angreifer genannt – geheimzuhalten. Der Schutz beruht dabei darauf, daß nur die Kenntnis eines speziellen kryptographischen Schlüssels es erlaubt, die in einem Kryptotext verschlüsselten Daten wieder in Klartext zu entschlüsseln und somit die darin codierte Information zu erhalten.

Ein kryptographisches System wird beschrieben durch ein Tupel $(\mathcal{M}, \mathcal{C}, EK, DK, E, D)$ mit:

1. der nichtleeren endlichen Menge von Klartextnachrichten $\mathcal{M}$,
2. der nichtleeren endlichen Menge von Kryptotextnachrichten $\mathcal{C}$,
3. der nichtleeren Menge von Verschlüsselungsschlüsseln EK,
4. der nichtleeren Menge von Entschlüsselungsschlüsseln DK, wobei es zwischen EK und DK eine Bijektion $f : EK \longrightarrow DK$ gibt mit: $K_D = f(K_E)$, $K_E \in EK, K_D \in DK$,
5. dem Verschlüsselungsalgorithmus $E : \mathcal{M} \times EK \longrightarrow \mathcal{C}$, wobei E linkstotal und injektiv ist,

6. dem Entschlüsselungsalgorithmus $D : \mathcal{C} \times DK \longrightarrow \mathcal{M}$ mit :

$$\forall M \in \mathcal{M} \ : \ D(E(M, K_E), K_D)) = M \quad \text{mit} \quad K_E \in EK, K_D \in DK \quad \text{und} \quad f(K_E) = K_D$$

Die Stärke oder Sicherheit eines kryptographischen Verfahrens wird danach bestimmt, wie schwer es für einen Angreifer ist, der Kenntnis der Algorithmen E und D sowie des Kryptotextes $C \in \mathcal{C}$ und eventuell zusätzlicher Information hat, den Klartext $M \in \mathcal{M}$ in der Regel durch Bestimmung des Entschlüsselungsschlüssels K_D zu erhalten. Die Wissenschaft von den Methoden der unbefugten Entschlüsselung von Daten zum Zweck der Rückgewinnung der ursprünglichen Information wird Kryptoanalysis genannt. Ein kryptographisches System ist **absolut sicher**, wenn es auch bei Vorhandensein unbeschränkter Rechenleistung und bei Kenntnis von beliebig vielen korrespondierenden Klartext/Kryptotext–Paaren nicht möglich ist, den Klartext bzw. den Entschlüsselungsschlüssel zu bestimmen. Das einzige kryptographische Verfahren, das als absolut sicher gilt, ist die *Vernam–Chiffre*. Dort wird zur Ver– und Entschlüsselung einer Zeichenfolge der Länge n eine zufällig erzeugte Zeichenfolge gleicher Länge als Schlüssel verwendet, der nur einmal benutzt wird. Der Schlüssel wird deshalb auch *one–time–pad* genannt. Ein Kryptosystem ist **praktisch sicher**, wenn kein Verfahren bekannt ist, mit dem bei Einsatz maximal verfügbarer Rechenleistung der Klartext bzw. der Entschlüsselungsschlüssel in vertretbarem Kosten– und Zeitaufwand bestimmt werden kann. Für die heutzutage in sicherheitskritischen Bereichen eingesetzten kryptographischen Verfahren wird im allgemeinen gefordert, daß sie praktisch sicher sind.

Es werden zwei Klassen von kryptographischen Verfahren unterschieden: die Private–Key und die Public–Key Verfahren.

Private–Key Verfahren, die auch symmetrische Kryptosysteme genannt werden, sind dadurch gekennzeichnet, daß der Verschlüsselungs– und der Entschlüsselungsschlüssel gleich sind bzw. der eine leicht aus dem anderen ableitbar ist. Das bedeutet, daß zwei Kommunikationspartner einen gemeinsamen, geheimen Schlüssel zur Ver– und Entschlüsselung verwenden. Dieser geheime Schlüssel muß sicher zwischen den Kommunikationspartnern ausgetauscht werden, so daß kein unautorisierter Dritter Kenntnis von diesem Schlüssel erlangen kann. Hierfür werden Schlüsselaustauschverfahren und –protokolle eingesetzt, die in der Regel auch die wechselseitige Authentifizierung der beiden Kommunikationspartner beinhalten. Auf sie wird in Abschnitt 2.1.3.2 noch näher eingegangen. Private–Key Verfahren sind in vielen Varianten seit langem in Gebrauch. Sie basieren im wesentlichen auf den grundlegenden Verschlüsselungstechniken der Transposition und der Substitution. Transposition (Permutation) ist die Umstellung von Einheiten des Klartextes nach einem vorgegebenen Schema. Bei einer Substitution wird jede Einheit eines Klartextes durch eine Einheit des Kryptotextes ersetzt. Da symmetrische Kryptosysteme, die allein auf Transposition oder Substitution beruhen, keine ausreichende Sicherheit bieten, werden diese beiden Techniken so miteinander verknüpft, daß mehrere Transpositionen und Substitutionen unterschiedlicher Art hintereinander ausgeführt werden. Gemäß den Einheiten, die verschlüsselt werden, unterscheidet man Strom– und Blockchiffren. Bei einer Stromchiffre wird der Klartext Bit–für–Bit verschlüsselt, während bei einer Blockchiffre der Klartext in Blöcke fester Länge aufgespalten wird, die dann verschlüsselt werden.

Das wohl bekannteste und heutzutage noch weit verbreitete Private–Key Verfahren ist der Data Encryption Standard (DES), der 1977 durch das National Bureau of Standards (NBS) zum US-Verschlüsselungsstandard für die Kommunikation zwischen und mit staatlichen Institutionen genormt wurde. Der DES ist eine Blockchiffre, die 64–Bit Eingabeblöcke mit einem 56–Bit Schlüssel verschlüsselt und einen 64–Bit Ausgabeblock erzeugt. Eine ausführliche Beschreibung des DES und seiner Sicherheitseigenschaften ist zum Beispiel in [Hor85]

und [FR94] zu finden. Der DES gilt aufgrund zwischenzeitlich bekannt gewordener Sicherheitsmängel (vgl. [BS92, Mat93]) und seiner vergleichweise geringen Schlüssellänge heutzutage als nicht mehr praktisch sicher. Ein Schlüsselraum der Größe 2^{56} ist bei heutiger Rechnertechnologie viel zu klein, um eine ausreichende kryptographische Sicherheit zu bieten. Bereits in [Wie93] wurde die Konstruktion einer Maschine zur Schlüsselraumsuche bestehend aus speziellen DES–Chips angegeben, die eine volle Schlüsselraumsuche in ca. 70 Stunden durchführen könnte. In [Fou98] wird eine DES–Cracker Maschine beschrieben, die aus Standard–Chips besteht und somit relativ kostengünstig ist, und mit der es gelungen ist, den DES–Schlüssel, der in der sogenannten DES Challenge II verwendet wurde, in weniger als drei Tagen zu brechen. Vor diesem Hintergrund ist nachvollziehbar, daß eine Verschlüsselung mit lediglich einem frei wählbaren 40–Bit Schlüssel, so wie sie bis vor kurzem in US–amerikanischen Sicherheitsprodukten aufgrund der dort geltenden Exportrestriktionen erfolgte, als kryptographisch sehr schwach gilt. Um die Sicherheit zu erhöhen, wird heute vielfach eine mehrfache Verschlüsselung mit dem DES unter Nutzung verschiedener Schlüssel durchgeführt. Eine bekannte Variante ist der sogenannte Triple–DES, bei dem eine dreifache Verschlüsselung mit zwei oder drei voneinander unabhängigen Schlüsseln erfolgt. Als Nachfolger des DES wurde 2001 der Advanced Encryption Standard (AES) vom National Institute of Standards and Technology (NIST) der USA genormt und als FIPS Standard 197 veröffentlicht ([NIS01]). Als kryptographischen Algorithmus, der im AES zum Einsatz kommt, entschied sich das NIST nach einem längeren öffentlichen Auswahlprozeß für den nach seinen Entwicklern John Daemen und Vincent Rijmen benannten Rijndael–Algorithmus. Der Rijndael–Algorithmus bietet drei Schlüssellängen (128, 192 und 256 Bit), so daß je nach gestellten Sicherheitsanforderungen eine ausreichende Schlüssellänge gewählt werden kann. Weitere technische Einzelheiten zum AES und dem Rijndael–Algorithmus sind unter `http://www.nist.gov/aes` verfügbar. Ein weiteres Beispiel für ein Private–Key Verfahren mit einer nach heutigem Stand der Technik als ausreichend geltenden Schlüssellänge von 128 Bit ist der IDEA (International Data Encryption Algorithm). Der IDEA wird zusammen mit weiteren symmetrischen Kryptoverfahren ausführlich in [Sch95] erläutert.

Bei Private–Key Verfahren besteht das Problem, daß die Kommunikationspartner einen gemeinsamen, geheimen Schlüssel sicher miteinander austauschen müssen, bevor sie unter Verwendung dieses Schlüssels vertraulich miteinander kommunizieren können. Mitte der 70'er Jahre wurde von Diffie und Hellman [DH76] mit den **Public–Key Verfahren** eine neue Klasse krytographischer Verfahren vorgeschlagen, bei denen das Problem des geheimen Schlüsselaustauschs nicht besteht. In Public–Key Verfahren, die auch als asymmetrische Kryptosysteme bezeichnet werden, werden verschiedene Schlüssel zur Ver- und Entschlüsselung benutzt. Jeder beteiligte Kommunikationspartner besitzt ein Schlüsselpaar bestehend aus einem Verschlüsselungsschlüssel, der veröffentlicht wird und somit allen anderen Kommunikationsteilnehmern bekannt ist, und einem Entschlüsselungsschlüssel, der geheimzuhalten ist. Dementsprechend wird der Verschlüsselungsschlüssel auch öffentlicher Schlüssel (engl.: *public key*) und der Entschlüsselungsschlüssel geheimer Schlüssel (engl.: *secret key*) genannt. Der Sender einer Nachricht verschlüsselt diese mit dem öffentlichen Schlüssel des Empfängers, der die verschlüsselte Nachricht dann mit seinem geheimen Schlüssel entschlüsselt. Es ist also kein geheimzuhaltender Schlüssel zwischen den Kommunikationspartnern auszutauschen, sonderen lediglich die Authentizität der öffentlichen Schlüssel zu gewährleisten. Für die Sicherheit eines Public–Key Verfahrens muß gewährleistet sein, daß sich aus einem öffentlichen Schlüssel nicht oder zumindest nicht mit vertretbarem Aufwand der dazugehörige geheime Schlüssel bestimmen läßt. Dementsprechend bilden Einwegfunktionen die Basis asymmetrischer Kryptosysteme. Eine Einwegfunktion (engl.: *oneway function*) ist eine Funktion, deren Funktionswerte mit vertretbarem Aufwand berechenbar sind, deren Inverse jedoch, obwohl

sie existieren, nur mit sehr großem Aufwand ermittelt werden können. Die Problematik liegt dabei nicht im Finden eines Algorithmus, sondern in dem großen Rechen–, Speicher– und Zeitaufwand zur Berechnung des Algorithmus. Bei Einwegfunktionen mit Falltür (engl.: *trapdoor oneway function*) ist die Berechnung des Inversen mit Hilfe einer geheimzuhaltenden Zusatzinformation (die *trapdoor* Information) effizient durchführbar. Die bekannten Public–Key Verfahren basieren überwiegend auf sich aus zahlentheoretischen Problemen ergebenden Einwegfunktionen, für deren Lösung zur Zeit keine effizienten Algorithmen bekannt sind. Hierzu gehören vor allem das Faktorisierungsproblem großer Primzahlen und das Problem des diskreten Logarithmus. Neue Möglichkeiten für die Entwicklung von Public–Key Verfahren ergeben sich aus dem Forschungsbereich der Elliptischen Kurven.

Public–Key Verfahren können neben der Anwendung zur Geheimhaltung von Informationen auch zur Erstellung **digitaler Signaturen** verwendet werden, wenn sie folgende Bedingung erfüllen (vgl. Seite 13):

$$\forall M \in \mathcal{M} \ : \ E(D(M, K_D), K_E)) = D(E(M, K_E), K_D)) = M$$

Die Erstellung einer digitalen Signatur zu einer Nachricht durch den Absender ermöglicht es dem Empfänger, die Authentizität des Senders und die Integrität der empfangenen Nachricht zu prüfen. Der Absender einer Nachricht signiert diese, indem er für diese durch Anwendung einer Hashfunktion zunächst eine Prüfsumme (Hashwert oder auch *message digest* genannt) erzeugt, die dann anschließend mit dem geheimen Schlüssel des Senders „verschlüsselt“ wird. Das Ergebnis dieser Berechnung ist die digitale Signatur der Nachricht, die der eigentlichen Nachricht angehängt wird. Der Empfänger kann die Signatur überprüfen, indem er sie mittels des öffentlichen Schlüssels des Absenders „entschlüsselt“ und somit die Prüfsumme erhält. Anschließend berechnet er mit der Hashfunktion die Prüfsumme zu der von ihm erhaltenen Nachricht und vergleicht das Ergebnis mit der entschlüsselten Prüfsumme. Stimmen die Prüfsummen überein, ist die Authentizität und die Integrität der Nachricht belegt. An Hashfunktionen für den Einsatz zur Erstellung digitaler Signaturen werden spezifische Anforderungen gestellt, auf die hier jedoch nicht weiter eingegangen wird (siehe hierzu z.B. [Sch95]).

Ein Problem beim Einsatz von Public–Key Verfahren betrifft die Gewährleistung der Authentizität der öffentlichen Schlüssel. Zur Lösung dieses Problem werden Zertifikate verwendet, die von sogenannten Zertifizierungsstellen (engl.: *Certification Authority* oder auch *Trustcenter*) ausgestellt werden. Ein Zertifikat ist ein digital signiertes Dokument, das die Zugehörigkeit eines öffentlichen Schlüssels zu einer Person oder einer Institution bescheinigt. Ein Zertifikat besteht im allgemeinen aus dem Namen der Person bzw. der Institution, ihres öffentlichen Schlüssels, dem Verfallsdatum des Schlüssels sowie dem Namen der ausstellenden Zertifizierungsstelle. Das Zertifikat ist mit dem geheimen Schlüssel dieser Zertifizierungsstelle signiert. Die Zertifizierungsstelle ist dafür verantwortlich, die Identität einer Person, für die ein Zertifikat ausgestellt werden soll, eindeutig festzustellen, zum Beispiel durch persönliches Erscheinen und Vorlage des Personalausweises. Die Zertifikate werden in der Regel in Verzeichnisdiensten veröffentlicht und können dort von anderen Kommunikationsteilnehmern abgefragt werden, die durch Überprüfung der digitalen Signatur des Zertifikats feststellen können, ob es sich um ein gültiges Zertifikat handelt. Hierzu muß wiederum der öffentliche Schlüssel der entsprechenden Zertifizierungsstelle authentisch bekannt sein. Die Authentizität dieses Schlüssels läßt sich wiederum durch ein Zertifikat einer übergeordneten Zertifizierungsstelle bestätigen. Im allgemeinen entsteht also eine Hierarchie von Zertifizierungsstellen mit einer Wurzelinstanz, deren öffentlicher Schlüssel auf einem vertrauenswürdigen Weg allgemein bekannt gemacht worden ist. Eine derartige Infrastruktur von Zertifizierungsstellen wird auch als Public–Key Infrastruktur (PKI) bezeichnet.

Digitale Signaturen und die für ihre Überprüfung erforderlichen Zertifikate gewinnen mit der Verbreitung des elektronischen Handels (*Electronic Commerce*) sowie des elektronischen Rechts- und Geschäftsverkehrs mit der öffentlichen Verwaltung (*Electronic Government*) zunehmend an Bedeutung. Dies zeigt sich auch darin, daß in Deutschland bereits 1997 mit dem sogenannten Signaturgesetz, das als Bestandteil des Informations– und Kommunikationsdienste–Gesetzes (IuKDG) im August 1997 in Kraft getreten ist ([Sig97]), frühzeitig die erforderlichen rechtlichen Rahmenbedingungen für den Einsatz digitaler Signaturen geschaffen wurden. Als zuständiges staatliches Organ für die Genehmigung signaturgesetzkonformer Zertifizierungsstellen wurde die Regulierungsbehörde für Telekommunikation und Post (RegTP) festgeschrieben, die Anfang 1999 auch die Wurzelinstanz in Betrieb genommen hat. Das deutsche Signaturgesetz von 1997 hat wichtige Impulse gegeben für die im Dezember 1999 veröffentlichte EG–Richtlinie über gemeinschaftliche Rahmenbedingungen für elektronische Signaturen. Die Umsetzung der EG–Richtlinie machte eine Neufassung des deutschen Signaturgesetzes von 1997 erforderlich, die als „Gesetz über Rahmenbedingungen für elektronische Signaturen (Signaturgesetz – SigG)“ im Mai 2001 in Kraft getreten ist ([Sig01]).

Das bekannteste Beispiel für ein Public–Key Verfahren ist das RSA–Verfahren, das bereits 1978 von Ronald Rivest, Adi Shamir und Leonard Adleman veröffentlicht wurde [RSA78]. Das RSA–Verfahren basiert auf dem Faktorisierungsproblem für Produkte sehr großer Primzahlen, d.h. darauf, daß der Aufwand zur Bestimmung großer Primzahlen im Vergleich zum Aufwand der Faktorisierung großer natürlicher Zahlen klein ist. Ein Teil eines öffentlichen RSA–Schlüssels besteht aus dem Produkt zweier großer Primzahlen, dem sogenannten Modul. Um eine ausreichende Sicherheit zu gewährleisten, sollte der Modul bei heutigem Stand der Technik mindestens eine Länge von 1024 Bit haben. Das RSA–Verfahren ist in vielen kommerziellen Produkten im Einsatz und ist Bestandteil offizieller Standards und Spezifikationen (z.B. dem Internet–Standard PEM (*Internet Privacy Enhanced Mail*, [Ken93]) oder der MailTrust–Spezifikation des deutschen TeleTrust–Vereins). Mit dem RSA–Verfahren ist auch die Erstellung digitaler Signaturen möglich. Weitere Public–Key Verfahren, mit denen digitale Unterschriften erstellt werden können, sind zum Beispiel das ElGamal–Verfahren [ElG85] und der vom US–amerikanischen National Institute of Standards and Technologie (NIST) vorgeschlagene *Digital Signature Standard* DSS, die beide auf dem diskreten Logarithmus–Problem basieren.

Kryptographische Verfahren werden insbesondere zur Gewährleistung einer sicheren, d.h. vertraulichen und vor Manipulationen geschützten Kommunikation über ein unsicheres Netz eingesetzt. Abhängig davon, auf welcher Kommunikationsprotokollschicht die Ver– und Entschlüsselung erfolgt, wird zwischen Leitungsverschlüsselung und Ende–zu–Ende Verschlüsselung unterschieden (siehe z.B. [VK83]). Bei der Leitungs– oder auch Verbindungsverschlüsselung (engl.: *link encryption*) werden die Daten unmittelbar vor ihrer Übertragung verschlüsselt. Die Daten sind somit nur während der Übertragung verschlüsselt und liegen innerhalb der Rechner im Klartext vor. Eine Leitungsverschlüsselung kann mittels spezieller Hardware, sogenannter Kryptoboxen, sehr effizient durchgeführt werden. Durch eine Ende–zu–Ende Verschlüsselung (engl.: *end–to–end encryption*) wird eine Nachricht verschlüsselt zwischen Sender und Empfänger übermittelt. Die Verschlüsselung erfolgt hierbei auf einer der oberen Protokollschichten. Im Gegensatz zur Leitungsverschlüsselung, bei der eine Verschlüsselung physikalischer Verbindungen erfolgt, wird die Ende–zu–Ende Verschlüsselung auf logische Verbindungen zwischen Benutzern angewendet. Eine Ende–zu–Ende Verschlüsselung ermöglicht es damit, einzelne Benutzer zu authentifizeren, während die Authentifikation bei der Leitungsverschlüsselung auf Rechner beschränkt bleibt. Ein Beispiel für eine Ende–zu–Ende Verschlüsselung ist der sichere Austausch von E-Mail gemäß des S/MIME–Standards

oder der deutschen MailTrust–Spezifikation. Verbindungsverschlüsselung kommt zum Beispiel im Informationsverbund Berlin–Bonn (IVBB) in Form von Kryptoboxen zwischen den einzelnen Netzknoten zum Einsatz. Um eine sichere Kommunikation bei der Nutzung von TCP/IP–Protokollen und darauf aufsetzender Dienste zu ermöglichen, wurde zwischenzeitlich eine Reihe von Protokollen entwickelt (z.B. SSL, S-HTTP, IPSEC), die zwischen den Protokollschichten der Leitungs– und Ende–zu–Ende Verschlüsselung anzusiedeln sind.

Neben dem Einsatz kryptographischer Verfahren werden zur Gewährleistung der Kommunikations– und Netzwerksicherheit Firewall–Systeme benötigt. Ein Firewall–System hat die Aufgabe, jeglichen Netzverkehr zwischen externen, als unsicher geltenden Netzen und einem internen sicheren Netz zu kontrollieren, mit dem Ziel, unautorisierte Zugriffe von außen auf Komponenten des internen Netzes zu verhindern. Auf Firewall–Systeme wird hier nicht weiter eingegangen. Eine gute Einführung in die Firewall–Thematik liefert zum Beispiel [Poh97].

2.1.3.2 Identifikation und Authentifikation

Bereits in Abschnitt 2.1.2 wurde darauf hingewiesen, daß als Grundlage für die Durchsetzung von Integritäts– und Vertraulichkeitsanforderungen die Authentizität der Subjekte und Objekte eines Rechensystems zu garantieren ist. Die Gewährleistung der Authentizität von Subjekten ist Aufgabe der Zugangskontrolle des Systems. Im Rahmen der Zugangskontrolle ist ein Subjekt eindeutig zu identifizieren, und die Korrektheit der behaupteten Identität ist nachzuweisen. Die Identifikation und Authentifikation erfolgt hierbei auf Basis von Authentifizierungsverfahren (auch Authentisierungsverfahren genannt), die sich in drei grundlegende Klassen einteilen lassen.

Authentifizierungsverfahren auf der Basis von **Wissen** überprüfen die Identität eines Subjekts anhand spezifischen Wissens, das das Subjekt vorweisen muß. In der Praxis am häufigsten anzutreffen sind Paßwortverfahren, bei denen sich der Benutzer durch die Kenntnis eines geheimen Paßworts oder einer persönlichen Identifikationsnummer (PIN) authentifizieren muß. Die Probleme und Sicherheitsrisiken von Paßwortverfahren sind allgemein bekannt, weshalb seit einiger Zeit Authentifizierungsverfahren auf der Basis von persönlichem **Besitz** zunehmende Verbreitung finden. Hier kommen vor allem chipkarten–basierte Verfahren zum Einsatz. Eine Chipkarte oder *Smart Card* ist eine Ausweiskarte, auf der die für die Identifikation und Authentifikation eines Subjekts benötigten Informationen, wie zum Beispiel kryptographische Schlüssel, sicher gespeichert sind und die sich unter Einsatz kryptographischer Verfahren gegenüber dem System authentifiziert. Die Authentifikation des Benutzers gegenüber der Chipkarte erfolgt in der Regel durch eine PIN oder ein Paßwort, also durch spezifisches Wissen. Bei chipkarten–basierten Verfahren erfolgt somit die Authentifikation eines Subjekts durch Besitz und Wissen. Authentisierungsverfahren auf Basis von Wissen und/oder Besitz haben den Nachteil, daß die Identifizierungsmerkmale nicht fest an die zu authentifizierende Person gebunden sind, zum Beispiel können Paßwörter weitergegeben oder geknackt werden. Dieses Problem wird durch Authentifizierungsverfahren auf der Basis biometrischer Merkmale behoben. **Biometrische Verfahren** basieren darauf, daß jeder Mensch über einzigartige, nicht kopierbare Körper– und Verhaltensmerkmale verfügt. Beispiele hierfür sind Fingerabdrücke, das Frequenzspektrum der Stimme oder die Netzhaut. Für die Authentifizierung auf Basis biometrischer Merkmale werden spezielle Geräte (Sensoren) benötigt, mit denen das jeweilige Merkmal erfaßt und mit einem gespeicherten Referenzwert verglichen werden kann. Mit fortschreitender Technologie auf dem Gebiet der biometrischen Erkennungssysteme werden diese zunehmend an Bedeutung gewinnen. Bereits heute sind Chipkarten verfügbar, die einen Sensor zur Erfassung von Fingerabdrücken enthalten und somit eine Authentifizierung des Benutzers gegenüber der Karte über den Fingerabdruck statt über die PIN ermöglichen.

Neben der Authentifizierung beim Zugang zu einem System kommt in vernetzten Systemen der wechselseitigen Authentifizierung von Kommunikationspartnern eine große Bedeutung zu. Hierfür werden **Authentifizierungsprotokolle** eingesetzt, die neben der wechselseitigen Authentifizierung zweier Kommunikationspartner in der Regel zusätzlich den sicheren Austausch eines kryptographischen Schlüssels beinhalten, der von den Kommunikationspartnern anschließend zur vertraulichen und authentischen Kommunikation genutzt werden kann. Derartige Protokolle werden deshalb auch Schlüsselaustausch– oder Schlüsselverteilungsprotokolle genannt. Authentifizierungsprotokolle basieren auf dem Einsatz kryptographischer Verfahren. Hierbei können sowohl Private–Key als auch Public–Key Verfahren verwendet werden. Authentifizierungsprotokolle bauen auf einer vertrauenswürdigen, sicheren Basis auf, d.h. es werden vertrauenswürdige Instanzen, die bereits untereinander authentifiziert sind und zwischen denen bereits eine sichere Kommunikationsbeziehung etabliert ist, als gegeben vorausgesetzt. Klassische Protokolle gehen von einem zentralen vertrauenswürdigen Authentifizierungsserver als Basis aus. Bei Verwendung eines Private–Key Verfahrens wird postuliert, daß zwischen jedem Kommunikationsteilnehmer und dem Authentifizierungsserver ein sogenannter Master–Schlüssel (engl.: *master key*) sicher vereinbart ist, der nur diesen beiden bekannt ist. Die wechselseitige Authentifizierung und der sichere Austausch eines Kommunikationsschlüssels zwischen zwei Kommunikationsteilnehmern erfolgt dann unter Nutzung des Authentifizierungsservers und der Master–Schlüssel der Teilnehmer. Beim Einsatz eines Public–Key Verfahrens übernimmt der Authentifizierungsserver die Aufgabe, die öffentlichen Schlüssel aller Kommunikationsteilnehmer zu registrieren und zu verwalten und diese auf entsprechende Anfragen authentisch an andere Teilnehmer zu übermitteln. Hierbei wird in der Regel vorausgesetzt, daß jeder Kommunikationsteilnehmer Kenntnis des öffentlichen Schlüssels des Authentifizierungsservers hat und dieser Kenntnis der öffentlichen Schlüssel aller Teilnehmer hat. Erfolgt die wechselseitige Authentifizierung zweier Kommunikationspartner unter Verwendung eines Public–Key Verfahrens und wird dabei ein geheimer Schlüssel für ein Private–Key Verfahren ausgetauscht, mit dem dann die zwischen den Kommunikationspartnern auszutauschenden Nachrichten verschlüsselt werden, spricht man von einem hybriden Authentifizierungs– und Schlüsselaustauschprotokoll.

Die wohl bekanntesten Authentifizierungsprotokolle sind die 1978 von Needham und Schroeder veröffentlichten und nach ihnen benannten Needham–Schroeder–Protokolle [NS78]. In den ursprünglichen Protokollen bestand das Problem, daß es einem Angreifer, dem es gelungen war, einen Kommunikationsschlüssel zu brechen, möglich war, sich durch Wiederholen von Nachrichten aus dem entsprechenden Schlüsselaustausch als ein anderer Kommunikationsteilnehmer zu maskieren. Dieses Problem wird in den in [DS81] angegebenen Protokollen unter Verwendung von Zeitstempeln (engl.: *timestamps*) gelöst. Needham und Schroeder gaben 1987 eine Verbesserung ihrer ursprünglichen Protokolle an, die ohne Verwendung von Zeitmarken auskommt und allein auf der Nutzung sogenannter Einmal–Identifikatoren (engl.: *nonce identifiers* oder kurz *nonces*) beruht. Es gibt zahlreiche weitere Varianten der Needham–Schroeder–Protokolle. Ein heute weit verbreitetes Authentifizierungssystem ist das Kerberos–System [SNS88, KNT94]. Die Kerberos–Protokolle basieren auf dem Needham–Schroeder–Protokoll für Private–Key Verfahren erweitert um die Zeitstempel aus [DS81]. Kerberos ist zum Beispiel Bestandteil der Sicherheitsdienste der verteilten Ausführungsumgebung DCE (*Distributed Computing Environment*) der OSF (*Open Software Foundation*) und wird auch in Microsoft Windows 2000, dem Nachfolger von Windows NT 4.0, für die Authentifizierung eingesetzt.

In den letzten zehn Jahren wurden verstärkt Ansätze zur formalen Spezifikation und Verifikation von Authentifizierungsprotokollen im Hinblick auf ihre Sicherheitseigenschaften entwickelt. Der bekannteste dieser Ansätze ist die von Burrows, Abadi und Needham entwickelte

BAN-Logik [BAN90, LABW92]. Ein guter Überblick über die verschiedenen Ansätze ist in [Gei95] zu finden.

2.1.3.3 Rechteverwaltung und –prüfung

Zur Durchsetzung von Zugriffskontrollpolitiken sind bei Zugriffen von Subjekten auf Objekte Zugriffskontrollen durchzuführen, die gewährleisten, daß nur autorisierte Zugriffe durchgeführt werden. Hierfür werden Realisierungskonzepte benötigt, mit denen die durch die vergebenen Rechte festgelegten Zugriffsbeschränkungen implementiert werden können. Traditionell werden im Bereich der Zugriffskontrolle zwei Klassen von Realisierungskonzepten unterschieden: die Zugriffskontrollisten und die Capability–Listen.

Mit dem Konzept der **Zugriffskontrollisten** (engl.: *access control list* (ACL)) wird jedem zu schützenden Objekt eine Liste zugeordnet, deren Einträge die aktuellen Zugriffsrechte für einzelne Subjekte oder Gruppen von Subjekten beschreiben. Bei einem Zugriff eines Subjekts auf ein Objekt wird anhand der Zugriffskontrolliste des Objekts überprüft, ob für das Subjekt das entsprechende Zugriffsrecht vorhanden ist. Die Zugriffskontrollisten sind vor unautorisierter Manipulation zu schützen. Sie werden deshalb in der Regel als Bestandteil der vom Betriebssystemkern angelegten Objektbeschreibung eines Objekts, dem sogenannten Objekt–Deskriptor, verwaltet. Damit ist der direkte Zugriff auf diese Datenstrukturen für Benutzerprozesse nicht möglich. Die Vorteile des Zugriffkontrollisten–Konzepts liegen in der einfachen Verwaltung der Zugriffsrechte. Für ein spezifisches Objekt ist anhand der Zugriffskontrolliste leicht zu bestimmen, welche Subjekte welche Zugriffsrechte an dem Objekt besitzen. Die Vergabe bzw. Rücknahme von Rechten an einem Objekt ist einfach zu realisieren, da nur die entsprechenden Einträge in der Zugriffskontrolliste des Objekts aktualisiert werden müssen. Demgegenüber steht der Nachteil, daß auf Basis von Zugriffskontrollisten die Menge der Zugriffsrechte, die ein einzelnes Subjekt aktuell an den Objekten des Systems besitzt, nur mit großem Aufwand bestimmt werden kann, da hierzu die Zugriffskontrollisten aller Objekte zu durchsuchen sind. Ein weiterer Nachteil ergibt sich daraus, daß in großen und komplexen Systemen mit vielen Subjekten die Zugriffskontrollisten sehr lang werden können. Da bei jedem Zugriff eines Subjekts auf ein Objekt die entsprechende Zugriffskontrolliste zu überprüfen ist, kann dies zu einer aufwendigen und ineffizienten Zugriffskontrolle führen. In diesem Zusammenhang ist auch die Semantik der Auswertung von Zugriffskontrollisten von Interesse. So kann zum Beispiel zwischen positiven und negativen Einträgen unterschieden werden. Mit einem positiven Eintrag wird ein Recht explizit an ein Subjekt vergeben während mit einem negativen Eintrag ein bestimmter Zugriff explizit verboten werden kann. Hieraus können sich insbesondere in Zusammenhang mit der Vergabe von Rechten an Gruppen von Subjekten Konflikte ergeben, die durch eine eindeutige Semantik der Auswertung derartiger Zugriffskontrollisten aufzulösen sind. Ein Konflikt liegt zum Beispiel dann vor, wenn in der Zugriffskontrolliste eines Objekts für ein Subjekt, das Mitglied einer Gruppe ist, ein positiver Eintrag bzgl. eines Zugriffs auf das Objekt und für die Gruppe ein negativer Eintrag für den entsprechenden Zugriff auf das Objekt enthalten ist. Derartige Konflikte können zum Beispiel dadurch aufgelöst werden, daß ein negativer Eintrag und damit ein Verbot stets Vorrang vor einem positiven Eintrag hat oder daß die Zugriffskontrolliste in der Reihenfolge ihrer Einträge ausgewertet wird und somit der erste für ein Subjekt „passende“ Eintrag gültig ist. Auf die Problematik der Vergabe positiver und negativer Rechte wird in Kapitel 5 in Zusammenhang mit der Konsistenz der in einem System vergebenen Rechte noch genauer eingegangen. Zugriffskontrollisten werden heute in fast allen kommerziellen Betriebssystemen, wie UNIX, Windows NT/2000, IBM OS390 oder BS2000/OSD als Basis für die Vergabe und Kontrolle von Zugriffsrechten eingesetzt.

Mit den **Capability–Listen** (engl.: *capability list*) wird im Gegensatz zu der objekt–bezogenen Sicht der Zugriffskontrollisten jedem Subjekt eine Liste zugeordnet, deren Einträge – die Capabilities – festlegen, welche Rechte das Subjekt an welchen Objekten des Systems besitzt. Eine Capability ist eine Art Ticket bzw. Ausweis, dessen Besitz zur Ausführung der in der Capability enthaltenen Operationen auf dem durch die Capability identifizierten Objekt berechtigt. Eine Capability muß unverfälschbar sein und darf nur von vertrauenswürdigen Instanzen, den sogenannten Capability–Managern erzeugt und manipuliert werden. In der Regel werden Capabilities durch den Betriebssytemkern verwaltet und damit vor unautorisierter Manipulation geschützt. Werden Capabilities auf Benutzerebene verwaltet, so muß durch zusätzliche Maßnahmen, wie beispielsweise die Verschlüsselung der Capabilities, deren Unverfälschbarkeit garantiert werden. Ein wesentlicher Vorteil des Capability–Konzepts gegenüber den Zugriffskontrollisten besteht in der einfachen Durchführung der Zugriffskontrollen basierend auf der Kontrolle der Capability, die von einem Subjekt bei einem Zugriffsversuch vorzuweisen ist. Das Subjekt darf genau dann auf das Objekt zugreifen, wenn es in dem Besitz einer gültigen Capability ist, die das entsprechende Zugriffsrecht enthält. In einem Capability–basierten System läßt sich einfach bestimmen, welche Rechte ein Subjekt an den Objekten des Systems aktuell besitzt. Demgegenüber ist es für ein spezifisches Objekt aufwendig zu ermitteln, welche Subjekte welche Zugriffsrechte an dem Objekt haben. Problematisch bei dem Einsatz von Capabilities ist die Rücknahme von Rechten. Dies gilt insbesondere dann, wenn Kopien von Capabilities angelegt werden können. Ein Ansatz zur Lösung dieses Problems besteht darin, Capabilities mit einem festen Gültigsintervall zu versehen, nach dessen Ablauf sie ungültig werden. Capabilities lassen sich durch geeignete hardwaremäßige Unterstützung effizient realisieren. In entsprechenden Systemen wird der Capability–Zugriff auf der Hardware–Ebene in den Speicheradressierungsmechanismus integriert. Eine derartige Capability–basierte Adressierung (siehe u.a. [Fab74, Lin76]) wurde bereits 1966 von Dennis und van Horn [DH66] eingeführt. Einen guten Überblick zu Capability–basierten Systemen liefert das Buch von Levy [Lev84]. Neben diesen klassischen Systemen werden Capabilities zunehmend in Betriebssystemkernen verteilter und objekt–orientierter Systeme als Basisschutzmechanismus eingesetzt, wie zum Beispiel in dem Mikrokern Amoeba [MT86, MvRT$^+$90] oder in dem Mach3.0–Mikrokern [ABG$^+$86, Loe91].

Es gibt verschiedene Ansätze, in denen versucht wird, die Vorteile der beiden vorgestellten Zugriffskontrollkonzepte miteinander zu kombinieren (u.a. [Gif82, RSC92]). Ein Beispiel hierfür ist der in [Gif82] vorgestellte Lock–Key Mechanismus. Hierbei wird zu jedem Objekt eine Art Zugriffskontrolliste gespeichert, die Paare bestehend aus einem Schloß und einer Menge von Zugriffsrechten enthält. Jedem Subjekt wird eine Art Capability–Liste zugeordnet, deren Einträge aus einem Objektidentifikator und einem Schlüssel bestehen. Ein Zugriff eines Subjekts auf ein Objekt wird erlaubt, wenn das Subjekt einen Schlüssel für das Objekt besitzt, der in eines der Schlösser des Objekts „paßt“. Eine Rechterücknahme ist leicht dadurch zu erreichen, daß in der Zugriffskontrolliste des betroffenen Objekts ein Schloß verändert wird, so daß die vergebenen Schlüssel nicht mehr passen. Der Lock–Key Mechanismus wird unter Einsatz kryptographischer Verfahren realisiert. In dieser Arbeit wird mit dem in Kapitel 6 erklärten Ticket–Konzept ein neuer Ansatz vorgestellt, der die Vorteile von Zugriffskontrollisten und Capabilities vereint.

Im Kontext klassischer Betriebssysteme dient häufig das Konzept des Referenzmonitors als Denkmodell für die Durchführung von Zugriffskontrollen. Der globale Referenzmonitor hat die Aufgabe, alle Zugriffe auf Objekte zu überprüfen und die erforderlichen Zugriffskontrollen durchzuführen. Der Referenzmonitor ist Bestandteil der **Trusted Computing Base** (TCB), die alle Sicherheitsmechanismen und –funktionen umfaßt, die zur Durchsetzung von Sicherheitspolitiken und –anforderungen benötigt werden. Werden die zur Durchsetzung von

Sicherheitspolitiken eingesetzten Maßnahmen in einem wohldefinierten, isolierten Teil der Systemarchitektur zusammengefaßt, so wird dieser Teil **Sicherheitskern** (engl.: *security kernel*) genannt. Der Sicherheitskern veredelt die Hardware und stellt Anwendungsprogrammen über entsprechende Kernel–Dienste eine Sicherheitsschnittstelle zur Verfügung. Er überwacht alle Zugriffe auf die Hardware und führt die sicherheitsrelevanten Operationen, wie Zugriffskontrolle und Authentifikation durch. Bei der Realisierung von Sicherheitskernen wird auf Mechanismen zurückgegriffen, die von der Hardware zur Verfügung gestellt werden. Dazu gehören Speicherschutzmechanismen oder unterschiedliche Befehlsmodi, wie zum Beispiel privilegierte und nicht privilegierte Befehle. Sicherheitskerne sollten möglichst klein sein, um eine formale Verifikation ihrer Eigenschaften zu ermöglichen. Beispiele für Systeme mit einem Sicherheitskern sind das UCLA Secure Unix System [PKK+79] und der VAX Sicherheitskern VMM [KZB+90].

Mit den in diesem Abschnitt vorgestellten Sicherheitsmechanismen und –verfahren steht ein breites Spektrum zur Realisierung anwendungsspezifischer Sicherheitsanforderungen zur Verfügung. Eine wesentliche Anforderung für die Wirksamkeit der Sicherheitsmaßnahmen besteht darin, daß diese integraler Bestandteil der Systemarchitektur sein müssen. Dies erfordert, daß die Sicherheitsaspekte bei der Konstruktion eines Rechensystems von Beginn an integriert in den gesamten Konstruktionsprozeß zu betrachten sind. Die Konstruktion sicherer Rechensysteme ist eine komplexe Aufgabe, die eine systematische Vorgehensweise erfordert. Hierauf wird in dem nun folgenden Abschnitt näher eingegangen.

2.2 Konstruktion sicherer Rechensysteme

Bevor im weiteren die einzelnen Phasen des Konstruktionsprozesses sicherer Rechensysteme erläutert werden, erfolgt zunächst die Angabe von allgemeinen Prinzipien, die bei der Konstruktion eines sicheren Rechensystems zu beachten sind.

2.2.1 Allgemeine Konstruktionsprinzipien

Allgemeine Konstruktionsprinzipien für sichere Rechensysteme wurden bereits 1975 von Saltzer und Schroeder in [SS75] formuliert. Diese Prinzipien haben nichts an Aktualität verloren und besitzen auch heutzutage noch Gültigkeit. Zu diesen Prinzipien gehören das Erlaubnisprinzip, das Need–to–know Prinzip, das Vollständigkeitsprinzip, das Prinzip der Benutzerakzeptanz und das Prinzip des offenen Entwurfs.

Das **Erlaubnisprinzip** (engl.: *fail–safe defaults*) besagt, daß grundsätzlich jeder Zugriff verboten ist und nur durch eine explizite Erlaubnis ein Zugriffrecht gewährt werden kann.

Durch das **Need–to–know Prinzip**, das auch Prinzip der minimalen Rechte (engl.: *least privilege*) genannt wird ([Lin76]), wird gefordert, daß jedes Subjekt nur genau die Zugriffsrechte erhalten darf, die es zur Erfüllung seiner Aufgaben unbedingt benötigt. Dieses Prinzip wird zum Beispiel in Systemen mit einem Super–User, der unbeschränkte Rechte besitzt (z.B. *root* in UNIX oder der Domänenadministrator unter Windows NT 4.0), verletzt.

Das **Vollständigkeitsprinzip** (engl.: *complete mediation*) fordert, daß jeder Zugriff auf das System zu autorisieren ist, d.h. jeder Zugriff muß im Rahmen der Zugriffskontrolle auf Zulässigkeit überprüft werden. Ein System, in dem zum Beispiel Zugriffsrechte an Dateien vergeben werden und in denen lediglich beim Öffnen einer Datei überprüft wird, ob das entsprechende Zugriffsrecht vorliegt, verstößt gegen dieses Prinzip, da alle nachfolgenden Lese–

bzw. Schreibzugriffe auf die Datei nicht kontrolliert werden. Ein Benutzer kann somit eine Datei lange Zeit geöffnet halten und damit das Zugriffsrecht nutzen, obwohl ihm beispielsweise durch den Besitzer der Datei dieses Recht zwischenzeitlich entzogen worden sein kann.

Das **Prinzip der Benutzerakzeptanz** (engl.: *economy of mechanism*) besagt, daß die eingesetzten Sicherheitsmechanismen und –verfahren einfach zu nutzen sein müssen und automatisch und routinemäßig angewendet werden.

Durch das **Prinzip des offenen Entwurfs** (engl.: *open design*) wird schließlich gefordert, daß die Verfahren und Mechanismen, die beim Entwurf eines Systems angewendet werden, offen gelegt werden müssen, da die Sicherheit eines Systems nicht von der Geheimhaltung spezieller Verfahren, wie zum Beispiel, welche kryptographischen Verfahren eingesetzt werden, abhängig sein darf.

2.2.2 Systematischer Konstruktionsprozeß

Die Konstruktion sicherer Rechensysteme erfordert in hohem Maße eine systematische und methodische Vorgehensweise einschließlich der Verwendung formaler Beschreibungs– und Modellierungstechniken. Analog zum Prozeß des Software–Engineering sind bei der Konstruktion eines sicheren Systems verschiedene Phasen – unter Umständen iterativ – zu durchlaufen (siehe z.B. [Bas93]). Für einen derartigen systematischen Konstruktionsprozeß für sichere Systeme hat sich zwischenzeitlich der Begriff des Security Engineering etabliert. Die wesentlichen Phasen dieses Konstruktionsprozesses sind in Abbildung 2.1 dargestellt.

Für ein zu konstruierendes sicheres System sind zunächst im Rahmen einer **Bedrohungs– und Risikoanalyse** die tatsächlichen Bedrohungen, denen das System ausgesetzt ist, und deren Relevanz für das System zu bestimmen. In der Bedrohungsanalyse sind unter Berücksichtigung der Einsatzumgebung des Systems die potentiellen Angriffsmöglichkeiten und die daraus resultierenden Bedrohungen systematisch zu ermitteln. Verfahren der Bedrohungs– und Risikoanalyse sind zum Beispiel in [BSI92] angegeben.

Ausgehend von der Bedrohungs– und Risikoanalyse sind die **Sicherheitsanforderungen**, die an das System gestellt werden, festzulegen. Die Sicherheitsanforderungen werden in der Regel zunächst verbal, d.h. in Worten formuliert. Anschließend sind sie, soweit dies möglich ist, formal zu spezifizieren. Hierfür werden Spezifikationskonzepte benötigt, die es ermöglichen, die Sicherheitsanforderungen eines Systems auf einem hohen Abstraktionsniveau unabhängig von bestimmten Realisierungsmechanismen zu erfassen. Ein Überblick über die in der Literatur vorhandenen Ansätze zur formalen Spezifikation von Sicherheitseigenschaften und –anforderungen ist in [Eck98] zu finden.

Sind für das zu konstruierende System die zu realisierenden Sicherheitsanforderungen informell oder formal festgelegt, so besteht der nächste Schritt darin, ein **Sicherheitsmodell** für das System zu entwickeln. Ein Sicherheitsmodell ist nach [AK93] eine abstrakte Beschreibung der nach den zugrundeliegenden Sicherheitsanforderungen für wesentlich gehaltenen Aspekte der Sicherheit eines Rechensystems, wobei die als nicht sicherheitsrelevant geltenden Aspekte unterdrückt werden. Der Begriff des Modells wird hier somit im Sinne einer Abstraktion verwendet. Ein Sicherheitsmodell beschreibt die Sicherheitseigenschaften eines Systems – in der Regel formal – auf einem hohem Abstraktionsniveau. In der Literatur existieren zahlreiche unterschiedliche Sicherheitsmodelle, auf die hier im Detail nicht näher eingegangen wird. Zu den bekanntesten zählen neben dem Zugriffsmatrix–Modell [Lam71, GD72] und dem Bell–LaPadula–Modell [BL75], das Non–Interference–Modell von Goguen und Meseguer [GM82], das Clark–Wilson–Modell [CW87], das Chinese–Wall–Modell [BN89] sowie

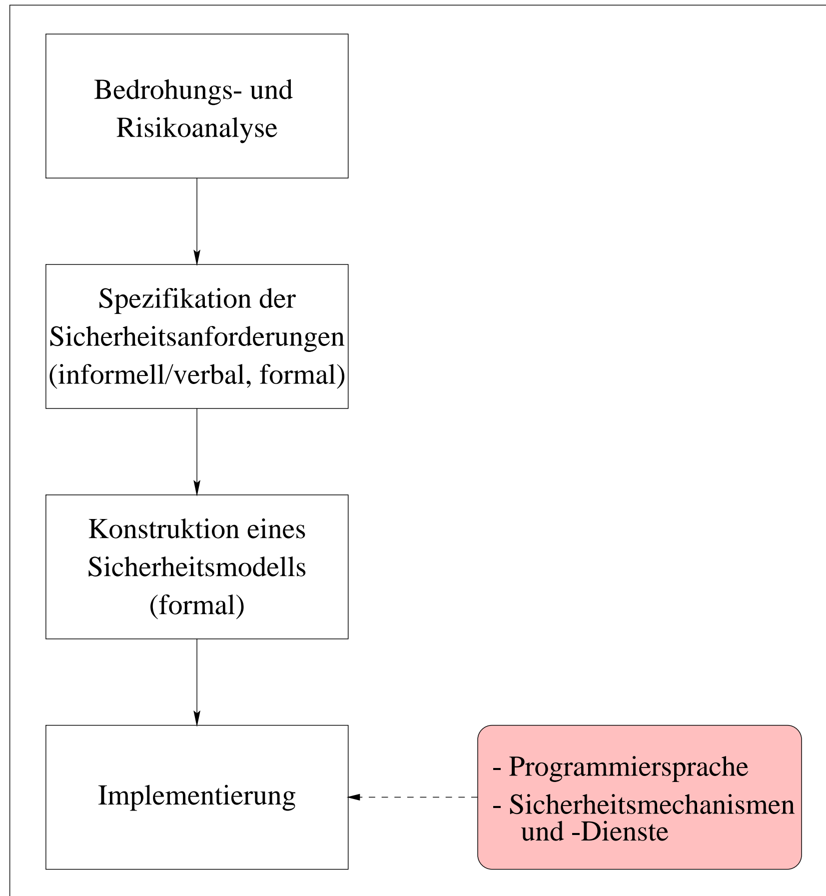

Abbildung 2.1: Systematischer Konstruktionsprozeß für sichere Systeme

das Terry–Wiseman–Modell [TW89]. Ausführliche Beschreibungen dieser Modelle sind zum Beispiel in [Eck93] zu finden. Die bekannten Sicherheitsmodelle sind – abhängig von ihrem jeweiligen Entstehungskontext – auf bestimmte Klassen von Sicherheitsanforderungen, wie zum Beispiel Integrität oder Vertraulichkeit, ausgerichtet. Das Bell–LaPadula–Modell konzentriert sich beispielsweise auf die Modellierung von Vertraulichkeitsanforderungen, wie sie typischerweise in hierarchischen Organisationsstrukturen, zu denen insbesondere der militärische Bereich gehört, vorliegen. Die Beschränkung auf spezielle Sicherheitsaspekte führt dazu, daß die Sicherheitsanforderungen eines Systems mit einem der vorhandenen Modelle unter Umständen nur unvollständig erfaßt werden. So spielt beispielsweise in militärischen Systemen neben der Vertraulichkeit auch die Integrität der Informationen eine wichtige Rolle. In der Regel ist deshalb bei der Systemkonstruktion ein individuelles anwendungsspezifisches Sicherheitsmodell zu erstellen. In [Eck93] wurde hierfür ein flexibel einsetzbares Modellierungsinstrumentarium entwickelt, mit dem Anforderungen sowohl an die Integrität als auch an die Vertraulichkeit anwendungsspezifisch modellierbar sind. Sicherheitsmodelle haben insbesondere in Zusammenhang mit der Bewertung der Sicherheit von Rechensystemen nach entsprechenden nationalen bzw. internationalen Kriterienkatalogen (z.B. die US–amerikanischen „Trusted Computing System Evaluation Criteria“ (TCSEC, Orange Book) [Def85], die harmonisierten europäischen „Information Technology Security Evaluation Criteria (ITSEC)“ [Com91] oder die neuen internationalen „Common Criteria for Information Technology Security Evaluation“ [CC98]) eine große Bedeutung. Für das Erreichen einer hohen Sicherheitsstufe verlangen diese Kriterienkataloge in der Regel ein formales Sicherheitsmodell.

Das modellierte System ist im nächsten Konstruktionsschritt unter Nutzung einer Programmiersprache und geeigneten Sicherheitsmechanismen und –diensten, die insbesondere durch das zugrundeliegende Betriebssystem bereitgestellt werden, zu **implementieren**. Der programmiersprachlichen Ebene kommt hierbei eine zentrale Rolle zur Überbrückung der Lücke zu, die im allgemeinen zwischen einer formalen Spezifikation bzw. einem formalen Modell und der Implementierung besteht. Wie wichtig diese Brückenfunktion ist, kommt u.a. in folgendem Auszug aus [Gas88] zum Ausdruck:

> *The state of the art in verification today does not permit us to eliminate the informal argument between the implementation level and the lowest level of formal specification, and this informal step remains the weakest link in the overall model–to–implementation correspondence argument.*

Die für die Implementierung eines sicheren Systems eingesetzte Programmiersprache kann somit einen wesentlichen Beitrag zur Überbrückung der Lücke und damit zur Realisierung der spezifizierten Sicherheitsanforderungen leisten. Trotz der großen Bedeutung, die Programmiersprachen für die Konstruktion sicherer Rechensysteme haben, bieten die herkömmlichen für die Implementierung sicherer Systeme verwendeten Sprachen keine speziellen oder lediglich unzureichende Konzepte auf niedrigem Abstraktionsniveau zur Unterstützung der Realisierung von Sicherheitsanforderungen an. Dadurch besteht für den Systementwickler die Notwendigkeit, die Durchsetzung der Sicherheitsanforderungen unter Nutzung vorhandener *low–level* Sicherheitsmechanismen und –dienste des zugrundeliegenden Betriebssystems oder der eingesetzten Programmier– bzw. Ausführungsumgebung explizit zu programmieren. Ein Beispiel hierfür ist die Implementierung von Zugriffsrechten für Dateien auf Basis einfacher Zugriffskontrollisten unter Nutzung entsprechender Systemaufrufe und Bibliotheksroutinen, wie sie in dem UNIX–Betriebssystem HP–UX zur Verfügung stehen [Hew91]. Ein weiteres Beispiel liefert das OSF *Distributed Computing Environment* (DCE), das dem Systementwickler einfache Bibliotheksroutinen zur expliziten Implementierung von Zugriffskontrollisten (ACL) und assoziierter Zugriffskontrollisten–Manager (ACL–Manager) bereitstellt [Hu95]. Die Implementierung von Sicherheitseigenschaften unter Nutzung derartiger *low–level* Mechanismen ist ein im allgemeinen mühsamer und fehlerträchtiger Vorgang, so daß die Gefahr besteht, daß aufgrund von Implementierungsfehlern Sicherheitslücken entstehen. Sicherheitsprobleme, die als Folge von Implementierungsfehlern auftreten, sind vielfältig und in der Literatur zahlreich bekannt (vgl. u.a. [LBMC94]). Beispiele für derartige Implementierungsfehler sind unzulängliche Bereichsprüfungen bei der Nutzung von Puffer–Datenstrukturen. Ein solcher Implementierungsfehler in dem UNIX–Kommando `finger` wurde beispielsweise von dem bekannten Internet–Wurm aus dem Jahr 1988 ausgenutzt [Spa89]. Der Internet–Wurm brachte in dem `finger`–Angriff den Eingabepuffer des Kommandos zum Überlauf und plazierte darüber auf dem Systemkeller ausführbaren Code, der dann im privilegierten Systemmodus ausgeführt wurde [Spa88]. Einige der in Zusammenhang mit der unter besonderer Beachtung von Sicherheitsaspekten konzipierten Programmiersprache Java [Sun95] aufgetretenen Sicherheitsprobleme sind ebenfalls auf Implementierungsfehler zurückzuführen, die in den gängigen Web–Browsern (Netscape Communicator/Navigator, Microsoft Internet Explorer) auftraten [DFW96].

In der Literatur gibt es nur wenige Arbeiten, die sich mit der Erweiterung von Programmiersprachen um spezielle Konzepte zur Unterstützung der Implementierung sicherer Systeme beschäftigen. Auf die vorhandenen Ansätze wird in dem folgenden Abschnitt noch näher eingegangen. Die bekannten Ansätze konzentrieren sich jedoch darauf, *low–level* Konzepte, wie zum Beispiel Capabilities, in die Programmiersprache zu integrieren. Der Systementwickler

muß also nach wie vor die Durchführung von Zugriffskontrollen sowie die Verwaltung der hierfür erforderlichen Datenstrukturen zum Beispiel durch die Implementierung entsprechender Manager explizit programmieren. Derartige Ansätze leisten somit nur einen sehr geringen Beitrag zum Schließen der Lücke zwischen der Spezifikation von Sicherheitsanforderungen und deren Realisierung. Zur Überbrückung dieser Lücke werden stattdessen programmierspachliche Konzepte benötigt, die zum einen an das für die Spezifikation und Modellbildung eingesetzte Modellierungsinstrumentarium angepaßt sind und es zum anderen ermöglichen, die Sicherheitseigenschaften auf hohem Abstraktionsniveau weitestgehend deklarativ zu implementieren. In dieser Arbeit werden entsprechende Sprachkonzepte für die Implementierung anwendungsspezifisch festgelegter Zugriffskontrollpolitiken entwickelt. Diese Sprachkonzepte sind so gewählt, daß durch sie insbesondere die systematische Implementierung von Zugriffsbeschränkungen, die mit dem in [Eck93] entwickelten Spezifikations– und Modellierungsinstrumentarium formal festgelegt sind, unterstützt wird. Bei kombiniertem Einsatz des formalen Rahmenwerks aus [Eck93] und der in dieser Arbeit beschriebenen Sprachkonzepte kann die aufgezeigte Lücke zwischen der formalen Spezifikation von Sicherheitsanforderungen und deren Implementierung deutlich verkleinert werden [EM97].

Die mit den Sprachkonzepten deklarativ festgelegten Zugriffsbeschränkungen werden automatisiert durch den eingesetzten Übersetzer unter Rückgriff auf Sicherheitsmechanismen und –dienste der zugrundeliegenden Ausführungsbasis realisiert. Der Systementwickler wird damit von realisierungstechnischen und an *low–level* Mechanismen orientierten Maßnahmen zur Implementierung der Sicherheitsanforderungen entlastet. So muß er zum Beispiel im Gegensatz zu herkömmlichen Ansätzen die Durchführung von Zugriffskontrollen und die Verwaltung der Datenstrukturen, auf deren Basis diese Kontrollen durchgeführt werden (wie z.B. Zugriffskontrollisten oder Capabilities), nicht explizit programmieren, sondern lediglich deklarativ angeben, welche Bedingungen bei den entsprechenden Kontrollen zu überprüfen sind. Implementierungsfehler, die zu einer nicht korrekten Realisierung der formulierten Zugriffsbeschränkungen führen, werden somit weitestgehend ausgeschlossen.

Für die Realisierung anwendungsspezifischer Sicherheitsanforderungen muß die zugrundeliegende Ausführungsbasis ein breites Spektrum von Sicherheitsmechanismen und -diensten bereitstellen. So werden neben Basismechanismen für die Durchführung von Zugriffskontrollen kryptographische Verfahren und darauf aufbauende Dienste wie zum Beispiel Authentifikations– und Verschlüsselungsdienste benötigt. Bei dem in dieser Arbeit verfolgten sprachbasierten *top–down* Ansatz erfolgt die Auswahl der für die Realisierung der jeweiligen mit den Sprachkonzepten festgelegten Sicherheitsanforderungen einzusetzenden Mechanismen und Dienste automatisiert durch die entwickelten Transformationswerkzeuge. Der Systementwickler kann sich bei diesem Konstruktionsansatz somit auf die Festlegung der anwendungsspezifischen Sicherheitsanforderungen auf einem hohen Abstraktionsniveau konzentrieren und wird von dem fehleranfälligen Einsatz realisierungsnaher Konzepte befreit.

2.3 Programmiersprachliche Konzepte zur Konstruktion sicherer Rechensysteme

Wie in dem vorhergehenden Abschnitt erläutert, kommt der programmiersprachlichen Ebene bei der systematischen *top–down* orientierten Konstruktion sicherer Rechensysteme eine zentrale Bedeutung zu. In diesem Abschnitt werden zunächst allgemeine programmiersprachliche Konzepte und Paradigmen vorgestellt, die vorrangig die Programmiersicherheit betreffen und die somit die Implementierung sicherer Systeme unterstützen. Im weiteren wird dann auf in

der Literatur vorhandene Ansätze eingegangen, die spezielle Sprachkonzepte zur Realisierung von Sicherheitsanforderungen zur Verfügung stellen.

2.3.1 Allgemeine Konzepte und Paradigmen

Zu den wichtigsten programmiersprachlichen Konzepten und Paradigmen, die die Konstruktion sicherer Systeme unterstützen, gehören die Modularisierung einschließlich des Konzepts der abstrakten Datentypen, Konzepte zur Festlegung von Sichtbarkeitsbereichen, das Konzept der Typisierung, Konzepte zur Ausnahmebehandlung sowie die Verhinderung von direkten Speicherzugriffen und Speicherverwaltungsoperationen durch den Programmierer.

Das Konzept der **abstrakten Datentypen** ([LZ75]), und damit eng zusammenhängend die **Modularisierung** ermöglichen die Kapselung (engl.: *encapsulation* oder auch *information hiding*) von Daten innerhalb von Objekten bzw. Komponenten, so daß auf diese Daten nur über geschützte Eintrittspunkte bzw. wohldefinierte Operationen (Methoden) zugegriffen werden kann. Die Modularisierung und die Konstruktion von abstrakten Datentypen bzw. Klassen sind grundlegende Konzepte der meisten heutzutage eingesetzten höheren Programmiersprachen, insbesondere der objekt–orientierten Sprachen, wie z.B. C++ [Str91] und Java [Sun95]. Diese Konzepte bilden die Basis für die Implementierung sicherer Systeme, da durch ihren Einsatz die Konstruktion von Objekten in dem in Abschnitt 2.1.1 angegebenen Sinn möglich ist.

Mit Konzepten zur **Festlegung von Sichtbarkeitsbereichen** (engl.: *scope rules*) bzw. Ausführungumgebungen lassen sich Zugriffe auf Objekte konzeptionell beschränken. Sie dienen damit als Basis für die Konstruktion von Systemen gemäß des Need–to–know Prinzips. Sichtbarkeitsbeschränkungen können zum einen implizit durch **Schachtelung** (engl.: *nesting*) von Komponenten in blockorientierten Programmiersprachen (z.B. Modula-2, Ada) und zum anderen explizit durch spezielle Sprachkonzepte festgelegt werden. Zu derartigen Sprachkonzepten gehören zum Beispiel Import– und Exportfestlegungen sowie die Kennzeichnung von Objekten bzw. Operationen als private oder öffentlich nutzbare Objekte. Die Programmiersprache Java bietet beispielsweise vier Notationen zur expliziten Festlegung der Sichtbarkeit von Methoden und Variablen. Es wird zwischen öffentlich zugreifbaren Methoden und Variablen (*public*), privaten Methoden und Variablen (*private*), die nur von Methoden der gleichen Klasse verwendet werden dürfen, privat–geschützten Komponenten (*private protected*), die lediglich von der gleichen Klasse und deren Unterklassen genutzt werden dürfen, und geschützten Komponenten (*protected*), die zusätzlich noch innerhalb des gleichen Packages nutzbar sind, unterschieden. Ähnliche Sichtbarkeitskategorien stehen in der Sprache Trellis/Owl [SCW85] zur Verfügung. Mit Konzepten für Importfestlegungen läßt sich die Menge der von einem Objekt von außen benötigten und damit innerhalb des Objekts sichtbaren Komponenten explizit angeben und damit auf das Notwendigste beschränken. Durch Exportfestlegungen kann die Menge der Komponenten eines Objekts, die anderen Objekten potentiell zur Nutzung zur Verfügung stehen sollen und die damit außerhalb des Objekts sichtbar sind, beschränkt werden. Ein Beispiel hierfür ist das in [MA79] beschriebene *grant*–Konstrukt. In Abschnitt 4.3 dieser Arbeit werden spezifische Konzepte angegeben, mit denen sich Sichtbarkeitsbereiche sowie die Ausführungsumgebungen von Objekten u.a. auf Basis von Import– und Exportfestlegungen systematisch und differenziert festlegen lassen. Je differenzierter sich Sichtbarkeitsbereiche mit den Konzepten einer Programmiersprache festlegen lassen, desto mehr kann das Potential für unautorisierte Zugriffe durch systematischen Einsatz dieser Konzepte von vornherein eingeschränkt werden. Die Einhaltung der konzeptionell festgelegten Sichtbarkeitsbeschränkungen ist durch den Übersetzer zu überprüfen. Damit lassen

sich bereits zur Übersetzungszeit potentielle Zugriffsverletzungen, die sich aus der Nichteinhaltung von Sichtbarkeitsregeln ergeben, ermitteln.

Einen wesentlichen Beitrag für die Implementierung sicherer Systeme leistet die **Typisierung**. Eine streng typisierte (engl.: *strongly typed* oder *type safe*) Programmiersprache ermöglicht es, Zugriffsverletzungen aufgrund falscher typverletzender Wertzuweisungen, die potentielle Bedrohungen der Sicherheit sein können, statisch zur Übersetzungszeit des Programms zu erkennen und auszuschließen. Durch ein strenges Typkonzept kann sichergestellt werden, daß eine Typkonvertierung (engl.: *type casting*) zwischen Objekten unterschiedlichen Typs nicht möglich ist. Dadurch werden Sicherheitsbedrohungen, die sich aus dem unbeschränkten Verwenden von Zeigern in untypisierten bzw. schwach typisierten Programmiersprachen wie zum Beispiel C ergeben, konzeptionell ausgeschlossen. In derartigen Sprachen ist es zum Beispiel möglich, einen Integer–Wert als eine Adresse zu interpretieren und auf diese Weise direkten Zugriff auf eine Speicheradresse zu erlangen. Eine strenge Typisierung ermöglicht es hingegen, bereits statisch durch den Übersetzer Kontrollen auf unzulässige Speicherzugriffe durchzuführen.

Die strenge Typisierung einer Programmiersprache liefert somit einen wichtigen Beitrag zur Reduzierung von Sicherheitsproblemen, die durch direkte Speicherzugriffe verursacht werden können. Neben einem strengen Typkonzept tragen hierzu auch Konzepte bei, die den Einsatz von Zeigern beschränken, so daß zum Beispiel wie in Java keine Zeigerarithmetik möglich ist. Daneben sollte eine für die Implementierung sicherer Systeme genutzte Programmiersprache keine Konzepte zum expliziten Belegen und Freigeben von Speicherbereichen enthalten. Durch die Möglichkeit der expliziten Verwaltung des Speichers kann zum Beispiel ein Speicherbereich zur Realisierung eines Objekts angelegt werden, ein Zeiger auf diesen Bereich erzeugt und der Speicherbereich anschließend wieder freigegeben werden, um dann sofort wieder für ein anderes Objekt allokiert zu werden, wodurch der Zeiger nunmehr auf dieses Objekt verweist. Dadurch kann eine unerlaubte Typkonvertierung erreicht werden und ein Zugriff durchgeführt werden, der gemäß der Typisierungsregeln und möglicherweise der festgelegten Sicherheitsanforderungen unzulässig ist.

Neben den genannten Aspekten ist die Typisierung noch unter einem anderen Gesichtspunkt von Interesse. Mit einer Typisierung bzw. der Klassenbildung in objekt–orientierten Sprachen lassen sich generische Einheiten festlegen, die für alle Instanzen eines Typs bzw. einer Klasse gelten. Damit können zum einen die Sicherheitseigenschaften wie zum Beispiel Zugriffsbeschränkungen, die allen Instanzen bzw. Objekten gemeinsam sind, bereits mit der Typ– bzw. Klassendefinition festgelegt werden. Zum anderen können Nutzungsrechte an Klassen statt an einzelne Objekte vergeben werden, wodurch das Rechtemanagement wesentlich vereinfacht wird. Das Konzept der Vererbung, wie es in objekt–orientierten Sprachen zur Verfügung steht, eröffnet hier noch weitergehende Möglichkeiten, die bisher jedoch nur ansatzweise untersucht worden sind (siehe z.B. [DABW96]).

Generell sei an dieser Stelle angemerkt, daß objekt–basierte und objekt–orientierte Sprachen (siehe z.B. [Weg87, Mey88]) sowie ihr zunehmender Einsatz, insbesondere von Java, zur Entwicklung „mobiler“ Internet– und Web–Anwendungen erhebliche Fortschritte im Bereich der allgemeinen Sprachkonzepte zur Implementierung sicherer Systeme und Anwendungen gebracht haben.

Zur Erkennung und Behandlung von Fehlersituationen, die zum Beispiel durch dynamische Zugriffsverletzungen verursacht werden, ist ein in die jeweilige Sprache integriertes Konzept für die **Ausnahmebehandlung** (engl.: *exception handling*) erforderlich. Mit derartigen Konzepten bietet sich insbesondere die Möglichkeit, dynamisch inkonsistente Rechtszustände zu

erkennen und individuell zu behandeln, indem beispielsweise im Rahmen der Ausnahmebehandlung Rechte nachgefordert werden, die aktuell für den gewünschten Zugriff benötigt werden. Geeignete Ausnahmebehandlungskonzepte sind zum Beispiel in den Sprachen Ada ([Ada83, Fel96]), Guide ([BLR94, Lac91]) und Java enthalten.

Die bisher unter dem Aspekt der Konstruktion sicherer Systeme vorgestellten allgemeinen programmiersprachlichen Konzepte und Paradigmen beschränken sich im wesentlichen auf die Unterstützung der Programmiersicherheit sowie der Realisierung statisch überprüfbarer Zugriffsbeschränkungen. So lassen sich zum Beispiel mit den Konzepten zur Kapselung sowie zur Festlegung von Sichtbarkeitsbereichen die Zugriffsmöglichkeiten auf Objekte konzeptionell einschränken, wodurch statische Rechtefestlegungen durchgesetzt werden können. Die angegebenen Konzepte bilden die Basis für die Realisierung sicherer Systeme und sollten deshalb von einer Programmiersprache, die für die Implementierung sicherer Systeme eingesetzt wird, zur Verfügung gestellt werden. Für die Implementierung dynamischer Sicherheitsanforderungen, wie zum Beispiel Rechtefestlegungen, die sich dynamisch zur Laufzeit ändern können und die somit auch dynamische Zugriffskontrollen erfordern, werden jedoch weitergehende Konzepte benötigt.

2.3.2 Spezielle Konzepte

Aus der Literatur sind nur wenige Ansätze bekannt, die sich mit der Erweiterung von Programmiersprachen um spezielle Konzepte zur Realisierung von Sicherheitsanforderungen beschäftigen. Ein Großteil dieser Ansätze, die überwiegend bereits aus den 70'er und 80'er Jahren stammen, konzentriert sich auf die Bereitstellung eines in die Sprache integrierten **Capability–Konzepts**. Daneben gibt es Ansätze, die spezielle programmiersprachliche Konzepte zur Definition bzw. Implementierung von **Zugriffskontrollisten** zur Verfügung stellen.

2.3.2.1 Capability–basierte Ansätze

Mit der Entwicklung von Programmiersprachen, die das Konzept der abstrakten Datentypen beinhalteten, und der zunehmenden Verbreitung des ursprünglich hardwarenah eingeführten Capability–Konzepts (siehe Abschnitt 2.1.3.3), wurden Ende der 70'er und Anfang der 80'er Jahre einige Arbeiten veröffentlicht, die sich mit der Bereitstellung von Capabilities auf programmiersprachlicher Ebene zur Realisierung von Zugriffsbeschränkungen bzw. Zugriffskontrollen beschäftigten ([JL78, KS78, MA79, ABL83, Spi84]). In diesem Zusammenhang wurden für derartige Capabilities die Begriffe **Software–Capability** und **Capability–Variable** geprägt. Die Ansätze ähneln sich darin, daß sie Capabilities als mit einem Datentyp qualifizierte Zugriffsobjekte (Zeiger) einführen, die als Variablen deklariert werden und neben einer Referenz auf ein Objekt dieses Typs eine Menge von Rechten zum Zugriff auf dieses Objekt enthalten. Die potentiell möglichen Rechte entsprechen dabei im einfachsten Fall den einzelnen Operationen, die mit dem Datentyp für die Objekte des Typs definiert sind. Da Rechte im allgemeinen jedoch nicht immer unmittelbar einzelnen Operationen entsprechen müssen, wird in [JL78] und [KS78] die Definition eines abstrakten Datentyps um die Spezifikation einer Menge von Rechten erweitert. Ein Recht ist hierbei ein Name, der einen erlaubten Zugriff auf Objekte des Typs repräsentiert. Damit ist es zum Beispiel möglich, neben einzelnen Operationen auch mehrere Operationen zu einem Recht zusammenzufassen.

Capability–Variablen, insbesondere deren Objektreferenzteil, können analog zu anderen Variablen bzw. Zeigern durch Anweisungen oder Parameterübergabe Werte zugewiesen werden. Damit ist es möglich, die Weitergabe bzw. den Transfer von Rechten zum Beispiel

zwischen Prozessen zu implementieren. Die Ansätze unterscheiden sich jedoch darin, ob neben dem Referenzteil auch der Rechteteil einer Capability–Variable veränderbar ist. In [JL78, KS78, ABL83, Spi84] werden die Rechte bzw. Operationen, zu deren Ausführung eine Capability berechtigt, bereits mit der Capability–Deklaration für die gesamte Lebensdauer der Capability festgelegt; der Rechteteil ist somit konstant. Dies macht spezielle Regeln für die Gültigkeit einer Wertzuweisung zwischen zwei Capability–Variablen erforderlich. Nach der in [JL78] definierten Regel, die dort *binding rule* genannt wird, ist eine Wertzuweisung $x := y$ gültig, wenn die beiden Capability-Variablen x und y mit dem gleichen Typ qualifiziert sind und die Rechtemenge von y die Rechtemenge von x umfaßt[1]. Dies besagt letzlich, daß über eine Capability–Zuweisung nicht mehr Rechte weitergegeben werden können als ursprünglich vorhanden sind. Diese Regel wurde in [ABL83, Spi84] sowie in [KS78] übernommen, wobei in dem Ansatz von [KS78] Änderungen an Capabilities lediglich von speziellen Managerobjekten, den Capability Managern, durchgeführt werden können. In [MA79] wird die Menge der Rechte einer Capability–Variable nicht mit ihrer Deklaration festgelegt. Stattdessen können die Zugriffsrechte einer Capability–Variable durch die Wertzuweisung einer anderen mit dem gleichen Typ qualifizierten Capability, die hierfür ein spezielles *copy*–Recht zur Rechteweitergabe beinhalten muß, verändert werden. Auch hier können jedoch maximal die Rechte der Ursprungs–Capability weitergegeben werden. Bei der Erzeugung eines neues Objekts erhält die Capability, die in der Erzeugungsanweisung angegeben ist, neben einer Referenz auf das Objekt alle Rechte an dem Objekt einschließlich des *copy*–Rechts. Der Ansatz von [MA79] hat den Nachteil, daß trotz des *copy*–Rechts nur eine begrenzte Kontrolle über die Weitergabe von Rechten möglich ist, da im allgemeinen nicht mehr statisch überprüft werden kann, welche Rechte eine Capability enthält. In den Ansätzen von [JL78] und [Spi84] erhält der Erzeuger eines Objekts ebenfalls alle Rechte an dem Objekt, was in diesem Fall allerdings in der Erzeugeanweisung eine Capability–Variable voraussetzt, die mit der gesamten Rechtemenge des Objekts qualifiziert ist. In [KS78] und [ABL83] ist es demgegenüber möglich, durch Angabe einer auf bestimmte Rechte beschränkten Capability in der Erzeugeanweisung Objekte zu erzeugen, auf die generell nur über einzelne der gemäß der Typdefinition potentiell möglichen Rechte zugegriffen werden kann.

Die genannten Ansätze unterscheiden sich weiterhin darin, ob bzw. inwieweit in den entsprechenden Systemen Objekte dynamisch gebunden (allokiert) und damit auch Rechte dynamisch bzw. temporär vergeben werden können. Mit dem Ansatz von [JL78] sind lediglich statische Systeme realisierbar, in denen weder eine dynamische Objektallokation noch eine dynamische Rechtevergabe möglich ist. In den anderen Arbeiten wird der Ansatz von [JL78] um die Möglichkeit der dynamischen Allokation bzw. Deallokation von Objekten verbunden mit einer entsprechend dynamischen Rechtevergabe sowie Konzepten zur Implementierung paralleler Einheiten (Prozesse) erweitert. In [KS78], [ABL83] und [Spi84] werden für die Verwaltung dynamisch allokierter Objekte und der Vergabe von Rechten an diesen über Capabilities spezielle Managerobjekte, die sogenannten **Capability–Manager**, eingeführt. Die Manager werden für Objekttypen definiert und sind die einzigen Objekte, in deren Operationen Bindungen zwischen Capabilities und Objekten des verwalteten Typs verändert sowie explizite Zuweisungen zwischen Capabilities enthalten sein dürfen. Ein Prozeß, der auf ein Objekt zugreifen möchte, muß hierzu auf dem für das Objekt zuständigen Manager eine Operation aufrufen, die die entsprechende Bindung zwischen der als Parameter übergebenen „leeren“, aber durch ihre Deklaration auf bestimmte Rechte beschränkten Capability und dem Objekt herstellt. Nach Nutzung des Objekts kann die Bindung durch Aufruf einer

[1]Die Regel gilt auch für Wertzuweisungen zwischen aktuellen und formalen Parametern im Rahmen einer Parameterübergabe.

entsprechenden Manageroperation wieder gelöst werden. Die Ansätze mit einem Managerkonzept unterscheiden sich darin, daß in [KS78] eine Capability sowohl den Zugriff auf einen Manager als auch auf ein von dem Manager verwaltetes Objekt ermöglicht, während hierfür in [ABL83] bzw. [Spi84] getrennte Capabilities benötigt werden. Eine Capability in [KS78] besteht somit aus zwei Teilen: einem statischen Teil, der die Referenz auf den Manager sowie entsprechende Rechte an diesem enthält, und einem Teil, dessen Referenz zunächst leer ist und die anschließend durch Aufruf einer entsprechenden Manageroperation dynamisch an ein von dem Manager verwaltetes Objekt gebunden werden kann. Durch diese Zusammenfassung kommt deutlich zum Ausdruck, daß die dynamische Zuweisung einer Objektreferenz und damit die Nutzung eines Objekts nur über den entsprechenden Manager erfolgen kann. Darüber hinaus enthält der dynamische Teil einer Capability gemäß [KS78] einen speziellen Gültigkeitsschlüssel, der Basis für die in diesem Ansatz mögliche explizite Rücknahme von Rechten durch Ungültigmachen von Capabilities ist. Obwohl in [MA79] kein explizites Managerkonzept eingeführt wird, können auch dort mit den zur Verfügung stehenden Sprachkonstrukten Manager implementiert werden, die analoge Eigenschaften zu den in [KS78], [ABL83] bzw. [Spi84] definierten Managern besitzen.

In [Eck93] wird ein Capability– und Managerkonzept vorgestellt, das mit den entsprechenden Konzepten aus [Spi84] vergleichbar ist, wobei Capabilities hier zusätzlich um einen kryptographischen Teil zur Realisierung dynamischer Rechterücknahmen erweitert wurden.

Mit den erklärten sprachbasierten Capability–Ansätzen wurde vorrangig das Ziel verfolgt, durch die Einführung von Capability–Variablen für den Zugriff auf Objekte erforderliche Zugriffsüberprüfungen weitestgehend durch den Übersetzer durchführen zu können und dynamische Zugriffskontrollen zur Laufzeit möglichst zu vermeiden (vgl. [JL78, ABL83]). In [JL78] wurde in diesem Zusammenhang für Programme die Eigenschaft der Zugriffskorrektheit (engl.: *access correctness*) definiert, die es gilt, statisch nachzuweisen. Zusätzlich wurde mit der programmiersprachlichen Integration von Capabilities die Hoffnung verbunden, eine große Anzahl von Zugriffskontrollpolitiken bzw. Zugriffsbeschränkungen einfach und verständlich implementieren zu können und somit einen Beitrag zur Entwicklung korrekter Software zu leisten ([JL78]). Durch die Anhebung des hardwarenahen Capability–Konzepts auf das Niveau der Programmierung wird dem Systementwickler jedoch lediglich ein relativ niedriges Abstraktionsniveau zur Festlegung bzw. Realisierung von Zugriffsbeschränkungen geboten. Der Systementwickler muß zum Beispiel die Vergabe von Rechten sowie den Transfer von Rechten über Ein– und Ausgabeparameter oder den Wechsel von Schutzdomänen explizit bei der Codierung der Funktionalität einzelner Komponenten mittels des Capability–Konzepts programmieren. Durch diese enge Kopplung mit dem Code sind zudem die Möglichkeiten zur dynamischen Rechteänderung, d.h. zur Vergabe und Rücknahme von Rechten zur Laufzeit, sehr eingeschränkt. Hierfür ist gegebenenfalls die Reimplementierung von Komponenten einschließlich einer Neuübersetzung erforderlich. Weiterhin ist die Implementierung komplexerer Zugriffsbeschränkungen, die zum Beispiel von dem aktuellen Wert bestimmter Objekte oder der Zugriffshistorie abhängen, mit Software–Capabilities nur sehr begrenzt möglich. Kieburtz und Silberschatz erweitern ihren Capability–basierten Ansatz aus [KS78] zum Beispiel in [KS83] um sogenannte Zugriffsrechtsausdrücke (engl.: *access–right expressions*), mit denen für einen Prozeß eine bestimmte Ausführungreihenfolge für die Operationen auf einem Objekt, zu deren Ausführung er berechtigt ist, festgelegt werden kann. Damit lassen sich für ein Objekt recht einfach Zugriffsberechtigungen vergeben, die von der Zugriffshistorie eines Prozesses bezogen auf das Objekt abhängig sind.

Neben den genannten Ansätzen gibt es einige wenige Arbeiten, in denen ein Capability–Konzept auf Sprachebene vorrangig unter dem Aspekt der Unterstützung einer zugrundelie-

genden Capability–basierten Hardwarearchitektur eingeführt wird ([CFL84, HS89, HS92]). Bei der in [CFL84] eingeführten Sprache handelt es sich um eine maschinennahe Sprache, die spezielle Instruktionen u.a. zum Laden, zur Weitergabe sowie zur Manipulation von Capabilities enthält. Die in [HS89, HS92] erklärte Sprache $\mathcal{X}$ wurde speziell zur Programmierung persistenter Systeme auf einer Capability–basierten Hardwarearchitektur entwickelt. $\mathcal{X}$ beinhaltet den vordefinierten Datentyp `CAP`, mit dem auf der Sprachebene Capabilities zur Verfügung gestellt werden, die denen der Hardwareebene entsprechen und über die Zugriffe auf persistente Objekte erfolgen. Die beiden Ansätze haben ebenfalls den Nachteil, daß das Abstraktionsniveau aufgrund der Hardwarenähe der eingeführten Capabilities sehr niedrig ist.

Im Zusammenhang mit der Bereitstellung geeigneter Betriebssysteminfrastrukturen für verteilte und objekt–orientierte Systeme wurden zahlreiche Betriebssysteme bzw. Mikrokerne entwickelt, die Capabilities als Basisschutzmechanismus einsetzen (z.B. Mach [ABG+86, Loe91], Amoeba [MT86, TMvR86, MvRT+90], Grasshopper [DBF+93a, DBF+93b], Opal [CLFL94]). In derartigen Capability–basierten Betriebssystemarchitekturen kann ein Systementwickler die Capabilities der Betriebssystemebene in der Regel über eine Programmierschnittstelle nutzen, die entsprechende Systembibliotheken (z.B. C-Header Datei `mach.h` in Mach, siehe [BKLL93]) sowie Systemaufrufe zur Allokation und Modifikation von Capbilities (z.B. die `mach_port`–Operationen in Mach) bereitstellt. Capabilities werden hierbei somit nicht als eigenständiges Konzept in eine Programmiersprache integriert, sondern über spezielle in den Systembibliotheken vordefinierte Datentypen (z.B. `mach_port_t` in Mach) zur Verfügung gestellt. Dies hat zum einen den Nachteil, daß statische Zugriffskontrollen durch den Übersetzer nicht bzw. nur sehr eingeschränkt möglich sind. Zum anderen lassen sich mit diesen Capabilities lediglich von dem Betriebssystem(kern) verwaltete Objekte, die vergleichweise grobgranular sind und die im allgemeinen nicht den auf der Anwendungsebene benötigten feingranularen Objekten bzw. der Objektabstraktion der genutzten Programmiersprache entsprechen, schützen. Derartige Capability–basierte Systeme sind daher zur Realisierung von anwendungsspezifischen Sicherheitsanforderungen, die differenzierte Rechtevergaben und feingranulare Zugriffskontrollen erfordern, nur bedingt geeignet.

Ein Ansatz, die bei den bisher erklärten sprachbasierten Capability–Ansätzen vorhandene enge Verknüpfung zwischen der Realisierung der Sicherheitsanforderungen und der Codierung der Anwendungsfunktionalität aufzulösen, wird in [HMRS96] mit dem Konzept der versteckten Capabilities (engl.: *hidden capabilities*) beschrieben. Die Grundphilosophie dieses Ansatzes besteht darin, Capabilities gegenüber einem Anwendungsentwickler in dem Sinne zu „verstecken“, daß die Formulierung der Sicherheitsanforderungen von der Implementierung der Anwendungsfunktionalität getrennt wird. Dies ermöglicht es, den gleichen Anwendungscode mit unterschiedlichen Sicherheitspolitiken ausführen zu können, womit die Flexibiliät und Wiederverwendbarkeit erhöht wird. Die Sicherheitsanforderungen einer Anwendung werden unter Nutzung einer erweiterten Schnittstellen–Definitionssprache (engl.: *Interface Definition Language, IDL*) definiert. Schnittstellen–Definitionssprachen (z.B. IDL innerhalb von CORBA [OMG91]) wurden in Zusammenhang mit der verteilten Client–Server-Programmierung insbesondere zur Realisierung des entfernten Prozeduraufruf–Konzepts (engl.: *Remote Procedure Call, RPC*, [BN84]) eingeführt. Sie bieten einfache Sprachkonzepte, um die Schnittstellen, d.h. Operationen von entfernt zu nutzenden Diensten, unabhängig von deren Implementierung so zu beschreiben, daß aus dieser Beschreibung unter Nutzung eines entsprechenden Compilers automatisch der Code der sogenannten *Stubs* generiert werden kann. Die Stubs nehmen u.a. die benötigten Datenkonvertierungen und Parameteranpassungen zwischen dem Dienstanbieter (Server) und dem Dienstnutzer (Client) vor. Das Schutzmodell von [HMRS96] basiert auf Capabilities und Schutzdomänen (engl.: *protection domain*), wobei eine Schutz-

domäne durch eine Menge von Capabilities charakterisiert ist, die die aktuelle Ausführungsumgebung eines Prozesses (in [HMRS96] als *agents* bezeichnet) festlegt. Jede Schutzdomäne exportiert eine Schnittstelle, die die Operationen der Domäne beinhaltet, die von anderen Domänen aus durch Nutzung einer sogenannten *domain capability* aufgerufen werden können. Mit der vorgeschlagenen erweiterten Schnittstellen–Definitionssprache kann getrennt von der Implementierung der Anwendungsfunktionalität für die einzelnen Schnittstellenoperationen einer Schutzdomäne durch Definition eines *Protected Procedure Interfaces* festgelegt werden, welche minimale Menge an Rechten in Form von Capabilities von der aufrufenden Schutzdomäne als Eingabeparameter an die Zieldomäne übergeben werden muß (z.B. das Leserecht an einer zu druckenden Datei, wenn es sich um einen Druckdienst handelt), bzw. welche Rechte von der aufgerufenen Domäne minimal als Rückgabe zur Nutzung der Ausgabeparameter erwartet werden. Aus dieser Schnittstellenbeschreibung können dann automatisch sogenannte *Protection Stubs* generiert werden, die den Code für die erforderlichen Zugriffskontrollen und das Capability–Handling enthalten. Das Konzept der versteckten Capabilities wurde zwischenzeitlich protoypisch in dem auf einem virtuell gemeinsamen Speicher (engl.: *Distributed Shared Memory, DSM)*) basierenden Ein–Adreßraum–Betriebssystem Arias sowie in einer CORBA– und einer Java–Umgebung implementiert ([HMR96], [HKMR96], [HHM97]). Die Möglichkeiten zur Formulierung komplexerer Zugriffsbeschränkungen sind jedoch auch in diesem Ansatz aufgrund des niedrigen Abstraktionsniveaus des Capability–Konzepts beschränkt.

In die Capability–basierten Ansätze ist auch das in [HO90] eingeführte View–Konzept einzuordnen. Ein View definiert eine spezifische Schnittstelle eines Objekts als Teilmenge der insgesamt von dem Objekt angebotenen Operationen und legt gleichzeitig fest, welche anderen Objekte diese Schnittstelle nutzen dürfen. Mit Views können für ein Objekt mehrere Schnittstellen spezifiziert werden, für die jeweils festgelegt ist, welche Objekte die Operationen der Schnittstelle aufrufen dürfen. Die Rechtefestlegungen eines Systems lassen sich somit als Menge von Views formulieren. In [HO90] werden jedoch keine konkreten Sprachkonstrukte angegeben, so daß weitestgehend unklar bleibt, wie Views programmiersprachlich formuliert werden können. Zudem lassen sich mit dem View–Konzept ebenfalls keine komplexen Zugriffsbeschränkungen, die zum Beispiel von dem aktuellen Ausführungkontext oder der Zugriffshistorie des zugreifenden Objekts abhängig sind, realisieren.

2.3.2.2 Zugriffskontrollisten–basierte Ansätze

Obwohl Zugriffskontrollisten in zahlreichen Betriebssystemen und Ausführungsumgebungen als Basismechanismus für die Zugriffskontrolle eingesetzt werden, gibt es nur sehr wenige Arbeiten, die sich mit der Bereitstellung spezifischer programmiersprachlicher Konzepte für Zugriffskontrollisten beschäftigen. In der Regel werden die Zugriffskontrollisten derartiger Betriebssysteme bzw. Ausführungsumgebungen den Anwendungsentwicklern über Systemaufrufe und Bibliotheksroutinen zur Implementierung von Zugriffsbeschränkungen zur Verfügung gestellt. Die Möglichkeiten zur Implementierung anwendungsspezifischer Zugriffskontrollpolitiken sind dabei stark davon abhängig, für welche Objekte und Rechte das Betriebssystem Zugriffskontrollisten verwaltet. Im allgemeinen sind dies lediglich grobgranulare Objekte (z.B. Dateien und Verzeichnisse) mit einfachen Operationen bzw. Rechten (z.B. lesen, schreiben, ausführen für Dateien). In dem UNIX–Betriebssystem HP–UX [Hew91], das im Vergleich zu anderen UNIX–Derivaten die Vergabe von Zugriffsrechten an einzelne Benutzer auf Basis von Zugriffskontrollisten erlaubt, werden Zugriffskontrollisten zum Beispiel nur für Dateien und Verzeichnisse und den auf diesen definierten Rechten zum Lesen (`r`), Schreiben (`w`) und Ausführen (`x`) unterstützt. Zugriffskontrollen werden lediglich beim Öffnen einer Datei bzw.

eines Verzeichnisses für die dabei anzugebenen Rechte durchgeführt, was zur Konsequenz hat, daß Rechterücknahmen im allgemeinen nicht unmittelbar wirksam werden. Zudem kann eine Zugriffskontrolliste in HP–UX maximal fünfzehn Einträge enthalten. Zugriffskontrollen für feingranulare Objekte der Anwendungsebene mit differenzierten Operationen lassen sich auf dieser Basis nicht implementieren.

Ein erweitertes Zugriffskontrollisten–Konzept, mit dem sich differenzierterer Zugriffsbeschränkungen und –kontrollen implementieren lassen, bietet das OSF *Distributed Computing Environment* (DCE) [OSF92, Hu95]. Das DCE stellt dem Anwendungsentwickler u.a. Bibliotheksroutinen zur expliziten Implementierung von Zugriffskontrollisten (ACL) für Anwendungsobjekte zur Verfügung. Dabei ist es möglich, bis zu 32 Operationen als Rechte für die Nutzung eines zu schützenden Objekts festzulegen. Die Verwaltung dieser Zugriffskontrollisten erfolgt durch anwendungsspezifisch zu implementierende Zugriffskontrollisten–Manager (ACL–Manager), für deren Realisierung ebenfalls Bibliotheksroutinen bereitgestellt werden. Analog zu den Capability–basierten Ansätzen wird einem Systementwickler mit den Zugriffskontrollisten lediglich ein niedriges Abstraktionsniveau zur Festlegung bzw. Realisierung von Zugriffsbeschränkungen geboten. Die Vergabe komplexerer Zugriffsberechtigungen ist auf Basis der üblichen Zugriffskontrollisten ebenfalls nicht möglich.

In [WL93] wird mit GACL eine eigenständige Sprache zur Spezifikation von Zugriffskontrollisten für verteilte Client–Server Systeme vorgestellt. Mit der Sprache GACL lassen sich für die von einem Server angebotenen Objekte bzw. Dienste sogenannte verallgemeinerte Zugriffskontrollisten (*gacl – generalized access control list*) spezifizieren, deren Ausdrucksmöglichkeiten deutlich über die der üblichen Zugriffskontrollisten hinausgehen. So lassen sich durch spezielle Zugriffskontrollisteneinträge Zugriffsrechte in Abhängigkeit von den Werten bestimmter Systemprädikate (z.B. der Systemlast) oder abhängig von dem Vorhandensein bestimmter Rechte an anderen Objekten vergeben. Weiterhin können für jede *gacl* durch die Angabe eines Schlüsselwortes spezielle Regeln für ihre Auswertung festgelegt werden. Die Zugriffskontrollen bzgl. der Nutzung eines Objekts werden anhand der für das Objekt spezifizierten *gacl* durch einen von dem für die Verwaltung des Objekts zuständigen Server beauftragten vertrauenswürdigen Autorisierungsserver durchgeführt. Die hierzu von dem Autorisierungsserver benötigten Autorisierungsinformationen sind im Rahmen der Registrierungsphase der Serverdienste zwischen dem Server und dem Autorisierungsserver über ein spezielles Protokoll in Form eines GACL–Programms auszutauschen. In [WL93] wird an die Autorisierungsserver die Anforderung gestellt, daß sie zur Durchführung der Zugriffskontrollen in der Lage sein müssen, entweder ein GACL–Programm direkt interpetieren zu können oder GACL–Programme in Form einer geeigneten – noch zu entwickelnden – Übersetzung ausführen zu können. Eine Implementierung für die Autorisierungsserver wird in [WL93] allerdings nicht angegeben. Der Ansatz von [WL93] ist vorrangig für die Spezifikation von Zugriffsberechtigungen in klassischen Client–Server Anwendungen, in denen Server Operationen zur Nutzung eher grobkörniger Objekte (wie z.B. Dateien oder Drucker) anbieten, geeignet. Der Zusammenhang zwischen einem Anwendungsprogramm und den darin definierten Objekten und einem GACL–Programm, in dem die Zugriffsbeschränkungen für diese Objekte spezifiziert werden, bleibt weitestgehend unklar. Weiterhin bleibt offen, wie dynamische Rechteänderungen formuliert und realisiert werden können.

2.3.3 Fazit

Für die Konstruktion sicherer Systeme sind geeignete programmiersprachliche Konzepte und darauf basierende statische und dynamische Zugriffskontrollen unverzichtbar. Hierzu gehören

insbesondere das Konzept der abstrakten Datentypen, ein strenges Typkonzept, Konzepte zur differenzierten Festlegung von Sichtbarkeitsbereichen und ein Ausnahmebehandlungskonzept. Daneben werden Konzepte zur Konstruktion feingranularer Objekte mit differenzierten Operationen benötigt. Existierende Sprachen stellen diese für die Implementierung sicherer Systeme grundlegenden Konzepte nur teilweise zur Verfügung. So bietet zum Beispiel C++ ein differenziertes Objektkonzept, ist jedoch aufgrund der Verwandtschaft mit C nicht streng typisiert. Am weitesten fortgeschritten ist in dieser Hinsicht die objekt–orientierte Programmiersprache Java, die neben einem flexiblen Objektkonzept ein Ausnahmebehandlungskonzept beinhaltet und streng typisiert ist.

Die genannten grundlegenden Konzepte unterstützen zum einen die Programmiersicherheit. Zum anderen lassen sich mit diesen Konzepten bereits einfache statische Zugriffsfestlegungen realisieren, deren Einhaltung durch den Übersetzer im Rahmen statischer Analysen und Kontrollen überprüft werden kann. Zur Implementierung von Systemen mit komplexeren Zugriffskontrollpolitiken sowie mit dynamischer Objektmenge und dynamischen Rechteänderungen sind diese Konzepte jedoch bei weitem nicht ausreichend.

Wie aus dem vorhergehenden Abschnitt ersichtlich ist, beschränken sich die vorhandenen Ansätze, die sich mit der Bereitstellung spezieller progammiersprachlicher Konzepte zur Realisierung sicherer Systeme beschäftigen, darauf, *low–level* Schutzmechanismen der Hardware– bzw. Betriebssystemebene in eine Programmiersprache zu integrieren bzw. diese auf der Sprachebene nutzbar zu machen. Durch diese *bottom–up* Vorgehensweise wird dem Systementwickler lediglich ein niedriges Abstraktionsniveau zur Realisierung von Sicherheitsanforderungen geboten. Er muß die Durchsetzung der Sicherheitsanforderungen, wie zum Beispiel die für die Realisierung von Zugriffsbeschränkungen erforderlichen Zugriffskontrollen, explizit unter Nutzung der bereitgestellten *low–level* Konzepte programmieren, was im allgemeinen aufwendig und fehlerträchtig ist. Zudem lassen sich mit diesen einfachen Konzepten komplexerer Zugriffsbeschränkungen, wie zum Beispiel die Abhängigkeit eines Zugriffsrechts von dem aktuellen Ausführungkontext eines Subjekts oder dessen Zugriffshistorie, sowie dynamische Rechteänderungen entweder gar nicht oder nur sehr eingeschränkt realisieren.

Für die systematische *top–down* orientierte Konstruktion sicherer Systeme werden stattdessen programmiersprachliche Konzepte auf hohem Abstraktionsniveau benötigt, mit denen Sicherheitsanforderungen unabhängig von Realisierungsmechanismen weitestgehend deklarativ implementiert werden können. Ein Beispiel hierfür ist die Formulierung von Zugiffsbeschränkungen durch die Angabe von Vorbedingungen für die Ausführung der Operationen zu schützender Objekte. Durch den Übersetzer ist dann automatisch Code zu generieren, der unter Rückgriff auf Sicherheitsmechanismen und –dienste der zugrundeliegenden Ausführungsbasis bei jedem Operationsaufruf überprüft, ob die entsprechende Vorbedingung für das aufrufende Subjekt erfüllt ist. Der Systementwickler wird damit von rein realisierungstechnischen Maßnahmen zur Realisierung der Zugriffsbeschränkungen, wie zum Beispiel die Implementierung von Zugriffskontrollisten oder die Verwaltung von Capabilities, entlastet.

Ein Aspekt, der in den bisher zitierten Arbeiten mit Ausnahme von [WL93] überhaupt nicht beleuchtet wird, ist die Frage der Konsistenz der mit den programmiersprachlichen Konzepten implementierten Rechtefestlegungen. Dabei eröffnet gerade die sprachbasierte Festlegung von Zugriffsbeschränkungen die Möglichkeit, durch geeignete statische und dynamische Analysen Aussagen über die Konsistenz der Rechtefestlegungen zu gewinnen. Ein Überblick über in der Literatur vorhandene Konsistenzbegriffe und Arbeiten, die sich allgemein mit der Konsistenzproblematik in zugriffskontroll–basierten Systemen beschäftigen, wird in Kapitel 5 dieser Arbeit gegeben.

In den beiden folgenden Kapiteln werden nunmehr geeignete Sprachkonzepte mit hohem

Abstraktionsniveau für die Implementierung von Systemen mit anwendungsspezifisch festgelegten Zugriffskontrollpolitiken angegeben. Die verteilte objekt–basierte Programmiersprache INSEL, die im Rahmen des MoDiS–Projekts [EW95, Eck96, PE97, EP98] (*Model–oriented Distributed Systems*) entwickelt wurde, stellt hierfür eine geeignete Basis dar. INSEL wird in dieser Arbeit um spezifische Konzepte zur Konstruktion sicherer Anwendungssysteme zu der Sprache INSEL$^+$ erweitert. INSEL$^+$ stellt im Vergleich zu INSEL neben zusätzlichen Möglichkeiten zur differenzierten Festlegung von Sichtbarkeitsbereichen bzw. Ausführungsumgebungen und einem einfachen Ausnahmebehandlungskonzept insbesondere Konzepte zur Konstruktion sogenannter zugriffskontrollierter Komponenten, für deren Nutzung komplexe Zugriffsbeschränkungen festgelegt werden können, zur Verfügung.

Kapitel 3

Konzepte der Sprache INSEL

INSEL (*Integration and Separation supporting Language*) ist eine imperative, objekt–basierte Programmiersprache, die im Rahmen des MoDiS–Projekts (vgl. Kapitel 1) entwickelt wurde. Sie bietet Konzepte mit hohem Abstraktionsniveau zur Konstruktion paralleler und kooperativer Anwendungssysteme nach einem sprachbasierten *top–down* Ansatz und stellt eine geeignete Ausgangsbasis für die Entwicklung einer Programmiersprache zur Realisierung sicherer Systeme dar. In diesem Kapitel werden deshalb die wesentlichen, für das Verständnis der weiteren Kapitel dieser Arbeit benötigten Konzepte der Sprache INSEL vorgestellt. Der Schwerpunkt liegt dabei auf der Beschreibung der grundlegenden Konzepte und Prinzipien. Auf die Angabe konkreter Sprachkonstrukte und Syntaxregeln wird weitestgehend verzichtet. Eine ausführliche Darstellung der INSEL zugrundeliegenden Konzepte ist in [SEL+96] zu finden. Die Syntaxregeln von INSEL sind in [RW96] angegeben.

Dieses Kapitel ist wie folgt aufgebaut. In Abschnitt 3.1 werden zunächst grundlegende Eigenschaften und Prinzipien der Sprache INSEL und der damit konstruierten Systeme erklärt. Abschnitt 3.2 erläutert die Konzepte von INSEL zur Konstruktion aktiver und passiver Komponenten. In Abschnitt 3.3 werden die Entwicklungsmöglichkeiten von INSEL–Systemen, die sich aus der dynamischen Erzeugung und Auflösung von Komponenten ergeben, dargestellt. Weiterhin werden dort Strukturrelationen eingeführt, mit denen die verschiedenen sich aus dem Einsatz der INSEL–Strukturierungskonzepte ergebenen Abhängigkeiten zwischen den Komponenten eines INSEL–Systems beschrieben werden. Die Ausführung einer Operation in einem INSEL–System läßt sich in mehrere Phasen untergliedern, die in Abschnitt 3.4 benannt und erläutert werden. Abschnitt 3.5 definiert schließlich die Ausführungsumgebung einer Komponente als die Menge der von der Komponente potentiell nutzbaren Komponenten. Die Ausführungsumgebung einer Komponente ergibt sich aus der strukturellen Einordnung der Komponente und den für INSEL festgelegten Sichtbarkeitsregeln. Abschließend wird im Abschnitt 3.6 verdeutlicht, daß die Konzepte der Sprache INSEL eine geeignete Ausgangsbasis für die Konstruktion sicherer Systeme bieten. Darauf aufbauend werden notwendige Erweiterungen von INSEL motiviert, die für die Realisierung anwendungsspezifischer Zugriffskontrollpolitiken benötigt werden.

3.1 INSEL–Systeme

Der INSEL zugrundeliegende Systembegriff orientiert sich an dem in Abschnitt 2.1 eingeführten Begriff eines Rechensystems. Unter einem **INSEL–System** wird im weiteren ein Rechensystem verstanden, das ausschließlich mit den Konzepten der Sprache INSEL konstruiert wur-

de. INSEL–Systeme bestehen aus aktiven und passiven Komponenten, die dynamisch erzeugt und aufgelöst werden und die ihrerseits aus Komponenten zusammengesetzt sein können. Die aktiven Komponenten führen Berechnungen aus und benutzen dafür passive Komponenten.

INSEL–Systeme sind objekt–basiert, d.h. daß ihre Komponenten nach dem **Objektprinzip** konstruiert werden. Das Objektprinzip besagt, daß die inneren und äußeren Eigenschaften einer Komponente (allgemeiner eines Objekts) durch innere und äußere Operationen zu definieren sind, diese wohl zu unterscheiden sind und daß eine Komponente von außen allein durch die Ausführung ihrer äußeren Operationen benutzbar ist.

Eng mit dem Objektprinzip zusammenhängend ist das **Klassenprinzip**. Dieses besagt, daß die Komponenten eines INSEL–Systems als Instanzen von Komponentenklassen erzeugt werden. Komponentenklassen werden in INSEL als **Generatoren** bezeichnet. Generatoren sind selbst wiederum Komponenten, die jedoch nicht als Instanz bzgl. eines Generators sondern allein durch Erarbeitung ihrer Deklaration erzeugt werden. Komponenten, die als Instanz bzgl. eines Generators erzeugt werden, werden in INSEL **Inkarnationen** genannt. Mit der Definition eines Generators werden die gemeinsamen Eigenschaften aller Inkarnationen, die bzgl. dieses Generators erzeugt werden, festgelegt. Generatoren sind parametrisierbar, so daß bei der Erzeugung einer Inkarnation als Instanz eines Generators weitere inkarnationsspezifische Eigenschaften festgelegt bzw. ausgeprägt werden können. Die konsequente Anwendung des Klassenprinzips führt dazu, daß INSEL eine streng typisierte Sprache ist. Da INSEL jedoch weder ein Vererbungskonzept beinhaltet noch Polymorphismus unterstützt, sind INSEL–Systeme nach der in [Weg87] eingeführten Begriffsdefinition nicht objekt–orientiert.

Ein weiteres Prinzip, das bei Konstruktion von INSEL–Systemen zur Anwendung kommt, ist das **Schachtelungsprinzip**. Dieses besagt, daß für jede Komponente eines Systems deren Einordnung innen bzw. außen zu anderen Komponenten des Systems als Komponenteneigenschaft und somit für ihre gesamte Lebenszeit festgelegt ist. Im Gegensatz zu anderen objekt–basierten bzw. objekt–orientierten Sprachen sind auch Generatoren schachtelbar und können in Deklarationsteilen von Komponenten auftreten. Die Einordnung eines Generators in die Gesamtstruktur eines INSEL–Systems ergibt sich aus der textuellen Aufschreibung des entsprechenden INSEL–Programms. Über die Einordnung eines Generators in die Gesamtstruktur wird die strukturelle Einordnung seiner Inkarnationen sowie die Menge der für diese sichtbaren und damit potentiell nutzbaren Komponenten festgelegt. Die strikte Anwendung des Schachtelungsprinzips hat zur Folge, daß sich ein INSEL–System aus einer Komponente, der Hauptkomponente des Systems, entwickelt, zu der alle weiteren Komponenten innen sind.

Die Menge der in einem INSEL–System zu einem Zeitpunkt existierenden Komponenten ist nach strukturellen Abhängigkeiten, die durch die Konstruktion des Systems festgelegt sind, geordnet. INSEL–Systeme sind in diesem Sinne konzeptionell strukturierte Systeme. Darin unterscheiden sich INSEL–Systeme wesentlich von Systemen, die mit anderen objekt–basierten bzw. objekt–orientierten Sprachen konstruiert werden. Dort bilden im allgemeinen die Klassen eine Hierarchie, wodurch jedoch keine Abhängigkeiten für deren Instanzen definiert werden. Die Objekte liegen in der Regel flach in einem großen Namensraum nebeneinander, und jedes Objekt kann potentiell jedes andere Objekt nutzen. Demgegenüber lassen sich in INSEL auf Basis der beliebig tiefen Schachtelung von Komponenten implizite Abhängigkeiten zwischen Komponenten sowie die Menge der für eine Komponente potentiell sichtbaren und damit nutzbaren Komponenten festlegen. Unter systematischer Ausnutzung der Schachtelungsmöglichkeiten können Subsysteme konstruiert werden, die Komponenten, die nur lokal in einem Subsystem benötigt werden, so kapseln, daß sie außerhalb des Subsystems nicht sichtbar sind. Auf diese Weise lassen sich per Konstruktion Nutzungsmöglichkeiten von Komponenten konzeptionell einschränken.

3.2 Komponenten–Konzepte

In diesem Abschnitt werden die in INSEL zur Verfügung stehenden Konzepte zur Konstruktion von Komponenten angegeben. Den einzelnen Konzepten entsprechen Komponentenarten, wobei mit jedem Konzept die grundlegenden Eigenschaften für die Komponenten seiner Art festgelegt sind.

3.2.1 Generatoren und Inkarnationen

Ausgehend von dem Klassenprinzip wird in INSEL zwischen Generatoren und Inkarnationen unterschieden. **Generatoren** sind die klassendefinierenden Komponenten. Auf jedem Generator ist implizit die äußere Operation *erzeuge* definiert. Die Ausführung von *erzeuge* auf einem Generator bewirkt, daß eine Inkarnation als Instanz der durch den Generator definierten Klasse erzeugt wird. Generatoren sind die einzigen Komponenten eines INSEL–Systems, die allein durch Erarbeitung ihrer Deklaration erzeugt werden. Alle anderen Komponenten werden durch die Ausführung der Operation *erzeuge* auf einem bereits existierenden Generator erzeugt. Das Generator–Konzept ist vergleichbar mit dem Typ–Konzept gängiger imperativer Programmiersprachen bzw. dem Klassenkonzept in objekt–basierten Sprachen. Im einfachsten Fall entspricht eine Generatordefinition der Definition eines Datentyps.

Gemäß dem im vorhergehenden Abschnitt erklärten Objektprinzip, sind die inneren und äußeren Eigenschaften einer Komponente durch innere und äußere Operationen zu definieren. Diese Operationen können explizit durch den Programmierer definiert werden oder implizit definiert (vordefiniert) sein. Inkarnationen, bei denen konzeptionell zwischen ihren inneren und äußeren Eigenschaften unterschieden wird, werden in INSEL **DA–Inkarnationen** genannt. Die Inkarnationen, deren innere und äußere Eigenschaften zusammenfallen und für die alle Operationen vordefiniert sind, werden als **DE–Inkarnationen** bezeichnet. Jede DE–Inkarnation ist als lokale Komponente in eine DA–Inkarnation eingeordnet. DE–Inkarnationen sind zum Beispiel einfache Datenobjekte wie eine `integer`–Variable oder ein Zeiger. Eine DA–Inkarnation kann zum Beispiel ein Modul sein, der Zugriffsoperationen zum Zugriff auf seine lokalen Komponenten anbietet. Die Eigenschaft, DA– oder DE–Inkarnation zu sein, wird bereits durch den entsprechenden Generator festgelegt. Dementsprechend können Generatoren in DE– und DA–Generatoren unterschieden werden. Abbildung 3.1 verdeutlicht die eingeführte Klassifikation der Komponenten eines INSEL–Systems. Die in der Abbildung bereits angegebenen Unterarten von Generatoren bzw. Inkarnationen werden in den beiden folgenden Abschnitten noch genauer erläutert.

In INSEL wird zwischen benannten und anonymen Komponenten unterschieden. Für **benannte** Komponenten, die auch **N–Komponenten** genannt werden, ist während ihrer gesamten Lebenszeit ein in ihrem jeweiligen Kontext eindeutiger Name festgelegt, über den sie identifiziert werden können. Benannte Komponenten werden durch die Erarbeitung einer Deklaration erzeugt. **Anonyme** bzw. **Z–Komponenten** werden demgegenüber dynamisch durch die Auswertung eines Generierungsausdrucks[1], der immer als Teil eines Konstrukts mit Wertzuweisung an einen Zeiger auftritt, erzeugt. Dabei wird ein Zeigerwert erzeugt, über den die jeweilige Komponente identifiziert werden kann und der dem entsprechenden Zeiger zugewiesen wird. Auf anonyme Komponenten kann somit nur mittelbar über Zeiger zugegriffen werden, während benannte Komponenten direkt über ihren Namen referenziert werden. DE–

[1]Generierungsausdrücke werden in INSEL auf Basis des aus anderen Programmiersprachen (z.B. C++ [Str91]) bekannten `new`–Operators formuliert.

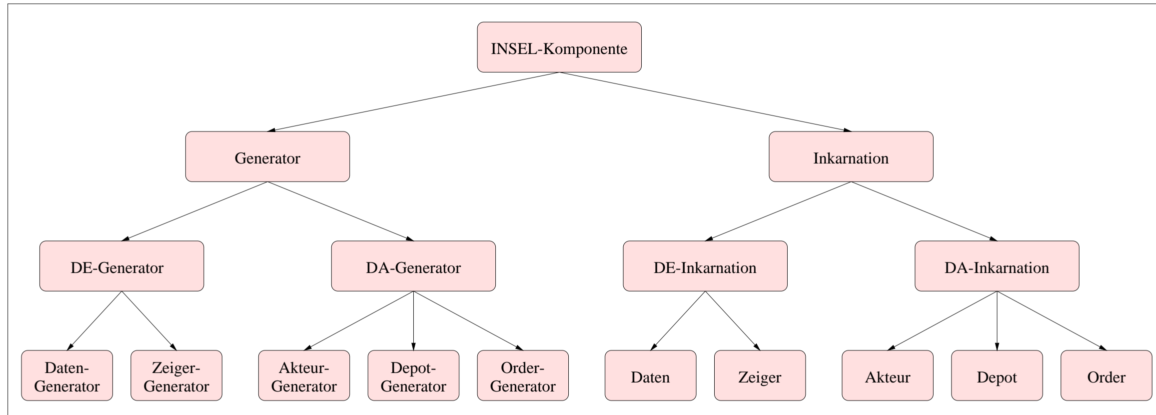

Abbildung 3.1: Klassifikation der INSEL–Komponenten

und DA–Inkarnationen können entweder benannt oder anonym sein; Generatoren sind stets benannte Komponenten.

3.2.2 DE–Inkarnationen

Gemäß Abbildung 3.1 lassen sich die DE–Inkarnationen in **Daten** und **Zeiger** klassifizieren. In INSEL gibt es verschiedene Arten von Daten, deren Struktur und Wertemenge durch entsprechende DE–Generatoren festgelegt werden. Neben den vordefinierten Generatoren für die elementaren Datentypen (`boolean, char, integer, real` und `string`) können in INSEL Bereichs–, Record– und Feld–Generatoren definiert werden.

Zeiger werden zur Identifikation anonymer Komponenten benutzt. Jeder Zeiger ist mit einem Generator qualifiziert und kann nur auf anonyme Inkarnationen verweisen, die bzgl. dieses Generators erzeugt wurden. Der Generator, mit dem ein Zeiger qualifiziert, wird mit dem entsprechenden Zeiger–Generator festgelegt. Ein Wertzuweisung zwischen zwei Zeigern ist nur dann erlaubt, wenn die beiden Zeiger Inkarnationen bzgl. des gleichen Generators sind. Eine Zeigerarithmetik gibt es in INSEL nicht, so daß damit verbundene Sicherheitsprobleme, die zum Beispiel aus der Möglichkeit zum direkten Speicherzugriff oder dem „Casting“ resultieren, von vornherein ausgeschlossen sind.

Für jede DE–Inkarnation sind die Operationen zum Lesen ihres Wertes und zum Schreiben eines Wertes vordefiniert.

3.2.3 DA–Inkarnationen

INSEL stellt Konzepte für die Konstruktion der folgenden drei Arten von DA–Inkarnationen zur Verfügung: Akteure, Depots und Order (vgl. Abbildung 3.1). Akteure sind die aktiven Komponenten, mit denen es möglich ist, explizit parallele Kontrollflüsse zu etablieren. Order und Depots sind die passiven Komponenten, wobei Order Operationen definieren und ein Depot abstrakten Speicher und Operationen zu dessen Benutzung bereitstellt. Bevor im weiteren näher auf die einzelnen DA–Inkarnationsarten eingegangen wird, werden zunächst die allen DA–Inkarnationen gemeinsamen Eigenschaften angegeben.

(3.1) Eigenschaften einer DA–Inkarnation

Eine DA–Inkarnation x hat folgende Eigenschaften:

- Sie besteht aus einem Deklarationsteil und einem Anweisungsteil. Der Deklarationsteil enthält Deklarationen benannter Komponenten. Die Menge der mit dem Deklarationsteil von x festgelegten lokalen N–Komponenten von x wird mit $L_0(x)$ bezeichnet. Abhängig von der DA–Inkarnationsart können Elemente aus $L_0(x)$ nach außen exportiert werden und damit von außen nutzbar gemacht werden. Dazu werden diese Elemente bei ihrer Deklaration explizit mit dem Attribut E (für Export) versehen. Ist eine lokale N–Komponente von x mit dem Attribut E definiert, dann sind alle äußeren Operationen dieser Komponente äußere Operationen von x. Die Menge der mit dem Attribut E definierten lokalen N–Komponenten von x wird mit $L_0^E(x)$ bezeichnet.
- Für sie ist explizit genau eine innere Operation definiert. Diese Operation wird **kanonische Operation** von x genannt und mit $op(x)$ bezeichnet. $op(x)$ wird im wesentlichen durch den Anweisungsteil von x definiert.
- Für sie ist implizit eine ausgezeichnete äußere Operation definiert, deren Aufruf die Ausführung von $op(x)$ bewirkt. Diese Operation wird im weiteren **Startoperation** genannt und mit $i(x)$ bezeichnet.
- Sie befindet sich zu jedem Zeitpunkt ihrer Existenz in genau einem der Zustände V (vorbereitet), A (in Ausführung), R (rechnend), W (wartend) oder T (terminiert). Unmittelbar nach ihrer Erzeugung befindet sich die DA–Inkarnation x im Zustand V. Dies ist Voraussetzung für die Ausführung der Startoperation $i(x)$ und damit der kanonischen Operation $op(x)$ von x. Abhängig von der Inkarnationsart bewirkt die Ausführung von $i(x)$ die Überführung von x in den Zustand A, falls x Order oder Depot ist bzw. in den Zustand R, falls x ein Akteur ist. Mit Abschluß der Ausführung der kanonischen Operation $op(x)$ wird x in den Zustand T überführt. Dieser Zustand ist Voraussetzung für die Auflösung von x. Auf den Zustand W, der nur für Akteure definiert ist, wird weiter unter noch eingegangen.

Im folgenden werden die spezifischen Eigenschaften der einzelnen DA–Inkarnationsarten erklärt.

Order

Order sind operationendefinierende Komponenten. Sie entsprechen im wesentlichen den aus anderen imperativen Programmiersprachen bekannten Prozeduren, Funktionen und Blöcken. Dementsprechend werden in INSEL **PS-Order**, die die Eigenschaften von Prozeduren haben, **FS-Order** mit den Eigenschaften von Funktionen und **BS-Order** mit den Eigenschaften von Blöcken unterschieden. Eine Order ist im wesentlichen durch ihre kanonische Operation charakterisiert. Die Startoperation einer Order heißt *führe_aus*; weitere äußere Operationen können für Order nicht definiert werden. Alle Order sind implizit anonyme Komponenten, die jedoch nicht explizit über Zeiger identifizierbar sind. Die drei genannten Unterarten der Order werden auch als **S-Order** bezeichnet, da ihre kanonische Operation sequentiell eingeordnet von dem jeweils erzeugenden Akteur ausgeführt wird. **K-Order** sind Kommunikationsoperationen, die die Kooperation zwischen Akteuren nach dem Operationen–orientierten Rendezvous–Konzepts ermöglichen. Hierauf wird weiter unten bei der Erklärung des Akteur–Konzepts noch genauer eingegangen. Order werden durch Ausführung entsprechender Order–Aufrufe, die in Anweisungsteilen von DA–Inkarnationen auftreten können, erzeugt.

Depots

Ein Depot definiert mit seinen lokalen Komponenten (abstrakten) Speicher und stellt explizit definierte äußere Operationen, die Zugriffsoperationen genannt werden, für dessen Benutzung zur Verfügung. Depots sind vergleichbar mit den üblichen Objekten objekt–basierter bzw. objekt–orientierter Sprachen, wobei die Zugriffsoperationen den Methoden entsprechen. Für jedes Depot ist implizit die Startoperation *initialisiere* definiert. Die Zugriffsoperationen eines Depots sind explizit zu definieren, in dem lokale N–Komponenten des Depots mit dem Attribut E deklariert werden. Obwohl konzeptionell für Depots alle Arten von lokalen N–Komponenten mit dem Attribut E definiert werden dürfen, also auch einfache DE–Inkarnationen, sind für die Definition der Zugriffsoperationen in erster Linie FS- und PS-Order–Generatoren von Interesse. Ist eine lokale DE–Inkarnation eines Depots mit dem Attribut E definiert, so bedeutet dies, daß die Lese– und Schreiboperation auf der DE–Inkarnation äußere Operationen des Depots sind und auf diese Inkarnation somit von außerhalb des Depots direkt zugegriffen werden kann. Dies entspricht jedoch nicht dem Prinzip, Daten innerhalb eines Depots zu kapseln und diese nur über wohldefinierte Zugriffsoperationen von außen nutzbar zu machen. Wird ein lokaler Generator eines Depots mit dem Attribut E definiert, so ist die *erzeuge*–Operation dieses Generators äußere Operation des Depots. Handelt es sich bei dem Generator um einen FS- oder PS-Order–Generator, wird beim Aufruf der damit festgelegten Zugriffsoperation des Depots eine entsprechende Order erzeugt und die durch diese Order definierte Operation ausgeführt. Dies entspricht dem, was intuitiv unter einer Zugriffsoperation verstanden wird. Die Ausführung der kanonischen Operation eines Depots dient im wesentlichen dazu, die lokalen speicherdefinierenden Komponenten des Depots zu initialisieren. Die Ausführung der Zugriffsoperationen eines Depots setzt voraus, daß sich dieses im Zustand T befindet, die Ausführung der kanonischen Operation und damit die Initialisierungsphase somit bereits abgeschlossen ist.

Depots können benannte oder anonyme Komponenten sein. Benannte Depots werden durch Erarbeitung von Depot–Deklarationen, die in Deklarationsteilen von DA–Inkarnationen auftreten können, erzeugt. Ein Depot ist eine lokale N–Komponente der DA–Inkarnation, in deren Deklarationsteil die entsprechende Depot–Deklaration enthalten ist. Ein anonymes Depot wird erzeugt, indem ein entsprechender Generierungsausdruck ausgewertet wird. Dieser kann im Rahmen einer Zeiger–Deklaration Bestandteil des Deklarationsteils einer DA–Inkarnation oder als Generierungsanweisung in deren Anweisungsteil enthalten sein.

In [Win96] wird das Depot–Konzept unter dem Aspekt der konzeptionellen Festlegung von Synchronisationsregeln für die Ausführung der Zugriffsoperationen eines Depots weiter differenziert. Neben nicht synchronisierten Depots (NS-Depots) werden Monitor–Depots und Leser–Schreiber–Depots eingeführt. Für Monitor–Depots gilt, daß ihre Zugriffsoperationen wechselseitig ausgeschlossen ausgeführt werden. Bei einem Leser–Schreiber–Depot sind die Zugriffsoperationen explizit in Lese– und Schreiboperationen zu klassifizieren. Die Ausführung der Zugriffsoperationen wird dann gemäß der üblichen Leser–Schreiber–Semantik synchronisiert.

Akteure

Akteure sind die aktiven Komponenten eines INSEL–Systems. Sie besitzen neben Speicherfähigkeiten zusätzlich Rechenfähigkeiten und führen damit Berechnungen durch, indem sie Operationen ausführen. Die Erzeugung eines Akteurs etabliert einen zusätzlichen Konrollfluß, der parallel zum Kontrollfluß der erzeugenden DA–Inkarnation ausgeführt wird. Akteure werden, wie alle anderen Komponenten eines INSEL–Systems auch, explizit dynamisch erzeugt und gemäß konzeptionell festgelegter Regeln wieder aufgelöst. Damit ermöglicht das

Akteur–Konzept die Konstruktion von Systemen mit expliziter Parallelverarbeitung sowie mit flexiblem und dynamischem Parallelitätsgrad ([Spi92]). Mit dem Akteur–Konzept steht dem Systementwickler ein uniformes Konzept mit hohem Abstraktionsniveau zur Konstruktion aktiver Einheiten zur Verfügung, das bewußt von „klassischen“ Realisierungskonzepten für Aktivitäten, wie leichtgewichtige Threads oder schwergewichtige Prozesse, abstrahiert.

Die Ausführung der implizit auf jedem Akteur definierten Startoperation *starte* bewirkt die Überführung des Akteurs in den Zustand R; damit beginnt der Akteur mit der Ausführung seiner kanonischen Operation. Bei Ausführung seiner kanonischen Operation kann ein Akteur neue Inkarnationen, insbesondere Order, Depots sowie weitere Akteure erzeugen. Ein Akteur, der eine Order oder ein Depot erzeugt, führt anschließend deren bzw. dessen kanonische Operation sequentiell eingeordnet in die Operation, bei deren Ausführung die neue Inkarnation erzeugt wurde, aus. Dementsprechend ist einem Akteur zu jedem Zeitpunkt seiner Existenz eindeutig eine **Ausführungskomponente** zugeordnet, nämlich die DA–Inkarnation, deren kanonische Operation er gerade ausführt. Dies kann entweder der Akteur selbst, eine Order oder ein Depot sein. Die Ausführungskomponente eines Akteurs wechselt mit der Erzeugung einer neuen Order bzw. eines neuen Depots durch den Akteur sowie mit dem Abschluß der Ausführung der kanonischen Operation der Order bzw. des Depots.

Erzeugt ein Akteur einen neuen Akteur, so wird damit ein neuer Kontrollfluß etabliert. Nach der Anfangssynchronisation des erzeugenden Akteurs mit dem erzeugten Akteur durch Aufruf dessen Startoperation führen die beiden Akteure ihre kanonischen Operationen parallel zueinander aus.

In INSEL können zwei Arten von Akteuren konstruiert werden: M-Akteure und K-Akteure.

M-Akteure können neben der vordefinierten Startoperation keine weiteren äußeren Operationen besitzen. Sie sind in diesem Sinne mono–operational d.h. sie führen die Berechnung ihrer kanonischen Operation ohne unmittelbare Beeinflussung durch andere Akteure aus. Alle M-Akteure sind implizit anonyme Komponenten, die jedoch – ebenso wie die Order – nicht explizit über Zeiger identifiziert werden können. M-Akteure werden durch Ausführung entsprechender M-Akteur–Aufrufe, die in Anweisungsteilen von DA–Inkarnationen auftreten können, erzeugt. Das M-Akteur–Konzept dient hauptsächlich zur Parallelisierung von Berechnungen.

K-Akteure sind Akteure, für die explizit äußere Operationen definiert sein können. Die explizit definierten äußeren Operationen eines K-Akteurs werden als Kommunikationsoperationen bezeichnet. Diese werden durch lokale mit dem Attribut E zu deklarierende K-Order–Generatoren, die lediglich in den Deklarationsteilen von K-Akteuren erlaubt sind, definiert. Weitere lokale N–Komponenten eines K-Akteurs können nicht mit dem Attribut E definiert werden. Andere Akteure können mit einem K-Akteur kooperieren, indem sie dessen Kommunikationsoperationen aufrufen. Diese Kooperation erfolgt nach dem **Operationen–orientierten Rendezvous–Konzept**, das im folgenden kurz erklärt wird.

Dazu sei k ein K-Akteur im Zustand R oder W. k ist Auftragnehmer für die Kommunikationsoperationen, die er anbietet. Andere Akteure – die Auftraggeber – können Aufträge an k zur Ausführung von Kommunikationsoperationen erteilen, indem sie eine spezielle Anweisung, einen K-Order–Aufruf bzgl. eines K-Order–Generators von k, ausführen. Dabei wird eine entsprechende K-Order erzeugt und in einen Warteraum, der für jeden K-Order–Generator von k implizit definiert ist, eingeordnet. Der K-Akteur k nimmt erteilte Aufträge durch die Ausführung von K-Order–Annahmeanweisungen, die sich jeweils auf einen seiner K-Order–Generatoren beziehen, an. Dabei wird von k eine K-Order aus dem zugehörigen Warteraum entnommen und anschließend die kanonische Operation der K-Order sequentiell eingeordnet

in $op(k)$ ausgeführt. Ein Rendezvous zwischen zwischen einem Auftraggeber und dem Auftragnehmer k findet statt, wenn der Auftraggeber einen Auftrag erteilt hat und k diesen Auftrag annimmt. Die jeweilige dem Auftrag entsprechende K-Order mit ihren Eingabeparametern definiert die Operation, die k im Rendezvous ausführt, und die Ergebnisse, die der Auftraggeber nach Abschluß der Ausführung der K-Order als Ausgabeparameter erhält. Auftraggeber und Auftragnehmer synchronisieren sich somit zur Ausführung von Kommunikationsoperationen. Ein Auftraggeber wird nach Erteilung eines Auftrags in den Zustand W überführt und solange blockiert, bis die dem Auftrag entsprechende K-Order von dem jeweiligen Auftragnehmer vollständig ausgeführt ist. Ein Auftragnehmer wird bei Ausführung einer K-Order–Annahmeanweisung in den Zustand W überführt, wenn in dem entsprechendem Warteraum keine K-Order enthalten ist und somit kein entsprechender Auftrag vorliegt. Er wartet dann auf Erteilung eines Auftrags bzgl. des in der Annahmeanweisung genannten K-Order–Generators. Liegen bei Ausführung einer K-Order–Annahmeanweisung mehrere entsprechende Aufträge vor, so wird von dem Auftragnehmer der nach der FIFO–Strategie (*First In First Out*) erste davon ausgewählt. Nach Ausführung einer Kommunikationsoperation führen sowohl der Auftragnehmer als auch der Auftraggeber ihre jeweiligen Berechnungen parallel fort.

K-Akteure sind somit multioperationale, kommunikationsfähige aktive Komponenten, die die Berechnung ihrer kanonischen Operation und in diese eingeordnet durch andere Akteure aufgerufene Kommunikationsoperationen ausführen. K-Akteure können benannte oder anonyme Komponenten sein. Ein K-Akteur wird dementsprechend entweder durch Erarbeitung einer K-Akteur–Deklaration oder durch die Auswertung eines Generierungsausdrucks erzeugt.

Damit sind die verschiedenen DA–Inkarnationsarten, aus denen ein INSEL–System bestehen kann, erklärt. Im folgenden Abschnitt wird nun näher auf die entsprechenden DA–Generatoren eingegangen.

3.2.4 DA–Generatoren

Die Eigenschaften von DA–Inkarnationen werden durch entsprechende DA–Generatoren definiert. DA–Generatoren können parametrisiert werden. Die **Parametrisierung** erfolgt wie in anderen imperativen Programmiersprachen durch die Festlegung formaler Parameter mit ihrem jeweiligen Übergabemodus und Typ bzw. Generator. In INSEL stehen als Übergabemodi Eingabeparameter (*call by value*), Ein–Ausgabeparameter (*call by value–result*) und Ausgabeparameter (*call by result*) zur Verfügung. Die für einen DA–Generator zulässigen Parameterarten und Übergabemodi sind abhängig von der Art des Generators. So können für PS-Order–Generatoren (vergleichbar mit Prozedurdeklarationen) und M-Akteur–Generatoren sowohl Eingabe–, Ein–Ausgabe– als auch Ausgabeparameter festgelegt werden, während für FS-Order–Generatoren (vergleichbar mit Funktionsdeklarationen) sowie K-Akteur– und Depot–Generatoren lediglich Eingabeparameter zugelassen sind. BS-Order–Generatoren (vergleichbar mit Blockdeklarationen) sind nicht parametrisierbar. Bei der Erzeugung einer Inkarnation bzgl. eines parametrisierten DA–Generators werden den formalen Parametern entsprechende – in dem jeweiligen DA–Erzeugungskonstrukt[2] anzugebene – aktuelle Parameter assoziiert.

Die programmiersprachliche Definition eines DA–Generators besteht im allgemeinen aus zwei Teilen: einem Spezifikations– und einem Implementierungsteil. Der **Spezifikationsteil** ist op-

[2] Als DA–Erzeugungskonstrukt werden im weiteren die Sprachkonstrukte von INSEL bezeichnet, deren Erarbeitung bzw. Ausführung die Erzeugung einer neuen DA–Inkarnation bewirkt. Hierzu gehören S-Order–, K-Order– und M-Akteur–Aufrufe, K-Akteur– und Depot–Deklarationen sowie K-Akteur– und Depot–Generierungsausdrücke.

tional und nur für die Generatoren solcher DA–Inkarnationen verpflichtend, die explizit definierte äußere Operationen anbieten, also für K-Akteur– und Depot–Generatoren. Der Spezifikationsteil besteht neben einer optionalen Liste von formalen Parametern im wesentlichen aus einer Menge von Deklarationen, die die Schnittstelle der entsprechenden DA–Inkarnationen nach außen definieren. Genau die Komponenten, deren Deklaration in dem Spezifikationsteil enthalten ist, sind die mit dem Attribut E definierten lokalen N–Komponenten der entsprechenden DA–Inkarnationen. Dementsprechend sind in dem Spezifikationsteil eines K-Order–Generators lediglich K-Order–Generatoren erlaubt, während in dem Spezifikationsteil eines Depot–Generators alle Arten von Deklarationen enthalten sein dürfen.

Der **Implementierungsteil** eines DA–Generators, der für alle Arten von DA–Generatoren angegeben werden muß, besteht neben einer ggf. vorhandenen Liste von formalen Parametern[3] aus einem Deklarations– und einem Anweisungsteil. Der Deklarationsteil enthält eine Folge von Deklarationen, mit der die Menge der lokalen nicht nach außen exportierten N–Komponenten der entsprechenden DA–Inkarnationen festgelegt wird.

Gemäß dem Schachtelungsprinzip wird ein DA–Generator – bis auf eine Ausnahme – selbst wiederum im Deklarationsteil eines anderes DA–Generators definiert, wobei es konzeptionell keine Beschränkungen hinsichtlich der Schachtelungsmöglichkeiten gibt. So kann zum Beispiel ein M-Akteur–Generator in dem Deklarationsteil eines Depot–Generators deklariert werden und umgekehrt. Bei der erwähnten Ausnahme handelt es sich um den „äußersten“ DA–Generator eines INSEL–Programms, zu dem alle anderen Generatoren innen sind. Dieser DA–Generator ist ein spezieller M-Akteur–Generator, bzgl. dem keine Inkarnationen explizit durch M-Akteur–Aufrufe erzeugt werden können. Er definiert die Hauptkomponente des entsprechenden INSEL–Systems. Die Hauptkomponente wird zu Beginn der Ausführung des INSEL–Programms als M-Akteur erzeugt und gestartet.

Ist für einen DA–Generator ein Spezifikationsteil angegeben, so muß der zu ihm gehörende Implementierungsteil im selben Deklarationsteil folgen. Durch die Aufteilung von DA–Generatoren in einen Spezifikations– und einen Implementierungsteil ist es möglich, wechselseitige Abhängigkeiten zwischen DA–Generatoren zu spezifizieren. In INSEL ist es jedoch nicht möglich, wie zum Beispiel in der CORBA–Architektur ([OMG91]), zu einer durch einen Spezifikationsteil definierten Schnittstelle unterschiedliche schnittstellenkonforme Implementierungen anzugeben.

Bei der programmiersprachlichen Definition eines DA–Generators kann gemäß der für INSEL festgelegten Sichtbarkeitsregeln grundsätzlich auf alle Komponenten zugegriffen werden, die textuell gemäß der Schachtelungsstruktur vor diesem Generator deklariert worden sind. INSEL bietet mit dem Konzept der Import– und Exportbeschränkungen jedoch die Möglichkeit, die Sichtbarkeit von Komponenten weiter einzuschränken. Mit den Import–Beschränkungen können Zugriffsmöglichkeiten auf die Komponenten eingeschränkt werden, die gemäß der Schachtelungsstruktur außen liegen. Export–Beschränkungen ermöglichen es, die Nutzungen der in einem DA–Generator lokal deklarierten Komponenten in den DA–Generatoren, die gemäß der Schachtelungsstruktur innen liegen, einzuschränken. Auf die Import– und Exportbeschränkungen wird in Abschnitt 3.5 noch genauer eingegangen.

3.2.5 Beispiel

Anhand des in Abbildung 3.2 angegebenen INSEL–Programms werden die in den vorhergehenden Abschnitten eingeführten Konzepte erläutert. Das INSEL–Programm implementiert

[3] Ist für den DA–Generator auch ein Spezifikationsteil festgelegt, so müssen die Parameterlisten des Spezifikations– und des Implementierungsteils übereinstimmen.

```
PROCESS ErzeugerVerbraucherSystem IS

  -- Spezifikationsteil des Depot-Generators PufferTyp
  DEPOT TYPE SPEC PufferTyp (MaxAnzahl : IN integer) IS
    PROCEDURE TYPE SPEC Einfuegen (Wert : IN  integer);  -- Zugriffsoperation
    PROCEDURE TYPE SPEC Entnehmen (Wert : OUT integer);  -- Zugriffsoperation
  END PufferTyp;
  -- Implementierungsteil des Depot-Generators PufferTyp
  DEPOT TYPE PufferTyp (MaxAnzahl : IN integer) IS
    TYPE IndexTyp IS 0 .. MaxAnzahl-1;
    TYPE FeldTyp  IS ARRAY [IndexTyp] OF integer;
    Feld         : FeldTyp;
    Anfang, Ende : IndexTyp;

    PROCEDURE TYPE Einfuegen (Wert : IN  integer) IS
    BEGIN
      Feld[Ende] := Wert; Ende := (Ende + 1) MOD MaxAnzahl;
    END Einfuegen;

    PROCEDURE TYPE Entnehmen (Wert : OUT integer) IS
    BEGIN
      Wert := Feld[Anfang]; Anfang := (Anfang + 1) MOD MaxAnzahl;
    END Entnehmen;
  BEGIN Anfang := 0; Ende := 0; END PufferTyp;

  -- Deklaration eines benannten Puffers
  DEPOT Puffer : PufferTyp (100);

  -- M-Akteur-Generator Erzeuger
  PROCESS TYPE Erzeuger (MaxErzeuge: IN integer; Start, Diff: IN integer) IS
    Wert : integer := Start;
  BEGIN
    FOR i IN 1..MaxErzeuge LOOP
      Puffer.Einfuegen(Wert); Wert := Wert + Diff;
    END LOOP;
  END Erzeuger;
  -- M-Akteur-Generator Verbraucher
  PROCESS TYPE Verbraucher (MaxVerbrauche: IN integer) IS
    Wert : integer;
  BEGIN
    FOR i IN 1..MaxVerbrauche LOOP
      Puffer.Entnehmen(Wert);
    END LOOP;
  END Verbraucher;
-- Anweisungsteil der Hauptkomponente
BEGIN
  FORK Erzeuger(4,1,1); FORK Verbraucher(4);
END ErzeugerVerbraucherSystem;
```

Abbildung 3.2: INSEL–Implementierung eines Erzeuger–Verbraucher–Systems

ein einfaches Erzeuger–Verbraucher–System, in dem ein Erzeuger `integer`–Werte erzeugt und diese in einen Puffer beschränkter Kapazität als Zwischenspeicher einfügt, aus dem sie dann von einem Verbraucher wieder entnommen werden.

Der Puffer wird in dem INSEL–Programm als benanntes Depot realisiert. Hierfür wird zunächst der Depot–Generator `PufferTyp` definiert. In dem Spezifikationsteil des Depot–Generators `PufferTyp` werden die beiden PS-Order–Generatoren `Einfuegen` und `Entnehmen` als Zugriffsoperationen der entsprechenden Depots festgelegt. Der lokale Speicher eines Puffer–Depots wird im Implementierungsteil von `PufferTyp` durch die Deklaration des Arrays `Feld`, für das vorher ein entsprechender Array-Generator (`Feldtyp`) definiert wird, implementiert. Die Größe dieses Arrays wird bei der Erzeugung einer Inkarnation bzgl. `PufferTyp` durch den Wert des Eingabeparameters `MaxAnzahl` festgelegt. Die Ausführung der Zugriffsoperation `Einfuegen` auf einem Puffer–Depot bewirkt, daß der als Eingabeparameter `Wert` übergebene `integer`–Wert in das lokale Array an die durch die DE–Inkarnation `Ende` festgelegte Stelle eingefügt wird. Die Ausführung von `Entnehmen` bewirkt, daß der Wert des durch die DE–Inkarnation `Anfang` festgelegten Array–Elements als Ausgabeparameter `Wert` an den Aufrufer zurückgegeben wird. Die durch den Anweisungsteil des Depot–Generators `PufferTyp` definierte Initialisierungsphase eines Puffer–Depots besteht in der Initialisierung der beiden `integer` DE–Inkarnationen `Anfang` und `Ende` mit dem Wert 0. Der Puffer des Erzeuger–Verbraucher–Systems wird durch eine Depot–Deklaration als benanntes Depot `Puffer` mit einer Kapazität von maximal 100 zu speichernden `integer`–Werten erzeugt.

Der Erzeuger und der Verbraucher werden jeweils als M-Akteur realisiert. Dazu werden die beiden parametrisierten M-Akteur–Generatoren `Erzeuger` und `Verbraucher` definiert. Der Eingabeparameter `MaxErzeuge` des M-Akteur–Generators `Erzeuger` legt die Anzahl der von einer entsprechenden Inkarnation erzeugten `integer`–Werte fest. Die `integer`–Werte werden dabei auf Basis der Eingabeparameter `Start` und `Diff` erzeugt. Der Eingabeparameter `MaxVerbrauche` von `Verbraucher` legt die Anzahl der von einem Verbraucher aus dem Puffer zu entnehmenden Werte fest. In dem M-Akteur–Generator `Erzeuger` wird die Zugriffsoperation `Einfuegen` auf dem Puffer durch den PS-Order–Aufruf `Puffer.Einfuegen` aufgerufen. Analog enthält der M-Akteur–Generator `Verbraucher` den PS-Order–Aufruf `Puffer.Entnehmen` zum Entnehmen eines Wertes aus dem Puffer. Die zwei Akteure des Erzeuger–Verbraucher–Systems werden durch die beiden in dem Anweisungsteil der Hauptkomponente angegebenen M-Akteur–Aufrufe erzeugt.

3.3 Entwicklungsmöglichkeiten und Systemstrukturen

Die Entwicklungsmöglichkeiten eines INSEL–Systems ergeben sich im wesentlichen aus der dynamischen Erzeugung und Auflösung von Komponenten. In diesem Abschnitt wird deshalb zunächst näher auf die Erzeugung und Auflösung von Komponenten eingegangen. Anschließend werden die Strukturrelationen, mit denen die verschiedenen konzeptionell festgelegten Abhängigkeiten zwischen den Komponenten eines INSEL–Systems beschrieben werden, und die sich daraus ergebenden Systemstrukturen eingeführt.

3.3.1 Erzeugung und Auflösung von Komponenten

Wie bereits erklärt, hat die strikte Anwendung des Schachtelungsprinzips zur Folge, daß ein INSEL–System zu Beginn seiner Existenz aus genau einer Komponente, der Hauptkomponente des Systems, besteht. Das System entwickelt sich dynamisch ausgehend von der Hauptkomponente, indem bei Ausführung der kanonischen Operation dieses speziellen M-Akteurs

Komponenten erzeugt und aufgelöst werden. Hierzu können insbesondere DA–Inkarnationen gehören, in deren kanonischen Operationen wiederum beliebige Komponenten erzeugt und aufgelöst werden können. Die Möglichkeiten zur Erzeugung der einzelnen Arten von Komponenten wurden bereits in dem vorhergehenden Abschnitt erläutert. Wesentlich für INSEL ist, daß die Auflösung von Komponenten nicht explizit durch Angabe entsprechender Sprachkonstrukte, sondern gemäß konzeptionell festgelegter Regeln erfolgt. Für jede Komponente ist bereits zum Zeitpunkt ihrer Erzeugung implizit festgelegt, wann bzw. mit welcher anderen Komponente sie aufgelöst wird. Voraussetzung für die Auflösung einer DA–Inkarnation ist, daß diese sich im Zustand T befindet. Die Bedingungen für die Überführung einer DA-Inkarnation in den Zustand T sind durch die Terminierungsregel von INSEL gegeben.

(3.2) Terminierungsregel

Eine DA–Inkarnation x wird genau dann in den Zustand T überführt, wenn

- sich alle Akteure, die parallel zu x eingebettet ausgeführt werden, im Zustand T befinden und
- der Abschluß der Ausführung der kanonischen Operation $op(x)$ von x erreicht ist.

Für die Auflösung der einzelnen Arten von INSEL–Komponenten gelten folgende Regeln.

(3.3) Auflösungsregeln

Sei k eine Komponente eines INSEL–Systems. Falls k DA–Inkarnation ist, befinde sie sich im Zustand T.

- Falls k benannte Komponente, d.h. Generator, benannte DE–Inkarnation, benanntes Depot oder benannter K-Akteur ist, wird k zusammen mit der DA–Inkarnation x aufgelöst, die k als lokale N–Komponente enthält, für die also gilt: $k \in L_0(x)$.
- Falls k Order ist, wird k mit ihrer Überführung in den Zustand T aufgelöst[4].
- Falls k M-Akteur ist, wird k mit Überführung der DA–Inkarnation in den Zustand T, in die er parallel eingeordnet ausgeführt wird, aufgelöst. Für die Hauptkomponente, die gemäß Festlegung ein M-Akteur ist, kann diese Regel nicht angewendet werden. Sie wird mit ihrer Überführung in den Zustand T aufgelöst.
- Falls k anonyme Komponente ist, wird k zusammen mit der DA–Inkarnation aufgelöst, die den Zeiger–Generator für Zeiger, die auf k zeigen können und die somit mit dem Generator von k qualifiziert sind, als lokale N–Komponente enthält.

Mit den angegebenen Auflösungsregeln wird konzeptionell gewährleistet, daß eine Komponente eines INSEL–Systems solange existiert, wie es noch potentielle Nutzer dieser Komponente geben kann. Dies gilt inbesondere auch für die anonymen Komponenten, deren Verwaltung in anderen objekt–basierten Systemen in der Regel aufwendig ist, da die auf sie verweisenden Zeiger zum Beispiel kopiert oder als Parameter übergeben werden können. Durch die INSEL–Auflösungsregel für anonyme Komponenten ist konzeptionell sichergestellt, daß nach Auflösung einer anonymen Komponente k keine Zeiger mehr existieren, die auf k zeigen und noch genutzt werden können. Dies gilt, da für die Auflösung von k die DA–Inkarnation ausschlaggebend ist, die den Zeiger–Generator für Zeiger, die auf k verweisen können, als lokale

[4] Ist die Order k mit Ein–Ausgabe– bzw. Ausgabeparametern parametrisiert, erfolgt die entsprechende Parameterübergabe an den Aufrufer mit der Terminierung von k unmittelbar vor Auflösung von k.

N–Komponente enthält. Mit der Auflösung von k wird somit ebenfalls der entsprechende Zeiger–Generator aufgelöst.

Die konzeptionell festgelegten Auflösungsregeln ermöglichen eine effiziente und automatisierte Speicherverwaltung. In INSEL ist es weder erforderlich, den für die Repräsentation von Komponenten benötigten Speicherbereich explizit zu allokieren noch diesen nach deren Auflösung wieder freizugeben. Dementsprechend gibt es in INSEL auch keine Sprachkonzepte zur Allokation bzw. Freigabe von Speicher, wie zum Beispiel in C oder C++. Dies entspricht der INSEL zugrundeliegenden Philosophie, dem Anwendungsentwickler Konstruktionskonzepte auf einem hohem Abstraktionsniveau zur Verfügung zu stellen.

Gemäß dem Erklärten besteht ein INSEL–System $\mathcal{S}$ zu jedem Zeitpunkt t seiner Existenz aus einer Menge von Komponenten. Am Beginn seiner Existenz zum Zeitpunkt $t = 0$ besteht das System $\mathcal{S}$ allein aus der Hauptkomponente. Diese wird im weiteren mit $\underline{a}$ bezeichnet. Vom Zeitpunkt $t = 0$ an führt der M-Akteur $\underline{a}$ seine kanonische Operation aus. Dabei können alle Arten von Komponenten erzeugt und aufgelöst werden. Es wird postuliert, daß nur endlich viele Komponenten erzeugt werden und daß der M-Akteur $\underline{a}$ nach endlicher Zeit in den Zustand T überführt und damit aufgelöst wird. Ist t_e dieser Zeitpunkt, so ist das Zeitintervall $\Lambda(\mathcal{S}) \triangleq [0, t_e]$ die **Lebenszeit des Systems $\mathcal{S}$**.

Im folgenden werden einige Bezeichnungen eingeführt, die im weiteren dieser Arbeit verwendet werden.

(3.4) Bezeichnungen

Sei $\mathcal{S}$ ein INSEL–System zum Zeitpunkt $t \in \Lambda(\mathcal{S})$. Dann bezeichnet:

- X_t die Menge der Komponenten von $\mathcal{S}$ zum Zeitpunkt t;
- $X_t^G \subset X_t$ die Menge der Generatoren von $\mathcal{S}$ zum Zeitpunkt t;
- $X_t^{DEG} \subset X_t^G$ die Menge der DE–Generatoren von $\mathcal{S}$ zum Zeitpunkt t;
- $X_t^{DAG} \subset X_t^G$ die Menge der DA–Generatoren von $\mathcal{S}$ zum Zeitpunkt t;
- $X_t^I \subseteq X_t$ die Menge der Inkarnationen von $\mathcal{S}$ zum Zeitpunkt t;
- $X_t^{DEI} \subseteq X_t^I$ die Menge der DE–Inkarnationen von $\mathcal{S}$ zum Zeitpunkt t;
- $X_t^{DAI} \subseteq X_t^I$ die Menge der DA–Inkarnationen von $\mathcal{S}$ zum Zeitpunkt t;
- $A_t \subseteq X_t^{DAI}$ die Menge der Akteure von $\mathcal{S}$ zum Zeitpunkt t;
- $SO_t \subset X_t^{DAI}$ die Menge der S-Order von $\mathcal{S}$ zum Zeitpunkt t;
- $KO_t \subset X_t^{DAI}$ die Menge der K-Order von $\mathcal{S}$ zum Zeitpunkt t;
- $D_t \subset X_t^{DAI}$ die Menge der Depots von $\mathcal{S}$ zum Zeitpunkt t.

Ist $x \in X_t^I$ eine Inkarnation, so bezeichnet $gen(x) \in X_t^G$ den Generator, bzgl. dem x erzeugt wurde.

Ist $x \in X_t^{DAI}$ eine DA–Inkarnation, so bezeichnet $\mu_t(x)$ den Zustand von x zum Zeitpunkt t.

Ist $a \in A_t$ ein Akteur, so bezeichnet $\varphi_t(a)$ die Ausführungskomponente von a zum Zeitpunkt t.

Ist k eine benannte Komponente, so bezeichnet $name(k)$ den Namen von k.

Ist $x \in X_t$ eine Komponente und $t_1 \in \Lambda(\mathcal{S})$ mit $t_1 < t$ der Zeitpunkt, zu dem x erzeugt wurde und $t_2 \in \Lambda(\mathcal{S})$ mit $t_2 > t$ der Zeitpunkt, zu dem x aufgelöst wird, so bezeichnet $\Lambda(x) \triangleq [t_1, t_2]$ die Lebenszeit von x.

Für jede DA–Inkarnation $x \in X_t^{DAI}$ eines INSEL–Systems wird in der folgenden Definition das Attribut *creator(x)* definiert, das den Akteur angibt, der die Komponente x bei Ausführung seiner kanonischen Operation erzeugt hat.

Definition 3.5.: Erzeugender Akteur einer DA–Inkarnation

Sei $a \in A_t$ ein Akteur. a führe zum Zeitpunkt t ein DA–Erzeugungskonstrukt $\mathcal{E}$ aus, dessen Ausführung bzw. Erarbeitung die Erzeugung einer DA–Inkarnation bewirkt[5]. x sei die durch Ausführung bzw. Erarbeitung von $\mathcal{E}$ erzeugte DA–Inkarnation. Dann gilt:

$$creator(x) \triangleq a$$

Die einzige DA–Inkarnation eines INSEL–Systems, die nicht durch einen anderen Akteur erzeugt wird, ist die Hauptkomponente $\underline{a}$ des Systems. Für sie wird festgelegt: $creator(\underline{a}) \triangleq \underline{a}$

□

3.3.2 Systemstrukturen

Die Menge der in einem INSEL–System zu einem Zeitpunkt existierenden DA–Inkarnationen ist nach strukturellen Abhängigkeiten geordnet. Diese Abhängigkeiten werden implizit bei der Konstruktion des Systems durch Anwendung der Komponenten–Konzepte und der grundlegenden Konstruktionsprinzipien, insbesondere des Schachtelungsprinzips, festgelegt. Die verschiedenen Abhängigkeiten werden durch Strukturrelationen beschrieben, die im folgenden erklärt werden. Dazu sei $\mathcal{S}$ ein INSEL–System zum Zeitpunkt $t \in \Lambda(\mathcal{S})$.

Definitionsstruktur

Die Definitionsstruktur beschreibt auf der Grundlage der Zusammenhänge zwischen Generatoren und Inkarnationen die definitorischen Abhängigkeiten zwischen den DA–Inkarnationen eines Systems. Auf diesen definitorischen Abhängigkeiten basiert die Beschränkung der Sichtbarkeit und damit der potentiellen Nutzbarkeit von Komponenten. Die Definitionsstruktur ist damit Basis für die Bestimmung der Ausführungsumgebung von DA–Inkarnationen. Hierauf wird in Abschnitt 3.5 noch detailliert eingegangen.

Definition 3.6.: Definitionsstruktur

Gegeben seien zwei DA–Inkarnationen $x, y \in X_t^{DAI}$. x ist **definitorisch abhängig** von y – in Zeichen $(x, y) \in \delta_t$ – genau dann, wenn der Generator $gen(x)$, bzgl. dem x erzeugt wurde, lokale N–Komponente von y ist, d.h. wenn gilt: $gen(x) \in L_0(y)$.

Das Paar (X_t^{DAI}, δ_t) ist die **Definitionsstruktur** des Systems $\mathcal{S}$ zum Zeitpunkt t.

□

Die Relation δ_t ist gemäß Definition (3.6) eine zweistellige Relation auf der Menge der zum Zeitpunkt t existierenden DA–Inkarnationen. Sie setzt eine DA–Inkarnation mit derjenigen in Beziehung, die ihren Generator als lokale N–Komponente enthält. Jede DA–Inkarnation ist für ihre gesamte Lebenszeit in die Definitionsstruktur eingeordnet, d.h. sie wird mit ihrer Erzeugung in δ eingeordnet und mit ihrer Auflösung aus δ ausgeordnet. Da es für jede DA–Inkarnation $x \in X_t^{DAI}$, mit Ausnahme der Hautpkomponente $\underline{a}$, genau eine DA–Inkarnation

[5] Vergleiche Fußnote auf Seite 44.

$y \in X_t^{DAI}$ mit $(x, y) \in \delta_t$ gibt, folgt, daß die Definitionsstruktur (X_t^{DAI}, δ_t) ein Baum mit der Hauptkomponente $\underline{a}$ als Wurzel ist. Dabei ist die Menge X_t^{DAI} die Knoten– und die Relation δ_t die Kantenmenge des entsprechenden Graphen.

Sind x und y zwei DA–Inkarnationen und gibt es eine Folge von DA–Inkarnationen $x_0, \ldots, x_n$ mit $n \geq 1$ und $x_0 = x$ und $x_n = y$, so daß $(x_{i-1}, x_i) \in \delta_t$ für alle $i \in \{1, \ldots, n\}$ gilt, dann wird im weiteren davon gesprochen, daß x **δ–innen** zu y und y **δ–außen** zu x eingeordnet ist.

Beispiel
In Abbildung 3.3 ist auf der linken Seite in einem Schnappschuß die Definitionsstruktur des INSEL–Systems dargestellt, das durch das in Abschnitt 3.2.5 angegebene INSEL–Programm (siehe Abbildung 3.2) festgelegt ist. Der dargestellte Schnappschuß zeigt die Definitionsstruktur des Erzeuger–Verbraucher–Systems zu einem Zeitpunkt t, in dem die Erarbeitung des Deklarationsteils der Hauptkomponente bereits abgeschlossen ist und die beiden M-Akteur–Aufrufe zur Erzeugung des Erzeugers und des Verbrauchers bereits ausgeführt wurden. Zu dem betrachteten Zeitpunkt führt der Erzeuger gerade die Zugriffsoperation `Einfuegen` auf dem Depot `Puffer` aus. Die S-Order `Einfuegen` ist δ–innen zu dem Depot `Puffer` und zu der Hauptkomponente `ErzeugerVerbraucherSystem` eingeordnet.

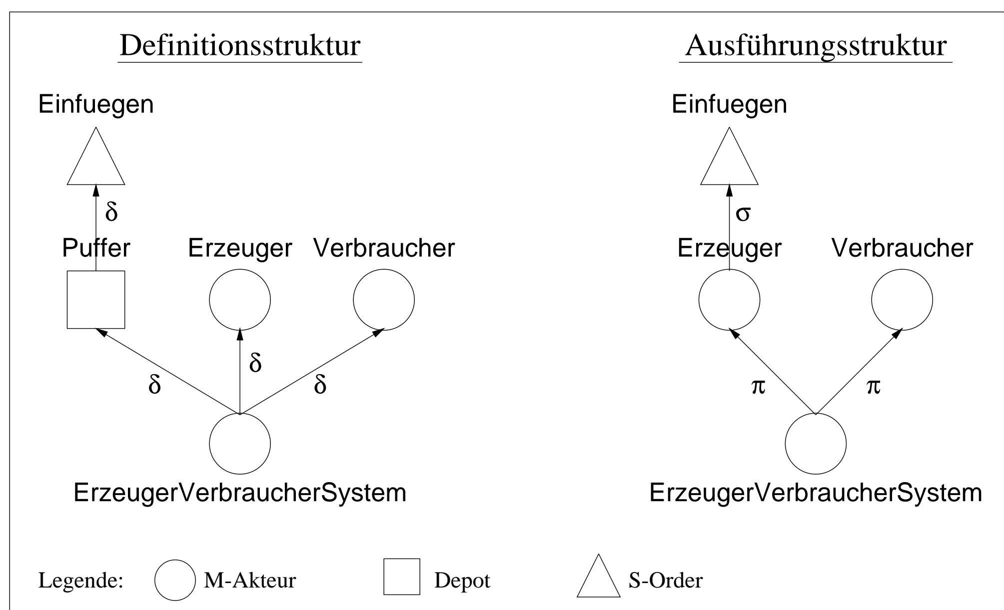

Abbildung 3.3: Definitions– und Ausführungsstruktur des Erzeuger–Verbraucher–Systems aus Abbildung 3.2 zu einem Zeitpunkt t

Ausführungsstruktur

Die Ausführungsstruktur eines INSEL–Systems beschreibt die Abhängigkeiten zwischen den Akteuren und damit den parallelen Kontrollflüssen des Systems sowie die Einbettung sequentiell ausgeführter Operationen in diese Kontrollflüsse. Sie erfaßt damit funktionale Abhängigkeiten zwischen den DA-Inkarnationen des Systems und setzt sich aus parallelen, sequentiellen sowie kooperativen Anteilen zusammen. Letztere ergeben sich durch die Ausführung von Kommunikationsoperationen.

Definition 3.7.: Ausführungsstruktur

Gegeben seien zwei DA–Inkarnationen $x, y \in X_t^{DAI}$.

- Sequentielle Einordnung
  x sei Order oder Depot im Zustand A. x ist **sequentiell** bzw. **σ–innen** zu y eingeordnet – in Zeichen $(x, y) \in \sigma_t$ – genau dann, wenn x S-Order oder Depot ist und bei Ausführung der kanonischen Operation $op(y)$ erzeugt worden ist oder x K-Order und y der K-Akteur mit $gen(x) \in L_0(y)$ ist und y $op(x)$ ausführt.
- Parallele Einordnung
  x sei Akteur. x ist **parallel** bzw. **π–innen** zu y eingeordnet – in Zeichen $(x, y) \in \pi_t$ – genau dann, wenn x bei Ausführung der kanonischen Operation $op(y)$ erzeugt worden ist.[6]
- Kooperationsabhängigkeit
  x sei K-Order. y **kooperiert** über x – in Zeichen $(x, y) \in \kappa_t$ – genau dann, wenn x bei Ausführung der kanonischen Operation $op(y)$ erzeugt worden ist.

Mit den Relationen σ_t, π_t und κ_t wird die **Ausführungsrelation** α_t wie folgt definiert:

$$(x, y) \in \alpha_t \overset{\triangle}{\Longleftrightarrow} (x, y) \in \sigma_t \ \vee \ (x, y) \in \pi_t \ \vee \ (x, y) \in \kappa_t$$

Ist $(x, y) \in \alpha_t$, so ist x **α–innen** zu y eingeordnet. Das Paar (X_t^{DAI}, α_t) ist die **Ausführungsstruktur** des Systems $\mathcal{S}$ zum Zeitpunkt t.

□

Die kanonischen Operationen von S-Order und Depots werden gemäß Definition (3.7) sequentiell eingebettet in die kanonische Operation der erzeugenden DA-Inkarnation ausgeführt. S-Order sind für ihre gesamte Lebenszeit in die Ausführungsstruktur eingeordnet, ein Depot lediglich für die Dauer seiner Initialisierungsphase. Eine K-Order wird spezifisch in die Ausführungsstruktur eingeordnet. Neben der sequentiellen Einordnung für die Dauer der Ausführung ihrer kanonischen Operation, die die Abhängigkeit zwischen der K-Order und ihrem jeweiligen K-Akteur als Auftragnehmer beschreibt, ist sie für ihre gesamte Lebenszeit in die Relation κ eingeordnet. Die Relation κ erfaßt den Zusammenhang zwischen einer K-Order und ihrem Auftraggeber. Ein Rendezvous zwischen einem Auftraggeber und einem Auftragnehmer ist daran erkennbar, daß die entsprechende K-Order sowohl in κ als auch in σ eingeordnet ist. Mit der Erzeugung eines Akteurs wird ein neuer Kontrollfluß etabliert, der parallel zum Kontrollfluß der erzeugenden DA–Inkarnation ausgeführt wird. M-Akteure und anonyme K-Akteure sind für ihre gesamte Lebenszeit in die Ausführungsstruktur eingeordnet. Da ein benannter K-Akteur erst zu Beginn der Ausführung des Anweisungsteils der DA–Inkarnation, in deren Deklarationsteil er deklariert wurde, gestartet wird und somit erst zu diesem Zeitpunkt mit seiner Berechnung beginnt, wird er auch erst zu diesem Zeitpunkt in die Ausführungsstruktur eingeordnet.

Aus den Regeln für die Einordnung von DA–Inkarnationen in die Ausführungsstruktur ergibt sich, daß nicht notwendig alle Elemente aus X_t^{DAI} in die Ausführungsstruktur eingeordnet sind. Zudem ist (X_t^{DAI}, α_t) im allgemeinen kein Baum, da K-Order im Zustand A in die Relation κ und in die Relation σ eingeordnet sind.

Beispiel (Fortsetzung)
Die rechte Seite der Abbildung 3.3 zeigt die Ausführungsstruktur des Erzeuger–Verbraucher–Systems zu dem im vorhergehenden Beispiel betrachteten Zeitpunkt t. Die M-Akteure

[6] Für die parallele Einordnung von K-Akteuren gelten besondere Regeln, die für diese Arbeit jedoch nicht weiter von Interesse sind. Siehe hierzu [SEL+96].

`Erzeuger` und `Verbraucher` sind jeweils parallel zu der sie erzeugenden Hauptkomponente `ErzeugerVerbraucherSystem` eingeordnet. Der Erzeuger führt gerade sequentiell eingebettet in seine kanonische Operation die S-Order `Einfuegen` als Zugriffsoperation auf dem Depot `Puffer` aus. Das benannte Depot `Puffer` ist nicht mehr in die Ausführungsstruktur eingeordnet, da es sich zu dem betrachteten Zeitpunkt t bereits im Zustand T befindet.

Lokalitätsstruktur

Benannte DA–Inkarnationen sind in einem INSEL–System immer lokal in eine andere DA–Inkarnation eingeordnet. Sie werden durch Erarbeitung einer entsprechenden Deklaration im Rahmen der Ausführung der kanonischen Operation einer DA–Inkarnation als lokale N–Komponente dieser Inkarnation erzeugt. In INSEL können benannte Depots und benannte K-Akteure im Deklarationteil aller Arten von DA–Inkarnationen deklariert werden. Die Lokalitätsstruktur beschreibt die Einordnung der benannten Depots und K-Akteure als lokale N–Komponenten in die DA–Inkarnationen des Systems.

Definition 3.8.: Lokalitätsstruktur

Gegeben seien zwei DA–Inkarnationen $x, y \in X_t^{DAI}$. x sei benanntes Depot im Zustand T oder benannter K-Akteur. x ist **lokal** in y eingeordnet – in Zeichen $(x, y) \in \lambda_t$ – genau dann, wenn x bei Ausführung der kanonischen Operation $op(y)$ erzeugt worden ist und x lokale N–Komponente von y ist, d.h. wenn gilt: $x \in L_0(y)$.

Das Paar (X_t^{DAI}, λ_t) ist die **Lokalitätsstruktur** des Systems $\mathcal{S}$ zum Zeitpunkt t.

□

Ein benannter K-Akteur ist während seiner gesamten Lebenszeit in der Lokalitätsstruktur erfaßt. Depots sind lediglich für den Teil ihrer Lebenszeit, in dem sie sich im Zustand T befinden, in die Lokalitätsstruktur eingeordnet. Dies entspricht der Tatsache, daß ein Depot erst nach Abschluß seiner Initialisierungsphase und damit der Überführung in den Zustand T von außen durch Aufruf seiner Zugriffsoperationen nutzbar ist.

Beispiel (Fortsetzung)
In dem Erzeuger–Verbraucher–System existiert zu dem betrachteten Zeitpunkt t lediglich eine benannte DA–Inkarnation. Dies ist das benannte Depot `Puffer`, das lokal in die Hauptkomponente eingeordnet ist, d.h. es gilt: $(\texttt{Puffer}, \texttt{ErzeugerVerbraucherSystem}) \in \lambda_t$.

Zeigerstruktur

Die Zeigerstruktur eines INSEL–Systems erfaßt die Nutzungsmöglichkeiten anonymer Depots und anonymer K-Akteure über Zeiger.

Definition 3.9.: Zeigerstruktur

Gegeben seien zwei DA–Inkarnationen $x, y \in X_t^{DAI}$. x sei ein anonymes Depot im Zustand T oder ein anonymer K-Akteur. x ist von y über einen **Zeiger nutzbar** – in Zeichen $(x, y) \in \zeta_t$ – genau dann, wenn es mindestens einen Zeiger z gibt, der auf x zeigt und der lokale N–Komponente von y ist, d.h. wenn gilt : $z \in L_0(y)$.

Das Paar (X_t^{DAI}, ζ_t) ist die **Zeigerstruktur** des Systems $\mathcal{S}$ zum Zeitpunkt t.

□

Die Relation ζ_t setzt eine anonyme DA–Inkarnation x mit den DA–Inkarnationen in Beziehung, die einen Zeiger als lokale N–Komponente besitzen, der auf x verweist. Jede Wertzuweisung an einen Zeiger, der mit einem Depot– oder einem K-Akteur–Generator qualifiziert ist, kann die Zeigerstruktur des Systems verändern. Da Generierungsausdrücke zur Erzeugung anonymer Inkarnationen stets als Teil eines Konstrukts mit Wertzuweisung an einen Zeiger auftreten, wird ein anonymer K-Akteur mit seiner Erzeugung in die Zeigerstruktur eingeordnet. Ein anonymes Depot wird demgegenüber erst mit Abschluß seiner Initialisierungsphase und Überführung in den Zustand T in die Zeigerstruktur eingeordnet.

Lebenszeitstruktur

Die Lebenszeitstruktur beschreibt auf Basis der für INSEL festgelegten Auflösungsregeln die Lebenszeitabhängigkeiten zwischen den DA–Inkarnationen eines INSEL–Systems. Die Definition der Lebenszeitrelation ε_t stützt sich auf einige der bisher eingeführten Relationen und auf die Abbildung γ_t, die zu einer anonymen DA–Inkarnation x die DA–Inkarnation angibt, die den Zeiger–Generator für Zeiger, die auf x zeigen können, als lokale N–Komponente enthält.

Definition 3.10.: Lebenszeitstruktur

Gegeben seien zwei DA–Inkarnationen $x, y \in X_t^{DAI}$. x ist **lebenszeitabhängig** von y – in Zeichen $(x, y) \in \varepsilon_t$ – genau dann, wenn

- $(x, y) \in \sigma_t$ falls x S-Order oder benanntes Depot im Zustand A ist;
- $(x, y) \in \pi_t$ falls x M-Akteur ist;
- $(x, y) \in \lambda_t$ falls x benanntes Depot im Zustand T oder benannter K-Akteur ist;
- $y = \gamma_t(x)$ falls x anonymes Depot oder anonymer K-Akteur ist;
- $(x, y) \in \kappa_t$ falls x K-Order ist.

Das Paar $(X_t^{DAI}, \varepsilon_t)$ ist die **Lebenszeitstruktur** des Systems $\mathcal{S}$ zum Zeitpunkt t.

□

Die Relation ε_t ist so definiert, daß sie eine DA–Inkarnation zu der DA–Inkarnation in Beziehung setzt, mit deren Terminierung bzw. Auflösung sie gemäß der INSEL–Auflösungsregeln aufgelöst wird. Dies gilt jedoch nicht für Order, da diese mit ihrer Überführung in den Zustand T und damit unabhängig von einer andere DA–Inkarnation aufgelöst werden. Ihre Einordnung in die Lebenszeitstruktur dient lediglich der Vollständigkeit und entspricht ihrer σ– bzw. κ–Einordnung. Aus Definition (3.10) folgt, daß jede DA–Inkarnation für ihre gesamte Lebenszeit in die Lebenszeitstruktur eingeordnet ist. Graphisch dargestellt ist die Lebenszeitstruktur $(X_t^{DAI}, \varepsilon_t)$ ein Baum mit der Hauptkomponente als Wurzel.

Beispiel (Fortsetzung)
Die Abbildung 3.4 zeigt die Lebenszeitstruktur des Erzeuger–Verbraucher–Systems zu dem bereits in den vorhergehenden Beispielen betrachteten Zeitpunkt t. Die beiden M-Akteure `Erzeuger` und `Verbraucher` sowie das benannte Depot `Puffer` sind jeweils lebenszeitabhängig von der Hauptkomponente `ErzeugerVerbraucherSystem`. Mit Terminierung der beiden M-Akteure terminiert auch die Ausführung der kanonischen Operation der Hauptkomponente. Damit werden zunächst die beiden M-Akteure und anschließend das benannte Depot und schließlich die Hauptkomponente selbst aufgelöst, womit die Ausführung des entsprechenden INSEL–Programms beendet ist.

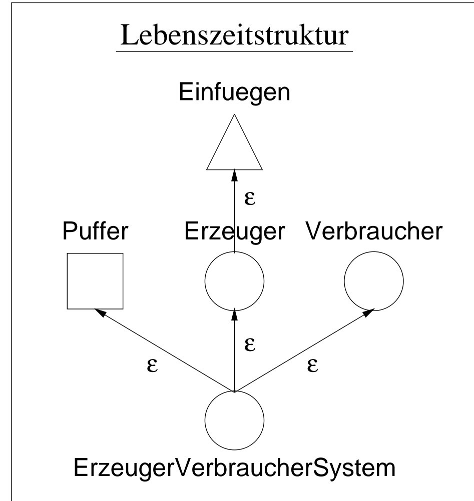

Abbildung 3.4: Lebenszeitstruktur des Erzeuger–Verbraucher–Systems aus Abbildung 3.2 zum Zeitpunkt t

Systemkonfigurationen

Die Eigenschaften, die ein INSEL–System zu einem Zeitpunkt seiner Existenz hat, können durch die Menge der existierenden DA–Inkarnationen und die mit den Strukturrelationen erfaßten Abhängigkeiten zwischen diesen beschrieben werden. Die sich damit ergebende Beschreibungsform für INSEL–Systeme wird Systemkonfiguration genannt.

Definition 3.11.: Systemkonfiguration

Sei $\mathcal{S}$ ein INSEL–System zum Zeitpunkt $t \in \Lambda(\mathcal{S})$. Die **Systemkonfiguration** $\mathcal{S}_t$ des Systems $\mathcal{S}$ zum Zeitpunkt t ist definiert durch:

$$\mathcal{S}_t = (X_t^{DAI}, \delta_t, \alpha_t, \lambda_t, \zeta_t, \varepsilon_t)$$

Die Systemkonfiguration $\mathcal{S}_0 = (\{\underline{a}\}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$ wird als **Anfangskonfiguration** des Systems $\mathcal{S}$ bezeichnet.

□

Die dynamische Entwicklung eines INSEL–Systems, die sich aus der Erzeugung und Auflösung von Inkarnationen ergibt, kann auf Basis der Definition (3.11) durch eine Folge von Systemkonfigurationen $(\mathcal{S}_t : t \in \Lambda(\mathcal{S}))$ beschrieben werden.

3.4 Ausführungsphasen einer kanonischen Operation

Die Ausführung der kanonischen Operation einer DA–Inkarnation x kann in verschiedene Phasen unterteilt werden (vgl. Abbildung 3.5). Nach Aufruf der implizit definierten Startoperation $i(x)$ auf x und damit der Anfangssynchronisation mit der x erzeugenden DA–Inkarnation beginnt die Ausführung der kanonischen Operation $op(x)$ von x. Die Ausführung von $op(x)$ untergliedert sich in die Aufbauphase und die Berechnungsphase.

In der **Aufbauphase** von $op(x)$ werden die lokalen N–Komponenten von x erarbeitet. Die Aufbauphase entspricht somit der Erarbeitung des Deklarationsteils von x. Sie beginnt mit der

Anfangssynchronisation und endet mit dem Abschluß der Erarbeitung des Deklarationsteils. In der Aufbauphase können neben Generatoren und DE–Inkarnationen insbesondere Depots und K-Akteure erzeugt werden.

Die **Berechnungsphase** von $op(x)$ besteht aus der **Hauptphase**, in der die explizit festgelegten Anweisungen des Anweisungsteils von x bis auf die Return–Anweisung, falls x FS-Order ist, ausgeführt werden, gefolgt von der Synchronisationsphase. In der implizit definierten **Synchronisationsphase** wird die Abschlußsychronisation mit allen π-inneren, d.h. parallel zu x eingeordneten Akteuren durchgeführt. Hierbei werden insbesondere die Werte eventuell vorhandener Ausgabe– und Ein–Ausgabeparameter dieser Akteure an x übergeben. Die Synchronisationsphase endet erst dann, wenn alle zu x π-inneren Akteure terminiert sind und deren Parameterwerte übergeben wurden. Ist x FS-Order, so folgt auf die Synchronisationsphase noch die Ergebnisphase, in der die Return–Anweisung von x ausgeführt wird. Der Ergebniswert einer FS-Order kann sich somit auf die Ergebnisse parallel eingeordneter Akteure stützen. Mit Abschluß der Ausführung der Synchronisationsphase bzw. der Ergebnisphase wird x in den Zustand T überführt.

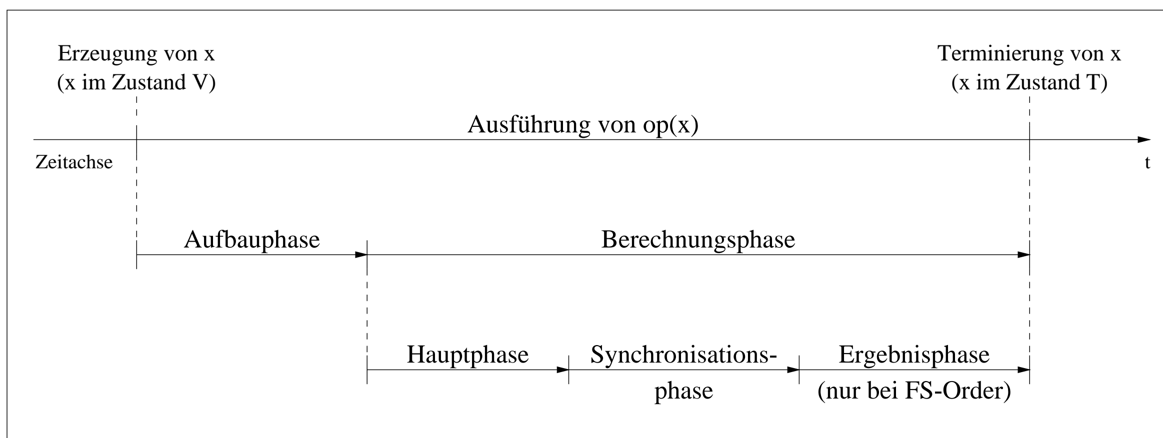

Abbildung 3.5: Ausführungsphasen einer kanonischen Operation

Die konzeptionelle Festlegung der Anfangs– und Abschlußsynchronisation in INSEL hat für den Anwendungsentwickler den Vorteil, daß er Akteure nicht explizit mit deren „Eltern"– und „Geschwister"–Akteuren unter Nutzung von *low–level* Synchronisationsprimitiven synchronisieren muß.

Während der Aufbau– und Hauptphase der kanonischen Operation einer DA–Inkarnation x können beliebige andere DA–Inkarnationen erzeugt werden. Innerhalb der Ausführung der kanonischen Operation dieser DA–Inkarnationen können wiederum alle möglichen Arten von Inkarnationen erzeugt werden. Dies hat zur Konsequenz, daß sich aus der kanonischen Operation $op(x)$ heraus ein beliebig komplexes Subsystem entwickeln kann. Dabei ist zwischen Inkarnationen, die unmittelbar von $op(x)$ ausgehend, und Inkarnationen, die mittelbar von $op(x)$ ausgehend erzeugt werden, zu unterscheiden. Eine Inkarnation wird **unmittelbar** von $op(x)$ ausgehend erzeugt, wenn sie durch Erarbeitung bzw. Ausführung eines im Deklarations– oder Anweisungsteil von x enthaltenen Erzeugungskonstrukts erzeugt wird. Demgegenüber wird sie **mittelbar** von $op(x)$ ausgehend erzeugt, wenn sie bei Ausführung der kanonischen Operation einer unmittelbar von $op(x)$ ausgehend erzeugten DA–Inkarnation inkarniert wird. Die ausgehend von $op(x)$ unmittelbar und mittelbar erzeugten DA–Inkarnationen leisten sämtlich einen Beitrag zur Funktionalität von $op(x)$. Der Teil der kanonischen Operation von $op(x)$, der durch die Deklarationen des Deklarations– und die Anweisungen des Anwei-

sungsteils von x festgelegt ist, wird **sequentieller Kern** von $op(x)$ genannt. So gehört die Erarbeitung bzw. Ausführung eines im Deklarations– oder Anweisungsteil von x enthaltenen DA–Erzeugungskonstrukts und das damit verbundene Erzeugen und Starten einer neuen DA–Inkarnation y zum sequentiellen Kern von $op(x)$. Die Ausführung von $op(y)$ gehört jedoch nicht zum sequentiellen Kern von $op(x)$. Sie erfolgt abhängig von der Inkarnationsart entweder sequentiell oder parallel eingeordnet in $op(x)$.

3.5 Ausführungsumgebung einer DA–Inkarnation

Die Akteure eines INSEL–Systems führen Berechnungen durch, indem sie Operationen auf Komponenten des Systems ausführen. Die Menge der Komponenten, die bei Ausführung der kanonischen Operation $op(x)$ einer DA–Inkarnation x zu einem Zeitpunkt $t \in \Lambda(x)$ konzeptionell benutzt werden können, wird **Ausführungsumgebung der DA–Inkarnation** x zum Zeitpunkt t genannt und mit $U_t(x)$ bezeichnet. Die Ausführungsumgebung einer DA–Inkarnation ergibt sich aus der strukturellen Einordnung der Inkarnation und den für INSEL festgelegten Sichtbarkeitsregeln. Diese Regeln entsprechen im wesentlichen den aus anderen block–orientierten Sprachen, wie z.B. Ada ([Ada83], [Fel96]), bekannten Regeln für die Sichtbarkeit von Komponenten. Besonderheiten ergeben sich daraus, daß in INSEL die beliebige Schachtelung unterschiedlicher Komponentenarten möglich ist. Im folgenden wird angegeben, wie die Ausführungsumgebung einer DA–Inkarnation bestimmt werden kann.

Die Ausführungsumgebung einer DA–Inkarnation x setzt sich aus den für x **sichtbaren** Komponenten und aus den über diese **mittelbar benutzbaren** Komponenten zusammen. Die Menge der für x sichtbaren Komponenten ergibt sich aus der Einordnung von x in die Definitionsstruktur (vgl. Abschnitt 3.3.2). Eine N–Komponente k ist für x sichtbar, wenn k entweder lokale N–Komponente von x ist oder k lokale N–Komponente einer zu x δ–außen eingeordneten DA–Inkarnation ist und k im Programmtext textuell vor dem Generator von x deklariert ist. In der folgenden Definition wird für eine DA–Inkarnation x die Menge der für x sichtbaren Komponenten festgelegt.

Definition 3.12.: Menge der für eine DA–Inkarnation sichtbaren Komponenten

Seien $x \in X_t^{DAI} \setminus \{\underline{a}\}$ eine DA–Inkarnation und $y \in X_t^{DAI}$ die DA–Inkarnation mit $gen(x) \in L_0(y)$, also $(x,y) \in \delta_t$. Weiter sei $\leq_{L_0(y)}$ die lineare Ordnung auf der Menge der lokalen N-Komponenten von y, die aus der sequentiellen, textuellen Aufschreibung der Deklarationen des Generators von y in dem entsprechenden INSEL–Programm abgeleitet ist. *PredefinedTypes* bezeichne die Menge der in jedem INSEL–System vordefinierten Generatoren für die elementaren Datentypen.

Die Menge $V(x,y,gen(x))$ der für x sichtbaren lokalen N–Komponenten von y sei definiert durch:

$$V(x,y,gen(x)) \triangleq \{k \in X_t \mid k \in L_0(y) \wedge k \leq_{L_0(y)} gen(x)\}$$

Dann ist die **Menge der für x sichtbaren Komponenten** – $Visible(x,y,gen(x))$ –

definiert durch[7]:

$$Visible(x, y, gen(x)) \triangleq \begin{cases} L_0(x) \cup V(x, y, gen(x)) \cup \mathit{PredefinedTypes} & \text{falls } y \text{ die Hauptkomponente ist} \\ L_0(x) \cup V(x, y, gen(x)) \cup Visible(y, z, gen(y)) & \text{sonst, mit } gen(y) \in L_0(z) \end{cases}$$

Für die Menge der für die Hauptkomponente $\underline{a}$ sichtbaren Komponenten gilt:

$$Visible(\underline{a}) \triangleq L_0(\underline{a}) \ \cup \ \mathit{PredefinedTypes}$$

□

Die Menge $Visible(x, y, gen(x))$ ist nach Erarbeitung des Deklarationsteils von x, also nach Erarbeitung der lokalen N–Komponenten von x konstant. Sie enthält die N–Komponenten des INSEL–Systems, die von der DA–Inkarnation x unmittelbar benutzt werden können. Über die Elemente aus $Visible(x, y, gen(x))$ können weitere Komponenten mittelbar benutzbar sein, wie z.B. die mit dem Attribut E definierten Komponenten eines Depots oder explizit anonyme Inkarnationen, die mit einem Zeiger aus $Visible(x, y, gen(x))$ identifiziert werden.

Definition 3.13.: Menge der über eine Komponente mittelbar benutzbaren Komponenten

Sei $x \in X_t$ eine Komponente. Die **Menge** $Usable_t(x)$ **der Komponenten, die über** x **zum Zeitpunkt** t **mittelbar benutzbar** sind, ist folgendermaßen definiert:

$$Usable_t(x) \triangleq \begin{cases} \{x\} \cup \bigcup_{k \in L_0^E(x)} Usable_t(k) & \text{falls } x \text{ ein Depot oder ein K-Akteur ist} \\ \{x\} \cup \bigcup_{i=1}^{n} Usable_t(y_i) & \text{falls } x \text{ ein Record ist und } y_i \text{ die Record–Elemente von } x \text{ sind} \\ \{x\} \cup \bigcup_{i=1}^{n} Usable_t(y_i) & \text{falls } x \text{ ein Feld ist und } y_i \text{ die Feld–Elemente von } x \text{ sind} \\ \{x\} \cup \ Usable_t(z) & \text{falls } x \text{ ein Zeiger ist, der auf } z \text{ verweist} \\ \{x\} & \text{sonst} \end{cases}$$

□

Auf Basis der Definitionen (3.12) und (3.13) kann nun die Ausführungsumgebung einer DA–Inkarnation angegeben werden.

Definition 3.14.: Ausführungsumgebung einer DA–Inkarnation

Sei $x \in X_t^{DAI}$ eine DA–Inkarnation. Die **Ausführungsumgebung** $U_t(x)$ **der DA–Inkarnation** x **zum Zeitpunkt** t ist wie folgt definiert:

$$U_t(x) \triangleq \begin{cases} \bigcup_{k \in Visible(x)} Usable_t(k) & \text{falls } x \text{ die Hauptkomponente ist} \\ \bigcup_{k \in Visible(x,y,gen(x))} Usable_t(k) & \text{sonst, mit } gen(x) \in L_0(y) \end{cases}$$

□

[7] Überdeckungen von Komponenten durch Namensgleichheit werden hier nicht betrachtet. Es wird vielmehr postuliert, daß alle N–Komponenten eines INSEL–Systems systemweit eindeutige Namen haben.

Die Ausführungsumgebung $U_t(x)$ einer DA–Inkarnation x enthält alle die Komponenten, die bei Ausführung der kanonischen Operation $op(x)$ von x potentiell benutzt werden können. Sie ändert sich im allgemeinen dynamisch, da sie Zeiger enthalten kann, deren Wert sich durch Zeigerwert–Zuweisungen ändern kann. Die Ausführungsumgebung $U_t(x)$ wird wesentlich durch die Einordnung von x in die Definitionsstruktur und durch die sequentielle textuelle Einordnung des Generators von x in den Deklarationsteil des Generators der DA–Inkarnation, zu der x unmittelbar δ–innen ist, bestimmt.

Beispiel

Gegeben sei das bekannte Erzeuger–Verbraucher–System (vgl. Abbildung 3.2) zu dem in Abbildung 3.3 betrachteten Zeitpunkt t. Für die Ausführungsumgebungen der zum Zeitpunkt t existierenden DA–Inkarnationen des Erzeuger–Verbraucher–Systems gilt[8]:

- $U_t(\texttt{ErzeugerVerbraucherSystem}) = \{\texttt{PufferTyp}, \texttt{Puffer}, \texttt{Einfuegen}, \texttt{Entnehmen}, \texttt{Erzeuger}, \texttt{Verbraucher}\}$
- $U_t(\texttt{Puffer}) = \{\texttt{MaxAnzahl}, \texttt{IndexTyp}, \texttt{Feldtyp}, \texttt{Feld}, \texttt{Anfang}, \texttt{Ende}, \texttt{Einfuegen}, \texttt{Entnehmen}\}$
- $U_t(\texttt{Einfuegen}) = \{\texttt{Wert}, \texttt{MaxAnzahl}, \texttt{IndexTyp}, \texttt{Feldtyp}, \texttt{Feld}, \texttt{Anfang}, \texttt{Ende}\}$
- $U_t(\texttt{Erzeuger}) = \{\texttt{MaxErzeuge}, \texttt{Start}, \texttt{Diff}, \texttt{Wert}, \texttt{Puffer}, \texttt{Einfuegen}, \texttt{Entnehmen}, \texttt{PufferTyp}\}$
- $U_t(\texttt{Verbraucher}) = \{\texttt{MaxVerbrauche}, \texttt{Wert}, \texttt{Erzeuger}, \texttt{Puffer}, \texttt{Einfuegen}, \texttt{Entnehmen}, \texttt{PufferTyp}\}$

◇

Die Möglichkeiten zur differenzierten Festlegung der Ausführungsumgebung einer DA–Inkarnation ergeben sich mit den bisher erklärten Konzepten von INSEL aus der Anwendung des Schachtelungsprinzips und aus der Reihenfolge der textuellen Aufschreibung der Deklarationen in den Deklarationsteilen der entsprechenden Generatoren. Diese Differenzierungsmöglichkeiten sind jedoch unter dem Aspekt, die Ausführungsumgebung einer DA–Inkarnation x auf die für die Ausführung der kanonischen Operation $op(x)$ unerläßlich notwendigen Komponenten zu beschränken, im allgemeinen nicht ausreichend. Betrachtet man zum Beispiel das Erzeuger–Verbraucher–System, so sollte es möglich sein, die Benutzungsmöglichkeiten des Puffers `Puffer` konzeptionell so einzuschränken, daß lediglich der Erzeuger die Operation `Einfuegen` und nur der Verbraucher die Operation `Entnehmen` auf dem Puffer `Puffer` aufrufen kann. Eine entsprechende Beschränkung der Ausführungsumgebung des Erzeugers und des Verbrauchers ist jedoch mit den bisher angegebenen Konzepten nicht möglich. Dies liegt zum einen daran, daß eine Komponente k, die lokale N–Komponente einer DA–Inkarnation y ist, und die über k mittelbar benutzbaren Komponenten allen DA–Inkarnationen, die δ–innen zu y existieren können und deren Generatoren in dem Programmtext des Generators von y nach der Deklaration von k stehen, zur Nutzung zur Verfügung stehen. y **δ–exportiert** k zu allen diesen DA–Inkarnationen. Zum anderen kann eine DA–Inkarnation y potentiell alle Komponenten benutzen, die von DA–Inkarnationen, zu denen y δ–innen eingeordnet ist, zu y δ–exportiert werden. y **δ–importiert** alle diese Komponenten.

[8] Auf die Angabe der in der Ausführungsumgebung jeder DA–Inkarnation enthaltenen Elemente der Menge der elementaren Datentypen $\mathit{PredefinedTypes} = \{\texttt{boolean}, \texttt{char}, \texttt{string}, \texttt{integer}, \texttt{real}\}$ wird verzichtet.

Zur differenzierten Festlegung der Ausführungsumgebung von DA–Inkarnationen werden also Konzepte benötigt, die es ermöglichen, die lokalen N–Komponenten einer DA–Inkarnation y nur zu einem Teil der zu y δ–inneren DA–Inkarnationen zu δ–exportieren sowie die Menge der Komponenten, die y δ–importiert, beschränken zu können. Weiterhin sind Konzepte erforderlich, durch deren Einsatz die Menge der über eine Komponente k mittelbar benutzbaren Komponenten für einzelne DA–Inkarnationen, zu denen k δ–exportiert wird bzw. von denen k δ–importiert wird, beschränkt werden kann.

In [SEL+96] wurden mit den Import– und Exportbeschränkungen Konzepte eingeführt, mit denen einerseits die Menge der lokalen N–Komponenten, die eine DA–Inkarnation δ–importiert, und andererseits die Menge der Komponenten, die eine DA–Inkarnation δ–exportiert, eingeschränkt werden kann. Die Konzepte für Import– und Exportbeschränkungen ermöglichen es, die Menge der für eine DA–Inkarnation sichtbaren Komponenten differenzierter festzulegen. Diese Konzepte werden in dieser Arbeit erweitert und deshalb zusammenfassend in Kapitel 4 in Abschnitt 4.3 im Rahmen der Erklärung der Konzepte, die die Sprache INSEL$^+$ zur differenzierten Festlegung der Ausführungsumgebung von DA–Inkarnationen zur Verfügung stellt, erklärt.

3.6 INSEL als Implementierungssprache für sichere Systeme

Die Sprache INSEL stellt mit den ihr zugrundeliegenden Prinzipien und ihrem homogenen Konzepterepertoire auf hohem Abstraktionsniveau eine geeignete Ausgangsbasis für die Implementierung sicherer verteilter Systeme dar. Sie erfüllt mit den zur Verfügung stehenden Konzepten einen Großteil der in Abschnitt 2.3 an eine für die Implementierung sicherer Systeme geeignete Programmiersprache gestellten allgemeinen Anforderungen.

Die Komponenten–Konzepte von INSEL ermöglichen die Kapselung von Daten und unterstützen die Konstruktion aktiver und passiver Objekte mit beliebiger Granularität und wohldefinierten Operationen. Durch die konsequente Anwendung des Klassenprinzips in Form des Generator–Konzepts ist INSEL eine streng typisierte Sprache. INSEL beinhaltet weder eine Zeigerarithmetik noch Sprachkonstrukte für die explizite Speicherverwaltung durch entsprechende Allokations– und Freigabeanweisungen, so daß damit verbundene Sicherheitsprobleme, die aus der Möglichkeit zum direkten Speicherzugriff resultieren, in INSEL von vornherein ausgeschlossen sind. Zudem unterstützt INSEL keinen Polymorphismus, wodurch hiermit in Zusammenhang stehende Sicherheitsprobleme ebenfalls nicht auftreten können. Wesentlich für INSEL sind die vielfältigen Strukturierungsmöglichkeiten, die sich insbesondere aus der Möglichkeit zur beliebigen Schachtelung von Komponenten ergeben. Auf Basis des Schachtelungsprinzips können Sichtbarkeitsbereiche und damit Nutzungsmöglichkeiten von Komponenten differenziert festgelegt werden. Durch die systematische Ausnutzung der Strukturierungsmöglichkeiten lassen sich Subsysteme konstruieren, die Komponenten, die nur lokal in dem Subsystem benötigt werden, so kapseln, daß sie außerhalb des Subsystems nicht sichtbar sind. Unzulässige Zugriffsversuche, die sich aus der Nichteinhaltung von Sichtbarkeitsregeln ergeben, wie zum Beispiel die Nutzung eines nicht sichtbaren Namens oder einer Referenz auf eine gekapselte Komponente können bereits zur Übersetzungszeit eines INSEL–Programms statisch ermittelt und somit von vornherein verhindert werden.

Mit dem Generator–Konzept sowie den anderen Komponenten–Konzepten, der strengen Typisierung, den fehlenden Möglichkeiten für direkte Speichermanipulationen sowie den vielfältigen Schachtelungsmöglichkeiten zur Festlegung von Sichtbarkeitsbereichen stellt INSEL wichtige für die programmiersprachliche Konstruktion sicherer Systeme benötigte Basiskonzepte

bzw. –eigenschaften zur Verfügung. INSEL fehlt jedoch ein Ausnahmebehandlungskonzept, mit dem Fehlersituationen, die zum Beispiel durch dynamische Zugriffsverletzungen verursacht werden, erkannt und systematisch behandelt werden können. Zudem sind die Möglichkeiten, die INSEL mit der Schachtelung sowie den bestehenden Import– und Exportbeschränkungen zur statischen Beschränkung von Sichtbarkeitsbereichen und Nutzungsmöglichkeiten bietet, nicht ausreichend, um die Ausführungsumgebungen von DA–Inkarnationen gemäß des Need–to–know Prinzips so differenziert festlegen zu können, daß die potentiellen Benutzungsmöglichkeiten von Komponenten auf das minimal Notwendigste eingeschränkt werden. Sie sind deshalb entsprechend zu erweitern.

INSEL stellt derzeit keine spezifischen Sprachkonzepte zur expliziten Formulierung von Sicherheitseigenschaften bzw. zur Implementierung anwendungspezifisch festgelegter Sicherheitspolitiken zur Verfügung. Sie bietet jedoch aufgrund der vorhandenen Konzepte und deren hohem Abstraktionsniveau eine geeignete Basis für die Erweiterung um Konzepte, mit denen dynamische Zugriffskontrollpolitiken weitestgehend deklarativ implementiert werden können. Hierzu gehören insbesondere ein Benutzer– und ein Rollenkonzept sowie Konzepte zur Festlegung komplexer Zugriffsbeschränkungen für die Nutzung von Komponenten.

Die motivierten notwendigen Ergänzungen der INSEL–Sprachkonzepte werden von der Sprache INSEL$^+$ bereitgestellt, die in dem nun folgenden Kapitel erklärt wird.

Kapitel 4

Konzepte der Sprache INSEL^{+}

In diesem Kapitel wird die in dem vorangegangenen Kapitel erklärte Sprache INSEL um Konzepte zur Konstruktion sicherer Anwendungssysteme erweitert. Die um diese Konzepte erweiterte Sprache INSEL wird INSEL^{+} genannt. In Abschnitt 3.6 wurden bereits Erweiterungen von INSEL motiviert, die zur Konstruktion sicherer Systeme nach einem sprachbasierten *top–down* Ansatz benötigt werden. Dementsprechend erweitert INSEL^{+} den Konzeptevorrat von INSEL um zwei Arten von Konzepten. Zum einen wird INSEL um Konzepte für die systematische und differenzierte Festlegung der Ausführungsumgebung von DA–Inkarnationen ergänzt. Durch Einsatz dieser Konzepte können potentielle Benutzungsmöglichkeiten von Komponenten auf das Notwendigste beschränkt werden. Sie dienen damit als Basis zur Durchsetzung des in Abschnitt 2.2 erläuterten Need–to–know Prinzips. Zum anderen stellt INSEL^{+} Konzepte zur Konstruktion sogenannter **zugriffskontrollierter Komponenten** zur Verfügung, für deren Nutzung Rechte explizit und dynamisch vergeben werden können und auf die die Zugriffe durch andere Komponenten dynamisch kontrolliert werden. Die für die Nutzung einer zugriffskontrollierten Komponente vergebenen Rechte sind Rechte zur Ausführung äußerer Operationen dieser Komponente. Basis für die Rechtevergabe sind Vorbedingungen, die auf programmiersprachlicher Ebene für die äußeren Operationen einer zugriffskontrollierten Komponente festgelegt werden können. Diese Vorbedingungen werden durch spezielle Boolesche Ausdrücke, die als **Zugriffsrestriktionsausdrücke** bezeichnet werden, formuliert. Beim Zugriff auf eine zugriffskontrollierte Komponente, d.h. bei Aufruf einer äußeren Operation dieser Komponente wird zur Laufzeit des Systems eine Zugriffskontrolle durchgeführt, die darin besteht, den für die Operation festgelegten Zugriffsrestriktionsausdruck auszuwerten, und damit zu überprüfen, ob die Vorbedingung für die Ausführung der Operation für den aufrufenden Akteur erfüllt ist. Wird der Zugriffsrestriktionsausdruck zu *true* ausgewertet, so hat der aufrufende Akteur das Recht zur Ausführung der Operation; der Zugriff ist somit erlaubt und die Operation wird ausgeführt. Anderenfalls ist der Zugriff verboten und die Operation wird nicht ausgeführt.

Dieses Kapitel ist wie folgt gegliedert. In Abschnitt 4.1 werden die Konzepte von INSEL^{+} zunächst in den Kontext der Konstruktion sicherer Rechensysteme eingeordnet sowie die die diesen Konzepten zugrundeliegenden Sichtweisen erläutert. Abschnitt 4.2 gibt dann einen Überblick über die im Vergleich zu INSEL–Systemen erweiterten Eigenschaften von INSEL^{+}–Systemen. Dazu gehören insbesondere ein Benutzerbegriff sowie Erweiterungen der Ein– und Ausgabemöglichkeiten durch sogenannte abstrakte Terminals. In Abschnitt 4.3 werden die Konzepte zur differenzierten Festlegung der Ausführungsumgebung von DA–Inkarnationen erklärt. Abschnitt 4.4 beschreibt die Konzepte, die für die Konstruktion zugriffskontrollierter Komponenten zur Verfügung stehen. Bei Aufruf einer äußeren Operation einer zugriffskon-

trollierten Komponente kann die Situation eintreten, daß der aufrufende Akteur nicht das Recht zur Ausführung dieser äußeren Operation hat. Für den aufrufenden Akteur liegt in diesem Fall ein sogenanntes Zugriffsverbot vor. Um das Auftreten solcher Zugriffsverbote signalisieren und behandeln zu können, stellt INSEL$^+$ ein einfaches Ausnahmebehandlungskonzept zur Verfügung. Dieses Konzept wird in Abschnitt 4.5 erklärt. Die in den einzelnen Abschnitten eingeführten Konzepte werden anhand eines Beispiels, und zwar dem Kontenverwaltungssystem einer Bank, das in INSEL$^+$ implementiert wurde, verdeutlicht. Diese Beispielanwendung wird ausführlich zusammen mit einem weiteren etwas kleineren Beispiel in Abschnitt 4.6 erläutert. Das Kapitel schließt mit einer Zusammenfassung und einigen Anmerkungen dazu, wie die eingeführten Konzepte zur Konstruktion sicherer Anwendungssysteme einzusetzen sind. Die in den einzelnen Abschnitten dieses Kapitels angegebenen Syntaxregeln der INSEL$^+$–Sprachkonstrukte sind in Anhang A zusammengefaßt.

4.1 Einordnung und Übersicht

Subjekte, Objekte und Rechte
Zunächst ist zu klären, was die Subjekte und was die Objekte (in dem in Abschnitt 2.1 angegebenen Sinn) in einem INSEL$^+$–System sind. Weiterhin sind die Zugriffsrechte anzugeben, die an Subjekte zur Nutzung von Objekten vergeben werden können. Die **Subjekte** eines INSEL$^+$–Systems sind alle Akteure des Systems. Diese führen ihre Berechnungen im Auftrag von Benutzern des Systems aus, die in unterschiedlichen Rollen agieren können (siehe dazu Abschnitt 4.2.2). In diesem Sinn gehören somit auch die Benutzer und die Rollen des Systems zu den Subjekten. Die **Objekte** sind alle Komponenten des Systems. Die **Rechte**, die für die Nutzung von Komponenten vergeben werden, sind Rechte zur Ausführung der äußeren Operationen dieser Komponenten. In INSEL und damit auch in INSEL$^+$ stehen Konzepte zur Verfügung, die die Konstruktion von Komponenten mit beliebiger Granularität ermöglichen. Die äußeren Operationen dieser Komponenten können ebenfalls beliebig granular und damit anwendungsspezifisch definiert werden. Damit steht eine Basis für die anwendungsspezifische Festlegung von Objekten und Zugriffsrechten für diese zur Verfügung. Die zugriffskontrollierten Komponenten eines INSEL$^+$–Systems sind die Komponenten, für deren Nutzung Rechte explizit und dynamisch vergeben werden können. Die Rechte an einer nicht zugriffskontrollierten Komponente sind hingegen implizit und statisch vergeben. Auf eine formale Beschreibung, welche Subjekte eines INSEL$^+$–Systems welche Rechte an welchen Objekten des Systems in einem bestimmten Systemzustand haben, wird in diesem Kapitel nicht eingegangen. Eine entsprechende Formalisierung wird erst in Kapitel 5 angegeben.

Need–to–know Prinzip und Erlaubnisprinzip
In Abschnitt 2.2 wurden als Prinzipien, die bei der Konstruktion sicherer Rechensysteme zu beachten sind, u.a. das Erlaubnisprinzip und das Need–to–know Prinzip genannt. Das Erlaubnisprinzip fordert, daß grundsätzlich jeder Zugriff verboten ist, der nicht explizit durch Vergabe eines Rechts erlaubt wird. Das Need–to–know Prinzip besagt, daß ein Subjekt nur die Zugriffsrechte haben darf, die es zur Ausführung seiner Berechnungen benötigt.

Basis für die Berücksichtigung dieser beiden Prinzipien bei der Konstruktion von INSEL$^+$–Systemen ist eine im Vergleich zu INSEL restriktive Festlegung der Standard–Ausführungsumgebung einer DA–Inkarnation. Unter der Standard–Ausführungsumgebung einer DA–Inkarnation wird die Ausführungsumgebung verstanden, die sich ergibt, wenn keine Konzepte zur expliziten Beschränkung bzw. expliziten Erweiterung der Ausführungsumgebung eingesetzt werden. In INSEL$^+$ wird die Standard–Ausführungsumgebung einer DA–Inkarnation auf

die lokalen N–Komponenten dieser DA–Inkarnation und die über diese mittelbar nutzbaren Komponenten beschränkt. Erweiterungen der Ausführungsumgebung einer DA–Inkarnation sind explizit durch den Einsatz der Konzepte vorzunehmen, die in INSEL$^+$ zur differenzierten Festlegung von Ausführungsumgebungen zur Verfügung stehen. Bei Ausführung der kanonischen Operation einer DA–Inkarnation x eines INSEL$^+$–Systems kann also standardmäßig höchstens auf die lokalen N–Komponenten von x und die über diese mittelbar nutzbaren Komponenten zugegriffen werden; auf alle anderen Komponenten ist zunächst kein Zugriff möglich. Die Basis für Zugriffe auf andere Komponenten ist durch explizite Erweiterung der Standard–Ausführungsumgebung der DA–Inkarnation x zu schaffen.

Sei nun a der Akteur, der die kanonische Operation der DA–Inkarnation x ausführt. Dann hat a an allen nicht zugriffskontrollierten Komponenten, die in der Ausführungsumgebung von x liegen, alle Rechte. An den zugriffskontrollierten Komponenten der Ausführungsumgebung von x hat a hingegen lediglich die Rechte, die durch die für diese Komponenten explizit definierten Zugriffsbeschränkungen festgelegt sind. Die Rechte, die die Akteure eines INSEL$^+$–Systems an den nicht zugriffskontrollierten Komponenten des Systems haben, ergeben sich somit aus den Ausführungsumgebungen der DA–Inkarnationen, deren kanonische Operation sie jeweils ausführen.

Aus dem bisher Gesagten ergibt sich im Hinblick auf die Berücksichtigung des Need–to–know Prinzips und des Erlaubnisprinzips bei der Konstruktion von INSEL$^+$–Systemen folgendes. Das Need–to–know Prinzip wird konzeptionell durch die restriktive Festlegung der Standard–Ausführungsumgebung von DA–Inkarnationen berücksichtigt. Darauf aufbauend können durch Einsatz der Konzepte zur differenzierten Festlegung der Ausführungsumgebung die Ausführungsumgebungen der Komponenten des Systems so konstruiert werden, daß in diesen Umgebungen lediglich die Komponenten enthalten sind, die zur Ausführung der jeweiligen kanonischen Operation benötigt werden. Das Erlaubnisprinzip ist konzeptionell dadurch berücksichtigt, daß bei Ausführung der kanonischen Operation einer DA–Inkarnation grundsätzlich alle Zugriffe verboten sind bis auf Zugriffe auf die lokalen nicht zugriffskontrollierten Komponenten dieser Inkarnation. Zugriffe auf alle anderen Komponenten sind explizit zu erlauben. Zugriffserlaubnisse auf nicht zugriffskontrollierte Komponenten werden erteilt, indem diese Komponenten in die Ausführungsumgebung der Inkarnation aufgenommen werden. Der Rahmen für die erlaubten Zugriffe auf zugriffskontrollierte Komponenten wird durch die für die äußeren Operationen dieser Komponenten zu definierenden Zugriffsrestriktionsausdrücke festgelegt.

Implementierung von Zugriffskontrollpolitiken

Die Konzepte von INSEL$^+$ sind so gewählt, daß sie eine geeignete Basis für die Implementierung von Systemen mit anwendungsspezifisch festgelegten Zugriffskontrollpolitiken darstellen. Mit den Konzepten zur differenzierten Festlegung der Ausführungsumgebung können die statischen Rechtsfestlegungen einer solchen Politik programmiersprachlich implementiert werden. Zur Implementierung von Komponenten, für die im Rahmen der Politik Rechte dynamisch vergeben werden können und für die somit auch dynamische Zugriffskontrollen durchgeführt werden müssen, stehen die Konzepte zur Konstruktion zugriffskontrollierter Komponenten zur Verfügung. Aufgrund des hohen Abstraktionsniveaus dieser Konzepte wird der Systementwickler weitestgehend von realisierungstechnischen Fragen der Implementierung der Politik entlastet. So muß er zum Beispiel nicht die Durchführung von Zugriffskontrollen sowie die Verwaltung der Datenstrukturen, auf deren Basis diese Kontrollen durchgeführt werden (wie z.B. Zugriffskontrollisten), explizit programmieren, sondern lediglich angeben, welche Bedingungen bei den entsprechenden Zugriffskontrollen zu überprüfen sind. Auch die Implementierung spezieller Manager, wie sie in den bereits in Abschnitt 2.3 beschrie-

benen Capability–basierten Ansätzen ([KS78], [MA79],[ABL83]) vorgeschlagen wurden, ist hier nicht notwendig. Stattdessen kann die jeweils durchzusetzende Zugriffskontrollpolitik zum größten Teil "deklarativ" implementiert werden. Beispiele für die Implementierung anwendungsspezifischer Zugriffskontrollpolitiken mit den Konzepten von INSEL$^+$ werden in Abschnitt 4.6 angegeben.

Die INSEL$^+$–Konzepte unterstützen die Implementierung eines breiten Spektrums von Zugriffskontrollpolitiken. Mit den Möglichkeiten, die zur Definition von Zugriffsrestriktionsausdrücken zur Verfügung stehen, lassen sich komplexe Zugriffsbeschränkungen festlegen, die weit über das hinausgehen, was üblicherweise mit einfachen Zugriffsmatrix–Modellen erfaßbar ist. So können zum Beispiel Kontextrestriktionen formuliert werden, die festlegen, daß eine äußere Operation einer zugriffskontrollierten Komponente nur im Kontext der Ausführung einer bestimmten anderen Operation ausgeführt werden darf. Oder es können Bedingungen angegeben werden, deren Erfüllung von den Werten bestimmter Variablen, wie zum Beispiel die aktuelle Uhrzeit, abhängig ist. Um die Implementierung nicht bzw. nicht allein benutzerbestimmbarer Politiken zu ermöglichen, gilt in INSEL$^+$ nicht das klassische, sehr inflexible Owner–Prinzip, das festlegt, daß der Erzeuger einer Komponente alle Rechte an der erzeugten Komponente inklusive des Rechts zur Rechteänderung hat. In INSEL$^+$ wird zwar jeder Komponente implizit ein Besitzer zugeordnet, um auch die Implementierung benutzerbestimmbarer Politiken zu unterstützen; dieser Besitzer hat jedoch im allgemeinen keine Rechte an der Komponente. Die Rechte, die ein Besitzer einer zugriffskontrollierten Komponente an dieser Komponente haben soll, sind ebenso wie das Recht zur Rechteänderung an dieser Komponente explizit zu vergeben.

Mit dem in diesem Abschnitt Gesagten sind die den Konzepten von INSEL$^+$ zugrundeliegenden Sichtweisen, soweit sie für das Verständnis der folgenden Abschnitte notwendig sind, erklärt. Im folgenden Abschnitt wird nun ein Überblick über die im Vergleich zu INSEL–Systemen erweiterten Eigenschaften von INSEL$^+$–Systemen gegeben.

4.2 INSEL$^+$–Systeme

Ein INSEL$^+$–System besteht, wie ein INSEL–System, aus einer Menge von Komponenten. Unter diesen Komponenten kann es neben den bereits aus INSEL–Systemen bekannten Komponenten **zugriffskontrollierte Komponenten** geben, für deren Nutzung Rechte explizit vergeben werden können und auf die die Zugriffe durch andere Komponenten des Systems dynamisch kontrolliert werden. Basis für die Rechtevergabe sind Zugriffsrestriktionsausdrücke, die für die äußeren Operationen einer zugriffskontrollierten Komponente festzulegen sind. Die Eigenschaft, zugriffskontrolliert zu sein, ist eine Zusatzeigenschaft für einige der im Rahmen der INSEL–Konzepte zur Verfügung stehenden Komponentenarten. Auf die zugriffskontrollierten Komponenten und die Möglichkeiten zur Festlegung von Zugriffsrestriktionsausdrücken wird in diesem Abschnitt nicht weiter eingegangen. Die entsprechenden Konzepte werden ausführlich in Abschnitt 4.4 beschrieben.

Die Vergabe von Rechten für die Nutzung zugriffskontrollierter Komponenten macht lediglich Sinn, wenn die Komponenten eines INSEL$^+$–Systems von außen von Benutzern benutzbar sind. Weiterhin sollte die Möglichkeit bestehen, daß mehrere Benutzer gleichzeitig ein INSEL$^+$–System benutzen. Daher wird für INSEL$^+$–Systeme zum einen ein Benutzerbegriff eingeführt, und zum anderen werden die in INSEL bestehenden Möglichkeiten zur Ein– und Ausgabe erweitert. Diese erweiterten Ein– und Ausgabemöglichkeiten werden in Abschnitt

4.2.1 erklärt. Anschließend wird in Abschnitt 4.2.2 erläutert, wie Benutzer in ein INSEL$^+$–System integriert werden und wie diese Zugang zu einem INSEL$^+$–System erhalten können.

4.2.1 Erweiterte Ein– und Ausgabemöglichkeiten

Die einzige Möglichkeit der Interaktion zwischen einem INSEL–System und seiner Umwelt besteht in der Ausführung von Ein– und Ausgabeanweisungen (`Input`– und `Output`–Anweisungen) durch Komponenten des Systems. Diese Anweisungen ermöglichen die Ein– bzw. Ausgabe von Werten der in INSEL vordefinierten elementaren Datentypen[1]. In den bisherigen Ansätzen zur Realisierung von INSEL–Systemen auf vernetzten Workstations ([Rad96] und [Win96]) wurden die Eingabe– und die Ausgabeanweisung so realisiert, daß alle Ein– und Ausgaben in dem Shell–Fenster auf dem physikalischen Terminal vorgenommen werden, in dem der jeweilige Befehl zum Starten des INSEL–Systems eingegeben wurde. Es besteht somit keine Möglichkeit, daß mehrere Benutzer gleichzeitig an verschiedenen Terminals mit dem System arbeiten, indem sie Eingaben vornehmen und Ausgaben des Systems erhalten.

Aus diesem Grunde wurde INSEL um die Möglichkeit zur Erzeugung sogenannter **abstrakter Terminals** erweitert. Auf einem abstrakten Terminal sind Operationen für die Ein– und Ausgabe definiert. Neben Operationen zur Ein– und Ausgabe von Werten elementarer Datentypen bietet ein abstraktes Terminal weitere Operationen an. Hierzu gehören u.a. eine Operation zur verdeckten Eingabe, die z.B. für die Abfrage von Paßwörtern verwendet werden kann, und Operationen zur Ausgabe einfacher grafischer Primitive, wie z.B. einem Punkt oder einer Linie. Für die Erzeugung abstrakter Terminals steht ein spezieller vordefinierter Generator zur Verfügung. Ein abstraktes Terminal kann entweder als benannte oder anonyme Inkarnation bzgl. dieses Generators erzeugt werden. Die abstrakten Terminals eines INSEL$^+$–Systems sind mit den physikalischen Terminals der Hardware–Konfiguration, auf der das INSEL$^+$–System zur Ausführung gebracht wird, zu realisieren. Dazu ist bei Erzeugung eines abstrakten Terminals der Name des physikalischen Terminals der zugrundeliegenden Hardware–Konfiguration anzugeben, auf dem das abstrakte Terminal realisiert werden soll.

Auf die Realisierung abstrakter Terminals mit Hilfe von physikalischen Terminals wird in dieser Arbeit nicht weiter eingegangen. Verwiesen sei dazu auf [Wei96]. Dort ist die Realisierung von abstrakten Terminals mit Fenstern auf Basis von X11 ([Jon89]) und OSF/Motif ([OSF91]) auf den physikalischen Terminals vernetzter UNIX–Workstations beschrieben. Die Ausführung einer Ein– bzw. Ausgabeoperation auf einem abstrakten Terminal entspricht in dieser Realisierung einer Eingabe bzw. Ausgabe in dem Fenster, mit dem das abstrakte Terminal realisiert ist.

4.2.2 Benutzerintegration und Benutzerzugang

Die Wechselwirkungen zwischen einem INSEL$^+$–System und seiner Umwelt bestehen darin, daß Benutzer Eingaben an das System machen und Ausgaben des Systems erhalten. Die Benutzer eines INSEL$^+$–Systems gehören somit zur Umwelt des Systems. Die Rechte, die in einem INSEL$^+$–System vergeben sind, werden letztendlich an Benutzer vergeben und von diesen wahrgenommen. Damit ein Benutzer seine Rechte wahrnehmen kann, muß er in der Lage sein, Operationen auf den Komponenten des Systems aufrufen zu können. Dies erfordert eine geeignete Repräsentation von Benutzern innerhalb des Systems. Als Basis für die

[1]Dies sind `boolean`, `char`, `integer`, `real` und `string`.

Durchsetzung der Rechtefestlegungen eines INSEL$^+$–Systems und der dazu durchzuführenden Zugriffskontrollen sind die Benutzer zu identifizieren und zu authentifizieren. Diese Identifikation und Authentifikation erfolgt beim Zugang eines Benutzers zu dem INSEL$^+$–System. Im folgenden wird zunächst die Integration von Benutzern in ein INSEL$^+$–System erläutert. Anschließend wird erklärt, wie ein Benutzer Zugang zu einem INSEL$^+$–System erhalten kann.

Benutzerrepräsentanten
Die Benutzer eines INSEL$^+$–Systems werden innerhalb des Systems durch Benutzerrepräsentanten repräsentiert. Ein Benutzerrepräsentant ist ein spezieller M-Akteur, der es einem Benutzer ermöglicht, in dem System aktiv zu sein und äußere Operationen auf Komponenten des Systems aufzurufen. Ein Benutzerrepräsentant wird erzeugt, wenn ein Benutzer Zugang zu dem INSEL$^+$–System erhält, d.h. wenn er erfolgreich identifiziert und authentifiziert worden ist. Die Aktivität des Benutzers in dem System wird dann durch die kanonische Operation des Benutzerrepräsentanten, der für ihn erzeugt wurde, bestimmt. Der Benutzerrepräsentant kann dem Benutzer ein Menü zur Verfügung stellen, aus dem dieser auswählen kann, welche Aufgaben bzw. Operationen er innerhalb des Systems durchführen möchte. Der Benutzerrepräsentant führt diese Aufgaben dann im Auftrag des Benutzers innerhalb des Systems aus, indem er existierende Komponenten benutzt und neue Komponenten erzeugt. Die Nutzung des Systems durch den Benutzer ist in diesem Fall auf die Möglichkeiten beschränkt, die über das Menü des Benutzerrepräsentanten zur Verfügung gestellt werden. Der Benutzerrepräsentant kann jedoch auch ein sogenannter **INSEL–Kommandointerpreter** sein, der es dem Benutzer ermöglicht, durch Eingabe der Namen von DA–Generatoren, die in der Ausführungsumgebung des Benutzerrepräsentanten liegen, Inkarnationen bzgl. dieser Generatoren zu erzeugen. In diesem Fall kann der Benutzer potentiell alle Komponenten nutzen, die in der Ausführungsumgebung seines Benutzerrepräsentanten enthalten sind.

Die kanonischen Operationen aller DA–Inkarnationen, die bei Ausführung der kanonischen Operation eines Benutzerrepräsentanten unmittelbar oder mittelbar erzeugt werden, werden im Auftrag des Benutzers ausgeführt, für den dieser Benutzerrepräsentant erzeugt wurde. Die kanonischen Operationen der DA–Inkarnationen, die nicht ausgehend von einem Benutzerrepräsentanten erzeugt werden, wie z.B. die Inkarnationen, die in einer Initialisierungsphase des Systems von der Hauptkomponente erzeugt werden, werden im Auftrag des für alle INSEL$^+$–Systeme vordefinierten abstrakten Benutzers *System* ausgeführt. Jeder Benutzer eines INSEL$^+$–Systems wird innerhalb des Systems durch einen systemweit eindeutigen Benutzeridentifikator (*uid*) identifiziert. Die Menge der Benutzeridentifikatoren eines INSEL$^+$–Systems wird im weiteren mit $\mathcal{B}$ bezeichnet, wobei $uid(System) \in \mathcal{B}$ der Identifikator des vordefinierten Benutzers *System* ist. Für Benutzeridentifikatoren ist in jedem INSEL$^+$–System der DE–Generator `UserUidType` vordefiniert. Das in der folgenden Definition definierte Attribut $user(x)$ ordnet einer DA–Inkarnation x für ihre Lebenszeit eindeutig den Identifikator des Benutzers zu, in dessen Auftrag die kanonische Operation von x ausgeführt wird. Dabei werden die in (3.4) eingeführten Bezeichnungen vorausgesetzt.

Definition 4.1.: Zugeordneter Benutzer einer DA–Inkarnation

Sei $x \in X_t^{DAI}$ eine DA–Inkarnation. Der x **zugeordnete Benutzer** – $user(x)$ – ist wie folgt definiert:

$$user(x) \triangleq \begin{cases} B & \text{falls } x \text{ Benutzerrepräsentant ist und } B \text{ der Identifi-} \\ & \text{kator des Benutzers ist, für den } x \text{ erzeugt wurde} \\ uid(System) & \text{falls } x \text{ die Hauptkomponente ist} \\ user(creator(x)) & \text{sonst} \end{cases}$$

Dabei ist $creator(x)$ das in Definition (3.5) festgelegte Attribut, das den erzeugenden Akteur der DA–Inkarnation x angibt.

□

Um die Implementierung benutzerbestimmbarer Politiken zu unterstützen, ist jeder Komponente eines INSEL$^+$–Systems für ihre Lebenszeit ein Benutzer als Besitzer fest zugeordnet. Der Besitzer einer Komponente x wird als $owner(x)$ bezeichnet und ist folgendermaßen definiert.

Definition 4.2.: Besitzer einer Komponente

Sei $x \in X_t$ eine Komponente. Der **Besitzer** von x – $owner(x)$ – ist wie folgt definiert:

$$owner(x) \triangleq \begin{cases} user(x) & \text{falls } x \in X_t^{DAI} \\ user(y) & \text{falls } x \text{ benannte DE–Inkarnation ist und } x \in L_0(y) \\ user(y) & \text{falls } x \text{ anonyme DE–Inkarnation ist und } x \\ & \text{ bei Ausführung von } op(y) \text{ erzeugt worden ist} \end{cases}$$

□

Das Attribut $owner(x)$ einer Komponente x gibt gemäß Definition (4.2) den Identifikator des Benutzers an, in dessen Auftrag x erzeugt wurde. Programmiersprachlich kann in INSEL$^+$ auf das Attribut *owner* einer Komponente über die vordefinierte Konstante `Owner`, die Inkarnation bzgl. des vordefinierten DE–Generators `UserUidType` ist, zugegriffen werden. Der Bezeichner `Owner` kann insbesondere in Zugriffsrestriktionsausdrücken zur Festlegung von Zugriffsbeschränkungen bzgl. zugriffskontrollierter Komponenten genutzt werden (vgl. Abschnitt 4.4.4).

Rollen

Die Benutzer eines INSEL$^+$–Systems können das System zur Erfüllung unterschiedlicher Aufgaben benutzen. Diese Aufgaben lassen sich zu Aufgabenklassen zusammenfassen, die entsprechende Rollen definieren, in denen Benutzer in dem System agieren können. Für jeden Benutzer ist festgelegt, in welchen Rollen er in dem System agieren darf. Die Rechte, die ein Benutzer an den zugriffskontrollierten Komponenten des Systems hat, sind im allgemeinen abhängig von der Rolle, in der der Benutzer jeweils in dem System agiert. Auf Basis des Rollenkonzepts können Benutzern lediglich die Rechte gewährt werden, die sie zur Erfüllung einer bestimmten Aufgabe bzw. Aufgabenklasse unbedingt benötigen. Das Rollenkonzept unterstützt damit die Vergabe von Rechten an Benutzer gemäß des Need–to–know Prinzips.[2]

Die Bezeichner der für ein INSEL$^+$–System initial definierten Rollen sind in dem entsprechenden INSEL$^+$–Programm in einem Rollen–Part zu vereinbaren, dessen Syntax wie folgt festgelegt ist:

<role-part> ::= `ROLES` *<identifier-list>* ;

Beispiel

Abbildung 4.1 zeigt als Beispiel den Rollen–Part des INSEL$^+$–Programms, das das Kontenverwaltungssystem einer Bank implementiert.

[2]Die Notwendigkeit einer auf Rollen basierenden Zugriffskontrolle als Basis für die Vergabe von Rechten an Benutzer gemäß des Need–to–know Prinzips wird u.a. in [HT95] und [SCFY96] erläutert.

```
PROCESS KontenVerwaltungsSystem IS

  -- Rollen des Kontenverwaltungssystems
  ROLES Kunde, Kundenbetreuer, Kassierer, Bankleiter;
...
END KontenVerwaltungsSystem;
```

Abbildung 4.1: Beispiel für einen Rollen–Part

Für das Kontenverwaltungssystem sind also initial die Rollen *Kunde*, *Kundenbetreuer*, *Kassierer* und *Bankleiter* explizit definiert.

◇

Die initiale Rollenmenge eines INSEL$^+$–Systems kann zur Laufzeit des Systems dynamisch erweitert werden. Hierzu steht in jedem INSEL$^+$–System eine entsprechend vordefinierte Operation zur Verfügung.

Die für ein INSEL$^+$–System definierten Rollen werden innerhalb des Systems durch systemweit eindeutige Identifikatoren identifiziert. Die Menge der Rollenidentifikatoren eines INSEL$^+$–Systems wird durch den vordefinierten DE–Generator `RoleUidType` bereitgestellt und im weiteren mit $\mathcal{R}$ bezeichnet. Für jedes INSEL$^+$–System ist die Rolle *SystemRole* vordefiniert. In dieser Rolle agiert einzig und allein der vordefinierte abstrakte Benutzer *System*. $uid(SystemRole)$ bezeichnet den Rollenidentifikator der Rolle *SystemRole*. Analog zur Zuordnung eines Benutzers zu einer DA–Inkarnation kann jeder DA–Inkarnation eines INSEL$^+$–Systems eine Rolle zugeordnet werden. Das im folgenden definierte Attribut $role(x)$ ordnet einer DA–Inkarnation x für ihre Lebenszeit eindeutig den Identifikator der Rolle zu, in der der x zugeordnete Benutzer $user(x)$ bei Erzeugung von x agiert hat.

Definition 4.3.: Zugeordnete Rolle einer DA–Inkarnation

Sei $x \in X_t^{DAI}$ eine DA–Inkarnation. Die x **zugeordnete Rolle** – $role(x)$ – ist wie folgt definiert:

$$role(x) \triangleq \begin{cases} R & \text{falls } x \text{ Benutzerrepräsentant ist und } R \text{ der Identifi-} \\ & \text{kator der Rolle ist, in der der Benutzer, für den } x \\ & \text{erzeugt wurde, bei der Erzeugung von } x \text{ agiert hat} \\ uid(SystemRole) & \text{falls } x \text{ die Hauptkomponente ist} \\ role(creator(x)) & \text{sonst} \end{cases}$$

□

Benutzerzugang

Der Zugang von Benutzern zu einem INSEL$^+$–System erfolgt über sogenannte **Login–Akteure**. Die Login–Akteure sind spezielle M-Akteure, die für die Identifikation und Authentifikation von Benutzern zuständig sind. Sie führen ihre Aufgabe unter Nutzung des **Benutzerdatenverwalters** durch, der ein spezieller K-Akteur ist. Der Benutzerdatenverwalter verwaltet die für die Identifikation und Authentifikation von Benutzern benötigten Informationen, wie z.B. Benutzerkennungen und zugehörige Paßwörter sowie Benutzer–Rollenzuordnungen.

Beim Start eines INSEL$^+$–Systems wird für jedes physikalische Terminal der zugrundeliegenden Hardware–Konfiguration, an dem Benutzer Zugang zu dem System erhalten sollen, ein Login–Akteur erzeugt. Jeder Login–Akteur erzeugt ein abstraktes Terminal, das mit dem physikalischen Terminal, für das der Login–Akteur erzeugt wurde, realisiert wird. Der Login–Akteur führt alle Ein– und Ausgabeoperationen auf diesem abstrakten Terminal durch. Möchte ein Benutzer an einem Terminal Zugang zu einem INSEL$^+$–System erlangen, so muß er gegenüber dem entsprechenden Login–Akteur neben den Informationen, die für die Identifikation und Authentifikation benötigt werden, angeben, in welcher Rolle er in dem System agieren möchte. Sind alle von dem Login–Akteur unter Nutzung des Benutzerdatenverwalters durchgeführten Zugangskontrollen erfolgreich, erzeugt er abhängig von der angegebenen Rolle für den Benutzer einen rollenspezifischen Benutzerrepräsentanten. Dieser Benutzerrepräsentant führt alle Ein– und Ausgabeoperationen auf dem abstrakten Terminal durch, das von dem Login–Akteur erzeugt wurde. Mit dem erzeugten Benutzerrepräsentanten ist der Benutzer innerhalb des Systems repräsentiert und kann damit das System zur Ausführung der mit der jeweiligen Rolle verbundenen Aufgaben nutzen.

Nach dem oben Gesagten sind der Benutzerdatenverwalter als K-Akteur und die Login–Akteure als M-Akteure Komponenten des jeweiligen INSEL$^+$–Systems, die beim Start des Systems in einer Initialisierungsphase zu erzeugen sind. Die Generatoren für den Benutzerdatenverwalter und die Login–Akteure stehen in jedem INSEL$^+$–Programm vordefiniert zur Verfügung.

Strukturelle Einordnung der Benutzerrepräsentanten

Ein Benutzer, der Zugang zu einem INSEL$^+$–System erhalten hat und für den somit ein Benutzerrepräsentant erzeugt wurde, kann höchstens die Komponenten des INSEL$^+$–Systems nutzen, die in der Ausführungsumgebung dieses Benutzerrepräsentanten liegen. Wie in Abschnitt 3.5 erklärt, wird die Ausführungsumgebung einer DA–Inkarnation wesentlich durch ihre Einordnung in die Definitionsstruktur (δ–Struktur) des Systems bestimmt. Aus diesem Grund kommt der Einordnung der Benutzerrepräsentanten in diese Struktur besondere Bedeutung zu. Der Rahmen für die δ–Einordnung der Benutzerrepräsentanten ist dadurch festgelegt, daß die Benutzerrepräsentanten von den Login–Akteuren erzeugt werden müssen. Die Generatoren für die Benutzerrepräsentanten müssen also in der Ausführungsumgebung der Login–Akteure liegen. Wie bereits erläutert, werden die Login–Akteure beim Start eines INSEL$^+$–Systems erzeugt, und zwar durch die Hauptkomponente des Systems. Der Generator für die Login–Akteure ist implizit als lokale N–Komponente der Hauptkomponente definiert. Dementsprechend sind die Login–Akteure unmittelbar δ-innen zu der Hauptkomponente eingeordnet. Damit die Generatoren für die Benutzerrepräsentanten in der Ausführungsumgebung der Login–Akteure liegen, müssen diese Generatoren entweder ebenfalls als lokale N–Komponenten der Hauptkomponente oder als lokale N–Komponenten eines von der Hauptkomponente nutzbaren Depots, das diese Generatoren nach außen exportiert, definiert werden. Die Benutzerrepräsentanten werden also unmittelbar δ-innen zu der Hauptkomponente oder zu dem entsprechenden Depot eingeordnet. Diese δ–Einordnung der Benutzerrepräsentanten hat Konsequenzen für die Konstruktion des gesamten INSEL$^+$–Systems: Generatoren, die von Benutzerrepräsentanten und damit von Benutzern unmittelbar nutzbar sein sollen, müssen entweder ebenfalls im Deklarationsteil der Hauptkomponente definiert werden oder über ein von der Hauptkomponente zugreifbares Depot nutzbar sein. Dies gilt insbesondere auch für die Generatoren von Depots und K-Akteuren, deren äußere Operationen unmittelbar von Benutzern aufrufbar sein sollen.

Grobstruktur von INSEL$^+$–Systemen

Aus dem bisher Erklärten dieses Abschnitts folgt, daß sich die Komponenten eines INSEL$^+$–

Systems zwei verschiedenen "Subsystemen" zuordnen lassen. Unter einem Subsystem eines INSEL^{+}–Systems wird in dieser Arbeit allgemein eine Teilmenge von Komponenten des Systems verstanden, die unter spezifischen Gesichtspunkten zusammengefaßt sind. Das erste der beiden angesprochenen Subsysteme besteht aus den Komponenten, die für den Benutzerzugang und die Repräsentation der Benutzer innerhalb des Systems benötigt werden. Hierzu gehören die Login–Akteure, die abstrakten Terminals, der Benutzerdatenverwalter sowie die Benutzerrepräsentanten. Neben diesen Komponenten existieren in einem INSEL^{+}–System die Komponenten, die die eigentliche Anwendungsfunktionalität zur Verfügung stellen. Die Menge dieser Komponenten bildet das zweite Subsystem. Wechselwirkungen zwischen den beiden Subsystemen ergeben sich daraus, daß die Benutzerrepräsentanten Operationen auf den Komponenten aufrufen, die die Anwendungsfunktionalität implementieren. Die äußeren Operationen der Komponenten des zweiten Subsystems, die in der Ausführungsumgebung der Benutzerrepräsentanten liegen, bilden die Schnittstelle der Anwendung zu den Benutzern. Diese Schnittstelle ist in dem Sinne dynamisch, daß Komponenten, die von den Benutzerrepräsentanten potentiell nutzbar sind, dynamisch erzeugt und aufgelöst werden können.

Beispiel

Die sich ergebende Grobstruktur von INSEL^{+}–Systemen ist in Abbildung 4.2 anhand eines Ausschnitts aus einer Systemkonfiguration des in Abschnitt 4.6 erläuterten INSEL^{+}–Systems, das die Kontenverwaltung einer Bank realisiert, verdeutlicht. In der Abbildung sind nicht alle DA–Inkarnationen der erfaßten Systemkonfiguration dargestellt. Es fehlen z.B. die abstrakten Terminals sowie lokale Depots des K-Akteurs *KontoVerwalter* und des K-Akteurs *Benutzerdatenverwalter*. Ein durchgezogener Pfeil zwischen zwei DA–Inkarnationen besagt, daß entweder eine δ– oder eine α–Abhängigkeit zwischen den beiden DA–Inkarnationen besteht. Ein gestrichelter Pfeil drückt eine Nutzungsbeziehung zwischen zwei DA–Inkarnationen aus, wobei die Pfeilrichtung der Nutzungsrichtung entspricht.

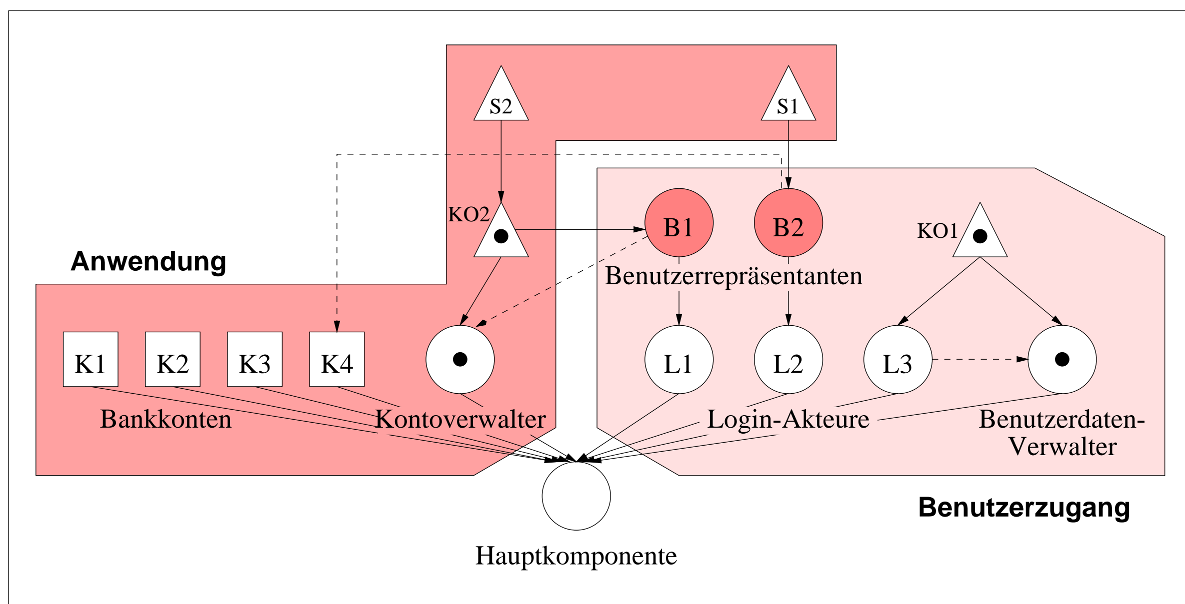

Abbildung 4.2: Ausschnitt aus einer Systemkonfiguration des Kontenverwaltungssystems

In dem hellgrau unterlegten Bereich der Abbildung sind die Komponenten des Benutzerzugangs und die Benutzerrepräsentanten zusammengefaßt. In der erfaßten Systemkonfiguration sind zwei Benutzer in dem System aktiv, was an den beiden dunkelgrau hervorgehobenen Benutzerrepräsentanten $B1$ und $B2$ erkennbar ist. Ein dritter Benutzer versucht gerade, Zu-

gang zu dem System zu erlangen. Für die dabei durchzuführende Zugangskontrolle nutzt der Login–Akteur $L3$ den *Benutzerdatenverwalter* über die K-Order $KO1$. Die Komponenten, die in dem dunkelgrau ausgefüllten Rahmen enthalten sind, implementieren die Funktionalität der Kontenverwaltung. Hierzu gehören insbesondere der K-Akteur *KontoVerwalter* und die als Depots realisierten Bankkonten $K1$ bis $K4$. Der Benutzer, der durch den Benutzerrepräsentant $B1$ repräsentiert ist, nutzt den *KontoVerwalter* über die K-Order $KO2$. Dieser Benutzer kann z.B. ein Kundenbetreuer sein, der ein neues Konto eröffnet und dazu eine Kommunikationsoperation des *KontoVerwalters* aufruft. Der Benutzerrepräsentant $B2$ führt gerade die S-Order $S1$ aus, die einer Zugriffsoperation des Kontodepots $K4$ entspricht. $B2$ kann z.B. einen Kunden der Bank repräsentieren, der Geld von dem Konto $K4$ abheben möchte und dazu eine Zugriffsoperation auf $K4$ aufruft.

◇

4.3 Konzepte zur differenzierten Festlegung der Ausführungsumgebung

In diesem Abschnitt werden die Konzepte erklärt, die in INSEL$^+$ zur differenzierten Festlegung der Ausführungsumgebung von DA–Inkarnationen zur Verfügung stehen. Diese Konzepte sind mit dem Ziel einzusetzen, die Ausführungsumgebung einer DA–Inkarnation x auf die für die Ausführung der kanonischen Operation von x unerläßlich notwendigen Komponenten einzuschränken und damit die potentiellen Nutzungsmöglichkeiten von Komponenten auf das Notwendigste zu beschränken. Mit diesen Konzepten lassen sich somit INSEL$^+$–Systeme gemäß des Need–to–know Prinzips konstruieren.

Zunächst werden in Abschnitt 4.3.1 Konzepte für Import– und Exportfestlegungen eingeführt. Durch Einsatz dieser Konzepte kann die Menge der für eine DA–Inkarnation sichtbaren Komponenten systematisch und differenziert festgelegt werden. Mit dem in Abschnitt 4.3.2 erklärten Konzept für Operationen–qualifizierte Zeiger steht ein Konzept zur Verfügung, mit dem die Nutzungsmöglichkeiten anonymer Depots und anonymer K-Akteure statisch beschränkt werden können.

4.3.1 Import– und Exportfestlegungen

Basis für eine dem Need–to–know Prinzip entsprechende Festlegung der Ausführungsumgebung von DA–Inkarnationen eines INSEL$^+$–Systems ist eine im Vergleich zu INSEL sehr restriktive Festlegung der Standard–Ausführungsumgebung einer DA–Inkarnation. Unter der Standard–Ausführungsumgebung einer DA–Inkarnation wird hier die Ausführungsumgebung verstanden, die sich ergibt, wenn keine Konzepte zur expliziten Beschränkung bzw. expliziten Erweiterung der Ausführungsumgebung eingesetzt werden. Wie in Abschnitt 3.5 erklärt, besteht die Standard–Ausführungsumgebung einer DA–Inkarnation x in INSEL aus den lokalen N–Komponenten von x und den lokalen N–Komponenten der DA–Inkarnationen, zu denen x δ–innen eingeordnet ist, sowie aus den jeweils über diese sichtbaren Komponenten mittelbar nutzbaren Komponenten. In INSEL$^+$ gehören zur Standard–Ausführungsumgebung einer DA–Inkarnation x lediglich die lokalen N–Komponenten von x und die über diese Komponenten mittelbar nutzbaren Komponenten sowie die vordefinierten elementaren Datentypen. Die Menge der für x sichtbaren Komponenten wird also auf die lokalen N–Komponenten von x und die elementaren Datentypen beschränkt. Diese restriktive Festlegung der Standard–Ausführungsumgebung ist dadurch motiviert, einer DA–Inkarnation neben Zugriffen auf die

elementaren Datentypen implizit höchstens den Zugriff auf ihre lokalen N–Komponenten und die über diese mittelbar nutzbaren Komponenten zu erlauben. Um einer DA–Inkarnation Zugriffe auf weitere Komponenten zu ermöglichen, ist ihre Standard–Ausführungsumgebung **explizit** zu erweitern. Durch die restriktive Festlegung der Standard–Ausführungsumgebung wird also neben dem Need–to–know Prinzip das für die Konstruktion sicherer Rechensysteme geforderte Erlaubnisprinzip geeignet berücksichtigt.

Importfestlegungen

Die Standard–Ausführungsumgebung einer DA–Inkarnation x kann explizit durch eine sogenannte **Importfestlegung** erweitert werden. In der Importfestlegung wird die Menge der lokalen N–Komponenten von DA–Inkarnationen, die δ–außen zu x eingeordnet sind, festgelegt, die für x sichtbar sein sollen. x **importiert** diese Komponenten. Wird eine lokale N–Komponente k importiert, so werden im allgemeinen auch alle über k mittelbar nutzbaren Komponenten implizit importiert. Eine Ausnahme ist lediglich für benannte Depots und benannte K-Akteure möglich. Für sie kann explizit festgelegt werden, welche ihrer mit dem Attribut E definierten lokalen N–Komponenten importiert werden sollen. Damit ist es möglich, die Menge der äußeren Operationen, die auf einem importierten benannten Depot oder benannten K-Akteur aufrufbar sind, statisch zu beschränken. Importfestlegungen sind Klasseneigenschaften. Die Importfestlegung einer DA–Inkarnation x wird dementsprechend mit dem Generator von x definiert.

Eine Importfestlegung wird entweder im Spezifikationsteil oder, falls kein solcher angegeben ist, im Implementierungsteil eines DA–Generators in einem **Import–Part** definiert, dessen Syntax wie folgt festgelegt ist:

<import-part> ::= `IMPORT` *<name-list>* ;

Der Import–Part eines DA–Generators G besteht aus dem Schlüsselwort `IMPORT` gefolgt von einer Liste von Namen. Die Liste dieser Namen legt die Menge der N–Komponenten fest, die von Inkarnationen bzgl. des Generators G importiert werden.

Sei nun x eine Inkarnation bzgl. G. $I(x)$ bezeichne im weiteren die Menge der N–Komponenten, die durch den Import–Part des Generators G festgelegt ist. $I(x)$ kann gemäß dem oben Gesagten lokale N–Komponenten von DA–Inkarnationen, zu denen x δ–innen eingeordnet ist sowie mit dem Attribut E definierte lokale N–Komponenten von benannten Depots und K-Akteuren, die selbst lokale N–Komponente einer zu x δ–außen eingeordneten DA–Inkarnation sind, enthalten. Enthält der DA–Generator G keinen Import–Part, so ist die Menge $I(x)$ leer.

Die Ausführungsumgebung einer DA–Inkarnation x eines INSEL$^+$–Systems setzt sich nun aus den lokalen N–Komponenten von x und den von x gemäß $I(x)$ importierten Komponenten sowie aus den jeweils über diese Komponenten mittelbar nutzbaren Komponenten zusammen.

Definition 4.4.: Ausführungsumgebung einer DA–Inkarnation eines INSEL$^+$–Systems

Sei $x \in X_t^{DAI}$ eine DA–Inkarnation eines INSEL$^+$–Systems. Die **Ausführungsumgebung $U_t(x)$ der DA–Inkarnation x zum Zeitpunkt t** ist wie folgt definiert:

$$U_t(x) \triangleq \bigcup_{k \in L_0(x)} Usable_t(k) \;\cup\; \bigcup_{k \in I(x)} Usable_t(k) \;\cup\; PredefinedTypes$$

$Usable_t(k)$ ist dabei die in Definition (3.13) definierte Menge, die aus der Komponente k und den über k mittelbar benutzbaren Komponenten besteht. *PredefinedTypes* bezeichnet die Menge der in INSEL$^+$ vordefinierten elementaren Datentypen.

□

Exportfestlegungen

Mit den Importfestlegungen ist es möglich, aus Sicht einer DA–Inkarnation x die Menge der Komponenten anzugeben, die für x sichtbar und damit nutzbar sein sollen. Nach dem bisher Gesagten können von x potentiell alle Komponenten importiert werden, die lokale N–Komponenten einer zu x δ–außen eingeordneten DA–Inkarnation sind und die textuell vor dem für die jeweilige δ–Einordnung verantwortlichen DA–Generator deklariert sind. Es besteht keine Möglichkeit, die Menge dieser potentiell von x importierbaren Komponenten aus Sicht der DA–Inkarnationen, die diese Komponenten deklarieren, differenziert festzulegen. Diese Möglichkeit wird durch die im folgenden erklärten Exportfestlegungen geschaffen.

Mit einer **Exportfestlegung** kann für eine lokale N–Komponente k einer DA–Inkarnation x bzw. für eine von x importierte N–Komponente k festgelegt werden, welchen DA–Inkarnationen, die δ–innen zu x existieren können, k aus Sicht von x potentiell zur Nutzung zur Verfügung stehen soll. x **δ–exportiert** k zu diesen DA–Inkarnationen. Ist für k keine Exportfestlegung explizit angegeben, so wird k von x zu keinen DA–Inkarnationen δ–exportiert. Dies bedeutet, daß k lediglich bei Ausführung der kanonischen Operation von x genutzt werden kann. Wird eine N–Komponente k von x zu einer DA–Inkarnation y δ–exportiert, so werden im allgemeinen auch alle über k mittelbar nutzbaren Komponenten zu y δ–exportiert. Eine Ausnahme ist lediglich möglich, wenn k ein benanntes Depot oder ein benannter K-Akteur ist. Dann kann explizit festgelegt werden, welche der mit dem Attribut E definierten lokalen N–Komponenten von k von x zu y δ–exportiert werden sollen. Damit können die Nutzungsmöglichkeiten eines von x δ–exportierten benannten Depots oder benannten K-Akteurs statisch beschränkt werden. Exportfestlegungen sind, wie Importfestlegungen auch, Klasseneigenschaften. Die Exportfestlegung für eine lokale bzw. importierte N–Komponente k der DA–Inkarnation x ist dementsprechend mit dem Generator von x definiert.

Exportfestlegungen werden entweder im Spezifikationsteil oder, falls kein solcher angegeben ist, im Implementierungsteil eines DA–Generators in einem **Export–Part** definiert, dessen Syntax wie folgt festgelegt ist:

<export-part>	::=	`EXPORT` *<export-list>* ;
<export-list>	::=	*<export-specification>* |
		<export-list> ; *<export-specification>*
<export-specification>	::=	*<export-name-list>* |
		<export-name-list> `TO` *<identifier-list>*

Der Export–Part eines DA–Generators G besteht aus dem Schlüsselwort `EXPORT` gefolgt von einer Liste von Exportspezifikationen. Eine Exportspezifikation hat entweder die Form *<export-name-list>* `TO` *<identifier-list>* oder *<export-name-list>*. Eine Exportspezifikation der ersten Form wird als beschränkte Exportspezifikation und eine der zweiten Form als unbeschränkte Exportspezifikation bezeichnet. Eine beschränkte Exportspezifikation setzt sich aus zwei durch das Schlüsselwort `TO` getrennten Namenslisten zusammen. Die erste dieser Namenslisten wird Exportnamensliste und die zweite Generatorliste genannt. Eine unbeschränkte Exportspezifikation besteht nur aus einer Exportnamensliste. Die Exportnamenslisten des Generators G legen die Menge der N–Komponenten fest, die von Inkarnationen

bzgl. G δ–exportiert werden. In einer Exportnamensliste von G sind dementsprechend die Namen der Komponenten erlaubt, die im Deklarationsteil von G deklariert werden oder die im Import–Part von G enthalten sind. Weiterhin sind die Namen lokaler mit dem Attribut E definierter Komponenten von benannten Depots und benannten K-Akteuren erlaubt, die im Deklarationsteil von G deklariert oder im Import–Part von G enthalten sind. Ein Name, der in den Exportnamenslisten von G erlaubt ist, darf höchstens in *einer* Exportnamensliste von G auftreten. In der Generatorliste einer beschränkten Exportspezifikation des Generators G können Namen von DA–Generatoren angegeben werden, die im Deklarationsteil von G definiert werden.

Sei nun x eine Inkarnation bzgl. G und k eine N–Komponente, die entweder lokale N–Komponente von x ist oder von x importiert wird. Ist der Name von k in der Exportnamensliste einer beschränkten Exportspezifikation des Generators G enthalten, so wird k von x zu allen DA–Inkarnationen δ–exportiert, die bzgl. eines Generators erzeugt wurden, der in der Generatorliste der entsprechenden Exportspezifikation enthalten ist. Ist der Name von k hingegen Element der Exportnamensliste einer unbeschränkten Exportspezifikation von G, so δ–exportiert x k zu allen DA–Inkarnationen, die δ–innen zu x sind. Tritt der Name von k in keiner Exportnamensliste von G auf, wird k von x nicht δ–exportiert. Daraus folgt unmittelbar für den Fall, daß G keinen Export–Part enthält, von x überhaupt keine Komponenten δ–exportiert werden.

Durch Exportfestlegungen für die lokalen bzw. importierten N–Komponenten einer DA–Inkarnation x kann die Menge der Komponenten, die potentiell von einer zu x δ–innen eingeordneten DA–Inkarnation y importiert werden können, differenziert festgelegt werden. Ist y eine DA–Inkarnation, die unmittelbar δ–innen zu x eingeordnet ist, so kann y höchstens die Komponenten importieren, die x zu y δ–exportiert. Die Voraussetzung für die Nutzung einer nicht lokalen Komponente k durch eine DA–Inkarnation y lautet also, daß k zu y δ–exportiert wird und y k importiert.

Beispiel

Im folgenden wird anhand eines kleinen Beispiels der Einsatz der Konzepte für Import– und Exportfestlegungen erläutert. Betrachtet wird das Erzeuger–Verbraucher–System, dessen INSEL–Implementierung in Abbildung 3.2 angegeben wurde. In Abschnitt 3.5 wurde bereits motiviert, daß es möglich sein sollte, die Benutzungsmöglichkeiten des als benannten Depots realisierten Puffers `Puffer` konzeptionell so einzuschränken, daß lediglich der Erzeuger die Operation `Einfuegen` und nur der Verbraucher die Operation `Entnehmen` auf dem Puffer aufrufen kann. Dies kann mit den für INSEL$^+$ eingeführten Import– und Exportfestlegungen leicht erreicht werden. In Abbildung 4.3 ist eine entsprechende INSEL$^+$–Implementierung des Erzeuger–Verbraucher–Systems auszugsweise angegeben.

Durch den Export–Part des Generators der Hauptkomponente wird festgelegt, daß die Zugriffsoperation `Einfuegen` des benannten Depots `Puffer` von der Hauptkomponente zu allen Akteuren, die bzgl. des Generators `Erzeuger` inkarniert werden, δ–exportiert wird. Die Zugriffsoperation `Entnehmen` von `Puffer` wird von der Hauptkomponente zu allen Akteuren δ–exportiert, die Inkarnationen bzgl. des Generators `Verbraucher` sind. Damit ist die Voraussetzung für die Zulässigkeit entsprechender Importe durch Inkarnationen bzgl. der Generatoren `Erzeuger` bzw. `Verbraucher` geschaffen. Die Inkarnationen bzgl. des Generators `Erzeuger` bzw. `Verbraucher` importieren lediglich die Zugriffsoperation `Einfuegen` bzw. `Entnehmen` des Depots `Puffer`. Da die Hauptkomponente keine weiteren Komponenten zu diesen Inkarnationen δ–exportiert, können von diesen Inkarnationen auch keine weiteren Komponenten

```
PROCESS ErzeugerVerbraucherSystem
  EXPORT Puffer.Einfuegen TO Erzeuger;
         Puffer.Entnehmen TO Verbraucher;
IS
------
  DEPOT TYPE SPEC PufferTyp (MaxAnzahl : IN integer)
    EXPORT MaxAnzahl, Feld; Anfang TO Entnehmen; Ende TO Einfuegen;
  IS
    PROCEDURE TYPE SPEC Einfuegen (Wert : IN  integer);
    PROCEDURE TYPE SPEC Entnehmen (Wert : OUT integer);
  END PufferTyp;

  DEPOT TYPE PufferTyp (MaxAnzahl : IN integer) IS
    TYPE IndexTyp IS 0 .. MaxAnzahl-1;
    TYPE FeldTyp  IS ARRAY [IndexTyp] OF integer;
    Feld         : FeldTyp;
    Anfang, Ende : IndexTyp;

    PROCEDURE TYPE Einfuegen (Wert : IN  integer)
      IMPORT Feld, MaxAnzahl, Ende; IS
    BEGIN
      Feld[Ende] := Wert; Ende := (Ende+1) MOD MaxAnzahl;
    END Einfuegen;

    PROCEDURE TYPE Entnehmen (Wert : OUT integer)
      IMPORT Feld, MaxAnzahl, Anfang; IS
    BEGIN
      Wert := Feld[Anfang]; Anfang := (Anfang+1) MOD MaxAnzahl;
    END Entnehmen;

  BEGIN Anfang := 0; Ende := 0; END PufferTyp;

  DEPOT Puffer : PufferTyp (100);
------
  PROCESS TYPE Erzeuger (MaxErzeuge: IN integer; Startwert, Diff: IN integer)
    IMPORT Puffer.Einfuegen;
  IS
  BEGIN ... Puffer.Einfuegen(...); ... END Erzeuger;
------
  PROCESS TYPE Verbraucher (MaxVerbrauche: IN integer)
    IMPORT Puffer.Entnehmen;
  IS
  BEGIN ... Puffer.Entnehmen(...); ... END Verbraucher;
------
BEGIN
  FORK Erzeuger(...);
  FORK Verbraucher(...);
END ErzeugerVerbraucherSystem;
```

Abbildung 4.3: INSEL^{+}–Implementierung des Erzeuger–Verbraucher–Systems

importiert werden. Der Export–Part des Depot–Generators `PufferTyp` legt fest, daß ein Depot, daß Inkarnation bzgl. `PufferTyp` ist, seine lokalen N–Komponenten `MaxAnzahl` und `Feld` zu allen δ–inneren Inkarnationen δ–exportiert. Diese N–Komponenten können also sowohl bei Ausführung der Zugriffsoperation `Einfuegen` als auch bei Ausführung der Zugriffsoperation `Entnehmen` benutzt werden. Für die lokalen Variablen `Anfang` und `Ende` sind jedoch Beschränkungen festgelegt. Diese Beschränkungen besagen, daß auf die Variable `Anfang` bzw. `Ende` lediglich im Rahmen der Ausführung der Operation `Entnehmen` bzw. `Einfuegen` zugegriffen werden darf.

◇

Die mit den Import– und Exportfestlegungen festgelegten Nutzungsbeschränkungen werden statisch durch den INSEL$^+$–Übersetzer, auf den hier nicht weiter eingegangen wird, überprüft. Würde zum Beispiel im Import–Part des Generators `Erzeuger` die Zugriffsoperation `Entnehmen` des Depots `Puffer` importiert, so würde der INSEL$^+$–Übersetzer bei Übersetzung des Programms eine Fehlermeldung liefern, da diese Operation gemäß des Export–Parts des Generators der Hauptkomponente nicht zu dem Generator `Erzeuger` δ–exportiert wird.

4.3.2 Operationen–qualifizierte Zeiger

Mit den im vorhergehenden Abschnitt erklärten Import– und Exportfestlegungen ist es unter anderem möglich, die Menge der äußeren Operationen eines benannten Depots oder benannten K-Akteurs, die bei Ausführung der kanonischen Operation einer DA–Inkarnation aufrufbar sind, zu beschränken. Für anonyme Depots und anonyme K-Akteure sind derartige Beschränkungen ihrer Nutzbarkeit bisher nicht festlegbar. Ist ein Zeiger z, der auf ein anonymes Depot oder einen anonymen K-Akteur x zeigt, Element der Ausführungsumgebung einer DA–Inkarnation y, so können bei Ausführung der kanonischen Operation von y potentiell alle der für x definierten äußeren Operationen aufgerufen werden. In diesem Abschnitt wird mit den Operationen–qualifizierten Zeigern ein Konzept eingeführt, mit dem die Menge der über einen Zeiger aufrufbaren äußeren Operationen eines anonymen Depots oder K-Akteurs statisch beschränkt werden kann. Das Konzept für Operationen–qualifizierte Zeiger entspricht im wesentlichen dem in [Spi84] erklärten Konzept für qualifizierte Zugriffsobjekte. Es unterscheidet sich von diesem darin, daß für Operationen–qualifizierte Zeiger keine sogenannten Verweis–Wert–Einschränkungen (siehe [Spi84]) möglich sind.

In einem INSEL$^+$–System werden anonyme Depots und anonyme K-Akteure über Operationen–qualifizierte Zeiger identifiziert und benutzt. Ein Operationen–qualifizierter Zeiger z ist wie jeder Zeiger eines INSEL– oder INSEL$^+$–Systems mit einem Generator qualifiziert. Dieser Generator muß für Operationen–qualifizierte Zeiger ein Depot– oder K-Akteur–Generator sein. Er wird im weiteren mit $\tau(z)$ bezeichnet. Zusätzlich ist ein Operationen–qualifizierter Zeiger z mit einer Teilmenge der äußeren Operationen, die auf Inkarnationen bzgl. des Generators $\tau(z)$ explizit definiert sind, qualifiziert. Die Menge der äußeren Operationen, mit der z qualifiziert ist, wird $\omega(z)$ genannt. z kann lediglich auf Inkarnationen verweisen, die bzgl. des Generators $\tau(z)$ erzeugt wurden. Bezeichnet $\varphi_t(z)$ die Inkarnation, auf die z zum Zeitpunkt t verweist, so können unter Verwendung von z auf $\varphi_t(z)$ lediglich die äußeren Operationen aufgerufen werden, die in $\omega(z)$ enthalten sind. $\tau(z)$ und $\omega(z)$ sind für die Lebenszeit von z konstant. Sie sind mit dem Zeiger–Generator, bzgl. dem z inkarniert wurde, festgelegt. $\varphi_t(z)$ kann sich durch Zeigerwertzuweisungen ändern. Für die Zulässigkeit einer Wertzuweisung zwischen zwei Operationen–qualifizierten Zeigern gelten einige Bedingungen, die weiter unten erklärt werden.

Operationen–qualifizierte Zeiger werden bzgl. entsprechender Zeiger–Generatoren inkarniert, die gemäß folgender Syntax zu definieren sind:

<pointer-type-definition> ::= `POINTER TYPE` *<identifier>* `IS ACCESS` *<name>* *<with-part>*
<with-part> ::= *<empty>* | `WITH` *<identifier-list>*

Die Definition eines Generators für Operationen–qualifizierte Zeiger enthält neben dem Bezeichner des definierten Generators (*<identifier>*) einen Generator–Qualifikationsteil (*<name>*) und ggf. einen Operationen–Qualifikationsteil (`WITH` *<identifier-list>*). Die *<identifier-list>* des Operationen–Qualifikationsteils muß eine Liste von Bezeichnern äußerer Operationen sein, die auf Inkarnationen bzgl. des mit *<name>* identifizierten Depot– oder K-Akteur–Generators explizit definiert sind. Der mit dem Generator–Qualifikationsteil eines Operationen–qualifizierten Zeiger–Generators ZG festgelegte Generator wird im weiteren mit $\tau(ZG)$ und die mit dem Operationen–Qualifikationsteil festgelegte Menge von Operationen mit $\omega(ZG)$ bezeichnet. Enthält ZG keinen Operationen–Qualifikationsteil, so definiert er eine Klasse Operationen–qualifizierter Zeiger, die mit allen äußeren Operationen qualifiziert sind, die auf Inkarnationen bzgl. $\tau(ZG)$ explizit definiert sind. Anderenfalls definiert er eine Klasse Operationen–qualifizierter Zeiger, die mit der Operationenmenge $\omega(ZG)$ qualifiziert sind. Inkarnationen bzgl. des Operationen–qualifizierten Zeiger–Generators ZG werden wie üblich, d.h. durch Erarbeitung einer entsprechenden Zeiger–Deklaration erzeugt.

Bzgl. eines Operationen–qualifizierten Zeiger–Generators können **Subtypen** definiert werden, die selbst wiederum Operationen–qualifizierte Zeiger–Generatoren sind. Für die Definition eines Subtyps ist folgende Syntax zu verwenden:

<pointer-subtype-definition> ::= `POINTER SUBTYPE` *<identifier>* `IS` *<name>* *<with-part>*

Die Definition eines Subtyps enthält neben dem Bezeichner des als Subtyp definierten Generators (*<identifier>*) den Bezeichner des Zeiger–Generators (*<name>*), auf den sich die Subtyp–Definition bezieht, sowie einen nichtleeren Operationen–Qualifikationsteil. Ist SG ein Subtyp und ZG der Generator, auf den sich SG bezieht, so muß für die durch den Operationen–Qualifikationsteil von SG festgelegte Menge $\omega(SG)$ gelten: $\omega(SG) \subseteq \omega(ZG)$. Für den Generator $\tau(SG)$, mit dem die Inkarnationen bzgl. SG qualifiziert sind, gilt: $\tau(SG) = \tau(ZG)$. Ein Operationen–qualifizierter Zeiger–Generator bildet zusammen mit seinen Subtypen eine **Familie von Operationen–qualifizierten Zeiger–Generatoren**.

Nachdem die Möglichkeiten für die Definition von Operationen–qualifizierten Zeiger–Generatoren erklärt sind, kann nun die Bedingung für die Zulässigkeit einer Wertzuweisung zwischen zwei Operationen–qualifizierten Zeigern angegeben werden. Dazu seien z_1 und z_2 zwei Operationen–qualifizierte Zeiger zu einem Zeitpunkt t. Die Wertzuweisung $z_1 := z_2$ ist genau dann zulässig, wenn folgende Bedingung erfüllt ist:

$$gen(z_1) = gen(z_2) \text{ oder } gen(z_1) \text{ ist ein Subtyp von } gen(z_2)$$

Ist die Wertzuweisung $z_1 := z_2$ gemäß dieser Bedingung zulässig, so ist das Ergebnis der Wertzuweisung: $\varphi_{t'}(z_1) = \varphi_t(z_2)$, wobei t' dem Zeitpunkt der Ausführung der Wertzuweisung entspricht.

Die angegebene Bedingung muß aus drei Gründen erfüllt sein. Erstens darf ein Zeiger lediglich auf Inkarnationen verweisen, die bzgl. des Generators erzeugt wurden, mit dem der Zeiger qualifiziert ist. Da mit Erfüllung der obigen Bedingung $\tau(z_1) = \tau(z_2)$ gilt, ist gewährleistet,

daß z_1 nach der Wertzuweisung auf eine Inkarnation bzgl. $\tau(z_1)$ verweist. Der zweite Grund liegt darin, daß durch eine Wertzuweisung zwischen zwei Operationen–qualifizierten Zeigern nicht die mit den konzeptionellen Lebenszeitfestlegungen von INSEL bestehende Lebenszeitgarantie verletzt werden darf. Diese Lebenszeitgarantie gewährleistet, daß eine Komponente mindestens solange existiert wie sie potentiell genutzt werden kann. Wie in Abschnitt 3.3 angegeben, wird ein anonymes Depot bzw. ein anonymer K-Akteur k mit der DA–Inkarnation aufgelöst, die den Zeiger–Generator, der bei der Erzeugung von k angegeben wurde, als lokale N–Komponente enthält. Wäre die obige Bedingung nicht erfüllt, so bestünde die Möglichkeit, daß $gen(z_2)$ lokale N–Komponente einer DA–Inkarnation ist, die vor der DA–Inkarnation, die $gen(z_1)$ als lokale N–Komponente enthält, aufgelöst würde. Würde in diesem Fall die Wertzuweisung durchgeführt, so bestünde also die Gefahr, daß z_1 irgendwann auf eine Inkarnation verweist, die gar nicht mehr existiert und damit wäre die Lebenszeitgarantie verletzt. Interpretiert man die obige Bedingung unter dem Aspekt, daß die äußeren Operationen, mit denen die Zeiger z_1 und z_2 qualifiziert sind, Zugriffsmöglichkeiten auf die Inkarnationen, auf die sie jeweils verweisen, festlegen, so ergibt sich ein dritter Grund, warum diese Bedingung bei der Wertzuweisung von z_2 an z_1 erfüllt sein muß. Mit z_1 sollten nämlich nicht mehr Zugriffe auf $\varphi_t(z_2)$ möglich sein, als mit z_2 möglich sind. Dies ist gewährleistet, wenn die obige Bedingung erfüllt ist, da dann gilt: $\omega(z_1) \subseteq \omega(z_2)$.

Beispiel

Das eingeführte Konzept der Operationen–qualifizierten Zeiger wird im folgenden anhand des in Abbildung 4.4 skizzierten INSEL$^+$–Programms beispielhaft erläutert. Das Programm implementiert wiederum ein Erzeuger–Verbraucher–System, wobei der Puffer diesmal nicht als benanntes Depot sondern als anonymes Depot realisiert wird. Der Vorteil der Realisierung des Puffers als anonymes Depot liegt darin, daß die Größe des Puffers dynamisch zur Laufzeit des Systems festgelegt werden kann. Bei der Erzeugung des Erzeugers bzw. des Verbrauchers wird ein Operationen–qualifizierter Zeiger, der auf das anonyme Puffer–Depot verweist, als Parameter übergeben. Die Anforderung, die an das Erzeuger–Verbraucher–System gestellt ist, lautet, daß der Erzeuger lediglich die Zugriffsoperation `Einfuegen` und der Verbraucher nur die Zugriffsoperation `Entnehmen` auf dem Puffer ausführen kann.

Der Depot–Generator `PufferTyp` entspricht dem in Abbildung 4.3 angegebenen Depot–Generator `PufferTyp`. Auf die Angabe seines Implementierungsteils wurde hier deshalb verzichtet. Nach dem Depot–Generator `PufferTyp` werden drei Generatoren für Operationen–qualifizierte Zeiger definiert, die jeweils mit `PufferTyp` qualifiziert sind. Diese drei Generatoren bilden eine Familie Operationen–qualifizierter Zeiger–Generatoren. Es gilt:

$$\begin{aligned}
&\tau(\texttt{PufferPtrTyp}) = \tau(\texttt{EinfuegePtrTyp}) = \tau(\texttt{EntnehmePtrTyp}) = \texttt{PufferTyp}\\
&\omega(\texttt{PufferPtrTyp}) \quad = \{\texttt{Einfuegen}, \texttt{Entnehmen}\}\\
&\omega(\texttt{EinfuegePtrTyp}) = \{\texttt{Einfuegen}\}\\
&\omega(\texttt{EntnehmePtrTyp}) = \{\texttt{Entnehmen}\}
\end{aligned}$$

Der Operationen–qualifizierte Zeiger `PufferPtr` wird bzgl. des Generators `PufferPtrTyp` inkarniert. Im Anweisungsteil der Hauptkomponente wird zunächst ein anonymes Depot bzgl. des Generators `PufferTyp` erzeugt. Der dabei erzeugte Zeigerwert wird dem Zeiger `PufferPtr` zugewiesen. Anschließend wird der Erzeuger inkarniert, wobei als aktueller Parameter der Zeiger `PufferPtr` übergeben wird. Die dabei durchzuführende Wertzuweisung zwischen `PufferPtr` und dem formalen Parameter `EinfuegePtr` von `Erzeuger` ist zulässig, da `EinfuegePtrTyp` ein Subtyp von `PufferPtrTyp` ist. `EinfuegePtr` ist lediglich mit der Operation `Einfuegen` qualifiziert. Da der Erzeuger neben dem Generator `EinfuegePtrTyp`,

```
PROCESS ErzeugerVerbraucherSystem
  EXPORT EinfuegePtrTyp TO Erzeuger;
         EntnehmePtrTyp TO Verbraucher;
IS
------
  DEPOT TYPE SPEC PufferTyp (MaxAnzahl : IN integer)
    EXPORT MaxAnzahl, Feld; Anfang TO Entnehmen; Ende TO Einfuegen;
  IS
    PROCEDURE TYPE SPEC Einfuegen (Wert : IN  integer);
    PROCEDURE TYPE SPEC Entnehmen (Wert : OUT integer);
  END PufferTyp;
  ...
------
  -- Familie Operationen-qualifizierter Zeigergeneratoren
  POINTER TYPE PufferPtrTyp IS ACCESS PufferTyp;
  POINTER SUBTYPE EinfuegePtrTyp IS PufferPtrTyp WITH Einfuegen;
  POINTER SUBTYPE EntnehmePtrTyp IS PufferPtrTyp WITH Entnehmen;
  -- Zeigerdeklaration
  PufferPtr : PufferPtrTyp;

------
  PROCESS TYPE Erzeuger (EinfuegePtr: IN EinfuegePtrTyp; ...)
    IMPORT EinfuegePtrTyp;
  IS
  BEGIN ... EinfuegePtr.Einfuegen(...); ... END Erzeuger;
------
  PROCESS TYPE Verbraucher (EntnehmePtr: IN EntnehmePtrTyp; ...)
    IMPORT EntnehmePtrTyp;
  IS
  BEGIN ... EntnehmePtr.Entnehmen(...); ... END Verbraucher;
------

BEGIN
  INPUT Kapazitaet;
  PufferPtr := NEW(PufferPtrTyp,PufferTyp) (Kapazitaet);
  FORK Erzeuger(PufferPtr, ...);
  FORK Verbraucher(PufferPtr, ...);
END ErzeugerVerbraucherSystem;
```

Abbildung 4.4: INSEL$^+$–Beispielprogramm für Operationen–qualifizierte Zeiger

bzgl. dem `EinfuegePtr` inkarniert wird, keine weiteren Komponenten importiert, kann der Erzeuger somit lediglich die Operation `Einfuegen` auf dem mit `EinfuegePtr` identifizierten Puffer–Depot ausführen. Das für den Erzeuger und die Operation `Einfuegen` Gesagte gilt entsprechend für den Verbraucher und die Operation `Entnehmen`.

◇

Die mit den Operationen–qualifizierten Zeigern festgelegten Benutzungsmöglichkeiten von anonymen Depots und anonymen K-Akteuren sowie die Zulässigkeit von Wertzuweisungen zwischen Operationen–qualifizierten Zeigern werden statisch durch den INSEL$^+$–Überset-

zer überprüft. Wäre in dem erläuterten Beispielprogramm in dem Anweisungsteil des Generators `Erzeuger` zum Beispiel die Anweisung `EinfuegePtr.Entnehmen(...)` enthalten, so würde die Übersetzung des Programms mit einer Fehlermeldung abgebrochen, da der Zeiger `EinfuegePtr` lediglich mit der Operation `Einfuegen` qualifiziert ist.

Mit den erklärten Konzepten für Import– und Exportfestlegungen sowie dem Konzept für Operationen–qualifizierte Zeiger können die Ausführungsumgebungen von DA–Inkarnationen und damit die Benutzungsmöglichkeiten von Komponenten differenziert festgelegt werden. Die Konzepte können insbesondere dazu eingesetzt werden, die Ausführungsumgebungen der rollenspezifischen Benutzerrepräsentanten eines INSEL$^+$–Systems so einzuschränken, daß in diesen Ausführungsumgebungen jeweils nur die Komponenten liegen, die zur Erledigung der mit der jeweiligen Rolle verbundenen Aufgabe benötigt werden. Darüberhinaus können die Benutzungsmöglichkeiten dieser Komponenten der Rolle entsprechend beschränkt werden. Die mit den genannten Konzepten festgelegten Benutzungsmöglichkeiten von Komponenten sind jedoch statisch festgelegt. Zur Implementierung von Systemen, in denen Rechte zur Nutzung von Komponenten dynamisch vergeben werden sollen, werden zusätzliche Konzepte benötigt. INSEL$^+$ stellt hierfür die Konzepte zur Konstruktion zugriffskontrollierter Komponenten zur Verfügung. Diese Konzepte werden in dem nun folgenden Abschnitt erklärt.

4.4 Konzepte zur Konstruktion zugriffskontrollierter Komponenten

In Abschnitt 4.4.1 werden zunächst die unterschiedlichen Arten zugriffskontrollierter Komponenten, die in einem INSEL$^+$–System existieren können, und die Rechte, die für die Nutzung dieser Komponenten vergeben werden können, angegeben. Anschließend werden die Konzepte, die in INSEL$^+$ zur Konstruktion zugriffskontrollierter Komponenten zur Verfügung stehen, erklärt. In Abschnitt 4.4.2 werden sogenannte Views eingeführt, die in INSEL$^+$ die Einheiten für die Rechtevergabe sind und mit denen sich spezifische Benutzungssichten für eine zugriffskontrollierte Komponente festlegen lassen. In INSEL$^+$ ist für jede zugriffskontrollierte Komponente implizit eine Zugriffskontrolliste definiert. Abschnitt 4.4.3 beschreibt den Aufbau und die Sprachkonstrukte zur Initialisierung dieser Zugriffskontrollisten sowie die Operationen, die zur Verwaltung der Zugriffskontrollisten zur Verfügung stehen. Die Rechte zur Ausführung der äußeren Operationen einer zugriffskontrollierten Komponente, für die Rechte explizit vergeben werden können, werden auf Basis von Zugriffsrestriktionsausdrücken vergeben. Die Möglichkeiten, die zur Definition von Zugriffsrestriktionsausdrücken zur Verfügung stehen, werden in Abschnitt 4.4.4 erklärt.

4.4.1 Zugriffskontrollierte Komponenten

Die zugriffskontrollierten Komponenten eines INSEL$^+$–Systems sind die Komponenten, für deren Nutzung Rechte explizit und dynamisch vergeben werden und auf die dementsprechend die Zugriffe durch anderen Komponenten des Systems dynamisch kontrolliert werden. Mit den zugriffskontrollierten Komponenten wird keine neue Komponentenart im Sinne der in Abschnitt 3.2 erklärten grundlegenden INSEL–Komponentenarten (wie Generatoren, Order, Depots und Akteure mit ihren jeweiligen Unterarten) eingeführt, sondern die Eigenschaft, zugriffskontrolliert zu sein, kann als zusätzliche Eigenschaft für einige der im Rahmen der INSEL–Konzepte zur Verfügung stehenden Komponentenarten definiert werden. Die Eigenschaft *zugriffskontrolliert* liefert damit ein weiteres Klassifikationskriterium für die Menge

der Komponenten eines INSEL$^+$–Systems, und zwar läßt sich diese Menge in die Menge der zugriffskontrollierten Komponenten und in die Menge der nicht zugriffskontrollierten Komponenten einteilen (vgl. Abbildung 4.5).

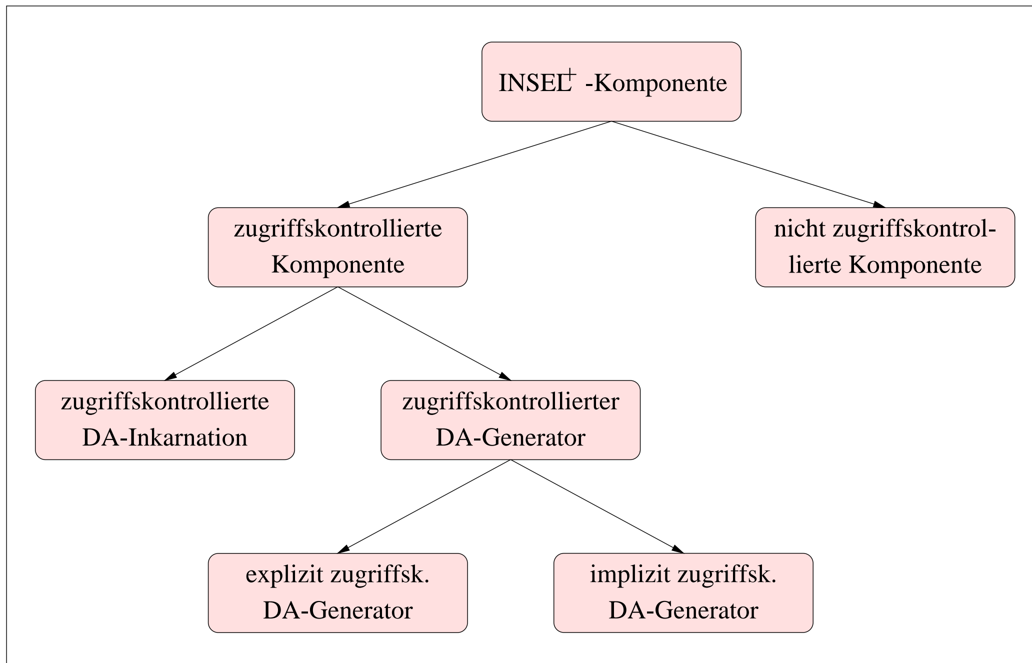

Abbildung 4.5: Klassifikation der INSEL$^+$–Komponenten

Die zugriffskontrollierten Komponenten lassen sich weiter klassifizieren in **zugriffskontrollierte DA–Generatoren** und **zugriffskontrollierte DA–Inkarnationen**. Eine DA–Inkarnation ist genau dann zugriffskontrolliert, wenn der DA–Generator, bzgl. dem die Inkarnation erzeugt wurde, zugriffskontrolliert ist. Die Eigenschaft einer DA–Inkarnation, zugriffskontrolliert zu sein, ist somit eine Klasseneigenschaft.

Die zugriffskontrollierten DA–Generatoren werden unterteilt in **explizit zugriffskontrollierte DA–Generatoren** und **implizit zugriffskontrollierte DA–Generatoren**. Ein DA–Generator ist explizit zugriffskontrolliert, wenn er explizit durch Voranstellung des Schlüsselwortes `PROTECTED` als solcher definiert ist:

<*protected-generator*> ::= `PROTECTED` <*da-generator*>

Mit Ausnahme der BS-Order–Generatoren können alle Arten von DA–Generatoren als explizit zugriffskontrollierte DA–Generatoren definiert werden.

Ein DA–Generator ist implizit zugriffskontrolliert, wenn er lokale mit dem Attribut E definierte N–Komponente eines zugriffskontrollierten Depots oder zugriffskontrollierten K-Akteurs ist. In diesem Fall ist die *erzeuge*–Operation des DA–Generators äußere Operation des zugriffskontrollierten Depots bzw. des zugriffskontrollierten K-Akteurs. Der DA–Generator definiert somit eine Zugriffsoperation eines zugriffskontrollierten Depots bzw. eine Kommunikationsoperation eines zugriffskontrollierten K-Akteurs.

Beispiel
In Abbildung 4.6 ist ein Ausschnitt aus dem Spezifikationsteil eines explizit zugriffskontrollierten Depot–Generators angegeben. Der Depot–Generator ist dem in Anhang B enthaltenen

INSEL$^+$–Programm, das die Funktionalität des Kontenverwaltungssystems einer Bank implementiert, entnommen. Dieses in Abschnitt 4.6 ausführlich beschriebene Beispielprogramm wird im Verlauf dieses Kapitels noch mehrfach zur Erläuterung der eingeführten Sprachkonstrukte verwendet werden.

```
-- Zugriffskontrollierter Depot-Generator fuer die Bankkonten
PROTECTED DEPOT TYPE SPEC KontoTyp (...)
  ...
IS
  -- Zugriffsoperationen
  FUNCTION  TYPE SPEC LeseKontostand RETURN real;
  PROCEDURE TYPE SPEC Einzahlen (Zweck: IN string; Betrag: IN real);
  PROCEDURE TYPE SPEC Abheben   (Zweck: IN string; Betrag: IN real;
                                 Flag: IN boolean; Error: OUT ErrorType);
  ...
END KontoTyp;
```

Abbildung 4.6: Beispiel für einen Spezifikationsteil

Der explizit zugriffskontrollierte Depot–Generator `KontoTyp`, dessen Spezifikationsteil in Abbildung 4.6 angegeben ist, ist der Generator für die von dem Kontenverwaltungssystem zu verwaltenden Bankkonten. Die Bankkonten werden als zugriffskontrollierte Depots bzgl. des Generators `KontoTyp` erzeugt. Die im Spezifikationsteil des Depot–Generators `KontoTyp` vereinbarten DA–Generatoren, wie z.B. `Einzahlen` und `Abheben`, definieren die Zugriffsoperationen eines Bankkontos. Diese DA–Generatoren sind implizit zugriffskontrolliert.

◇

Rechte an zugriffskontrollierten Komponenten

Die Rechte, die für die Nutzung zugriffskontrollierter Komponenten vergeben werden, sind Rechte zur Ausführung äußerer Operationen dieser Komponenten. Auf Generatoren ist gemäß Abschnitt 3.2 lediglich eine äußere Operation definiert, und zwar die Operation *erzeuge*, deren Ausführung die Erzeugung einer Inkarnation bzgl. der durch den jeweiligen Generator definierten Klasse bewirkt. Für die Nutzung zugriffskontrollierter DA–Generatoren kann also das Recht zur Ausführung der Operation *erzeuge* explizit vergeben werden.

Auf einer DA–Inkarnation ist nach dem in Abschnitt 3.2 Erklärten implizit genau eine äußere Operation definiert, deren Ausführung den Start der kanonischen Operation der DA–Inkarnation bewirkt. Diese Operation wird als Startoperation bezeichnet. Für die Ausführung der Startoperation einer zugriffskontrollierten DA–Inkarnation können keine Rechte explizit vergeben werden. Das Recht zur Ausführung der Startoperation wird implizit an den Akteur vergeben, der die zugriffskontrollierte DA–Inkarnation erzeugt hat, d.h. der die Operation *erzeuge* auf dem entsprechenden zugriffskontrollierten DA–Generator ausgeführt hat. Eine Ausnahme gilt lediglich für die zugriffskontrollierten K-Order. Das Recht zur Ausführung der Startoperation *führe_aus* auf einer zugriffskontrollierten K-Order hat nicht der die K-Order erzeugende Akteur, sondern der zugriffskontrollierte K-Akteur, der den entsprechenden K-Order–Generator als lokale N–Komponente enthält, der also die K-Order annehmen kann. Dies entspricht der Tatsache, daß die Startoperation einer K-Order nicht unmittelbar nach Erzeugung der K-Order von dem erzeugenden Akteur aufgerufen wird, sondern erst bei Annahme der K-Order durch den annehmenden K-Akteur.

Auf zugriffskontrollierten Depots und zugriffskontrollierten K-Akteuren können neben der implizit definierten Startoperation weitere äußere Operationen explizit definiert sein. Für einen zugriffskontrollierten K-Akteur sind dies die *erzeuge*–Operationen auf seinen lokalen K-Order–Generatoren, die die Kommunikationsoperationen des K-Akteurs definieren. Diese K-Order–Generatoren sind nach dem oben Gesagten implizit zugriffskontrolliert. Für ihre *erzeuge*–Operationen und damit für die Kommunikationsoperationen des K-Akteurs können Rechte explizit vergeben werden.

Für zugriffskontrollierte Depots können lokale DA–Generatoren sowie lokale benannte Depots und benannte K-Akteure mit dem Attribut E definiert werden. Die Zugriffsoperationen eines zugriffskontrollierten Depots sind dann die *erzeuge*–Operationen auf den mit dem Attribut E definierten lokalen DA–Generatoren sowie die äußeren Operationen der mit dem Attribut E definierten lokalen Depots und K-Akteure. Im Gegensatz zu nicht zugriffskontrollierten Depots können lokale benannte DE-Inkarnationen und DE–Generatoren eines zugriffskontrolliertes Depots nicht mit dem Attribut E definiert werden. Dies bedeutet jedoch keine wesentliche Einschränkung, sondern entspricht der objekt–basierten Sicht, daß ein Objekt lokale Daten und Methoden zur deren Nutzung definiert. Soll eine lokale benannte DE-Inkarnation eines zugriffskontrollierten Depots von außen benutzbar sein, so sind explizit entsprechende lokale DA–Generatoren mit dem Attribut E zu definieren, die das Lesen bzw. Schreiben auf der entsprechenden DE-Inkarnation ermöglichen.

Die mit dem Attribut E definierten lokalen DA–Generatoren eines zugriffskontrollierten Depots sind nach dem oben Gesagten implizit zugriffskontrolliert. Für ihre *erzeuge*–Operationen und damit für die durch sie definierten Zugriffsoperationen des Depots können Rechte explizit vergeben werden. Für die Zugriffsoperationen eines zugriffskontrollierten Depots zd, die den äußeren Operationen eines lokalen mit dem Attribut E definierten Depots ld bzw. K-Akteurs lk entsprechen, sind die Rechte vergeben, die für die Ausführung dieser äußeren Operationen auf ld bzw. lk vergeben sind. Die Rechte, die zur Ausführung der äußeren Operationen von ld bzw. lk vergeben sind, werden also implizit für die Zugriffsoperationen von zd, die diesen äußeren Operationen entsprechen, übernommen. Falls ld bzw. lk nicht zugriffskontrolliert ist, können somit für die Zugriffsoperationen von zd, die äußeren Operationen von ld bzw. lk entsprechen, keine Rechte explizit vergeben werden. Sollen für diese Zugriffsoperationen Rechte explizit vergeben werden können, so ist ld bzw. lk selbst als zugriffskontrollierte Komponente zu konstruieren. Für die äußeren Operationen von ld bzw. lk und damit für die diesen entsprechenden Zugriffsoperationen von zd können dann Rechte explizit vergeben werden.

Neben den bisher genannten äußeren Operationen sind auf jedem explizit zugriffskontrollierten DA–Generator sowie jedem zugriffskontrollierten Depot und zugriffskontrollierten K-Akteur implizit zwei weitere äußere Operationen definiert, die für die Ausgabe bzw. die Modifikation der für jede dieser Komponenten implizit definierten Zugriffskontrolliste benötigt werden. Auf diese implizit definierte Zugriffskontrolliste und die beiden Operationen *list_acl* und *change_acl*, die zu ihrer Verwaltung zur Verfügung stehen, wird an dieser Stelle nicht weiter eingegangen; sie werden in Abschnitt 4.4.3 erklärt. Die Rechte zur Ausführung der Operationen *list_acl* und *change_acl* auf einer zugriffskontrollierten Komponente können explizit vergeben werden.

Tabelle 4.1 faßt zusammen, welche Rechte an zugriffskontrollierten Komponenten vergeben werden können und ob diese implizit oder explizit vergeben werden. Die in der Tabelle verwendete Abkürzung 'zk' steht für 'zugriffskontrolliert'.

Komponentenart	Rechte	Rechtevergabe
explizit zk DA–Generator	*erzeuge*	explizit
	list_acl	explizit
	change_acl	explizit
implizit zk DA–Generator	*erzeuge*	explizit
zk S–Order	*führe_aus*	implizit
zk K–Order	*führe_aus*	implizit
zk M–Akteur	*starte*	implizit
zk K–Akteur	*starte*	implizit
	Kommunikationsoperationen	explizit
	list_acl	explizit
	change_acl	explizit
zk Depot	*initialisiere*	implizit
	Zugriffsoperationen definiert durch lokale DA–Generatoren	explizit
	list_acl	explizit
	change_acl	explizit

Tabelle 4.1: Rechte an zugriffskontrollierten Komponenten

4.4.2 Views

Nach dem oben Gesagten sind die Rechte, die für die Nutzung zugriffskontrollierter Komponenten vergeben werden können, Rechte zur Ausführung äußerer Operationen dieser Komponenten. Für die Vergabe dieser Rechte ist es häufig sinnvoll, die äußeren Operationen einer zugriffskontrollierten Komponente so zu Mengen zusammenzufassen, daß diese Operationenmengen im Hinblick auf die Rechtevergabe als Einheiten zu betrachten sind, und damit die Rechtevergabe zu vereinfachen. Dies wird im folgenden anhand eines Beispiels aus dem bereits angesprochenen Kontenverwaltungssystem einer Bank motiviert.

Beispiel
In dem Kontenverwaltungssystem kann für jedes Konto ein maximaler Überziehungskredit festgelegt werden. Zur Berechnung des maximalen Überziehungskredits eines Kontos werden u.a. der Kontostand des Kontos, die Kontobewegungen innerhalb bestimmter Zeiträume sowie persönliche Daten des Kontoinhabers, wie z.B. regelmäßige Einkünfte, benötigt. Auf den Konten, die als zugriffskontrollierte Depots realisiert werden, sind zur Ermittlung dieser Informationen entsprechende Zugriffsoperationen definiert. Hierzu gehören z.B. die Zugriffsoperationen `LeseKontostand`, `ZeigeKontobewegungen` und `LesePersKundenDaten`. Der Überziehungskredit eines Kontos darf von dem Bankleiter und von dem Kundenbetreuer, der für die Betreuung dieses Kontos zuständig ist, berechnet werden. Der Bankleiter und der entsprechende Kundenbetreuer benötigen also Rechte zur Ausführung der genannten Zugriffsoperationen. Eine Möglichkeit, diese Rechte zu vergeben, besteht darin, das Recht zur Ausführung dieser Zugriffsoperationen für jede Operation einzeln an den Bankleiter und den zuständigen Kundenbetreuer zu vergeben. Dies erfordert jedoch insbesondere im Hinblick darauf, die Rechte an diesen Operationen konsistent zu vergeben, einigen Aufwand. Es ist daher sinnvoll, die Zugriffsoperationen eines Kontos, die für die Berechnung des maximalen Überziehungskredits benötigt werden, zu einer Menge zusammenzufassen, und an den Bankleiter und den zuständigen Kundenbetreuer das Recht an dieser Operationenmenge zu vergeben.

Mit dem Recht an dieser Operationenmenge haben der Bankleiter und der Kundenbetreuer dann das Recht zur Ausführung aller der in der Menge enthaltenen Zugriffsoperationen.

◇

In INSEL$^+$ wird eine Teilmenge der äußeren Operationen einer zugriffskontrollierten Komponente als **View**[3] bezeichnet. Views sind in INSEL$^+$ die Einheiten für die Rechtevergabe. Das Recht an einem View impliziert das Recht zur Ausführung aller äußeren Operationen, die in dem View enthalten sind. Im folgenden werden Views und die Sprachkonstrukte, die zur expliziten Festlegung von Views zur Verfügung stehen, ausführlich erklärt.

Definition 4.5.: Views

Sei x ein zugriffskontrollierter DA–Generator, ein zugriffskontrolliertes Depot oder ein zugriffskontrollierter K-Akteur. Weiter sei $O^E(x)$ die Menge der äußeren Operationen von x, für die Rechte explizit vergeben werden können.

Dann ist für x eine **Menge von Views** – $Views(x)$ – definiert, für die gilt:

$$Views(x) \subseteq POT(O^E(x)) \setminus \{\emptyset\}$$

Ein Element $V \in Views(x)$ heißt **View von** x.

Ist x ein zugriffskontrollierter DA–Generator, so ist die Menge $Views(x)$ implizit wie folgt definiert:

$$Views(x) \triangleq \begin{cases} \{\{erzeuge\}, \{change_acl\}, \{list_acl\}\} & \text{falls } x \text{ explizit zugriffskontrolliert ist} \\ \{\{erzeuge\}\} & \text{falls } x \text{ implizit zugriffskontrolliert ist} \end{cases}$$

□

Gemäß Definition (4.5) sind die Views eines zugriffskontrollierten DA–Generators implizit definiert. Jede auf einem solchen Generator definierte äußere Operation bildet einen eigenen View. Für zugriffskontrollierte Depots und zugriffskontrollierte K-Akteure können Views explizit definiert werden, da dies die Komponenten sind, für die äußere Operationen explizit definiert werden können. Die Views eines zugriffskontrollierten Depots bzw. eines zugriffskontrollierten K-Akteurs sind Klasseneigenschaften. Sie sind also mit dem entsprechenden Depot– bzw. K-Akteur–Generator zu definieren.

Sei x im weiteren ein zugriffskontrolliertes Depot bzw. ein zugriffskontrollierter K-Akteur. Die Menge der Views von x muß gemäß Definition (4.5) keine besonderen Eigenschaften erfüllen. Sie muß insbesondere keine Zerlegung der Menge $O^E(x)$ sein, woraus folgt, daß die für x explizit definierten Views nicht disjunkt sein müssen. Ein explizit definierter View von x muß eine Teilmenge der explizit definierten äußeren Operation von x sein. Für die auf x implizit definierten äußeren Operationen *change_acl* und *list_acl* (siehe Abschnitt 4.4.3) ist implizit jeweils ein eigener View definiert, der lediglich aus dieser Operation besteht. Sind für x keine Views explizit definiert, so bildet jede äußere Operation $op \in O^E(x)$ implizit einen eigenen View.

Die Einheiten für die explizite Vergabe von Rechten zur Ausführung äußerer Operationen einer zugriffskontrollierten Komponente x sind die für x definierten Views. Das Recht an

[3]Festlegung: *der* View, -s, -s

einem View $V \in Views(x)$ impliziert das Recht zur Ausführung aller äußerer Operationen, die zu V gehören. Ein Akteur a hat also genau dann das Recht zur Ausführung einer äußeren Operation op von x, wenn a das Recht an mindestens einem View $V \in Views(x)$ hat, für den gilt: $op \in V$. Bei Aufruf einer äußeren Operation op von x durch den Akteur a muß somit überprüft werden, ob a das Recht an mindestens einem der Views hat, zu denen op gehört. Ist dies der Fall, ist der Zugriff erlaubt und die Operation wird ausgeführt, anderenfalls ist der Zugriff verboten.

Mit der Möglichkeit der expliziten Definition der Views eines zugriffskontrollierten Depots bzw. eines zugriffskontrollierten K-Akteurs lassen sich verschiedene Schnittstellen dieser Komponente festlegen, die spezifischen Benutzungssichten dieser Komponente entsprechen. Ein View kann z.B. rollenspezifisch definiert werden, indem alle äußeren Operationen des Depots bzw. K-Akteurs, die im Rahmen der Erledigung einer Aufgabe, die zur Aufgabenklasse einer bestimmten Rolle gehört, ausgeführt werden müssen, zu einer Menge zusammengefaßt werden. Die Festlegung, daß die Views einer zugriffskontrollierten Komponente nicht disjunkt sein müssen, ermöglicht eine flexible aufgabenangepaßte Definition von Views und entspricht der Tatsache, daß eine äußere Operation einer zugriffskontrollierten Komponente im allgemeinen im Kontext der Erledigung unterschiedlicher Aufgaben ausgeführt werden kann. Neben dieser rollenspezifischen Festlegung von Views kann ein View auch kontextspezifisch definiert werden, indem die äußeren Operationen, die lediglich im Kontext der Ausführung einer anderen Operation ausgeführt werden dürfen, zu einer Menge zusammengefaßt werden.

Beispiel (Fortsetzung)
In dem obigen Beispiel haben der Bankleiter und der Kundenbetreuer lediglich im Kontext der Ausführung der Operation `BerechneKredit` das Recht an dem View, in dem die Zugriffsoperationen eines Kontos, die zur Berechnung des Überziehungskredits benötigt werden, zusammengefaßt sind. Dieser View ist somit kontextspezifisch definiert.

◇

Im folgenden werden nun die Sprachkonstrukte angegeben, mit denen die Views zugriffskontrollierter Depots und zugriffskontrollierter K-Akteure explizit definiert werden können.

Sprachkonstrukte
Die Views, die für die Inkarnationen bzgl. eines zugriffskontrollierten Depot– bzw. K-Akteur–Generators definiert sind, werden im Spezifikationsteil des Generators in einem **View–Part** vereinbart, dessen Syntax wie folgt festgelegt ist:

<view-part>	::=	*<empty>* | `VIEWS` *<view-list>* ;
<view-list>	::=	*<view-definition>* |
		<view-list> ; *<view-definition>*
<view-definition>	::=	*<identifier>* : *<name-list>* |
		<name-list>

Der View–Part eines zugriffskontrollierten Depot– oder K-Akteur–Generators G kann leer sein. In diesem Fall sind für alle Inkarnationen x bzgl. G keine Views explizit definiert. Jede äußere Operation von x bildet dann implizit einen eigenen View, der nur aus dieser Operation besteht und mit dem Namen der Operation bezeichnet wird. Ist der View–Part von G nicht leer, so besteht er aus dem Schlüsselwort `Views` gefolgt von einer Liste von View–Definitionen. Eine View–Definition besteht entweder aus einem Bezeichner und einer Namensliste oder nur aus einer Namensliste. Im ersten Fall wird durch die View–Definition ein View definiert, der mit dem in der View–Definition angegebenen Bezeichner bezeichnet wird und die äußeren Operationen enthält, deren Name in der Namensliste enthalten ist. Als Namen sind in der

Namensliste die Namen der DA–Generatoren erlaubt, die im Spezifikationsteil des Generators G definiert sind, da diese Generatoren die explizit definierten äußeren Operation der Inkarnationen bzgl. G festlegen. Besteht eine View–Definition nur aus einer solchen Namensliste, so bildet jede in der Liste enthaltene äußere Operation einen eigenen View, der nur aus dieser einen Operation besteht und mit dem Namen der Operation bezeichnet wird. Ist x eine Inkarnation bzgl. G, so sind die in dem View–Part von G definierten Views die Einheiten für die Vergabe von Rechten an x. Für äußere Operationen von x, die keinen eigenen View definieren, können somit keine Rechte operationsspezifisch vergeben werden. Soll dies möglich sein, so ist der Name dieser Operation in einer View–Definition von G, die nur aus einer Namensliste besteht, aufzuführen.

Beispiel (Fortsetzung)
Als Beispiel für einen View–Part ist in Abbildung 4.7 der View–Part des zugriffskontrollierten Depot–Generators `KontoTyp`, der der Generator für die von dem Kontenverwaltungssystem zu verwaltenden Konten ist, angegeben. In dem View `KreditBerechnung` sind die Zugriffsoperationen eines Konto–Depots zusammengefaßt, die für die Berechnung des maximalen Überziehungskredits des Kontos benötigt werden. Die nach der Definition des Views `KreditBerechnung` angegebene Namensliste enthält die Bezeichner der Zugriffsoperationen eines Konto–Depots, die jeweils einen eigenen View bilden.

```
-- Zugriffskontrollierter Depot-Generator fuer die Bankkonten
PROTECTED DEPOT TYPE SPEC KontoTyp (...)
     ...
IS
  -- Zugriffsoperationen
  FUNCTION  TYPE SPEC LeseKontostand  RETURN real;
  PROCEDURE TYPE SPEC Einzahlen (Zweck: IN string; Betrag: IN real);
  PROCEDURE TYPE SPEC Abheben   (Zweck: IN string; Betrag: IN real;
                                 Flag: IN boolean; Error: OUT ErrorType);
  FUNCTION  TYPE SPEC LeseKontonummer RETURN integer;
  FUNCTION  TYPE SPEC LeseAllgKundenDaten RETURN AllgKundenDatenTyp;
  FUNCTION  TYPE SPEC LesePersKundenDaten RETURN PersKundenDatenTyp;
  PROCEDURE TYPE SPEC AendereKundenDaten (NeueKundenDaten: IN KundenDatenPtrTyp);
  PROCEDURE TYPE SPEC ZeigeKontoauszug (Term: IN TermPointer);
  PROCEDURE TYPE SPEC ZeigeKontobewegungen (Von, Bis: IN DatumTyp;
                                            Term: IN TermPointer);
  PROCEDURE TYPE SPEC SetzeMaxKredit (NeuerMaxKredit: IN real);
  FUNCTION  TYPE SPEC LeseMaxKredit RETURN real;
  FUNCTION  TYPE SPEC Aufloesen RETURN ErrorType;

  -- View-Part
  VIEWS
    KreditBerechnung: LeseKontostand, ZeigeKontobewegungen, LeseMaxKredit,
                      LeseAllgKundenDaten, LesePersKundenDaten;
    LeseKontostand, Einzahlen, Abheben, LeseKontonummer, LeseAllgKundenDaten,
    AendereKundenDaten, ZeigeKontoauszug, SetzeMaxKredit, Aufloesen;
  ...
END KontoTyp;
```

Abbildung 4.7: Beispiel für einen View–Part

Ist k eine Inkarnation bzgl. des Generators `KontoTyp`, so gilt für die Menge der Views von k:

$$\begin{aligned} Views(k) = \; & \{\mathit{KreditBerechnung}, \mathit{LeseKontostand}, \mathit{Einzahlen}, \mathit{Abheben}, \mathit{LeseKontonummer}, \\ & \mathit{LeseAllgKundenDaten}, \mathit{AendereKundenDaten}, \mathit{SetzeMaxKredit}, \mathit{Aufloesen}, \\ & \mathit{ZeigeKontoauszug}, \mathit{ChangeACL}, \mathit{ListACL}\} \end{aligned}$$

mit

$$\begin{aligned} \mathit{KreditBerechnung} &= \{\mathit{LeseKontostand}, \mathit{ZeigeKontobewegungen}, \mathit{LeseMaxKredit}, \\ & \quad \mathit{LeseAllgKundenDaten}, \mathit{LesePersKundenDaten}\} \\ \mathit{LeseKontostand} &= \{\mathit{LeseKontostand}\} \\ \mathit{Einzahlen} &= \{\mathit{Einzahlen}\} \\ \mathit{Abheben} &= \{\mathit{Abheben}\} \\ \mathit{LeseKontonummer} &= \{\mathit{LeseKontonummer}\} \\ \mathit{LeseAllgKundenDaten} &= \{\mathit{LeseAllgKundenDaten}\} \\ \mathit{AendereKundenDaten} &= \{\mathit{AendereKundenDaten}\} \\ \mathit{SetzeMaxKredit} &= \{\mathit{SetzeMaxKredit}\} \\ \mathit{Aufloesen} &= \{\mathit{Aufloesen}\} \\ \mathit{ZeigeKontoauszug} &= \{\mathit{ZeigeKontoauszug}\} \\ \mathit{ChangeACL} &= \{\mathit{ChangeACL}\} \\ \mathit{ListACL} &= \{\mathit{ListACL}\} \end{aligned}$$

◇

Zum Abschluß dieses Abschnitts wird der erklärte View–Begriff von INSEL$^+$ kurz mit sonstigen aus der Literatur bekannten View–Begriffen verglichen.

Literatureinordnung
Der View–Begriff von INSEL$^+$ ist vergleichbar mit dem View–Begriff, so wie er in [Hag94] und [CD94] verwendet wird. Dort wird unter einem View eine Teilmenge der auf einem Objekt definierten Operationen (Methoden) verstanden. Sowohl in [Hag94] als auch in [CD94] sind Views die Einheiten für die Rechtevergabe. In [Hag94] wird in der Zugriffskontrolliste eines Objekts jedem Benutzer ein View des Objekts zugeordnet. Ein Benutzer hat dann das Recht zur Ausführung der Operationen auf dem Objekt, die in dem ihm zugeordneten View enthalten sind. In [CD94] werden Views als Einträge in Zugriffskontroll–Matrizen verwendet. Aufgrund des beiden Arbeiten zugrundeliegenden Zugriffsmatrix–Modells ist das Recht an einem View lediglich von einem entsprechenden Eintrag in der Zugriffskontroll–Matrix bzw. der Zugriffskontrolliste abhängig. Komplexe Zugriffsbeschränkungen für Views, wie sie in INSEL$^+$ formuliert werden können (siehe dazu Abschnitt 4.4.4), sind nicht möglich.

Die in [HO90] eingeführten Views unterscheiden sich von den INSEL$^+$ Views darin, daß ein View gemäß [HO90] neben einer Teilmenge der auf einem Objekt definierten Operationen zusätzlich aus der Menge von Objekten besteht, die diese Operationen auf dem Objekt aufrufen dürfen. Ein solcher View definiert also nicht nur eine spezielle Schnittstelle eines Objekts, sondern legt gleichzeitig fest, welche Objekte diese Schnittstelle nutzen dürfen. Auch hier können wiederum keine komplexen Zugriffsbeschränkungen für Views festgelegt werden. In INSEL$^+$ wird deshalb die Definition verschiedener Schnittstellen eines Objekts und die Festlegung von Zugriffsbeschränkungen für diese bewußt getrennt.

Außer in den zitierten Arbeiten aus dem Bereich objekt–orientierter Sprachen und Systeme wird der Begriff *View* seit langem auf dem Gebiet der Datenbank–Systeme verwendet. Dort wird unter einem View im allgemeinen eine virtuelle Relation verstanden, die einen

spezifischen Ausschnitt aus der gesamten Datenbank zeigt (siehe z.B. [KS91], [KE96]). Die Relation ist virtuell in dem Sinn, daß sie nicht in der Datenbank abgespeichert ist, sondern bei jeder Anfrage, in der sie enthalten ist, neu berechnet werden muß. Im Kontext der Sicherheit von Datenbank–Systemen sind die für eine Datenbank definierten Views in der Regel die Einheiten, an denen Rechte vergeben werden. Dies gilt sowohl für den Bereich der benutzerbestimmten Zugriffskontrollen (z.B. [GW76], [SS95]) als auch für den Bereich der systembestimmten Zugriffskontrollen und hier speziell für Multilevel sichere Datenbank–Systeme (z.B. [DAH+87], [Wil88], [Qui96]). Die Views aus dem Bereich der Datenbank–Systeme sind also in dem Sinn, daß sie die Einheiten der Rechtevergabe sind, mit den in dieser Arbeit eingeführten Views vergleichbar.

4.4.3 Zugriffskontrollisten

Als Basis für die Konstruktion zugriffskontrollierter Komponenten, für deren Nutzung Rechte durch Benutzer dynamisch vergeben werden sollen, ist in INSEL$^+$ für jeden explizit zugriffskontrollierten DA–Generator, jedes zugriffskontrollierte Depot sowie jeden zugriffskontrollierten K-Akteur implizit eine Zugriffskontrolliste definiert. Diese Zugriffskontrolliste wird mit Erzeugung der Komponente implizit erzeugt, wobei sie explizit initialisiert werden kann. Wird die Zugriffskontrolliste nicht explizit initialisiert, so enthält sie zunächst keine Einträge. Um die Zugriffskontrolliste einer zugriffskontrollierten Komponente ändern zu können, ist auf jeder zugriffskontrollierten Komponente implizit die äußere Operation *change_acl* definiert.

Zur Vermeidung von Mißverständnissen sei ausdrücklich darauf hingewiesen, daß in INSEL$^+$ weder explizite Konzepte zur Definition bzw. Erzeugung von Zugriffskontrollisten noch vordefinierte Generatoren für Zugriffskontrollisten zur Verfügung stehen. Es stehen lediglich Sprachkonstrukte zur Verfügung, mit denen festgelegt werden kann, wie die implizit definierte Zugriffskontrolliste einer Komponente bei ihrer Erzeugung zu initialisieren ist.

Die Rechte an einem View einer zugriffskontrollierten Komponente können abhängig von dem Vorhandensein bestimmter Einträge in den Zugriffskontrollisten vergeben werden. Dazu steht das Zugriffsrestriktionsprädikat `IN_ACL` zur Verfügung, das in Zugriffsrestriktionsausdrücken verwendet werden kann. Auf das `IN_ACL` Prädikat und damit die Semantik der Auswertung von Zugriffskontrollisten wird in diesem Abschnitt nicht weiter eingegangen. Dies wird in Abschnitt 4.4.4 in Zusammenhang mit der Erklärung der Zugriffsrestriktionsausdrücke erläutert. In diesem Abschnitt wird zunächst der Aufbau der Zugriffskontrollisten von INSEL$^+$ beschrieben. Anschließend werden die Sprachkonstrukte eingeführt, die zur Initialisierung von Zugriffskontrollisten zur Verfügung stehen, und am Ende dieses Abschnitts wird die Semantik der Operation *change_acl* angegeben.

Aufbau der Zugriffskontrollisten

Die Zugriffskontrolliste einer zugriffskontrollierten Komponente kann für jeden View dieser Komponente eine Menge von Tripeln enthalten, die jeweils aus einem Benutzeridentifikator, einem Rollenidentifikator und der Angabe, ob es sich um einen positiven oder negativen Eintrag handelt, bestehen. Dies wird in der folgenden Definition präzisiert.

Definition 4.6.: Zugriffskontrollisten

Sei x ein explizit zugriffskontrollierter DA–Generator, ein zugriffskontrolliertes Depot oder ein zugriffskontrollierter K-Akteur eines INSEL$^+$–Systems $\mathcal{S}$. Für x ist implizit

eine Zugriffskontrolliste definiert, die mit $acl(x)$ bezeichnet wird. Die **Zugriffskontrolliste** $acl(x)$ ist eine Menge, die entweder leer ist oder aus Elementen der Form $(V, Entries(V))$ besteht, mit:

- $V \in Views(x)$ und
- $Entries(V) \subset \mathcal{B} \cup \{-1\} \times \mathcal{R} \cup \{-1\} \times \{+,-\}$.

Dabei ist $\mathcal{B}$ die Menge der Benutzeridentifikatoren und $\mathcal{R}$ die Menge der Rollenidentifikatoren von $\mathcal{S}$ (vgl. Abschnitt 4.2.2).

Für die Zugriffskontrolliste $acl(x)$ muß gelten:

1. Für jeden View $V \in Views(x)$ gibt es höchstens ein Element $(V', Entries(V')) \in acl(x)$ mit: $V = V'$.
2. Für alle Elemente $(V, Entries(V)) \in acl(x)$ gilt, daß die Menge $Entries(V)$ keine zwei Elemente (B_1, R_1, M_1) und (B_2, R_2, M_2) enthält mit: $B_1 = B_2$, $R_1 = R_2$ und $M_1 = +$ und $M_2 = -$.

□

Sei im weiteren $acl(x)$ die Zugriffskontrolliste einer zugriffskontrollierten Komponente x und $(V, Entries(V))$ ein Element aus $acl(x)$. Dann wird $(V, Entries(V))$ als **Zugriffskontrollisteneintrag** des Views V und $Entries(V)$ als **Benutzer–Rollen–Liste** des Views V bezeichnet. Gemäß Bedingung 1 aus Definition (4.6) kann die Zugriffskontrolliste $acl(x)$ für jeden View $V \in Views(x)$ höchstens einen Zugriffskontrollisteneintrag enthalten. Die Benutzer–Rollen–Liste eines Views V besteht aus Tripeln (B, R, M) mit:

- $B \in \mathcal{B}$ ist ein Benutzeridentifikator oder $B = -1$
- $R \in \mathcal{R}$ ist ein Rollenidentifikator oder $R = -1$
- $M \in \{+,-\}$ ist der Modus, der angibt, ob es sich um einen positiven oder negativen Eintrag für das Benutzer–Rollen–Paar (B, R) handelt.

Der Wert -1 steht dabei im Fall $B = -1$ stellvertretend für alle Benutzer und im Fall $R = -1$ stellvertretend für alle Rollen des Systems. Die Bedingung 2 aus Definition (4.6) besagt, daß in der Benutzer–Rollen–Liste eines Views für jedes Benutzer–Rollen–Paar (B, R) höchstens ein Element (B, R, M) enthalten sein darf. Durch diese Bedingung sind widersprüchliche Festlegungen, die sich aus dem Vorhandensein eines positiven und eines negativen Eintrags für ein Benutzer–Rollen–Paar ergeben würden, per Definition ausgeschlossen.

Die Elemente der Benutzer–Rollen–Liste eines Views werden abhängig von den Werten ihrer Komponenten unterschiedlich bezeichnet. Sei $(B, R, M) \in Entries(V)$.

- (B, R, M) ist ein positiver bzw. negativer **Benutzer–Rollen–Eintrag** falls gilt:
  $B \neq -1$ und $R \neq -1$ und $M = +$ bzw. $M = -$
- (B, R, M) ist ein positiver bzw. negativer **Benutzer–Eintrag** falls gilt:
  $B \neq -1$ und $R = -1$ und $M = +$ bzw. $M = -$
- (B, R, M) ist ein positiver bzw. negativer **Rollen–Eintrag** falls gilt:
  $B = -1$ und $R \neq -1$ und $M = +$ bzw. $M = -$

- (B, R, M) ist ein positiver bzw. negativer **Welt–Eintrag** falls gilt:
  $B = -1$ und $R = -1$ und $M = +$ bzw. $M = -$

Initialisierung der Zugriffskontrollisten

Die Zugriffskontrolliste $acl(x)$ eines zugriffskontrollierten Depots, eines zugriffskontrollierten K-Akteurs bzw. eines explizit zugriffskontrollierten DA–Generators x wird implizit mit der Erzeugung von x erzeugt. Mit welchen Einträgen die Zugriffskontrolliste $acl(x)$ bei ihrer Erzeugung zu initialisieren ist, kann explizit festgelegt werden. Dazu stehen in INSEL$^+$ entsprechende Sprachkonstrukte zur Verfügung, die im folgenden erklärt werden. Sind für die Initialisierung einer Zugriffskontrolliste keine expliziten Festlegungen getroffen, so wird eine leere Zugriffskontrolliste erzeugt.

Wie die Zugriffskontrolliste $acl(x)$ eines explizit zugriffskontrollierten DA–Generators x zu initialisieren ist, ist mit x zu definieren. Ist x ein zugriffskontrolliertes Depot oder ein zugriffskontrollierter K-Akteur, so ist die Initialisierung der Zugriffskontrolliste $acl(x)$ mit dem Generator von x festzulegen. Für zugriffskontrollierte Depots und zugriffskontrollierte K-Akteure ist die Initialisierung ihrer Zugriffskontrolliste somit eine Klasseneigenschaft. Wie noch zu sehen sein wird, bedeutet dies jedoch nicht, daß die Zugriffskontrollisten aller Inkarnationen bzgl. eines zugriffskontrollierten Depot– bzw. K-Akteur–Generators mit den gleichen Werten initialisiert werden. Differenzierungen sind u.a. über die Verwendung von Parametern und globalen Variablen möglich.

Sprachkonstrukte

Die Initialisierung der Zugriffskontrolliste eines explizit zugriffskontrollierten DA–Generators G und zusätzlich die Initialisierung der Zugriffskontrolliste von Inkarnationen bzgl. G, falls G ein Depot– oder K-Akteur–Generator ist, wird in dem Spezifikationsteil von G in einem **ACL–Part** angegeben, dessen Syntax wie folgt festgelegt ist:

<acl-part>	::=	*<empty>* | `ACL` *<acl-list>* ;
<acl-list>	::=	*<acl-entry>* |
		<acl-list> ; *<acl-entry>*
<acl-entry>	::=	*<view-name>* : *<user-role-list>*
<user-role-list>	::=	*<user-role-entry>* |
		<user-role-list> , *<user-role-entry>*
<user-role-entry>	::=	(*<user-identifier>* , *<role-identifier>* , *<mode>*)
<mode>	::=	+ | −

Der ACL–Part des Generators G kann leer sein. In diesem Fall sind keine expliziten Festlegungen für die Initialisierung der entsprechenden Zugriffskontrollisten getroffen, was nach dem oben Gesagten bedeutet, daß jeweils eine leere Zugriffskontrolliste erzeugt wird. Ist der ACL–Part von G nicht leer, so besteht er aus einer Liste von Zugriffskontrollisteneinträgen (*<acl-entry>*). Ein Zugriffskontrollisteneintrag besteht aus einem View–Namen (*<view-name>*) und einer Benutzer–Rollen–Liste (*<user-role-list>*). Ist G ein explizit zugriffskontrollierter S-Order– oder M-Akteur–Generator, so sind als View–Namen in dem ACL–Part von G die Namen der gemäß Definition (4.5) implizit für G definierten Views erlaubt. Dabei bezeichnet:

- `Create` den View, der aus der implizit für G definierten äußeren Operation *erzeuge* besteht,
- `ChangeGenACL` den View, der aus der implizit für G definierten äußeren Operation *change_acl* besteht und

– `ListGenACL` den View, der aus der implizit für G definierten äußeren Operation *list_acl* besteht.

Ist G ein explizit zugriffskontrollierter Depot– oder K-Akteur–Generator, so können in dem ACL–Part von G neben den drei genannten View–Namen zusätzlich die Namen der Views auftreten, die für die Inkarnationen bzgl. G definiert sind. Enthält G einen nicht leeren View–Part, so sind dies die Bezeichner der im View–Part explizit definierten Views. Ist der View–Part von G leer, so definiert laut Abschnitt 4.4.2 jede auf einer Inkarnation bzgl. G explizit definierte äußere Operation einen eigenen View, der mit dem Namen dieser Operation bezeichnet wird. Dementsprechend können in dem ACL–Part von G als View–Namen die Namen der im Spezifikationsteil von G definieren DA–Generatoren auftreten. Weiterhin sind in dem ACL–Part eines zugriffskontrollierten Depot– oder K-Akteur–Generators G die Bezeichner `ChangeACL` und `ListACL` als View–Namen erlaubt. `ChangeACL` bezeichnet den für jedes zugriffskontrollierte Depot– bzw. jeden zugriffskontrollierten K-Akteur implizit definierten View, der aus der Operation *change_acl* besteht, und `ListACL` bezeichnet entsprechend den View, der die Operation *list_acl* enthält.

Die Benutzer–Rollen–Liste eines Zugriffskontrolllisteneintrags besteht aus einer Folge von Benutzer–Rollen–Einträgen (<*user-role-entry*>). Das erste Element eines Benutzer–Rollen–Eintrags (<*user-identifier*>) muß -1 oder der Bezeichner einer DE–Inkarnation sein, die Inkarnation bzgl. des in INSEL+ für Benutzeridentifikatoren vordefinierten Generators `UserUidType` ist[4]. Das zweite Element (<*role-identifier*>) muß entweder -1, der Bezeichner einer im Role–Part des INSEL+–Programms definierten Rolle oder der Bezeichner einer DE–Inkarnation bzgl. des für Rollenidentifikatoren vordefinierten Generators `RoleUidType` sein[5]. Das dritte Element (<*Mode*>) legt fest, ob es sich um einem positiven Eintrag (+) oder um einen negativen Eintrag handelt (–).

Um die in Definition (4.6) mit den Bedingungen 1. und 2. festgelegten Eigenschaften einer Zugriffskontrollliste zu gewährleisten, darf in einem ACL–Part jeder dort mögliche View–Name höchstens einmal auftreten. Zum anderen dürfen in der Benutzer–Rollen–Liste des Zugriffskontrolllisteneintrags eines View–Namens keine zwei Elemente mit jeweils textuell gleichem Benutzer– und textuell gleichem Rollenidentifikator enthalten sein. Die Erfüllung dieser Bedingungen wird im Rahmen der Übersetzung eines INSEL+–Programms überprüft.

Beispiel
In Abbildung 4.8 ist ein Ausschnitt aus dem ACL–Part des bereits aus den vorhergehenden Abschnitten bekannten zugriffskontrollierten Depot–Generators `KontoTyp` des Kontenverwaltungssystems angegeben. In den Benutzer–Rollen–Listen des ACL–Parts werden die Parameter `InhaberUid` und `BetreuerUid` des Generators `KontoTyp` verwendet. Bei der Erzeugung einer Inkarnation bzgl. `KontoTyp` wird der Parameter `InhaberUid` mit der Uid des Inhabers des zu erzeugenden Kontos belegt, und der Parameter `BetreuerUid` wird mit der Uid des für dieses Konto zuständigen Kundenbetreuers belegt. `Kunde`, `Kundenbetreuer`, `Kassierer` und `Bankleiter` sind Bezeichner von Rollen, die für das Kontenverwaltungssystem definiert sind.

Im weiteren wird der erste Zugriffskontrolllisteneintrag des ACL–Parts betrachtet. Dieser legt fest, wie die Benutzer–Rollen–Liste des Views `LeseKontostand` zu initialisieren ist. Bei der Erzeugung einer Inkarnation x bzgl. des Generators `KontoTyp` wird in der Zugriffskontrollliste $acl(x)$ für den View `LeseKontostand` eine Benutzer–Rollen–Liste mit zwei positiven

[4]Hier können somit insbesondere die in Abschnitt 4.2.2 eingeführten Bezeichner `System` für den vordefinierten Benutzer *System* und `Owner` für den Besitzer einer Komponente verwendet werden.

[5]Hierzu gehört insbesondere der in Abschnitt 4.2.2 eingeführte Bezeichner `SystemRole` für die vordefinierte Rolle *SystemRole*.

```
-- Zugriffskontrollierter Depot-Generator fuer die Bankkonten
PROTECTED DEPOT TYPE SPEC KontoTyp (InhaberUid   : IN UserUidType;
                                    BetreuerUid  : IN UserUidType; ...)
     ...
IS
  -- Zugriffsoperationen
  FUNCTION  TYPE SPEC LeseKontostand  RETURN real;
  PROCEDURE TYPE SPEC Einzahlen (Zweck: IN string; Betrag: IN real);
  PROCEDURE TYPE SPEC Abheben   (Zweck: IN string; Betrag: IN real; ...);
  ...

  -- View-Part
  VIEWS
    KreditBerechnung: LeseKontostand, ZeigeKontobewegungen, LeseMaxKredit,
                      LeseAllgKundenDaten, LesePersKundenDaten;
    LeseKontostand, Einzahlen, Abheben, ... ;

  -- ACL-Part
  ACL
    LeseKontostand   : (InhaberUid,Kunde,+), (BetreuerUid,Kundenbetreuer,+);
    Abheben          : (InhaberUid,Kunde,+), (BetreuerUid,Kundenbetreuer,+),
                       (-1,Kassierer,+);
    KreditBerechnung : (BetreuerUid,Kundenbetreuer,+);
    ...;
  ...
END KontoTyp;
```

Abbildung 4.8: Beispiel für einen ACL–Part

Benutzer–Rollen–Einträgen $(B_1, R_1, +)$ und $(B_2, R_2, +)$ erzeugt, wobei gilt: B_1 ist der Wert des Parameters `InhaberUid`; B_2 ist Wert des Parameters `BetreuerUid`, R_1 ist der Identifikator der Rolle `Kunde` und R_2 ist der Identifikator der Rolle `Kundenbetreuer`. Analog werden Einträge für die Views `Abheben` und `Kreditberechnung` erzeugt. Für den Depot–Generator `KontoTyp` selbst wird eine leere Zugriffskontrolliste erzeugt, da in seinem ACL–Part keine Zugriffskontrollisteneinträge für die Views `Create`, `ChangeGenACL` und `ListGenACL` enthalten sind.

◇

Änderung von Zugriffskontrollisten

Um die Einträge von Zugriffskontrollisten ändern zu können, ist auf jedem explizit zugriffskontrollierten DA–Generator, jedem zugriffskontrollierten Depot und jedem zugriffskontrollierten K-Akteur implizit die äußere Operation *change_acl* definiert. Ist x eine zugriffskontrollierte Komponente und $acl(x)$ die Zugriffskontrolliste von x, so kann durch Aufruf der Operation *change_acl* auf x ein Benutzer–Rollen–Eintrag in die Benutzer–Rollen–Liste eines für x definierten Views aufgenommen werden bzw. aus dieser Liste gelöscht werden. Das Recht zur Ausführung der Operation *change_acl* ist dabei explizit zu vergeben.

Im folgenden wird zunächst der Kopf der Operation *change_acl* in INSEL$^+$–Syntax angegeben. Anschließend wird die Semantik der Operation *change_acl* informell erklärt. In

INSEL$^+$ wird die Operation *change_acl* für einen explizit zugriffskontrollierten DA–Generator mit `ChangeGenACL` und für ein zugriffskontrolliertes Depot bzw. einen zugriffskontrollierten K-Akteur mit `ChangeACL` bezeichnet. Da die Unterscheidung des Operationsnamens von *change_acl* für Generatoren und Depots bzw. K-Akteure in INSEL$^+$ rein syntaktischer Natur ist, wird im folgenden lediglich der Kopf von `ChangeACL` angegeben.

```
ChangeACL(ViewName:IN string; UserId:IN UserUidType; RoleId:IN RoleUidType;
          Mode    :IN boolean; AddOrDelete:IN boolean; Error:OUT ErrorType)
```

Der Parameter `ViewName` gibt den View an, dessen Benutzer–Rollen–Liste geändert werden soll. Mit den Parametern `UserId`, `RoleId` und `Mode` wird der Benutzer–Rollen–Eintrag festgelegt, der abhängig von dem Wert des Parameters `AddOrDelete` in die Benutzer–Rollen–Liste des Views eingefügt (Wert von `AddOrDelete` gleich *true*) oder aus dieser Liste gelöscht (Wert von `AddOrDelete` gleich *false*) werden soll. Über den Parameter `Error` wird der Ergebniswert einer `ChangeACL`–Ausführung an den jeweiligen Aufrufer übergeben.

Für die Erklärung der informellen Semantik der Operation *change_acl* sei x eine zugriffskontrollierte Komponente und *change_acl(ViewName, UserId, RoleId, Mode, AddOrDelete, Error)* ein *change_acl*–Aufruf auf x. Die Werte der aktuellen Parameter *ViewName*, *UserId*, usw. seien mit $\omega(\mathit{ViewName})$, $\omega(\mathit{UserId})$, usw. bezeichnet. Dabei gilt: $\omega(\mathit{UserId}) \in \mathcal{B} \cup \{-1\}$, $\omega(\mathit{RoleId}) \in \mathcal{R} \cup \{-1\}$, $\omega(\mathit{Mode}) \in \{\mathit{true}, \mathit{false}\}$ und $\omega(\mathit{AddOrDelete}) \in \{\mathit{true}, \mathit{false}\}$. Der Zustand der Zugriffskontrolliste von x bei Beginn der Ausführung von *change_acl* wird mit $\mathit{acl}(x)$ bezeichnet und ihr Zustand nach Ausführung von *change_acl* mit $\mathit{acl}'(x)$.
Die Ausführung des *change_acl*–Aufrufs auf x bewirkt folgendes:

(a) Zunächst wird überprüft, ob $\omega(\mathit{ViewName})$ der Name eines für x definierten Views ist, d.h. ob gilt: $\omega(\mathit{ViewName}) \in Views(x)$. Ist dies nicht der Fall, bleibt die Zugriffskontrolliste unverändert:
$$\mathit{acl}'(x) = \mathit{acl}(x) \text{ und } \omega(\mathit{Error}) = 1$$

(b) Gilt $\omega(\mathit{ViewName}) \in Views(x)$, so wird überprüft, ob in der Zugriffskontrolliste $\mathit{acl}(x)$ bereits ein Zugriffskontrollisteneintrag für den View $\omega(\mathit{ViewName})$ existiert.

1. Es gibt ein Element $(V, \mathit{Entries}(V)) \in \mathit{acl}(x)$ mit $V = \omega(\mathit{ViewName})$
   In diesem Fall wird weiter überprüft, ob in der Benutzer–Rollen–Liste $Entries(V)$ von V bereits ein Element (B, R, M) mit $B = \omega(\mathit{UserId})$ und $R = \omega(\mathit{RoleId})$ enthalten ist.

   1.1 Es gibt ein $(B, R, M) \in Entries(V)$ mit $B = \omega(\mathit{UserId})$ und $R = \omega(\mathit{RoleId})$
   Dann ist zu unterscheiden, ob (B, R, M) ein positiver oder negativer Eintrag ist:

   1.1.1 $M = +$
   In diesem Fall hat die Ausführung von *change_acl* abhängig von den Werten von *AddOrDelete* und *Mode* folgende Wirkung:

   1.1.1.1 $\omega(\mathit{AddOrDelete}) = true$ und $\omega(Mode) = true$:
   $\mathit{acl}'(x) = \mathit{acl}(x)$ und $\omega(\mathit{Error}) = 2$

   1.1.1.2 $\omega(\mathit{AddOrDelete}) = true$ und $\omega(Mode) = false$:
   $\mathit{Entries}'(V) = \mathit{Entries}(V) \setminus \{(B, R, +)\} \cup \{(B, R, -)\}$ und $\omega(\mathit{Error}) = 3$

   1.1.1.3 $\omega(\mathit{AddOrDelete}) = false$ und $\omega(Mode) = true$:
   $\mathit{Entries}'(V) = \mathit{Entries}(V) \setminus \{(B, R, +)\}$ und $\omega(\mathit{Error}) = 0$

1.1.1.4 $\omega(AddOrDelete) = false$ und $\omega(Mode) = false$:
$acl'(x) = acl(x)$ und $\omega(Error) = 4$

1.1.2 $M = -$
In diesem Fall hat die Ausführung von *change_acl* abhängig von den Werten von *AddOrDelete* und *Mode* folgende Wirkung:

1.1.2.1 $\omega(AddOrDelete) = true$ und $\omega(Mode) = true$:
$Entries'(V) = Entries(V) \backslash \{(B, R, -)\} \cup \{(B, R, +)\}$ und $\omega(Error) = 5$

1.1.2.2 $\omega(AddOrDelete) = true$ und $\omega(Mode) = false$:
$acl'(x) = acl(x)$ und $\omega(Error) = 6$

1.1.2.3 $\omega(AddOrDelete) = false$ und $\omega(Mode) = true$:
$acl'(x) = acl(x)$ und $\omega(Error) = 7$

1.1.2.4 $\omega(AddOrDelete) = false$ und $\omega(Mode) = false$:
$Entries'(V) = Entries(V) \setminus \{(B, R, -)\}$ und $\omega(Error) = 0$

1.2 Es gibt kein $(B, R, M) \in Entries(V)$ mit $B = \omega(UserId)$ und $R = \omega(RoleId)$
Dann hat die Ausführung von *change_acl* abhängig von den Werten von *AddOrDelete* und *Mode* folgende Wirkung:

1.2.1 $\omega(AddOrDelete) = true$ und $\omega(Mode) = true$:
$Entries'(V) = Entries(V) \cup \{(\omega(UserId), \omega(RoleId), +)\}$ und
$\omega(Error) = 0$

1.2.2 $\omega(AddOrDelete) = true$ und $\omega(Mode) = false$:
$Entries'(V) = Entries(V) \cup \{(\omega(UserId), \omega(RoleId), -)\}$ und
$\omega(Error) = 0$

1.2.3 $\omega(AddOrDelete) = false$:
$acl'(x) = acl(x)$ und $\omega(Error) = 8$

2. Es gibt kein Element $(V, Entries(V)) \in acl(x)$ mit $V = \omega(ViewName)$
Dann hat die Ausführung von *change_acl* abhängig von den Werten von *AddOrDelete* und *Mode* folgende Wirkung:

2.1 $\omega(AddOrDelete) = true$ und $\omega(Mode) = true$:
$acl'(x) = acl(x) \cup \{(\omega(ViewName), \{(\omega(UserId), \omega(RoleId), +)\})\}$ und
$\omega(Error) = 0$

2.2 $\omega(AddOrDelete) = true$ und $\omega(Mode) = false$:
$acl'(x) = acl(x) \cup \{(\omega(ViewName), \{(\omega(UserId), \omega(RoleId), -)\})\}$ und
$\omega(Error) = 0$

2.3 $\omega(AddOrDelete) = false$:
$acl'(x) = acl(x)$ und $\omega(Error) = 9$

Die Funktionalität der Operation *change_acl* gewährleistet, daß in der Zugriffskontrolliste $acl(x)$ für jeden View V von x höchstens ein Zugriffskontrollisteneintrag existiert und daß in der Benutzer–Rollen–Liste von V für jedes Benutzer–Rollen–Paar (B, R) höchstens ein Element (B, R, M) enthalten ist. Damit werden die in Definition (4.6) mit den Bedingungen 1. und 2. festgelegten Eigenschaften einer Zugriffskontrolliste erfüllt. Insbesondere wird in dem Fall, daß für ein Benutzer–Rollen–Paar (B, R) ein positiver Eintrag in die Benutzer–Rollen–Liste eines Views eingefügt werden soll, darin jedoch bereits ein negativer Eintrag für (B, R) enthalten ist, der negative Eintrag in einen positiven Eintrag umgewandelt (Fall 1.1.2.1). Analoges gilt für den Fall, daß ein negativer Eintrag eingefügt werden soll und bereits ein entsprechender positiver Eintrag existiert (Fall 1.1.1.2).

Die Realisierung der Funktionalität von *change_acl* ist im Rahmen der Realisierung von INSEL$^+$–Systemen durchzuführen. Sie wird in Kapitel 6 erklärt. Da *change_acl* den Zustand einer Zugriffskontrolliste ändert, ist als Anforderung an die Realisierung gestellt, daß *change_acl*–Ausführungen auf einer zugriffskontrollierten Komponente wechselseitig ausgeschlossen ausgeführt werden.

Beispiel
In Abbildung 4.9 ist ein Beispiel für einen `ChangeACL`–Aufruf angegeben. Der `ChangeACL`–Aufruf ist Bestandteil des Implementierungsteils des zugriffskontrollierten M-Akteur–Generators `Kunde` des Kontenverwaltungssystems. `Kunde` ist der Generator für die Benutzerrepräsentanten, die für Benutzer erzeugt werden, die in der Rolle *Kunde* agieren. `Kontozeiger` ist ein Zeiger, der auf ein zugriffskontrolliertes anonymes Depot zeigt, das Inkarnation bzgl. des in den vorhergehenden Beispielen erläuterten zugriffskontrollierten Depot–Generators `KontoTyp` ist. ◇

```
-- Benutzerrepraesentant fuer Benutzer in der Rolle Kunde
PROTECTED PROCESS TYPE Kunde (...)
  ...
  Kontozeiger : KontoPtrTyp;
  ViewName    : string;
  BenutzerId  : UserUidType;
  Flag        : boolean;
  ActError    : ErrorType:
  ...
BEGIN
  ...
  -- ChangeACL-Aufruf
  Kontozeiger.ChangeACL(ViewName, BenutzerId, "Kunde", TRUE, Flag, ActError)
  ...
END Kunde;
```

Abbildung 4.9: Beispiel für einen `ChangeACL`–Aufruf

Neben der Operation *change_acl* ist auf jedem explizit zugriffskontrollierten DA–Generator, jedem zugriffskontrollierten Depot und jedem zugriffskontrollierten K-Akteur implizit die äußere Operation *list_acl* definiert. Die Ausführung eines Aufrufs der Operation *list_acl* auf einer zugriffskontrollierten Komponente x bewirkt, daß der Inhalt der Zugriffskontrolliste von x auf dem als Parameter des *list_acl*–Aufrufs zu übergebenen abstrakten Terminal ausgegeben wird. Wie für die Operation *change_acl* kann auch das Recht zur Ausführung der Operation *list_acl* explizit vergeben werden. Auf die Operation *list_acl* wird im weiteren nicht näher eingegangen. Sie steht in INSEL$^+$ für explizit zugriffskontrollierte DA–Generatoren unter dem Bezeichner `ListGenACL` und für zugriffskontrollierte Depots und K-Akteure unter dem Bezeichner `ListACL` zur Verfügung.

4.4.4 Zugriffsrestriktionsausdrücke

Die Rechte, die an einer zugriffskontrollierten Komponente eines INSEL$^+$–Systems explizit vergeben werden können, werden auf Basis von Zugriffsrestriktionsausdrücken vergeben, die für die Views dieser Komponente bei der Systemkonstruktion explizit zu definieren sind.

Zugriffsrestriktionsausdrücke sind spezielle Boolesche Ausdrücke, mit denen sich komplexe Zugriffsbeschränkungen für die Nutzung zugriffskontrollierter Komponenten festlegen lassen. Ruft ein Akteur a eine äußere Operation *op* einer zugriffskontrollierten Komponente x auf, so werden zunächst die Zugriffsrestriktionsausdrücke aller Views ausgewertet, zu denen die Operation *op* gehört. Das Ergebnis der Auswertung eines solchen Zugriffsrestriktionsausdrucks kann u.a. von dem Benutzer und der Rolle, die dem Akteur a zugeordnet sind, sowie von dem Kontext, in dem a die Operation *op* aufruft, abhängig sein. Hat mindestens einer dieser Zugriffsrestriktionsausdrücke den Wert *true*, so hat der Akteur a das Recht zur Ausführung der Operation *op*, und die Operation *op* wird ausgeführt. Anderenfalls hat der Akteur a nicht das Recht zur Ausführung von *op*, was zur Konsequenz hat, daß *op* nicht ausgeführt wird. Eine formale Darstellung der Rechte, die in einem INSEL$^+$–System zu einem Zeitpunkt t auf Basis der Zugriffsrestriktionsausdrücke an die zum Zeitpunkt t existierenden Akteure vergeben sind, wird in Kapitel 5 angegeben. Auf die Realisierung der mit den Zugriffsrestriktionsausdrücken festgelegten Zugriffsbeschränkungen eines INSEL$^+$–Systems wird in Abschnitt 6 eingegangen. Als Anforderung an die Realisierung ist zu stellen, daß die Auswertung eines Zugriffsrestriktionsausdrucks atomar in dem Sinne sein muß, daß sich der Wert eines Teilausdrucks während der Auswertung des gesamten Ausdrucks nicht ändern darf.

In diesem Abschnitt wird die Syntax von Zugriffsrestriktionsausdrücken angegeben sowie die Semantik dieser Ausdrücke informell erklärt. Auf eine formale Definition der Zugriffsrestriktionsausdrücke und ihrer Semantik wird hier verzichtet. Aus diesem Grunde werden im folgenden lediglich Bezeichnungen für den Zugriffsrestriktionsausdruck eines Views einer zugriffskontrollierten Komponente und den Wert dieses Ausdrucks zu einem Zeitpunkt t festgelegt.

(4.7) Bezeichnungen für Zugriffsrestriktionsausdrücke

Sei $x \in X_t$ ein explizit zugriffskontrollierter DA–Generator, ein zugriffskontrolliertes Depot oder ein zugriffskontrollierter K-Akteur eines INSEL$^+$–Systems $\mathcal{S}$ zu einem Zeitpunkt $t \in \Lambda(\mathcal{S})$. Weiter sei $v \in Views(x)$ ein View von x und $a \in A_t$ ein Akteur mit $x \in U_t(a)$. Dann bezeichnet:

- R_v^x den Zugriffsrestriktionsausdruck des Views v von x.
- $\omega(R_v^x, a, t)$ mit $\omega(R_v^x, a, t) \in \{true, false\}$ den Wert von R_v^x zum Zeitpunkt t für den Akteur a.

Ist für einen View v einer zugriffskontrollierten Komponente x kein Zugriffsrestriktionsausdruck explizit definiert, so wird für R_v^x implizit der Zugriffsrestriktionsausdruck festgelegt, der lediglich aus dem Booleschen Literal *false* besteht, d.h. es gilt:

$$\omega(R_v^x, a, t) = false \quad \text{für alle } t \in \Lambda(x) \text{ und alle } a \in A_t.$$

Ist also für einen View kein Zugriffsrestriktionsausdruck explizit definiert, hat zu keinem Zeitpunkt ein Akteur das Recht an diesem View. Durch diese Festlegung wird somit das Erlaubnisprinzip berücksichtigt (vgl. Abschnitt 4.1).

Die Zugriffsrestriktionsausdrücke der Views eines zugriffskontrollierten Depots bzw. eines zugriffskontrollierten K-Akteurs x sind, analog wie die Views von x, als Klasseneigenschaften festzulegen. Ist G ein explizit zugriffskontrollierter DA–Generator, so sind die Zugriffsrestriktionsausdrücke der Views von G mit G in einem **Zugriffsrestriktions–Part** zu definieren. Falls G ein zugriffskontrollierter Depot– oder K-Akteur–Generator ist, sind in dem

Zugriffsrestriktions–Part von G zusätzlich die Zugriffsrestriktionsausdrücke der Views, die für die Inkarnationen bzgl. G definiert sind, festzulegen.

Syntax des Zugriffsrestriktions–Parts
Ein Zugriffsrestriktions–Part ist Bestandteil des Spezifikationsteils eines explizit zugriffskontrollierten DA–Generators. Seine Syntax ist wie folgt festgelegt:

<*access-restriction-part*>	::=	`ACCESS RESTRICTIONS` <*access-restriction-list*> ;
<*access-restriction-list*>	::=	<*access-restriction-entry*> |
		<*access-restriction-list*> ; <*access-restriction-entry*>
<*access-restriction-entry*>	::=	<*view-name*> : <*acc-expression*>

Der Zugriffsrestriktions–Part eines explizit zugriffskontrollierten DA–Generators G wird durch die Schlüsselwörter `ACCESS RESTRICTIONS` eingeleitet und besteht aus einer Liste von Zugriffsrestriktionseinträgen (<*access-restriction-entry*>). Ein Zugriffsrestriktionseintrag besteht aus einem View–Namen (<*view-name*>) und aus einem Zugriffsrestriktionsausdruck (<*acc-expression*>). Bezüglich der View–Namen, die in dem Zugriffsrestriktions–Part von G erlaubt sind, gelten die in Abschnitt 4.4.3 gemachten Aussagen zu den View–Namen, die in einem ACL–Part auftreten können. Insbesondere darf in einem Zugriffsrestriktionsausdruck jeder dort mögliche View–Name lediglich einmal auftreten, was besagt, daß für jeden View höchstens ein Zugriffsrestriktionsausdruck definiert werden kann.

Beispiel
In Abbildung 4.10 ist der Anfang des Zugriffsrestriktions–Parts des zugriffskontrollierten Depot–Generators `KontoTyp` des Kontenverwaltungssystems angegeben. Der erste Eintrag in dem Zugriffsrestriktions–Part definiert den Zugriffsrestriktionsausdruck für den auf dem Generator `KontoTyp` implizit definierten View `Create`. Die beiden folgenden Einträge legen den Zugriffsrestriktionsausdruck für die auf den Inkarnationen bzgl. `KontoTyp` explizit definierten Views `LeseKontostand` und `KreditBerechnung` fest. Die für die Views angegebenen Zugriffsrestriktionsausdrücke werden weiter unten erklärt.

◇

Syntax für Zugriffsrestriktionsausdrücke
Im folgenden werden die Syntaxregeln, nach denen Zugriffsrestriktionsausdrücke gebildet werden, vollständig angegeben. Die Syntax für Zugriffsrestriktionsausdrücke folgt in ihrem Aufbau den in imperativen Programmiersprachen für die Bildung von Ausdrücken üblichen Syntaxregeln.

(4.8) Syntax für Zugriffsrestriktionsausdrücke

<*acc-expression*>	::=	<*acc-relation*> |
		<*acc-relation*> <*logical-operator*> <*acc-relation*>
<*acc-relation*>	::=	<*acc-term*> |
		<*acc-term*> <*relational-operator*> <*acc-term*>
<*acc-term*>	::=	<*acc-primary*> | <*unary-operator*> <*acc-primary*>
<*acc-primary*>	::=	<*literal*> | <*caller-attribut*> | <*inselp-path*> |
		<*name*> | <*user-identifier*> | <*role-identifier*> |
		<*acc-predicate*> | (<*acc-expression*>)
<*literal*>	::=	<*boolean-literal*> | <*integer-literal*> | <*real-literal*> |
		<*character-literal*> | <*string-literal*>
<*caller-attribut*>	::=	<*name*>

```
-- Zugriffskontrollierter Depot-Generator fuer die Bankkonten
PROTECTED DEPOT TYPE SPEC KontoTyp (...)
    ...
IS
  -- Zugriffsoperationen
  FUNCTION  TYPE SPEC LeseKontostand  RETURN real;
  ...

  -- View-Part
  VIEWS ...;

  -- ACL-Part
  ACL ...;

  -- Zugriffsrestriktions-Part
  ACCESS RESTRICTIONS
    Create          : Caller.ConGen = KontoVerwalter.KontoEroeffnen AND
                      Caller.Role = Kundenbetreuer AND
                      8 <= Zeit.Stunde AND Zeit.Stunde < 17 AND
                      1 <= Zeit.Wochentag AND Zeit.Wochentag < 6;
    LeseKontostand  : (IN_ACL(Caller.User, Caller.Role, THIS, THIS) AND
                       (Caller.Role = Kunde OR (Caller.Role = Kundenbetreuer
                        AND 8 <= Zeit.Stunde AND Zeit.Stunde < 17 AND
                        1 <= Zeit.Wochentag AND Zeit.Wochentag < 6)))
                      OR Caller.Role = Bankleiter;
    KreditBerechnung: Caller.ConGen = BerechneKredit AND
                      ((Caller.Role = Kundenbetreuer AND
                        IN_ACL(Caller.User, Caller.Role, THIS, THIS) AND
                        8 <= Zeit.Stunde AND Zeit.Stunde < 17 AND
                        1 <= Zeit.Wochentag AND Zeit.Wochentag < 6) OR
                       Caller.Role = Bankleiter);
    ...;
END KontoTyp;
```

Abbildung 4.10: Beispiel für einen Zugriffsrestriktions–Part

<*inselp-path*>	::=	<*identifier*> | <*inselp-path*> . <*identifier*>
<*acc-predicate*>	::=	`IN_ACL` (<*user-identifier*> , <*role-identifier*> , <*comp-identifier*> , <*this-or-view-name*>) | `ACCESSED` (<*user-identifier*> , <*comp-identifier*> , <*any-or-op-name*>)
<*comp-identifier*>	::=	<*name*> | `THIS`
<*this-or-view-name*>	::=	<*view-name*> | `THIS`
<*any-or-op-name*>	::=	<*op-name*> | `ANY`
<*logical-operator*>	::=	`AND` | `OR`
<*relational-operator*>	::=	= | /= | < | <= | > | >=
<*unary-operator*>	::=	+ | − | `NOT`

Die Regel '<*caller-attribut*> ::= <*name*>' wird an sich nicht benötigt, da <*name*> bereits aus <*acc-primary*> abgeleitet werden kann. Diese Regel ist hier lediglich angegeben, um im

weiteren den Aufbau und die Semantik von Zugriffsrestriktionsausdrücken besser erklären zu können. Wie gewöhnlich, sind nicht alle aus den Regeln ableitbaren Ausdrücke gültige Ausdrücke. Neben den für die logischen Operatoren und für die Vergleichsoperatoren üblichen Beschränkungen bzgl. der erlaubten Operanden–Datentypen (siehe hierzu [RW96]) gelten weitere Restriktionen, die im Verlauf dieses Abschnitts noch angegeben werden.

Ein Zugriffsrestriktionsausdruck kann neben den logischen Operatoren `AND`, `OR` und `NOT` sowie den auf den vordefinierten elementaren Datentypen definierten Vergleichsoperatoren, die im weiteren auch als Vergleichsprädikate bezeichnet werden, die **Zugriffsrestriktionsprädikate** `IN_ACL` und `ACCESSED` enthalten. Die Bedeutung der logischen Operatoren und der Vergleichsprädikate für die elementaren Datentypen ist wie üblich definiert. Die Semantik der Zugriffsrestriktionsprädikate `IN_ACL` und `ACCESSED` wird in den beiden folgenden Abschnitten 4.4.4.1 und 4.4.4.2 angegeben.

Das Vergleichsprädikat '=' ist neben seiner üblichen Verwendung auch in Ausdrücken der Form

- *<caller-attribut>* = *<user-identifier>*
- *<caller-attribut>* = *<role-identifier>*
- *<caller-attribut>* = *<inselp-path>*

erlaubt. Ein Ausdruck der letztgenannten Form wird im weiteren als **Kontextrestriktion** bezeichnet. Das Attribut `Caller` ist in einem INSEL$^+$–System für jeden Operationsaufruf definiert. Es beschreibt den Akteur, der den Operationsaufruf bei Ausführung seiner kanonischen Operation ausführt. Zu dieser Akteurbeschreibung gehören neben dem Namen des Generators des Akteurs und des Generators seiner Ausführungskomponente u.a. die Identifikatoren des Benutzers und der Rolle, die der Ausführungskomponente des Akteurs zugeordnet sind. Die einzelnen Elemente, aus denen sich das Attribut `Caller` zusammensetzt, stehen in INSEL$^+$ unter vordefinierten Bezeichnern zur Verfügung, die für *<caller-attribut>* verwendet werden dürfen. Tritt in einem Zugriffsrestriktionsausdruck ein solcher Bezeichner auf, so ist das Ergebnis der Auswertung dieses Zugriffsrestriktionsausdrucks also von dem Akteur abhängig, der den Operationsaufruf ausführt, der zur Auswertung des Ausdrucks führt. Das Attribut `Caller` und die Bezeichner, unter denen seine Elemente in INSEL$^+$ zur Verfügung stehen, werden genauer in Abschnitt 4.4.4.3 erklärt. Dort wird auch der Aufbau der INSEL$^+$–Pfade (*<inselp-path>*) angegeben, mit denen DA–Generatoren und benannte DA–Inkarnationen in einem INSEL$^+$–Programm identifiziert werden können und die zur Festlegung von Kontextrestriktionen verwendet werden können.

Bei der Definition eines Zugriffsrestriktionsausdrucks gelten in bezug auf die Namen von DE– und DA–Inkarnationen, die in dem Ausdruck auftreten dürfen, im wesentlichen die für INSEL$^+$ festgelegten Sichtbarkeitsregeln. Es gelten jedoch einige Ausnahmen. Diese werden in Abschnitt 4.4.4.4 erläutert.

Beispiele
Die in Abbildung 4.10 für die Views `Create`, `LeseKontostand` und `KreditBerechnung` angegebenen Zugriffsrestriktionsausdrücke sind Beispiele für gültige Zugriffsrestriktionsausdrücke. Die in den Zugriffsrestriktionsausdrücken auftretenen Bezeichner `Caller.ConGen`, `Caller.User` und `Caller.Role` bezeichnen Elemente des `Caller`–Attributs (siehe Abschnitt 4.4.4.3). `KontoVerwalter` ist der Name eines benannten K-Akteurs, und `KontoEroeffnen` bezeichnet eine Kommunikationsoperation von `KontoVerwalter`. `BerechneKredit` ist der

Bezeichner eines PS-Order–Generators, und der Record `Zeit` gibt die aktuelle Uhrzeit einschließlich Wochentag und aktuellem Datum an.

Obwohl die Semantik der einzelnen Teilausdrücke eines Zugriffsrestriktionsausdrucks erst in den folgenden Abschnitten erläutert wird, wird die Bedeutung der drei angegebenen Zugriffsrestriktionsausdrücke bereits an dieser Stelle etwas näher erklärt, um ein erstes intuitives Verständnis der Semantik von Zugriffsrestriktionsausdrücken zu vermitteln.

Der für den View `Create` definierte Zugriffsrestriktionsausdruck besteht aus der Konjunktion einer Kontextrestriktion, einer Rollenrestriktion und vier Bedingungen über der globalen DE–Inkarnation `Zeit`. Dieser Zugriffsrestriktionsausdruck besagt, daß ein Akteur lediglich dann das Recht zur Erzeugung einer Inkarnation bzgl. des Generators `KontoTyp` hat, wenn

– er sich im unmittelbaren Kontext der Ausführung der Kommunikationsoperation `KontoEroeffnen` des K-Akteurs `KontoVerwalter` befindet (`Caller.ConGen = KontoVerwalter.KontoEroeffnen`) und

– er in der Rolle `Kundenbetreuer` agiert (`Caller.Role = Kundenbetreuer`) und

– er die *erzeuge*–Operation in der Zeit zwischen 8.00 und 17.00 Uhr (`8 <= Zeit.Stunde AND Zeit.Stunde < 17`) an einem Werktag (Montag bis Freitag) (`1 <= Zeit.Wochentag AND Zeit.Wochentag < 6`) aufruft.

Der Zugriffsrestriktionsausdruck des Views `LeseKontostand` enthält u.a. ein `IN_ACL`–Prädikat. Mit diesem Zugriffsrestriktionsausdruck wird festgelegt, daß ein Akteur das Recht an dem View `LeseKontostand` einer Inkarnation x bzgl. des Generators `KontoTyp` und damit das Recht zur Ausführung der Zugriffsoperation `LeseKontostand` von x hat, wenn

– er in der Rolle `Bankleiter` agiert oder

– er in der Rolle `Kunde` agiert und ein entsprechender Benutzer–Rollen–Eintrag in der Zugriffskontrolliste von x vorhanden ist oder

– er in der Rolle `Kundenbetreuer` agiert und ein entsprechender Benutzer–Rollen–Eintrag in der Zugriffskontrolliste von x vorhanden ist und er die Operation `LeseKontostand` in der Zeit zwischen 8.00 und 17.00 Uhr an einem Werktag aufruft.

Der für den View `KreditBerechnung` angegebene Zugriffsrestriktionsausdruck besteht aus der Konjunktion einer Kontextrestriktion mit einem disjunktiven Teilausdruck. Ein Akteur hat gemäß dieses Zugriffsrestriktionsausdrucks das Recht zur Ausführung der Operationen des Views `KreditBerechnung`, wenn

– er sich im unmittelbaren Kontext der Ausführung der Operation `BerechneKredit` befindet und

– er in der Rolle `Bankleiter` agiert oder

– er in der Rolle `Kundenbetreuer` agiert und die Zugriffskontrolliste von x einen entsprechenden Benutzer–Rollen–Eintrag enthält und er die jeweilige Operation werktags zwischen 8.00 und 17.00 Uhr aufruft.

◇

4.4.4.1 Zugriffsrestriktionsprädikat IN_ACL

Nach dem in Abschnitt 4.4.3 Gesagten ist jedem explizit zugriffskontrollierten DA–Generator, jedem zugriffskontrollierten Depot und jedem zugriffskontrollierten K-Akteur eines INSEL$^+$–Systems implizit eine Zugriffskontrolliste zugeordnet. Mit dem im folgenden definierten vierstelligen Prädikat `IN_ACL` können Rechte in Abhängigkeit von dem Vorhandensein bestimmter Einträge in diesen Zugriffskontrollisten vergeben werden. Tritt in dem Zugriffsrestriktionsausdruck R_v^x eines Views v einer zugriffskontrollierten Komponente x ein `IN_ACL`–Prädikat auf, so wird bei Auswertung von R_v^x überprüft, ob in der Benutzer–Rollen–Liste des Zugriffskontrollisteneintrags eines Views w einer zugriffskontrollierten Komponente y für ein bestimmtes Benutzer–Rollen–Paar ein positiver oder negativer Eintrag enthalten ist.

Definition 4.9.: Zugriffsrestriktionsprädikat `IN_ACL`

Sei $\mathcal{S}$ ein INSEL$^+$–System zum Zeitpunkt $t \in \Lambda(\mathcal{S})$. X_t^{EZK} bezeichne die Menge der explizit zugriffskontrollierten DA–Generatoren, der zugriffskontrollierten Depots und der zugriffskontrollierten K-Akteure von $\mathcal{S}$ zum Zeitpunkt t. Sei $x \in X_t^{EZK}$ und $acl(x)$ die Zugriffskontrolliste von x zum Zeitpunkt t. $v \in Views(x)$ sei ein View von x. Weiter sei $B \in \mathcal{B} \cup \{-1\}$ ein Benutzeridentifikator und $R \in \mathcal{R} \cup \{-1\}$ ein Rollenidentifikator. Das Prädikat `IN_ACL` ist wie folgt definiert:

$$\texttt{IN_ACL} \;:\; \mathcal{B} \cup \{-1\} \times \mathcal{R} \cup \{-1\} \times X_t^{EZK} \times \bigcup_{y \in X_t^{EZK}} Views(y) \longrightarrow \{true, false\}$$

mit

$$\texttt{IN_ACL}(B,R,x,v) \triangleq \begin{cases} true & \text{falls es ein Element } (v', Entries(v')) \in acl(x) \text{ gibt mit:} \\ & -\ v = v' \text{ und} \\ & -\ \text{es gibt kein Element } (B',R',M') \in Entries(v) \text{ mit:} \\ & \quad M = - \text{ und} \\ & \quad -\ B' = B \text{ und } R' = R \text{ oder} \\ & \quad -\ B' = B \text{ und } R' = -1 \text{ oder} \\ & \quad -\ B' = -1 \text{ und } R' = R \text{ oder} \\ & \quad -\ B' = -1 \text{ und } R' = -1 \\ & -\ \text{es gibt ein Element } (B',R',M') \in Entries(v) \text{ mit:} \\ & \quad M = + \text{ und} \\ & \quad -\ B' = B \text{ und } R' = R \text{ oder} \\ & \quad -\ B' = B \text{ und } R' = -1 \text{ oder} \\ & \quad -\ B' = -1 \text{ und } R' = R \text{ oder} \\ & \quad -\ B' = -1 \text{ und } R' = -1 \\ false & \text{sonst} \end{cases}$$

□

Das Zugriffsrestriktionsprädikat `IN_ACL`(B,R,x,v) hat gemäß Definition (4.9) genau dann den Wert *true*, wenn in der Benutzer–Rollen–Liste des Views v in der Zugriffskontrolliste $acl(k)$

- weder der negative Benutzer–Rollen–Eintrag $(B,R,-)$, der negative Benutzer–Eintrag $(B,-1,-)$, der negative Rollen–Eintrag $(-1,R,-)$ noch der negative Welt–Eintrag $(-1,-1,-)$ enthalten ist und

– entweder der positive Benutzer–Rollen–Eintrag $(B, R, +)$, der positive Benutzer–Eintrag $(B, -1, +)$, der positive Rollen–Eintrag $(-1, R, +)$ oder der positive Welt–Eintrag $(-1, -1, +)$ enthalten ist.

Zur Auswertung des Prädikats IN_ACL(B, R, x, v) ist also zunächst zu überprüfen, ob in der Benutzer–Rollen–Liste des Views v der negative Welt–Eintrag oder für das Benutzer–Rollen–Paar (B, R), für den Benutzer B oder für die Rolle R ein negativer Eintrag enthalten ist. Ist dies der Fall, so wird das Prädikat zu *false* ausgewertet. Ist kein solcher negativer Eintrag vorhanden, ist zu untersuchen, ob in der Benutzer–Rollen–Liste ein positiver Eintrag für das Benutzer–Rollen–Paar (B, R), für den Benutzer B oder für die Rolle R vorhanden ist oder der positive Welt–Eintrag enthalten ist. Ist mindestens einer dieser Einträge vorhanden, wird das Prädikat zu *true* ausgewertet. Anderenfalls ist das Prädikat nicht erfüllt. In Abbildung 4.11 ist der Algorithmus zur Auswertung des IN_ACL–Prädikats anhand eines Ablaufplanes verdeutlicht.

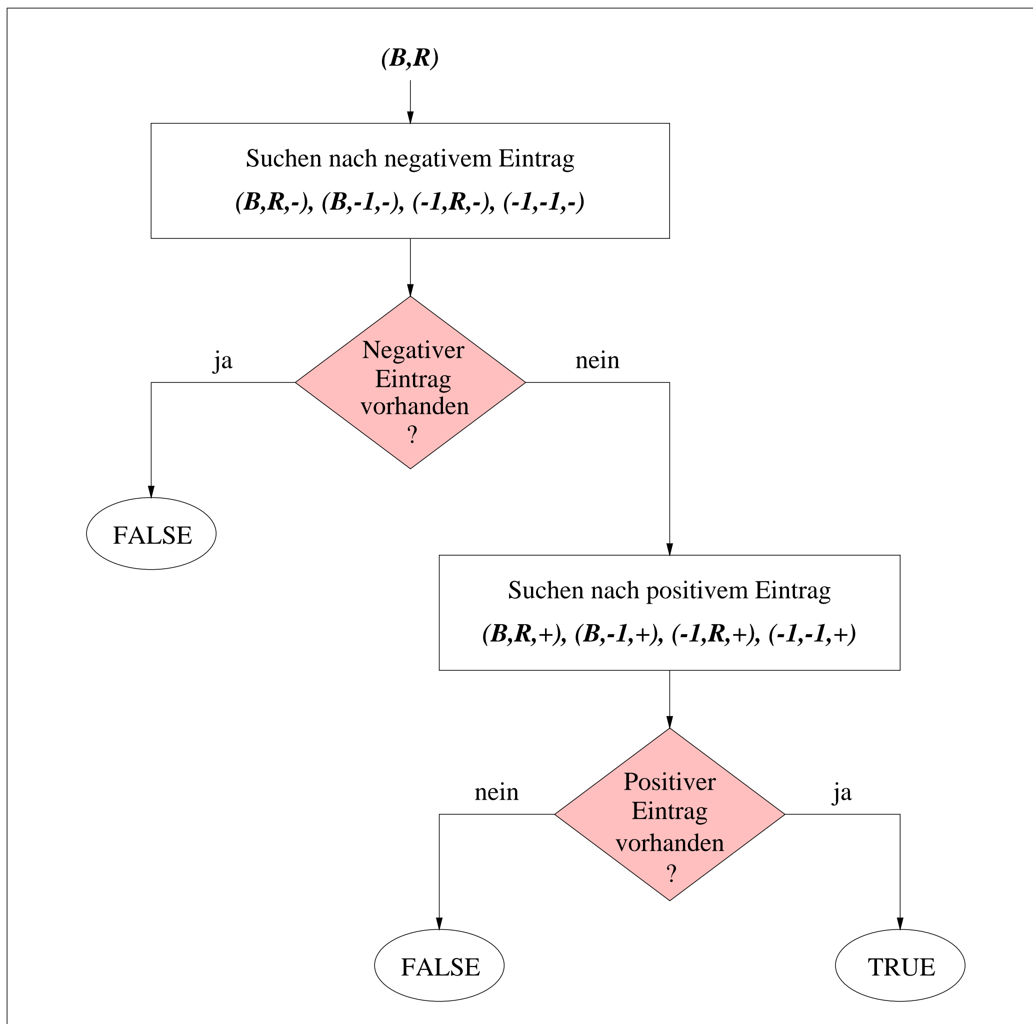

Abbildung 4.11: Auswertung des IN_ACL-Prädikats

Gemäß der Semantik des IN_ACL–Prädikats hat die "Reihenfolge" der Elemente in der Benutzer–Rollen–Liste des jeweiligen Views auf den Wert des Prädikats keinen Einfluß. Negative Einträge schlagen immer in dem Sinne durch, daß sie auch spezifischere positive Einträge

dominieren. Dabei ist ein Benutzer–Rollen–Eintrag spezifischer als ein Benutzer–Eintrag, ein Benutzer–Eintrag spezifischer als ein Rollen–Eintrag und ein Rollen–Eintrag spezifischer als ein Welt–Eintrag. Enthält die Benutzer–Rollen–Liste eines Views v einer zugriffskontrollierten Komponente x einen negativen Benutzer–Eintrag $(B, -1, -)$, bedeutet dies also, daß das Prädikat $\texttt{IN_ACL}(B, R, x, v)$ unabhängig von der Rolle R immer zu *false* ausgewertet wird, selbst wenn ein positiver Benutzer–Rollen–Eintrag $(B, R, +)$ in der Benutzer–Rollen–Liste vorhanden ist. Analoges gilt für einen negativen Rollen–Eintrag $(-1, R, -)$. In diesem Fall wird das `IN_ACL`–Prädikat für alle Benutzer, die in der Rolle R agieren, zu *false* ausgewertet, selbst wenn für den jeweiligen Benutzer ein positiver Benutzer–Eintrag enthalten ist. Ist in der Benutzer–Rollen–Liste der negative Welt–Eintrag $(-1, -1, -)$ enthalten, so hat das `IN_ACL`–Prädikat für alle Benutzer– und Rollenidentifikatoren den Wert *false*. Das Vorhandensein eines positiven Welt–Eintrags in der Benutzer–Rollen–Liste hat, sofern keine negativen Einträge in der Liste enthalten sind, zur Folge, daß das `IN_ACL`–Prädikat für alle Benutzer in beliebiger Rolle zu *true* ausgewertet wird.

Auf Basis des `IN_ACL`–Prädikats kann ein breites Spektrum von Zugriffsbeschränkungen für zugriffskontrollierte Komponenten durch geeignete Belegung der Zugriffskontrollisten dieser Komponenten und Verwendung des `IN_ACL`–Prädikats in den Zugriffsrestriktionsausdrücken der Views dieser Komponenten recht einfach festgelegt werden. Dafür werden im folgenden einige Beispiele aus dem Kontext des Kontenverwaltungssystems angegeben.

Beispiele

1. Die Zugriffsbeschränkung

   > Der Benutzer *Meyer* darf nur in der Rolle *Kunde* die Operation *Abheben* auf einem Konto K, dessen Inhaber er ist, ausführen

   wird durch folgenden Zugriffskontrollisteneintrag in der Zugriffskontrolliste von K erfaßt:

   $$(\textit{Abheben}, \{(\text{uid}(\textit{Meyer}), \text{uid}(\textit{Kunde}), +)\})$$

2. Die Zugriffsbeschränkung

   > Alle Benutzer, die in der Rolle *Kundenbetreuer* agieren, außer der Benutzer *Müller* dürfen die Operation *KontoEroeffnen* auf dem *KontoVerwalter* aufrufen

   wird durch folgenden Zugriffskontrollisteneintrag in der Zugriffskontrolliste des *KontoVerwalters* erfaßt:

   $$(\textit{KontoEroeffnen}, \{(-1, \text{uid}(\textit{Kundenbetreuer}), +), (\text{uid}(\textit{Müller}), -1, -)\})$$

3. Die Zugriffsbeschränkung

   > Alle Benutzer außer die Benutzer, die in der Rolle *Kunde* agieren, dürfen eine Inkarnation bezüglich des Generators *InterneMitteilungen* erzeugen

   wird durch folgenden Zugriffskontrollisteneintrag in der Zugriffskontrolliste des Generators *InterneMitteilungen* erfaßt:

   $$(\textit{Create}, \{(-1, -1, +), (-1, \text{uid}(\textit{Kunde}), -)\})$$

◇

Syntax des `IN_ACL`–Prädikats

Die Syntax des `IN_ACL`–Prädikats ist gemäß (4.8) wie folgt festgelegt:

<acc-predicate>	::=	`IN_ACL` (*<user-identifier>* , *<role-identifier>* , *<comp-identifier>* , *<this-or-view-name>*) | ...
<comp-identifier>	::=	*<name>* | `THIS`
<this-or-view-name>	::=	*<view-name>* | `THIS`

Bezüglich der für die einzelnen Stellen eines `IN_ACL`–Prädikats zulässigen Bezeichner gelten folgende Restriktionen:

– *<user-identifier>* muß -1 oder der Bezeichner einer DE–Inkarnation sein, die Inkarnation bzgl. des in INSEL$^+$ für Benutzeridentifikatoren vordefinierten Generators `UserUidType` ist. *<user-identifier>* kann insbesondere der Bezeichner `Caller.User` sein. `Caller.User` ist der Bezeichner des Elements des Attributs `Caller`, das den Benutzeridentifikator des aufrufenden Akteurs angibt (vgl. Abschnitt 4.4.4.3).

– *<role-identifier>* muß entweder -1, der Bezeichner einer im Role–Part des INSEL$^+$–Programms definierten Rolle oder der Bezeichner einer DE–Inkarnation sein, die Inkarnation bzgl. des vordefinierten Generators `RoleUidType` ist. Für *<role-identifier>* kann insbesondere der Bezeichner `Caller.Role` angegeben werden. `Caller.Role` bezeichnet das Element des Attributs `Caller`, das den Rollenidentifikator des aufrufenden Akteurs angibt.

– *<comp-identifier>* kann entweder das Schlüsselwort `THIS` oder ein Name sein. Mit *<comp-identifier>* ist die zugriffskontrollierte Komponente zu identifizieren, deren Zugriffskontrolliste bei Auswertung des `IN_ACL`–Prädikats zu überprüfen ist. `THIS` ist in INSEL$^+$ das Schlüsselwort für Selbstreferenzierung. Wird für *<comp-identifier>* also `THIS` angegeben, so wird damit die Komponente identifiziert, für die der View, in dessen Zugriffsrestriktionsausdruck das `IN_ACL`–Prädikat auftritt, definiert ist. Wird für *<comp-identifier>* ein Name angegeben, so muß dies der Name eines explizit zugriffskontrollierten DA–Generators, eines benannten zugriffskontrollierten Depots, eines benannten zugriffskontrollierten K-Akteurs oder eines Zeigers, der mit einem explizit zugriffskontrollierten Depot– oder K-Akteur–Generator qualifiziert ist, sein.

– *<this-or-view-name>* identifiziert den View, dessen Zugriffskontrollisteneintrag bei Auswertung des `IN_ACL`–Prädikats überprüft wird. Mit dem Schlüsselwort `THIS` wird der View identifiziert, für den der Zugriffsrestriktionsausdruck, in dem das `IN_ACL`–Prädikat auftritt, definiert wird. `THIS` ist hier lediglich erlaubt, wenn für *<comp-identifier>* auch `THIS` angegeben wurde. Anderenfalls muß *<this-or-view-name>* der Bezeichner eines Views sein, der für die mit *<comp-identifier>* identifizierte Komponente definiert ist.

Beispiel

Als Beispiel ist in Abbildung 4.12 der Zugriffsrestriktionsausdruck des Views `KontoAufloesen`, der für den benannten zugriffskontrollierten K-Akteur `KontoVerwalter` des Kontenverwaltungssystems definiert ist, angegeben. Der View `KontoAufloesen` besteht lediglich aus der Kommunikationsoperation `KontoAufloesen` des K-Akteurs `KontoVerwalter`. Bei Aufruf der Operation `KontoAufloesen` auf dem K-Akteur `KontoVerwalter` ist ein Zeiger auf das aufzulösende Konto–Depot als Eingabeparameter zu übergeben (`KontoZeiger`). Dieser Zeiger ist mit dem bereits erklärten explizit zugriffskontrollierten Depot–Generator `KontoTyp`

```
-- Zugriffskontrollierter K-Akteur-Generator fuer die Kontenverwaltung
PROTECTED TASK TYPE SPEC KontoVerwalterTyp
     ...
IS
  -- Kommunikationsoperationen
  ENTRY TYPE SPEC KontoEroeffnen (...);
  ENTRY TYPE SPEC KontoAufloesen (KontoZeiger: IN KontoPtrTyp;
                                        Error: OUT ErrorType);
  ...

  -- Zugriffsrestriktions-Part
  ACCESS RESTRICTIONS
    KontoAufloesen : Caller.Role = Kundenbetreuer AND
                     IN_ACL(Caller.User, Caller.Role, KontoZeiger, Aufloesen)
                     AND 8 <= Zeit.Stunde AND Zeit.Stunde < 17 AND
                     1 <= Zeit.Wochentag AND Zeit.Wochentag < 6;
    ...;
END KontoVerwalterTyp;
...
-- Inkarnierung des Kontoverwalters
TASK KontoVerwalter : KontoVerwalterTyp ...;
```

Abbildung 4.12: Beispiel für ein `IN_ACL`–Prädikat

qualifiziert. `Aufloesen` ist ein View, der auf Inkarnationen bzgl. des Generators `KontoTyp` definiert ist (vgl. Abbildung 4.7).

Der Zugriffsrestriktionsausdruck des Views `KontoAufloesen` besteht aus der Konjunktion einer Rollenrestriktion, eines `IN_ACL`–Prädikats und vier Bedingungen über der globalen DE–Inkarnation `Zeit`. Zur Auswertung des `IN_ACL`–Prädikats ist die Benutzer–Rollen–Liste des Views `Aufloesen` in der Zugriffskontrolliste des Konto–Depots, auf das der aktuelle Eingabeparameter `KontoZeiger` des `KontoAufloesen`–Aufrufs verweist, zu durchsuchen. Wird das `IN_ACL`–Prädikat zu *true* ausgewertet und agiert der jeweils aufrufende Akteur in der Rolle `Kundenbetreuer`, so hat er das Recht an der Kommunikationsoperation `KontoAufloesen` des K-Akteurs `KontoVerwalter`, wenn der Aufruf werktags in der Zeit zwischen zwischen 8.00 und 17.00 Uhr erfolgt.

◇

4.4.4.2 Zugriffsrestriktionsprädikat `ACCESSED`

Mit dem im folgenden erklärten Zugriffsrestriktionsprädikat `ACCESSED` können Rechte in Abhängigkeit der Zugriffshistorie von Benutzern vergeben werden. Tritt in dem Zugriffsrestriktionsausdruck R_v^x eines Views v einer zugriffskontrollierten Komponente x ein `ACCESSED`–Prädikat auf, wird bei Auswertung von R_v^x überprüft, ob ein Benutzer B bereits eine bestimmte äußere Operation op einer Komponente y ausgeführt hat. Der Identifikator B des Benutzers, die äußere Operation op sowie die Komponente y sind dabei als Eingaben des Prädikats festzulegen. Mit dem `ACCESSED`–Prädikat lassen sich Zugriffsbeschränkungen der Art:

> Wenn Benutzer A bereits auf Komponente y zugegriffen hat, darf er nicht auf Komponente x zugreifen

leicht auf programmiersprachlicher Ebene festlegen.

Definition 4.10.: Zugriffsrestriktionsprädikat `ACCESSED`

Sei $\mathcal{S}$ ein INSEL$^+$–System zum Zeitpunkt $t \in \Lambda(\mathcal{S})$. X_t^{GAD} bezeichne die Menge der DA–Generatoren, der Depots und der K-Akteure von $\mathcal{S}$ zum Zeitpunkt t.
Seien $x \in X_t^{GAD}$, $op \in O(x)$ eine äußere Operation von x und $B \in \mathcal{B}$ ein Benutzeridentifikator. Das Prädikat `ACCESSED` ist wie folgt definiert:

$$\texttt{ACCESSED} \;:\; \mathcal{B} \times X_t^{GAD} \times \{any\} \cup \bigcup_{y \in X_t^{GAD}} O(y) \;\longrightarrow\; \{true, false\}$$

mit

$$\texttt{ACCESSED}(B, x, op) \;\triangleq\; \begin{cases} true & \text{falls } op \in O(x) \text{ gilt und es einen Zeitpunkt } t' \leq t \text{ und einen Akteur } a \in A_{t'} \text{ gibt, so daß gilt: } user(a) = B \text{ und } a \text{ hat bei Ausführung seiner kanonischen Operation } op(a) \text{ die Operation } op \text{ auf } x \text{ aufgerufen und in } t \text{ ist bereits mit der } op\text{–Ausführung begonnen worden oder die entsprechende } op\text{–Ausführung ist zum Zeitpunkt } t \text{ bereits terminiert.} \\ false & \text{sonst} \end{cases}$$

$$\texttt{ACCESSED}(B, x, any) \;\triangleq\; \begin{cases} true & \text{falls es eine äußere Operation } op \in O(x) \text{ gibt, so daß gilt: } \texttt{ACCESSED}(B, x, op) = true \\ false & \text{sonst} \end{cases}$$

□

Das Zugriffsrestriktionsprädikat $\texttt{ACCESSED}(B, x, op)$ hat gemäß Definition (4.10) genau dann den Wert *true*, wenn der Benutzer, der mit B identifiziert wird, mindestens einmal die äußere Operation *op* von x ausgeführt hat. $\texttt{ACCESSED}(B, x, any)$ ist genau dann *true*, wenn der mit B identifizierte Benutzer mindestens eine der für x definierten äußeren Operationen ausgeführt hat, d.h. wenn er bereits auf die Komponente x zugegriffen hat. Wichtig dabei ist, daß mit Beginn und nicht erst mit Terminierung der entsprechenden *op*–Ausführung das `ACCESSED`–Prädikat den Wert *true* erhält. Die Motivation für diese Festlegung wird im folgenden anhand eines Beispiels erläutert.

Beispiel
Es wird die zu Beginn dieses Abschnitts informell formulierte Zugriffsbeschränkung betrachtet. Diese besagt, daß der Benutzer A zu einem Zeitpunkt t lediglich dann das Recht zur Ausführung einer äußeren Operation der Komponente x haben darf, wenn er bis zum Zeitpunkt t noch keine äußere Operation *op* von y ausgeführt hat und in t auch keine solche Operation ausführt. Zur Durchsetzung dieser Zugriffsbeschränkung ist für jeden View der Komponente x folgender Zugriffsrestriktionsausdruck festzulegen:

$$\texttt{NOT}(\texttt{ACCESSED}(A, y, any))$$

Mit der angegebenen Semantik des `ACCESSED`–Prädikats ist gewährleistet, daß der Benutzer A lediglich dann das Recht zur Ausführung einer äußeren Operation der Komponente x hat, wenn er bzw. ein Akteur, dem er zugeordnet ist, noch nicht mit der Ausführung einer äußeren Operation der Komponente y begonnen hat. Sobald ein entsprechender Akteur mit der Ausführung einer äußeren Operation *op* von y beginnt, wird das obige `ACCESSED`–Prädikat zu *true* ausgewertet. Damit hat der Benutzer A ab Beginn der *op*–Ausführung nicht mehr das Recht zur Ausführung einer äußeren Operation von x, da die Zugriffsrestriktionsausdrücke aller Views von x ab diesem Zeitpunkt den Wert *false* haben. Würde das `ACCESSED`–Prädikat erst mit Terminierung der *op*–Ausführung den Wert *true* erhalten, so würden die Zugriffsrestriktionsausdrücke der Views von x während der *op*–Ausführung zu *true* ausgewertet. In diesem Zeitraum hätte der Benutzer A somit das Recht zur Ausführung einer äußeren Operation der Komponente x, obwohl er bereits eine äußere Operation der Komponente y ausführt. Damit wäre die obige Zugriffsbeschränkung jedoch nicht korrekt durchgesetzt.

◇

Syntax des `ACCESSED`–Prädikats

Die Syntax des `ACCESSED`–Prädikats ist gemäß (4.8) wie folgt festgelegt:

<acc-predicate>	::=	`ACCESSED` (*<user-identifier>* , *<comp-identifier>* , *<any-or-op-name>*)
<comp-identifier>	::=	*<name>* | `THIS`
<any-or-op-name>	::=	*<op-name>* | `ANY`

Bezüglich der für die einzelnen Stellen eines `ACCESSED`–Prädikats zulässigen Bezeichner gelten folgende Restriktionen:

– *<user-identifier>* muß der Bezeichner einer DE–Inkarnation sein, die Inkarnation bzgl. des für Benutzeridentifikatoren vordefinierten Generators `UserUidType` ist. Für *<user-identifier>* kann insbesondere der Bezeichner `Caller.User` angegeben werden.

– Mit *<comp-identifier>* ist die Komponente zu identifizieren, für die bei Auswertung des `ACCESSED`–Prädikats zu überprüfen ist, ob der mit *<user-identifier>* identifizierte Benutzer bereits auf diese Komponente zugegriffen hat. *<comp-identifier>* kann wie in einem `IN_ACL`–Prädikat das Schlüsselwort `THIS` oder ein Name sein. Wird für *<comp-identifier>* ein Name angegeben, muß dies der Name eines DA–Generators, eines Depots, eines K-Akteurs oder eines Zeigers, der mit einem Depot– oder K-Akteur–Generator qualifiziert ist, sein.

– *<any-or-op-name>* legt die äußere Operation fest, die der mit *<user-identifier>* identifizierte Benutzer auf der mit *<comp-identifier>* identifizierten Komponente ausgeführt haben muß, damit das `ACCESSED`–Prädikat den Wert *true* erhält. *<any-or-op-name>* kann entweder das Schlüsselwort `ANY` oder der Name einer äußeren Operation sein, die für die mit *<comp-identifier>* identifizierte Komponente definiert ist. Wird mit *<comp-identifier>* ein DA–Generator identifiziert, so ist als Operationsname der Bezeichner `Create` zulässig. Für einen explizit zugriffskontrollierten DA–Generator sind zusätzlich die Operationsnamen `ChangeGenACL` und `ListGenACL` erlaubt. Identifiziert *<comp-identifier>* ein Depot oder einen K-Akteur x und ist G der Generator von x, sind als Operationsnamen die Namen der im Spezifikationsteil von G definierten DA–Generatoren zulässig. Ist x zugriffskontrolliert, sind zusätzlich die Operationsnamen `ChangeACL` und `ListACL` erlaubt.

Beispiel
Abbildung 4.13 zeigt den Spezifikationsteil eines explizit zugriffskontrollierten Depot–Generators, der Bestandteil einer Beispielanwendung ist, die ausführlich in Abschnitt 4.6.2 erklärt wird. Die Zugriffsrestriktionsausdrücke der beiden angegebenen Zugriffsoperationen bestehen jeweils aus einem `ACCESSED`–Prädikat. Der Zugriffsrestriktionsausdruck der Operation `LoesungAbgeben` legt fest, daß ein Benutzer lediglich dann das Recht zur Ausführung der Operation `LoesungAbgeben` auf einer Inkarnation bzgl. des Depot–Generators hat, wenn er noch nicht die Zugriffsoperation `MusterloesungEinsehen` auf dieser Inkarnation ausgeführt hat. Die Operation `MusterloesungEinsehen` darf hingegen erst dann ausgeführt werden, wenn der Benutzer mindestens einmal die Operation `LoesungAbgeben` auf der Inkarnation ausgeführt hat.

◇

```
-- Zugriffskontrollierter Depot-Generator fuer die Hausaufgabenverwaltung
PROTECTED DEPOT TYPE SPEC HausaufgabenVerwaltungsTyp IS
  -- Zugriffsoperationen
  PROCEDURE TYPE SPEC LoesungAbgeben (...);
  PROCEDURE TYPE SPEC MusterloesungEinsehen (...);

  -- Zugriffsrestriktions-Part
  ACCESS RESTRICTIONS
    LoesungAbgeben : NOT ACCESSED(Caller.User, THIS, MusterloesungEinsehen);
    MusterloesungEinsehen : ACCESSED(Caller.User, THIS, LoesungAbgeben);
    ...
END HausaufgabenVerwaltungsTyp;
```

Abbildung 4.13: Beispiel für `ACCESSED`–Prädikate

4.4.4.3 Aufruferbeschreibung `Caller` und Kontextrestriktionen

Um Rechte spezifisch an einzelne Akteure vergeben zu können, muß es möglich sein, in einem Zugriffsrestriktionsausdruck Bedingungen formulieren zu können, deren Erfüllung von Eigenschaften der Akteure, die Operationsaufrufe ausführen, die die Auswertung des Zugriffsrestriktionsausdrucks bewirken, abhängig ist. Als Basis für die Formulierung solcher Bedingungen steht in INSEL$^+$ das für jeden Operationsaufruf implizit definierte Attribut `Caller` zur Verfügung. Das Attribut `Caller` eines Operationsaufrufs beschreibt den Akteur, der den Operationsaufruf bei Ausführung seiner kanonischen Operation ausführt. Die Bestandteile einer solchen Akteurbeschreibung werden durch das in der folgenden Definition eingeführte **Kontextattribut** eines Akteurs festgelegt. Dabei werden die in (3.4) eingeführten Bezeichnungen $\varphi_t(a)$ für die Ausführungskomponente eines Akteurs a zum Zeitpunkt t, $gen(x)$ für den Generator einer Inkarnation x und $name(y)$ für den Namen einer benannten Komponente y verwendet. Außerdem werden die in Definition (4.1) und (4.3) definierten Attribute $user(x)$ und $role(x)$ einer DA–Inkarnation x vorausgesetzt.

Definition 4.11.: Kontextattribut eines Akteurs

Sei $\mathcal{S}$ ein INSEL$^+$–System zum Zeitpunkt $t \in \Lambda(\mathcal{S})$ und $a \in A_t$ ein Akteur. Das **Kontextattribut** $Con_t(a)$ des Akteurs a zum Zeitpunkt t ist wie folgt definiert:

$$Con_t(a) \triangleq (User_t(a), Role_t(a), ActGen_t(a), ActName_t(a), ConGen_t(a), ConName_t(a))$$

mit:

$$\begin{array}{lcl} User_t(a) & \triangleq & user(\varphi_t(a)) \\ Role_t(a) & \triangleq & role(\varphi_t(a)) \\ ActGen_t(a) & \triangleq & name(gen(a)) \\ ActName_t(a) & \triangleq & \begin{cases} name(a) & \text{falls } a \text{ N–Komponente ist} \\ \bot & \text{sonst} \end{cases} \\ ConGen_t(a) & \triangleq & name(gen(\varphi_t(a))) \\ ConName_t(a) & \triangleq & \begin{cases} name(\varphi_t(a)) & \text{falls } \varphi_t(a) \text{ N–Komponente ist} \\ \bot & \text{sonst} \end{cases} \end{array}$$

□

Das Kontextattribut $Con_t(a)$ eines Akteurs a ist gemäß Definition (4.11) ein 6–Tupel. Das erste Element von $Con_t(a)$ gibt den Identifikator des Benutzers an, der der aktuellen Ausführungskomponente von a zugeordnet ist. a führt die kanonische Operation seiner Ausführungskomponente im Auftrag dieses Benutzers aus. Das zweite Element enthält den Identifikator der der Ausführungskomponente von a zugeordneten Rolle. Das dritte und vierte Element beschreiben den Akteur a durch Angabe des Namens des Generators, bzgl. dem a inkarniert wurde, und zusätzlich durch seinen Namen, falls a benannter K-Akteur ist. Durch die letzten beiden Elemente des Kontextattributs wird die Ausführungskomponente des Akteurs a, also die Komponente, deren kanonische Operation a zum Zeitpunkt t ausführt, beschrieben. Die Ausführungskomponente wird ebenfalls durch Angabe des Namens ihres Generators und zusätzlich durch ihren eigenen Namen, falls sie benannt ist, charakterisiert.

Aus Definition (4.11) folgt, daß sich der Wert des Kontextattributs eines Akteurs immer dann ändert, wenn sich seine Ausführungskomponente ändert. Für einen M-Akteur a können sich lediglich die Elemente $ConGen_t(a)$ und $ConName_t(a)$ seines Kontextattributs ändern. $User_t(a)$ und $Role_t(a)$ sind für die Lebenszeit von a konstant. Dies ergibt sich unmittelbar aus den Definitionen (4.1) und (4.3) der Attribute *user* und *role*. Ist a ein K-Akteur, so können sich $User_t(a)$ und $Role_t(a)$ immer dann ändern, wenn a eine K-Order annimmt. Während der Ausführung der kanonischen Operation der K-Order gibt $User_t(a)$ den Identifikator des Benutzers an, der die K-Order erzeugt hat, und nicht des Benutzers, in dessen Auftrag der K-Akteur a erzeugt wurde. Gleiches gilt für die Rolle $Role_t(a)$. Diese Festlegung hat zur Konsequenz, daß im Kontext der Ausführung einer Kommunikationsoperation die Rechte des Benutzers und der Rolle des Aufrufers der Kommunikationsoperation zur Verfügung stehen. Zur Ausführung einer Kommunikationsoperation werden also die Rechte, die der aufrufende Akteur aufgrund des ihm zugeordneten Benutzers und der ihm zugeordneten Rolle besitzt, an den K-Akteur delegiert, der die Kommunikationsoperation ausführt. Darüberhinaus können bei Ausführung der Kommunikationsoperation spezifisch an den K-Akteur und an dessen Generator durch Kontextrestriktionen vergebene Rechte wahrgenommen werden.

Beispiel
Als Beispiel, an dem die Vorteile der angesprochenen Rechtedelegierung bei Ausführung von Kommunikationsoperationen erkennbar sind, wird ein System betrachtet, das aus einem Drucker–Server besteht, der als benannter K-Akteur `DruckerVerwalter` realisiert ist. Der K-Akteur `DruckerVerwalter` bietet eine Operation `DruckeDokument` zum Ausdrucken eines von einem Benutzer erstellten Dokuments an. Zum Ausdrucken des Dokuments muß der `DruckerVerwalter` das Recht zum Lesen des Dokuments haben. Ruft ein Benutzer B die Operation `DruckeDokument` auf dem `DruckerVerwalter` auf, so hat der `DruckerVerwalter` gemäß der obigen Festlegungen im Kontext der Ausführung von `DruckeDokument` die Rechte von B und damit auch das Recht zum Lesen des von B erstellten Dokuments. Würden die Rechte von B bei Ausführung von `DruckeDokument` nicht an den `DruckerVerwalter` delegiert, so müßte der Benutzer B dem K-Akteur `DruckerVerwalter` bzw. dem Benutzer, der diesen K-Akteur erzeugt hat, das Recht zum Lesen des von ihm erstellten Dokuments erteilen. Damit könnte dann jedoch nicht nur der Benutzer B das Dokument ausdrucken, sondern potentiell könnten alle Benutzer, die das Recht an der Operation `DruckeDokument` des `DruckerVerwalter` haben und in deren Ausführungsumgebung das Dokument liegt, dieses Dokument drucken.

◇

Auf Basis des erklärten Kontextattributs der Akteure wird nun die Definition des Attributs `Caller` eines Operationsaufrufs angegeben.

Definition 4.12.: Attribut `Caller`

Sei $a \in A_t$ ein Akteur. a führe zum Zeitpunkt t einen Operationsaufruf C aus. Dann ist das Attribut `Caller`(C) für den Operationsaufruf C wie folgt definiert:

$$\texttt{Caller}(C) \triangleq Con_t(a)$$

□

Das Attribut `Caller`(C) eines Operationsaufrufs C ist also ein 6–Tupel, das dem Kontextattribut des Akteurs, der den Operationsaufruf ausführt, entspricht. Durch Angabe von Bedingungen, die für die einzelnen Elemente des `Caller`–Attributs gelten müssen, können in einem Zugriffsrestriktionsausdruck Bedingungen formuliert werden, deren Erfüllung von dem Wert des `Caller`–Attributs des jeweiligen Operationsaufrufs, der zur Auswertung des Zugriffsrestriktionsausdrucks führt, abhängig ist. Hierzu gehören z.B. Bedingungen, die festlegen, daß dem Akteur, der den Operationsaufruf ausführt, eine bestimmte Rolle zugeordnet sein muß oder die Ausführungskomponente dieses Akteurs Inkarnation bzgl. eines bestimmten Generators sein muß. Bevor näher darauf eingegangen wird, wie solche Bedingungen in INSEL$^+$ syntaktisch zu formulieren sind, werden zunächst die Bezeichner angegeben, unter denen die Elemente des `Caller`–Attributs in INSEL$^+$ zur Verfügung stehen.

Syntax des `Caller`–Attributs
Die in (4.8) für das `Caller`–Attribut angegebene Syntaxregel lautet:

<caller-attribut> ::= *<name>*

Diese Regel soll andeuten, daß die einzelnen Elemente des `Caller`–Attributs in INSEL$^+$ unter vordefinierten Bezeichnern zur Verfügung stehen. In Tabelle 4.2 ist die Zuordnung zwischen

Element des `Caller`–Attributs	vordefinierter Bezeichner
$User_t$	`Caller.User`
$Role_t$	`Caller.Role`
$ActGen_t$	`Caller.ActorGen`
$ActName_t$	`Caller.ActorName`
$ConGen_t$	`Caller.ConGen`
$ConName_t$	`Caller.ConName`

Tabelle 4.2: INSEL$^+$–Bezeichner der Elemente des `Caller`–Attributs

den durch die Definitionen (4.11) und (4.12) festgelegten Elementen des `Caller`–Attributs und den Bezeichnern, unter denen diese Elemente in INSEL$^+$ programmiersprachlich zur Verfügung stehen, angegeben.

Die Bezeichner der Elemente des `Caller`–Attributs können lediglich in Zugriffsrestriktionsausdrücken verwendet werden. Tritt ein solcher Bezeichner in dem Zugriffsrestriktionsausdruck R_v^x eines Views v einer zugriffskontrollierten Komponente x auf, so ist bei Auswertung von R_v^x der Wert des mit diesem Bezeichner identifizierten Elements des `Caller`–Attributs gemäß der Definitionen (4.11) und (4.12) festgelegt.

Im folgenden wird getrennt für die einzelnen Elemente des `Caller`–Attributs erklärt, an welchen Stellen der entsprechende vordefinierte Bezeichner in einem Zugriffsrestriktionsausdruck zulässig ist und welche Bedingungen damit jeweils formulierbar sind.

Caller.User

Der Bezeichner `Caller.User` ist in einem Zugriffsrestriktionsausdruck an den Stellen zulässig, an denen ein Benutzeridentifikator (<*user-identifier*>) stehen muß. Er kann also an erster Stelle eines `IN_ACL`–Prädikats (vgl. Abschnitt 4.4.4.1) sowie an erster Stelle eines `ACCESSED`–Prädikats (vgl. Abschnitt 4.4.4.2) auftreten. Außerdem kann der Bezeichner `Caller.User` in einer Gleichung der Form

`Caller.User` = <*user-identifier*>

stehen, wobei <*user-identifier*> der Bezeichner einer DE–Inkarnation bzgl. des für Benutzeridentifikatoren vordefinierten Generators `UserUidType` sein muß. Eine solche Gleichung, die im weiteren als **Benutzerrestriktion** bezeichnet wird, hat genau dann den Wert *true*, wenn der Wert der mit <*user-identifier*> bezeichneten DE–Inkarnation gleich dem Wert von `Caller.User` ist.

Caller.Role

Der Bezeichner `Caller.Role` ist in einem Zugriffsrestriktionsausdruck an den Stellen zulässig, an denen ein Rollenidentifikator (<*role-identifier*>) stehen muß. Er kann also insbesondere an zweiter Stelle eines `IN_ACL`–Prädikats angegeben werden. Darüberhinaus kann der Bezeichner `Caller.Role` in einer Gleichung der Form

`Caller.Role` = <*role-identifier*>

stehen, wobei <*role-identifier*> der Bezeichner einer im Role–Part des entsprechenden INSEL$^+$–Programms definierten Rolle oder der Bezeichner einer DE–Inkarnation, die Inkarnation bzgl. des für Rollenidentifikatoren vordefinierten Generators `RoleUidType` ist, sein

muß. Eine Gleichung der obigen Form wird im weiteren als **Rollenrestriktion** bezeichnet. Eine Rollenrestriktion hat genau dann den Wert *true*, wenn die Rolle, die mit <*role-identifier*> identifiziert wird, gleich der Rolle ist, die dem Wert von `Caller.Role` entspricht.

Beispiel
Als Beispiel für die zulässige Verwendung der Bezeichner `Caller.User` und `Caller.Role` in Zugriffsrestriktionsausdrücken kann der bereits in Abbildung 4.10 angegebene Zugriffsrestriktionsausdruck des Views `LeseKontostand`, der für Inkarnationen bzgl. des Depot–Generators `KontoTyp` definiert ist, betrachtet werden.

```
LeseKontostand  : (IN_ACL(Caller.User, Caller.Role, THIS, THIS) AND
                   (Caller.Role = Kunde OR (Caller.Role = Kundenbetreuer
                    AND 8 <= Zeit.Stunde AND Zeit.Stunde < 17 AND
                    1 <= Zeit.Wochentag AND Zeit.Wochentag < 6)))
                  OR Caller.Role = Bankleiter;
```

◇

Die Benutzer– und Rollenrestriktionen können dazu genutzt werden, statische Rechte an Benutzer bzw. Rollen zu vergeben. Besteht ein Zugriffsrestriktionsausdruck z.B. aus einer Disjunktion von Benutzerrestriktionen, für die gilt, daß die mit <*user-identifier*> bezeichneten DE–Inkarnationen Konstanten sind, so werden durch den Zugriffsrestriktionsausdruck statische Rechte an die Benutzer vergeben, die mit den Werten dieser DE–Inkarnationen identifiziert werden. Gleiches gilt für Rollenrestriktionen, wenn <*role-identifier*> entweder der Bezeichner einer Rolle oder eine Konstante bzgl. des Generators `RoleUidType` ist. In dem oben angegebenen Zugriffsrestriktionsausdruck des Views `LeseKontostand` wird z.B. durch den Teilausdruck `OR Caller.Role = Bankleiter` ein statisches Recht zur Ausführung der Zugriffsoperation `LeseKontostand` der Konto–Depots an die Rolle `Bankleiter`, d.h. an alle Benutzer, die in dieser Rolle agieren, vergeben. Durch die Konjunktion einer Benutzer– und einer Rollenrestriktion mit konstantem <*user-identifier*> und <*role-identifier*> wird ein statisches Recht an die mit den Werten dieser beiden Konstanten festgelegte Benutzer–Rollen–Kombination vergeben. Dies bedeutet, daß der mit dem Wert von <*user-identifier*> identifizierte Benutzer lediglich in der mit dem Wert von <*role-identifier*> identifizierten Rolle das Recht an dem View hat, für den der entsprechende Zugriffsrestriktionsausdruck festgelegt ist.

Caller.ActorGen, Caller.ActorName, Caller.ConGen und Caller.ConName

Die Bezeichner `Caller.ActorGen`, `Caller.ActorName`, `Caller.ConGen` sowie `Caller.ConName` sind lediglich auf der linken Seite einer Gleichung zulässig, in der auf der rechten Seite ein sogenannter INSEL$^+$–Pfad steht, also:

```
Caller.ActorGen    =  <inselp-path>
Caller.ActorName   =  <inselp-path>
Caller.ConGen      =  <inselp-path>
Caller.ConName     =  <inselp-path>
```

Eine Gleichung der obigen Form wird im weiteren als **Kontextrestriktion** bezeichnet. Durch Kontextrestriktionen kann in dem Zugriffsrestriktionsausdruck eines Views einer zugriffskontrollierten Komponente festgelegt werden, daß die äußeren Operationen des Views lediglich im Kontext der Ausführung bestimmter anderer Operationen ausgeführt werden dürfen. Die

Operationen, in deren Kontext die Ausführung erlaubt ist, werden dabei durch den INSEL$^+$–Pfad der Kontextrestriktion festgelegt. Bevor näher auf die Semantik der den vier obigen Gleichungen entsprechenden unterschiedlichen Arten von Kontextrestriktionen eingegangen wird, wird zunächst der Aufbau von INSEL$^+$–Pfaden erklärt.

Ein **INSEL$^+$–Pfad** (<*inselp-path*>) besteht gemäß der in (4.8) angegebenen Syntaxregel

<*inselp-path*> ::= <*identifier*> | <*inselp-path*> . <*identifier*>

aus einer durch '.' getrennten Folge von Bezeichnern. Die Bezeichner, die in einem INSEL$^+$–Pfad auftreten dürfen, müssen bestimmten Bedingungen genügen, die in der folgenden Definition angegeben werden.

Definition 4.13.: INSEL$^+$–Pfad

Sei $\mathcal{F} = G_1.G_2. \ \ldots \ .G_n$, $n \geq 1$, eine Folge von Bezeichnern, die in einem INSEL$^+$–Programm auftreten. Sei weiter G der Bezeichner des explizit zugriffskontrollierten DA–Generators, in dessen Zugriffsrestriktions–Part die Bezeichnerfolge $\mathcal{F}$ auftritt, und D der Bezeichner des DA–Generators, in dessen Deklarationsteil G definiert wird.

Die Bezeichnerfolge $\mathcal{F}$ ist ein **INSEL$^+$–Pfad** genau dann, wenn

1. für alle $i \in \{1, \ldots, n\}$ gilt: G_i ist der Bezeichner eines DA–Generators oder der Bezeichner einer benannten DA–Inkarnation;
2. für G_1 gilt: G_1 ist der Bezeichner eines DA–Generators oder einer benannten DA–Inkarnation, der bzw. die entweder im Deklarationsteil von G nach dem Zugriffsrestriktions–Part oder im Deklarationsteil von D nach G definiert ist;
3. für alle $i \in \{2, \ldots, n\}$ gilt: G_i ist der Bezeichner eines DA–Generators oder einer benannten DA–Inkarnation, der bzw. die im Deklarationsteil von G_{i-1} definiert ist.

□

Die in einem INSEL$^+$–Pfad $\mathcal{F}$ auftretenen Bezeichner müssen also gemäß Definition (4.13) Bezeichner von DA–Generatoren und benannten DA–Inkarnationen sein, die textuell in dem INSEL$^+$–Programm nach dem Zugriffsrestriktions–Part, in dem $\mathcal{F}$ enthalten ist, definiert bzw. deklariert werden.

Beispiel
Abbildung 4.14 zeigt einen groben Ausschnitt aus dem INSEL$^+$–Programm, das das Kontenverwaltungssystem implementiert. In Zugriffsrestriktionsausdrücken, die in dem Zugriffsrestriktions–Part des Depot–Generators `KontoTyp` definiert werden (in der Abbildung ist die entsprechende Stelle mit `--**` gekennzeichnet), sind z.B. folgende Bezeichnerfolgen INSEL$^+$–Pfade:

```
KontoVerwalterTyp,
KontoVerwalterTyp.KontoAufloesen,
KontoVerwalter,
KontoVerwalter.KontoAufloesen,
KontoVerwalterTyp.Zinsberechnung,
KontoVerwalter.Zinsberechnung.Zinsberechner
```

```
PROCESS KontenVerwaltungsSystem IS
  ...
  -- Zugriffskontrollierter Depot-Generator fuer die Bankkonten
  PROTECTED DEPOT TYPE SPEC KontoTyp (...)
    ...
    --** Zugriffsrestriktions-Part
    ...
  END KontoTyp;
  ...
  -- Zugriffskontrollierter K-Akteur-Generator fuer die Kontoverwaltung
  PROTECTED TASK TYPE SPEC KontoVerwalterTyp ... IS
    -- Kommunikationsoperationen
    ENTRY TYPE SPEC KontoAufloesen (...)
    ...
  END KontoVerwalterTyp;

  PROTECTED TASK TYPE KontoVerwalterTyp ... IS
    ...
    -- M-Akteur-Generator fuer die automatische Zinsberechnung
    PROTECTED PROCESS TYPE Zinsberechnung ... IS
      ...
      -- M-Akteur-Generator zur Berechnung der monatlichen Zinsen
      PROCESS TYPE Zinsberechner ... IS
        ...
      END Zinsberechner;
      ...
    END Zinsberechnung;
    ...
  END KontoVerwalterTyp;

  -- Inkarnierung des Kontoverwalters
  TASK KontoVerwalter : KontoVerwalterTyp ...;
  ...
END KontenVerwaltungsSystem;
```

Abbildung 4.14: Ausschnitt aus dem INSEL$^+$–Programm des Kontenverwaltungssystems

◇

Ist $\mathcal{F} = G_1.G_2. \ \ldots \ .G_n$ ein INSEL$^+$–Pfad, der in dem Zugriffsrestriktions–Part eines zugriffskontrollierten DA–Generators G auftritt, so wird mit $\mathcal{F}$ ausgehend von dem Generator G eindeutig der DA–Generator oder die benannte DA–Inkarnation G_n identifiziert, indem $\mathcal{F}$ den Pfad von G nach G_n gemäß der Schachtelungsstruktur des INSEL$^+$–Programms angibt. Mit den für INSEL$^+$ festgelegten Sichtbarkeitsregeln folgt, daß der Generator G und Inkarnationen bzgl. G potentiell für die mit $\mathcal{F}$ identifizierte Komponente G_n sichtbar sein können. Damit sind die äußeren Operationen von G bzw. von Inkarnationen bzgl. G potentiell von G_n, falls G_n benannte DA–Inkarnation ist, bzw. von Inkarnationen bzgl. G_n, falls G_n DA–Generator ist, aus aufrufbar. Ist v ein View, in dessen Zugriffsrestriktionsausdruck eine Kontextrestriktion, in der auf der rechten Seite der Pfad $\mathcal{F}$ steht, enthalten ist, so wird durch den entsprechende Kontextrestriktion festgelegt, daß die äußeren Operationen, die zu

v gehören, lediglich im Kontext der Ausführung der kanonischen Operation von G_n bzw. der kanonischen Operationen von Inkarnationen bzgl. G_n ausgeführt werden dürfen. Abhängig von der Art der Kontextrestriktion, d.h. abhängig davon, welcher Bezeichner des `Caller`–Attributs auf der linken Seite der Kontextrestriktion steht, ist die Ausführung der äußeren Operationen von v entweder nur im unmittelbaren Kontext oder zusätzlich auch im mittelbaren Kontext der Ausführung der entsprechenden kanonischen Operationen erlaubt. Auf die unterschiedlichen Arten von Kontextrestriktionen wird nun im folgenden näher eingegangen.

Caller.ActorGen

Der letzte Bezeichner G_n eines INSEL$^+$–Pfades, der auf der rechten Seite einer Gleichung der Form

`Caller.ActorGen` = <*inselp-path*>

steht, muß der Bezeichner eines Akteur–Generators sein. Eine solche Gleichung hat genau dann den Wert *true*, wenn der durch den Wert von `Caller.ActorGen` identifizierte Akteur–Generator gleich dem durch den INSEL$^+$–Pfad <*inselp-path*> identifizierten Akteur–Generator ist. Besteht der Zugriffsrestriktionsausdruck eines Views aus einer derartigen Kontextrestriktion, so folgt mit der Semantik von `Caller.ActorGen`, daß alle Akteure, die Inkarnationen bzgl. des mit dem INSEL$^+$–Pfad identifizierten Akteur–Generators sind, das Recht zur Ausführung der äußeren Operationen des Views haben. Die äußeren Operationen des Views dürfen also lediglich im unmittelbaren und mittelbaren Kontext der Ausführung der kanonischen Operationen dieser Akteure ausgeführt werden.

Beispiel

Der Zugriffsrestriktionsausdruck für den View `Create` des explizit zugriffskontrollierten Depot–Generators `KontoTyp` könnte zum Beispiel wie folgt definiert sein:

```
Create : Caller.ActorGen = KontoVerwalterTyp;
```

Durch diese Kontextrestriktion würde festgelegt, daß alle Akteure, die Inkarnationen bzgl. des K-Akteur–Generators `KontoVerwalterTyp` sind, das Recht zum Erzeugen einer Inkarnation bzgl. des Generators `KontoTyp` hätten.

◇

Caller.ActorName

Der letzte Bezeichner G_n eines INSEL$^+$–Pfades, der auf der rechten Seite einer Gleichung der Form

`Caller.ActorName` = <*inselp-path*>

steht, muß der Bezeichner eines benannten K-Akteurs sein. Eine solche Gleichung hat genau dann den Wert *true*, wenn der mit dem Wert von `Caller.ActorName` identifizierte benannte K-Akteur gleich dem benannten K-Akteur ist, der mit dem INSEL$^+$–Pfad <*inselp-path*> identifiziert wird. Besteht der Zugriffsrestriktionsausdruck eines Views aus einer derartigen Gleichung, so folgt mit der Semantik von `Caller.ActorName`, daß lediglich der mit dem INSEL$^+$–Pfad identifizierte benannte K-Akteur das Recht zur Ausführung der äußeren Operationen des Views hat. Die äußeren Operationen des Views dürfen also sowohl im unmittelbaren als auch im mittelbaren Kontext der Ausführung der kanonischen Operation dieses K-Akteurs ausgeführt werden. Dies bedeutet, daß die äußeren Operationen des Views insbesondere auch im Kontext der Ausführung der Kommunikationsoperationen des K-Akteurs ausgeführt werden dürfen.

Beispiel

Würde der Zugriffsrestriktionsausdruck des Views `Create` des Depot–Generators `KontoTyp`

wie folgt lauten

`Create : Caller.ActorName = KontoVerwalter;`

so würde damit festgelegt, daß lediglich im Kontext der Ausführung der kanonischen Operation des K-Akteurs `KontoVerwalter` eine Inkarnation bzgl. des Generators `KontoTyp` erzeugt werden dürfte.

◇

Caller.ConGen

Der letzte Bezeichner G_n eines INSEL$^+$–Pfades, der auf der rechten Seite einer Gleichung der Form

`Caller.ConGen` = <*inselp-path*>

steht, muß der Bezeichner eines DA–Generators sein. Eine solche Gleichung hat genau dann den Wert *true*, wenn der mit dem Wert von `Caller.ConGen` identifizierte DA–Generator gleich dem DA–Generator ist, der mit dem INSEL$^+$–Pfad <*inselp-path*> identifiziert wird. Besteht der Zugriffsrestriktionsausdruck eines Views aus einer derartigen Gleichung, so folgt mit der Semantik von `Caller.ConGen`, daß ein Akteur lediglich dann das Recht zur Ausführung der äußeren Operationen des Views hat, wenn seine aktuelle Ausführungskomponente Inkarnation bzgl. des mit dem INSEL$^+$–Pfad identifizierten DA–Generators ist. Die äußeren Operationen des Views dürfen somit nur im unmittelbaren Kontext der Ausführung der kanonischen Operation einer DA–Inkarnation, die bzgl. des mit dem INSEL$^+$–Pfad identifizierten DA–Generators erzeugt wurde, ausgeführt werden. Führt ein Akteur die kanonische Operation einer solchen Inkarnation aus, so besitzt er also das Recht zur Ausführung der äußeren Operationen des Views. In diesem Sinne sind die Rechte, die mit einem Zugriffsrestriktionsausdruck, der lediglich aus einer Kontextrestriktion der obigen Form besteht, vergeben werden, generische Rechte (siehe [Eck93]).

Beispiel
Als Beispiel kann der für den View `Create` des Depot–Generators `KontoTyp` tatsächlich in dem INSEL$^+$–Programm des Kontenverwaltungssystems definierte Zugriffsrestriktionsausdruck dienen. Dieser lautet:

```
Create : Caller.ConGen = KontoVerwalter.KontoEroeffnen AND ...
```

Mit diesem Zugriffsrestriktionsausdruck wird festgelegt, daß lediglich im unmittelbaren Kontext der Ausführung der Kommunikationsoperation `KontoEroeffnen` des K-Akteurs `KontoVerwalter` eine Inkarnation bzgl. des Generators `KontoTyp` erzeugt werden darf. Im Vergleich zu dem im vorhergehenden Beispiel für den View `Create` angegebenen Zugriffsrestriktionsausdruck wird hier also eine restriktivere Beschränkung für die Erzeugung von Inkarnationen bzgl. des Generators `KontoTyp` festgelegt. In dem vorhergehenden Beispiel hat der K-Akteur `KontoVerwalter` generell, d.h. unabhängig von seiner jeweiligen Ausführungskomponente, das Recht zum Erzeugen einer Inkarnation bzgl. `KontoTyp` während er in diesem Beispiel dieses Recht nur im unmittelbaren Kontext der Ausführung seiner Kommunikationsoperation `KontoEroeffnen` besitzt.

◇

Caller.ConName

Der letzte Bezeichner G_n eines INSEL$^+$–Pfades, der auf der rechten Seite einer Gleichung der Form

`Caller.ConName` = <*inselp-path*>

steht, muß der Bezeichner einer benannten DA–Inkarnation, also entweder eines benannten Depots oder eines benannten K-Akteurs sein. Eine solche Gleichung hat genau dann den Wert *true*, wenn die mit dem Wert von `Caller.ConName` identifizierte benannte DA–Inkarnation der mit dem INSEL$^+$–Pfad <*inselp-path*> identifizieren DA–Inkarnation entspricht. Besteht der Zugriffsrestriktionsausdruck eines Views aus einer derartigen Gleichung, so folgt mit der Semantik von `Caller.ConName`, daß ein Akteur lediglich dann das Recht zur Ausführung der äußeren Operationen des Views hat, wenn seine aktuelle Ausführungskomponente gleich der DA–Inkarnation ist, die mit dem INSEL$^+$–Pfad identifiziert wird. Die äußeren Operationen des Views dürfen somit nur im unmittelbaren Kontext der Ausführung der kanonischen Operation der mit dem INSEL$^+$–Pfad identifizierten DA–Inkarnation ausgeführt werden.

Beispiel
Wäre der Zugriffsrestriktionsausdruck des Views `Create` des Depot–Generators `KontoTyp` wie folgt definiert

```
Create : Caller.ConName = KontoVerwalter;
```

so würde damit festgelegt, daß der K-Akteur `KontoVerwalter` lediglich im unmittelbaren Kontext seiner kanonischen Operation das Recht zum Erzeugen einer Inkarnation bzgl. des Generators `KontoTyp` besäße. Im Kontext der Ausführung seiner Kommunikationsoperationen sowie aller weiteren kanonischen Operationen passiver DA–Inkarnationen, die er ausführt, stünde dieses Recht jedoch nicht zur Verfügung.

◇

Mit den erklärten Arten von Kontextrestriktionen können unterschiedlich "starke" Restriktionen bezüglich der kanonischen Operationen, in deren Kontext die Ausführung einer äußeren Operation einer zugriffskontrollierten Komponente erlaubt sein soll, festgelegt werden. Mit einer `Caller.ActorGen`–Restriktion wird eine recht schwache Restriktion festgelegt, da diese sich auf die kanonischen Operationen einer ganzen Klasse von Akteuren sowie auf alle kanonischen Operationen, die von diesen Akteuren ausgeführt werden, bezieht. Demgegenüber wird mit einer `Caller.ConName`–Restriktion eine recht starke Restriktion festgelegt, da durch diese die Ausführung lediglich im unmittelbaren Kontext der Ausführung der kanonischen Operation einer spezifischen DA–Inkarnation erlaubt wird. Durch Konjunktion verschiedener Kontextrestriktionen ergeben sich weitere Differenzierungsmöglichkeiten im Hinblick auf die Stärke der damit festgelegten Restriktion. In Tabelle 4.3 ist zusammengefaßt, bei Ausführung welcher kanonischen Operationen die einzelnen Arten von Kontextrestriktionen erfüllt sind. Die Kontextrestriktionen sind dabei in aufsteigender Reihenfolge nach ihrer "Stärke" geordnet.

4.4.4.4 Bedingungen über lokale und globale DE–Inkarnationen

In einem Zugriffsrestriktionsausdruck können neben den mit den bisher erklärten Konstrukten festlegbaren Zugriffsbeschränkungen Bedingungen formuliert werden, deren Erfüllung von den Werten lokaler und globaler DE–Inkarnationen abhängig ist. Solche Bedingungen lassen sich durch Boolesche Teilausdrücke festlegen, die über den logischen Operatoren `AND`, `OR` und `NOT` sowie den auf den elementaren Datentypen definierten Vergleichsoperatoren gebildet werden. In einem derartigen Ausdruck sind im wesentlichen alle DE–Inkarnationen als Operanden zulässig, die in der Ausführungsumgebung der zugriffskontrollierten Komponente, für die der entsprechende Zugriffsrestriktionsausdruck definiert wird, liegen. Es gelten jedoch einige Besonderheiten, die von der Art der zugriffskontrollierten Komponente und dem View dieser Komponente, für den der Zugriffsrestriktionsausdruck festgelegt wird, abhängen. Diese Besonderheiten werden im folgenden erklärt.

Kontextrestriktion	erfüllt im
`Caller.ActorGen` = <*inselp-path*>	unmittelbaren und mittelbaren Kontext der kanonischen Operationen aller Akteure, die Inkarnationen bzgl. des mit <*inselp-path*> identifizierten Akteur–Generators sind
`Caller.ActorName` = <*inselp-path*>	unmittelbaren und mittelbaren Kontext der kanonischen Operation des mit <*inselp-path*> identifizierten benannten K-Akteurs
`Caller.ConGen` = <*inselp-path*>	unmittelbaren Kontext der kanonischen Operationen aller DA–Inkarnationen, die Inkarnationen bzgl. des mit <*inselp-path*> identifizierten DA–Generators sind
`Caller.ConName` = <*inselp-path*>	unmittelbaren Kontext der kanonischen Operation des mit <*inselp-path*> identifizierten benannten K-Akteurs bzw. benannten Depots

Tabelle 4.3: Kontextrestriktionen

Zunächst wird der Fall betrachtet, daß es sich bei dem Zugriffsrestriktionsausdruck um den Zugriffsrestriktionsausdruck eines Views eines explizit zugriffskontrollierten DA–Generators handelt. Dazu sei G ein explizit zugriffskontrollierter DA–Generator, der lokale N–Komponente einer DA–Inkarnation y ist. In den Zugriffsrestriktionsausdrücken der für G implizit definierten Views `Create`, `ChangeGenACL` und `ListACL` sind alle lokalen DE–Inkarnationen von y, die im Deklarationsteil von y textuell vor G deklariert sind, sowie alle nicht lokalen DE–Inkarnationen, die gemäß der Importfestlegung von y zur Ausführungsumgebung von y gehören, als Operanden zulässig. In dem Zugriffsrestriktionsausdruck des Views `Create` sind zusätzlich die formalen Eingabe– und Ein–Ausgabeparameter von G zulässig. Damit können in dem Zugriffsrestriktionsausdruck von `Create` Bedingungen angegeben werden, deren Erfüllung von den Werten der jeweils aktuellen Parameter eines Erzeugungskonstrukts bzgl. G abhängig ist. Das Recht zur Erzeugung einer Inkarnation bzgl. G kann somit abhängig von den Werten der aktuellen Parameter des jeweiligen Erzeugungskonstrukts vergeben werden. In den Zugriffsrestriktionsausdrücken der Views `ChangeGenACL` und `ListACL` sind analog die formalen Eingabe– und Ein–Ausgabeparameter der diesen Views entsprechenden Operationen zulässig. Damit kann z.B. das Vorhandensein des Rechts zur Ausführung der Operation `ChangeGenACL` und damit des Rechts zum Ändern der Einträge in der Zugriffskontrolliste von G von dem Wert des Parameters `AddOrDelete` abhängig sein, also davon, ob ein Eintrag eingefügt oder gelöscht werden soll.

In den Zugriffsrestriktionsausdrücken der Views eines zugriffskontrollierten Depots bzw. eines zugriffskontrollierten K-Akteurs x sind als Operanden die lokalen DE–Inkarnationen von x sowie alle nicht lokalen DE–Inkarnationen, die zur Ausführungsumgebung von x gehören, zulässig. In dem Zugriffsrestriktionsausdruck eines einelementigen Views, d.h. eines Views, der lediglich aus einer der für x definierten äußeren Operationen besteht, sind zusätzlich die formalen Eingabe– und Ein–Ausgabeparameter dieser äußeren Operation als Operanden zulässig. Es ist also möglich, bei Aufruf einer Zugriffsoperation bzw. einer Kommunikationsoperation eines zugriffskontrollierten Depots bzw. K-Akteurs, die einen eigenen View definiert, in Abhängigkeit der Werte der aktuellen Parameter des Operationsaufrufs zu entscheiden, ob

der aufrufende Akteur das Recht zur Ausführung der Operation hat oder nicht. Zu den einelementigen Views eines zugriffskontrollierten Depots bzw. K-Akteurs gehören insbesondere die implizit definierten Views `ChangeACL` und `ListACL`. Das Recht zur Änderung der Einträge der Zugriffskontrolliste eines zugriffskontrollierten Depots bzw. K-Akteurs kann also auch hier von den Werten der aktuellen Parameter des jeweiligen `ChangeACL`–Aufrufs abhängig sein.

Insgesamt lassen sich mit den zur Verfügung stehenden Möglichkeiten zur Formulierung von Bedingungen über lokale und globale DE–Inkarnationen komplexe Zugriffsbeschränkungen für die Nutzung zugriffskontrollierter Komponenten festlegen.

Beispiel

1. Eine Sicherheitsanforderung des Kontenverwaltungssystems lautet, daß Kunden lediglich dreimal am Tag einen Betrag von jeweils maximal 1000,– DM von einem Konto bar abheben dürfen. Ein Kunde darf also höchstens dann das Recht zur Ausführung der Zugriffsoperation `Abheben` auf einem Konto haben, wenn die Anzahl der bereits am selben Tag durch Kunden vorgenommenen Barabhebungen von dem Konto kleiner 3 ist und der abzuhebende Betrag 1000,– DM nicht übersteigt. Um diese Anforderung programmiersprachlich formulieren zu können, ist für jedes Konto–Depot eine lokale Integer–Variable `AnzahlBarAbhebungen` deklariert, die so verwaltet wird, daß sie die Anzahl der bereits an einem Tag durch Kunden vorgenommenen Barabhebungen von dem Konto angibt. Der abzuhebende Betrag wird in dem Eingabeparameter `Betrag` der Operation `Abheben` angegeben. Die Anforderung läßt sich dann in dem Zugriffsrestriktionsausdruck des Views `Abheben` wie folgt formulieren:

```
Abheben :  ... (Caller.Role = Kunde AND AnzahlBarAbhebungen < 3
           AND Betrag <= 1000) ...
```

2. Ein weiterer Bestandteil der Zugriffskontrollpolitik des Kontenverwaltungssystems ist die Festlegung, daß ein Kundenbetreuer lediglich zu den üblichen Schalterzeiten, d.h. werktags (montags bis freitags) in der Zeit zwischen 8.00 und 17.00 Uhr Zugriffsoperationen auf den Konten, die er betreut, ausführen darf. Die aktuelle Zeit, das Datum und der jeweilige Wochentag werden innerhalb des Kontenverwaltungssystem über die globale DE–Inkarnation `Zeit` zur Verfügung gestellt.[6] `Zeit` ist ein Record bestehend aus den Komponenten `Datum`, `Wochentag`, `Stunde`, `Minute` und `Sekunde`. Die oben für Kundenbetreuer angegebene Zugriffsbeschränkung wird in den Zugriffsrestriktionsausdrücken der für die Konto–Depots definierten Views durch folgenden Teilausdruck festgelegt:

```
Caller.Role = Kundenbetreuer AND 8 <= Zeit.Stunde AND Zeit.Stunde < 17
AND 1 <= Zeit.Wochentag AND Zeit.Wochentag < 6
```

3. In dem Kontenverwaltungssystem darf ein Kunde in beschränktem Maß Rechteänderungen an den Konten vornehmen, deren Inhaber er ist, indem er bestimmte Einträge in den Zugriffskontrollisten dieser Konten ändern darf. Er darf einem anderen Kunden erlauben, den Kontostand eines solchen Kontos zu lesen, Geld von dem Konto abzuheben sowie sich einen Kontoauszug des Kontos ausgeben zu lassen. Eine von ihm erteilte Erlaubnis kann er auch wieder zurücknehmen. Ein Kunde hat also lediglich das Recht, die Einträge für die Views `LeseKontostand`, `Abheben` und `ZeigeKontoauszug` in den Zugriffskontrollisten seiner Konten zu ändern.

[6] Um Mißverständnissen vorzubeugen, sei darauf hingewiesen, daß es sich hierbei nicht um die physikalische Zeit, sondern um eine virtuelle Zeit des Kontenverwaltungssystems handelt.

Diese Anforderung kann in dem Zugriffsrestriktionsausdruck des Views `ChangeACL` der Konto–Depots auf Basis der für jedes Konto–Depot deklarierten lokalen Konstante `InhaberUid` und des Eingabeparameters `ViewName` der Operation `ChangeACL` festgelegt werden. Die Konstante `InhaberUid` wird bei Erzeugung eines Konto–Depots mit dem Benutzeridentifikator des Inhabers des Kontos initialisiert. Der Parameter `ViewName` vom Typ `string` gibt den View an, dessen Zugriffskontrollisteneintrag durch einen `ChangeACL`–Aufruf geändert werden soll. Die obige Anforderung läßt sich damit in dem Zugriffsrestriktionsausdruck des Views `ChangeACL` der Konto–Depots wie folgt formulieren:

```
ChangeACL :  ... Caller.User = InhaberUid AND Caller.Role = Kunde AND
             (ViewName = 'LeseKontostand' OR ViewName = 'Abheben' OR
             ViewName = 'ZeigeKontoauszug') ...
```

◇

An dem letzten Beispiel ist gut erkennbar, daß das Recht zur Änderung der Einträge einer Zugriffskontrolliste und damit das Recht zur Durchführung von Rechteänderungen in INSEL$^+$–Systemen im Vergleich zu Systemen mit dem klassischen Owner–Prinzip sehr feingranular und flexibel vergeben werden kann. So kann, wie im obigen Beispiel, festgelegt werden, daß ein Benutzer lediglich das Recht hat, die Rechte zur Ausführung einer Teilmenge der äußeren Operationen einer zugriffskontrollierte Komponente zu ändern. In Systemen mit klassischem Owner–Prinzip ist eine solche Beschränkung nicht möglich. Dort hat der Owner einer Komponente per se das Recht zur Änderung der Rechte zur Ausführung aller äußeren Operationen dieser Komponente.

4.5 Konzept zur Behandlung von Zugriffsverboten

In INSEL$^+$–Systemen kann bei Aufruf einer äußeren Operation einer zugriffskontrollierten Komponente die Situation eintreten, daß die Zugriffsrestriktionsausdrücke aller Views, zu denen die äußere Operation gehört, zum Zeitpunkt des Operationsaufrufs für den aufrufenden Akteur den Wert *false* haben und somit der aufrufende Akteur nicht das Recht zur Ausführung der äußeren Operation hat. Für den aufrufenden Akteur liegt in diesem Fall ein sogenanntes **Zugriffsverbot** für die äußere Operation vor. Das Vorliegen eines Zugriffsverbots für eine äußere Operation hat zur Konsequenz, daß die Operation nicht ausgeführt wird, d.h. die durch Ausführung des entsprechenden Operationsaufrufs zu erzeugende DA–Inkarnation, deren kanonische Operation die Funktionalität der äußeren Operation definiert, nicht erzeugt wird. Um einem Akteur, der eine äußere Operation einer zugriffskontrollierten Komponente aufruft, ein vorliegendes Zugriffsverbot für diese Operation signalisieren zu können und eine Behandlung dieses Zugriffsverbots innerhalb des Akteurs zu ermöglichen, steht in INSEL$^+$ ein spezielles Konzept zur Verfügung.

Dieses Konzept besteht darin, daß für jeden Aufruf einer äußeren Operation einer zugriffskontrollierten Komponente eine boolesche Variable anzugeben ist, die nach Auswertung der Zugriffsrestriktionsausdrücke aller Views, zu denen die äußere Operation gehört, mit dem Wert dieser Auswertung belegt wird. Durch Abfrage des Wertes der booleschen Variable nach Ausführung des Operationsaufrufs kann innerhalb des aufrufenden Akteurs überprüft werden, ob ein Zugriffsverbot vorliegt und damit, ob die Operation ausgeführt wurde oder nicht. Falls ein Zugriffsverbot vorliegt, können Maßnahmen zur Behandlung des Zugriffsverbots durchgeführt werden. Diese Maßnahmen sind explizit vom Anwendungsentwickler zu programmieren. So kann zum Beispiel bei Vorliegen eines Zugriffsverbots im einfachsten

Fall eine Ausgabe auf dem abstrakten Terminal des dem Akteur zugeordneten Benutzers durchgeführt werden, die besagt, daß das Recht zur Ausführung der äußeren Operation nicht vorhanden ist. Ein komplexerer Fall liegt vor, wenn eine Teilberechnung, die für die Gesamtberechnung eines Akteurs benötigt wird, aufgrund eines Zugriffsverbots nicht ausgeführt werden kann. In diesem Fall kann die Behandlung des Zugriffsverbots zum Beispiel darin bestehen, Zustandsänderungen, die durch Teilberechnungen bewirkt wurden, die vor Auftreten des Zugriffsverbots ausgeführt wurden, wieder rückgängig zu machen. Je später ein Zugriffsverbot innerhalb der Ausführung einer komplexen Berechnung auftritt, desto aufwendiger ist im allgemeinen die Behandlung des Zugriffsverbots. Aus diesem Grunde ist es sinnvoll, bereits vor Beginn der Ausführung einer von einem Benutzerrepräsentanten durch einen entsprechenden Operationsaufruf angestoßenen Berechnung festzustellen, ob dieser Benutzerrepräsentant alle für die vollständige Ausführung dieser Operation benötigten Rechte hat. Ist dies nicht der Fall, so wird mit der Ausführung der Operation erst gar nicht begonnen, sondern dem Benutzerrepräsentanten ein Zugriffsverbot signalisiert. Dieses Zugriffsverbot kann dann innerhalb des Benutzerrepräsentanten einfach behandelt werden, indem eine entsprechende Meldung auf dem abstrakten Terminal des Benutzerrepräsentanten ausgegeben wird. Die Frage, ob bereits bei Aufruf einer Operation entschieden werden kann, ob alle für die vollständige Ausführung dieser Operation benötigten Rechte vorhanden sind, hängt unmittelbar mit der Frage der Konsistenz der in einem INSEL$^+$–System vergebenen Rechte zusammen. Auf diese Frage und auf Kriterien, anhand derer sich Aussagen über die Konsistenz der Rechte in einem INSEL$^+$–Systems gewinnen lassen, wird in Kapitel 5 eingegangen.

Zugriffsverbote sind spezielle Ausnahmen (*exceptions*), die als solche im Rahmen eines umfassenden Ausnahmebehandlungskonzepts (*exception handling*) von INSEL$^+$ behandelt werden sollten. Da die Entwicklung und Realisierung eines solchen Konzepts für INSEL$^+$ jedoch über den Rahmen dieser Arbeit hinausgeht, eine Behandlung von Zugriffsverboten aber möglich sein soll, wurde in INSEL$^+$ das oben erklärte spezielle Konzept eingeführt. In weiterführenden Arbeiten ist jedoch ein umfassendes Ausnahmebehandlungskonzept für INSEL$^+$ zu entwickeln, in das die Signalisierung und Behandlung von Zugriffsverboten zu integrieren ist. Als Orientierung bei der Entwicklung eines Ausnahmebehandlungskonzepts für INSEL$^+$ können die Ausnahmebehandlungskonzepte der Sprachen Ada ([Ada83], [Fel96]) und Guide ([BLR94], [Lac91]) dienen.

Sprachkonstrukt
Hinter jedem Erzeugungskonstrukt, das sich auf einen implizit oder explizit zugriffskontrollierten DA–Generator bezieht[7] ist ein **Status–Part** anzugeben, dessen Syntax wie folgt festgelegt ist:

<status-part> ::= `STATUS` *<boolean-name>*

<boolean-name> muß der Name einer booleschen Variable sein. Diese boolesche Variable wird bei Erarbeitung bzw. Ausführung des jeweiligen Erzeugungskonstrukts wie folgt belegt:

- Mit *true*, falls der das Erzeugungskonstrukt ausführende Akteur das Recht zur Ausführung der entsprechenden äußeren Operation hat, d.h., falls der Zugriffsrestriktionsausdruck mindestens eines Views, zu dem diese Operation gehört, den Wert *true* hat.

[7] Zur Erinnerung: Ein Erzeugungskonstrukt, das sich auf einen implizit zugriffskontrollierten DA–Generator bezieht, entspricht dem Aufruf einer Zugriffsoperation eines zugriffskontrollierten Depots oder einer Kommunikationsoperation eines zugriffskontrollierten K-Akteurs. Das Recht zur Ausführung einer solchen Zugriffs– bzw. Kommunikationsoperation entspricht dem Recht zur Ausführung der *erzeuge*–Operation auf dem implizit zugriffskontrollierten DA–Generator, der die Zugriffs– bzw. Kommunikationsoperation definiert.

- Mit *false*, falls für den das Erzeugungskonstrukt ausführenden Akteur ein Zugriffsverbot für die entsprechende äußere Operation vorliegt. Dies ist genau dann der Fall, wenn die Zugriffsrestriktionsausdrücke aller Views, zu dem diese äußere Operation gehört, für den Akteur den Wert *false* haben.

Der Wert der booleschen Variablen eines Status–Parts kann in einer textuell nach dem entsprechenden Erzeugungskonstrukt stehenden If–Anweisung[8] abgefragt werden. Die Maßnahmen zur Behandlung eines ggf. vorliegenden Zugriffsverbots sind in dem Zweig der If–Anweisung zu programmieren, der betreten wird, wenn die boolesche Variable den Wert *false* hat.

Beispiel

1. In Abbildung 4.15 ist ein Ausschnitt aus dem Implementierungsteil des zugriffskontrollierten M-Akteur–Generators `Kunde` angegeben. `Kunde` ist der Generator für die Benutzerrepräsentanten, die für Benutzer des Kontenverwaltungssystems, die in der Rolle *Kunde* agieren, erzeugt werden. In dem dargestellten Ausschnitt ist ein Aufruf der Zugriffsoperation `Abheben` auf einem Konto–Depot, das durch den Zeiger `Kontozeiger` identifiziert wird, angegeben. Falls für einen Benutzerrepräsentanten ein Zugriffsverbot für die Operation `Abheben` des mit `Kontozeiger` referenzierten Konto–Depots vorliegt, wird eine entsprechende Meldung auf dem abstrakten Terminal des Benutzerrepräsentanten ausgegeben.

```
-- Benutzerrepraesentant fuer Benutzer in der Rolle Kunde
PROTECTED PROCESS TYPE Kunde (Term: IN TermPointer)
  ...
  Kontozeiger       : KontoPtrTyp;
  Betrag            : real;
  ActError          : ErrorType;
  ErlaubterZugriff  : boolean;
  ...
BEGIN
  ...
  -- Aufruf der Zugriffsoperation Abheben
  KontoZeiger.Abheben("Barabhebung", Betrag, TRUE, ActError)
                                                    STATUS ErlaubterZugriff;
  IF NOT ErlaubterZugriff THEN
    -- Behandlung des Zugriffsverbots
    Term.out_str("Sie sind nicht berechtigt,Geld von diesem Konto abzuheben!");
  ELSIF ActError = 1 THEN
    ...
  ELSE
    Term.out_str("Auszahlung erfolgt!");
  END IF;
  ...
END Kunde;
```

Abbildung 4.15: Beispiel 1 für einen Status–Part

[8] Die Syntax der If–Anweisung von INSEL$^+$ ist in [RW96] angegeben.

2. Als Beispiel für einen Fall, in dem ein bei Ausführung einer Kommunikationsoperation eines zugriffskontrollierten K-Akteurs auftretendes Zugriffsverbot zur Terminierung der Operation führt und dieses Zugriffsverbot dem Aufrufer der Kommunikationsoperation über einen Ausgabeparameter "mitgeteilt" wird, ist in Abbildung 4.16 ein Ausschnitt aus dem Implementierungsteil des K-Order–Generators `KontoEroeffnen` angegeben. `KontoEroeffnen` wird innerhalb des zugriffskontrollierten K-Akteur–Generators `KontoVerwalterTyp` des Kontenverwaltungssystems definiert. Bei Ausführung der kanonischen Operation einer K-Order, die Inkarnation bzgl. `KontoEroeffnen` ist, soll ein neues Konto als anonymes Depot bzgl. des zugriffskontrollierten Depot–Generators `KontoTyp` erzeugt werden. Dazu muß der entsprechende K-Akteur im Kontext der Ausführung der kanonischen Operation dieser K-Order das Recht zum Erzeugen einer Inkarnation bzgl. `KontoTyp` haben. Ist dies nicht der Fall, liegt ein Zugriffsverbot für die Operation *erzeuge* des Generators `KontoTyp` vor, und es kann kein neues Konto erzeugt werden. Die Behandlung eines derartigen Zugriffsverbots besteht darin, daß die K-Order terminiert und dem Auftraggeber der K-Order über den Ausgabeparameter `Error` signalisiert wird, daß kein neues Konto erzeugt wurde. Durch Abfrage des Wertes dieses Ausgabeparameters durch den Auftraggeber kann dann eine weitere Behandlung des Zugriffsverbots innerhalb des Auftraggebers erfolgen.

```
-- Kommunikationsoperation KontoEroeffnen
ENTRY TYPE KontoEroeffnen (...; Error: OUT ErrorType)
  ...
  NeuerEintrag      : KontoListenEintragPtrTyp;
  ErlaubterZugriff : boolean;
  ...
BEGIN
  ...
  -- Erzeugen eines neuen Kontos
  NeuerEintrag := NEW(KontoListenEintragPtrTyp, KontoListenEintragTyp);
  NeuerEintrag.Konto := NEW(KontoPtrTyp, KontoTyp) (...)
                                              STATUS ErlaubterZugriff;
  IF NOT ErlaubterZugriff THEN
    Error := 2;  -- Zugriffsverletzung bei Ausfuehrung von KontoEroeffnen
  ELSE
    -- Einfuegen des neues Kontos in die Kontoliste
    ...
    Error := 0;  -- Konto wurde erzeugt und in Liste eingefuegt
  END IF;
END KontoEroeffnen;
```

Abbildung 4.16: Beispiel 2 für einen Status–Part

◇

Liegt bei der Erarbeitung bzw. Ausführung eines Erzeugungskonstrukts, das sich auf einen zugriffskontrollierten DA–Generator bezieht, ein Zugriffsverbot für die *erzeuge*–Operation des Generators vor, so hat dies zur Konsequenz, daß die durch Erarbeitung bzw. Ausführung des Erzeugungskonstrukts zu erzeugende DA–Inkarnation nicht erzeugt wird. Abhängig von dem Erzeugungskonstrukt und dem dieses umfassenden Sprachkonstrukt kann die Erarbeitung

bzw. Ausführung des Erzeugungskonstrukts bei Vorliegen eines Zugriffsverbots weitergehende Wirkungen haben. Im folgenden wird für die einzelnen Arten von Erzeugungskonstrukten informell angegeben, welche Wirkung ihre Erarbeitung bzw. Ausführung bei Vorliegen eines Zugriffsverbots hat.

1. K-Akteur– oder Depot–Deklaration:

   Bei Vorliegen eines Zugriffsverbots wird kein neuer K-Akteur bzw. kein neues Depot erzeugt. K-Akteur– und Depot–Deklarationen sind lediglich im Deklarationsteil eines DA–Generators zulässig. Die boolesche Variable des Status–Parts der Deklaration kann jedoch erst am Beginn des Anweisungsteils des DA–Generators, der die Deklaration enthält, überprüft werden, da im Deklarationsteil keine Anweisungen und damit auch keine Wertabfragen bzgl. der booleschen Variable zulässig sind. Ein Zugriffsverbot, daß bei Erarbeitung des Deklarationsteils einer DA–Inkarnation auftritt, kann also frühestens zu Beginn der Ausführung des Anweisungsteils der DA–Inkarnation behandelt werden. Aus diesem Grund dürfen K-Akteure und Depots, die bei Erarbeitung eines Deklarationsteils erzeugt werden (sollen), nicht bereits im Deklarationsteils genutzt werden.

2. Generierungsausdruck bzgl. eines K-Akteur– bzw. Depot–Generators:

   Ein solcher Generierungsausdruck ist entweder Bestandteil einer Zeiger–Deklaration mit Initialisierungsausdruck oder einer Zeiger–Zuweisung. Eine Zeiger–Deklaration mit Initialisierungsausdruck ist lediglich im Deklarationsteil und eine Zeiger–Zuweisung nur im Anweisungsteil eines DA–Generators zulässig. Bei Vorliegen eines Zugriffsverbots wird kein neuer anonymer K-Akteur bzw. kein neues anonymes Depot erzeugt und der Wert des Zeigers der Zeiger–Deklaration bzw. der Zeiger–Zuweisung bleibt unverändert. Bzgl. der Behandlung eines Zugriffsverbots, das bei Erarbeitung einer Zeiger–Deklaration mit Initialisierungsausdruck auftritt, gilt das unter 1. Gesagte entsprechend. Ein Zugriffsverbot, das bei Ausführung einer Zeiger–Zuweisung auftritt, kann unmittelbar nach der Zeiger–Zuweisung behandelt werden.

3. M-Akteur–Aufruf:

   Bei Vorliegen eines Zugriffsverbots wird kein neuer M-Akteur erzeugt. Die Behandlung des Zugriffsverbots kann unmittelbar nach dem M-Akteur–Aufruf erfolgen.

4. PS-Order oder K-Order–Aufruf:

   Bei Vorliegen eines Zugriffsverbots wird keine neue PS-Order bzw. K-Order erzeugt. Handelt es sich um einen K-Order–Aufruf, wird insbesondere kein Auftrag an den K-Akteur, auf den sich der Aufruf bezieht, erteilt. Die Behandlung des Zugriffsverbots kann unmittelbar nach dem PS-Order bzw. K-Order–Aufruf erfolgen.

5. FS-Order–Aufruf:

   Bei Vorliegen eines Zugriffsverbots wird keine neue FS-Order erzeugt. Da ein FS-Order–Aufruf immer in einem Ausdruck auftritt, hat ein bei Ausführung eines FS-Order–Aufrufs vorliegendes Zugriffsverbot weitergehende Konsequenzen. Tritt bei der Auswertung eines Ausdrucks ein Zugriffsverbot bei der Ausführung eines FS-Order Aufrufs auf, so wird die Auswertung des Ausdrucks abgebrochen. Mit dem Abbruch der Auswertung des Ausdrucks terminiert die Ausführung des umfassenden Sprachkonstrukts, das diesen Ausdruck enthält. Im folgenden wird für die einzelnen Sprachkonstrukte, die Ausdrücke

enthalten können, erklärt, welche Wirkung ihre Ausführung bei Abbruch der Auswertung eines ihrer Ausdrücke aufgrund eines Zugriffsverbots hat. Die Bezeichnungen der einzelnen Sprachkonstrukte entsprechen den in [RW96] eingeführten Bezeichnungen und die jeweils in Klammern angegebene Nummer entspricht der Nummer der in [RW96] für das jeweilige Sprachkonstrukt angegebenen Syntaxregel. Für jedes Sprachkonstrukt wird an einem kurzen Beispiel verdeutlicht, an welchen Stellen Ausdrücke und damit FS-Order–Aufrufe zulässig sind. Dazu sei `BspFunkt` der Bezeichner eines zugriffskontrollierten FS-Order–Generators, dessen Inkarnationen einen Rückgabewert vom Typ `integer` liefern, und `b` eine boolesche Variable.

(a) Zuweisung (Regel 74):
In einer Zuweisung steht ein Ausdruck auf der rechten Seite. Bei Abbruch der Auswertung des Ausdrucks bleibt der Wert der Variablen auf der linken Seite der Zuweisung unverändert.

Beispiel:
```
x := BspFunkt(...) STATUS b + 5;
```

(b) If–Anweisung (Regel 64):
In einer If–Anweisung sind die Bedingungen des If–Teils und des Elsif–Teils boolesche Ausdrücke. Bei Abbruch der Auswertung eines dieser Ausdrücke terminiert die If-Anweisung, ohne daß einer ihrer Anweisungsteile ausgeführt wird.

Beispiel:
```
IF BspFunkt(...) STATUS b < 5 THEN ... ELSE ... END IF;
```

(c) Case–Anweisung (Regel 68):
Die auszuführende Case–Alternative wird abhängig von dem Wert eines Ausdrucks bestimmt. Wird die Auswertung dieses Ausdrucks abgebrochen, terminiert die Case–Anweisung, ohne daß eine der Case–Alternativen ausgeführt wird.

Beispiel:
```
CASE BspFunkt(...) STATUS b IS
  WHEN 1 .. 100 DO ...;
  WHEN OTHERS   DO ...;
END CASE;
```

(d) While–Anweisung (Regel 63):
Die Bedingung einer While–Anweisung ist ein boolescher Ausdruck. Wird die Auswertung dieses Ausdrucks abgebrochen, terminiert die Ausführung der While-Anweisung.

Beispiel:
```
WHILE BspFunkt(...) STATUS b < 5 LOOP ... END LOOP;
```

(e) Select–Anweisung (Regel 50):
In einer Select–Anweisung sind die Bedingungen der Auswahlalternativen boolesche Ausdrücke. Diese Ausdrücke werden in der ersten Phase der Ausführung der Select–Anweisung ausgewertet. Wird die Auswertung eines dieser Ausdrücke abgebrochen, terminiert die Select–Anweisung, ohne daß eine der Annahmealternativen ausgeführt wird.

Beispiel:
```
SELECT
  WHEN BspFunkt(...) STATUS b < 5 DO ACCEPT ...
  ...
END SELECT;
```

(f) Return–Anweisung (Regel 75):
Bei Abbruch der Auswertung des Ausdrucks der Return–Anweisung terminiert die entsprechende FS-Order nicht mit dieser Return–Anweisung. Die Behandlung des Zugriffsverbots, das zum Abbruch der Auswertung des Ausdrucks führt, muß in einem Anweisungsteil erfolgen, der unmittelbar nach der Return–Anweisung steht und der eine weitere Return–Anweisung enthalten muß. Die Anweisungen dieses Anweisungsteils werden nur dann ausgeführt, wenn die Auswertung des Ausdrucks der Return–Anweisung abgebrochen wird. Damit ist eine Behandlung von Zugriffsverboten, die bei Ausführung der Return–Anweisung einer FS-Order auftreten, innerhalb der FS-Order möglich. Der Fall, daß nach einer Return-Anweisung, in deren Ausdruck ein FS-Order–Aufruf bzgl. eines zugriffskontrollierten FS-Order–Generators enthalten ist, keine Anweisungen mehr stehen, ist nicht zulässig.

Beispiel:
```
FUNCTION TYPE Demo RETURN integer IS
  ...
  RETURN BspFunkt(...) STATUS b + 5;
  Term.Output("Zugriffsverbot bei FS-Order Ausfuehrung");
  RETURN -1;
END Demo;
```

Wird die Auswertung des Ausdrucks der ersten Return–Anweisung aufgrund eines vorliegenden Zugriffsverbots für `BspFunkt` abgebrochen, wird zunächst eine entsprechende Meldung ausgegeben und anschließend die zweite Return–Anweisung ausgeführt und damit -1 als Ergebnis der `Demo`–Ausführung zurückgegeben.

(g) DE–Objektdeklaration (Regel 36):
Enthält eine DE–Objektdeklaration einen nicht leeren Initialisierungsteil, so besteht dieser aus einem Ausdruck. Bei Abbruch der Auswertung dieses Ausdrucks, wird die deklarierte DE–Inkarnation nicht initialisiert, d.h. ihr Wert bleibt undefiniert.

Beispiel:
```
x : integer := BspFunkt(...) STATUS b;
```

(h) Aktueller Parameter–Part (Regel 40):
Ein aktueller Parameter–Part ist immer Bestandteil eines Erzeugungskonstrukts bzgl. eines DA–Generators, also einer DA–Objektdeklaration, eines Generierungsausdrucks, eines M-Akteur–Aufrufs oder eines PS-Order–, K-Order– oder FS–Order–Aufrufs. Er besteht aus einer Liste von Ausdrücken, die die Werte der aktuellen Parameter der zu erzeugenden DA–Inkarnation bestimmen. Wird die Auswertung eines der Ausdrücke des Parameter–Parts abgebrochen, wird die Erarbeitung bzw. Ausführung des entsprechenden Erzeugungskonstrukts ebenfalls abgebrochen. Insbesondere wird die zu erzeugende DA–Inkarnation nicht erzeugt. Bzgl. weiterer Wirkungen dieses Abbruchs und der Behandlung des Zugriffsverbots, das zu diesem Abbruch führt, gilt das unter den Punkten 1. bis 5. Gesagte entsprechend.

Beispiel:
```
FORK BspMActor(BspFunkt(...) STATUS b);
IF b THEN
  ...
ELSE
  -- Behandlung des Zugriffsverbots
END IF;
```

(i) Array–Komponentenname (Regel 91):
In einem Array–Komponentennamen werden die Indizes durch die Auswertung von Ausdrücken bestimmt. Es wird festgelegt, daß in den Ausdrücken eines Array–Komponentennames keine FS-Order–Aufrufe bzgl. zugriffskontrollierter FS-Order–Generatoren auftreten dürfen. Damit können bei der Auswertung der Ausdrücke eines Array–Komponentennames keine Zugriffsverbote auftreten, die zum Abbruch der Auswertung dieser Ausdrücke führen. Die Festlegung wird getroffen, da Array–Komponentennamen als Bestandteile von Komponentennamen in einem INSEL$^+$–Programm an vielen Stellen auftreten können. Könnte es bei der Erarbeitung eines Array–Komponentennamens zu dem Abbruch der Auswertung eines in diesem Namen enthaltenen Ausdrucks aufgrund eines Zugriffsverbots kommen, müßte für alle Sprachkonstrukte, die Namen enthalten können, die Wirkung bei Auftreten eines Zugriffsverbots festgelegt werden. Darauf wurde aus Vereinfachungsgründen in dieser Arbeit verzichtet. Im Rahmen der Entwicklung eines umfassenden Ausnahmebehandlungskonzepts für INSEL$^+$ ist jedoch für das Auftreten von Ausnahmen bei der Erarbeitung von Namen eine Lösung zu erarbeiten.

Die Festlegung, daß in einem Array–Komponentennamen keine FS-Order–Aufrufe bzgl. zugriffskontrollierter FS-Order–Generatoren zulässig sind, bedeutet keine funktionale Einschränkung. Der Wert, der durch einen solchen FS-Order–Aufruf in einem Array–Komponentennamen berechnet werden soll, kann auch vor Erarbeitung des Array–Komponentennamens berechnet werden, indem der FS-Order–Aufruf in einer Zuweisung ausgeführt wird und das Ergebnis der entsprechenden FS-Order einer Hilfsvariablen zugewiesen wird, die an Stelle des FS-Order–Aufrufs in dem Array–Komponentennamen auftritt.

Beispiel: Der Array–Komponentenname in der folgenden Zuweisung ist nicht zulässig.

```
x := a[BspFunkt(...) STATUS b, 10];
```

Der durch den Aufruf von `BspFunkt` zu berechnende erste Index des Array–Komponentennamens ist stattdessen wie folgt zu berechnen:

```
hilf := BspFunkt(...) STATUS b;
IF b THEN
  x := a[hilf, 10];
ELSE
  -- Behandlung des Zugriffsverbots
END IF;
```

Ein Zugriffsverbot, das bei Auswertung eines Ausdrucks auftritt, der in einem der in (a) bis (i) genannten Sprachkonstrukte enthalten ist, kann abhängig davon, ob es sich bei dem Sprachkonstrukt um eine Anweisung oder eine Deklaration handelt, entweder unmittelbar nach der Anweisung oder erst am Beginn des entsprechenden Anweisungsteils

behandelt werden. Sind in einem Ausdruck mehrere FS-Order–Aufrufe bzgl. zugriffskontrollierter FS-Order–Generatoren enthalten, so ist hinter jedem dieser FS-Order–Aufrufe ein Status–Part anzugeben. Für jeden dieser Status–Parts sollte eine eigene boolesche Variable deklariert werden, damit bei Abbruch des Ausdrucks festgestellt werden kann, welcher FS-Order–Aufruf durch ein vorliegendes Zugriffsverbot zum Abbruch geführt hat und welche spezifischen Behandlungsmaßnahmen dementsprechend durchführen zu sind.

4.6 Beispiele

In diesem Abschnitt wird anhand zweier Beispiele verdeutlicht, wie die in den vorhergehenden Abschnitten dieses Kapitels eingeführten Sprachkonzepte und –konstrukte zur Implementierung von Systemen mit anwendungsspezifisch festgelegten Zugriffskontrollpolitiken eingesetzt werden können. In dem ersten Beispiel wird das Kontenverwaltungssystem einer Bank beschrieben, auf das in den Beispielen, die zur Erläuterung der einzelnen Sprachkonstrukte angegeben wurden, bereits mehrfach Bezug genommen wurde. Bei dem zweiten Beispiel handelt es sich um ein aus der Literatur bekanntes Anwendungssystem, und zwar um ein System zur Verwaltung der Hausaufgaben und Musterlösungen einer vorlesungsbegleitenden Übung in Form elektronischer Dokumente. Dieses Beispielsystem wird in der Literatur häufig als Beispiel für ein System mit einer anwendungsspezifischen Zugriffskontrollpolitik betrachtet (z.B. [TNT92], [Kü95]).

4.6.1 Kontenverwaltungssystem einer Bank

Im folgenden werden zunächst die funktionalen Anforderungen des zu konstruierenden Kontenverwaltungssystems informell angegeben. Anschließend wird die Zugriffskontrollpolitik des Systems informell spezifiziert. Es sei darauf hingewiesen, daß sowohl die funktionalen Anforderungen als auch die Festlegungen der Zugriffskontrollpolitik lediglich ausschnittsweise angegeben sind, um das Beispiel überschaubar zu halten. Die Anforderungen, die an ein in einer Bank "real" einsetzbares Kontenverwaltungssystem zu stellen sind, gehen sowohl in funktionaler als auch in qualitativer Hinsicht weit über die hier genannten hinaus.
Nach der informellen Beschreibung der funktionalen Anforderungen und der Zugriffskontrollpolitik des Kontenverwaltungssystems wird für einige der Anforderungen und Politikfestlegungen beispielhaft aufgezeigt, wie diese mit den INSEL$^+$–Konzepten umgesetzt wurden. Das vollständige INSEL$^+$–Programm, das das Kontenverwaltungssystem implementiert, ist in Anhang B aufgelistet.

4.6.1.1 Funktionale Anforderungen

Das Kontenverwaltungssystem soll die Möglichkeit bieten, Konten für Kunden einer Bank eröffnen zu können und diese ggf. auch wieder auflösen zu können. Die wesentliche Funktionalität des Kontenverwaltungssystems soll darin bestehen, die Daten der vorhandenen Konten korrekt zu verwalten. Zu den Daten, die für ein Konto zu verwalten sind, gehören u.a. der aktuelle Kontostand, der maximale Überziehungskredit, die Kontobewegungen sowie die Daten des Kontoinhabers. Das Kontenverwaltungssystem soll sowohl von Angestellten der Bank als auch von Kunden der Bank über Terminals benutzbar sein. So sollen zum Beispiel Kunden der Bank die Möglichkeit haben, Geld von ihren Konten abzuheben sowie sich den Kontostand

ihrer Konten anzeigen zu lassen. Die Benutzer des Kontenverwaltungssystem agieren gemäß ihrer Aufgabe und Funktion in unterschiedlichen Rollen. Aus den Aufgabenbeschreibungen für die einzelnen Rollen, die im folgenden angegeben werden, ergeben sich die wesentlichen funktionalen Eigenschaften des Kontenverwaltungssystems, die an der Schnittstelle des Systems zur Verfügung stehen sollen.
Für das Kontenverwaltungssystem sind die folgenden fünf Rollen definiert:

- **Kunde**
  Ein Kunde ist eine Person, die Inhaber eines Kontos bei der Bank ist. Die Kunden sollen über Kundenterminals die Möglichkeit haben, Bargeld von ihren Konten abzuheben, den Kontostand ihrer Konten zu lesen, sich einen Kontoauszug ausgeben zu lassen sowie Überweisungen von ihren Konten auf andere Konten der Bank tätigen zu können.

- **Kundenbetreuer**
  Ein Kundenbetreuer ist ein Bankangestellter, der für die Eröffnung und Auflösung von Konten sowie für die Ermittlung des maximalen Überziehungskredits eines Kontos zuständig ist. Für letztere Aufgabe muß er die Möglichkeit haben, den Kontostand, die Kontobewegungen innerhalb bestimmter Zeiträume sowie die persönlichen Daten des Inhabers eines Kontos (wie z.B. dessen regelmäßige Einkünfte) zu ermitteln. Desweiteren soll er die Daten eines Kontoinhabers (wie z.B. dessen Adresse oder regelmäßige Einkünfte) ändern sowie Überweisungen durchführen können. Ein Kundenbetreuer soll jeweils nur für die Betreuung der Konten zuständig sein, die er eröffnet hat. Ihm kann die Betreuung weiterer Konten übertragen werden, z.B. für die Konten eines anderes Kundenbetreuers, falls dieser krank ist oder sich im Urlaub befindet.

- **Kassierer**
  Ein Kassierer ist ein Bankangestellter, der für die Bareinzahlung und Barauszahlung von Geldbeträgen an der Bankkasse zuständig ist. Um diese Aufgabe erfüllen zu können, muß er die Möglichkeit haben, Geldbeträge auf Konten einzahlen bzw. von Konten abheben zu können.

- **Bankleiter**
  Die Rolle des Bankleiters ist die Rolle mit den weitestgehenden Kompetenzen. Zu den Aufgaben des Bankleiters gehört u.a. die Festsetzung des maximalen Überziehungskredits eines Kontos. Um den von dem Kundenbetreuer eines Kontos berechneten Überziehungskredit überprüfen zu können, muß der Bankleiter selbst in der Lage sein, diesen Überziehungskredit zu ermitteln. Dazu muß er die Möglichkeit haben, den Kontostand und die Kontobewegungen von Konten sowie die persönlichen Daten der Kontoinhaber einsehen zu können. Weiterhin ist der Bankleiter für die Übertragung der Betreuung eines Kontos an einen anderen Kundenbetreuer als denjenigen, der das Konto eröffnet hat, zuständig.

- **SysAdmin**
  Die Aufgabe des Systemadministrators besteht u.a. in dem Anlegen und Löschen von Kennungen für die Benutzer des Kontenverwaltungssystems. Weitere Aufgaben, die üblicherweise von einem Systemadministrator wahrgenommen werden, werden in diesem Beispiel nicht betrachtet. Ebenso wird auf die Funktionalität des Benutzerzugangs und der Benutzerdatenverwaltung nicht weiter eingegangen. Hierzu sei auf die zu diesen Punkten in Abschnitt 4.2 gemachten Anmerkungen verwiesen.

Neben den Aktionen, die von Benutzern des Kontenverwaltungssystems ausgelöst werden, sollen innerhalb des Kontenverwaltungssystems jeweils am Monatsende automatisch die Zinsen für die einzelnen Konten berechnet werden und bei einem "Haben" dem jeweiligen Konto gutgeschrieben bzw. einem "Soll" von diesem abgehoben werden. Die Zinsen sollen dabei von einem zentralen Konto der Bank auf das jeweilige Kundenkonto bzw. von dem Kundenkonto auf das zentrale Bankkonto überwiesen werden.

4.6.1.2 Zugriffskontrollpolitik

Die Rechtefestlegungen der Zugriffskontrollpolitik des Kontenverwaltungssystems werden im folgenden informell, gruppiert nach Festlegungen bzgl. der einzelnen Rollen, angegeben.

- **Kunde**
  Es ist zu gewährleisten, daß ein Kunde lediglich Geld von den Konten abheben kann, deren Inhaber er ist, d.h. die auf seinen Antrag hin eröffnet wurden, bzw. von den Konten, für die er explizit das Recht zum Abheben von anderen Kunden erhalten hat. Dies bezieht sich sowohl auf Barabhebungen als auch auf Abhebungen, die im Kontext von Überweisungen vorgenommen werden. Bei Barabhebungen ist darauf zu achten, daß pro Konto höchstens dreimal am Tag ein Betrag von jeweils maximal 1000,– DM von Kunden selbständig abgehoben werden darf. Ein Kunde darf lediglich den Kontostand der Konten lesen bzw. sich Kontoauszüge der Konten ausgeben lassen, deren Inhaber er ist bzw. für die er explizit die entsprechenden Rechte erhalten hat. Weiterhin ist sicherzustellen, daß ein Kunde kein Geld unmittelbar auf ein Konto einzahlen kann. Ein Kunde darf in beschränktem Maße Rechteänderungen an den Konten vornehmen, deren Inhaber ist. Und zwar darf er einem anderen Kunden erlauben, den Kontostand eines seiner Konten zu lesen, sich einen Kontoauszug ausgeben zu lassen sowie Geld von diesem Konto abzuheben. Er darf die von ihm vergebenen Rechte auch wieder zurücknehmen. Das Recht eines Kontoinhabers, die genannten Rechteänderungen an seinem Konto vorzunehmen, kann ihm von dem Bankleiter entzogen werden.

- **Kassierer**
  Ein Kassierer ist berechtigt Barein– und Barauszahlungen auf bzw. von Konten vorzunehmen. Dies darf er jedoch nur zu den üblichen Schalterzeiten, also werktags (montags bis freitags) zwischen 8.00 und 17.00 Uhr. Wie in der Realität auch, darf ein Kassierer Ein– und Auszahlungen auf ein Konto nur bei Vorliegen eines entsprechenden Auftrags von dem Kundenbetreuer oder dem Inhaber des Kontos vornehmen. Aus Vereinfachungsgründen wird dieser Auftrag innerhalb des hier betrachteten Kontenverwaltungssystems jedoch nicht erfaßt. Stattdessen wird die Vertrauenswürdigkeit der Kassierer postuliert, d.h. es wird davon ausgegangen, daß sie nur bei Vorliegen eines Auftrags von ihrem Recht zum Einzahlen und Abheben Gebrauch machen. Ein Kassierer hat weder das Recht, die persönlichen Daten von Konteninhabern zu lesen noch das Recht, sich Kontoauszüge oder Kontobewegungen ausgeben zu lassen. Kassierer sind nicht berechtigt, Rechteänderungen in dem Kontenverwaltungssystem durchzuführen.

- **Kundenbetreuer**
  Kundenbetreuer dürfen Konten eröffnen und auflösen. Ein Kundenbetreuer hat jedoch nur dann das Recht, ein Konto aufzulösen, wenn er für die Betreuung des Kontos zuständig ist. Es ist sicherzustellen, daß ein Kundenbetreuer weder Geld von einem Konto bar abheben noch Beträge auf ein Konto bar einzahlen kann. Er darf lediglich im Kontext von Überweisungen Geldbeträge von Konten, die er betreut, abheben und

auf andere Konten der Bank überweisen. Geldbeträge von anderen Konten darf er auch im Kontext von Überweisungen nicht abheben. Ein Kundenbetreuer hat das Recht, sich die Kontostände der Konten, die er betreut, anzeigen zu lassen. Außerdem hat er das Recht, den maximalen Überziehungskredit für ein von ihm betreutes Konto zu berechnen. Dazu darf er im Kontext einer solchen Berechnung die persönlichen Daten des Kontoinhabers sowie die Kontobewegungen des Kontos innerhalb bestimmter Zeiträume lesen. Weiterhin ist ein Kundenbetreuer berechtigt, die variablen Daten der Inhaber der von ihnen betreuten Konten (wie z. B. die Adresse oder die regelmäßigen Einkünfte) zu ändern. Generell dürfen Kundenbetreuer nur Rechte an den Konten haben, für deren Betreuung sie zuständig sind. Weiterhin dürfen sie lediglich zu den üblichen Schalterzeiten, also werktags zwischen 8.00 und 17.00 Uhr, mit dem Kontenverwaltungssystem arbeiten. Rechteänderungen dürfen Kundenbetreuer in dem Kontenverwaltungssystem nicht durchführen.

- **Bankleiter**
  Der Bankleiter hat als einziger das Recht, den maximalen Überziehungskredit für ein Konto festzusetzen. Dabei kann er sich auf das Ergebnis, das ein das Konto betreuender Kundenbetreuer im Rahmen der Berechnung des Überziehungskredits ermittelt hat, stützen oder er kann diese Berechnung selbst durchführen, um z.B. das Ergebnis des Kundenbetreuers zu überprüfen. Dazu darf er im Kontext der Berechnung des maximalen Überziehungskredits für ein Konto auf die persönlichen Daten des Kontoinhabers sowie die Kontobewegungen dieses Kontos zugreifen. Der Bankleiter darf die Rechte, die ein Kundenbetreuer an einem Konto hat, das er betreut, an andere Kundenbetreuer weitergeben. Damit kann der Bankleiter die Betreuung eines Kontos an einen anderen Kundenbetreuer als denjenigen, der das Konto eröffnet hat, übertragen. Durch Zurücknehmen dieser Rechte kann er einem Kundenbetreuer die Betreuung eines Kontos entziehen. Der Bankleiter ist berechtigt einem Kontoinhaber alle Rechte an seinem Konto zu entziehen. Hierzu gehört insbesondere das Recht eines Kontoinhabers, Rechteänderungen bzgl. der Operationen zum Abheben, zum Kontostand lesen und zum Kontoauszug anzeigen an seinem Konto durchführen zu dürfen. Damit ist es dem Bankleiter möglich, ein Konto für jeglichen Zugriff durch einen Kunden zu sperren. Die Tätigkeit des Bankleiters innerhalb des Kontenverwaltungssystems ist nicht auf die üblichen Schalterzeiten begrenzt.

- **SysAdmin**
  Ein Systemadministrator ist berechtigt, Benutzerkennungen anzulegen und diese wieder zu löschen. Er darf Operationen des Kontenverwaltungssystems, die nicht mit der Benutzerverwaltung zusammenhängen, nicht ausführen, also weder Konten eröffnen oder auflösen noch auf Konten zugreifen.

Neben den bereits genannten Rechtefestlegungen gilt weiterhin, daß der Prozeß bzw. die Prozesse, die jeweils am Monatsende die Zinsen für die einzelnen Konten berechnen, das Recht haben, den jeweiligen Zinsbetrag von einem Konto abzuheben bzw. auf ein Konto einzuzahlen. Dies gilt sowohl für die Kundenkonten als auch für das zentrale Bankkonto.

4.6.1.3 INSEL^{+}–Implementierung

In der in Anhang B angegebenen INSEL^{+}–Implementierung des Kontenverwaltungssystems sind die Konten als zugriffskontrollierte anonyme Depots realisiert. Die Konten werden von

einem zentralen Kontoverwalter, der als zugriffskontrollierter benannter K-Akteur implementiert ist, in einer Liste verwaltet. Bestandteil jedes Kontos ist ein zugriffskontrolliertes benanntes Depot, in dem die Kontobewegungen des Kontos gespeichert werden. Die automatische Zinsberechnung am Monatsende wird von einem zugriffskontrollierten M-Akteur gesteuert, der von dem zentralen Kontoverwalter erzeugt wird. Dieser M-Akteur erzeugt am Monatsende für jedes existierende Konto einen M-Akteur, der die Zinsen für dieses Konto berechnet und die entsprechenden Zinsbeträge abbucht bzw. einzahlt. Die Zinsberechnung am Monatsende wird somit parallel für alle Konten durchgeführt. Für jede Rolle des Systems ist ein M-Akteur–Generator für die Benutzerrepräsentanten der in der Rolle agierenden Benutzer definiert. Jeder Benutzerrepräsentant stellt dem Benutzer ein rollenspezifisches Menü zur Verfügung, über das dieser die mit der jeweiligen Rolle verbundenen Aufgaben und Funktionen ausführen kann.

Zentraler Kontoverwalter

Der zentrale Kontoverwalter stellt Operationen zum Eröffnen, zum Auflösen sowie zum Auffinden eines Kontos zur Verfügung. In Abbildung 4.17 ist der Spezifikationsteil des zugriffskontrollierten K-Akteur–Generators `KontoVerwalterTyp` angegeben, bzgl. dem der zentrale Kontoverwalter inkarniert wird.

```
PROTECTED TASK TYPE SPEC KontoVerwalterTyp
  IMPORT ...;
IS
  ENTRY TYPE SPEC KontoEroeffnen (KontoInhaber: IN KundenDatenPtrTyp;
                                  KontoNummer : OUT integer);
  ENTRY TYPE SPEC KontoAufloesen (KontoZeiger : IN KontoPtrTyp;
                                  Error: OUT ErrorType);
  ENTRY TYPE SPEC GibKontoZeiger (KontoNummer : IN integer;
                                  KontoZeiger: OUT KontoPtrTyp;
                                  Error : OUT ErrorType);

  ACCESS RESTRICTIONS
    KontoEroeffnen : (Caller.Role = Kundenbetreuer AND
                      8 <= Zeit.Stunde AND Zeit.Stunde < 17 AND
                      1 <= Zeit.Wochentag AND Zeit.Wochentag < 6) OR
                     (Caller.User = System AND
                      KontoInhaber.AllgKundenDaten.Nachname = "Zentralkonto");
    KontoAufloesen : Caller.Role = Kundenbetreuer AND
                     IN_ACL(Caller.User, Caller.Role, KontoZeiger, Aufloesen)
                     AND 8 <= Zeit.Stunde AND Zeit.Stunde < 17
                     AND 1 <= Zeit.Wochentag AND Zeit.Wochentag < 6;
    GibKontoZeiger : NOT (Caller.Role = SysAdmin);
    Create         : Caller.User = System;
END KontoVerwalterTyp;
```

Abbildung 4.17: Spezifikationsteil des K-Akteur–Generators `KontoVerwalterTyp`

Die sich aus der Zugriffskontrollpolitik ergebenen Rechtefestlegungen bzgl. der Operationen des Kontoverwalters werden durch die für diese Operationen definierten Zugriffsrestriktionsausdrücke implementiert.

Der Zugriffsrestriktionsausdruck für die Operation `KontoEroeffnen` besagt, daß alle Benutzer, die in der Rolle des Kundenbetreuers agieren, werktags in der Zeit zwischen 8.00 Uhr und 17.00 Uhr das Recht zum Eröffnen eines neuen Kontos haben. Damit wird die Festlegung der Zugriffskontrollpolitik des Kontenverwaltungssystems, daß lediglich Kundenbetreuer innerhalb der üblichen Schalterzeiten das Recht zum Eröffnen eines Kontos haben, umgesetzt. Darüberhinaus legt der Zugriffsrestriktionsausdruck der Operation `KontoEroeffnen` fest, daß ausschließlich der abstrakte Benutzer *System* das Recht hat, das zentrale Bankkonto zu eröffnen. Dem entspricht, daß dieses Konto bei der Initialisierung des Kontenverwaltungssystems von der Hauptkomponente erzeugt wird.

Gemäß der Zugriffskontrollpolitik hat ein Kundenbetreuer zu den üblichen Schalterzeiten nur dann das Recht zum Auflösen eines Kontos, wenn er für die Betreuung des Kontos zuständig ist. Diese Festlegung wird in dem Zugriffsrestriktionsausdruck der Operation `KontoAufloesen` durch das `IN_ACL`–Prädikat umgesetzt. Dieses `IN_ACL`–Prädikat hat genau dann den Wert *true*, wenn in der Zugriffskontrolliste des aufzulösenden Kontos in der Benutzer–Rollen–Liste der Operation `Aufloesen` ein entsprechender Eintrag für den aufrufenden Kundenbetreuer vorhanden ist. Ein solcher Eintrag existiert jedoch nur für die das Konto betreuenden Kundenbetreuer.

In der Zugriffskontrollpolitik werden keine Festlegungen dahingehend getroffen, daß einem Kundenbetreuer das generelle Recht zum Erzeugen und Auflösen von Konten entzogen werden kann oder Benutzer in anderen Rollen dieses Recht erhalten können. Dem entspricht, daß dieses Recht in den Zugriffsrestriktionsausdrücken der Operationen `KontoEroeffnen` und `KontoAufloesen` statisch an die Rolle des Kundenbetreuers vergeben wird und Rechteänderungen durch Änderung der Einträge der Zugriffskontrolliste des zentralen Kontoverwalters nicht möglich sind. Letzteres ist daraus ersichtlich, daß für die Operation `ChangeACL` kein Zugriffsrestriktionsausdruck explizit definiert ist und dieser somit implizit aus dem booleschen Literal *false* besteht.

Die Operation `GibKontoZeiger` dient dazu, in der Kontenliste des Kontoverwalters ein Konto anhand seiner Kontonummer zu finden. Falls ein Konto mit der als Eingabeparameter übergebenen Kontonummer in der Liste vorhanden ist, wird ein Zeiger darauf an die aufrufende Komponente zurückgegeben. Dieser Zeiger ist Voraussetzung für den Aufruf von Operationen auf dem Konto. Die Operation `GibKontoZeiger` muß somit von allen Benutzern aufrufbar sein, die Operationen auf Konten ausführen dürfen. Gemäß der Zugriffskontrollpolitik gilt dies für alle Benutzer mit Ausnahme der Systemadministratoren. Der Zugriffsrestriktionsausdruck von `GibKontoZeiger` ist entsprechend definiert.

Der Zugriffsrestriktionsausdruck für die Operation `Create` des K-Akteur-Generators `KontoVerwalterTyp` legt fest, daß der zentrale Kontoverwalter nur von dem abstrakten Benutzer *System*, d. h. ausgehend von der Hauptkomponente und nicht ausgehend von einem Benutzerrepräsentanten erzeugt werden darf.

Konten

Auf den Konten sind neben den Operationen zum Einzahlen und Abheben eines Betrages, zum Lesen des aktuellen Kontostands und der Kontonummer sowie zum Ausgeben eines Kontoauszugs bzw. der Kontobewegungen innerhalb eines Zeitraums Operationen zum Lesen und Ändern der Daten des Kontoinhabers sowie zum Ausgeben und Setzen des maximalen Überziehungskredits definiert. Im folgenden wird beispielhaft für die Operation zum Abheben eines Betrages aufgezeigt, wie die mit der Zugriffskontrollpolitik gegebenen Rechtefestlegungen für diese Operation umgesetzt wurden. Weiterhin wird verdeutlicht, wie die Festlegungen bzgl. der Durchführung von Rechteänderungen an einem Konto implementiert

wurden. In Abbildung 4.18 sind die hierfür relevanten Ausschnitte aus dem Spezifikationsteil des zugriffskontrollierten Depot–Generators für die Konten angegeben. Der vollständige Spezifikationsteil kann dem Programm in Anhang B entnommen werden.

```
PROTECTED DEPOT TYPE SPEC KontoTyp (KontoNummer  : IN integer;
                                    InhaberDaten : IN KundenDatenPtrTyp;
                                    InhaberUid   : IN UserUidType;
                                    BetreuerUid  : IN UserUidType)
  IMPORT ...;
IS
  PROCEDURE TYPE SPEC Abheben (Zweck: IN string; Betrag: IN real;
                               Flag: IN boolean; Error: OUT ErrorType);
  ...

  AnzahlBarAbhebungen : integer := 0;

  ...

  ACL
    Abheben   : (InhaberUid, Kunde, +), (BetreuerUid, Kundenbetreuer, +),
                (-1, Kassierer, +);
    ChangeACL : (InhaberUid, Kunde, +);
    ...

  ACCESS RESTRICTIONS
    Abheben   : (IN_ACL(Caller.User, Caller.Role, THIS, THIS) AND
                 ((Caller.Role = Kunde AND AnzahlBarAbhebungen < 3 AND
                   Betrag <= 1000) OR
                  (Caller.Role = Kunde AND Caller.ConGen = Ueberweiser) OR
                  (Caller.Role = Kundenbetreuer AND Caller.ConGen = Ueberweiser
                   AND 8 <= Zeit.Stunde AND Zeit.Stunde < 17 AND
                   1 <= Zeit.Wochentag AND Zeit.Wochentag < 6) OR
                  (Caller.Role = Kassierer AND
                   8 <= Zeit.Stunde AND Zeit.Stunde < 17 AND
                   1 <= Zeit.Wochentag AND Zeit.Wochentag < 6))) OR
                Caller.ActorGen = Zinsberechnung.Zinsberechner;
    ChangeACL : (IN_ACL(Caller.User, Caller.Role, THIS, THIS) AND
                 Caller.User = InhaberUid AND Caller.Role = Kunde AND
                 (ViewName = 'LeseKontostand' OR ViewName = 'Abheben' OR
                  ViewName = 'ZeigeKontoauszug')) OR
                Caller.Role = Bankleiter;
    ...
END KontoTyp;
```

Abbildung 4.18: Ausschnitt aus dem Spezifikationsteil des Depot–Generators `KontoTyp`

Gemäß der Zugriffskontrollpolitik des Kontenverwaltungsssystems sind für die Operation zum Abheben eines Betrags von einem Konto die folgenden Rechtefestlegungen gegeben.

- Ein Kunde hat das Recht zum Abheben eines Betrags von einem Konto, wenn er

  - Inhaber des Kontos ist und ihm dieses Recht nicht durch den Bankleiter entzogen wurde oder
  - er dieses Recht explizit von dem Kontoinhaber erhalten hat und ihm dieses Recht nicht wieder durch den Kontoinhaber oder den Bankleiter entzogen wurde.

  Im Fall einer Barabhebung hat ein Kunde jedoch nur dann das Recht zum Abheben, wenn noch nicht dreimal am Tag Geld bar von dem Konto abgehoben wurde und der abzuhebende Betrag höchstens 1000,– DM beträgt. Bei Abhebungen im Kontext von Überweisungen gelten diese zusätzlichen Restriktionen nicht.

- Ein Kundenbetreuer hat während der üblichen Schalterzeiten das Recht zum Abheben eines Betrags von einem Konto, wenn er für die Betreuung des Kontos zuständig ist und die Abhebung im Kontext einer Überweisung durchgeführt wird.
- Ein Kassierer hat während der üblichen Schalterzeiten das Recht zum Abheben eines Betrags von einem Konto, wenn es sich um eine Barabhebung handelt.
- Der M-Akteur, der am Monatsende die Zinsen für ein Konto berechnet, hat im Fall eines negativen Zinsbetrags das Recht, diesen Betrag von dem Konto abzuheben.

Diese Rechtefestlegungen werden durch den Zugriffsrestriktionsausdruck der Operation `Abheben` implementiert. Da das Recht zur Ausführung dieser Operation an Kunden, Kundenbetreuer und Kassierer dynamisch vergeben werden kann, enthält der Zugriffsrestriktionsausdruck am Anfang ein entsprechendes `IN_ACL`–Prädikat. Der Zugriffskontrollisteneintrag der Operation `Abheben` wird bei Erzeugung eines Kontos so initialisiert, daß das `IN_ACL`–Prädikat lediglich für den Kunden, der Inhaber des Kontos ist (Parameter `InhaberUid`), den Kundenbetreuer, der das Konto eröffnet hat (Parameter `BetreuerUid`), und alle Kassierer den Wert *true* liefert. Die beiden Parameter `InhaberUid` und `BetreuerUid` werden bei der Erzeugung eines Kontos mit den Werten der entsprechenden Benutzeridentifikatoren belegt. Rechteänderungen bzgl. der Operation `Abheben` können über den Aufruf der Operation `ChangeACL` durch Änderung des Zugriffskontrollisteneintrags von `Abheben` durchgeführt werden. Dazu sind entsprechende Rechte an der Operation `ChangeACL` vergeben, die weiter unten noch genauer erläutert werden.

Die Zusatzbedingung, daß Kunden höchstens dreimal am Tag einen Betrag von jeweils maximal 1000,– DM bar von einem Konto abheben dürfen, wird in dem Zugriffsrestriktionsausdruck durch den Teilausdruck:

```
(Caller.Role = Kunde AND AnzahlBarAbhebungen < 3 AND Betrag <= 1000)
```

implementiert. Dabei ist `AnzahlBarAbhebungen` eine lokale Integer–Variable eines Kontos, die so verwaltet wird, daß sie die Anzahl der bereits an einem Tag durch Kunden vorgenommenen Barabhebungen von dem Konto angibt. `Betrag` ist der Eingabeparameter der Operation `Abheben`, der den abzuhebenden Betrag bestimmt.

Die Festlegung, daß für Kunden im Kontext von Überweisungen außer dem Vorhandensein eines entsprechenden Zugriffskontrollisteneintrags keine weitere Restriktionen bzgl. des Abhebens eines Betrags von einem Konto bestehen, wird durch den Teilausdruck:

```
(Caller.Role = Kunde AND Caller.ConGen = Ueberweiser)
```

umgesetzt. `Ueberweiser` ist der Bezeichner des zugriffskontrollierten M-Akteur–Generators, bzgl. dem eine Inkarnation zur Durchführung einer Überweisung erzeugt wird.

Der Teilausdruck des Zugriffsrestriktionsausdrucks der Operation `Abheben`, der mit `(Caller.Role = Kundenbetreuer ...)` beginnt, implementiert zusammen mit dem `IN_ACL`–Prädikat das Recht eines Kundenbetreuers zum Abheben eines Betrags von einem Konto. Die Festlegung, daß Kundenbetreuer lediglich im Kontext von Überweisungen Geldbeträge abheben dürfen, wird dabei durch die Kontextrestriktion `Caller.ConGen = Ueberweiser` umgesetzt.

Das Recht eines Kassierers zum Abheben eines Betrags im Rahmen einer Barauszahlung wird durch den mit `(Caller.Role = Kassierer ...)` beginnenden Teilausdruck implementiert. Damit ist jedoch noch nicht die implizite Rechtefestlegung umgesetzt, daß Kassierer im Kontext von Überweisungen nicht das Recht zum Abheben haben. Diese Festlegung wird durch den Zugriffsrestriktionsausdruck der Operation `Create` des M-Akteur–Generators `Ueberweiser` implementiert. Dieser Zugriffrestriktionausdruck ist so definiert, daß Kassierer generell kein Recht zur Erzeugung einer Inkarnation bzgl. des Generators `Ueberweiser` bekommen können und somit überhaupt keine Überweisungen durchführen können.

Die Festlegung, daß die M-Akteure, die am Monatsende die Zinsen berechnen, das Recht zum Abheben eines möglichen negativen Zinsbetrags von einem Konto haben, wird in dem Zugriffsrestriktionsausdruck der Operation `Abheben` durch die Kontextrestriktion `Caller.ActorGen = Zinsberechnung.Zinsberechner` implementiert. `Zinsberechnung` ist der Bezeichner des Generators für den zugriffskontrollierten M-Akteur, der von dem zentralen Kontoverwalter erzeugt wird und die automatische Zinsberechnung am Monatsende steuert. `Zinsberechner` ist der M-Akteur–Generator bzgl. dem jeweils am Monatsende für jedes Konto eine Inkarnation zur Berechnung und Abbuchung bzw. Einzahlung der entsprechenden Zinsbeträge erzeugt wird.

Gemäß der Zugriffskontrollpolitik des Kontenverwaltungssystems können an einem Konto folgende Rechteänderungen vorgenommen werden:

- Der Kunde, der Inhaber des Kontos ist, darf – sofern ihm dieses Recht nicht durch den Bankleiter entzogen wurde – einem anderen Kunden jeweils das Recht geben,
  - den Kontostand des Kontos zu lesen,
  - sich den Kontoauszug des Kontos anzeigen zu lassen,
  - einen Betrag von dem Konto abzuheben.
- Der Kunde, der Inhaber des Kontos ist, darf – sofern ihm dieses Recht nicht durch den Bankleiter entzogen wurde – von ihm an andere Kunden vergebene Rechte wieder zurücknehmen.
- Der Bankleiter hat das Recht, beliebige Rechteänderungen an dem Konto durchzuführen, sofern diese nicht durch anderweitige Festlegungen der Zugriffskontrollpolitik ausgeschlossen sind.[9]

Rechteänderungen an einem Konto können durch Aufruf der Operation `ChangeACL` durch Änderung der Einträge der Zugriffskontrolliste des Kontos durchgeführt werden. Dementsprechend werden die obigen Festlegungen bzgl. der Durchführung von Rechteänderungen an einem Konto durch den Zugriffsrestriktionsausdruck der Operation `ChangeACL` umgesetzt.

Das Recht eines Kunden, in dem oben angegebenen Rahmen Rechteänderungen an einem Konto vorzunehmen, wird durch den ersten Teilausdruck des Zugriffsrestriktionsausdrucks

[9] So darf er z.B. nicht das Recht eines Kunden zur Durchführung von Rechteänderungen an einem Konto, dessen Inhaber dieser ist, weitergeben.

von ChangeACL implementiert. Durch die darin enthaltenen drei Bedingungen, die sich auf den Wert des Eingabeparameters ViewName der Operation ChangeACL beziehen, wird dieses Recht auf die Operationen LeseKontostand, ZeigeKontoauszug und Abheben eingeschränkt. Die in dem Teilausdruck definierte Benutzerrestriktion Caller.User = InhaberUid gewährleistet zusammen mit der Rollenrestriktion Caller.Role = Kunde, daß ein Kunde höchstens dann Rechte für diese drei Operationen an einem Konto ändern darf, wenn er Inhaber des Kontos ist. Da ihm dieses Recht jedoch durch den Bankleiter entzogen werden kann, enthält der Teilausdruck zusätzlich ein IN_ACL–Prädikat. Der Zugriffskontrollisteneintrag der Operation ChangeACL wird bei Erzeugung eines Kontos so initialisiert, daß dieses IN_ACL–Prädikat lediglich für den Kontoinhaber zu *true* ausgewertet wird. Durch Löschen des entsprechenden Benutzer–Rollen–Eintrags aus der Zugriffskontrolliste kann der Bankleiter dem Kontoinhaber das Recht zur Durchführung von Rechteänderungen entziehen, da das IN_ACL–Prädikat dann den Wert *false* liefert. Enthielte der Zugriffsrestriktionsausdruck von ChangeACL dieses IN_ACL–Prädikat nicht, so wäre das Recht zur Ausführung von ChangeACL statisch an den Kontoinhaber vergeben und könnte ihm nicht entzogen werden.

Durch den zweiten Teilausdruck des Zugriffsrestriktionsausdrucks von ChangeACL, der lediglich aus der Rollenrestriktion Caller.Role = Bankleiter besteht, wird das Recht zur Durchführung aller Rechteänderungen an einem Konto, die durch Änderung der Einträge der Zugriffskontrolliste des Kontos möglich sind, statisch an die Rolle die Bankleiters vergeben.

4.6.2 System zur Hausaufgabenverwaltung eines Übungskurses

Als zweites Beispiel wird ein Anwendungssystem betrachtet, in dem Studenten Lösungen einer Hausaufgabe in Form elektronischer Dokumente an einen Server übergeben sollen. Dieser Server verwaltet ebenfalls eine Musterlösung der Hausaufgabe, die ein Student allerdings erst dann einsehen darf, wenn er seine Lösung der Hausaufgabe an den Server übergeben hat. Ein Student darf keine neue Version seiner Lösung der Hausaufgabe mehr nachreichen, nachdem er die Musterlösung eingesehen hat. Die Zugriffskontrollpolitik für dieses System ist informell also wie folgt festgelegt:

- Ein Student darf Lösungen zu der Hausaufgabe solange abgeben, bis er einmal auf die Musterlösung zugegriffen hat.
- Ein Student darf die Musterlösung erst dann einsehen, wenn er eine eigene Lösung der Hausaufgabe abgegeben hat.

In INSEL$^+$ kann der Hausaufgaben–Server als zugriffskontrolliertes Depot realisiert werden[10], das zwei Zugriffsoperationen zur Verfügung stellt: eine zum Abgeben einer Lösung der Hausaufgabe (LoesungAbgeben) und eine zum Erhalten der Musterlösung (MusterloesungEinsehen). Die Lösungen sind als Depots vom Typ DocumentType zu erstellen und werden über Zeiger an das Hausaufgaben–Depot übergeben. Die mit der Zugriffskontrollpolitik gegebenen Rechtefestlegungen für die beiden Zugriffsoperationen können mit den Konzepten von INSEL$^+$ auf zwei unterschiedliche Arten implementiert werden. Zum einen unter Verwendung lokaler boolescher DE–Inkarnationen, die angeben, ob ein Student bereits eine Lösung abgegeben hat bzw. bereits die Musterlösung eingesehen hat. Zum anderen unter Verwendung des Zugriffsrestriktionsprädikats ACCESSED.

[10]Eine Realisierung als zugriffskontrollierter K-Akteur wäre ebenfalls möglich. Hier ist jedoch die Realisierung als Depot gewählt worden, da der Hausaufgaben–Server eher einer Datenbank entspricht, in die etwas abgelegt werden kann bzw. aus der etwas ausgelesen werden kann.

4.6.2.1 Lösung unter Verwendung boolescher DE–Inkarnationen

Bei dieser Lösung (vgl. Abbildung 4.19) werden in dem Hausaufgaben–Depot zwei eindimensionale lokale Arrays deklariert, die für jeden Studenten durch einen booleschen Wert angeben, ob dieser bereits eine Lösung der Hausaufgabe abgegeben hat (Array `Abgegeben`) bzw. ob dieser bereits die Musterlösung eingesehen hat (Array `LoesungGesehen`). Um die Zugriffskontrollpolitik korrekt umzusetzen, müssen die Werte dieser Arrays explizit verwaltet werden.

```
PROTECTED DEPOT TYPE SPEC HausaufgabenVerwaltungsTyp IS
  TYPE StudentenArrayTyp IS ARRAY[UserUidType] OF boolean;
  LoesungGesehen : StudentenArrayTyp;
  Abgegeben      : StudentenArrayTyp;
  ...

  PROCEDURE TYPE SPEC LoesungAbgeben (Loesung: IN DocumentPtr);

  PROCEDURE TYPE SPEC MusterloesungEinsehen (Loesung: OUT DocumentPtr);

  ACCESS RESTRICTIONS
    LoesungAbgeben : NOT LoesungGesehen[Caller.User];
    MusterloesungEinsehen : Abgegeben[Caller.User];
    Create : Caller.User = System;
END HausaufgabenVerwaltungsTyp;

PROTECTED DEPOT TYPE HausaufgabenVerwaltungsTyp IS
  ...

  PROCEDURE TYPE LoesungAbgeben (Loesung: IN DocumentPtr) IS
  BEGIN
    ...
    Abgegeben[Caller.User] := TRUE;
  END LoesungAbgeben;

  PROCEDURE TYPE MusterloesungEinsehen (Loesung: OUT DocumentPtr) IS
  BEGIN
    ...
    LoesungGesehen[Caller.User] := TRUE;
  END MusterloesungEinsehen;

BEGIN
  FOR i IN UserUidType LOOP
    LoesungGesehen[i] := FALSE;
    Abgegeben[i] := FALSE;
  END LOOP;
  ...
END HausaufgabenVerwaltungsTyp;
```

Abbildung 4.19: Implementierung des Hausaufgabenverwaltungssystems unter Verwendung boolescher DE–Inkarnationen

Es ist zu beachten, daß die im Spezifikationsteil von `HausaufgabenVerwaltungsTyp` deklarierten lokalen DE-Komponenten `StudentenArrayTyp, LoesungGesehen` und `Abgegeben` gemäß der in Abschnitt 4.4.1 getroffenen Festlegungen nicht mit dem Attribut E definiert sind (und damit auch nicht von außen benutzbar sind), da es sich hier um einen zugriffskontrollierten Depot–Generator handelt.

4.6.2.2 Lösung unter Verwendung des Zugriffsrestriktionsprädikats `ACCESSED`

Bei dieser Lösung (vgl. Abbildung 4.20) werden die mit der Zugriffskontrollpolitik gegebenen Rechtefestlegungen des Hausaufgabenverwaltungssystems unmittelbar mit dem Zugriffsrestriktionsprädikat `ACCESSED` implementiert. Der Vorteil dieser Lösung im Vergleich zu der im vorhergehenden Abschnitt angegebenen Implementierung besteht darin, daß hier zur Durchsetzung der Zugriffskontrollpolitik keine zusätzlichen lokalen DE–Komponenten deklariert und verwaltet werden müssen.

```
PROTECTED DEPOT TYPE SPEC HausaufgabenVerwaltungsTyp IS

  PROCEDURE TYPE SPEC LoesungAbgeben (Loesung: IN DocumentPtr);

  PROCEDURE TYPE SPEC MusterloesungEinsehen (Loesung: OUT DocumentPtr);

  ACCESS RESTRICTIONS
    LoesungAbgeben : NOT ACCESSED(Caller.User, THIS, MusterloesungEinsehen);
    MusterloesungEinsehen : ACCESSED(Caller.User, THIS, LoesungAbgeben);
    Create : Caller.User = System;;
END HausaufgabenVerwaltungsTyp;

PROTECTED DEPOT TYPE HausaufgabenVerwaltungsTyp IS
  ...
  PROCEDURE TYPE LoesungAbgeben (Loesung: IN DocumentPtr) IS
  BEGIN
    ...
  END LoesungAbgeben;

  PROCEDURE TYPE MusterloesungEinsehen (Loesung: OUT DocumentPtr) IS
  BEGIN
    ...
  END MusterloesungEinsehen;

BEGIN
  ...
END HausaufgabenVerwaltungsTyp;
```

Abbildung 4.20: Implementierung des Hausaufgabenverwaltungssystems unter Verwendung des Zugriffsrestriktionsprädikats `ACCESSED`

4.7 Zusammenfassung

Mit den in diesem Kapitel erklärten Konzepten der Sprache INSEL$^+$ steht ein Repertoire von Konzepten auf hohem Abstraktionsniveau zur Implementierung von Systemen mit anwendungsspezifisch festgelegten Zugriffskontrollpolitiken zur Verfügung.

Im weiteren stellt sich die Frage, wie diese Konzepte systematisch zur Implementierung einer für ein zu realisierendes System informell oder formal spezifizierten Zugriffskontrollpolitik einzusetzen sind. Ist die Zugriffskontrollpolitik des Systems lediglich informell formuliert, liegt die Aufgabe der programmiersprachlichen Realisierung der Politikfestlegungen mit den INSEL$^+$–Konzepten allein in der Hand des Systementwicklers. Leitlinien für die systematische Umsetzung der Politikfestlegungen einschließlich entsprechender Werkzeugunterstützung sind in diesem Fall aufgrund der fehlenden formalen Spezifikation der Zugriffsbeschränkungen nicht bzw. nur sehr eingeschränkt möglich. Liegt die Zugriffskontrollpolitik des Systems hingegen in formal spezifizierter Form vor, so liefert das dabei verwendete Spezifikationsinstrumentarium eine Basis für die Erarbeitung einer Methodik zur systematischen Realisierung der festgelegten Zugriffsbeschränkungen mit den INSEL$^+$–Konzepten. Eine derartige Methodik bzw. entsprechende Regeln für die Transformation der formalen Spezifikation in eine programmiersprachliche Realisierung ist stark von dem Instrumentarium abhängig, das zur Modellierung des Systemverhaltens und der Spezifikation der Zugriffskontrollpolitik eingesetzt wird. Die Konzepte von INSEL$^+$ sind so gewählt, daß durch sie die systematische Implementierung von Zugriffsbeschränkungen, die mit dem im Rahmen des SecreDS–Projekts[11] entwickelten Spezifikationsinstrumentarium ([Eck93], [Eck95]) formal festgelegt sind, besonders unterstützt wird. So sind die Möglichkeiten, die in INSEL$^+$ zur Festlegung von Zugriffsrestriktionsausdrücken zur Verfügung stehen, abgestimmt auf die Bedingungen, die mit den Zugriffsrestriktionsformeln der Logik–Sprache, in der in SecreDS Sicherheitseigenschaften spezifiziert werden, formulierbar sind. Im Kontext des SecreDS–Projekts wurden dementsprechend bereits erste Leitlinien für die systematische Implementierung von Zugriffsbeschränkungen, die mit der Logik–Sprache von SecreDS formal spezifiziert sind, in INSEL$^+$ erarbeitet ([EM97]). Da die INSEL$^+$–Konzepte weitestgehend auf das Modell und die Spezifikationskonzepte von SecreDS zugeschnitten sind, kann bei kombiniertem Einsatz des SecreDS–Spezifikationsinstrumentariums und INSEL$^+$ als Implementierungssprache die in Abschnitt 2.2 aufgezeigte Lücke zwischen der formalen Spezifikation von Sicherheitseigenschaften und deren Realisierung deutlich verkleinert werden.

Neben der Problematik der systematischen Implementierung einer spezifizierten Zugriffskontrollpolitik mit dem eingeführten Konzeptevorrat von INSEL$^+$ stellt sich die Frage nach der Konsistenz der gemäß dieser Zugriffskontrollpolitik in dem System vergebenen Rechte. Die programmiersprachliche Formulierung der Zugriffsbeschränkungen ermöglicht es insbesondere, im Rahmen von statischen und dynamischen Analysen werkzeugunterstützt Aussagen über die Konsistenz der auf Basis dieser Beschränkungen vergebenen Rechte zu gewinnen. In dem folgenden Kapitel wird ausführlich auf das Thema der Konsistenz der in einem INSEL$^+$–System vergebenen Rechte eingegangen.

[11] *Secure Distributed Systems*

Kapitel 5

INSEL$^+$–Systeme mit konsistenten Rechtefestlegungen

Mit den im vorhergehenden Kapitel erklärten Konzepten der Sprache INSEL$^+$ können Systeme konstruiert werden, die Komponenten enthalten, für deren Nutzung Rechte explizit vergeben sind. Diese Rechte sind auf Basis der Zugriffsrestriktionsausdrücke, die für die Views der zugriffskontrollierten Komponenten des Systems festgelegt sind, vergeben. Betrachtet man die Menge der in einem System vergebenen Rechte, so ist insbesondere von Interesse, ob diese Rechte widerspruchsfrei, d. h. konsistent – in einem noch zu präzisierenden Sinn – sind.

In der Literatur über zugriffskontroll–basierte Systeme gibt es bisher nur sehr wenige Arbeiten, in denen intensiver auf das Problem der Konsistenz der in derartigen Systemen vergebenen Rechte eingegangen wird. Insbesondere ist in diesem Bereich noch keine einheitliche Begriffsbildung erkennbar, d.h. es wird unterschiedliches darunter verstanden, wann die in einem System vergebenen Rechte konsistent vergeben sind. In Abschnitt 5.1 wird ein kurzer Überblick über einige dieser Arbeiten und über die unterschiedlichen Konsistenzbegriffe gegeben. Als Grundlage für die weiteren Abschnitte dieses Kapitels wird in Abschnitt 5.2 der Begriff des **Rechts eines INSEL$^+$–Systems** eingeführt, und es wird definiert, wann das Recht eines INSEL$^+$–Systems **konsistent** ist. Der definierte Konsistenzbegriff ist sehr weitreichend. Er besagt informell folgendes: ein Akteur, der das Recht zur Ausführung einer äußeren Operation *op* einer Komponente hat, muß im Kontext der *op*–Ausführung auch das Recht zur Ausführung aller Operationen haben, die im Kontext der *op*–Ausführung auszuführen sind. Ist dies nicht gewährleistet, liegt eine Inkonsistenz in dem Recht des Systems vor, die dazu führt, daß die Ausführung von *op* aufgrund eines fehlenden Rechts für eine im Kontext der *op*–Ausführung auszuführende Operation abgebrochen werden muß und damit die Funktionalität von *op* nicht vollständig erbracht werden kann. Die Konsistenz des Rechts eines INSEL$^+$–Systems ist also Voraussetzung, um die mit der Wahrnehmung eines Rechts verbundene Leistungsanforderung (siehe [Spi85]) erfüllen zu können. Dementsprechend sind Maßnahmen von Interesse, die dazu beitragen können, die Konsistenz des Rechts eines INSEL$^+$–Systems zu gewährleisten. Das Spektrum derartiger Maßnahmen wird in Abschnitt 5.2 aufgezeigt.

Zu den Maßnahmen gehören insbesondere **statische Konsistenzanalysen**, durch die ausgehend von einem INSEL$^+$–Programm Aussagen über die Konsistenz des Rechts des durch dieses Programm definierten INSEL$^+$–Systems gewonnen werden können. Dabei wird unterschieden zwischen **potentieller Konsistenz** und **absoluter Konsistenz**. Das Recht eines INSEL$^+$–Systems ist potentiell konsistent, wenn es keine statisch feststellbaren Inkonsistenzen

enthält. Es ist absolut konsistent, wenn die Konsistenz statisch nachgewiesen werden kann. Die Kriterien, deren Erfüllung im Rahmen der statischen Konsistenzanalysen überprüft wird, werden in Abschnitt 5.3 erarbeitet.

Das vorrangige Ziel dieses Kapitels besteht darin, Leitlinien anzugeben, an denen sich ein Systementwickler orientieren kann, wenn er ein INSEL$^+$–System mit konsistentem Recht konstruieren möchte. In Abschnitt 5.4 werden entsprechende Leitlinien, die aus den Kriterien für die potentielle und absolute Konsistenz abgeleitet sind, angegeben.

5.1 Literaturüberblick

Wie bereits einleitend gesagt, wird dem Problem der Konsistenz der in einem zugriffskontroll–basierten System vergebenen Rechte in der Literatur bisher wenig Aufmerksamkeit geschenkt. Einige Arbeiten, die sich explizit mit diesem Problem beschäftigen, sind z.B. [BJS96], [Lau95], [Bru93], [Kel90], [WL93], [Eck93] und [Spi85]. Die ersten vier dieser Arbeiten behandeln die Konsistenzproblematik speziell im Kontext zugriffskontroll–basierter Datenbank–Systeme während sich die zuletzt genannten Arbeiten mit dieser Problematik im Bereich der verteilten Systeme beschäftigen. Die in den zitierten Arbeiten verwendeten Konsistenzbegriffe unterscheiden sich zum Teil recht stark voneinander. Im folgenden wird ein Überblick über die verschiedenen Konsistenzbegriffe gegeben.

5.1.1 Objektlokaler Konsistenzbegriff: Konsistenz bezogen auf positive und negative Rechte an einem Objekt

In [BJS96], [Lau95], [Bru93] sowie [WL93] werden Konsistenzbegriffe festgelegt, die sich auf die in den jeweils betrachteten Systemen bestehende Möglichkeit beziehen, sowohl positive als auch negative Rechte an Subjekte vergeben zu können. Durch ein positives Recht wird ein Zugriff explizit erlaubt, während durch ein negatives Recht ein Zugriff explizit verboten wird. Positive Rechte werden deshalb auch als Erlaubnisse (engl.: *permissions*) und negative Rechte als Verbote (engl.: *prohibitions*) bezeichnet. Ein Konflikt bzw. ein Widerspruch liegt dann vor, wenn an ein Subjekt gleichzeitig ein positives und ein negatives Recht bzgl. eines Zugriffs auf ein Objekt vergeben ist. Der in den angegebenen Arbeiten verwendete Konsistenzbegriff ist dadurch charakterisiert, daß eine Menge von Rechten als konsistent bezeichnet wird, wenn sie keine Konflikte enthält, d.h. nicht gleichzeitig eine Erlaubnis und ein Verbot bzgl. eines Zugriffs auf ein Objekt für ein Subjekt in der Rechtemenge enthalten ist. Enthält die Rechtemenge Konflikte, wird sie als inkonsistent bezeichnet. Dieser Konsistenzbegriff bezieht sich lediglich auf objektlokale Eigenschaften, da für jedes Objekt die Menge der an dem Objekt vergebenen Rechte isoliert von den Rechten, die für andere Objekte vergeben sind, betrachtet wird.

Konflikte und damit inkonsistent vergebene Rechte können sich in den entsprechenden Systemen insbesondere dadurch ergeben, daß zum einen ein Benutzer Mitglied mehrerer Gruppen bzw. Rollen sein kann, die ebenfalls Subjekte sind, und zum anderen an einen Benutzer die Rechte der Gruppen bzw. der Rollen vergeben sind, in denen er Mitglied ist. Ein Konflikt liegt also z.B. dann vor, wenn ein Benutzer Mitglied zweier Gruppen G_1 und G_2 ist und an G_1 ein positives Recht bzgl. eines Zugriffs auf ein Objekt und an G_2 ein negatives Recht für den entsprechenden Zugriff auf das Objekt vergeben ist. Die Problematik der durch mehrere Gruppenmitgliedschaften entstehenden Konflikte verschärft sich noch, wenn – wie in [BJS96] – Gruppenhierarchien möglich sind, d.h. eine Gruppe selbst wiederum Mitglied mehrerer anderer Gruppen sein kann. Analoges gilt für sogenannte Rollenhierarchien (siehe z.B. [JD95]).

In den meisten zugriffskontroll–basierten Systemen, in denen sowohl positive als auch negative Rechte vergeben werden können, wird das Auftreten von Konflikten und damit die inkonsistente Vergabe von Rechten zugelassen. Vorliegende Konflikte werden dann im allgemeinen durch eine festgelegte Regel aufgelöst. Die in der überwiegenden Anzahl derartiger Systeme angewendete Regel zur Konfliktauflösung lautet, daß ein Verbot Vorrang vor einer Erlaubnis hat, d.h. daß einem Subjekt, an das gleichzeitig ein positives und ein negatives Recht bzgl. eines Zugriffs auf ein Objekt vergeben ist, der entsprechende Zugriff auf das Objekt verweigert wird (siehe z.B. [JD95], [Ooi93], [Kel90], [Sat89]). Eine andere Möglichkeit der Konfliktauflösung besteht darin, dem spezifischerem Recht Vorrang vor einem allgemeinerem Recht einzuräumen, wobei z.B. ein direkt an einen Benutzer vergebenes Recht spezifischer als ein an eine Gruppe vergebenes Recht sein kann. Eine solche Konfliktauflösungsregel wird z.B. in [Lun89] angewandt. [Bru93] und [Lau95] äußern Kritik an den beiden erwähnten impliziten Regeln zur Auflösung von Konflikten und fordern, daß durch Zusatzinformationen explizit festzulegen ist, wie Konflikte zu lösen sind. Sie schlagen deshalb vor, Rechte mit Prioritäten zu versehen. Im Falle eines Konflikts zwischen zwei Rechten hat dann das Recht mit der höchsten Priorität Vorrang. Um auf diese Weise alle Konflikte lösen zu können, wird in [Lau95] die Konsistenzbedingung aufgestellt, daß die Prioritäten aller Rechte, die den gleichen Zugriff betreffen, eindeutig sein müssen, d.h. nicht mehrere dieser Rechte die gleiche Priorität haben dürfen. In [Bru93] wird eine derartige Bedingung nicht aufgestellt. Dort ergeben sich Konflikte zwischen Erlaubnissen und Verboten, wenn diese sich auf den gleichen Zugriff beziehen und die gleiche Priorität haben. Es wird allerdings erwähnt, daß solche Konflikte erkannt werden müssen und die Rechteadministratoren durch entsprechende Fehlermeldungen über derartige Konflikte zu informieren sind.

In [BJS96] wird zwischen starken und schwachen Rechten unterschieden, wobei starke Rechte stets Vorrang vor schwachen Rechten haben. Dieser Ansatz entspricht damit dem oben erklärten Ansatz zur expliziten Auflösung von Konflikten durch Vergabe von Prioritäten, wobei hier allerdings nur zwei Prioritäten (*stark* und *schwach*) möglich sind. Der in [BJS96] definierte Konsistenzbegriff bezieht sich lediglich auf Konflikte zwischen starken Rechten. Der Rechtszustand eines der dort betrachteten Datenbank–Systeme wird als konsistent bezeichnet, wenn er keine Konflikte zwischen starken Rechten enthält. Konflikte zwischen schwachen Rechten werden zugelassen und nach der Regel "Verbote vor Erlaubnisse" aufgelöst. Interessant an diesem Ansatz ist, daß das Auftreten von Konflikten zwischen starken Rechten explizit ausgeschlossen wird und damit die Konsistenz des Rechtszustands eine Invariante der betrachteten Systeme ist. Es wird also gewährleistet, daß nicht gleichzeitig ein positives und ein negatives starkes Recht bzgl. desselben Objekts an ein Subjekt vergeben ist. Um dies zu erreichen, werden Operationen, mit denen der Rechtszustand der betrachteten Systeme verändert werden kann (hierzu gehören z.B. Operationen zur Vergabe und Rücknahme von Rechten) nur dann ausgeführt, wenn der jeweils resultierende Rechtszustand konsistent ist. Dazu werden Algorithmen angegeben, mit denen überprüft werden kann, ob durch die Ausführung einer den Rechtszustand modifizierenden Operation ein inkonsistenter Rechtszustand entstehen würde.

In dem Ansatz von [WL93] werden Rechte auf der Basis sogenannter verallgemeinerter Zugriffskontrollisten (*gacl – generalized access control list*), deren Ausdrucksmöglichkeiten weit über die der üblichen Zugriffskontrollisten hinausgehen, spezifiziert. So können z.B. neben Erlaubnissen und Verboten Rechte in Abhängigkeit von den Werten bestimmter Systemprädikate oder abhängig von dem Vorhandensein anderer Rechte vergeben werden. Verallgemeinerte Zugriffskontrollisten für die Objekte eines Systems werden mit Hilfe der Sprache GACL spezifiziert. Dabei kann für jede *gacl* angegeben werden, ob die Reihenfolge ihrer Einträge bei der Auswertung berücksichtigt werden soll (Attribut: *ordered*) oder nicht (Attribut: *unordered*).

Für die Sprache GACL wird eine Semantik angegeben, die zu einer *gacl* die Menge der durch diese *gacl* vergebenen Rechte liefert. Für eine *ordered gacl* ist die Semantik so festgelegt, daß in der durch diese *gacl* spezifizierten Rechtemenge keine Konflikte enthalten sind. Mögliche Konflikte werden dabei auf Basis der Reihenfolge der Einträge der *gacl* aufgelöst, d.h. ein Eintrag in der *gacl* hat jeweils Vorrang vor seinen nachfolgenden Einträgen. Dem entspricht, daß eine *ordered gacl* eines Objekts bei einem Zugriff auf das Objekt in der Reihenfolge ihrer Einträge ausgewertet wird, bis der erste "passende" Eintrag gefunden ist. Beim Zugriff auf ein Objekt, für das eine *unordered gacl* spezifiziert ist, werden stattdessen alle Einträge der *gacl* berücksichtigt, wobei Konflikte zwischen verschiedenen Einträgen vorliegen können. Derartige Konflikte werden jedoch nicht durch eine implizite Regel aufgelöst, sondern es wird eine Fehlermeldung ausgegeben, die besagt, daß die durch die *gacl* spezifizierte Rechtemenge inkonsistent ist. Als Alternative zu diesem Vorgehen wird vorgeschlagen, durch sogenannte Präferenz–Deklarationen in *unordered gacl*'s explizit anzugeben, wie Konflikte aufzulösen sind. Als Beispiel für derartige Präferenz–Deklarationen werden Operations–Präferenzen (z.B. Verbot vor Erlaubnis) und Gruppen–Präferenzen (z.B. Gruppe G_1 vor Gruppe G_2) genannt.

5.1.2 Konsistenz bezogen auf Schachtelungsabhängigkeiten zwischen Objekten

Der bisher erklärte Konsistenzbegriff bezieht sich lediglich auf das Vorhandensein positiver und negativer Rechte bezogen auf *ein* Objekt. Abhängigkeiten zwischen einzelnen Objekten oder Operationen werden in diesem Konsistenzbegriff nicht berücksichtigt. In [Kel90] wird ein Konsistenzbegriff definiert, der Abhängigkeiten zwischen den dort betrachteten komplexen Objekten strukturierter objekt–orientierter Datenbank–Systeme, die Basis sogenannter Objektmanagement–Systeme sind, berücksichtigt. Die Abhängigkeiten zwischen den Objekten ergeben sich zum einen daraus, daß komplexe Objekte ineinander geschachtelt sind, was bedeutet, daß komplexe Objekte aus Komponenten zusammengesetzt sind, die selbst wiederum komplexe Objekte sind, und zum anderen daraus, daß sich komplexe Objekte überlappen können, d.h. ein Objekt Komponente mehrerer komplexer Objekte sein kann. Ein derartiges Objekt wird in [Kel90] als *shared object* bezeichnet. Da es sich bei den in [Kel90] betrachteten Systemen um Objektmanagement–Systeme handelt, erfolgt die Schachtelung von Objekten nicht vorrangig unter funktionalen Gesichtspunkten, sondern danach, die Objektmenge geeignet zu strukturieren. Für komplexe Objekte ist eine feste Menge von einfachen Zugriffsmodi (z.B. *lesen*, *löschen*) festgelegt, die die Einheiten für die Vergabe von Rechten zur Nutzung von komplexen Objekten sind. Es können sowohl positive als auch negative Rechte vergeben werden, wobei ein Recht an einem komplexen Objekt das entsprechende Recht an allen Komponenten des Objekts impliziert. Ein positives Leserecht an einem komplexen Objekt impliziert also z.B. ein positives Leserecht an allen Komponenten des Objekts. Durch diese Festlegung können sich leicht Widersprüche in den Rechtsfestlegungen für *shared objects* ergeben. Ein solcher Widerspruch in den Rechten für ein *shared object* liegt dann vor, wenn ein Subjekt an einem der komplexen Objekte, zu denen das *shared object* gehört, ein positives Recht und an einem anderen dieser Objekte ein negatives Recht bzgl. des gleichen Zugriffsmodus hat. Um derartige widersprüchliche Festlegungen auszuschließen, wird in [Kel90] eine Konsistenzregel aufgestellt, die besagt, daß ein positives bzw. negatives Recht an einem komplexen Objekt stets ein positives bzw. negatives Recht an allen seinen Komponenten implizieren muß. Zur Durchsetzung dieser Konsistenzregel wird eine Rechteänderung an einem komplexen Objekt an alle Komponenten des Objekts propagiert. Wird jedoch durch diese Rechtepropagierung, von der insbesondere auch *shared objects* betroffen sind, die Konsistenzregel verletzt, wird die Rechteänderung zurückgewiesen. Betrachtet man z.B. ein komplexes Objekt o_1, an dem ein

Subjekt s für den Zugriffsmodus m ein negatives Recht hat, und eine Komponente k von o_1, die gleichzeitig Komponente eines komplexes Objekts o_2 ist (k also *shared object* ist), so wird jeder Versuch, an das Subjekt s ein positives Recht für den Zugriffsmodus m des Objekts o_2 zu vergeben, zurückgewiesen. Auf diese Weise wird stets ein konsistenter Rechtszustand bewahrt. Es ist noch anzumerken, daß in den betrachteten Systemen ebenfalls Widersprüche dadurch entstehen können, daß ein Subjekt Mitglied mehrerer Gruppen sein kann und für diese Gruppen unterschiedliche Rechte an einem komplexen Objekt bzgl. des gleichen Zugriffsmodus vergeben werden können. Derartige Konflikte werden in [Kel90] zugelassen und durch die bereits bekannte Regel, daß Verbote Vorrang vor Erlaubnissen haben, aufgelöst.

5.1.3 Konsistenz bezogen auf funktionale Abhängigkeiten zwischen Operationen

In [Spi85] und [Eck93] werden Systeme betrachtet, die aus Objekten bestehen, deren Operationen explizit und damit anwendungsspezifisch definiert werden können. Für diese Systeme wird ein Konsistenzbegriff eingeführt, der funktionale Abhängigkeiten zwischen den Operationen der Objekte berücksichtigt. Eine funktionale Abhängigkeit zwischen einer Operation op_1 eines Objekts o_1 und einer Operation op_2 eines Objekts o_2 besteht dann, wenn op_2 im Kontext der Ausführung von op_1 direkt oder indirekt aufgerufen wird. Die Funktionalität von op_2 wird also zur Erbringung der Funktionalität von op_1 benötigt. Eine Menge von Rechten wird als konsistent bezeichnet, wenn für alle Subjekte s, die das Recht zur Ausführung einer Operation op eines Objekts o haben, gilt, daß diese Subjekte im Kontext einer op–Ausführung auch das Recht zur Ausführung aller Operationen haben, von denen op funktional abhängig ist. Das Recht zur Ausführung von op muß also das Recht zur Ausführung aller Operationen, die im Kontext von op auszuführen sind, implizieren. Die in diesem Sinne konsistente Vergabe der Rechte in einem System ist Voraussetzung dafür, daß die mit der Wahrnehmung eines Rechts verbundene Leistungsanforderung, die besagt, daß die Funktionalität der entsprechenden Operation vollständig zu erbringen ist, erfüllt werden kann (siehe [Spi85]).

Dieser Konsistenzbegriff ist sehr weitreichend, und es ist im allgemeinen nicht einfach nachzuweisen bzw. zu erreichen, daß die Rechte in einem System in dem erklärten Sinne konsistent vergeben sind bzw. vergeben werden. Dies liegt zum einen daran, daß die funktionalen Abhängigkeiten zwischen den Operationen im allgemeinen komplex sein können, und zum anderen in der Regel nur potentielle Abhängigkeiten ermittelbar sind. Der in [Spi85] formulierte Konsistenzbegriff bezieht sich auf Rechtemengen, die durch eine Zugriffsmatrix festlegbar sind, wobei lediglich positive Rechte spezifiziert werden können. Es wird zwar die Forderung aufgestellt, daß die Rechte im obigen Sinn konsistent zu vergeben sind. Auf Maßnahmen, die die konsistente Vergabe von Rechten unterstützen bzw. gewährleisten, wird jedoch nicht eingegangen. In [Eck93] wird der Konsistenzbegriff für die mit dem dort eingeführten formalen Instrumentarium spezifizierbaren Zugriffskontrollpolitiken definiert.[1] Es werden Bedingungen für Zugriffskontrollpolitiken formuliert, deren Gültigkeit die Konsistenz der Politik gewährleistet. Der Nachweis, daß diese Bedingungen für eine Politik erfüllt sind, läßt sich jedoch im allgemeinen nicht statisch führen. Eine Einschränkung der in [Eck93] betrachteten Systeme besteht darin, daß diese statisch in dem Sinne sind, daß keine neuen Objekte und Subjekte erzeugt bzw. aufgelöst werden können. Damit sind z.B. Situationen, in denen bei Ausführung einer Operation op ein neues Objekt erzeugt wird, auf das anschließend

[1] Der Konsistenzbegriff für Zugriffskontrollpolitiken wird in [Eck93] später zu einem Konsistenzbegriff für allgemeine Sicherheitspolitiken, die neben Zugriffskontrollanforderungen auch Informationsflußanforderungen enthalten können, erweitert. Da die vorliegende Arbeit sich jedoch lediglich mit Zugriffskontrollpolitiken beschäftigt, wird darauf nicht weiter eingegangen.

im Kontext der *op*–Ausführung zugegriffen werden soll, nicht modellierbar. Zur Durchsetzung konsistent festgelegter Zugriffskontrollpolitiken wird in [Eck93] ein auf Capabilities basierendes Konzept zur dezentralen Zugriffskontrolle angegeben, das insbesondere die Konsistenz der vergebenen Rechte bei Rechteänderungen gewährleistet. Die Vergabe eines Rechts für eine Operation op eines Objekts o an ein Subjekt s ist lediglich dann zulässig, wenn das Subjekt s bereits Rechte an allen Operationen op' hat, von denen op funktional abhängig ist. Die Rücknahme eines Rechts, das ein Subjekt s für die Operation op eines Objekts o besitzt, führt dazu, daß diese Rechterücknahme bzgl. des Subjekts s auch für alle Operationen op', die von op funktional abhängig sind, wirksam wird. Durch diese Maßnahmen wird die Konsistenz des Rechtszustands bewahrt. Problematisch dabei ist jedoch zum einen, daß die bestimmbaren funktionalen Abhängigkeiten – wie bereits oben erwähnt – im allgemeinen nur potentielle Abhängigkeiten sind und somit für die Zulässigkeit einer Rechtevergabe ggf. "zuviele" Rechte bereits vorhanden sein müssen bzw. bei einer Rechterücknahme "zuviele" Rechte zurückgenommen werden. Um die Konsistenz des Rechtszustands zu gewährleisten, wird also ein *worst case* Ansatz verfolgt. Zum anderen können die genannten Maßnahmen zur Konsistenzerhaltung nur dann erfolgreich eingesetzt werden, wenn alle funktionalen Abhängigkeiten statisch sind und damit von vornherein bekannt sind. Für Systeme, in denen Objekte dynamisch erzeugt werden können, die über Zeiger, deren Werte sich dynamisch ändern können, genutzt werden können, sind diese Maßnahmen somit wenig geeignet.

5.1.4 Fazit

Als Fazit des Literaturüberblicks läßt sich festhalten, daß es in Systemen, in denen sowohl positive als auch negative Rechte an Subjekte vergeben werden können, Regeln geben muß, durch die möglicherweise vorliegende Konflikte aufgelöst werden können. Der damit gegebene objektlokale Konsistenzbegriff ist jedoch nicht geeignet, Ausführungsabbrüche begonnener Operationsausführungen aufgrund fehlender Rechte zu vermeiden und damit die Voraussetzung für die mit der Wahrnehmung eines Rechts verbundene Leistungsanforderung zu schaffen. Um diese Leistungsanforderung erfüllen zu können, sind funktionale Abhängigkeiten zwischen Operationen zu berücksichtigen.

Der im folgenden Abschnitt definierte Konsistenzbegriff für INSEL$^+$–Systeme orientiert sich deshalb an den in [Spi85] und [Eck93] eingeführten Konsistenzbegriffen. Besonderheiten ergeben sich daraus, daß in INSEL$^+$–Systemen komplexe funktionale Abhängigkeiten zwischen Komponenten bestehen können und Komponenten dynamisch erzeugt und aufgelöst werden können. Im weiteren werden dann Kriterien angegeben, die es ermöglichen, anhand statischer Analysen Aussagen über die Konsistenz des Rechts eines INSEL$^+$–Systems zu gewinnen. Aus diesen Kriterien werden schließlich Richtlinien abgeleitet, die bei der Konstruktion eines INSEL$^+$–Systems im Hinblick auf die Durchsetzung der Forderung nach Konsistenz des Rechts des Systems zu beachten sind. Mit den Analyseverfahren ist es erstmals möglich, die Konsistenz des Rechts eines Systems (im Sinne des sehr weitreichenden Konsistenzbegriffs nach [Spi85] und [Eck93]) statisch nachzuweisen. Zusammen mit den Richtlinien steht einem Systementwickler damit ein Instrumentarium zur Verfügung, das ihn bei der Konstruktion von Systemen mit konsistent vergebenen Rechten unterstützt.

Abschließend sei darauf hingewiesen, daß in INSEL$^+$–Systemen mögliche Konflikte, die sich durch positive und negative Einträge in einer Benutzer–Rollen–Liste der Zugriffskontrolliste einer zugriffskontrollierten Komponente ergeben könnten, gemäß der Semantik des `IN_ACL`–Prädikats (siehe Abschnitt 4.4.4.1) dadurch aufgelöst werden, daß negative Einträge immer durchschlagen. Dies entspricht der Regel, daß ein Verbot Vorrang vor einer Erlaubnis hat.

5.2 Konsistenz des Rechts eines INSEL$^+$–Systems

In diesem Abschnitt wird zunächst der Begriff des Rechts eines INSEL$^+$–Systems eingeführt. Anschließend wird definiert, wann das Recht eines INSEL$^+$–Systems konsistent ist, und es werden Maßnahmen aufgezeigt, die dazu beitragen können, die Konsistenz des Rechts eines INSEL$^+$–Systems zu gewährleisten.

5.2.1 Recht eines INSEL$^+$–Systems

Die Definition des Rechts eines INSEL$^+$–Systems orientiert sich an der in [Spi85] angegebenen Definition für das Recht eines Systems, das aus einer Menge von Objekten, auf denen Operationen definiert sind, und aus einer Menge von Subjekten, die Operationen auf den Objekten ausführen, besteht. In einem INSEL$^+$–System sind die Akteure die Subjekte und alle Komponenten des Systems die Objekte (vgl. Abschnitt 4.1).

Die äußeren Operationen der Komponenten entsprechen den auf den Objekten definierten Operationen. Die Komponenten eines INSEL$^+$–Systems lassen sich nach dem in Kapitel 4 Gesagten in zugriffskontrollierte Komponenten und nicht zugriffskontrollierte Komponenten unterscheiden. Die Rechte zur Nutzung nicht zugriffskontrollierter Komponenten sind damit implizit vergeben: ein Akteur hat das Recht zur Ausführung der äußeren Operationen einer nicht zugriffskontrollierten Komponente, wenn diese Komponente in der Ausführungsumgebung des Akteurs liegt. Die zugriffskontrollierten Komponenten sind die Komponenten, für deren Nutzung Rechte explizit auf Basis der für ihre Views festgelegten Zugriffsrestriktionsausdrücke vergeben sind. Gemäß Abschnitt 4.4.4 hat ein Akteur a genau dann das Recht zur Ausführung einer äußeren Operation op einer zugriffskontrollierten Komponente x, die nicht die Startoperation von x ist, wenn mindestens einer der Zugriffsrestriktionsausdrücke der Views von x, zu denen die Operation op gehört, für den Akteur a den Wert *true* hat. Der Akteur a hat also das Recht zur Ausführung von op auf x, wenn die Disjunktion aller Zugriffsrestriktionsausdrücke der Views, zu denen op gehört, *true* ist. Dementsprechend wird, vorbereitend für die Definition des Rechts eines INSEL$^+$–Systems, jeder äußeren Operation op von x, für die Rechte explizit vergeben werden können, ein Zugriffsrestriktionsausdruck zugeordnet, der aus der Disjunktion der Zugriffsrestriktionsausdrücke der Views besteht, die op als Element enthalten. Zunächst werden jedoch noch einige Bezeichnungen und Notationen vereinbart, die im weiteren neben den in (3.4) und (4.7) eingeführten Bezeichnungen verwendet werden.

(5.1) Bezeichnungen und Notationen

Sei $\mathcal{S}$ ein INSEL$^+$–System zum Zeitpunkt $t \in \Lambda(\mathcal{S})$. Dann bezeichnet:

- X_t^{ZK} die Menge der zugriffskontrollierten Komponenten von $\mathcal{S}$ zum Zeitpunkt t,
- $X_t^{EZK} \subseteq X_t^{ZK}$ die Menge der explizit zugriffskontrollierten Komponenten von $\mathcal{S}$ zum Zeitpunkt t,
- $O(x)$ für $x \in X_t$ die Menge der äußeren Operationen von x,
- $O^E(x) \subseteq O(x)$ für $x \in X_t^{ZK}$ die Menge der äußeren Operationen von x, für die Rechte explizit vergeben werden können,
- $i(x) \in O(x)$ für $x \in X_t^{DAI}$ die implizit auf x definierte Startoperation (*führe_aus* bzw. *starte* bzw. *initialisiere*).

Für die logischen Operatoren `AND`, `OR` und `NOT`, die in Zugriffsrestriktionsausdrücken auftreten können, gelten im weiteren die folgenden Notationen und Bezeichnungen:

Operator	Notation	Bezeichnung
`AND`	$\wedge$	Konjunktion
`OR`	$\vee$	Disjunktion
`NOT`	$\neg$	Negation

Ferner werden folgende abkürzende Schreibweisen eingeführt:

$$\bigwedge_{i \in \{1,\ldots,n\}} t_i \triangleq t_1 \wedge t_2 \wedge \ldots \wedge t_n$$

$$\bigvee_{i \in \{1,\ldots,n\}} t_i \triangleq t_1 \vee t_2 \vee \ldots \vee t_n$$

□

Definition 5.2.: Zugriffsrestriktionsausdruck einer äußeren Operation

Seien $x \in X_t^{ZK}$ und $op \in O^E(x)$. Dann ist der **Zugriffsrestriktionsausdruck** R_{op}^x **der äußeren Operation** op **von** x wie folgt festgelegt:

$$R_{op}^x \triangleq \bigvee_{\substack{v \,\in\, Views(x)\,: \\ op \,\in\, v}} R_v^x$$

□

Ist eine äußere Operation $op \in O^E(x)$ lediglich Element eines Views[2], so entspricht der Zugriffsrestriktionsausdruck dieser Operation dem für den entsprechenden View festgelegten Zugriffsrestriktionsausdruck. Anderenfalls besteht der Zugriffsrestriktionsausdruck von op aus der Disjunktion der Zugriffsrestriktionsausdrücke der Views, zu denen op gehört. Auf Basis von Definition (5.2) kann nun das Recht eines INSEL$^+$–Systems definiert werden.

Definition 5.3.: Recht eines INSEL$^+$–Systems

Sei $\mathcal{S}$ ein INSEL$^+$–System zum Zeitpunkt $t \in \Lambda(\mathcal{S})$. Das **Recht** $R_t(\mathcal{S})$ **des Systems** $\mathcal{S}$ **zum Zeitpunkt** t ist definiert durch:

$$R_t(\mathcal{S}) = (A_t, X_t, (\rho_t(\cdot\,, x) \mid x \in X_t))$$

mit

$$\rho_t(\cdot\,, x) : A_t \longrightarrow POT(O(x)) \quad \text{für} \quad x \in X_t$$

[2]Zur Erinnerung: Dies gilt insbesondere für alle äußeren Operationen eines zugriffskontrollierten DA–Generators (siehe Definition (4.5)).

und

$$\rho_t(a,x) \triangleq \begin{cases} O(x) & \text{falls } x \in X_t^{DEG} \cup X_t^{DEI} \wedge \\ & x \in U_t(\varphi_t(a)) \\ O(x) & \text{falls } x \in X_t^{DAG} \setminus X_t^{ZK} \wedge \\ & x \in U_t(\varphi_t(a)) \\ O(x) \setminus i(x) & \text{falls } x \in X_t^{DAI} \setminus X_t^{ZK} \wedge \\ & x \in U_t(\varphi_t(a)) \\ \{op \in O^E(x) \mid \omega(R_{op}^x, a, t) = true\} & \text{falls } x \in X_t^{ZK} \wedge x \in U_t(\varphi_t(a)) \\ i(x) & \text{falls } x \in X_t^{DAI} \setminus KO_t \wedge \mu_t(x) = V \wedge \\ & a = creator(x) \\ i(x) & \text{falls } x \in KO_t \wedge \mu_t(x) = V \wedge \\ & gen(x) = L_0(a) \\ \emptyset & \text{sonst} \end{cases}$$

Das **Recht** $R(\mathcal{S})$ **des Systems** $\mathcal{S}$ ist gegeben durch:

$$(R_t(\mathcal{S}) \mid t \in \Lambda(\mathcal{S}))$$

$\omega(R_{op}^x, a, t)$ bezeichnet gemäß (4.7) den Wert von R_{op}^x zum Zeitpunkt t für den Akteur a. $gen(x)$ bezeichnet laut (3.4) den Generator bzgl. dem x inkarniert wurde und $\mu_t(x)$ den Zustand von x zum Zeitpunkt t. *creator(x)* gibt den Akteur an, der x bei Ausführung seiner kanonischen Operation erzeugt hat (siehe Definition (3.5)), und $U_t(\varphi_t(a))$ ist gemäß Definition (4.4) die Ausführungsumgebung der Ausführungskomponente des Akteurs a zum Zeitpunkt t.

□

Sei im weiteren $R_t(\mathcal{S})$ gemäß Definition (5.3) das Recht eines INSEL$^+$-Systems $\mathcal{S}$ zum Zeitpunkt t. Ein Akteur $a \in A_t$ hat zum Zeitpunkt t genau dann das Recht zur Ausführung einer äußeren Operation *op* einer Komponente $x \in X_t$, wenn gilt: $op \in \rho_t(a,x)$. Mit Definition (5.3) ist die Menge $\rho_t(a,x)$ formal festgelegt.

Ist x keine zugriffskontrollierte Komponente, so hat a das Recht zur Ausführung aller äußeren Operationen von x – mit Ausnahme der Startoperation von x, falls x DA-Inkarnation ist – wenn x in der Ausführungsumgebung der Ausführungskomponente $\varphi_t(a)$ von a liegt. Dies entspricht der Tatsache, daß für die Nutzung nicht zugriffskontrollierter Komponenten keine Rechte explizit vergeben werden können. Ist x zugriffskontrolliert und ist in dem Fall, daß x DA–Inkarnation ist, *op* nicht die implizit auf x definierte Startoperation, dann hat a zum Zeitpunkt t das Recht zur Ausführung von *op*, wenn x Element der Ausführungsumgebung der Ausführungskomponente $\varphi_t(a)$ von a ist und der gemäß (5.2) für *op* festgelegte Zugriffsrestriktionsausdruck R_{op}^x in t für a erfüllt ist.

Das Recht zur Ausführung der auf einer DA-Inkarnation x implizit definierten Startoperation $i(x)$ ist gesondert zu betrachten. $i(x)$ wird unmittelbar nach der Erzeugung von x von dem erzeugenden Akteur implizit[3] aufgerufen. Eine Ausnahme bilden lediglich die K-Order. Die Startoperation einer K-Order wird nicht unmittelbar nach Erzeugung der K-Order von dem

[3] Das bedeutet, daß ein Aufruf dieser Operation in einem INSEL$^+$–Programm nicht möglich ist.

erzeugenden Akteur aufgerufen, sondern erst bei Annahme der K-Order durch den annehmenden K-Akteur. Dementsprechend ist das Recht an der Startoperation $i(x)$ vergeben. Das Recht zur Ausführung von $i(x)$ hat der x erzeugende Akteur, falls x keine K-Order ist. Ist x K-Order, so hat der K-Akteur, der den entsprechenden K-Order–Generator als lokale N-Komponente enthält, der also die K-Order x annehmen kann, das Recht zur Ausführung von $i(x)$. Die Ausführung von $i(x)$ auf einer DA–Inkarnation x bewirkt den Start der Ausführung der kanonischen Operation von x und damit für x den Übergang vom Zustand V in den Zustand A bzw. R. Das Recht an $i(x)$ ist also so vergeben, daß ein Akteur, der das Recht zur Erzeugung von Inkarnationen bzgl. eines DA–Generators hat, auch das Recht zum Starten der Ausführung der kanonischen Operationen aller Inkarnationen hat, die durch Wahrnehmung dieses Rechts von ihm erzeugt wurden. Dies gilt jedoch nur für DA–Generatoren, die nicht K-Order–Generatoren sind. Das Recht zur Ausführung der Startoperation von K-Order wird so vergeben, daß ein K-Akteur das Recht zur Ausführung der kanonischen Operationen der K-Order hat, die Inkarnationen bzgl. seiner K-Order–Generatoren sind.

5.2.2 Konsistenzbegriff für INSEL$^+$–Systeme

Hat ein Akteur a zu einem Zeitpunkt t das Recht zur Ausführung der äußeren Operation einer Komponente, so ist damit bei Wahrnehmung dieses Rechts durch a eine Leistungsanforderung verbunden, die besagt, daß die Funktionalität der entsprechenden Operation vollständig zu erbringen ist. Dies stellt einerseits hohe Anforderungen an die Zuverlässigkeit und Fehlertoleranz des Systems, die hier jedoch nicht weiter betrachtet werden sollen. Andererseits wird damit jedoch auch eine Anforderung an das Recht des Systems gestellt, die sich daraus ergibt, daß im allgemeinen bei Ausführung einer äußeren Operation einer Komponente zur Erbringung der Funktionalität der Operation äußere Operationen anderer Komponenten des Systems ausgeführt werden. Die gestellte Anforderung ist die Forderung nach Konsistenz des Rechts des Systems, die im folgenden definiert wird.

Definition 5.4.: Konsistenz des Rechts eines INSEL$^+$–Systems

Sei $\mathcal{S}$ ein INSEL$^+$–System. Das **Recht** $R(\mathcal{S})$ **von** $\mathcal{S}$, das durch $(R_t(\mathcal{S}) \mid t \in \Lambda(\mathcal{S}))$ gegeben ist, ist **konsistent** genau dann, wenn für alle Zeitpunkte $t_1 \in \Lambda(\mathcal{S})$ und alle Akteure $a \in A_{t_1}$ gilt:

> Wenn a zu dem Zeitpunkt t_1 das Recht zur Ausführung einer äußeren Operation *op* einer Komponente $x \in X_{t_1}$ hat, d.h. $op \in \rho_{t_1}(a, x)$ gilt, und a dieses Recht in t_1 wahrnimmt, d.h. die Operation *op* auf x in t aufruft, so gilt: wenn im Kontext der Ausführung von *op* zu einem Zeitpunkt $t_2 > t_1$ eine äußere Operation *op*$'$ einer Komponente $y \in X_{t_2}$ aufgerufen wird, hat der die Operation *op*$'$ aufrufende Akteur $b \in A_{t_2}$ das Recht zur Ausführung von *op*$'$, d.h. es gilt: $op' \in \rho_{t_2}(b, y)$.

Anderenfalls ist das **Recht** $R(\mathcal{S})$ **von** $\mathcal{S}$ **inkonsistent**.

□

Die beiden in Definition (5.4) genannten Akteure a und b sind im allgemeinen nicht identisch. Entspricht b nicht dem Akteur a, so ist b entweder

- der K-Akteur, auf dem die Operation *op* aufgerufen wird, falls *op* Kommunikationsoperation ist, oder

- ein K-Akteur, auf dem im Kontext der Ausführung von *op* eine Kommunikationsoperation aufgerufen wird, oder
- ein Akteur, der im Kontext der Ausführung von *op* erzeugt wurde.

Gemäß Definition (5.4) liegt eine Inkonsistenz in dem Recht eines INSEL$^+$–Systems vor, wenn ein Akteur a ein vorhandenes Recht an einer Operation op einer Komponente x wahrnimmt und bei Ausführung von op die Situation eintritt, daß das Recht für eine Operation op', die im Kontext von op auszuführen ist, für den op' aufrufenden Akteur b nicht vorhanden ist. Da die Operation op' einen Beitrag zur Funktionalität der Operation op leistet, das Recht zur Ausführung von op' für den op' aufrufenden Akteur b jedoch nicht vorhanden ist, bleibt in dieser Situation, sofern keine expliziten Maßnahmen zur Behandlung dieses Zugriffsverbotes vorgesehen sind[4], kein anderer Ausweg, als die Ausführung von op und damit aller Operationen, deren Ausführung durch op angestoßen wurde und die noch nicht terminiert sind, abzubrechen. Mit dem Abbruch von op wird die Funktionalität von op nicht vollständig erbracht, was bedeutet, daß die gestellte Leistungsanforderung nicht erfüllt wird. Das Auftreten durch Inkonsistenzen in dem Recht eines INSEL$^+$–Systems verursachter Ausführungsabbrüche ist also nach Möglichkeit zu vermeiden, da anderenfalls – bei Fehlen entsprechender Behandlungsmaßnahmen – die Funktionalität des Gesamtsystems in Frage gestellt wird. Die Konsistenz des Rechts eines INSEL$^+$–Systems gewährleistet, daß es in dem System keine Ausführungsabbrüche begonnener Operationsausführungen aufgrund fehlender Rechte gibt; sie ist damit eine wesentliche Voraussetzung für die Erfüllung der mit dem Recht an einer Komponente verbundenen Leistungsanforderung. Bevor näher auf die Problematik des Nachweises der Konsistenz des Rechts eines INSEL$^+$–Systems eingegangen wird und Maßnahmen beschrieben werden, die dazu beitragen können, die Konsistenz des Rechts zu gewährleisten, wird zunächst erklärt, was mit dem in Definition (5.4) auftretenen Begriff "im Kontext der Ausführung von *op*" gemeint ist. Dies ist für das vollständige Verständnis der Tragweite des in (5.4) definierten Konsistenzbegriffs notwendig.

Zu der Menge der äußeren Operationen, die im Kontext der Ausführung einer äußeren Operation op einer Komponente x aufgerufen werden, gehören alle Operationen, die bei Ausführung von op direkt oder indirekt aufgerufen werden. Die Menge dieser Operationen ist zum einen von der Art der Komponente x und zum anderen von der Art der Operation op abhängig. x kann entweder Generator, DE–Inkarnation oder DA–Inkarnation sein, und op kann entweder eine für x implizit definierte äußere Operation oder – falls x K-Akteur bzw. Depot ist – eine der für x explizit definierten äußeren Operationen sein. Im folgenden wird anhand einer Fallunterscheidung informell die Menge der äußeren Operationen angegeben, die im Kontext der Ausführung einer äußeren Operation op einer Komponente x aufgerufen werden. Diese Menge wird mit $CalledOps(x, op)$ bezeichnet. Es sei darauf hingewiesen, daß sich die Menge $CalledOps(x, op)$ im allgemeinen nicht statisch ermitteln läßt, da diese Menge in der Regel von der tatsächlichen Berechnung der op–Ausführung abhängig ist. Die statisch bestimmbare Menge der Operationen, die *potentiell* im Kontext der Ausführung einer Operation op aufgerufen werden, wird zu einem späteren Zeitpunkt (siehe Abschnitt 5.3.2, Definition (5.18)) betrachtet werden.

Zur Angabe der Menge $CalledOps(x, op)$ sei a ein Akteur mit der Ausführungskomponente $\varphi_t(a) = y$, der bei Ausführung des sequentiellen Kerns der kanonischen Operation $op(y)$ die äußere Operation op auf x aufruft.

[4] Zur expliziten Behandlung von Zugriffsverboten steht das in Abschnitt 4.5 beschriebene INSEL$^+$–Konzept zur Verfügung.

1. x ist DE–Generator
   Auf x ist als einzige äußere Operation die Operation *erzeuge* implizit definiert, d.h. es gilt: $op = erzeuge$. Die Ausführung von *erzeuge* auf x bewirkt die Erzeugung einer DE–Inkarnation bzgl. der durch den Generator definierten Klasse. Weitere Operationen werden bei Ausführung von *erzeuge* nicht aufgerufen. Für die Menge $CalledOps(x, erzeuge)$ ergibt sich somit:
   $$CalledOps(x, erzeuge) = \emptyset$$

2. x ist DE–Inkarnation
   Auf DE–Inkarnationen ist implizit eine Lese– und eine Schreiboperation definiert. op ist also entweder Lese– oder Schreiboperation.

   2.1 Ist x skalare wertorientierte Komponente, also ein Datum[5] oder ein Zeiger, werden bei Ausführung der Lese– bzw. Schreiboperation auf x keine weiteren Operationen aufgerufen, d.h. es gilt:
   $$CalledOps(x, op) = \emptyset$$

   2.2 Ist x strukturierte wertorientierte Komponente, also ein Array oder Record, bewirkt die Ausführung der Lese– bzw. Schreiboperation auf x die Ausführung entsprechender Lese– bzw. Schreiboperationen auf den Komponenten, aus denen x zusammengesetzt ist. Es gilt somit:
   $$CalledOps(x, op) = \bigcup_{k \in Comp(x)} \{(k, op)\} \cup CalledOps(k, op)$$
   wobei $Comp(x)$ die Menge der DE–Inkarnationen ist, aus denen x zusammengesetzt ist.

3. x ist DA–Generator
   Auf x ist die Operation *erzeuge* implizit definiert, d.h. es gilt: $op = erzeuge$. Die Ausführung von *erzeuge* auf x durch a bewirkt die Erzeugung einer DA–Inkarnation k bzgl. der durch den Generator x definierten Klasse.

   3.1 Es wird zunächst der Fall betrachtet, daß x kein K-Order–Generator ist. In diesem Fall wird unmittelbar nach Erzeugung der Inkarnation k die implizit auf k definierte Startoperation $i(k)$ aufgerufen. Die Ausführung von $i(k)$ bewirkt den Start der Ausführung der kanonischen Operation von k.
   (a) Ist k S-Order oder Depot, so wird $op(k)$ von dem Akteur a sequentiell in $op(y)$ eingeordnet ausgeführt. Die Ausführung von *erzeuge* terminiert in diesem Fall mit Terminierung der kanonischen Operation $op(k)$. Insofern werden also im Kontext der Ausführung von *erzeuge* auch all die äußeren Operationen aufgerufen, die im Kontext von $op(k)$ aufgerufen werden.
   (b) Ist k Akteur, so wird $op(k)$ von k parallel in $op(y)$ eingeordnet ausgeführt. Die Ausführung von *erzeuge* durch a terminiert in diesem Fall mit der Anfangssynchronisation von a mit k, also mit der Ausführung von $i(k)$ auf k. Es wird jedoch auch in diesem Fall die Sichtweise eingenommen, daß die Ausführung von *erzeuge* den Aufruf all der äußeren Operationen bewirkt, die im Kontext von $op(k)$ aufgerufen werden, da $op(k)$ die Operation ist, deren Ausführung durch den Aufruf von *erzeuge* auf x letztlich bezweckt wird.

[5]Dazu gehören Inkarnationen bzgl. der vordefinierten Generatoren und der aus diesen abgeleiteten Bereichsgeneratoren.

3.2 Ist x K-Order–Generator, so wird die implizit auf der K-Order k definierte Startoperation $i(k)$ bei der Annahme der K-Order durch den K-Akteur ka, der x als lokale N-Komponente enthält, ausgeführt. Die Ausführung von $i(k)$ auf k bewirkt die Ausführung von $op(k)$ durch ka. Die Ausführung von *erzeuge* durch a terminiert mit Terminierung von $op(k)$ und der Übernahme der Ergebnisparameter. Unter der Voraussetzung, daß der K-Akteur ka die K-Order k nach endlicher Zeit annimmt, bewirkt die Ausführung von *erzeuge* auf dem K-Order–Generator x also den Aufruf all der äußeren Operationen, die im Kontext der kanonischen Operation von k aufgerufen werden.

Mit der Ausführung des *erzeuge*–Aufrufs auf dem DA–Generator x wird also die Ausführung der kanonischen Operation der damit erzeugten DA–Inkarnation k bezweckt. Dementsprechend bewirkt die Ausführung von *erzeuge* den Aufruf der Startoperation $i(k)$ von k und all der äußeren Operationen, die im Kontext der Ausführung der kanonischen Operation $op(k)$ aufgerufen werden[6]. Die Menge der äußeren Operationen, die im Kontext der Ausführung von $op(k)$ aufgerufen werden, besteht aus den äußeren Operationen, die bei Ausführung des sequentiellen Kerns von $op(k)$ aufgerufen werden und aus allen äußeren Operationen, die bei Ausführung der sequentiellen Kerne der kanonischen Operationen der DA–Inkarnationen, die α–innen zu k sind, aufgerufen werden. Für die Menge $CalledOps(x, erzeuge)$ gilt somit:

$$CalledOps(x, erzeuge) = \{i(k)\} \cup CalledOps(x, op(k))$$

wobei $CalledOps(x, op(k))$ die Menge der äußeren Operationen bezeichnet, die im Kontext der Ausführung von $op(k)$ aufgerufen werden.

4. x ist DA–Inkarnation

4.1 op ist die implizit auf x definierte Startoperation $i(x)$
Die Ausführung von $i(x)$ bewirkt den Start der Ausführung der kanonischen Operation $op(x)$ von x und damit den Aufruf all der äußeren Operationen, die im Kontext der Ausführung von $op(x)$ aufgerufen werden (vgl. 3.). Es gilt somit:

$$CalledOps(x, i(x)) = CalledOps(x, op(x))$$

4.2 x ist Depot und op ist eine Zugriffsoperation von x
Die Zugriffsoperationen des Depots x sind die äußeren Operationen der lokalen N–Komponenten von x, die mit dem Attribut E definiert sind. Die Ausführung einer Zugriffsoperation von x entspricht also der Ausführung einer äußeren Operation einer lokalen N–Komponente k von x. Lokale N–Komponenten von x, die mit dem Attribut E definiert sein können, sind entweder Generatoren, DE–Inkarnationen, benannte Depots oder benannte K-Akteure. Im Fall, daß op eine Zugriffsoperation eines Depots x ist, gilt somit:

$$CalledOps(x, op) = CalledOps(k, op) \text{ mit } k \in L_0^E(x) \text{ und } op \in O(k)$$

4.3 x ist K-Akteur und op ist Kommunikationsoperation von x
Die Kommunikationsoperationen des K-Akteurs x werden durch die lokalen K-Order–Generatoren von x definiert und entsprechen den *erzeuge*–Operationen

[6] Unter der Voraussetzung, daß ein K-Akteur alle erteilten Aufträge zur Ausführung von K-Order nach endlicher Zeit annimmt.

auf diesen K-Order–Generatoren. Der Aufruf einer Kommunikationsoperation (K-Order–Aufruf) entspricht also dem Aufruf von *erzeuge* auf dem entsprechenden K-Order Generator. Für die Wirkungen im Hinblick auf den Aufruf weiterer äußerer Operationen bei Ausführung einer Kommunikationsoperation gelten somit die unter 3.2 für die Ausführung von *erzeuge* auf K-Order–Generatoren gemachten Aussagen. Daraus folgt:

$$CalledOps(x, op) = CalledOps(k, erzeuge)$$

wobei k der K-Order–Generator von x ist, durch den op definiert wird.

Mit 1. – 4. ist die Menge der äußeren Operationen, die im Kontext der Ausführung einer äußeren Operation op einer Komponente x aufgerufen werden, informell angegeben. Damit ist der in (5.4) definierte Konsistenzbegriff vollständig erklärt. Wie bereits erläutert, ist die Konsistenz des Rechts eines INSEL$^+$–Systems wesentliche Voraussetzung für die Erfüllung der mit der Wahrnehmung eines Rechts an einer Komponente verbundenen Leistungsanforderung. Es sind somit im weiteren Maßnahmen von Interesse, die dazu beitragen können, die Konsistenz des Rechts eines INSEL$^+$–Systems zu gewährleisten. Bevor das Spektrum derartiger Maßnahmen aufgezeigt wird, werden zunächst die Ursachen für Inkonsistenzen in dem Recht eines INSEL$^+$–Systems näher erläutert.

Ursachen für Inkonsistenzen

Die Komponenten eines INSEL$^+$–Systems lassen sich in zugriffskontrollierte und nicht zugriffskontrollierte Komponenten einteilen. Die Rechte zur Ausführung der äußeren Operationen nicht zugriffskontrollierter Komponenten sind gemäß Definition (5.3) implizit vergeben. Ist k eine nicht zugriffskontrollierte Komponente, so haben alle Akteure, in deren Ausführungsumgebung k liegt, implizit das Recht zur Ausführung aller äußeren Operationen von k[7]. Die Rechte an den nicht zugriffskontrollierten Komponenten eines INSEL$^+$–Systems sind damit implizit so vergeben, daß bei Aufruf einer äußeren Operation einer nicht zugriffskontrollierten Komponente kein Zugriffsverbot vorliegen kann. Daraus folgt unmittelbar, daß das Recht eines INSEL$^+$–Systems, das nur aus nicht zugriffskontrollierten Komponenten besteht, konsistent ist.

Inkonsistenzen in dem Recht eines INSEL$^+$–Systems können also lediglich durch die für die Nutzung zugriffskontrollierter Komponenten explizit vergebenen Rechte verursacht werden. Dabei muß es mindestens eine äußere Operation einer zugriffskontrollierten Komponente geben, die im Kontext der Ausführung einer äußeren Operation einer anderen Komponente aufgerufen wird. Ist diese Voraussetzung erfüllt, ist die Beantwortung der Frage, ob das Recht des INSEL$^+$–Systems tatsächlich inkonsistent ist, davon abhängig, wie die Rechte an den äußeren Operationen der zugriffskontrollierten Komponenten des Systems vergeben sind. Da diese Rechte auf Basis der Zugriffsrestriktionsausdrücke dieser Operationen vergeben werden, werden Inkonsistenzen in dem Recht eines INSEL$^+$–Systems letztlich dadurch verursacht, daß die Zugriffsrestriktionsausdrücke des Systems nicht geeignet bzw. nicht aufeinander abgestimmt konstruiert sind. Maßnahmen zur Gewährleistung der Konsistenz müssen dementsprechend bereits bei der Systemkonstruktion einsetzen.

5.2.3 Maßnahmen zur Gewährleistung der Konsistenz

Die Maßnahmen, die dazu beitragen können, die Konsistenz des Rechts eines INSEL$^+$–Systems zu gewährleisten, lassen sich unterscheiden in Maßnahmen, die im Zuge der Sy-

[7]Mit Ausnahme der Startoperation von k, falls k DA–Inkarnation ist.

stemkonstruktion, d.h. bei der Entwicklung und Übersetzung des entsprechenden INSEL$^+$–Programms Anwendung finden, und in Maßnahmen, die zur Laufzeit des Systems ergriffen werden. Die Maßnahmen der ersten Art bestehen zum einen in der **Berücksichtigung von Leitlinien**, die bei der Konstruktion eines INSEL$^+$–Systems im Hinblick auf die Durchsetzung der Forderung nach Konsistenz des Rechts des Systems zu beachten sind, und zum anderen in der **Durchführung statischer Analysen**, durch die ausgehend von einem INSEL$^+$–Programm Aussagen über die Konsistenz des Rechts des durch dieses Programm definierten INSEL$^+$–Systems gewonnen werden können. Die **Laufzeit–Maßnahmen** sind in INSEL$^+$–Systemen einzusetzen, für die die Konsistenz des Rechts nicht statisch nachgewiesen werden kann und in denen somit potentiell Inkonsistenzen auftreten können. Mit den Laufzeit–Maßnahmen wird das Ziel verfolgt, das tatsächliche Auftreten von Inkonsistenzen bzw. von Ausführungsabbrüchen begonnener Operationsausführungen aufgrund von Inkonsistenzen zu vermeiden.

1. Statische Konsistenzanalysen

Die Frage, ob das Recht eines INSEL$^+$–Systems konsistent ist, kann im allgemeinen nicht im Rahmen einer statischen Analyse des entsprechenden INSEL$^+$–Programms beantwortet werden, da der Wert von Zugriffsrestriktionsausdrücken in der Regel nicht konstant ist und somit die auf ihrer Basis vergebenen Rechte dynamisch zur Laufzeit vergeben werden. Das Vorhandensein eines Rechts kann zum Beispiel von einem Eintrag in einer Zugriffskontrolliste oder dem Wert einer Variablen abhängig sein. Dies soll an einem kleinen Beispiel verdeutlicht werden.

Beispiel

Gegeben seien zwei zugriffskontrollierte Komponenten $k1$ und $k2$ eines INSEL$^+$-Systems sowie die Zugriffsrestriktionsausdrücke zweier Operationen $op1 \in O^E(k1)$ und $op2 \in O^E(k2)$. Es gelte, daß $op2$ potentiell im Kontext der Ausführung von $op1$ aufgerufen wird:

R^{k1}_{op1} : `8 <= Zeit.Stunde AND Zeit.Stunde < 17;`
R^{k2}_{op2} : `IN_ACL(Caller.User, Caller.Role, THIS, THIS);`

`Zeit` sei eine globale DE-Inkarnation, über die in dem System eine virtuelle Zeit zur Verfügung gestellt wird (siehe dazu auch das in Abschnitt 4.6.1 erläuterte Beispiel eines Kontenverwaltungssystems). R^{k1}_{op1} legt fest, daß ein Akteur a, in dessen Ausführungsumgebung die Komponente $k1$ liegt, das Recht zur Ausführung von $op1$ auf $k1$ nur in der Zeit zwischen 8.00 Uhr und 17.00 Uhr besitzt. R^{k2}_{op2} legt fest, daß a das Recht zur Ausführung von $op2$ auf $k2$ hat, wenn für den Benutzer, in dessen Auftrag der Akteur seine Berechnungen ausführt, und dessen Rolle ein entsprechender Eintrag in der Zugriffskontrolliste von $k2$ vorhanden ist. Statisch läßt sich nun nicht nachweisen, daß in der Zeit zwischen 8.00 Uhr und 17.00 Uhr immer ein solcher Eintrag in der Zugriffskontrolliste von $k2$ enthalten ist. Potentiell kann also bei der Ausführung von $op1$ durch a eine Inkonsistenz auftreten, die dazu führt, daß die Ausführung von $op1$ aufgrund des fehlenden Rechts für die Operation $op2$ abgebrochen werden muß. Generell ist es jedoch möglich, daß immer ein entsprechender Zugriffskontrollisten–Eintrag vorhanden ist, was bedeuten würde, daß bei der Ausführung von $op1$ durch a kein Ausführungsabbruch aufgrund eines fehlenden Rechts für $op2$ auftreten würde.

◇

Die Konsistenz des Rechts eines INSEL$^+$–Systems läßt sich also im allgemeinen nicht statisch nachweisen. Es lassen sich jedoch Kriterien angeben, die statisch überprüfbar sind und deren Erfüllung für ein INSEL$^+$–System *gewährleistet*, daß das Recht des Systems konsistent

ist. Das Recht eines INSEL$^+$–Systems, dessen Konsistenz anhand dieser Kriterien statisch nachgewiesen werden kann, wird als **absolut konsistent** bezeichnet. Ist das Recht eines INSEL$^+$–Systems absolut konsistent, können zur Laufzeit des Systems keine Inkonsistenzen auftreten.

Beispiel

Gegeben seien wiederum die Zugriffsrestriktionsausdrücke zweier Operationen $op1 \in O^E(k1)$ und $op2 \in O^E(k2)$. Es gelte, daß $op2$ potentiell im Kontext der Ausführung von $op1$ aufgerufen wird. `Rolle1` und `Rolle2` seien die Bezeichner von Rollen, die für das entsprechende INSEL$^+$–System definiert sind.

R^{k1}_{op1} : `IN_ACL(Caller.User, Caller.Role, THIS, THIS) AND Caller.Role = Rolle1;`
R^{k2}_{op2} : `Caller.Role = Rolle1 OR Caller.Role = Rolle2;`

In diesem Fall ist statisch nachweisbar, daß immer dann, wenn der Zugriffsrestriktionsausdruck R^{k1}_{op1} erfüllt ist, auch R^{k2}_{op2} erfüllt ist und somit die Rechte an den Operationen $op1$ und $op2$ konsistent vergeben sind.

◇

Aus den Kriterien für die absolute Konsistenz lassen sich Leitlinien ableiten, an denen sich ein Systementwickler orientieren kann, wenn er ein INSEL$^+$–System mit absolut konsistentem Recht konstruieren möchte. Diese Leitlinien werden in Abschnitt 5.4 angegeben.

INSEL$^+$–Systeme mit absolut konsistentem Recht müssen einige restriktive Anforderungen erfüllen. Diese Anforderungen beziehen sich zum einen darauf, daß die funktionalen Abhängigkeiten zwischen Operationen in derartigen Systemen weitestgehend statisch sind. So ist z.B. die Verwendung von Zeigern nur sehr eingeschränkt möglich. Zum anderen sind die Möglichkeiten zur dynamischen Vergabe von Rechten zur Ausführung von Operationen, die potentiell im Kontext anderer Operationen aufgerufen werden, stark beschränkt. Die Zugriffsrestriktionsausdrücke dieser Operationen müssen bestimmte Eigenschaften haben, die lediglich die Formulierung relativ statischer Zugriffskontrollpolitiken erlauben. Für einige Anwendungssysteme sind die Kriterien, die für die absolute Konsistenz erfüllt sein müssen, zu restriktiv. Aus diesem Grund sind weniger restriktive statisch überprüfbare Bedingungen von Interesse, deren Erfüllung nicht die Konsistenz des Rechts gewährleistet, jedoch die Aussage zuläßt, daß das Recht des Systems konsistent sein *kann*. Dies motiviert die Einführung des Begriffs der potentiellen Konsistenz. Das Recht eines INSEL$^+$–Systems wird als **potentiell konsistent** bezeichnet, wenn es keine statisch feststellbaren Inkonsistenzen enthält. Ein Beispiel für eine statisch feststellbare Inkonsistenz ist im folgenden angegeben.

Beispiel

Gegeben seien wiederum die Zugriffsrestriktionsausdrücke zweier Operationen $op1 \in O^E(k1)$ und $op2 \in O^E(k2)$ mit den im vorhergehenden Beispiel gemachten Annahmen.

R^{k1}_{op1} : `Caller.Role = Rolle1;`
R^{k2}_{op2} : `Caller.Role = Rolle2;`

Es ist statisch nachweisbar, daß immer dann, wenn der Zugriffsrestriktionsausdruck R^{k1}_{op1} erfüllt ist, der Zugriffsrestriktionsausdruck R^{k2}_{op2} den Wert *false* hat. Hat ein Akteur a das Recht zur Ausführung von $op1$ und nimmt a dieses Recht wahr, so tritt also bei Ausführung

von $op1$ auf jeden Fall die Situation ein, daß a nicht das Recht zur Ausführung von $op2$ hat. Dies bedeutet jedoch gerade, daß in dem Recht des Systems eine Inkonsistenz vorliegt. ◇

Enthält das Recht eines INSEL$^+$–Systems mindestens eine statisch feststellbare Inkonsistenz, so wird dieses Recht als **absolut inkonsistent** bezeichnet. An jedes INSEL$^+$–System wird die Forderung gestellt, daß das Recht des Systems zumindest potentiell konsistent ist. Diese Minimalanforderung stellt sicher, daß das Recht eines INSEL$^+$–Systems keine Inkonsistenzen enthält, die bereits statisch feststellbar sind. Statisch festgestellte Inkonsistenzen sind somit vom Systementwickler auf jeden Fall zu beseitigen.

Jedes INSEL$^+$–Programm wird im Rahmen einer statischen Analysephase dahingehend untersucht, ob das Recht des durch das Programm definierten Systems absolut konsistent, lediglich potentiell konsistent oder absolut inkonsistent ist. Im Fall eines absolut inkonsistenten Rechts werden als Ergebnis der Analyse die statisch festgestellten Inkonsistenzen ausgegeben.

Zwischen den eingeführten Begriffen der absoluten Konsistenz, der potentiellen Konsistenz sowie der Konsistenz gemäß Definition (5.4) besteht folgender Zusammenhang:

$$\text{absolute Konsistenz} \implies \text{Konsistenz} \implies \text{potentielle Konsistenz}$$

Aus der absoluten Konsistenz des Rechts eines INSEL$^+$–Systems folgt also dessen Konsistenz und aus dieser wiederum die potentielle Konsistenz. Es zu beachten, daß diese Aussagen nicht umkehrbar sind. Dies bedeutet zum einen, daß ein konsistentes Recht im allgemeinen nicht absolut konsistent ist, und zum anderen, daß aus der potentiellen Konsistenz eines Rechts nicht notwendig dessen Konsistenz folgt.

Für Systeme, deren Recht lediglich potentiell konsistent ist, ist es somit möglich, daß zur Laufzeit des Systems Inkonsistenzen auftreten können. In derartigen Systemen sind Laufzeit-Maßnahmen mit dem Ziel einzusetzen, Ausführungsabbrüche begonnener Operationsausführungen aufgrund von Inkonsistenzen soweit wie möglich zu vermeiden.

2. Laufzeit–Maßnahmen

Um in Systemen mit lediglich potentiell konsistentem Recht nicht erst nach Beginn einer Operationsausführung festzustellen, daß ein für die vollständige Ausführung der Operation benötigtes Recht aufgrund einer Inkonsistenz fehlt, ist es sinnvoll, bereits bei Aufruf einer Operation zu überprüfen, ob alle für die vollständige Ausführung der Operation benötigten Rechte vorhanden sind und mit der Operationsausführung nur dann zu beginnen, wenn diese Überprüfung ein positives Ergebnis liefert. Durch diese **Maßnahme der vollständigen Rechteüberprüfung** bei Operationsaufruf läßt sich zwar nicht die Konsistenz des Rechts eines INSEL$^+$–Systems gewährleisten; es lassen sich jedoch Ausführungsabbrüche begonnener Operationsausführungen aufgrund eines inkonsistenten Rechts vermeiden, wenn gewährleistet ist, daß nach Beginn einer Operationsausführung keine Rechteänderungen bzgl. Operationen, von denen die ausgeführte Operation funktional abhängig ist, möglich sind. Damit die Überprüfung auf Vorhandensein aller benötigten Rechte bei Operationsaufruf möglich ist, müssen bereits zum Zeitpunkt des Aufrufs einer Operation zum einen alle zugriffskontrollierten Komponenten bekannt sein, auf denen im Kontext der Operationsausführung eine äußere Operation aufgerufen wird, und zum anderen die Zugriffsrestriktionsausdrücke all dieser äußeren Operationen vollständig auswertbar sein. Voraussetzung für die vollständige Rechteüberprüfung bei Operationsaufruf ist somit eine geeignete Konstruktion des Systems. Ob ein INSEL$^+$–System die entsprechenden Bedingungen erfüllt, kann anhand statisch überprüfbarer Kriterien analysiert werden. Auf eine detaillierte Beschreibung dieser Kriterien wird im weiteren verzichtet, da ihre Ausarbeitung analog zur Erarbeitung der Kriterien erfolgen kann, deren Erfüllung im Rahmen der statischen Konsistenzanalysen überprüft wird.

Zu den Maßnahmen, die zur Laufzeit eines INSEL$^+$–Systems durchgeführt werden können, gehören auch Verfahren, die gewährleisten, daß eine explizite Rechteänderung auf Basis der Änderung einer Zugriffskontrolliste (`ChangeACL`–Aufruf, siehe Abschnitt 4.4.3) nur dann erlaubt und ausgeführt wird, wenn dadurch keine Inkonsistenz in dem Recht des Systems verursacht wird. Diese Maßnahmen sind jedoch nur dann einsetzbar, wenn zum Zeitpunkt der Rechteänderung bekannt ist, von welchen Operationen die Operation, auf die sich die Rechteänderung bezieht, funktional abhängig ist, bzw. welche Operationen von dieser Operation funktional abhängig sind. Voraussetzung für die Durchführung dieser Verfahren ist somit wiederum eine geeignete Konstruktion des Systems. Auf entsprechende Kriterien sowie auf die Verfahren selbst wird in dieser Arbeit nicht weiter eingegangen. Verwiesen sei dazu auf die in [Eck93] im Rahmen der Beschreibung des Konzepts zur dezentralen Zugriffskontrolle angegebenen Verfahren zur konsistenten Rechteerweiterung bzw. Rechterücknahme.

In Systemen, für die sowohl die Kriterien für die absolute Konsistenz als auch die Anforderungen, deren Erfüllung Voraussetzung für die Durchführung der genannten Laufzeit–Maßnahmen ist, zu restriktiv sind, kann nicht sichergestellt werden, daß bei Ausführung einer Operation kein Zugriffsverbot auftreten kann. Die in diesen Systemen möglichen Zugriffsverbote sind explizit durch das in Abschnitt 4.5 beschriebene INSEL$^+$–Konzept zur Ausnahmebehandlung zu behandeln.

Zusammenfassung

In dem letzten Abschnitt wurde das Spektrum der Maßnahmen aufgezeigt, die dazu beitragen können, die Konsistenz des Rechts eines INSEL$^+$–Systems zu gewährleisten bzw. Abbrüche von Operationsausführungen aufgrund von Inkonsistenzen zu vermeiden. Dabei wurden die einzelnen Maßnahmen zunächst weitestgehend isoliert voneinander betrachtet. Im allgemeinen sind jedoch alle diese Maßnahmen bei der Konstruktion sowie zur Laufzeit eines INSEL$^+$–Systems in Kombination und aufeinander abgestimmt einzusetzen (siehe Abbildung 5.1).

Bei der Entwicklung eines INSEL$^+$–Systems sollten die **Leitlinien für die Konstruktion von Systemen mit absolut konsistentem Recht** soweit wie möglich berücksichtigt werden. Läßt sich das System aufgrund funktionaler Anforderungen oder der zu realisierenden Zugriffskontrollpolitik nicht komplett anhand dieser Leitlinien konstruieren, so sollte das Ziel verfolgt werden, nach diesen Leitlinien möglichst große Subsysteme mit absolut konsistentem Recht zu bilden.

Die **statischen Konsistenzanalysen** liefern Aussagen darüber, ob das Recht des Systems absolut konsistent, potentiell konsistent oder absolut inkonsistent ist. In dem Fall, daß das Recht des Systems absolut inkonsistent ist, sind die statisch festgestellten Inkonsistenzen zu beseitigen. Anschließend sind die statischen Konsistenzanalysen erneut durchzuführen, um zu überprüfen, ob die Änderungen zu einem zumindest potentiell konsistenten Recht geführt haben. Ist das Recht des Systems absolut konsistent, sind keine weitergehenden Maßnahmen einzusetzen, da gewährleistet ist, daß zur Laufzeit keine Inkonsistenzen auftreten können. Für ein System mit lediglich potentiell konsistentem Recht ist durch weitergehende statische Analysen zu überprüfen, ob die Voraussetzungen für die Durchführung der **Laufzeit–Maßnahmen** erfüllt sind. Wenn dies der Fall ist, sind diese Maßnahmen einzusetzen. Damit kann gewährleistet werden, daß in dem System keine Abbrüche begonnener Operationsausführungen aufgrund möglicher Inkonsistenzen auftreten. Sind die Voraussetzungen für den Einsatz der Laufzeit–Maßnahmen nicht erfüllt, kann nicht ausgeschlossen werden, daß im Kontext der Ausführung einer Operation ein Zugriffsverbot auftritt. Diese Zugriffsverbote und die damit möglichen Abbrüche von Operationausführungen sind durch eine explizite Ausnahmebehandlung zu behandeln.

Abbildung 5.1: Übersicht über die Maßnahmen zur Gewährleistung der Konsistenz des Rechts eines INSEL$^+$–Systems

5.3 Statische Konsistenzanalysen

In diesem Abschnitt werden die statischen Analysen erklärt, die ausgehend von einem INSEL$^+$–Programm Aussagen über die Konsistenz des Rechts des durch dieses Programm definierten INSEL$^+$–Systems liefern. Der Schwerpunkt liegt dabei auf der Erarbeitung und der Erläuterung der Kriterien, deren Erfüllung im Rahmen dieser Analysen überprüft wird.

Für die statische Analyse eines INSEL$^+$–Programms ist es zunächst notwendig, ausgehend von der textuellen Darstellung des Programms das Programm so aufzubereiten, daß die im Rahmen der jeweiligen Analyse interessierenden Eigenschaften in der aufbereiteten Programmbeschreibung zur Verfügung stehen. Die in diesem Abschnitt beschriebenen Konsistenzanalysen werden auf Basis des attributierten statischen Aufrufgraphen eines INSEL$^+$–Programms, der potentielle Aufrufbeziehungen und damit Nutzungsbeziehungen vergröbert

auf die in dem Programm definierten Generatoren und Inkarnationen beschreibt, durchgeführt. Der attributierte statische Aufrufgraph eines INSEL$^+$–Programms wird in Abschnitt 5.3.1 definiert. In Abschnitt 5.3.2 werden dann die Kriterien angegeben, anhand derer auf Basis des attributierten statischen Aufrufgraphen analysiert wird, ob das Recht des durch das Programm definierten INSEL$^+$–Systems potentiell konsistent oder absolut inkonsistent ist. Anschließend werden in Abschnitt 5.3.3 die Kriterien erläutert, die für die absolute Konsistenz des Rechts eines INSEL$^+$–Systems erfüllt sein müssen.

Die statischen Konsistenzanalysen werden in dieser Arbeit immer bezogen auf ein *gesamtes* INSEL$^+$–System betrachtet. Die einzelnen Kriterien lassen sich jedoch auch für die Konsistenzanalyse von Subsystemen eines INSEL$^+$–Systems verwenden. Sie können insbesondere als Basis für die Entwicklung von Analyseverfahren dienen, durch die in einem INSEL$^+$–System Subsysteme mit absolut konsistentem Recht identifiziert werden können.

5.3.1 Attributierter statischer Aufrufgraph eines INSEL$^+$–Programms

In diesem Abschnitt wird von einem INSEL$^+$–Programm in seiner textuellen Darstellung ausgegangen. Weiter wird für den Rest des gesamten Kapitels die folgende Annahme getroffen:

(5.5) Alle in einem INSEL$^+$–Programm verwendeten Bezeichner sind programmweit eindeutig.

Ein INSEL$^+$–Programm in textueller Darstellung besteht aus einer Menge von DA–Generatordefinitionen, die ineinander geschachtelt sind. Dabei ist zu beachten, daß mit der textuellen Definition eines Generators im allgemeinen nicht genau ein einzelner Generator festgelegt wird, sondern eine ganze Klasse von Generatoren.

Beispiel

Zur Verdeutlichung dieser Aussage sei das in Abbildung 5.2 dargestellte INSEL$^+$–Programm gegeben, das ein Erzeuger–Verbraucher–System mit einem Puffer beschränkter Kapazität implementiert. Dieses Programm entspricht im wesentlichen dem in Abbildung 3.2 angegebenen Programm. Der Unterschied besteht lediglich darin, daß nun gemäß Annahme (5.5) alle in dem Programm verwendeten Bezeichner programmweit eindeutig sind.

Im Deklarationsteil der Depot–Generatordefinition `PufferTyp` wird der Bereichsgenerator `IndexTyp` definiert, wobei die obere Grenze von `IndexTyp` durch den formalen Parameter `MaxAnzahl` festgelegt wird. Jede Erzeugung eines Depots bzgl. `PufferTyp` bewirkt also die Erarbeitung eines Bereichsgenerators, dessen obere Grenze sich aus dem jeweiligen aktuellen Wert für `MaxAnzahl` ergibt. Mit der durch den Text

```
TYPE IndexTyp IS 0 .. MaxAnzahl - 1;
```

festgelegten Bereichsgeneratordefinition `IndexTyp` wird somit nicht ein einzelner Bereichsgenerator definiert, sondern eine Klasse von Bereichsgeneratoren, die sich jeweils durch den Wert ihrer oberen Grenze unterscheiden können.

◇

Die in dem Beispiel anhand einer Bereichsgeneratordefinition verdeutlichte Betrachtungsweise kann auf alle Arten von Generatordefinitionen verallgemeinert werden. Mit jeder Generatordefinition eines INSEL$^+$–Programms wird somit eine **Generatorklasse** festgelegt. Die Begriffe Generatordefinition und Generatorklasse werden dementsprechend im weiteren synonym verwendet. Die in einem INSEL$^+$–Programm festgelegten Generatorklassen können nach den

```
PROCESS ErzeugerVerbraucherSystem IS

  -- Spezifikationsteil des Depot-Generators PufferTyp
  DEPOT TYPE SPEC PufferTyp (MaxAnzahl : IN integer) IS
    PROCEDURE TYPE SPEC Einfuegen (EinfuegeWert : IN  integer);
    PROCEDURE TYPE SPEC Entnehmen (EntnahmeWert : OUT integer);
  END PufferTyp;

  -- Implementierungsteil des Depot-Generators PufferTyp
  DEPOT TYPE PufferTyp (MaxAnzahl : IN integer) IS
    TYPE IndexTyp IS 0 .. MaxAnzahl-1;
    TYPE FeldTyp  IS ARRAY [IndexTyp] OF integer;
    Feld         : FeldTyp;
    Anfang, Ende : IndexTyp;

    PROCEDURE TYPE Einfuegen (EinfuegeWert : IN  integer) IS
    BEGIN
      Feld[Ende] := EinfuegeWert; Ende := (Ende + 1) MOD MaxAnzahl;
    END Einfuegen;

    PROCEDURE TYPE Entnehmen (EntnahmeWert : OUT integer) IS
    BEGIN
      EntnahmeWert := Feld[Anfang]; Anfang := (Anfang + 1) MOD MaxAnzahl;
    END Entnehmen;
  BEGIN Anfang := 0; Ende := 0; END PufferTyp;

  -- Deklaration eines benannten Puffers
  DEPOT Puffer : PufferTyp (100);
  --
  PROCESS TYPE Erzeuger (MaxErzeuge: IN integer; Start, Diff: IN integer) IS
    ErzeugerWert : integer := Start;
  BEGIN
    FOR i1 IN 1..MaxErzeuge LOOP
      Puffer.Einfuegen(ErzeugerWert); ErzeugerWert := ErzeugerWert + Diff;
    END LOOP;
  END Erzeuger;
  --
  PROCESS TYPE Verbraucher (MaxVerbrauche: IN integer) IS
    VerbraucherWert : integer;
  BEGIN
    FOR i2 IN 1..MaxVerbrauche LOOP
      Puffer.Entnehmen(VerbraucherWert);
    END LOOP;
  END Verbraucher;
  --
BEGIN
  FORK Erzeuger(4,1,1); FORK Verbraucher(4);
END ErzeugerVerbraucherSystem;
```

Abbildung 5.2: INSEL$^+$–Beispielprogramm: Erzeuger–Verbraucher–System

für INSEL– und INSEL$^+$-Systeme eingeführten Generatorarten klassifiziert werden (vgl. Abbildungen 4.5 und 3.1). So ist zu unterscheiden zwischen DE– und DA–Generatorklassen. Die DA–Generatorklassen können weiter unterteilt werden in Akteur–, Order– und Depot–Generatorklassen mit ihren jeweiligen Unterarten.

Beispiel (Fortsetzung)

In dem INSEL$^+$–Programm, das in Abbildung 5.2 dargestellt ist, werden neben der Generatorklasse `ErzeugerVerbraucherSystem`, die die Hauptkomponente definiert, die Depot-Generatorklasse `PufferTyp`, die PS-Order–Generatorklassen `Einfuegen` und `Entnehmen` sowie die M-Akteur–Generatorklassen `Erzeuger` und `Verbraucher` festgelegt. Im Deklarationsteil der DA–Generatordefinition `PufferTyp` werden die DE–Generatorklassen `IndexTyp` und `FeldTyp` festgelegt.[8]

◇

Wie in Kapitel 3 erklärt, enthält eine DA–Generatordefinition einen Deklarationsteil und einen Anweisungsteil. Der Deklarationsteil einer DA–Generatordefinition kann neben den Definitionen von DE– und DA–Generatoren Deklarationen benannter DE– und DA–Inkarnationen sowie darüberhinaus in Zeiger–Deklarationen mit einem Initialisierungsteil Generierungsausdrücke für anonyme DE– und DA–Inkarnationen enthalten. In dem Anweisungsteil einer DA–Generatordefinition können insbesondere Anweisungen auftreten, deren Ausführung die Erzeugung von DA–Inkarnationen bewirkt. Die INSEL$^+$–Sprachkonstrukte, deren Erarbeitung bzw. Ausführung die Erzeugung einer Inkarnation bewirkt, werden – wie in Kapitel 3 eingeführt – als Erzeugungskonstrukte bezeichnet. Erzeugungskonstrukte für DE–Inkarnationen sind DE–Deklarationen und Generierungsausdrücke für anonyme DE–Inkarnationen. Zu den Erzeugungskonstrukten für DA–Inkarnationen gehören S-Order–Aufrufe, K-Order–Aufrufe, M-Akteur–Aufrufe, Depot– und K-Akteur–Deklarationen sowie Generierungsausdrücke für anonyme Depots und K-Akteure. Die oben für Generatordefinitionen eingenommene Betrachtungsweise kann analog auf Erzeugungskonstrukte übertragen werden. So wird durch ein Erzeugungskonstrukt im allgemeinen nicht genau eine Inkarnation festgelegt, sondern eine Klasse von Inkarnationen.

Beispiel (Fortsetzung)

Der Anweisungsteil der M-Akteur–Generatordefinition `Erzeuger` besteht aus einer For–Anweisung, die einen PS-Order–Aufruf enthält. Dieser PS-Order–Aufruf wird im allgemeinen abhängig von dem Wert des formalen Parameters `MaxErzeuge` mehrfach ausgeführt. Dementsprechend wird mit dem PS-Order–Aufruf

```
Puffer.Einfuegen (ErzeugerWert);
```

nicht eine einzelne PS-Order festgelegt, sondern eine Klasse von PS-Ordern, die sich jeweils in dem Wert ihres formalen Parameters `ErzeugerWert` unterscheiden.

◇

Die in dem Beispiel anhand eines PS-Order–Aufrufs verdeutlichte Betrachtungsweise kann auf alle Arten von Erzeugungkonstrukten übertragen werden. Durch jedes in einem INSEL$^+$–Programm auftretende Erzeugungskonstrukt wird somit eine **Inkarnationsklasse** festgelegt.

Ein Erzeugungskonstrukt enthält genau einen INSEL$^+$–Namen, der die Generatorklasse identifiziert, bzgl. der bei Erarbeitung bzw. Ausführung des Erzeugungskonstrukts eine Inkarnation zu erzeugen ist. Ist I die durch ein Erzeugungskonstrukt $\mathcal{E}$ festgelegte Inkarnationsklasse

[8] Zu den DE–Generatorklassen jedes INSEL$^+$–Programms gehören zusätzlich zu den in dem Programm explizit definierten DE–Generatoren die vordefinierten Generatoren `integer`, `real`, `boolean`, `character` und `string`.

und G die Generatorklasse, die mit dem in $\mathcal{E}$ enthaltenen INSEL$^+$–Namen identifiziert wird, so wird im weiteren davon gesprochen, daß sich die Inkarnationsklasse I auf die Generatorklasse G **bezieht**.

Beispiel (Fortsetzung)

Durch die im Deklarationsteil der Generatorklasse `ErzeugerVerbraucherSystem` enthaltene Depot–Deklaration

```
DEPOT Puffer : PufferTyp (100);
```

wird die Depot–Inkarnationsklasse `Puffer` festgelegt. Die Inkarnationsklasse `Puffer` bezieht sich dabei auf die Depot–Generatorklasse `PufferTyp`.

Inkarnationsklassen, die nicht durch Deklarationen, sondern durch Erzeugungskonstrukte, deren Ausführung die Erzeugung einer anonymen Inkarnation bewirkt, festgelegt sind, können nicht mit dem in der Deklaration festgelegten Namen bezeichnet werden. Sie werden deshalb im weiteren mit einem Namen bezeichnet, der sich aus dem Namen der Generatorklasse, auf die sich die jeweilige Inkarnationsklasse bezieht, und einer fortlaufenden Nummer zusammensetzt.[9] Die durch den PS-Order–Aufruf

```
Puffer.Einfuegen (ErzeugerWert);
```

festgelegte Inkarnationsklasse wird dementsprechend mit `Einfuegen1` bezeichnet. Weiterhin werden in dem Beispielprogramm die PS-Order–Inkarnationsklasse `Entnehmen1` sowie die M-Akteur–Inkarnationsklassen `Erzeuger1` und `Verbraucher1` festgelegt. Zu den durch DE-Deklarationen festgelegten DE–Inkarnationsklassen gehören z.B. `MaxAnzahl`, `Feld`, `Anfang` und `Ende`.

◇

Aus den bisherigen Erläuterungen dieses Abschnitts sollte deutlich geworden sein, daß ein INSEL$^+$–Programm als eine strukturierte Menge von Generator– und Inkarnationsklassen beschrieben werden kann. In der folgenden Definition wird diese Beschreibungsform eines INSEL$^+$–Programms präzisiert.

Definition 5.6.: INSEL$^+$–Programm

Ein **INSEL$^+$–Programm** $\mathcal{P}$ wird durch das Tripel $\mathcal{P} = (\mathcal{D}, \delta, \gamma, \iota)$ beschrieben, wobei $\mathcal{D}$ die Menge der Generator– und Inkarnationsklassen des Programms $\mathcal{P}$ ist und δ, γ und ι zweistellige Relationen über Teilmengen von $\mathcal{D}$ sind. Es werden folgende Bezeichnungen für Teilmengen von $\mathcal{D}$ eingeführt:

$\mathcal{D}^G$	Menge der Generatorklassen
$\mathcal{D}^{DEG}$	Menge der DE–Generatorklassen
$\mathcal{D}^{DAG}$	Menge der DA–Generatorklassen
$\mathcal{D}^I$	Menge der Inkarnationsklassen
$\mathcal{D}^{DEI}$	Menge der DE–Inkarnationsklassen
$\mathcal{D}^{DAI}$	Menge der DA–Inkarnationsklassen
$\mathcal{D}^{NDAI}$	Menge der benannten DA–Inkarnationsklassen

Die Relation $\delta \subset \mathcal{D}^{DAG} \times \mathcal{D}^{DAG}$ beschreibt die Abhängigkeiten, die sich aus der Schachtelungsstruktur des Programms zwischen den DA–Generatorklassen des Programms

[9] Der INSEL$^+$–Übersetzer erzeugt im Rahmen der syntaktischen und semantischen Analyse eines INSEL$^+$–Programms für jede in dem Programm enthaltene Generator– und Inkarnationsklasse einen eindeutigen Integer–Wert als Namen. Darauf wird hier jedoch nicht weiter eingegangen.

ergeben. Seien $x, y \in \mathcal{D}^{DAG}$, dann gilt:

$$(x, y) \in \delta \overset{\triangle}{\Longleftrightarrow} y \text{ ist im Deklarationsteil von } x \text{ enthalten}$$

Die Relation $\gamma \subset \mathcal{D}^G \times \mathcal{D}^I$ ordnet jeder Inkarnationsklasse die Generatorklasse zu, auf die sich die Inkarnationsklasse bezieht. Seien $g \in \mathcal{D}^G$ und $i \in \mathcal{D}^I$, dann gilt:

$$(g, i) \in \gamma \overset{\triangle}{\Longleftrightarrow} g \text{ ist die Generatorklasse, auf die sich } i \text{ bezieht}$$

Die Relation $\iota \subset \mathcal{D}^{DAG} \times \mathcal{D}^I$ ordnet jeder Inkarnationsklasse die Generatorklasse zu, in der das Erzeugungskonstrukt, das die Inkarnationsklasse festlegt, enthalten ist. Seien $g \in \mathcal{D}^{DAG}$ und $i \in \mathcal{D}^I$, $\mathcal{E}$ sei das Erzeugungskonstrukt, durch das i festgelegt ist, dann gilt:

$$(g, i) \in \iota \overset{\triangle}{\Longleftrightarrow} \mathcal{E} \text{ ist in } g \text{ enthalten}$$

□

Die Relation δ ordnet jeder DA–Generatorklasse $y \in \mathcal{D}^{DAG}$ die DA–Generatorklasse $x \in \mathcal{D}^{DAG}$ zu, in deren Deklarationsteil y enthalten ist. Sie beschreibt damit, wie die DA–Generatorklassen eines INSEL$^+$–Programms $\mathcal{P}$ ineinander geschachtelt sind. Aus diesem Grund wird das Paar $(\mathcal{D}^{DAG}, \delta)$ **Schachtelungsstruktur** des Programms $\mathcal{P}$ genannt. Interpretiert man $\mathcal{D}^{DAG}$ als Knotenmenge und δ als Kantenmenge eines Graphen, so ergibt sich, daß die Schachtelungsstruktur $(\mathcal{D}^{DAG}, \delta)$ ein Baum ist, dessen Wurzel die Generatorklasse ist, die die Hauptkomponente definiert.

Beispiel

Das INSEL$^+$–Beispielprogramm aus Abbildung 5.2, das im weiteren mit EVS bezeichnet wird, wird durch das Tripel $EVS = (\mathcal{D}_{EVS}, \delta_{EVS}, \gamma_{EVS}, \iota_{EVS})$ beschrieben. Dabei gilt:

– $\mathcal{D}_{EVS} = \{\mathcal{D}^{DAG}_{EVS} \cup \mathcal{D}^{DEG}_{EVS} \cup \mathcal{D}^{DAI}_{EVS} \cup \mathcal{D}^{DEI}_{EVS}\}$

  $\mathcal{D}^{DAG}_{EVS} = \{\texttt{ErzeugerVerbraucherSystem}, \texttt{PufferTyp}, \texttt{Einfuegen}, \texttt{Entnehmen}, \texttt{Erzeuger}, \texttt{Verbraucher}\}$

  $\mathcal{D}^{DEG}_{EVS} = \{\texttt{IndexTyp}, \texttt{FeldTyp}, \texttt{integer}, \texttt{real}, \texttt{boolean}, \texttt{character}, \texttt{string}\}$

  $\mathcal{D}^{DAI}_{EVS} = \{\texttt{Puffer}, \texttt{Erzeuger1}, \texttt{Verbraucher1}, \texttt{Einfuegen1}, \texttt{Entnehmen1}\}$

  $\mathcal{D}^{DEI}_{EVS} = \{\texttt{MaxAnzahl}, \texttt{Feld}, \texttt{Anfang}, \texttt{Ende}, \texttt{EinfuegeWert}, \texttt{EntnahmeWert}, \texttt{MaxErzeuge}, \texttt{Start}, \texttt{Diff}, \texttt{ErzeugerWert}, \texttt{i1}, \texttt{MaxVerbrauche}, \texttt{VerbraucherWert}, \texttt{i2}\}$

– $\delta_{EVS} = \{(\texttt{ErzeugerVerbraucherSystem}, \texttt{PufferTyp}), (\texttt{PufferTyp}, \texttt{Einfuegen}), (\texttt{PufferTyp}, \texttt{Entnehmen}), (\texttt{ErzeugerVerbraucherSystem}, \texttt{Erzeuger}), (\texttt{ErzeugerVerbraucherSystem}, \texttt{Verbraucher})\}$

– $\gamma_{EVS} = \{(\texttt{PufferTyp}, \texttt{Puffer}), (\texttt{Erzeuger}, \texttt{Erzeuger1}), (\texttt{Verbraucher}, \texttt{Verbraucher1}), (\texttt{Einfuegen}, \texttt{Einfuegen1}), (\texttt{Entnehmen}, \texttt{Entnehmen1}), (\texttt{integer}, \texttt{MaxAnzahl}), (\texttt{FeldTyp}, \texttt{Feld}), (\texttt{IndexTyp}, \texttt{Anfang}), (\texttt{IndexTyp}, \texttt{Ende}), (\texttt{integer}, \texttt{EinfuegeWert}), (\texttt{integer}, \texttt{EntnahmeWert}), (\texttt{integer}, \texttt{MaxErzeuge}), (\texttt{integer}, \texttt{Start}), (\texttt{integer}, \texttt{Diff}), (\texttt{integer}, \texttt{ErzeugerWert}), (\texttt{integer}, \texttt{i1}), (\texttt{integer}, \texttt{MaxVerbrauche}), (\texttt{integer}, \texttt{VerbraucherWert}), (\texttt{integer}, \texttt{i2})\}$

– $\iota_{EVS} = \{(\texttt{ErzeugerVerbraucherSystem}, \texttt{Puffer}), (\texttt{Erzeuger}, \texttt{Einfuegen1}), (\texttt{ErzeugerVerbraucherSystem}, \texttt{Erzeuger1}), (\texttt{ErzeugerVerbraucherSystem}, \texttt{Verbraucher1}), (\texttt{Verbraucher}, \texttt{Entnehmen1}), (\texttt{PufferTyp}, \texttt{MaxAnzahl}), (\texttt{PufferTyp}, \texttt{Feld}), (\texttt{PufferTyp}, \texttt{Anfang}), (\texttt{PufferTyp}, \texttt{Ende}), (\texttt{Einfuegen}, \texttt{EinfuegeWert}), (\texttt{Entnehmen}, \texttt{EntnahmeWert}), (\texttt{Erzeuger}, \texttt{MaxErzeuge}), (\texttt{Erzeuger}, \texttt{Start}), (\texttt{Erzeuger}, \texttt{Diff}), (\texttt{Erzeuger}, \texttt{ErzeugerWert}), (\texttt{Erzeuger}, \texttt{i1}), (\texttt{Verbraucher}, \texttt{MaxVerbrauche}), (\texttt{Verbraucher}, \texttt{VerbraucherWert}), (\texttt{Verbraucher}, \texttt{i2})\}$

Die Schachtelungsstruktur $(\mathcal{D}_{EVS}^{DAG}, \delta_{EVS})$ des Programms EVS ist in Abbildung 5.3 dargestellt.

◇

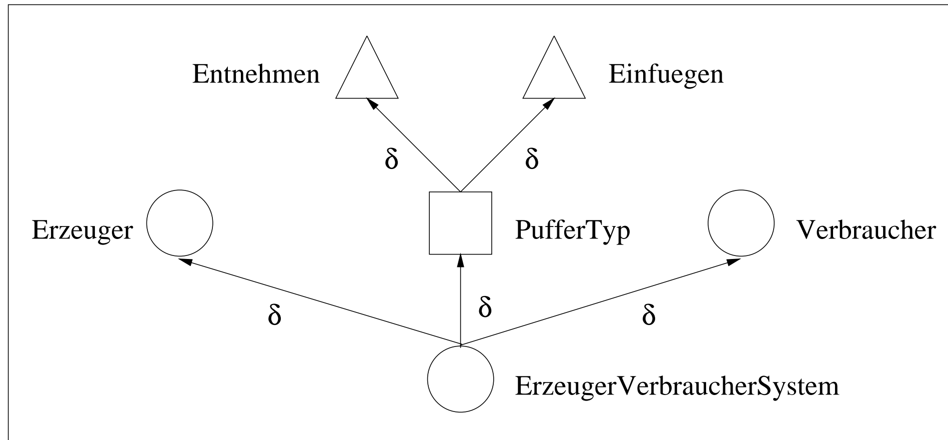

Abbildung 5.3: Schachtelungsstruktur des INSEL$^+$–Programms aus Abbildung 5.2

Vorbereitend für die Definition des attributierten statischen Aufrufgraphen eines INSEL$^+$–Programms werden im folgenden auf Basis der angegebenen Beschreibungsform zwei zweistellige Relationen über der Menge der DA–Generatorklassen des Programms definiert, die potentielle Nutzungsbeziehungen auf Ebene der DA–Generatorklassen beschreiben.

Definition 5.7.: Nutzungsbeziehung zwischen DA–Generatorklassen (Relation β)

Gegeben sei ein INSEL$^+$–Programm $\mathcal{P} = (\mathcal{D}, \delta, \gamma, \iota)$ in der in Definition (5.6) festgelegten Beschreibungsform. Weiter seien $x, y \in \mathcal{D}^{DAG}$ zwei DA–Generatorklassen.
Die Relation $\beta \subseteq \mathcal{D}^{DAG} \times \mathcal{D}^{DAG}$ ist wie folgt definiert:

$$(x, y) \in \beta \overset{\triangle}{\Longleftrightarrow} \exists i \in \mathcal{D}^{DAI} : (y, i) \in \gamma \wedge (x, i) \in \iota$$

Die Relation $\beta_a \subseteq \beta$ ist eine Teilmenge von β; sie ist folgendermaßen definiert:

$$(x, y) \in \beta_a \overset{\triangle}{\Longleftrightarrow} \exists i \in \mathcal{D}^{DAI} \setminus \mathcal{D}^{NDAI} : (y, i) \in \gamma \wedge (x, i) \in \iota$$

□

Zwei DA–Generatorklassen x und y sind also genau dann Element der Relation β – $(x,y) \in \beta$ – wenn in x mindestens ein Erzeugungskonstrukt enthalten ist, durch das eine Inkarnationsklasse festgelegt ist, die sich auf y bezieht. Oder anders gesagt: wenn in x mindestens ein Erzeugungskonstrukt auftritt, dessen Erarbeitung bzw. Ausführung die Erzeugung einer Inkarnation bzgl. eines Generators der DA–Generatorklasse y bewirkt. Die Relation β_a schränkt β auf die Paare ein, die durch Erzeugungskonstrukte festgelegt sind, deren Erarbeitung bzw. Ausführung die Erzeugung einer anonymen DA–Inkarnation bewirkt. Gilt $(x,y) \in \beta$ bzw. $(x,y) \in \beta_a$, so folgt daraus, daß zur Laufzeit des Programms bei Ausführung der kanonischen Operation einer Inkarnation bzgl. eines Generators der Generatorklasse x potentiell eine Inkarnation bzgl. eines Generators der Generatorklasse y erzeugt wird. Falls $(x,y) \in \beta$ und $(x,y) \notin \beta_a$ gilt[10], werden bei Ausführung der kanonischen Operation einer Inkarnation bzgl. eines Generators der Generatorklasse x höchstens benannte DA–Inkarnationen bzgl. eines Generators der Generatorklasse y erzeugt. Die mit β bzw. β_a erfaßten Abhängigkeiten sind potentiell, da Erzeugungskonstrukte u.a. in bedingten Anweisungen auftreten können. Der Wert des Bedingungsausdrucks einer bedingten Anweisung läßt sich jedoch im allgemeinen nicht zur Übersetzungszeit bestimmen.

Beispiel

Für das in Abbildung 5.2 dargestellte INSEL$^+$–Programm EVS ergeben sich die Relationen β_{EVS} und $\beta_{a_{EVS}}$ wie folgt:

$$\begin{aligned}\beta_{EVS} &= \{(\texttt{ErzeugerVerbraucherSystem}, \texttt{PufferTyp}),\\ &\quad (\texttt{ErzeugerVerbraucherSystem}, \texttt{Erzeuger}),\\ &\quad (\texttt{ErzeugerVerbraucherSystem}, \texttt{Verbraucher}),\\ &\quad (\texttt{Erzeuger}, \texttt{Einfuegen}), (\texttt{Verbraucher}, \texttt{Entnehmen})\}\\ \beta_{a_{EVS}} &= \{(\texttt{ErzeugerVerbraucherSystem}, \texttt{Erzeuger}), (\texttt{Erzeuger}, \texttt{Einfuegen}),\\ &\quad (\texttt{ErzeugerVerbraucherSystem}, \texttt{Verbraucher}), (\texttt{Verbraucher}, \texttt{Entnehmen})\}\end{aligned}$$

◇

Auf Basis der in (5.7) eingeführten Relation β_a wird nun der attributierte statische Aufrufgraph eines INSEL$^+$–Programms definiert, der Ausgangspunkt für die im weiteren erklärten statischen Konsistenzanalysen ist. Der attributierte statische Aufrufgraph eines INSEL$^+$–Programms beschreibt potentielle Aufrufbeziehungen (Kanten des Graphen) vergröbert auf die in dem Programm festgelegten DA–Generatorklassen und benannten DA–Inkarnationsklassen (Knoten des Graphen). Er stellt damit eine Vergröberung aller möglichen Ausprägungen der Ausführungsstruktur der mit dem Programm definierten INSEL$^+$–Systeme dar. Die benannten DA–Inkarnationsklassen werden in dem Graph erfaßt, um im Rahmen der Konsistenzanalysen die in Zugriffsrestriktionsausdrücken mit Gleichungen der Form

`Caller.ActorName` = <*inselp-path*> bzw. `Caller.ConName` = <*inselp-path*>

festlegbaren Kontextrestriktionen (vgl. Abschnitt 4.4.4.3) berücksichtigen zu können.

Definition 5.8.: Attributierter statischer Aufrufgraph

Gegeben seien das INSEL$^+$–Programm $\mathcal{P} = (\mathcal{D}, \delta, \gamma, \iota)$ in der in Definition (5.6) festgelegten Beschreibungsform und die Relation β_a gemäß Definition (5.7).
Der **attributierte statische Aufrufgraph** $\mathcal{G}(\mathcal{P}) = (V, K, \mu)$ des Programms $\mathcal{P}$ ist definiert durch:

[10] y muß in diesem Fall Depot– oder K-Akteur–Generatordefinition sein.

1. die Menge $V \triangleq \mathcal{D}^{DAG} \cup \mathcal{D}^{NDAI}$ der Knoten;
2. die Menge $K \subseteq V \times V$ der Kanten mit:

$$(n_1, n_2) \in K \stackrel{\triangle}{\Longleftrightarrow} \begin{cases} (n_1, n_2) \in \beta_a & \text{falls} \quad n_1 \in \mathcal{D}^{DAG} \wedge n_2 \in \mathcal{D}^{DAG} \\ (n_1, n_2) \in \iota & \text{falls} \quad n_1 \in \mathcal{D}^{DAG} \wedge n_2 \in \mathcal{D}^{NDAI} \\ (g_1, n_2) \in \beta_a & \text{falls} \quad n_1 \in \mathcal{D}^{NDAI} \wedge n_2 \in \mathcal{D}^{DAG} \wedge (g_1, n_1) \in \gamma \\ (g_1, n_2) \in \iota & \text{falls} \quad n_1 \in \mathcal{D}^{NDAI} \wedge n_2 \in \mathcal{D}^{NDAI} \wedge (g_1, n_1) \in \gamma \end{cases}$$

3. die Markierungsfunktion μ für Knoten mit:

$$\mu : V \longrightarrow \{\mathit{true}, \mathit{false}\} \times NodeType$$
$$\mu(n) \triangleq (zra(n), \mathit{kindof}(n)) \quad \text{für} \quad n \in V$$

wobei $NodeType$, zra und kindof wie folgt festgelegt sind:

– $NodeType \triangleq \{\mathit{MAkteurGen}, \mathit{KAkteurGen}, \mathit{DepotGen}, \mathit{SOrderGen}, \mathit{KOrderGen}, \mathit{NKAkteur}, \mathit{NDepot}, \mathit{BenRep}, \mathit{HKomp}\}$

– $zra : V \longrightarrow \{\mathit{true}, \mathit{false}\}$

$$zra(n) \triangleq \begin{cases} \mathit{true} & \text{falls } n \text{ zugriffskontrollierte DA–Generatorklasse oder} \\ & \text{zugriffskontrollierte DA–Inkarnationsklasse ist} \\ \mathit{false} & \text{sonst} \end{cases}$$

– $\mathit{kindof} : V \longrightarrow NodeType$

$$\mathit{kindof}(n) \triangleq \begin{cases} \mathit{BenRep} & \text{falls } n \text{ Generatorklasse für Benutzer-} \\ & \text{repräsentanten ist} \\ \mathit{HKomp} & \text{falls } n \text{ Generatorklasse der Haupt-} \\ & \text{komponente ist} \\ \mathit{MAkteurGen} & \text{falls } n \text{ M-Akteur–Generatorklasse ist} \\ \mathit{KAkteurGen} & \text{falls } n \text{ K-Akteur–Generatorklasse ist} \\ \mathit{DepotGen} & \text{falls } n \text{ Depot–Generatorklasse ist} \\ \mathit{SOrderGen} & \text{falls } n \text{ S-Order–Generatorklasse ist} \\ \mathit{KOrderGen} & \text{falls } n \text{ K-Order–Generatorklasse ist} \\ \mathit{NKAkteur} & \text{falls } n \text{ benannte K-Akteur–Inkarnations-} \\ & \text{klasse ist} \\ \mathit{NDepot} & \text{falls } n \text{ benannte Depot–Inkarnations-} \\ & \text{klasse ist} \end{cases}$$

□

Gemäß Definition (5.8) existiert in dem attributierten statischen Aufrufgraph $\mathcal{G}(\mathcal{P})$ eines INSEL$^+$–Programms $\mathcal{P}$ zwischen zwei DA–Generatorklassen n_1 und n_2 genau dann eine Kante, wenn n_1 ein Erzeugungskonstrukt enthält, dessen Erarbeitung bzw. Ausführung die Erzeugung einer anonymen DA–Inkarnation bzgl. eines Generators der Generatorklasse n_2 bewirkt. Ist n_2 eine benannte DA–Inkarnationsklasse, so ist das Paar (n_1, n_2) Element der Kantenmenge, wenn n_1 die DA–Generatorklasse ist, in der das Erzeugungskonstrukt auftritt, mit dem n_2 festgelegt ist. Zwischen einer benannten DA–Inkarnationsklasse n_1 und einer DA–Generatorklasse n_2 gibt es genau dann eine Kante, wenn die Generatorklasse, auf die sich n_1 bezieht, ein Erzeugungskonstrukt für eine anonyme DA–Inkarnation bzgl. eines Generators der Generatorklasse n_2 enthält. Sind n_1 und n_2 zwei benannte DA–Inkarnationsklassen, so gilt $(n_1, n_2) \in K$ genau dann, wenn das Erzeugungskonstrukt, mit dem n_2 festgelegt

ist, in der Generatorklasse, auf die sich n_1 bezieht, auftritt. Eine Kante zwischen zwei DA–Generatorklassen n_1 und n_2 steht somit stellvertretend für eine zur Laufzeit bestehende α–Abhängigkeit zwischen anonymen Inkarnationen bzgl. n_1 und n_2. Im Fall, daß n_1 und n_2 benannte DA–Inkarnationsklassen sind, steht eine derartige Kante für eine α–Abhängigkeit zwischen Inkarnationen dieser Inkarnationsklassen.

Jedem Knoten $n \in V$ des Aufrufgraphen wird durch die Abbildung μ das Tupel $(zra(n), kindof(n))$ als Attribut zugeordnet. $zra(n)$ gibt an, ob die mit n festgelegte Klasse von DA–Generatoren bzw. benannten DA–Inkarnationen zugriffskontrolliert ist oder nicht. *kindof(n)* beschreibt die Art von n; dabei wird unterschieden zwischen den Unterarten der DA–Generatorklassen, zwischen benannten Depot– und K-Akteur–Inkarnationsklassen sowie den Generatorklassen für Benutzerrepräsentanten. Wie noch zu sehen sein wird, kommt insbesondere letzteren in den Konsistenzanalysen eine besondere Bedeutung zu.

Der statische Aufrufgraph eines INSEL$^+$–Programms ist im allgemeinen nicht zyklenfrei, da in INSEL$^+$ rekursive Operationsaufrufe möglich sind. Ferner ist der Aufrufgraph in der Regel nicht zusammenhängend; z.B. haben Depot– und K-Akteur–Generatorklassen, auf die sich lediglich benannte Depot– und K-Akteur–Inkarnationsklassen beziehen, keine eingehenden Kanten. Hat eine DA–Generatorklasse g, die entweder M-Akteur– oder Order–Generatorklasse ist oder Depot– bzw. K-Akteur–Generatorklasse ist, auf die sich keine benannten Inkarnationsklassen beziehen, keine eingehenden Kanten, so bedeutet dies, daß zur Laufzeit keine Inkarnationen bzgl. der zu g gehörenden Generatoren erzeugt werden.

Beispiel

Als Beispiel wird im folgenden der attributierte statische Aufrufgraph

$$\mathcal{G}(EVS) = (V_{EVS}, K_{EVS}, \mu_{EVS})$$

des INSEL$^+$–Programms aus Abbildung 5.2 angegeben. Es gilt:

- $V_{EVS} = \{$`ErzeugerVerbraucherSystem`, `PufferTyp`, `Einfuegen`, `Entnehmen`, `Erzeuger`, `Verbraucher`, `Puffer`$\}$

- $K_{EVS} = \{$(`ErzeugerVerbraucherSystem`, `Erzeuger`), (`Erzeuger`, `Einfuegen`), (`ErzeugerVerbraucherSystem`, `Verbraucher`), (`Verbraucher`, `Entnehmen`), (`ErzeugerVerbraucherSystem`, `Puffer`)$\}$

- μ_{EVS}(`ErzeugerVerbraucherSystem`) = (*false*, *MAkteurGen*)
  μ_{EVS}(`PufferTyp`) = (*false*, *DepotGen*)
  μ_{EVS}(`Einfuegen`) = (*false*, *SOrderGen*)
  μ_{EVS}(`Entnehmen`) = (*false*, *SOrderGen*)
  μ_{EVS}(`Erzeuger`) = (*false*, *MAkteurGen*)
  μ_{EVS}(`Verbraucher`) = (*false*, *MAkteurGen*)
  μ_{EVS}(`Puffer`) = (*false*, *NDepot*)

In Abbildung 5.4 ist der statische Aufrufgraph $\mathcal{G}(EVS)$ bildlich dargestellt.

Der Aufrufgraph ist aufgrund der Einfachheit des Beispielprogramms wenig komplex. Abbildung 5.5 zeigt deshalb auszugsweise den statischen Aufrufgraph des in Anhang B angegebenen INSEL$^+$–Programms, das das in Abschnitt 4.6.1 erklärte Kontenverwaltungssystem einer Bank implementiert. Der Aufrufgraph des Kontenverwaltungssystems wird im weiteren dieses Kapitels noch mehrfach zur Verdeutlichung der eingeführten Begriffe und Analyseverfahren herangezogen werden.

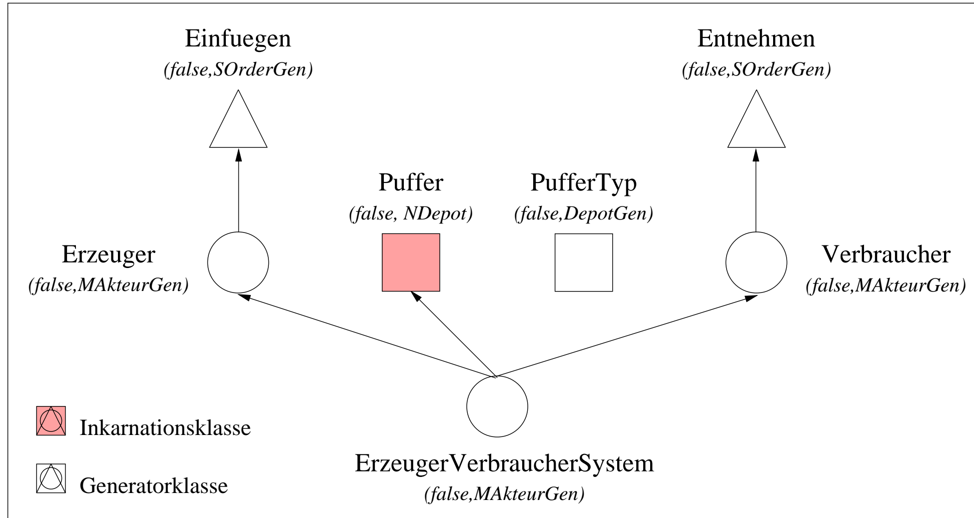

Abbildung 5.4: Statischer Aufrufgraph des INSEL$^+$–Programms aus Abbildung 5.2

$\diamond$

Für die auf Basis des statischen Aufrufgraphen durchzuführenden Konsistenzanalysen sind die Pfade, die zwischen zwei Knoten des Graphen existieren, von Interesse. Da der Graph – wie bereits erwähnt – Zyklen enthalten kann, ist zwischen zyklischen und zyklenfreien Pfaden zu unterscheiden.

Definition 5.9.: Zyklischer und zyklenfreier Aufrufpfad

Gegeben seien der attributierte statische Aufrufgraph $\mathcal{G}(\mathcal{P}) = (V, K, \mu)$ eines INSEL$^+$–Programms $\mathcal{P}$ und zwei Knoten $x, y \in V$.

1. Eine Folge von Knoten $(v_0, v_1, \ldots, v_n)$ mit $n \geq 1$, $v_i \in V$ für alle $i \in \{0, \ldots, n\}$ und $v_0 = x$ und $v_n = y$ heißt **Aufrufpfad der Länge n zwischen x und y** $\stackrel{\triangle}{\Longleftrightarrow}$

$$(v_{i-1}, v_i) \in K \quad \text{für alle } i \in \{1, \ldots, n\}$$

Sei $p = (v_0, v_1, \ldots, v_n)$ im weiteren ein Aufrufpfad der Länge n zwischen zwei Knoten v_0 und v_n.

2. Der Aufrufpfad p heißt **Zyklus** $\stackrel{\triangle}{\Longleftrightarrow}$

$$v_0 = v_n \wedge v_i \neq v_j \text{ für alle } i, j \in \{1, \ldots, n\} \text{ mit } i \neq j$$

3. Der Aufrufpfad p ist **zyklisch** $\stackrel{\triangle}{\Longleftrightarrow}$

$$\text{es gibt } i, j \in \{0, \ldots, n\} \text{ mit } i < j \text{ so, daß } (v_i, \ldots, v_j) \text{ ein Zyklus ist}$$

Ein Aufrufpfad ist also genau dann zyklisch, wenn er mindestens einen Zyklus enthält, d.h. wenn es mindestens einen Knoten gibt, der in dem Aufrufpfad mehrfach auftritt.

4. Der Aufrufpfad p ist **zyklenfrei** $\stackrel{\triangle}{\Longleftrightarrow}$

$$\text{für alle } i, j \in \{0, \ldots, n\} \text{ mit } i < j \text{ gilt: } (v_i, \ldots, v_j) \text{ ist kein Zyklus}$$

Ein Aufrufpfad ist also genau dann zyklenfrei, wenn er keinen Zyklus enthält, d.h. wenn es keinen Knoten gibt, der in dem Aufrufpfad mehrfach auftritt.

Abbildung 5.5: Statischer Aufrufgraph des Kontenverwaltungssystems (Ausschnitt)

5. $AP(x, y)$ bezeichne die Menge aller Aufrufpfade zwischen den Knoten x und y.
6. $AP_{zf}(x, y) \subseteq AP(x, y)$ bezeichne die Menge aller Aufrufpfade zwischen x und y, die zyklenfrei sind.
7. $AP(\mathcal{G}(\mathcal{P})) \triangleq \bigcup_{x,y \in V} AP(x, y)$ bezeichne die Menge aller Aufrufpfade von $\mathcal{G}(\mathcal{P})$.

□

Gibt es zwischen zwei Knoten x und y einen zyklischen Aufrufpfad, so folgt, daß die Menge $AP(x, y)$ aller Aufrufpfade zwischen x und y unendlich ist; die Menge $AP_{zf}(x, y)$ der zyklenfreien Aufrufpfade zwischen x und y ist immer endlich. Im folgenden werden für Aufrufpfade fünf Abbildungen definiert, die im Rahmen der Konsistenzanalysen für die Berücksichtigung der in Zugriffsrestriktionsausdrücken festlegbaren Kontextrestriktionen benötigt werden. Die Abbildungen lcg (LastConGen), lcn (LastConName), lag (LastActorGen) und lan (LastActorName) sind so definiert, daß sie zu den in Definition (4.11) festgelegten Elementen $ConGen_t(a)$, $ConName_t(a)$, $ActorGen_t(a)$ und $ActorName_t(a)$ des Kontextattributes eines Akteurs a korrespondieren. Sie ordnen einem Aufrufpfad jeweils das Element des Pfades zu, dem der Wert des jeweiligen Elements des Kontextattributs eines Akteurs, der gemäß des Aufrufpfades einen Operationsaufruf bzgl. des letzten Elements des Pfades ausführt, entspricht. Die Abbildung lak (LastActorOrKOrder) ist lediglich eine Hilfsabbildung, die für die Definition von lag und lan benötigt wird.

Definition 5.10.: Pfad–Abbildungen

Gegeben seien das INSEL$^+$–Programm $\mathcal{P} = (\mathcal{D}, \delta, \gamma, \iota)$ und der attributierte statische Aufrufgraph $\mathcal{G}(\mathcal{P}) = (V, K, \mu)$ von $\mathcal{P}$. $p \in AP(\mathcal{G}(\mathcal{P}))$ sei ein Aufrufpfad mit $p = (v_0, v_1, \ldots, v_n)$.

Die **Pfad–Abbildungen** lcg (LastConGen), lcn (LastConName), lak (LastActorOrKOrder), lag (LastActorGen), und lan (LastActorName) sind wie folgt definiert:

1. $$lcg : AP(\mathcal{G}(\mathcal{P})) \longrightarrow \mathcal{D}^{DAG}$$

$$lcg(p) \triangleq \begin{cases} v_{n-1} & \text{falls } v_{n-1} \in \mathcal{D}^{DAG} \\ \gamma(v_{n-1}) & \text{falls } v_{n-1} \in \mathcal{D}^{NDAI} \end{cases}$$

Die Abbildung lcg ordnet dem Aufrufpfad p die DA–Generatorklasse des vorletzten Elements von p zu, also v_{n-1}, falls v_{n-1} DA–Generatorklasse ist, bzw. die DA–Generatorklasse, auf die sich v_{n-1} bezieht, falls v_{n-1} benannte DA–Inkarnationsklasse ist. Für das Element $ConGen_t(a)$ des Kontextattributs eines Akteurs, der gemäß des Pfades p einen Operationsaufruf bzgl. des letzten Elements v_n von p ausführt, gilt somit: $ConGen_t(a) = lcg(p)$.

2. $$lcn : AP(\mathcal{G}(\mathcal{P})) \longrightarrow \mathcal{D}^{NDAI} \cup \{\perp\}$$

$$lcn(p) \triangleq \begin{cases} v_{n-1} & \text{falls } v_{n-1} \in \mathcal{D}^{NDAI} \\ \perp & \text{sonst} \end{cases}$$

Die Abbildung lcn ordnet dem Aufrufpad p das vorletzte Element von p zu, falls dieses eine benannte Depot– oder K-Akteur–Inkarnationsklasse ist. Ist v_{n-1} keine benannte DA–Inkarnationsklasse, ist der Wert von $lcn(p)$ undefiniert. Für das Element $ConName_t(a)$ des Kontextattributs eines Akteurs, der gemäß des Pfades p einen Operationsaufruf bzgl. des letzten Elements v_n von p ausführt, gilt somit: $ConName_t(a) = lcn(p)$.

3.
$$lak : AP(\mathcal{G}(\mathcal{P})) \longrightarrow \mathcal{D} \cup \{\bot\}$$

$$lak(p) \triangleq \begin{cases} v_i & \text{falls } i \in \{0,\ldots,n-1\} \wedge (kindof(v_i) = MAkteurGen \vee \\ & \quad kindof(v_i) = KAkteurGen \vee kindof(v_i) = NKAkteur \vee \\ & \quad kindof(v_i) = KOrderGen) \vee kindof(v_i) = BenRep \wedge \\ & \quad \forall j \in \{i+1,\ldots,n-1\} : (kindof(v_j) \neq MAkteurGen \wedge \\ & \quad kindof(v_j) \neq KAkteurGen \wedge kindof(v_j) \neq NKAkteur \wedge \\ & \quad kindof(v_j) \neq KOrderGen) \wedge kindof(v_j) \neq BenRep) \\ \bot & \text{sonst} \end{cases}$$

Die Abbildung lak projiziert den Aufrufpfad p auf das letzte in p enthaltene Element ungleich v_n, das Akteur–Generatorklasse, benannte K-Akteur–Inkarnationsklasse oder K-Order–Generatorklasse ist. Existiert kein solches Element in p, ist der Wert von $lak(p)$ undefiniert.

4.
$$lag : AP(\mathcal{G}(\mathcal{P})) \longrightarrow \mathcal{D}^{DAG} \cup \{\bot\}$$

$$lag(p) \triangleq \begin{cases} lak(p) & \text{falls } lak(p) \neq \bot \wedge \\ & \quad kindof(lak(p)) = MAkteurGen \vee \\ & \quad kindof(lak(p)) = BenRep \vee \\ & \quad kindof(lak(p)) = KAkteurGen) \\ \gamma(lak(p)) & \text{falls } lak(p) \neq \bot \wedge \\ & \quad kindof(lak(p)) = NKAkteur \\ y & \text{falls } lak(p) \neq \bot \wedge \\ & \quad kindof(lak(p)) = KOrderGen \wedge y \in \mathcal{D}^{DAG} \wedge \\ & \quad (y, lak(p)) \in \delta \\ \bot & \text{sonst} \end{cases}$$

Wenn $lak(p)$ einen definierten Wert hat und das durch $lak(p)$ festgelegte Element des Aufrufpfades p keine K-Order–Generatorklasse ist, wird dem Pfad p durch die Abbildung lag die letzte in p enhaltene Akteur–Generatorklasse bzw. die Generatorklasse der letzten in p enthaltenen benannten K-Akteur–Inkarnationsklasse zugeordnet. Ist das mit $lak(p)$ festgelegte Element eine K-Order–Generatorklasse, ordnet lag dem Pfad p die K-Akteur-Generatorklasse zu, in deren Deklarationsteil diese K-Order–Generatorklasse enthalten ist. Gibt es in dem Pfad p weder eine Akteur– oder K-Order–Generatorklasse noch eine benannte K-Akteur–Inkarnationsklasse ist der Wert von $lag(p)$ undefiniert. Hat $lag(p)$ einen definierten Wert, so kann es zur Laufzeit von $\mathcal{P}$ einen Akteur geben, der Inkarnation bzgl. eines Generators des Generatorklasse $lag(p)$ ist und der einen Operationsaufruf bzgl. des letzten Elements v_n von p ausführt.

5.
$$lan : AP(\mathcal{G}(\mathcal{P})) \longrightarrow POT(\mathcal{D}^{NDAI})$$

$$lan(p) \triangleq \begin{cases} \{lak(p)\} & \text{falls } lak(p) \neq \bot \wedge \\ & \quad kindof(lak(p)) = NKAkteur \\ \{y \in \mathcal{D}^{NDAI} \mid (\gamma(y), lak(p)) \in \delta\} & \text{falls } lak(p) \neq \bot \wedge \\ & \quad kindof(lak(p)) = KOrderGen \\ \emptyset & \text{sonst} \end{cases}$$

Ist das durch $lak(p)$ festgelegte Element eines Aufrufpfades p eine benannte K-Akteur–Inkarnationsklasse, wird dem Pfad p durch die Abbildung lan die Menge zugeordnet, die aus dieser Inkarnationsklasse besteht. Ist das mit $lak(p)$ festgelegte Element eine K-Order–Generatorklasse, enthält die Menge $lan(p)$ alle benannten K-Akteur–Inkarnationsklassen, die sich auf die K-Akteur-Generatorklasse beziehen, in deren Deklarationsteil die K-Order–Generatorklasse enthalten ist. In allen anderen Fällen liefert $lan(p)$ die leere Menge als Ergebnis. Hat $lan(p)$ einen definierten Wert, so kann es zur Laufzeit von $\mathcal{P}$ einen benannten Akteur geben, der Element der Menge $lan(p)$ ist und der einen Operationsaufruf bzgl. des letzten Elements v_n von p ausführt. □

Der Zusammenhang zwischen den in Definition (5.10) festgelegten Pfad–Abbildungen und den oben angegebenen Elementen des Kontextattributs eines Akteurs läßt sich wie folgt zusammenfassen:

Ist $p = (v_0, v_1, \ldots, v_n)$ mit $lag(p) \neq \perp$ ein Pfad in dem statischen Aufrufgraph eines INSEL$^+$–Programms $\mathcal{P}$, so kann es zur Laufzeit von $\mathcal{P}$, d.h. in dem entsprechenden INSEL$^+$–Systems zu einem Zeitpunkt t einen Zustand geben, in dem ein Akteur a existiert,

- der Inkarnation bzgl. eines Generators der Generatorklasse $lag(p)$ ist,
- der die *erzeuge*–Operation auf einem Generator der Generatorklasse v_n aufruft, falls v_n Generatorklasse ist bzw. die *erzeuge*–Operation auf einem Generator der Generatorklasse $\gamma(v_n)$ aufruft und dabei eine benannte Inkarnation der Inkarnationsklasse v_n erzeugt, falls v_n benannte Inkarnationsklasse ist,
- für dessen Kontextattribut zum Zeitpunkt t gilt:

$$\begin{array}{lcll} ActGen_t(a) & = & lag(p) & \\ ActName_t(a) & \in & lan(p) & \text{falls } lan(p) \neq \emptyset \\ ActName_t(a) & = & \perp & \text{sonst} \\ ConGen_t(a) & = & lcg(p) & \\ ConName_t(a) & = & lcn(p) & \end{array}$$

Aufgrund des erklärten Zusammenhangs wird es möglich, die mit Gleichungen der Form

`Caller.ConGen` = *<inselp-path>*,
`Caller.ConName` = *<inselp-path>*,
`Caller.ActorGen` = *<inselp-path>*,
`Caller.ActorName` = *<inselp-path>*

in Zugriffsrestriktionsausdrücken formulierbaren Kontextrestriktionen (siehe hierzu Abschnitt 4.4.4.3) im Rahmen der statischen Konsistenzanalysen zu berücksichtigen. Die eingeführten Pfad–Abbildungen und der Zusammenhang zwischen diesen und den Kontextattributen der Akteure werden im folgenden exemplarisch anhand des statischen Aufrufgraphen des Kontenverwaltungssystems erklärt.

Beispiel

Gegeben sei der in Abbildung 5.6 dargestellte Ausschnitt aus dem attributierten statischen Aufrufgraph des INSEL$^+$–Programms, das das Kontenverwaltungssystem implementiert (vgl. auch Abbildung 5.5). Die gestrichelte Linie zwischen der benannten K-Akteur–Inkarnationsklasse `KontoVerwalter` und der K-Order–Generatorklasse `GibKontoZeiger` zeigt

an, daß `GibKontoZeiger` im Deklarationsteil von `KontoVerwalter` (bzw. genauer im Deklarationsteil der Generatorklasse `KontoVerwalterTyp`, auf die sich `KontoVerwalter` bezieht) definiert wird.

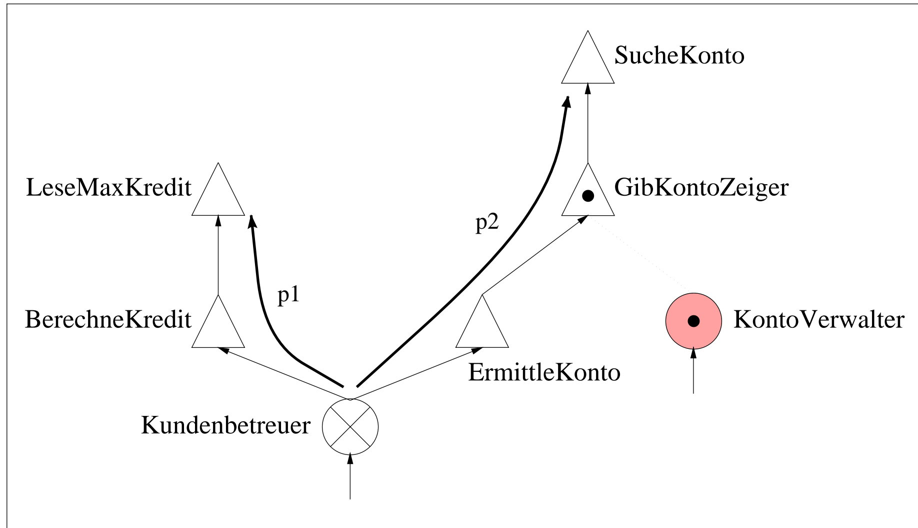

Abbildung 5.6: Ausschnitt aus dem statischen Aufrufgraph des Kontenverwaltungssystems

Zunächst wird der Pfad $p_1 = (\texttt{Kundenbetreuer}, \texttt{BerechneKredit}, \texttt{LeseMaxKredit})$ betrachtet. Für die Werte der Pfad–Abbildungen gilt:

$$\begin{aligned} lcg(p_1) &= \texttt{BerechneKredit} \\ lcn(p_1) &= \bot \\ lak(p_1) &= \texttt{Kundenbetreuer} \\ lag(p_1) &= \texttt{Kundenbetreuer} \\ lan(p_1) &= \emptyset \end{aligned}$$

Aus dem Pfad p_1 und den Werten der Pfad–Abbildungen für p_1 läßt sich folgende Aussage über das Verhalten des Kontenverwaltungssystems zur Laufzeit ableiten.

> Es kann zu einem Zeitpunkt t einen Zustand geben, in dem ein M-Akteur $a \in A_t$ existiert,
>
> – der Inkarnation bzgl. eines Generators der Benutzerrepräsentanten–Generatorklasse `Kundenbetreuer` ist,
>
> – der die kanonische Operation einer S-Order $s \in X_t^{DAI}$ ausführt, die Inkarnation bzgl. eines Generators der S-Order–Generatorklasse `BerechneKredit` ist, d.h. es gilt: $\varphi_t(a) = s$, und
>
> – der bei Ausführung von $op(s)$ zum Zeitpunkt t die *erzeuge*–Operation auf einem Generator der S-Order–Generatorklasse `LeseMaxKredit` aufruft.
>
> Für das Kontextattribut des Akteurs a zum Zeitpunkt t und damit für den Wert des für den *erzeuge*–Aufruf bzgl. `LeseMaxKredit` definierten Attributs `Caller` gilt:
>
> $$Con_t(a) = (User_t(a), Role_t(a)^{11}, \texttt{Kundenbetreuer}, \bot, \texttt{BerechneKredit}, \bot)$$

[11] Da a in diesem Fall ein Benutzerrepräsentant bzgl. der Rolle *Kundenbetreuer* ist, gilt: $Role_t(a) = uid(Kundenbetreuer)$.

Als zweites Beispiel wird der Aufrufpfad

$$p_2 = (\texttt{Kundenbetreuer}, \texttt{ErmittleKonto}, \texttt{GibKontoZeiger}, \texttt{SucheKonto})$$

betrachtet. Für die Werte der Pfad–Abbildungen gilt:

$$\begin{array}{lcl} lcg(p_2) & = & \texttt{GibKontoZeiger} \\ lcn(p_2) & = & \perp \\ lak(p_2) & = & \texttt{GibKontoZeiger} \\ lag(p_2) & = & \texttt{KontoVerwalterTyp} \\ lan(p_2) & = & \{\texttt{KontoVerwalter}\} \end{array}$$

Aus dem Pfad p_2 und den Werten der Pfad–Abbildungen läßt sich folgende Aussage über das Verhalten des Kontenverwaltungssystems zur Laufzeit ableiten.

> Es kann zu einem Zeitpunkt t einen Zustand geben,
>
> – in dem sich ein M-Akteur $a \in A_t$, der Inkarnation bzgl. eines Generators der Benutzerrepräsentanten–Generatorklasse `Kundenbetreuer` ist, und ein benannter K-Akteur $k \in A_t$, der der benannten K-Akteur–Inkarnationsklasse `KontoVerwalter` angehört, im Rendezvous bzgl. einer K-Order *ko* befinden, die Inkarnation bzgl. eines Generators der K-Akteur–Generatorklasse `GibKontoZeiger` ist, und
>
> – in dem der K-Akteur k die kanonische Operation der K-Order *ko* ausführt und dabei die *erzeuge*–Operation auf einem Generator der S-Order–Generatorklasse `SucheKonto` aufruft.
>
> Für das Kontextattribut des Akteurs k zum Zeitpunkt t und damit für den Wert des für den *erzeuge*–Aufruf bzgl. `SucheKonto` definierten Attributs `Caller` gilt:

$$Con_t(k) = (User_t(k), Role_t(k)^{12}, \texttt{KontoVerwalterTyp}, \texttt{KontoVerwalter}, \texttt{GibKontoZeiger}, \perp)$$

◇

Mit dem attributierten statischen Aufrufgraph eines INSEL$^+$–Programms und den für Aufrufpfade dieses Graphen definierten Pfad–Abbildungen steht nun eine geeignete Basis für die in den folgenden Abschnitten erklärten Konsistenzanalysen zur Verfügung.

5.3.2 Potentielle Konsistenz

In diesem Abschnitt werden die Kriterien und Analysen angegeben, die es ermöglichen, ausgehend von einem INSEL$^+$–Programm eine Aussage darüber zu treffen, ob das Recht des durch dieses Programm definierten INSEL$^+$–Systems absolut inkonsistent oder potentiell konsistent ist. Gemäß Abschnitt 5.2 ist das Recht eines INSEL$^+$–Systems absolut inkonsistent, wenn es mindestens eine statisch feststellbare Inkonsistenz enthält. Es ist potentiell konsistent, wenn es keine statisch feststellbaren Inkonsistenzen enthält. Im weiteren geht es also darum, zu analysieren, ob das Recht eines INSEL$^+$–Systems Inkonsistenzen enthält, die bereits zur Übersetzungszeit des entsprechenden INSEL$^+$–Programms ermittelbar sind.

Dazu seien k und x zwei zugriffskontrollierte Komponenten eines INSEL$^+$–Systems, *op* sei eine äußere Operation von k und op' eine äußere Operation von x. R^k_{op} und $R^x_{op'}$ seien die für

[12] Da a ein Benutzerrepräsentant bzgl. der Rolle *Kundenbetreuer* ist, gilt: $Role_t(k) = role(ko) = role(creator(ko)) = role(a) = uid(Kundenbetreuer)$.

op und *op'* definierten Zugriffsrestriktionsausdrücke. Weiterhin gelte, daß im Kontext einer *op*–Ausführung die Operation *op'* auf x aufgerufen wird, also: $op' \in CalledOps(k, op)$. Wie in Abschnitt 5.2 angegeben, liegt eine Inkonsistenz in dem Recht des INSEL$^+$–Systems genau dann vor, wenn ein Akteur a, in dessen Ausführungsumgebung k liegt, zu einem Zeitpunkt t das Recht zur Ausführung von *op* auf k hat, a dieses Recht in t wahrnimmt, und der für *op'* definierte Zugriffsrestriktionsausdruck $R^x_{op'}$ für den Akteur a', der *op'* im Kontext der durch a initiierten *op*–Ausführung aufruft[13], in t' nicht erfüllt ist, wobei $t' > t$ der Zeitpunkt ist, zu dem *op'* durch a' aufgerufen wird. Wie leicht nachzuvollziehen ist, liegt eine Inkonsistenz auf jeden Fall dann vor, wenn

1. der Zugriffsrestriktionsausdruck $R^x_{op'}$ unerfüllbar ist, d.h. wenn er in jedem möglichen Zustand des INSEL$^+$–Systems den Wert *false* hat;

2. die Zugriffsrestriktionsausdrücke R^k_{op} und $R^x_{op'}$ widersprüchlich sind, d.h. wenn – unter Berücksichtigung der in R^k_{op} und $R^x_{op'}$ spezifizierten Kontextrestriktionen – aus der Gültigkeit von R^k_{op} immer die Ungültigkeit von $R^x_{op'}$ folgt.

Die Begriffe der Unerfüllbarkeit eines Zugriffsrestriktionsausdrucks und der Widersprüchlichkeit zweier Zugriffsrestriktionsausdücke werden in den folgenden beiden Definitionen präzisiert.

Definition 5.11.: Unerfüllbarkeit eines Zugriffsrestriktionsausdrucks

Seien $\mathcal{S}$ ein INSEL$^+$–System zu einem Zeitpunkt $t \in \Lambda(\mathcal{S})$, $x \in X^{ZK}_t$ eine zugriffskontrollierte Komponente von $\mathcal{S}$ und $op' \in O^E(x)$ eine äußere Operation von x. $R^x_{op'}$ sei der für *op'* definierte Zugriffsrestriktionsausdruck.

$R^x_{op'}$ ist **unerfüllbar** $\overset{\triangle}{\Longleftrightarrow}$ Für alle $t' \in \Lambda(x)$ und $a \in A_{t'}$ gilt:

$$\omega(R^x_{op'}, a, t') = false$$

$R^x_{op'}$ ist **erfüllbar** $\overset{\triangle}{\Longleftrightarrow}$ $R^x_{op'}$ ist nicht unerfüllbar

□

Gemäß Definition (5.11) ist der Zugriffsrestriktionsausdruck einer äußeren Operation einer zugriffskontrollierten Komponente also genau dann unerfüllbar, wenn sein Wert für alle Zustände *false* ist, d.h. wenn die durch ihn definierte prädikatenlogische Aussage eine Kontradiktion ist. Er ist erfüllbar, wenn es mindestens einen Zustand gibt, in dem er den Wert *true* hat. Es lassen sich statisch überprüfbare Kriterien für Zugriffsrestriktionsausdücke angeben, die hinreichende Bedingungen für die Erfüllbarkeit bzw. Unerfüllbarkeit eines Zugriffsrestriktionsausdrucks formulieren. Diese Kriterien werden in dem folgenden Abschnitt 5.3.2.1 erklärt. Jeder in einem INSEL$^+$–Programm auftretende Zugriffsrestriktionsausdruck wird anhand dieser Kriterien auf Erfüllbarkeit überprüft. Wird im Rahmen dieser Analyse festgestellt, daß ein Zugriffsrestriktionsausdruck, der für eine DA–Generatorklasse G definiert ist, unerfüllbar ist, so bedeutet dies, daß zur Laufzeit des Programms nie ein Akteur das Recht zur Erzeugung einer Inkarnation bzgl. eines Generators der Generatorklasse G hat. Eine solche Rechtefestlegung macht jedoch keinen Sinn, da die durch G definierte Funktionalität für Berechnungen des entsprechenden INSEL$^+$–Systems nicht eingesetzt werden kann

[13] Es kann gelten: $a = a'$.

und damit überflüssig ist. Aus diesem Grund wird gefordert, daß alle Zugriffsrestriktionsausdrücke, die in einem INSEL$^+$–Programm auftreten, erfüllbar sein müssen. Entsprechende Überprüfungen werden vom INSEL$^+$–Übersetzer im Rahmen der semantischen Analyse von Zugriffsrestriktionsausdrücken durchgeführt.

Definition 5.12.: Widersprüchlichkeit zweier Zugriffsrestriktionsausdrücke

Seien $\mathcal{S}$, x, op' und $R^x_{op'}$ wie in Definition (5.11) gegeben. Weiter seien $k \in X^{ZK}_t$ eine zugriffskontrollierte Komponente, $op \in O^E(k)$ eine äußere Operation von k und R^k_{op} der für op definierte Zugriffsrestriktionsausdruck. Es gelte: $op' \in CalledOps(k, op)$.

R^k_{op} und $R^x_{op'}$ sind **widersprüchlich** $\overset{\triangle}{\Longleftrightarrow}$ Für alle $t' \in \Lambda(k)$ und $a \in A_{t'}$ gilt:

$$\omega(R^k_{op}, a, t') = true \implies \omega(R^x_{op'}, a', t'') = false$$

für alle Akteure a' und alle Zeitpunkte $t'' > t'$ mit:
a' ruft zum Zeitpunkt t'' im Kontext der durch
a in t' initiierten op–Ausführung op' auf x auf

R^k_{op} und $R^x_{op'}$ sind **widerspruchsfrei** $\overset{\triangle}{\Longleftrightarrow}$ R^k_{op} und $R^x_{op'}$ sind nicht widersprüchlich

□

Die Zugriffsrestiktionsausdrücke R^k_{op} und $R^x_{op'}$ zweier Operationen op und op', für die gilt, daß op' im Kontext der Ausführung von op aufgerufen wird, sind laut Definition (5.12) genau dann widersprüchlich, wenn unter Berücksichtigung der Kontextattribute der Akteure, die die Operationen aufrufen, aus der Gültigkeit von R^k_{op} immer die Ungültigkeit von $R^x_{op'}$ folgt. R^k_{op} und $R^x_{op'}$ sind demgegenüber widerspruchsfrei, wenn es – grob gesprochen – Akteure a und a' sowie Zeitpunkte t' und t'' geben kann, so daß R^k_{op} zum Zeitpunkt t' für den Akteur a und $R^x_{op'}$ zum Zeitpunkt t'' für den Akteur a' erfüllt ist. Die Akteure und damit ihre Kontextattribute müssen in der Definition mit erfaßt werden, um die in Zugriffsrestriktionsausdrücken festlegbaren Kontextrestriktionen geeignet zu berücksichtigen. Die Widersprüchlichkeit bzw. Widerspruchsfreiheit zweier Zugriffsrestiktionsausdrücke läßt sich im allgemeinen nicht statisch nachweisen (vgl. dazu auch Abschnitt 5.2). Es lassen sich jedoch hinreichende, statisch überprüfbare Bedingungen dafür angeben, daß zwei Zugriffsrestriktionsausdrücke widersprüchlich bzw. widerspruchsfrei sind. Diese Bedingungen werden in Abschnitt 5.3.2.2 erläutert. Bevor die Analyse auf Widersprüchlichkeit zweier Zugriffsrestriktionsausdrücke anhand dieser Kriterien durchgeführt werden kann, müssen zunächst die Paare von Zugriffsrestriktionsausdrücken ermittelt werden, für die überhaupt eine solche Analyse durchzuführen ist. Dazu müssen Paare von zugriffskontrollierten DA-Generatorklassen (G_1, G_2) ermittelt werden, für die gilt, daß im Kontext der Ausführung einer kanonischen Operation einer Inkarnation bzgl. eines Generators der Generatorklasse G_1 potentiell eine Inkarnation bzgl. eines Generators der Generatorklasse G_2 erzeugt wird. Diese Paare werden von dem INSEL$^+$–Übersetzer anhand des attributierten statischen Aufrufgraphen eines INSEL$^+$–Programms ermittelt (siehe dazu Abschnitt 5.3.2.2). Die Zugriffsrestriktionsausdrücke der so ermittelten Paare von zugriffskontrollierten DA-Generatorklassen werden dann auf Widerspruchsfreiheit überprüft. Wird im Rahmen dieser Analysen festgestellt, daß ein Paar zu überprüfender Zugriffsrestriktionsausdrücke widersprüchlich ist, so besagt dies, daß das Recht des durch das analysierte Programm definierten INSEL$^+$–Systems absolut inkonsistent ist. Dementsprechend hat der Systementwickler vom INSEL$^+$–Übersetzer erkannte Widersprüche zu beseitigen, indem er die die Widersprüche verursachenden Zugriffsrestriktionsausdrücke geeignet modifiziert. Das Recht eines INSEL$^+$–Systems, das durch ein INSEL$^+$–Programm definiert wird, für das gilt, daß alle zu überprüfenden Paare von Zugriffsrestriktionsausdrücken widerspruchsfrei sind und das somit keine statisch feststellbaren Inkonsistenzen enthält, ist potentiell konsistent.

5.3.2.1 Kriterien zur Analyse der Erfüllbarkeit eines Zugriffsrestriktionsausdrucks

In diesem Abschnitt werden die Kriterien angegeben, anhand derer der INSEL$^+$–Übersetzer die Erfüllbarkeit eines Zugriffsrestriktionsausdrucks überprüft. Für diese Überprüfung werden die Zugriffsrestriktionsausdrücke in einer einheitlichen Form dargestellt. Da ein Zugriffsrestriktionsausdruck ein Boolescher Ausdruck ist, bietet es sich an, als einheitliche Darstellungsform für Zugriffsrestriktionsausdrücke die Disjunktive Normalform zu wählen. Die Möglichkeiten für die Definition von Zugriffsrestriktionsausdrücken wurden in Abschnitt 4.4.4 erklärt. In einem Zugriffsrestriktionsausdruck können neben den Zugriffsrestriktionsprädikaten `IN_ACL` und `ACCESSED` insbesondere alle der in INSEL$^+$ auf den vordefinierten elementaren Datentypen definierten Vergleichsoperatoren auftreten; dies sind die Operatoren '=', '/=', '<', '>', '<=', '>='. Diese Vergleichsoperatoren werden im weiteren als zweistellige Prädikate aufgefaßt und als Vergleichsprädikate bezeichnet. Um in einer Disjunktiven Normalform möglichst wenig unterschiedliche Prädikate zu haben, wird ein Zugriffsrestriktionsausdruck R^k_{op} vor Bildung der Disjunktiven Normalform in einen äquivalenten Zugriffsrestriktionsausdruck $N(R^k_{op})$ transformiert, der als Vergleichsprädikate lediglich '=' und '<' enthält, und als **normalisierter Zugriffsrestriktionsausdruck** bezeichnet wird. Für die Transformation eines Zugriffsrestriktionsausdrucks in einen äquivalenten normalisierten Zugriffsrestriktionsausdruck gelten die bekannten Regeln, nach denen sich die Prädikate '/=', '>', '<=' und '>=' aus den Prädikaten '=' und '<' sowie den logischen Operatoren `AND`, `OR` und `NOT` ergeben[14].

(5.13) Seien a und b zwei Terme[15]. Es gelten die folgenden Regeln:

$$\begin{array}{lll} (1.) & a\,/=b & \equiv \neg(a=b) \\ (2.) & a>b & \equiv \neg(a=b)\wedge\neg(a<b) \\ (3.) & a>=b & \equiv \neg(a<b) \\ (4.) & a<=b & \equiv (a=b)\vee(a<b) \end{array}$$

Ein normalisierter Zugriffsrestriktionsausdruck $N(R^k_{op})$ kann nun wie für Boolesche Ausdrücke üblich in Disjunktive Normalform umgewandelt werden.

Definition 5.14.: Disjunktive Normalform eines Zugriffsrestriktionsausdrucks

Sei R^k_{op} ein Zugriffsrestriktionsausdruck, der gemäß der in (4.8) angegebenen Syntax konstruiert ist.
Die **Disjunktive Normalform** $DNF(R^k_{op})$ von R^k_{op} ist wie folgt definiert:

$$DNF(R^k_{op}) \triangleq DNF(N(R^k_{op})) = \bigvee_{i=1}^{m} \bigwedge_{j=1}^{n_i} L_{ij}$$

wobei $N(R^k_{op})$ der zu R^k_{op} äquivalente normalisierte Zugriffsrestriktionsausdruck ist, der als Vergleichsprädikate lediglich '=' und '<' enthält.

Die L_{ij} in der Disjunktiven Normalform $DNF(R^k_{op})$ werden als **Literale** bezeichnet.

□

Gemäß der in Abschnitt 4.4.4 erklärten Konzepte zur Konstruktion von Zugriffsrestriktionsausdrücken können in der Disjunktiven Normalform eines Zugriffsrestriktionsausdrucks folgende Literale auftreten:

[14]Für die logischen Operatoren `AND`, `OR` und `NOT` gelten die in (5.1) festgelegten Notationen $\wedge$, $\vee$ und $\neg$.

[15]Auf eine formale Definition von Termen wird hier verzichtet. Sie werden auf Grundlage der in (4.8) angegebenen Syntax für Zugriffsrestriktionsausdrücke wie üblich gebildet.

(5.15) Literale in Zugriffsrestriktionsausdrücken

(a) die Zugriffsrestriktionsprädikate `IN_ACL` und `ACCESSED` sowie ihre Negationen $\neg$`IN_ACL` und $\neg$`ACCESSED`;

(b) Gleichungen, in denen auf der linken Seite der Bezeichner eines Elements des `Caller`–Attributs steht, also Gleichungen der folgenden Art und ihre jeweiligen Negationen:

- Benutzerrestriktionen: `Caller.User` = <*user-identifier*> und $\neg$(`Caller.User` = <*user-identifier*>);
- Rollenrestriktonen: `Caller.Role` = <*role-identifier*> und $\neg$(`Caller.Role` = <*role-identifier*>);
- Kontextrestriktionen: `Caller.ActorGen` = <*inselp-path*> und $\neg$(`Caller.ActorGen` = <*inselp-path*>);

  `Caller.ActorName` = <*inselp-path*> und $\neg$(`Caller.ActorName` = <*inselp-path*>);

  `Caller.ConGen` = <*inselp-path*> und $\neg$(`Caller.ConGen` = <*inselp-path*>);

  `Caller.ConName` = <*inselp-path*> und $\neg$(`Caller.ConGen` = <*inselp-path*>)

(c) Wertvergleiche auf DE–Inkarnationen bzgl. der vordefinierten elementaren Datentypen und der aus diesen abgeleiteten Bereichsgeneratoren, also Gleichungen bzw. Ungleichungen der folgenden Art und ihre jeweiligen Negationen:

- $\mathcal{A} = \mathcal{B}$ und $\neg(\mathcal{A} = \mathcal{B})$,
- $\mathcal{A} < \mathcal{B}$ und $\neg(\mathcal{A} < \mathcal{B})$,

wobei $\mathcal{A}$ und $\mathcal{B}$ Bezeichner bzw. Bezeichnerfolgen sind, die jeweils eine DE–Inkarnation identifizieren.

Die Zugriffsrestriktionsprädikate `IN_ACL` und `ACCESSED` sowie die Vergleichsprädikate '=' und '<' werden im weiteren als **Basisprädikate** bezeichnet. Ein Literal ist also entweder ein Basisprädikat oder die Negation eines Basisprädikats. In der folgenden Definition werden die Begriffe der Widersprüchlichkeit und der Widerspruchsfreiheit für zwei Literale eingeführt.

Definition 5.16.: Widersprüchlichkeit von Literalen

Seien $\mathcal{S}$ ein INSEL$^+$–System, k eine zugriffskontrollierte Komponente von $\mathcal{S}$ und $op \in O^E(k)$ eine äußere Operation von k. R^k_{op} sei der für op definierte Zugriffsrestriktionsausdruck und

$$DNF(R^k_{op}) = \bigvee_{i=1}^{m} \bigwedge_{j=1}^{n_i} L_{ij}$$

sei die Disjunktive Normalform von R^k_{op}.

Zwei **Literale** $L_{i_1j_1}$ und $L_{i_1j_2}$ eines konjunktiven Teilterms $\bigwedge_{j=1}^{n_{i_1}} L_{i_1j}$ mit $i_1 \in \{1,\ldots,m\}$ und $j_1, j_2 \in \{1,\ldots,n_{i_1}\}$ sind **widersprüchlich** $\overset{\triangle}{\Longleftrightarrow}$ Für alle $t \in \Lambda(k)$ und alle Akteure $a \in A_t$ gilt:[16]

$$\omega(L_{i_1j_1} \wedge L_{i_1j_2}, a, t) = false$$

[16] $\omega(L_{i_1j_1} \wedge L_{i_1j_2}, a, t)$ bezeichnet analog zu der in (4.7) für Zugriffsrestriktionsausdrücke eingeführten Bezeichnung den Wert von $L_{i_1j_1} \wedge L_{i_1j_2}$ zum Zeitpunkt t für den Akteur a.

Zwei **Literale** $L_{i_1j_1}$ und $L_{i_1j_2}$ sind **widerspuchsfrei** $\stackrel{\triangle}{\Longleftrightarrow}$ $L_{i_1j_1}$ und $L_{i_1j_2}$ sind nicht widersprüchlich.

Ein **konjunktiver Teilterm** $\bigwedge_{j=1}^{n_{i_1}} L_{i_1j}$ mit $i_1 \in \{1,\ldots,m\}$ ist **unerfüllbar** $\stackrel{\triangle}{\Longleftrightarrow}$ Es gibt $j_1, j_2 \in \{1,\ldots,n_{i_1}\}$, so daß gilt: $L_{i_1j_1}$ und $L_{i_1j_2}$ sind widersprüchlich.

Ein **konjunktiver Teilterm** ist **erfüllbar** $\stackrel{\triangle}{\Longleftrightarrow}$ Er ist nicht unerfüllbar.

□

Zwei Literale sind also genau dann widersprüchlich, wenn ihre Konjunktion immer den Wert *false* hat, also eine Kontradiktion ist. Ein konjunktiver Teilterm der Disjunktiven Normalform eines Zugriffsrestriktionsausdrucks ist unerfüllbar, wenn er mindestens ein Paar widersprüchlicher Literale enthält; in diesem Fall ist der Wert des konjunktiven Teilterms ebenfalls in jedem Zustand *false*. Der folgende Satz formuliert aufbauend auf Definition (5.16) eine notwendige und hinreichende Bedingung dafür, daß ein Zugriffsrestriktionsausdruck unerfüllbar ist. Da ein Zugriffsrestriktionsausdruck genau dann erfüllbar ist, wenn er nicht unerfüllbar ist, liefert dieser Satz gleichzeitig eine notwendige und hinreichende Bedingung dafür, wann ein Zugriffsrestriktionsausdruck erfüllbar ist.

Satz 5.1.: Unerfüllbarkeit eines Zugriffsrestriktionsausdrucks

Seien $\mathcal{S}$, k, *op*, R_{op}^k und $DNF(R_{op}^k)$ wie in Definition (5.16) gegeben. Dann gilt:

R_{op}^k ist unerfüllbar $\Longleftrightarrow$ Für alle $i \in \{1,\ldots,m\}$ gilt: $\bigwedge_{j=1}^{n_i} L_{ij}$ ist unerfüllbar

△

Ein Zugriffsrestriktionsausdruck ist also genau dann unerfüllbar, wenn alle konjunktiven Teilterme seiner Disjunktiven Normalform unerfüllbar sind.

Beweis:
Der Beweis basiert auf der Äquivalenz von R_{op}^k und $DNF(R_{op}^k)$.
"$\Longrightarrow$": Beweis durch Widerspruch

Voraussetzung: R_{op}^k ist unerfüllbar; d.h. es gilt für alle $t \in \Lambda(k)$ und $a \in A_t$:

$$(*) \quad \omega(R_{op}^k, a, t) = false.$$

Annahme: es gibt ein $i_1 \in \{1,\ldots,m\}$, so daß $\bigwedge_{j=1}^{n_{i_1}} L_{i_1j}$ erfüllbar ist. Dann gibt es ein $t_1 \in \Lambda(k)$ und einen Akteur $a \in A_{t_1}$, so daß $\omega(\bigwedge_{j=1}^{n_{i_1}} L_{i_1j}, a, t_1) = true$ und damit $\omega(DNF(R_{op}^k), a, t_1) = \omega(R_{op}^k, a, t_1) = true$ gilt. Dies ist jedoch ein Widerspruch zu $(*)$.

"$\Longleftarrow$": Beweis durch Widerspruch

Voraussetzung: Für alle $i \in \{1,\ldots,m\}$, alle $t \in \Lambda(k)$ und alle $a \in A_t$ gilt: $\omega(\bigwedge_{j=1}^{n_i} L_{ij}, a, t) = false$. Daraus folgt:

$$(**) \quad \omega(DNF(R_{op}^k), a, t) = \omega(R_{op}^k, a, t) = false$$

Annahme: R_{op}^k ist nicht widersprüchlich. Dann gibt es ein $t_1 \in \Lambda(k)$ und einen Akteur $a \in A_{t_1}$, so daß $\omega(R_{op}^k, a, t_1) = true$ gilt, was ein Widerspruch zu $(**)$ ist.

▽

Im weiteren dieses Abschnitts werden hinreichende, statisch überprüfbare Bedingungen für die Widersprüchlichkeit zweier Literale erarbeitet. Auf Basis dieser Bedingungen und des Satzes (5.1) kann die Erfüllbarkeit eines Zugriffsrestriktionsausdrucks R_{op}^k vom INSEL$^+$–Übersetzer wie folgt überprüft werden: zunächst wird die Disjunktive Normalform zu R_{op}^k gebildet;

anschließend werden die konjunktiven Teilterme der Disjunktiven Normalform nacheinander auf Unerfüllbarkeit überprüft, indem anhand der noch anzugebenen Bedingungen untersucht wird, ob sie mindestens ein Paar widersprüchlicher Literale enthalten. Wird ein konjunktiver Teilterm gefunden, der nicht unerfüllbar ist, d.h. in dem kein Paar widersprüchlicher Literale enthalten ist, ist R^k_{op} erfüllbar, und die Analyse von R^k_{op} kann abgebrochen werden. Sind alle konjunktiven Teilterme der Disjunktiven Normalform unerfüllbar, so ist nach Satz (5.1) auch R^k_{op} unerfüllbar.

Sind L_1 und L_2 zwei Literale, die in einem konjunktiven Teilterm der Disjunktiven Normalform eines in einem INSEL$^+$–Programm auftretenden Zugriffsrestriktionsausdrucks enthalten sind, so sind im weiteren also Kriterien von Interesse, anhand derer bereits zur Übersetzungszeit nachweisbar ist, daß die Konjunktion von L_1 und L_2 für alle Zustände zur Laufzeit des Programms den Wert *false* hat. Voraussetzung dafür ist, daß der INSEL$^+$–Übersetzer erkennen kann, ob zwei Basisprädikate der gleichen Art[17] „äquivalent“ sind. Dabei ist zwischen syntaktischer und semantischer Äquivalenz zu unterscheiden. Zwei Basisprädikate der gleichen Art sind **syntaktisch äquivalent**, wenn sie textuell gleich sind, d.h. die in ihnen auftretenden Bezeichner bzw. Bezeichnerfolgen identisch sind[18]. Sie sind **semantisch äquivalent**, wenn sie für alle Zustände stets den gleichen Wert haben. Treten die beiden Basisprädikate in einem Zugriffsrestriktionsausdruck auf, so folgt aus der syntaktischen stets die semantische Äquivalenz; die Umkehrung dieser Aussage gilt jedoch nicht. Die semantische Äquivalenz zweier Basisprädikate läßt sich im allgemeinen nicht statisch nachweisen. Deshalb prüft der INSEL$^+$–Übersetzer bei der Analyse der Widersprüchlichkeit zweier Literale in der Regel auf syntaktische Äquivalenz, d.h. auf Gleichheit bzw. Ungleichheit der in den jeweiligen Basisprädikaten auftretenden Bezeichnerfolgen.

Definition 5.17.: Gleichheit von Bezeichnerfolgen

Seien $\mathcal{F} \triangleq F_1.F_2. \;\ldots\; .F_i$ und $\mathcal{G} \triangleq G_1.G_2. \;\ldots\; .G_j$ mit $i, j \geq 1$ zwei Bezeichnerfolgen.

$\mathcal{F}$ und $\mathcal{G}$ sind gleich – $\mathcal{F} =_N \mathcal{G}$ – $\overset{\triangle}{\Longleftrightarrow}$ es gilt: $i = j$ und $F_l =_s G_l$ für alle $l \in \{1, \ldots, i\}$ mit $=_s$ ist die Gleichheit von Zeichenketten.

$\mathcal{F}$ und $\mathcal{G}$ sind ungleich – $\mathcal{F} \neq_N \mathcal{G}$ – $\overset{\triangle}{\Longleftrightarrow}$ $\mathcal{F}$ und $\mathcal{G}$ sind nicht gleich.

□

In dem folgenden Satz werden nun die Bedingungen angegeben, anhand derer der INSEL$^+$–Übersetzer überprüft, ob zwei Literale widersprüchlich sind. Zu jeder Bedingung wird informell begründet, warum bei Erfüllung der Bedingung die Konjunktion der beiden Literale stets den Wert *false* hat.

Satz 5.2.: Kriterien für die Widersprüchlichkeit zweier Literale eines konjunktiven Teilterms

Seien L_1 und L_2 zwei Literale gemäß (5.15), die in einem konjunktiven Teilterm der Disjunktiven Normalform eines in einem INSEL$^+$–Programm für eine äußere Operation festgelegten Zugriffsrestriktionsausdrucks enthalten sind. `GenName` sei der Name der zugriffskontrollierten DA–Generatorklasse, in deren Zugriffsrestriktions–Part der Zugriffsrestriktionsausdruck festgelegt ist und `OpName` sei der Name der Operation, für die der Zugriffsrestriktionsausdruck festgelegt ist.

[17] Also zwei `IN_ACL`–, zwei `ACCESSED`–, zwei '='– oder zwei '<'–Prädikate.

[18] Eine in einem Basisprädikat enthaltene Bezeichnerfolge $\mathcal{F} = F_1.F_2. \;\ldots\; .F_i$, $i \geq 1$, ist entweder ein INSEL$^+$–Pfad (vgl. Definition (4.13)) oder ein Name, der einen DA–Generator, eine benannte DA–Inkarnation oder eine DE–Inkarnation identifiziert.

Die Literale L_1 und L_2 sind **widersprüchlich**, wenn eine der folgenden Bedingungen gilt:

1. L_1 ist `IN_ACL(UserId1, RoleId1, CompId1, OpId1)` und
   L_2 ist ¬`(IN_ACL(UserId2, RoleId2, CompId2, OpId2))` und es gilt:
   (a) `UserId1` $=_N$ `UserId2` und `RoleId1` $=_N$ `RoleId2` und `CompId1` $=_N$ `CompId2` und `OpId1` $=_N$ `OpId2` oder
   (b) `UserId1` $=_N$ `UserId2` und `RoleId1` $=_N$ `RoleId2` und `CompId1` $=_N$ `THIS` und `OpId1` $=_N$ `THIS` und `CompId2` $=_N$ `THIS` und `OpId2` $=_N$ `OpName`

   Begründung:
   Gilt (a) oder (b), so sind die beiden `IN_ACL`–Prädikate semantisch äquivalent (siehe hierzu Definition (4.9)). Da L_2 die Negation von L_1 ist, gilt also: $L_1 \wedge L_2 \equiv L_1 \wedge \neg L_1 \equiv false$.
2. L_1 ist `ACCESSED(UserId1, CompId1, OpId1)` und
   L_2 ist ¬`(ACCESSED(UserId2, CompId2, OpId2))` und es gilt:
   (a) `UserId1` $=_N$ `UserId2` und `CompId1` $=_N$ `CompId2` und `OpId1` $=_N$ `OpId2` oder
   (b) `UserId1` $=_N$ `UserId2` und `CompId1` $=_N$ `CompId2` und `OpId2` $=_N$ `ANY`

   Begründung:
   Gilt (a), so sind die beiden `ACCESSED`–Prädikate semantisch äquivalent, d.h. L_2 ist Negation von L_1 und damit gilt: $L_1 \wedge L_2 \equiv L_1 \wedge \neg L_1 \equiv false$. Gilt (b), so folgt $L_1 \wedge L_2 \equiv false$ aus der in Definition (4.10) angegebenen Semantik des `ACCESSED`–Prädikats.
3. L_1 ist `Caller.User = UserId1` und L_2 ist `Caller.User = UserId2` und es gilt: `UserId1` $\neq_N$ `UserId2`

   Begründung:
   `Caller.User` gibt bei einem Aufruf von `OpName` den Identifikator des Benutzers an, der der Ausführungskomponente des aufrufenden Akteurs zugeordnet ist. Da `UserId1` und `UserId2` zwei unterschiedliche Benutzer identifizieren, müßte für die Erfüllung von $L_1 \wedge L_2$ gelten, daß einer DA–Inkarnation unterschiedliche Benutzer zugeordnet sein können. Da jeder DA–Inkarnation gemäß Definition (4.1) für ihre Lebenszeit jedoch genau ein Benutzer zugeordnet ist, folgt unmittelbar: $L_1 \wedge L_2 \equiv false$.
4. L_1 ist `Caller.User = UserId1` und L_2 ist ¬`(Caller.User = UserId2)` und es gilt: `UserId1` $=_N$ `UserId2`

   Begründung:
   L_2 ist in diesem Fall Negation von L_1, es gilt also: $L_1 \wedge L_2 \equiv L_1 \wedge \neg L_1 \equiv false$.
5. L_1 ist `Caller.Role = RoleId1` und L_2 ist `Caller.Role = RoleId2` und es gilt: `RoleId1` $\neq_N$ `RoleId2`

   Begründung:
   `Caller.Role` gibt bei einem Aufruf von `OpName` den Identifikator der Rolle an, die der Ausführungskomponente des aufrufenden Akteurs zugeordnet ist. Da `RoleId1` und `RoleId2` zwei unterschiedliche Rollen identifizieren, müßte für die Erfüllung von $L_1 \wedge L_2$ gelten, daß einer DA–Inkarnation unterschiedliche Rollen zugeordnet sein können. Da jeder DA–Inkarnation gemäß Definition (4.3) für ihre Lebenszeit jedoch genau eine Rolle zugeordnet ist, folgt unmittelbar: $L_1 \wedge L_2 \equiv false$.
6. L_1 ist `Caller.Role = RoleId1` und L_2 ist ¬`(Caller.Role = RoleId2)` und es gilt: `RoleId1` $=_N$ `RoleId2`

Begründung:
L_2 ist in diesem Fall Negation von L_1, es gilt also: $L_1 \wedge L_2 \equiv L_1 \wedge \neg L_1 \equiv false$.

7. L_1 ist `Caller.ActorGen =` $\mathcal{F}$ und L_2 ist `Caller.ActorGen =` $\mathcal{G}$, wobei $\mathcal{F}$ und $\mathcal{G}$ INSEL$^+$–Pfade sind, und es gilt: $\mathcal{F} \neq_N \mathcal{G}$

   Begründung:
   `Caller.ActorGen` gibt bei einem Aufruf von `OpName` den Namen des Generators des aufrufenden Akteurs an. Da die mit $\mathcal{F}$ und $\mathcal{G}$ identifizierten Akteur–Generatorklassen unterschiedlich sind, müßte für die Erfüllung von $L_1 \wedge L_2$ gelten, daß ein Akteur Inkarnation bzgl. mehrerer Generatoren sein kann. Da jede DA–Inkarnation jedoch Inkarnation bzgl. genau eines DA–Generators ist, folgt unmittelbar: $L_1 \wedge L_2 \equiv false$.

8. L_1 ist `Caller.ActorGen =` $\mathcal{F}$ und L_2 ist $\neg$`(Caller.ActorGen =` $\mathcal{G}$`)`, wobei $\mathcal{F}$ und $\mathcal{G}$ INSEL$^+$–Pfade sind, und es gilt: $\mathcal{F} =_N \mathcal{G}$

   Begründung:
   L_2 ist in diesem Fall Negation von L_1, es gilt also: $L_1 \wedge L_2 \equiv L_1 \wedge \neg L_1 \equiv false$.

9. L_1 ist `Caller.ActorName =` $\mathcal{F}$ und L_2 ist `Caller.ActorName =` $\mathcal{G}$, wobei $\mathcal{F}$ und $\mathcal{G}$ INSEL$^+$–Pfade sind, und es gilt: $\mathcal{F} \neq_N \mathcal{G}$

   Begründung:
   `Caller.ActorName` gibt bei einem Aufruf von `OpName` den Namen des aufrufenden Akteurs an, falls dieser benannter K-Akteur ist. Da die mit $\mathcal{F}$ und $\mathcal{G}$ identifizierten benannten K-Akteur–Inkarnationsklassen verschieden sind, folgt unmittelbar: $L_1 \wedge L_2 \equiv false$.

10. L_1 ist `Caller.ActorName =` $\mathcal{F}$ und L_2 ist $\neg$`(Caller.ActorName =` $\mathcal{G}$`)`, wobei $\mathcal{F}$ und $\mathcal{G}$ INSEL$^+$–Pfade sind, und es gilt: $\mathcal{F} =_N \mathcal{G}$

    Begründung:
    L_2 ist in diesem Fall Negation von L_1, es gilt also: $L_1 \wedge L_2 \equiv L_1 \wedge \neg L_1 \equiv false$.

11. L_1 ist `Caller.ConGen =` $\mathcal{F}$ und L_2 ist `Caller.ConGen =` $\mathcal{G}$, wobei $\mathcal{F}$ und $\mathcal{G}$ INSEL$^+$–Pfade sind, und es gilt: $\mathcal{F} \neq_N \mathcal{G}$

    Begründung:
    `Caller.ConGen` gibt bei einem Aufruf von `OpName` den Namen des Generators der Ausführungskomponente des aufrufenden Akteurs an. Jedem Akteur ist zu jedem Zeitpunkt seiner Existenz genau eine Ausführungskomponente zugeordnet. Da die mit $\mathcal{F}$ und $\mathcal{G}$ identifizierten DA–Generatorklassen unterschiedlich sind, folgt unmittelbar: $L_1 \wedge L_2 \equiv false$.

12. L_1 ist `Caller.ConGen =` $\mathcal{F}$ und L_2 ist $\neg$`(Caller.ConGen =` $\mathcal{G}$`)`, wobei $\mathcal{F}$ und $\mathcal{G}$ INSEL$^+$–Pfade sind, und es gilt: $\mathcal{F} =_N \mathcal{G}$

    Begründung:
    L_2 ist in diesem Fall Negation von L_1, es gilt also: $L_1 \wedge L_2 \equiv L_1 \wedge \neg L_1 \equiv false$.

13. L_1 ist `Caller.ConName =` $\mathcal{F}$ und L_2 ist `Caller.ConName =` $\mathcal{G}$, wobei $\mathcal{F}$ und $\mathcal{G}$ INSEL$^+$–Pfade sind, und es gilt: $\mathcal{F} \neq_N \mathcal{G}$

    Begründung:
    `Caller.ConName` gibt bei einem Aufruf von `OpName` den Namen der Ausführungskomponente des aufrufenden Akteurs an, falls diese benannt ist. Da die mit $\mathcal{F}$ und $\mathcal{G}$ identifizierten benannten DA–Inkarnationsklassen verschieden sind, folgt unmittelbar: $L_1 \wedge L_2 \equiv false$.

14. L_1 ist `Caller.ConName =` $\mathcal{F}$ und L_2 ist $\neg$(`Caller.ConName =` $\mathcal{G}$), wobei $\mathcal{F}$ und $\mathcal{G}$ INSEL$^+$–Pfade sind, und es gilt: $\mathcal{F} =_N \mathcal{G}$
    Begründung:
    L_2 ist in diesem Fall Negation von L_1, es gilt also: $L_1 \wedge L_2 \equiv L_1 \wedge \neg L_1 \equiv false$.
15. L_1 ist `Caller.ConGen =` $\mathcal{F}$ und L_2 ist `Caller.ConName =` $\mathcal{G}$ und die mit $\mathcal{G}$ identifizierte benannte DA–Inkarnationsklasse bezieht sich nicht auf die mit $\mathcal{F}$ identifizierte DA–Generatorklasse
    Begründung:
    `Caller.ConGen` gibt bei einem Aufruf von `OpName` den Namen des Generators der Ausführungskomponente des aufrufenden Akteurs an und `Caller.ConName` den Namen der Ausführungskomponente, falls diese benannt ist. Die mit `Caller.ConName` identifizierte benannte DA–Inkarnation ist also immer Inkarnation bzgl. des mit `Caller.ConGen` identifizierten Generators. Da die mit $\mathcal{G}$ identifizierte benannte DA–Inkarnationsklasse sich nicht auf die mit $\mathcal{F}$ identifizierte DA–Generatorklasse bezieht, folgt unmittelbar: $L_1 \wedge L_2 \equiv false$.
16. L_1 ist $\neg$(`Caller.ConGen =` $\mathcal{F}$) und L_2 ist `Caller.ConName =` $\mathcal{G}$ und die mit $\mathcal{G}$ identifizierte benannte DA–Inkarnationsklasse bezieht sich auf die mit $\mathcal{F}$ identifizierte DA–Generatorklasse
    Begründung:
    Bei einem Aufruf von `OpName` ist die mit `Caller.ConName` identifizierte benannte DA–Inkarnation Inkarnation bzgl. des mit `Caller.ConGen` identifizierten Generators. Die mit $\mathcal{G}$ identifizierte benannte DA–Inkarnationsklasse bezieht sich zwar auf die mit $\mathcal{F}$ identifizierte DA–Generatorklasse, L_1 ist jedoch eine Negation, woraus folgt: $L_1 \wedge L_2 \equiv false$
17. L_1 ist `Caller.ActorGen =` $\mathcal{F}$ und L_2 ist `Caller.ActorName =` $\mathcal{G}$ und die mit $\mathcal{G}$ identifizierte benannte K-Akteur–Inkarnationsklasse bezieht sich nicht auf die mit $\mathcal{F}$ identifizierte Akteur–Generatorklasse
    Begründung:
    `Caller.ActorGen` gibt bei einem Aufruf von `OpName` den Namen des Generators des aufrufenden Akteurs an und `Caller.ActorName` den Namen des Akteurs, falls dieser benannt ist. Der mit `Caller.ActorName` identifizierte benannte K-Akteur ist also immer Inkarnation bzgl. des mit `Caller.ActorGen` identifizierten Akteur–Generators. Da die mit $\mathcal{G}$ identifizierte benannte K-Akteur–Inkarnationsklasse sich nicht auf die mit $\mathcal{F}$ identifizierte Akteur–Generatorklasse bezieht, folgt unmittelbar: $L_1 \wedge L_2 \equiv false$.
18. L_1 ist $\neg$(`Caller.ActorGen =` $\mathcal{F}$) und L_2 ist `Caller.ActorName =` $\mathcal{G}$ und die mit $\mathcal{G}$ identifizierte benannte K-Akteur–Inkarnationsklasse bezieht sich auf die mit $\mathcal{F}$ identifizierte K-Akteur–Generatorklasse
    Begründung:
    Bei einem Aufruf von `OpName` ist der mit `Caller.ActorName` identifizierte benannte K-Akteur Inkarnation bzgl. des mit `Caller.ActorGen` identifizierten K-Akteur–Generators. Die mit $\mathcal{G}$ identifizierte benannte K-Akteur–Inkarnationsklasse bezieht sich zwar auf die mit $\mathcal{F}$ identifizierte K-Akteur–Generatorklasse, L_1 ist jedoch eine Negation, woraus folgt: $L_1 \wedge L_2 \equiv false$
19. L_1 ist $\mathcal{A} = \mathcal{B}$ und L_2 ist $\mathcal{C} < \mathcal{D}$, wobei $\mathcal{A}, \mathcal{B}, \mathcal{C}$ und $\mathcal{D}$ Namen von DE–Inkarnationen sind, und es gilt: $\mathcal{A} =_N \mathcal{C}$ und $\mathcal{B} =_N \mathcal{D}$ oder $\mathcal{A} =_N \mathcal{D}$ und $\mathcal{B} =_N \mathcal{C}$
    Begründung:
    Mit der üblichen Semantik für die auf den vordefinierten elementaren Datentypen

definierten Vergleichsprädikate '=' und '<' ergibt sich unmittelbar: $L_1 \wedge L_2 \equiv false$.

20. L_1 ist $\mathcal{A} = \mathcal{B}$ und L_2 ist $\neg(\mathcal{C} = \mathcal{D})$, wobei $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ und $\mathcal{D}$ Namen von DE–Inkarnationen sind, und es gilt: $\mathcal{A} =_N \mathcal{C}$ und $\mathcal{B} =_N \mathcal{D}$ oder $\mathcal{A} =_N \mathcal{D}$ und $\mathcal{B} =_N \mathcal{C}$
    Begründung:
    L_2 ist in diesem Fall Negation von L_1, es gilt also: $L_1 \wedge L_2 \equiv L_1 \wedge \neg L_1 \equiv false$.
21. L_1 ist $\mathcal{A} < \mathcal{B}$ und L_2 ist $\neg(\mathcal{C} < \mathcal{D})$, wobei $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ und $\mathcal{D}$ Namen von DE–Inkarnationen sind, und es gilt: $\mathcal{A} =_N \mathcal{C}$ und $\mathcal{B} =_N \mathcal{D}$
    Begründung:
    L_2 ist in diesem Fall Negation von L_1, es gilt also: $L_1 \wedge L_2 \equiv L_1 \wedge \neg L_1 \equiv false$.
22. L_1 ist $\mathcal{A} < \mathcal{B}$ und L_2 ist $\mathcal{C} < \mathcal{D}$, wobei $\mathcal{A}, \mathcal{B}, \mathcal{C}$ und $\mathcal{D}$ Namen von DE–Inkarnationen sind, und es gilt: $\mathcal{A} =_N \mathcal{D}$ und $\mathcal{B} =_N \mathcal{C}$
    Begründung:
    Mit der üblichen Semantik für das Vergleichsprädikat '<' ergibt sich unmittelbar: $L_1 \wedge L_2 \equiv false$.
23. L_1 ist `FALSE` und L_2 ist ein beliebiges Literal gemäß (5.15).
    Begründung:
    Es gilt: $L_1 \wedge L_2 \equiv false \wedge L_2 \equiv false$

△

Beispiel

In dem INSEL$^+$–Programm, das das Kontenverwaltungssystem implementiert, wären zum Beispiel folgende Paare von Literalen widersprüchlich, wenn sie in einem konjunktiven Teilterm der Disjunktiven Normalform eines in dem Zugriffsrestriktions–Part der Depot–Generatorklasse `KontoTyp` festgelegten Zugriffsrestriktionsausdrucks enthalten wären:

1. Zwei widersprüchliche `IN_ACL`–Prädikate:
   `IN_ACL(Caller.User, Caller.Role, THIS, THIS)` und
   `NOT(IN_ACL(Caller.User, Caller.Role, THIS, THIS))`

2. Zwei widersprüchliche Benutzerrestriktionen:
   `Caller.User = InhaberUid` und `Caller.User = BetreuerUid`

3. Zwei widersprüchliche Rollenrestriktionen:
   `Caller.Role = Kunde` und `Caller.Role = Kundenbetreuer`

4. Zwei widersprüchliche Kontextrestriktionen:

   (a) `Caller.ConGen = KontoVerwalter.KontoEroeffnen` und
       `Caller.ConGen = KontoVerwalter.KontoAufloesen`

   (b) `Caller.ConGen = BerechneKredit` und
       `Caller.ConName = KontoVerwalter`

5. Zwei widersprüchliche Wertvergleiche auf DE–Inkarnationen:
   `AnzahlBarAbhebungen < 3` und `AnzahlBarAbhebungen = 3`

◇

Mit Satz (5.2) sind die Kriterien, auf deren Basis der INSEL$^+$–Übersetzer die Erfüllbarkeit eines Zugriffsrestriktionsausdrucks analysieren kann, vollständig erklärt. Jeder in einem INSEL$^+$–Programm festgelegte Zugriffsrestriktionsausdruck wird vom INSEL$^+$–Übersetzer anhand dieser Kriterien auf Erfüllbarkeit überprüft. Stellt der INSEL$^+$–Übersetzer fest, daß ein Zugriffsrestriktionsausdruck unerfüllbar ist, so wird eine entsprechende Fehlermeldung ausgegeben, die u.a. den Zugriffsrestriktionsausdruck in Disjunktiver Normalform anzeigt und Hinweise auf die in deren konjunktiven Teiltermen auftretenden widersprüchlichen Paare von Literalen enthält.

5.3.2.2 Kriterien zur Analyse der Widerspruchsfreiheit zweier Zugriffsrestriktionsausdrücke

In diesem Abschnitt werden die Kriterien angegeben, die der INSEL$^+$–Übersetzer im Rahmen der Analyse zweier Zugriffsrestriktionsausdrücke auf Widersprüchlichkeit bzw. Widerspruchsfreiheit überprüft. Bevor diese Analyse durchgeführt werden kann, müssen zunächst die Paare von Zugriffsrestriktionsausdrücken eines INSEL$^+$–Programms ermittelt werden, die auf Widersprüchlichkeit bzw. Widerspruchsfreiheit zu analysieren sind.

Ermittlung der Paare zu analysierender Zugriffsrestriktionsausdrücke

Zwei Zugriffsrestriktionsausdrücke R_1 und R_2 sind höchstens dann auf Widerspruchsfreiheit zu überprüfen, wenn die Operation, für die R_2 festgelegt ist, potentiell im Kontext der Ausführung der Operation, für die R_1 definiert ist, aufgerufen wird. Die Operationen, für die Zugriffsrestriktionsausdrücke festgelegt werden können, sind die *erzeuge*–Operationen auf explizit und implizit zugriffskontrollierten DA–Generatoren. Ist ein INSEL$^+$–Programm gegeben, so sind also zwei in dem Programm definierte Zugriffsrestriktionsausdrücke R_1 und R_2 höchstens dann auf Widerspruchsfreiheit zu analysieren, wenn für die zugriffskontrollierten DA–Generatorklassen G_1 und G_2, für die R_1 und R_2 festgelegt sind, gilt, daß im Kontext der Ausführung der kanonischen Operation einer Inkarnation bzgl. G_1 potentiell eine Inkarnation bzgl. G_2 erzeugt wird, also potentiell die *erzeuge*–Operation auf einem Generator der Klasse G_2 aufgerufen wird[19]. Diese Abhängigkeiten zwischen DA–Generatorklassen werden in dem in Definition (5.8) definierten attributierten statischen Aufrufgraph eines INSEL$^+$–Programms erfaßt. Sind G_1 und G_2 zwei DA–Generatorklassen eines INSEL$^+$–Programms, zwischen denen es in dem statischen Aufrufgraph mindestens einen Aufrufpfad gibt, dann besagt dies gerade, daß im Kontext der Ausführung der kanonischen Operation einer Inkarnation bzgl. G_1 potentiell eine Inkarnation bzgl. G_2 erzeugt wird. Daraus folgt also, daß zwei Zugriffsrestriktionsausdrücke dann auf Widerspruchsfreiheit zu analysieren sind, wenn es zwischen den beiden DA–Generatorklassen, für deren *erzeuge*–Operationen die Zugriffsrestriktionsausdrücke festgelegt sind, mindestens einen Aufrufpfad gibt.

Die Menge der sich nach diesem Kriterium ergebenden Paare von zu überprüfenden Zugriffsrestriktionsausdrücken läßt sich unter Berücksichtigung der besonderen Rolle, die die Benutzerrepräsentanten in einem INSEL$^+$–System spielen, weiter einschränken. Wie in Abschnitt 4.2.2 erläutert, sind die Benutzerrepräsentanten eines INSEL$^+$–Systems spezielle M-Akteure, von denen aus Benutzer Operationsausführungen anstoßen können. Einem Benutzerrepräsentanten und allen DA–Inkarnationen, die ausgehend von diesem unmittelbar oder mittelbar

[19]Anmerkung: Ist G eine DA–Generatorklasse, so wird im weiteren häufig für *„... Inkarnation bzgl. eines Generators der Generatorklasse G ...“* abkürzend *„... Inkarnation bzgl. G ...“* geschrieben, obwohl eine Inkarnation immer bzgl. eines Generators und nicht einer Generatorklasse erzeugt wird.

erzeugt werden, wird der Benutzer zugeordnet, für den der Benutzerrepräsentant erzeugt wurde. Die Rolle des Benutzerrepräsentanten und aller von ihm aus erzeugten DA–Inkarnationen ist durch die Rolle dieses Benutzers festgelegt. Allen DA–Inkarnationen, die nicht ausgehend von einem Benutzerrepräsentanten erzeugt werden, wird der vordefinierte abstrakte Benutzer *System* und die vordefinierte Rolle *SystemRole* zugeordnet. In einem INSEL$^+$–System ist somit zwischen Operationsausführungen, die von einem Benutzerrepräsentanten aus initiiert werden, und Operationsausführungen, die nicht von einem Benutzerrepräsentanten ausgehen, zu unterscheiden. Entsprechend dieser Unterscheidung sind die Zugriffsrestriktionsausdrücke zweier DA–Generatorklassen nur dann auf Widerspruchsfreiheit zu analysieren, wenn zwischen ihnen mindestens ein Aufrufpfad existiert, der keine Generatorklasse eines Benutzerrepräsentanten enthält. Die Generatorklassen für Benutzerrepräsentanten bilden somit eine Art „Trennlinie" in dem statischen Aufrufgraph eines INSEL$^+$–Programms, die bei der Ermittlung der Paare von DA–Generatorklassen, deren Zugriffsrestriktionsausdrücke auf Widerspruchsfreiheit zu überprüfen sind, zu berücksichtigen ist.

Vorbereitend für die präzise Angabe der Menge der auf Widerspruchsfreiheit zu analysierenden Paare von Zugriffsrestriktionsausdrücken eines INSEL$^+$–Programms $\mathcal{P}$, wird in der folgenden Definition eine Abbildung eingeführt, die jedem Knoten v des attributierten statischen Aufrufgraphen von $\mathcal{P}$ unter Berücksichtigung der angesprochenen „Trennlinie" die Menge der DA–Generatorklassen zuordnet, auf deren Generatoren im Kontext der Ausführung der kanonischen Operation einer Inkarnation bzgl. v potentiell die Operation *erzeuge* aufgerufen wird. Treten in einem INSEL$^+$–Programm Deklarationen benannter DA–Inkarnationen auf, so enthält der statische Aufrufgraph neben den DA–Generatorklassen des Programms auch benannte DA–Inkarnationsklassen als Knoten. Letztere sind bei Definition der durch die Abbildung festgelegten Menge von „potentiell aufgerufenen" DA–Generatorklassen entsprechend zu berücksichtigen.

Definition 5.18.: Abbildung *PotCalledGenerators*

Gegeben seien das INSEL$^+$–Programm $\mathcal{P} = (\mathcal{D}, \delta, \gamma, \iota)$ und der attributierte statische Aufrufgraph $\mathcal{G}(\mathcal{P}) = (V, K, \mu)$ von $\mathcal{P}$.

Die Abbildung *PotCalledGenerators* – kurz *pcg* – ist wie folgt definiert:

$$pcg : V \longrightarrow POT(\mathcal{D}^{DAG})$$

$$\begin{aligned} pcg(v) \triangleq \quad \{y \in \mathcal{D}^{DAG} \mid\ & \exists p \in AP(v,y) \text{ mit } p = (v_0, v_1, \ldots, v_n), v_0 = v, v_n = y \text{ und } n \geq 1 : \\ & \forall i \in \{1, \ldots, n-1\} : kindof(v_i) \neq BenRep \ \vee \\ & \exists d \in \mathcal{D}^{NDAI} : (y, d) \in \gamma \ \wedge \\ & \exists p \in AP(v,d) \text{ mit } p = (v_0, v_1, \ldots, v_n), v_0 = v, v_n = d \text{ und } n \geq 1 : \\ & \forall i \in \{1, \ldots, n-1\} : kindof(v_i) \neq BenRep\} \end{aligned}$$

□

Gemäß Definition (5.18) ist eine DA–Generatorklasse y genau dann Element der Menge $pcg(v)$, wenn

1. es mindestens einen Aufrufpfad zwischen v und y gibt, der keine Generatorklasse eines Benutzerrepräsentanten enthält, oder
2. es eine benannte DA–Inkarnationsklasse d gibt, die sich auf y bezieht und es mindestens einen Aufrufpfad zwischen v und d gibt, der keine Generatorklasse eines Benutzerrepräsentanten enthält.

Aufbauend auf Definition (5.18) kann nun die Menge der Paare von zugriffskontrollierten DA–Generatorklassen eines INSEL$^+$–Programms angegeben werden, deren zugeordnete Zugriffsrestriktionsausdrücke auf Widerspruchsfreiheit zu überprüfen sind.

Definition 5.19.: Paare zu analysierender Zugriffsrestriktionsausdrücke

Seien $\mathcal{P}$ und $\mathcal{G}(\mathcal{P})$ wie in Definition (5.18) gegeben. Die Menge

$$AnalysePairs(\mathcal{P}) \subset \mathcal{D}^{DAG} \times \mathcal{D}^{DAG}$$

der Paare von DA–Generatorklassen, deren zugeordnete Zugriffsrestriktionsausdrücke auf Widerspruchsfreiheit zu analysieren sind, ist wie folgt definiert:

$$\begin{array}{lcl} (v_1, v_2) \in AnalysePairs(\mathcal{P}) & \stackrel{\triangle}{\Longleftrightarrow} & kindof(v_1) \neq BenRep \wedge zra(v_1) = zra(v_2) = true \wedge \\ \text{mit } v_1, v_2 \in \mathcal{D}^{DAG} & & (v_2 \in pcg(v_1) \ \vee \ \exists d \in \mathcal{D}^{NDAI} : (v_1, d) \in \gamma \wedge \\ & & \qquad v_2 \in pcg(d)) \end{array}$$

□

Beispiel

Als Beispiel wird im folgenden die Menge $AnalysePairs(KVS)$ des INSEL$^+$–Programms angegeben, das das Kontenverwaltungssystem implementiert (vgl. hierzu auch Abbildung 5.5)[20].

$$\begin{array}{lcl} AnalysePairs(KVS) & = & \{(\texttt{Abheben}, \texttt{FuegeEin}), (\texttt{Einzahlen}, \texttt{FuegeEin}), \\ & & (\texttt{ZeigeKontoauszug}, \texttt{ListeKontoauszug}), \\ & & (\texttt{ZeigeKontobewegungen}, \texttt{ListeKontobewegungen}), \\ & & (\texttt{KontoTyp}, \texttt{KontobewegungsListenTyp}), \\ & & (\texttt{KontoEroeffnen}, \texttt{KontoTyp}), \\ & & (\texttt{KontoEroeffnen}, \texttt{KontobewegungsListenTyp}), \\ & & (\texttt{KontoAufloesen}, \texttt{Aufloesen}), \\ & & (\texttt{GibKontoZeiger}, \texttt{LeseKontonummer}) \\ & & (\texttt{Zinsberechnung}, \texttt{Abheben}), (\texttt{Zinsberechnung}, \texttt{Einzahlen}), \\ & & (\texttt{Zinsberechnung}, \texttt{LeseKontostand}), \\ & & (\texttt{Zinsberechnung}, \texttt{FuegeEin}), \\ & & (\texttt{KontoVerwalterTyp}, \texttt{Zinsberechnung}), \\ & & (\texttt{KontoVerwalterTyp}, \texttt{Abheben}), \\ & & (\texttt{KontoVerwalterTyp}, \texttt{FuegeEin}), \\ & & (\texttt{KontoVerwalterTyp}, \texttt{Einzahlen}), \\ & & (\texttt{KontoVerwalterTyp}, \texttt{LeseKontostand}), \\ & & (\texttt{Ueberweiser}, \texttt{Abheben}), (\texttt{Ueberweiser}, \texttt{Einzahlen}), \\ & & (\texttt{Ueberweiser}, \texttt{FuegeEin}), \\ & & (\texttt{BerechneKredit}, \texttt{LeseMaxKredit}), \\ & & (\texttt{BerechneKredit}, \texttt{LesePersKundenDaten}), \\ & & (\texttt{BerechneKredit}, \texttt{ZeigeKontobewegungen}), \\ & & (\texttt{BerechneKredit}, \texttt{ListeKontobewegungen})\} \end{array}$$

◇

Für das Weitere seien G_1 und G_2 zwei zugriffskontrollierte DA–Generatorklassen eines INSEL$^+$–Programms $\mathcal{P}$ mit $(G_1, G_2) \in AnalysePairs(\mathcal{P})$, und R_1 und R_2 seien die G_1 bzw. G_2

[20]In der angegebenen Menge fehlen die Elemente, die sich aus den DA–Generatorklassen und benannten DA–Inkarnationsklassen der Benutzerverwaltung ergeben.

zugeordneten Zugriffsrestriktionsausdrücke. Dann sind R_1 und R_2 gemäß (5.19) auf Widerspruchsfreiheit zu analysieren. Für diese Analyse wird vorausgesetzt, daß R_1 und R_2 jeweils erfüllbar sind (vgl. Abschnitt 5.3.2.1). Laut Definition (5.12) sind die beiden Zugriffsrestriktionsausdrücke R_1 und R_2 genau dann widerspruchsfrei, wenn sie nicht widersprüchlich sind. R_1 und R_2 sind widersprüchlich genau dann, wenn unter Berücksichtigung der in R_1 und R_2 spezifizierten Kontextrestriktionen aus der Gültigkeit von R_1 stets die Ungültigkeit von R_2 folgt. Dafür werden im folgenden hinreichende, statisch überprüfbare Bedingungen angegeben. Da in dem Fall, daß R_2 Kontextrestriktionen enthält[21], weitergehende Prüfungen erforderlich sind, um festzustellen, ob diese Kontextrestriktionen unter Berücksichtigung eventuell in R_1 spezifizierter Kontextrestriktionen und der möglichen Aufrufpfade zwischen G_1 und G_2 erfüllbar sind, wird zunächst der Fall betrachtet, daß R_2 keine Kontextrestriktionen beinhaltet.

R_2 enthält keine Kontextrestriktionen

Um R_1 und R_2 auf Widersprüchlichkeit zu überprüfen, werden R_1 und R_2 zunächst in ihre Disjunktive Normalform gemäß Definition (5.14) umgewandelt. Es seien:

(5.20)

$$DNF(R_1) = \bigvee_{i=1}^{m_1} \bigwedge_{j=1}^{n_i} L_{ij} \qquad \text{die Disjunktive Normalform von } R_1$$

$$DNF(R_2) = \bigvee_{k=1}^{m_2} \bigwedge_{l=1}^{n_k} K_{kl} \qquad \text{die Disjunktive Normalform von } R_2$$

Für $i \in \{1, \ldots, m_1\}$ sei $L_i \stackrel{\triangle}{=} \bigwedge_{j=1}^{n_i} L_{ij}$ der i-te konjunktive Teilterm von $DNF(R_1)$ und für $k \in \{1, \ldots, m_2\}$ sei $K_k \stackrel{\triangle}{=} \bigwedge_{l=1}^{n_k} K_{kl}$ der k-te konjunktive Teilterm von $DNF(R_2)$.

□

In der folgenden Definition werden die Begriffe der Widersprüchlichkeit und der Widerspruchsfreiheit für zwei konjunktive Teilterme der Disjunktiven Normalformen zweier Zugriffsrestriktionsausdrücke eingeführt.

Definition 5.21.: Widersprüchlichkeit zweier konjunktiver Teilterme

Seien R_1 und R_2 zwei Zugriffsrestriktionsausdrücke, die jeweils erfüllbar sind; R_2 enthalte keine Kontextrestriktionen. L_i und K_k seien gemäß (5.20) zwei konjunktive Teilterme der Disjunktiven Normalformen von R_1 und R_2.

L_i und K_k sind **widersprüchlich** $\stackrel{\triangle}{\Longleftrightarrow}$ Es gibt $j \in \{1, \ldots, n_i\}$ und $l \in \{1, \ldots, n_k\}$, so daß gilt: die Literale L_{ij} und K_{kl} sind widersprüchlich.

L_i und K_k sind **widerspruchsfrei** $\stackrel{\triangle}{\Longleftrightarrow}$ L_i und K_k sind nicht widersprüchlich.

□

Sind zwei konjunktive Teilterme L_i und K_k widersprüchlich, so folgt aus Definition (5.21) unmittelbar, daß die Konjunktion $L_i \wedge K_k$ immer den Wert *false* hat. Aufbauend auf dieser Definition formuliert der folgende Satz eine notwendige und hinreichende Bedingung dafür, daß zwei Zugriffsrestriktionsausdrücke widersprüchlich sind.

[21] Zur Erinnerung: Kontextrestriktionen werden durch Gleichungen der Form `Caller.ConGen` = <*inselp-path*>, `Caller.ConName` = <*inselp-path*>, `Caller.ActorGen` = <*inselp-path*> und `Caller.ActorName` = <*inselp-path*> spezifiziert.

Satz 5.3.: Widersprüchlichkeit zweier Zugriffsrestriktionsausdrücke

Seien R_1 und R_2 sowie L_i und K_k wie in Definition (5.21) gegeben. Dann gilt:

R_1 und R_2 sind widersprüchlich $\Longleftrightarrow$ Für alle $i \in \{1, \ldots, m_1\}$ und $k \in \{1, \ldots, m_2\}$ gilt: L_i und K_k sind widersprüchlich

$\triangle$

Die Zugriffsrestriktionsausdrücke R_1 und R_2 sind also genau dann widersprüchlich, wenn alle Paare von konjunktiven Teiltermen ihrer Disjunktiven Normalformen widersprüchlich sind.

Beweis:
Seien x und y zwei zugriffskontrollierte Komponenten. $op \in O^E(y)$ sei die äußere Operation von y, für die R_1 festgelegt ist, und $op' \in O^E(x)$ sei die äußere Operation von x, für die R_2 festgelegt ist.

"$\Longrightarrow$": Beweis durch Widerspruch
Voraussetzung: R_1 und R_2 sind widersprüchlich; d.h. (vgl. Definition (5.12)) es gilt für alle $t \in \Lambda(y)$ und $a \in A_t$:

(i) $\omega(R_1, a, t) = true \Longrightarrow \omega(R_2, a', t') = false$
für alle Zeitpunkte $t' > t$ und alle Akteure a', die zum Zeitpunkt t' im Kontext einer durch a in t initiierten op–Ausführung op' auf x aufrufen.

Aufgrund der Annahme, daß R_2 keine Kontextrestriktionen enthält, ist der Wert $\omega(R_2, a', t')$ unabhängig von den Werten $ActGen_{t'}(a')$, $ActName_{t'}(a')$, $ConGen_{t'}(a')$ und $ConName_{t'}(a')$ des Kontextattributs des Akteurs a'. Da weiter für alle Akteure a', die im Kontext einer durch a initiierten op–Ausführung erzeugt werden, gilt: $User_t(a) = User_{t'}(a')$ und $Role_t(a) = Role_{t'}(a')$, kann die Voraussetzung (i) wie folgt vereinfacht werden:

$$(ii) \quad \omega(R_1, a, t) = true \Longrightarrow \omega(R_2, a, t) = false$$

Annahme: es gibt $i \in \{1, \ldots, m_1\}$ und $k \in \{1, \ldots, m_2\}$, so daß L_i und K_k widerspruchsfrei sind. Dann gibt es, da R_1 und R_2 jeweils erfüllbar sind, ein $t_1 \in \Lambda(y)$ und einen Akteur $a \in A_{t_1}$, so daß $\omega(L_i, a, t_1) = true$ und $\omega(K_k, a, t_1) = true$ gilt. Damit gilt jedoch auch $\omega(DNF(R_1), a, t_1) = \omega(R_1, a, t_1) = true$ und $\omega(DNF(R_2), a, t_1) = \omega(R_2, a, t_1) = true$. Dies ist jedoch ein Widerspruch zu (ii).

"$\Longleftarrow$": Voraussetzung: Für alle $i \in \{1, \ldots, m_1\}$ und $k \in \{1, \ldots, m_2\}$ gilt: L_i und K_k sind widersprüchlich, d.h. für alle $t \in \Lambda(y)$ und alle $a \in A_t$ gilt: $\omega(L_i \wedge K_k, a, t) = false$. Daraus folgt für alle $i \in \{1, \ldots, m_1\}$:

$$(iii) \quad \omega(L_i, a, t) = true \Longrightarrow \omega(K_k, a, t) = false \text{ für alle } k \in \{1, \ldots, m_2\}$$

und damit

$$(iv) \quad \omega(DNF(R_1), a, t) = true \Longrightarrow \omega(DNF(R_2), a, t) = false$$

Aus (*iv*) folgt aufgrund der Äquivalenz eines Zugriffsrestriktionsausdrucks und seiner Disjunktiven Normalform unmittelbar (*ii*). (*ii*) besagt jedoch gerade für den Fall, daß R_2 keine Kontextrestriktionen enthält, daß R_1 und R_2 widersprüchlich sind.

$\bigtriangledown$

Analyseverfahren

Auf Basis des Satzes (5.3) kann die Widerspruchsfreiheit zweier Zugriffsrestriktionsausdrücke R_1 und R_2, die die Voraussetzungen des Satzes erfüllen, vom INSEL$^+$–Übersetzer wie folgt analysiert werden: nach Umwandlung von R_1 und R_2 in ihre Disjunktiven Normalformen $DNF(R_1)$ und $DNF(R_2)$ wird zunächst der erste konjunktive Teilterm von $DNF(R_1)$ mit allen konjunktiven Teiltermen von $DNF(R_2)$ auf Widersprüchlichkeit analysiert, indem überprüft wird, ob in der Konjunktion der jeweiligen konjunktiven Teilterme mindestens ein Paar widersprüchlicher Literale enthalten ist. Anschließend wird der zweite konjunktive Teilterm von $DNF(R_1)$ analog mit allen konjunktiven Teiltermen von $DNF(R_2)$ auf Widersprüchlichkeit überprüft, usw. Die Analyse kann abgebrochen werden, sobald ein Paar konjunktiver Teilterme gefunden wird, das nicht widersprüchlich ist, d.h. in dem keine widersprüchlichen Literale enthalten sind. In diesem Fall sind R_1 und R_2 widerspruchsfrei. Wird kein solches Paar widerspruchsfreier konjunktiver Teilterme gefunden, so sind R_1 und R_2 nach Satz (5.3) widersprüchlich.

In Satz (5.2) wurden bereits statisch überprüfbare Bedingungen für die Widersprüchlichkeit zweier Literale, die in der Disjunktiven Normalform eines Zugriffsrestriktionsausdrucks auftreten können, angegeben. Ein Teil der in diesem Satz aufgeführten Bedingungen kann analog für die im Rahmen des beschriebenen Analyseverfahrens zu überprüfende Widersprüchlichkeit zweier Literale, die in einem Paar konjunktiver Teilterme von $DNF(R_1)$ und $DNF(R_2)$ enthalten sind, genutzt werden. Die Bedingungen des Satzes (5.2), die Kontextrestriktionen betreffen, sind für das hier betrachtete Analyseverfahren nicht relevant, da laut Voraussetzung der Zugriffsrestriktionsausdruck R_2 und damit auch alle konjunktiven Teilterme von $DNF(R_2)$ keine Kontextrestriktionen enthalten. In dem folgenden Satz sind die Kriterien, anhand derer der INSEL$^+$–Übersetzer die Widersprüchlichkeit zweier Literale eines Paares konjunktiver Teilterme von $DNF(R_1)$ und $DNF(R_2)$ überprüft, zusammengefaßt. Zu jeder Bedingung wird wiederum informell begründet, warum bei Erfüllung der Bedingung die Konjunktion der beiden Literale stets den Wert *false* hat. Die Begründungen basieren wesentlich auf der in (5.5) getroffenen Annahme der programmweiten Eindeutigkeit der Bezeichner eines INSEL$^+$–Programms.

Satz 5.4.: Kriterien für die Widersprüchlichkeit zweier Literale eines Paares konjunktiver Teilterme

Seien L_{ij} und K_{kl} zwei Literale gemäß (5.21). `G1` sei der Name der zugriffskontrollierten DA–Generatorklasse, für die der Zugriffsrestriktionsausdruck R_1 festgelegt ist, und `G2` sei der Name der zugriffskontrollierten DA–Generatorklasse, der der Zugriffsrestriktionsausdruck R_2 zugeordnet ist.

Die Literale L_{ij} und K_{kl} sind **widersprüchlich**, wenn eine der folgenden Bedingungen erfüllt ist:

1. L_{ij} ist `IN_ACL(UserId1, RoleId1, CompId1, OpId1)` und K_{kl} ist $\neg$`(IN_ACL(UserId2, RoleId2, CompId2, OpId2))` und es gilt:
   (a) `G2` ist explizit zugriffskontrollierte DA–Generatorklasse und
      - `UserId1` $=_N$ `UserId2` und

- `RoleId1` $=_N$ `RoleId2` und
- `CompId1` $=_N$ `G2` und
- `CompId2` $=_N$ `THIS` und
- `OpId1` $=_N$ `Create` und
- `OpId2` $=_N$ `THIS` oder `OpId2` $=_N$ `Create`

oder

(b) `G2` ist explizit zugriffskontrollierte DA–Generatorklasse und

- `UserId1` $=_N$ `UserId2` und
- `RoleId1` $=_N$ `RoleId2` und
- `CompId1` $\neq_N$ `G2` und
- `CompId2` $\neq_N$ `THIS` und
- `CompId1` $=_N$ `CompId2` und
- `OpId1` $=_N$ `OpId2`

oder

(c) `G2` ist implizit zugriffskontrollierte DA–Generatorklasse und

- `UserId1` $=_N$ `UserId2` und
- `RoleId1` $=_N$ `RoleId2` und
- `CompId1` ist entweder der Name einer benannten DA–Inkarnationsklasse, die sich auf `G3` bezieht, oder der Name einer Zeiger–Inkarnationsklasse, die mit `G3` qualifiziert ist, wobei `G3` die explizit zugriffskontrollierte DA-Generatorklasse ist, in deren Deklarationsteil `G2` enthalten ist, und
- `CompId2` $=_N$ `THIS` und
- `OpId1` $=_N$ `G2` und
- `OpId2` $=_N$ `THIS` oder `OpId2` $=_N$ `G2`

oder

(d) `G2` ist implizit zugriffskontrollierte DA–Generatorklasse und

- `UserId1` $=_N$ `UserId2` und
- `RoleId1` $=_N$ `RoleId2` und
- `CompId1` ist entweder der Name einer benannten DA–Inkarnationsklasse, die sich auf `G3` bezieht, oder der Name einer Zeiger–Inkarnationsklasse, die mit `G3` qualifiziert ist, wobei `G3` die explizit zugriffskontrollierte DA-Generatorklasse ist, in deren Deklarationsteil `G2` enthalten ist, und
- `CompId2` $=_N$ `THIS` und
- `OpId1` $=_N$ `G4`, wobei `G4` eine implizit zugriffskontrollierte DA–Generatorklasse ist, die im gleichen Deklarationsteil wie `G2` definiert ist, und
- `OpId2` $=_N$ `G4`

oder

(e) `G2` ist implizit zugriffskontrollierte DA–Generatorklasse und

- `UserId1` $=_N$ `UserId2` und
- `RoleId1` $=_N$ `RoleId2` und
- `CompId1` ist weder der Name einer benannten DA–Inkarnationsklasse, die sich auf `G3` bezieht, noch der Name einer Zeiger–Inkarnationsklasse, die mit `G3` qualifiziert ist, wobei `G3` die explizit zugriffskontrollierte DA-Generatorklasse ist, in deren Deklarationsteil `G2` enthalten ist, und

- CompId2 $\neq_N$ THIS und
- CompId1 $=_N$ CompId2
- OpId1 $=_N$ OpId2

Begründung:
Ist (a), (b), (c), (d) oder (e) erfüllt, so identifizieren CompId1 und CompId2 zur Laufzeit die gleiche zugriffskontrollierte Komponente sowie OpId1 und OpId2 die gleiche äußere Operation dieser Komponente. Damit sind die beiden IN_ACL–Prädikate jedoch semantisch äquivalent (siehe hierzu Definition (4.9)). In diesem Fall ist K_{kl} also Negation von L_{ij} und damit: $L_{ij} \wedge K_{kl} \equiv L_{ij} \wedge \neg L_{ij} \equiv false$.

2. L_{ij} ist ¬(IN_ACL(UserId1, RoleId1, CompId1, OpId1)) und
K_{kl} ist IN_ACL(UserId2, RoleId2, CompId2, OpId2)
und es ist eine der Bedingungen (a), (b), (c), (d) oder (e) von 1. erfüllt.
Begründung:
In diesem Fall ist L_{ij} Negation von K_{kl}, und damit gilt:
$L_{ij} \wedge K_{kl} \equiv \neg K_{kl} \wedge K_{kl} \equiv false$.
3. L_{ij} ist ACCESSED(UserId1, CompId1, OpId1) und
K_{kj} ist ¬(ACCESSED(UserId2, CompId2, OpId2)) und es gilt:
(a) G2 ist explizit zugriffskontrollierte DA–Generatorklasse und
   - UserId1 $=_N$ UserId2 und
   - CompId1 $=_N$ G2 und
   - CompId2 $=_N$ THIS und
   - OpId1 $=_N$ OpId2 oder OpId2 $=_N$ ANY

   oder
(b) G2 ist implizit zugriffskontrollierte DA–Generatorklasse und
   - UserId1 $=_N$ UserId2 und
   - CompId1 ist entweder der Name einer benannten DA–Inkarnationsklasse, die sich auf G3 bezieht, oder der Name einer Zeiger–Inkarnationsklasse, die mit G3 qualifiziert ist, wobei G3 die explizit zugriffskontrollierte DA-Generatorklasse ist, in deren Deklarationsteil G2 enthalten ist, und
   - CompId2 $=_N$ THIS und
   - OpId1 $=_N$ OpId2 oder OpId2 $=_N$ ANY

   oder
(c) G2 ist explizit zugriffskontrollierte DA–Generatorklasse und
   - UserId1 $=_N$ UserId2 und
   - CompId1 $\neq_N$ G2 und
   - CompId2 $\neq_N$ THIS und
   - CompId1 $=_N$ CompId2 und
   - OpId1 $=_N$ OpId2 oder OpId2 $=_N$ ANY

   oder
(d) G2 ist implizit zugriffskontrollierte DA–Generatorklasse und
   - UserId1 $=_N$ UserId2 und
   - CompId1 ist weder der Name einer benannten DA–Inkarnationsklasse, die sich auf G3 bezieht, noch der Name einer Zeiger–Inkarnationsklasse, die mit G3 qualifiziert ist, wobei G3 die explizit zugriffskontrollierte DA-Generatorklasse ist, in deren Deklarationsteil G2 enthalten ist, und

- `CompId2` $\neq_N$ `THIS` und
- `CompId1` $=_N$ `CompId2` und
- `OpId1` $=_N$ `OpId2` oder oder `OpId2` $=_N$ `ANY`

Begründung:
Gilt (a), (b), (c) oder (d), so sind die beiden `ACCESSED`–Prädikate semantisch äquivalent, d.h. K_{kl} ist Negation von L_{ij}, und damit gilt:
$L_{ij} \wedge K_{kl} \equiv L_{ij} \wedge \neg L_{ij} \equiv false$.

4. L_{ij} ist ¬`(ACCESSED(UserId1, CompId1, OpId1))` und
K_{kj} ist `ACCESSED(UserId2, CompId2, OpId2)` und es gilt:

(a) `G2` ist explizit zugriffskontrollierte DA–Generatorklasse und
- `UserId1` $=_N$ `UserId2` und
- `CompId1` $=_N$ `G2` und
- `CompId2` $=_N$ `THIS` und
- `OpId1` $=_N$ `OpId2` oder `OpId1` $=_N$ `ANY`

oder

(b) `G2` ist implizit zugriffskontrollierte DA–Generatorklasse und
- `UserId1` $=_N$ `UserId2` und
- `CompId1` ist entweder der Name einer benannten DA–Inkarnationsklasse, die sich auf `G3` bezieht, oder der Name einer Zeiger–Inkarnationsklasse, die mit `G3` qualifiziert ist, wobei `G3` die explizit zugriffskontrollierte DA-Generatorklasse ist, in deren Deklarationsteil `G2` enthalten ist, und
- `CompId2` $=_N$ `THIS` und
- `OpId1` $=_N$ `OpId2` oder `OpId1` $=_N$ `ANY`

oder

(c) `G2` ist explizit zugriffskontrollierte DA–Generatorklasse und
- `UserId1` $=_N$ `UserId2` und
- `CompId1` $\neq_N$ `G2` und
- `CompId2` $\neq_N$ `THIS` und
- `CompId1` $=_N$ `CompId2` und
- `OpId1` $=_N$ `OpId2` oder `OpId1` $=_N$ `ANY`

oder

(d) `G2` ist implizit zugriffskontrollierte DA–Generatorklasse und
- `UserId1` $=_N$ `UserId2` und
- `CompId1` ist weder der Name einer benannten DA–Inkarnationsklasse, die sich auf `G3` bezieht, noch der Name einer Zeiger–Inkarnationsklasse, die mit `G3` qualifiziert ist, wobei `G3` die explizit zugriffskontrollierte DA-Generatorklasse ist, in deren Deklarationsteil `G2` enthalten ist, und
- `CompId2` $\neq_N$ `THIS` und
- `CompId1` $=_N$ `CompId2` und
- `OpId1` $=_N$ `OpId2` oder oder `OpId1` $=_N$ `ANY`

Begründung:
Gilt (a), (b), (c) oder (d), so sind die beiden `ACCESSED`–Prädikate semantisch äquivalent, d.h. L_{ij} ist Negation von K_{kl}, und damit gilt:
$L_{ij} \wedge K_{kl} \equiv \neg K_{kl} \wedge K_{kl} \equiv false$.

5. L_{ij} ist `Caller.User = UserId1` und K_{kl} ist `Caller.User = UserId2` und es gilt: `UserId1` $\neq_N$ `UserId2`

   Begründung:
   Aus der Annahme $(G_1, G_2) \in \mathit{AnalysePairs}(\mathcal{P})$ folgt, daß `Caller.User` in beiden Literalen stets den gleichen Wert hat. `UserId1` und `UserId2` haben jedoch einen unterschiedlichen Wert. Daraus folgt unmittelbar: $L_{ij} \wedge K_{kl} \equiv false$.

6. L_{ij} ist `Caller.User = UserId1` und K_{kl} ist $\neg$`(Caller.User = UserId2)` oder L_{ij} ist $\neg$`(Caller.User = UserId1)` und K_{kl} ist `Caller.User = UserId2` und es gilt: `UserId1` $=_N$ `UserId2`

   Begründung:
   K_{kl} (bzw. L_{ij}) ist in diesem Fall Negation von L_{ij} (bzw. K_{kl}). Es gilt also: $L_{ij} \wedge K_{kl} \equiv false$.

7. L_{ij} ist `Caller.Role = RoleId1` und K_{kl} ist `Caller.Role = RoleId2` und es gilt: `RoleId1` $\neq_N$ `RoleId2`

   Begründung:
   Aus der Annahme $(G_1, G_2) \in \mathit{AnalysePairs}(\mathcal{P})$ folgt, daß `Caller.Role` in beiden Literalen stets den gleichen Wert hat. `RoleId1` und `RoleId2` haben jedoch einen unterschiedlichen Wert. Daraus folgt unmittelbar: $L_{ij} \wedge K_{kl} \equiv false$.

8. L_{ij} ist `Caller.Role = RoleId1` und K_{kl} ist $\neg$`(Caller.Role = RoleId2)` oder L_{ij} ist $\neg$`(Caller.Role = RoleId1)` und K_{kl} ist `Caller.Role = RoleId2` und es gilt: `RoleId1` $=_N$ `RoleId2`

   Begründung:
   K_{kl} (bzw. L_{ij}) ist in diesem Fall Negation von L_{ij} (bzw. K_{kl}). Es gilt also: $L_{ij} \wedge K_{kl} \equiv false$.

9. L_{ij} ist $\mathcal{A} = \mathcal{B}$ und K_{kl} ist $\mathcal{C} < \mathcal{D}$ oder L_{ij} ist $\mathcal{C} < \mathcal{D}$ und K_{kl} ist $\mathcal{A} = \mathcal{B}$, wobei $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ und $\mathcal{D}$ Namen von DE–Inkarnationen sind, und es gilt:
   $$\mathcal{A} =_N \mathcal{C} \text{ und } \mathcal{B} =_N \mathcal{D} \text{ oder } \mathcal{A} =_N \mathcal{D} \text{ und } \mathcal{B} =_N \mathcal{C}$$
   Begründung:
   Mit der üblichen Semantik für die auf den vordefinierten elementaren Datentypen definierten Vergleichsprädikate '=' und '<' ergibt sich unmittelbar: $L_{ij} \wedge K_{kl} \equiv false$.

10. L_{ij} ist $\mathcal{A} = \mathcal{B}$ und K_{kl} ist $\neg(\mathcal{C} = \mathcal{D})$ oder L_{ij} ist $\neg(\mathcal{C} = \mathcal{D})$ und K_{kl} ist $\mathcal{A} = \mathcal{B}$, wobei $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ und $\mathcal{D}$ Namen von DE–Inkarnationen sind, und es gilt:
    $$\mathcal{A} =_N \mathcal{C} \text{ und } \mathcal{B} =_N \mathcal{D} \text{ oder } \mathcal{A} =_N \mathcal{D} \text{ und } \mathcal{B} =_N \mathcal{C}$$
    Begründung:
    K_{kl} (bzw. L_{ij}) ist in diesem Fall Negation von L_{ij} (bzw. K_{kl}). Es gilt also: $L_{ij} \wedge K_{kl} \equiv false$.

11. L_{ij} ist $\mathcal{A} < \mathcal{B}$ und K_{kl} ist $\neg(\mathcal{C} < \mathcal{D})$ oder L_{ij} ist $\neg(\mathcal{C} < \mathcal{D})$ und K_{kl} ist $\mathcal{A} < \mathcal{B}$, wobei $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ und $\mathcal{D}$ Namen von DE–Inkarnationen sind, und es gilt:
    $$\mathcal{A} =_N \mathcal{C} \text{ und } \mathcal{B} =_N \mathcal{D}$$
    Begründung:
    K_{kl} (bzw. L_{ij}) ist in diesem Fall Negation von L_{ij} (bzw. K_{kl}). Es gilt also: $L_{ij} \wedge K_{kl} \equiv false$.

12. L_{ij} ist $\mathcal{A} < \mathcal{B}$ und K_{kl} ist $\mathcal{C} < \mathcal{D}$, wobei $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ und $\mathcal{D}$ Namen von DE–Inkarnationen sind, und es gilt:
    $$\mathcal{A} =_N \mathcal{D} \text{ und } \mathcal{B} =_N \mathcal{C}$$

Begründung:
Mit der üblichen Semantik für das Vergleichsprädikat '<' ergibt sich unmittelbar:
$L_{ij} \wedge K_{kl} \equiv false$.

△

Beispiel

Als Beispiel für die Analyse zweier Zugriffsrestriktionsausdrücke auf Widerspruchsfreiheit werden im folgenden die Zugriffsrestriktionsausdrücke der beiden DA–Generatorklassen `KontoAufloesen` und `Aufloesen` aus dem INSEL$^+$–Programm des Kontenverwaltungssystems betrachtet. Da $(\texttt{KontoAufloesen}, \texttt{Aufloesen}) \in \mathit{AnalysePairs}(KVS)$ gilt, sind diese beiden Zugriffsrestriktionsausdrücke auf Widerspruchsfreiheit zu überprüfen. Für das Beispiel wird davon ausgegegangen, daß der Zugriffsrestriktionsausdruck der DA–Generatorklasse `Aufloesen` keine Kontextrestriktion enthält. Die beiden hier betrachteten Zugriffsrestriktionsausdrücke lauten damit:

$$
\begin{aligned}
R_{\texttt{KontoAufloesen}} = \;& \texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\
& \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{KontoZeiger}, \texttt{Aufloesen}) \wedge \\
& (8 < \texttt{Zeit.Stunde} \vee 8 = \texttt{Zeit.Stunde}) \wedge \texttt{Zeit.Stunde} < 17 \wedge \\
& (1 < \texttt{Zeit.Wochentag} \vee 1 = \texttt{Zeit.Wochentag}) \wedge \texttt{Zeit.Wochentag} < 6
\end{aligned}
$$

$$
\begin{aligned}
R_{\texttt{Aufloesen}} = \;& \texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\
& \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{THIS}, \texttt{THIS}) \wedge \\
& (8 < \texttt{Zeit.Stunde} \vee 8 = \texttt{Zeit.Stunde}) \wedge \texttt{Zeit.Stunde} < 17 \wedge \\
& (1 < \texttt{Zeit.Wochentag} \vee 1 = \texttt{Zeit.Wochentag}) \wedge \texttt{Zeit.Wochentag} < 6
\end{aligned}
$$

Für die Analyse der Widerspruchsfreiheit sind $R_{\texttt{KontoAufloesen}}$ und $R_{\texttt{Aufloesen}}$ jeweils in ihre Disjunktive Normalform umzuwandeln:

$$
\begin{aligned}
DNF(R_{\texttt{KontoAufloesen}}) = \;& (\texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\
& \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{KontoZeiger}, \texttt{Aufloesen}) \wedge \\
& 8 < \texttt{Zeit.Stunde} \wedge \texttt{Zeit.Stunde} < 17 \wedge \\
& 1 < \texttt{Zeit.Wochentag} \wedge \texttt{Zeit.Wochentag} < 6) \vee \\
& (\texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\
& \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{KontoZeiger}, \texttt{Aufloesen}) \wedge \\
& 8 < \texttt{Zeit.Stunde} \wedge \texttt{Zeit.Stunde} < 17 \wedge \\
& 1 = \texttt{Zeit.Wochentag} \wedge \texttt{Zeit.Wochentag} < 6) \vee \\
& (\texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\
& \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{KontoZeiger}, \texttt{Aufloesen}) \wedge \\
& 8 = \texttt{Zeit.Stunde} \wedge \texttt{Zeit.Stunde} < 17 \wedge \\
& 1 < \texttt{Zeit.Wochentag} \wedge \texttt{Zeit.Wochentag} < 6) \vee \\
& (\texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\
& \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{KontoZeiger}, \texttt{Aufloesen}) \wedge \\
& 8 = \texttt{Zeit.Stunde} \wedge \texttt{Zeit.Stunde} < 17 \wedge \\
& 1 = \texttt{Zeit.Wochentag} \wedge \texttt{Zeit.Wochentag} < 6)
\end{aligned}
$$

$$
\begin{aligned}
DNF(R_{\texttt{Aufloesen}}) \quad = \quad & (\texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\
& \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{THIS}, \texttt{THIS}) \wedge \\
& 8 < \texttt{Zeit.Stunde} \wedge \texttt{Zeit.Stunde} < 17 \wedge \\
& 1 < \texttt{Zeit.Wochentag} \wedge \texttt{Zeit.Wochentag} < 6) \vee \\
& (\texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\
& \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{THIS}, \texttt{THIS}) \wedge \\
& 8 < \texttt{Zeit.Stunde} \wedge \texttt{Zeit.Stunde} < 17 \wedge \\
& 1 = \texttt{Zeit.Wochentag} \wedge \texttt{Zeit.Wochentag} < 6) \vee \\
& (\texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\
& \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{THIS}, \texttt{THIS}) \wedge \\
& 8 = \texttt{Zeit.Stunde} \wedge \texttt{Zeit.Stunde} < 17 \wedge \\
& 1 < \texttt{Zeit.Wochentag} \wedge \texttt{Zeit.Wochentag} < 6) \vee \\
& (\texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\
& \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{THIS}, \texttt{THIS}) \wedge \\
& 8 = \texttt{Zeit.Stunde} \wedge \texttt{Zeit.Stunde} < 17 \wedge \\
& 1 = \texttt{Zeit.Wochentag} \wedge \texttt{Zeit.Wochentag} < 6)
\end{aligned}
$$

Nun wird der erste konjunktive Teilterm L_1 von $DNF(R_{\texttt{KontoAufloesen}})$ mit dem erstem konjunktiven Teilterm K_1 von $DNF(R_{\texttt{Aufloesen}})$ auf Widersprüchlichkeit analysiert, indem überprüft wird, ob in diesen beiden konjunktiven Teilterme mindestens ein Paar widersprüchlicher Literale enthalten ist. Dabei ist:

$$
\begin{aligned}
L_1 = \quad & \texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\
& \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{KontoZeiger}, \texttt{Aufloesen}) \wedge \\
& 8 < \texttt{Zeit.Stunde} \wedge \texttt{Zeit.Stunde} < 17 \wedge \\
& 1 < \texttt{Zeit.Wochentag} \wedge \texttt{Zeit.Wochentag} < 6 \\
K_1 = \quad & \texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\
& \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{THIS}, \texttt{THIS}) \wedge \\
& 8 < \texttt{Zeit.Stunde} \wedge \texttt{Zeit.Stunde} < 17 \wedge \\
& 1 < \texttt{Zeit.Wochentag} \wedge \texttt{Zeit.Wochentag} < 6
\end{aligned}
$$

Wie leicht zu sehen ist, ist für jedes mögliche Paar (L_{1j}, K_{1l}) mit $j, l \in \{1, \ldots, 6\}$ von Literalen aus L_1 und K_1 keine der in Satz (5.4) angegebenen Bedingungen erfüllt. Damit sind jedoch L_1 und K_1 und somit auch $R_{\texttt{KontoAufloesen}}$ und $R_{\texttt{Aufloesen}}$ widerspruchsfrei.

Als Beispiel für ein Paar konjunktiver Teilterme von $DNF(R_{\texttt{KontoAufloesen}})$ und $DNF(R_{\texttt{Aufloesen}})$, die widersprüchlich sind, werden L_1 (siehe oben) und K_4 betrachtet.

$$
\begin{aligned}
K_4 = \quad & \texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\
& \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{THIS}, \texttt{THIS}) \wedge \\
& 8 = \texttt{Zeit.Stunde} \wedge \texttt{Zeit.Stunde} < 17 \wedge \\
& 1 = \texttt{Zeit.Wochentag} \wedge \texttt{Zeit.Wochentag} < 6
\end{aligned}
$$

Die Literale $L_{13} = (8 < \texttt{Zeit.Stunde})$ und $K_{43} = (8 = \texttt{Zeit.Stunde})$ sind gemäß Bedingung 9. aus Satz (5.4) widersprüchlich. Analog sind auch die Literale $L_{15} = (1 < \texttt{Zeit.Wochentag})$ und $K_{45} = (1 = \texttt{Zeit.Wochentag})$ widersprüchlich. Laut Definition (5.21) sind also die konjunktiven Teilterme L_1 und K_4 widersprüchlich.

$\diamond$

Die in Satz (5.3) eingehende Voraussetzung, daß R_2 keine Kontextrestriktionen enthält, ist für die Gültigkeit des Satzes und damit für die Anwendbarkeit des erklärten Analyseverfahrens wesentlich. Ist diese Voraussetzung nicht erfüllt, so kann es sein, daß zwar alle Paare konjunktiver Teilterme der Disjunktiven Normalformen gemäß der in den Sätzen (5.2) und (5.4) angegebenen Kriterien für die Widersprüchlichkeit zweier Literale widersprüchlich sind, die beiden Zugriffsrestriktionsausdrücke jedoch unter Berücksichtigung der möglichen Aufrufpfade zwischen den beiden Generatorklassen, für die die Zugriffsrestriktionsausdrücke festgelegt sind, widerspruchsfrei sind. Dies soll an einem Beispiel aus dem INSEL$^+$–Programm des Kontenverwaltungssystems verdeutlicht werden.

Beispiel

Betrachtet werden die beiden DA–Generatorklassen `KontoTyp` und `KontobewegungsListen-Typ`, deren Zugriffsrestriktionsausdruck jeweils eine Kontextrestriktion enthält.

$$\begin{aligned} R_{\texttt{KontoTyp}} = \; & \texttt{Caller.ConGen} = \texttt{KontoVerwalter.KontoEroeffnen} \wedge \\ & \texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\ & (8 < \texttt{Zeit.Stunde} \vee 8 = \texttt{Zeit.Stunde}) \wedge \texttt{Zeit.Stunde} < 17 \wedge \\ & (1 < \texttt{Zeit.Wochentag} \vee 1 = \texttt{Zeit.Wochentag}) \wedge \texttt{Zeit.Wochentag} < 6 \end{aligned}$$

$$R_{\texttt{KontobewegungsListenTyp}} = \texttt{Caller.ConGen} = \texttt{KontoTyp}$$

Da $(\texttt{KontoTyp}, \texttt{KontobewegungsListenTyp}) \in \mathit{AnalysePairs}(KVS)$ gilt, sind $R_{\texttt{KontoTyp}}$ und $R_{\texttt{KontobewegungsListenTyp}}$ auf Widerspruchsfreiheit zu analysieren. Das auf Satz (5.3) basierende Analyseverfahren würde in diesem Fall als Ergebnis liefern, daß $R_{\texttt{KontoTyp}}$ und $R_{\texttt{KontobewegungsListenTyp}}$ widersprüchlich sind, da jedes Paar konjunktiver Teilterme von $DNF(R_{\texttt{KontoTyp}})$ und $DNF(R_{\texttt{KontobewegungsListenTyp}})$ mit

$$\begin{aligned} L_{i1} &= \texttt{Caller.ConGen} = \texttt{KontoVerwalter.KontoEroeffnen} \quad \text{und} \\ K_{11} &= \texttt{Caller.ConGen} = \texttt{KontoTyp} \end{aligned}$$

gemäß Bedingung 11. aus Satz (5.2) zwei widersprüchliche Literale enthält.

Betrachtet man die Menge

$$AP(\texttt{KontoTyp}, \texttt{KontobewegungsListenTyp}) = \{(\texttt{KontoTyp}, \texttt{KontobewegungsListenTyp})\}$$

aller Aufrufpfade zwischen `KontoTyp` und `KontobewegungsListenTyp` so ergibt sich jedoch, daß $R_{\texttt{KontoTyp}}$ und $R_{\texttt{KontobewegungsListenTyp}}$ widerspruchsfrei sind, da die Kontextrestriktion `Caller.ConGen = KontoTyp`, aus der $R_{\texttt{KontobewegungsListenTyp}}$ besteht, für den Aufruf von *erzeuge* auf `KontobewegungsListenTyp` bei Ausführung der kanonischen Operation einer Inkarnation bzgl. `KontoTyp` immer erfüllt ist.

◇

Wie das Beispiel zeigt, sind Kontextrestriktionen bei der Analyse der Widerspruchsfreiheit zweier Zugriffsrestriktionsausdrücke R_1 und R_2 gesondert zu betrachten. Enthält R_2 Kontextrestriktionen, so ist zu überprüfen, ob diese Kontextrestriktionen unter Berücksichtigung eventuell in R_1 spezifizierter Kontextrestriktionen sowie der möglichen Aufrufpfade zwischen den DA–Generatorklassen G_1 und G_2, für die R_1 bzw. R_2 festgelegt sind, erfüllbar sind. Aus dieser Überprüfung läßt sich dann in Kombination mit der Überprüfung der bereits angegebenen Kriterien eine Aussage über die Widerspruchsfreiheit der beiden Zugriffsrestriktionsausdrücke machen. Die Analyse zweier Zugriffsrestriktionsausdrücke R_1 und R_2 auf Widerspruchsfreiheit für den Fall, daß R_2 Kontextrestriktionen enthält, wird im folgenden erklärt[22].

[22] Über das Vorhandensein von Kontextrestriktionen in R_1 wird hier zunächst keine Annahme getroffen, d.h. R_1 kann Kontextrestriktionen enthalten oder nicht.

R_2 enthält Kontextrestriktionen

R_1 und R_2 werden zunächst in ihre Disjunktive Normalform umgewandelt; dabei gelten die in (5.20) eingeführten Bezeichnungen. In den konjunktiven Teiltermen der Disjunktiven Normalform $DNF(R_2)$ von R_2 können aufgrund der Annahme, daß R_2 Kontextrestriktionen enthält, Literale auftreten, die Kontextrestriktionen spezifizieren. Diese Literale sind Gleichungen der Form

$$\begin{array}{lcl} \texttt{Caller.ConGen} & = & \mathcal{F}, \\ \texttt{Caller.ConName} & = & \mathcal{F}, \\ \texttt{Caller.ActorGen} & = & \mathcal{F}, \\ \texttt{Caller.ActorName} & = & \mathcal{F}, \end{array}$$

wobei $\mathcal{F}$ ein INSEL$^+$–Pfad ist. Analog können die konjunktiven Teilterme der Disjunktiven Normalform $DNF(R_1)$ von R_1 Kontextrestriktionen enthalten. Im weiteren wird davon ausgegangen, daß die konjunktiven Teilterme der Disjunktiven Normalformen in einer einheitlichen Form dargestellt werden. In dieser Form stehen die möglicherweise in einem konjunktiven Teilterm auftretenden Kontextrestriktionen „am Ende“ des Teilterms.

(5.22) Sei $L_i \triangleq \bigwedge_{j=1}^{n_i} L_{ij}$ der i-te konjunktive Teilterm der Disjunktiven Normalform eines Zugriffsrestriktionsausdrucks. Dann läßt sich L_i in folgender Form darstellen:

$$L_i = L_i^{Res} \wedge L_i^{Con}$$

mit:

- $L_i^{Res} \triangleq \bigwedge_{r \in \{r_1,\ldots,r_i\}} L_{ir}$, wobei $\{r_1, \ldots, r_i\} \subseteq \{1, \ldots, n_i\}$ die Menge der Indizes der Literale von L_i ist, die keine Kontextrestriktionen sind;
- $L_i^{Con} \triangleq \bigwedge_{c \in \{c_1,\ldots,c_i\}} L_{ic}$, wobei $\{c_1, \ldots, c_i\} \subseteq \{1, \ldots, n_i\}$ die Menge der Indizes der Literale von L_i ist, die Kontextrestriktionen sind.

Gilt $|\{r_1, \ldots, r_i\}| = 0$, d.h. besteht L_i nur aus Kontextrestriktionen, so wird dies verkürzt durch folgende Schreibweise ausgedrückt: $L_i^{Res} = \emptyset$. Analog wird für den Fall, daß $|\{c_1, \ldots, c_i\}| = 0$ gilt, L_i also keine Kontextrestriktionen enthält, abkürzend geschrieben: $L_i^{Con} = \emptyset$.

Da davon ausgegangen wird, daß die in diesem Abschnitt betrachteten Zugriffsrestriktionsausdrücke erfüllbar sind und damit jede Art von Kontextrestriktion höchstens einmal in einem konjunktiven Teilterm auftreten kann, gilt $|\{c_1, \ldots, c_i\}| \leq 4$. Ein konjunktiver Teilterm L_i läßt sich damit im allgemeinsten Fall wie folgt schreiben:

$$\begin{array}{rll} L_i = & L_i^{Res} \wedge & \texttt{Caller.ConGen} = \mathcal{F} \quad \wedge \texttt{Caller.ConName} = \mathcal{G} \wedge \\ & & \texttt{Caller.ActorGen} = \mathcal{H} \wedge \texttt{Caller.ActorName} = \mathcal{I} \end{array}$$

wobei $\mathcal{F}$, $\mathcal{G}$, $\mathcal{H}$, und $\mathcal{I}$ INSEL$^+$–Pfade sind. Wie in Abschnitt 4.4.4.3 erklärt, identifiziert ein INSEL$^+$–Pfad $\mathcal{F} \triangleq F_1.F_2.\ \ldots\ .F_i$ eindeutig entweder eine DA–Generatorklasse oder eine benannte DA–Inkarnationsklasse. Ein INSEL$^+$–Pfad kann also durch den gemäß Annahme (5.5) programmweit eindeutigen Bezeichner bzw. Namen der DA–Generatorklasse bzw. benannten DA–Inkarnationsklasse, die mit dem INSEL$^+$–Pfad identifiziert wird, ersetzt werden. Für das Weitere wird angenommen, daß die konjunktiven Teilterme der Disjunktiven Normalformen von R_1 und R_2 in der Form gemäß (5.22) vorliegen und die in ihnen auftretenden

INSEL$^+$–Pfade dem Gesagten entsprechend durch den eindeutigen Bezeichner der jeweils identifizierten DA–Generatorklasse bzw. benannten DA–Inkarnationsklasse ersetzt sind.

Analyseverfahren

R_1 und R_2 werden nun auf Widersprüchlichkeit analysiert, indem alle möglichen Paare konjunktiver Teilterme ihrer Disjunktiven Normalformen unter Berücksichtigung der Kontextrestriktionen auf Widersprüchlichkeit überprüft werden. Seien also L_i mit $i \in \{1, \ldots, m_1\}$ ein konjunktiver Teilterm von $DNF(R_1)$ und K_k mit $k \in \{1, \ldots, m_2\}$ ein konjunktiver Teilterm von $DNF(R_2)$. Bei der Analyse von L_i und K_k auf Widersprüchlichkeit ist zu unterscheiden, ob K_k Kontextrestriktionen enthält oder nicht.

1. Fall: K_k enthält keine Kontextrestriktionen, d.h. es gilt: $K_k^{Con} = \emptyset$

In diesem Fall wird anhand der in Satz (5.4) angegebenen Kriterien überprüft, ob die konjunktiven Teilterme L_i und K_k mindestens ein Paar widersprüchlicher Literale enthalten. Ist dies der Fall, so sind L_i und K_k widersprüchlich. Anderenfalls sind L_i und K_k und damit auch R_1 und R_2 widerspruchsfrei.

2. Fall: K_k enthält Kontextrestriktionen

O.b.d.A sei: $K_k^{Con} \triangleq \texttt{Caller.ConGen} = G \wedge \texttt{Caller.ConName} = N \wedge$
$\texttt{Caller.ActorGen} = AG \wedge \texttt{Caller.ActorName} = AN$
wobei G und AG die eindeutigen Bezeichner der DA–Generatorklassen sowie N und AN die eindeutigen Bezeichner der benannten DA–Inkarnationsklassen sind, die mit den entsprechenden INSEL$^+$–Pfaden identifiziert werden.

Es wird zunächst analog zum 1. Fall überprüft, ob L_i^{Res} und K_k^{Res} widersprüchlich sind. Ist dies der Fall, so sind die konjunktiven Teilterme L_i und K_k widersprüchlich und die Analyse von L_i und K_k ist beendet. Im anderen Fall, d.h. wenn L_i^{Res} und K_k^{Res} kein Paar widersprüchlicher Literale enthalten, ist zu analysieren, ob die in K_k^{Con} spezifizierten Kontextrestriktionen unter Berücksichtigung der möglicherweise in L_i^{Con} festgelegten Kontextrestriktionen erfüllbar sind. Dazu ist zu überprüfen, ob es zwischen den DA–Generatorklassen G_1 und G_2, für die R_1 und R_2 festgelegt sind, einen Aufrufpfad gibt, mit dem die Kontextrestriktionen erfüllbar sind. Basis für diese Überprüfung ist der in Abschnitt 5.3.1 erklärte Zusammenhang zwischen den in Definition (5.10) definierten Pfad–Abbildungen und dem Kontextattribut der Akteure.

Bei der Analyse der Erfüllbarkeit der Kontextrestriktionen sind neben den Aufrufpfaden zwischen den DA–Generatorklassen G_1 und G_2 auch Aufrufpfade zu berücksichtigen, die eine benannte DA–Inkarnationsklasse, die sich auf G_1 bezieht, als Anfangsknoten und/oder eine benannte DA–Inkarnationsklasse, die sich auf G_2 bezieht, als Endknoten haben. Diese Notwendigkeit ergibt sich bereits daraus, daß es aufgrund der Definition des attributierten statischen Aufrufgraphen (vgl. Definition (5.8)) Fälle geben kann, in denen die Menge der Aufrufpfade zwischen den DA–Generatorklassen G_1 und G_2 leer ist; die G_1 bzw. G_2 zugeordneten Zugriffsrestriktionsausdrücke jedoch auf Widerspruchsfreiheit zu analysieren sind, da es einen Aufrufpfad zwischen G_1 oder einer sich auf G_1 beziehenden benannten DA–Inkarnationsklasse und einer benannten DA–Inkarnationsklasse, die sich auf G_2 bezieht, gibt (vgl. hierzu auch die Definitionen (5.18) und (5.19)).

Wie in Abschnitt 5.3.1 erwähnt, kann der statische Aufrufgraph eines INSEL$^+$–Programms Zyklen enthalten. Die Menge aller Aufrufpfade zwischen zwei Knoten v_1 und v_2 des Aufrufgraphen, die gemäß Definition (5.9) mit $AP(v_1, v_2)$ bezeichnet wird, ist also im allgemeinen nicht endlich. Müßten also im Rahmen der Analyse der Erfüllbarkeit der Kontextrestriktionen immer alle Pfade zwischen den jeweiligen Knoten überprüft werden, würde dies bedeuten,

daß diese Analyse im allgemeinen nicht terminieren würde. Für die Überprüfung der Erfüllbarkeit der Kontextrestriktionen genügt es jedoch, bis auf einen Ausnahmefall, der weiter unten erklärt wird, zyklenfreie Aufrufpfade zu analysieren, da sich jeder zyklische Aufrufpfad $p_1 \in AP(v_1, v_2)$ auf einen zyklenfreien Aufrufpfad $p_2 \in AP_{zf}(v_1, v_2)$ so abbilden läßt, daß der Wert der in Definition (5.10) definierten Pfad–Abbildungen für p_1 und p_2 gleich ist; p_1 und p_2 bzgl. der Pfad–Abbildungen also äquivalent sind. Da die Menge $AP_{zf}(v_1, v_2)$ der zyklenfreien Aufrufpfade zwischen zwei Knoten v_1 und v_2 endlich ist, ist die Terminierung der Analyse unter dem Gesichtspunkt der Anzahl zu überprüfender Aufrufpfade somit gewährleistet.

Auf einen formalen Beweis der obigen Aussage, daß sich jeder zyklische Aufrufpfad auf einen bzgl. der Werte der Pfad–Abbildungen äquivalenten zyklenfreien Aufrufpfad abbilden läßt, sowie die formale Angabe einer entsprechenden Abbildung wird hier verzichtet. Informell gesprochen, ergibt sich der zu einem zyklischen Aufrufpfad p_1 zwischen zwei Knoten v_1 und v_2 bzgl. der Pfad–Abbildungen äquivalente zyklenfreie Aufrufpfad $p_2 \in AP_{zf}(v_1, v_2)$ dadurch, daß in p_1 alle Zyklen "rausgestrichen" werden. Dies soll an einem kleinen Beispiel verdeutlicht werden.

Beispiel

Gegeben sei der in Abbildung 5.7 dargestellte Ausschnitt aus dem attributierten statischen Aufrufgraph eines nicht näher spezifizierten INSEL$^+$–Programms[23].

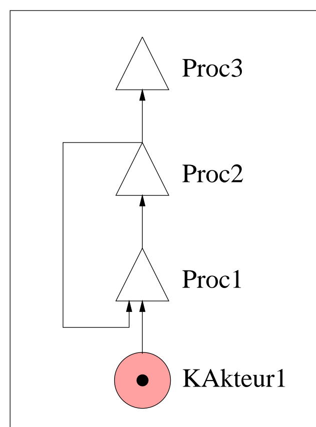

Abbildung 5.7: Ausschnitt aus dem statischen Aufrufgraph eines INSEL$^+$–Programms

Betrachtet wird der folgende zyklische Aufrufpfad zwischen der benannten K-Akteur–Inkarnationsklasse `KAkteur1` und der S-Order–Generatorklasse `Proc3`:

$$p_1 = (\texttt{KAkteur1}, \texttt{Proc1}, \texttt{Proc2}, \texttt{Proc1}, \texttt{Proc2}, \texttt{Proc3})$$

Für die Werte der Pfad–Abbildungen von p_1 gilt:

$$\begin{aligned} lcg(p_1) &= \texttt{Proc2} \\ lcn(p_1) &= \perp \\ lak(p_1) &= \texttt{KAkteur1} \\ lag(p_1) &= \texttt{KAkteur1} \\ lan(p_1) &= \{\texttt{KAkteur1}\} \end{aligned}$$

In p_1 ist der Zyklus (`Proc1`, `Proc2`, `Proc1`) enthalten. Streicht man nun hieraus die den Zyklus verursachenden letzten beiden Knoten, so ergibt sich aus p_1 der zyklenfreie Aufrufpfad

[23] Das INSEL$^+$–Programm des Kontenverwaltungssystems kann an dieser Stelle nicht als Beispiel herangezogen werden, da dessen Aufrufgraph keine Zyklen enthält.

$$p_2 = (\texttt{KAkteur1}, \texttt{Proc1}, \texttt{Proc2}, \texttt{Proc3})$$

Wie unmittelbar ersichtlich ist, sind die Werte der Pfad–Abbildungen für p_1 und p_2 gleich, was bedeutet, daß p_1 und p_2 bzgl. der Pfad–Abbildungen äquivalent sind.

◇

Der oben erwähnte eine Ausnahmefall liegt dann vor, wenn G_1 gleich G_2 und damit auch R_1 gleich R_2 ist. Die Notwendigkeit für die Analyse der Widerspruchsfreiheit von R_1 und R_2 ergibt sich hierbei daraus, daß es einen Zyklus mit G_1 als Anfangs– und Endknoten gibt.[24] In diesem Fall sind alle Zyklen, die G_1 als Anfangs– und Endknoten haben und die keinen weiteren Zyklus enthalten, zu untersuchen. Da die Menge dieser Zyklen endlich ist, ist auch in diesem Fall die Terminierung der Analyse der Erfüllbarkeit der Kontextrestriktionen gewährleistet. Der Ausnahmefall unterscheidet sich also von den anderen Fällen lediglich darin, welche Aufrufpfade zu überprüfen sind. Die Analyse der einzelnen Aufrufpfade erfolgt in beiden Fällen analog.

Beispiel (Fortsetzung)

In dem vorhergehenden Beispiel liegen Ausnahmefälle für die S-Order–Generatorklassen `Proc1` und `Proc2` vor, da es Zyklen gibt, die `Proc1` oder `Proc2` als Anfangs– und Endknoten haben. Für `Proc1` ist im Rahmen der Analyse der Erfüllbarkeit der Kontextrestriktionen der Aufrufpfad $p_3 = (\texttt{Proc1}, \texttt{Proc2}, \texttt{Proc1})$ und für `Proc2` der Aufrufpfad $p_4 = (\texttt{Proc2}, \texttt{Proc1}, \texttt{Proc2})$ zu untersuchen.

◇

In der folgenden Definition wird nun die Menge der Aufrufpfade formal präzisiert, die im Rahmen der Überprüfung der Erfüllbarkeit der in K_k^{Con} spezifizierten Kontextrestriktionen zu analysieren sind. Diese Menge wird mit $\mathit{AnalysePaths}(G_1, G_2)$ bezeichnet. Die obigen Ausführungen sind Basis für das Verständnis der Definition. Anschließend werden Eigenschaften angegeben, die ein Aufrufpfad aus $\mathit{AnalysePaths}(G_1, G_2)$ haben muß, damit mit ihm die in K_k^{Con} enthaltenen Kontextrestriktionen erfüllbar sind. Gibt es mindestens einen Aufrufpfad in $\mathit{AnalysePaths}(G_1, G_2)$, der diesen Eigenschaften genügt, so sind die Kontextrestriktionen potentiell erfüllbar und damit sind die konjunktiven Teilterme L_i und K_k widerspruchsfrei. Wenn es keinen Pfad mit diesen Eigenschaften gibt, folgt daraus, daß die in K_k^{Con} spezifizierten Kontextrestriktionen nicht erfüllbar sind. In diesem Fall sind die konjunktiven Teilterme L_i und K_k widersprüchlich.

Definition 5.23.: Menge zu analysierender Aufrufpfade (*AnalysePaths*)

Gegeben seien ein INSEL⁺–Programm $\mathcal{P} = (\mathcal{D}, \delta, \gamma, \iota)$ und der attributierte statische Aufrufgraph $\mathcal{G}(\mathcal{P}) = (V, K, \mu)$ von $\mathcal{P}$. G_1 und G_2 seien zwei DA–Generatorklassen mit $(G_1, G_2) \in \mathit{AnalysePairs}(\mathcal{P})$, und R_1 und R_2 seien die G_1 bzw. G_2 zugeordneten Zugriffsrestriktionsausdrücke. R_2 enthalte Kontextrestriktionen. K_k^{Con} sei wie in (5.22) gegeben.

Die Menge $\mathit{AnalysePaths}(G_1, G_2)$ der Aufrufpfade, die im Rahmen der Überprüfung der Erfüllbarkeit der in K_k^{Con} spezifizierten Kontextrestriktionen zu analysieren sind,

[24] Anmerkung: In diesem Fall ist im Rahmen der Analyse der Widerspruchsfreiheit von R_1 und R_1 lediglich zu überprüfen, ob die in K_k^{Con} spezifizierten Kontextrestriktionen erfüllbar sind. Sonstige Widersprüche sind aufgrund der postulierten Erfüllbarkeit von R_1 nicht vorhanden.

ist wie folgt definiert:

$$AnalysePaths(G_1, G_2) \triangleq \begin{cases} AP_{zf}(G_1, G_2) \cup \\ \bigcup_{v \in NInk(G_2)} AP_{zf}(G_1, v) \cup \\ \bigcup_{v \in NInk(G_1)} AP_{zf}(v, G_2) \cup & \text{falls } G_1 \neq G_2 \\ \bigcup_{v_1 \in NInk(G_1)} \bigcup_{v_2 \in NInk(G_2)} AP_{zf}(v_1, v_2) \\ \{p \in AP(G_1, G_1) \mid p = (G_1, v_1, \ldots, v_n, G_1), & \text{falls } G_1 = G_2 \\ \quad n \geq 0 \text{ und } (v_1, \ldots, v_n) \text{ ist zyklenfrei}\} \end{cases}$$

wobei die Abbildung $NInk$ einer DA–Generatorklasse G die Menge der benannten DA–Inkarnationsklassen zuordnet, die sich auf G beziehen:

$$NInk : \mathcal{D}^{DAG} \longrightarrow POT(\mathcal{D}^{NDAI})$$

$$NInk(G) \triangleq \begin{cases} \{n \in \mathcal{D}^{NDAI} \mid (G, n) \in \gamma\} & \text{falls } G \text{ K-Akteur– oder} \\ & \text{Depot-Generatorklasse ist} \\ \emptyset & \text{falls } G \text{ M-Akteur– oder} \\ & \text{Order-Generatorklasse ist} \end{cases}$$

□

Wie die in Definition (5.23) eingeführte Menge $AnalysePaths(G_1, G_2)$ anhand des statischen Aufrufgraphen bestimmt wird, wird an einem Beispiel aus dem INSEL$^+$–Programm des Kontenverwaltungssystems erläutert.

Beispiel

Da (`KontoVerwalterTyp`, `FuegeEin`) $\in$ *AnalysePairs*(*KVS*) gilt (vgl. Beispiel auf Seite 192), sind die Zugriffsrestriktionsausdrücke der DA–Generatorklassen `KontoVerwalterTyp` und `FuegeEin` auf Widerspruchsfreiheit zu analysieren. Der Zugriffsrestriktionsausdruck von `FuegeEin` enthält mit `Caller.ConGen = Einzahlen` und `Caller.ConGen = Abheben` zwei Kontextrestriktionen.

In Abbildung 5.8 ist der für die Angabe der Menge

AnalysePaths(`KontoVerwalterTyp`, `FuegeEin`)

relevante Ausschnitt aus dem statischen Aufrufgraph des Kontenverwaltungssystems angegeben (vgl. auch Abbildung 5.5). Es gilt:

AnalysePaths(`KontoVerwalterTyp`, `FuegeEin`) =
{(`KontoVerwalterTyp`, `Zinsberechnung`, `Zinsberechner`, `Abheben`, `FuegeEin`),
(`KontoVerwalterTyp`, `Zinsberechnung`, `Zinsberechner`, `Einzahlen`, `FuegeEin`),
(`KontoVerwalter`, `Zinsberechnung`, `Zinsberechner`, `Abheben`, `FuegeEin`),
(`KontoVerwalter`, `Zinsberechnung`, `Zinsberechner`, `Einzahlen`, `FuegeEin`)}

◇

Im allgemeinen müssen im Rahmen der Überprüfung der Erfüllbarkeit der in K_k^{Con} festgelegten Kontextrestriktionen nicht alle in der Menge $AnalysePaths(G_1, G_2)$ enthaltenen Aufrufpfade analysiert werden. Eine Möglichkeit, die Menge der tatsächlich zu untersuchenden Aufrufpfade einzuschränken, besteht darin, die in $AnalysePaths(G_1, G_2)$ enthaltenen Pfade in Äquivalenzklassen einzuteilen. Kriterium für die Äquivalenz zweier Aufrufpfade ist dabei die *Pfadabbildungs–Äquivalenz*: zwei Aufrufpfade sind *pfadabbildungs–äquivalent*, wenn

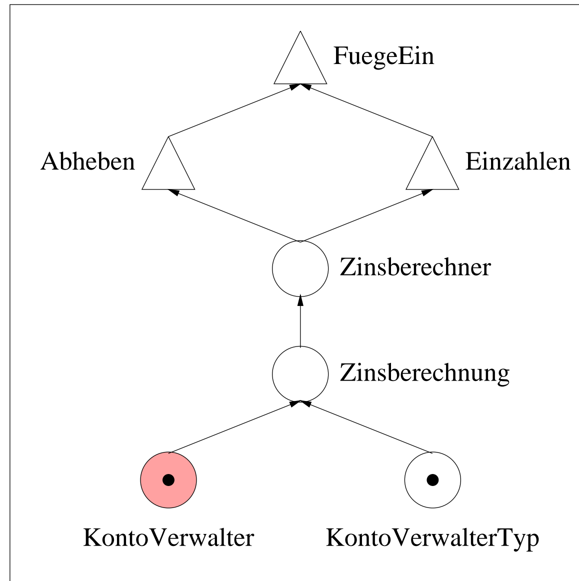

Abbildung 5.8: Ausschnitt aus dem statischen Aufrufgraph des Kontenverwaltungssystems zur Erläuterung der Definition (5.23)

sie bzgl. der in Definition (5.10) definierten Pfad–Abbildungen den gleichen Wert haben. Im Rahmen der Überprüfung der Erfüllbarkeit der Kontextrestriktionen genügt es dann, für jede Äquivalenzklasse genau einen Repräsentanten zu analysieren.

Die Menge der zu untersuchenden Äquivalenzklassen bzw. der entsprechenden Repräsentanten kann dann in Abhängigkeit von den in K_k^{Con} enthaltenen Arten von Kontextrestriktionen ggf. noch weiter eingeschränkt werden. Enthält K_k^{Con} z.B. keine Kontextrestriktion der Form `Caller.ActorName` $= AN$, so brauchen die Äquivalenzklassen, deren Pfade durch die Pfad–Abbildung lan auf eine nicht leere Menge abgebildet werden, nicht analysiert zu werden.

Beispiel (Fortsetzung)

Für das obige Beispiel ergeben sich folgende zwei Äquivalenzklassen:

$$\begin{aligned}
A_1 &= \{(\texttt{KontoVerwalterTyp}, \texttt{Zinsberechnung}, \texttt{Zinsberechner}, \texttt{Abheben}, \texttt{FuegeEin}),\\
&\quad (\texttt{KontoVerwalter}, \texttt{Zinsberechnung}, \texttt{Zinsberechner}, \texttt{Abheben}, \texttt{FuegeEin})\}\\
A_2 &= \{(\texttt{KontoVerwalterTyp}, \texttt{Zinsberechnung}, \texttt{Zinsberechner}, \texttt{Einzahlen}, \texttt{FuegeEin}),\\
&\quad (\texttt{KontoVerwalter}, \texttt{Zinsberechnung}, \texttt{Zinsberechner}, \texttt{Einzahlen}, \texttt{FuegeEin})\}
\end{aligned}$$

Weitere Möglichkeiten zur Einschränkung der Menge der zu überprüfenden Aufrufpfade bestehen in diesem Beispiel nicht. Für jede in dem Zugriffsrestriktionsausdruck von `FuegeEin` enthaltene Kontextrestriktion sind somit zwei Aufrufpfade zu analysieren.

◇

Die angesprochenen Möglichkeiten für die Einschränkung der Menge der zu analysierenden Aufrufpfade werden hier nicht weiter präzisiert, da diese lediglich unter dem Gesichtspunkt der Aufwandsreduzierung für die durchzuführenden Analysen von Bedeutung sind.

Für das Weitere sei $p \in AnalysePaths(G_1, G_2)$ ein Aufrufpfad, für den zu überprüfen ist, ob mit ihm die in K_k^{Con} spezifizierten Kontextrestriktionen erfüllbar sind. Die einzelnen durch die unterschiedlichen Gleichungen in K_k^{Con} festgelegten Kontextrestriktionen (vgl. Seite 204)

werden nacheinander getrennt auf Erfüllbarkeit mit dem Pfad p überprüft. Sind alle einzelnen Kontextrestriktionen mit dem Pfad p erfüllbar, so sind die in K_k^{Con} spezifizierten Kontextrestriktionen insgesamt mit p erfüllbar.

1. Erfüllbarkeit der mit `Caller.ConGen` spezifizierten Kontextrestriktion

Die in K_k^{Con} durch die Gleichung `Caller.ConGen` $= G$ spezifizierte Kontextrestriktion ist mit dem Aufrufpfad p erfüllbar, wenn gilt: $lcg(p) = G$. In diesem Fall kann es nach dem in Abschnitt 5.3.1 erklärten Zusammenhang zwischen den Pfad–Abbildungen und den Kontextattributen der Akteure zur Laufzeit einen Akteur a geben, der im Kontext der Ausführung der kanonischen Operation einer Inkarnation bzgl. eines Generators der Generatorklasse G_1 die kanonische Operation einer Inkarnation bzgl. eines Generators der Generatorklasse G ausführt und dabei die *erzeuge*–Operation auf einem Generator der Generatorklasse G_2 aufruft. Da der Wert des Elements $ConGen_t(a)$ des Kontextattributs des Akteurs a zum Zeitpunkt t dieses Aufrufs gleich G ist, wird die bei diesem Aufruf auszuwertende Gleichung `Caller.ConGen` $= G$ zu *true* ausgewertet. Die durch `Caller.ConGen` $= G$ spezifizierte Kontextrestriktion ist also erfüllbar.

Beispiel (Fortsetzung)

Die in dem Zugriffsrestriktionsausdruck von `FuegeEin` enthaltene Kontextrestriktion `Caller.ConGen = Einzahlen` ist mit dem Aufrufpfad

$$p = (\texttt{KontoVerwalterTyp}, \texttt{Zinsberechnung}, \texttt{Zinsberechner}, \texttt{Einzahlen}, \texttt{FuegeEin})$$

erfüllbar, da gilt: $lcg(p) =$ `Einzahlen`.

◇

2. Erfüllbarkeit der mit `Caller.ConName` spezifizierten Kontextrestriktion

Die in K_k^{Con} durch die Gleichung `Caller.ConName` $= N$ spezifizierte Kontextrestriktion ist mit dem Aufrufpfad p erfüllbar, wenn gilt: $lcn(p) = N$. In diesem Fall kann es zur Laufzeit einen Akteur a geben, der im Kontext der Ausführung der kanonischen Operation einer Inkarnation bzgl. eines Generators der Generatorklasse G_1 die kanonische Operation einer benannten DA–Inkarnation der Inkarnationsklasse N ausführt und dabei die *erzeuge*–Operation auf einem Generator der Generatorklasse G_2 aufruft. Für den Wert des Elements $ConName_t(a)$ des Kontextattributs des Akteurs a zum Zeitpunkt t dieses Aufrufs gilt: $ConName_t(a) = N$. Daraus folgt, daß der Wert der bei diesem Aufruf auszuwertenden Gleichung `Caller.ConName` $= N$ gleich *true* ist und damit die durch `Caller.ConName` $= N$ spezifizierte Kontextrestriktion erfüllbar ist.

3. Erfüllbarkeit der mit `Caller.ActorGen` spezifizierten Kontextrestriktion

Abhängig davon, ob die Pfad–Abbildung $lag(p)$ eine Akteur–Generatorklasse oder $\perp$ als Ergebnis liefert, muß der Aufrufpfad p unterschiedlichen Bedingungen genügen, damit mit ihm die durch die Gleichung `Caller.ActorGen` $= AG$ spezifizierte Kontextrestriktion erfüllbar ist.

1. Fall: $lag(p) \neq \perp$

In diesem Fall ist die in K_k^{Con} durch die Gleichung `Caller.ActorGen` $= AG$ spezifizierte Kontextrestriktion mit dem Aufrufpfad p erfüllbar, wenn gilt: $lag(p) = AG$. Ist diese Bedingung erfüllt, kann es zur Laufzeit einen Akteur a geben, der Inkarnation bzgl. eines Generators der Akteur–Generatorklasse AG ist und der im Kontext der Ausführung der kanonischen Operation einer Inkarnation bzgl. eines Generators der Generatorklasse G_1 die *erzeuge*–Operation auf einem Generator der Generatorklasse G_2 aufruft. Da der Wert des Elements $ActorGen_t(a)$ des Kontextattributs des Akteurs a zum Zeitpunkt

t dieses Aufrufs gleich AG ist, wird die bei diesem Aufruf auszuwertende Gleichung `Caller.ActorGen` $= AG$ zu *true* ausgewertet. Die durch `Caller.ActorGen` $= AG$ spezifizierte Kontextrestriktion ist also erfüllbar.

2. Fall: $lag(p) = \perp$
In diesem Fall ist weiter zu unterscheiden, ob in L_i^{Con} eine Kontextrestriktion der Form `Caller.ActorGen` $= OG$ enthalten ist oder nicht.

2.1.: L_i^{Con} enthält keine Kontextrestriktion der Form `Caller.ActorGen` $= OG$

In diesem Fall ist zu untersuchen, ob es zur Laufzeit einen Akteur a geben kann, der Inkarnation bzgl. eines Generators der Akteur–Generatorklasse AG ist und der die kanonische Operation einer Inkarnation o bzgl. eines Generators der Generatorklasse G_1 ausführt. Gibt es einen solchen Akteur a, so kann dieser bei Ausführung der kanonischen Operation von o die *erzeuge*–Operation auf einem Generator der Generatorklasse G_2 aufrufen. Für den Wert des Elements $ActorGen_t(a)$ des Kontextattributs von a zum Zeitpunkt t dieses Aufrufs gilt dann: $ActorGen_t(a) = AG$. Die bei diesem Aufruf auszuwertende Gleichung `Caller.ActorGen` $= AG$ wird also zu *true* ausgewertet, woraus folgt, daß die durch `Caller.ActorGen` $= AG$ spezifizierte Kontextrestriktion erfüllbar ist.
Einen Akteur, der Inkarnation bzgl. eines Generators der Akteur–Generatorklasse AG ist und der die kanonische Operation einer Inkarnation bzgl. eines Generators der Generatorklasse G_1 ausführt, kann es genau dann geben, wenn es in dem statischen Aufrufgraph entweder

- einen Aufrufpfad p_1 mit G_1 als Endknoten gibt, für den gilt: $lag(p_1) = AG$ oder
- einen Aufrufpfad p_2 mit einer benannten DA–Inkarnationsklasse, die sich auf G_1 bezieht, als Endknoten gibt, für den gilt: $lag(p_2) = AG$.

Die Menge aller Aufrufpfade, die zu einer DA–Generatorklasse oder zu einer benannten Inkarnationsklasse, die sich auf diese DA–Generatorklasse bezieht, führen, wird im folgenden definiert.

Definition 5.24.: Menge *PathsTo(G)*
Gegeben seien der attributierte statische Aufrufgraph $\mathcal{G}(\mathcal{P}) = (V, K, \mu)$ eines INSEL$^+$–Programms $\mathcal{P}$ und eine DA–Generatorklasse $G \in \mathcal{D}^{DAG}$. Dann ist die Menge $PathsTo(G)$ wie folgt definiert:

$$PathsTo(G) \triangleq \bigcup_{x \in V} AP(x, G) \quad \cup \quad \bigcup_{y \in NInk(G)} \; \bigcup_{x \in V} AP(x, y)$$

□

Ist G eine DA–Generatorklasse eines INSEL$^+$-Programms, so enthält *PathsTo(G)* gemäß Definition (5.24) alle Pfade des statischen Aufrufgraphen, die G oder eine benannte DA–Inkarnationsklasse, die sich auf G bezieht, als Endknoten haben.

Nach dem oben Gesagten ist also die in K_k^{Con} durch die Gleichung `Caller.ActorGen` $=$ AG spezifizierte Kontextrestriktion mit dem Aufrufpfad p erfüllbar, wenn es mindestens einen Pfad $w \in PathsTo(G_1)$ gibt, für den gilt: $lag(w) = AG$.

2.2.: L_i^{Con} enthält eine Kontextrestriktion der Form `Caller.ActorGen` $= OG$

In diesem Fall müssen die durch die beiden Gleichungen `Caller.ActorGen` $= OG$ und `Caller.ActorGen` $= AG$ spezifizierten Kontextrestriktionen gleich sein, d.h. es muß gelten: $AG = OG$. Wenn diese Bedingung erfüllt ist, ist anschließend analog zu 2.1. zu untersuchen, ob es zur Laufzeit einen Akteur a geben kann, der Inkarnation bzgl. eines Generators der Akteur–Generatorklasse AG ist und der die kanonische Operation einer Inkarnation bzgl. eines Generators der Generatorklasse G_1 ausführt. Die in K_k^{Con} durch die Gleichung `Caller.ActorGen` $= AG$ spezifizierte Kontextrestriktion ist also mit dem Aufrufpfad p erfüllbar, wenn $AG = OG$ gilt und es mindestens einen Pfad $w \in PathsTo(G_1)$ gibt, für den gilt: $lag(w) = AG$.

Beispiel

In dem Kontenverwaltungssystem sind die Zugriffsrestriktionsausdrücke der DA-Generatorklassen `KontoVerwalterTyp` und `Einzahlen` auf Widerspruchsfreiheit zu analysieren. Die Disjunktive Normalform des Zugriffsrestriktionsausdrucks von `Einzahlen` enthält u.a. den konjunktiven Teilterm `Caller.ActorGen = Zinsberechner`. Für die Menge der Aufrufpfade, die im Rahmen der Analyse der Erfüllbarkeit dieser Kontextrestriktion zu untersuchen sind, gilt (vgl. auch Abbildung 5.8):

$$\begin{aligned} &AnalysePaths(\texttt{KontoVerwalterTyp}, \texttt{Einzahlen}) = \\ &\quad \{(\texttt{KontoVerwalterTyp}, \texttt{Zinsberechnung}, \texttt{Zinsberechner}, \texttt{Einzahlen}), \\ &\quad (\texttt{KontoVerwalter}, \texttt{Zinsberechnung}, \texttt{Zinsberechner}, \texttt{Einzahlen})\} \end{aligned}$$

Die Kontextrestriktion `Caller.ActorGen = Zinsberechner` ist mit beiden in *AnalysePaths*(`KontoVerwalterTyp, Einzahlen`) enthaltenen Aufrufpfaden erfüllbar, da für diese gilt: $lag(p) =$ `Zinsberechner`.

◇

4. Erfüllbarkeit der mit `Caller.ActorName` spezifizierten Kontextrestriktion

Analog zur Überprüfung der Erfüllbarkeit der mit `Caller.ActorGen` spezifizierten Kontextrestriktion ist hier zu unterscheiden, ob die Pfad–Abbildung $lan(p)$ eine Menge benannter K-Akteur–Inkarnationsklassen oder die leere Menge als Ergebnis liefert. In den beiden Fällen muß der Aufrufpfad p unterschiedlichen Bedingungen genügen, damit mit ihm die durch die Gleichung `Caller.ActorName` $= AN$ spezifizierte Kontextrestriktion erfüllbar ist.

1. Fall: $lan(p) \neq \emptyset$

In diesem Fall ist die in K_k^{Con} durch die Gleichung `Caller.ActorName` $= AN$ spezifizierte Kontextrestriktion mit dem Aufrufpfad p erfüllbar, wenn gilt: $AN \in lan(p)$. Ist diese Bedingung erfüllt, kann es zur Laufzeit einen benannten K-Akteur an der Inkarnationsklasse AN geben, der im Kontext der Ausführung der kanonischen Operation einer Inkarnation bzgl. eines Generators der Generatorklasse G_1 die *erzeuge*–Operation auf einem Generator der Generatorklasse G_2 aufruft. Für den Wert des Elements $ActorName_t(an)$ des Kontextattributs des K-Akteurs an zum Zeitpunkt t dieses Aufrufs gilt dann: $ActorName_t(an) = AN$. Daraus folgt, daß der Wert der bei diesem Aufruf auszuwertenden Gleichung `Caller.ActorName` $= AN$ gleich *true* ist und damit die durch `Caller.ActorName` $= AN$ spezifizierte Kontextrestriktion erfüllbar ist.

2. Fall: $lan(p) = \emptyset$

In diesem Fall ist zunächst zu überprüfen, ob die Pfad–Abbildung $lag(p)$ eine Akteur–Generatorklasse als Ergebnis liefert. Ist dies der Fall, so ist die durch `Caller.ActorName` $= AN$ spezifizierte Kontextrestriktion mit p nicht erfüllbar, da

es zwar laut p einen Akteur geben kann, der die *erzeuge*–Operation auf einem Generator der Generatorklasse G_2 aufruft, dieser jedoch nicht benannt ist. Gilt $lag(p) = \perp$, so ist weiter zu unterscheiden, ob in L_i^{Con} eine Kontextrestriktion der Form `Caller.ActorName` $= ON$ enthalten ist oder nicht.

2.1.: L_i^{Con} enthält eine Kontextrestriktion der Form `Caller.ActorName` $= ON$

In diesem Fall müssen die durch die beiden Gleichungen `Caller.ActorName` $= ON$ und `Caller.ActorName` $= AN$ spezifizierten Kontextrestriktionen gleich sein, d.h. es muß gelten: $AN = ON$. Wenn diese Bedingung erfüllt ist, ist zu untersuchen, ob es zur Laufzeit einen benannten K-Akteur an der Inkarnationsklasse AN geben kann, der die kanonische Operation einer Inkarnation o bzgl. eines Generators der Generatorklasse G_1 ausführt. Gibt es einen solchen K-Akteur an, so kann dieser im Kontext der Ausführung der kanonischen Operation von o die *erzeuge*–Operation auf einem Generator der Generatorklasse G_2 aufrufen. Für den Wert des Elements $ActorName_t(an)$ des Kontextattributs des K-Akteurs an zum Zeitpunkt t dieses Aufrufs gilt dann: $ActorName_t(an) = AN$, woraus folgt, daß die durch `Caller.ActorName` $= AN$ spezifizierte Kontextrestriktion erfüllbar ist. Einen benannten K-Akteur an der Inkarnationsklasse AN, der die kanonische Operation einer Inkarnation bzgl. eines Generators der Generatorklasse G_1 ausführt, kann es genau dann geben, wenn es mindestens eine Aufrufpfad $w \in PathsTo(G_1)$ gibt, für den gilt: $lan(w) = AN$.

2.2.: L_i^{Con} enthält keine Kontextrestriktion der Form `Caller.ActorName` $= ON$

In diesem Fall ist weiter zu unterscheiden, ob L_i^{Con} eine Gleichung der Form `Caller.ActorGen` $= OG$ enthält oder nicht. Ist in L_i^{Con} eine solche Gleichung enthalten, so ist zunächst überprüfen, ob OG die Generatorklasse ist, auf die sich die benannte K-Akteur–Inkarnationsklasse AN bezieht. Wenn dies nicht der Fall ist, so ist die durch `Caller.ActorName` $= AN$ spezifizierte Kontextrestriktion mit p nicht erfüllbar, da sich die in L_i^{Con} mit `Caller.ActorGen` $= OG$ und die in K_k^{Con} implizit mit `Caller.ActorName` $= AN$ festgelegten Restriktionen bzgl. des Generators des aufrufenden Akteurs widersprechen.
Gilt hingegen $(OG, AN) \in \gamma$ bzw. enthält L_i^{Con} keine Kontextrestriktion der Form `Caller.ActorGen` $= OG$, so ist zu untersuchen, ob es zur Laufzeit einen benannten K-Akteur der Inkarnationsklasse AN geben kann, der die kanonische Operation einer Inkarnation bzgl. eines Generators der Generatorklasse G_1 ausführt. Analog zu 2.1. ist in diesen Fällen die in K_k^{Con} durch die Gleichung `Caller.ActorName` $= AN$ spezifizierte Kontextrestriktion mit dem Aufrufpfad p erfüllbar, wenn es mindestens eine Pfad $w \in PathsTo(G_1)$ gibt, für den gilt: $lan(w) = AN$.

Beispiel

Der Zugriffsrestriktionsausdruck der S-Order–Generatorklasse `LeseKontonummer` des Kontenverwaltungssystems besteht aus der Kontextrestriktion `Caller.ActorName = KontoVerwalter`. Dieser Zugriffsrestriktionsausdruck ist mit dem Zugriffsrestriktionsausdruck der K-Order–Generatorklasse `GibKontoZeiger` auf Widerspruchsfreiheit zu analysieren, da (`GibKontoZeiger`, `LeseKontonummer`) $\in AnalysePairs(KVS)$ gilt. Für die Analyse der Erfüllbarkeit der Kontextrestriktion `Caller.ActorName = KontoVerwalter` ist der Aufrufpfad

$$p = (\texttt{GibKontoZeiger}, \texttt{SucheKonto}, \texttt{LeseKontonummer})$$

zu untersuchen. Für p gilt: $lan(p) = \{\texttt{KontoVerwalter}\}$. Damit ist die Kontextrestriktion `Caller.ActorName = KontoVerwalter` des Zugriffsrestriktionsausdrucks von

`LeseKontonummer` mit p erfüllbar.

◇

Mit der Angabe der Eigenschaften, denen ein Aufrufpfad $p \in \mathit{AnalysePaths}(G_1, G_2)$ genügen muß, damit mit ihm die in K_k^{Con} spezifizierten Kontextrestriktionen erfüllbar sind, ist das Verfahren für die Analyse zweier Zugriffsrestiktionsausdrücke R_1 und R_2 auf Widerspruchsfreiheit für den Fall, daß R_2 Kontextrestriktioen enthält, vollständig erklärt.

Zusammenfassung des Abschnitts 5.3.2.2

In diesem Abschnitt wurde die Analyse zweier Zugriffsrestriktionsausdrücke R_1 und R_2 auf Widersprüchlichkeit bzw. Widerspruchsfreiheit erklärt. Dabei wurde unterschieden, ob R_2 Kontextrestriktionen enthält oder nicht. Diese Unterscheidung wurde vor allem vorgenommen, um die Prüfungen, die in den beiden Fällen durchzuführen sind, zu motivieren und verständlich zu machen. Die für die beiden Fälle angegebenen Prüfungen können zu dem im folgenden erklärten einheitlichen Verfahren zur Analyse der Widerspruchsfreiheit zweier Zugriffsrestriktionsausdrücke zusammengefaßt werden (vgl. Abbildung 5.9).

Die beiden Zugriffsrestriktionsausdrücke R_1 und R_2 werden zunächst in ihre Disjunktive Normalform $DNF(R_1)$ und $DNF(R_2)$ gemäß (5.20) transformiert, wobei die einzelnen konjunktiven Teilterme der Disjunktiven Normalformen in der Form gemäß (5.22) dargestellt werden. Anschließend werden nacheinander alle Paare von konjunktiven Teiltermen L_i und K_k, wobei L_i konjunktiver Teilterm von $DNF(R_1)$ und K_k konjunktiver Teilterm von $DNF(R_2)$ ist, auf Widersprüchlichkeit analysiert. Sind alle möglichen Paare konjunktiver Teilterme widersprüchlich, so sind die Zugriffsrestriktionsausdrücke R_1 und R_2 widersprüchlich, anderenfalls, d.h. gibt es mindestens ein Paar konjunktiver Teilterme, das nicht widersprüchlich ist, sind R_1 und R_2 widerspruchsfrei.

Ein Paar von konjunktiven Teiltermen L_i und K_k wird auf Widersprüchlichkeit analysiert, indem zunächst überprüft wird, ob ihre keine Kontextrestriktionen enthaltenen Teilterme L_i^{Res} und K_k^{Res} widersprüchlich sind, d.h. mindestens zwei widersprüchliche Literale enthalten. Ist dies der Fall, so sind die beiden konjunktiven Teilterme L_i und K_k widersprüchlich. Im anderen Fall ist zu unterscheiden, ob in K_k Kontextrestriktionen enthalten sind oder nicht. Wenn in K_k keine Kontextrestriktionen auftreten[25], ist die Analyse beendet, da L_i und K_k in diesem Fall widerspruchsfrei sind. Anderenfalls ist zu überprüfen, ob die in K_k festgelegten Kontextrestriktionen erfüllbar sind. Dazu wird untersucht, ob es in der Menge $\mathit{AnalysePaths}(G_1, G_2)$ einen Aufrufpfad p gibt, mit dem ebendiese Kontextrestriktionen erfüllbar sind. Gibt es einen solchen Aufrufpfad, sind L_i und K_k widerspruchsfrei. Im anderen Fall sind L_i und K_k widersprüchlich.

Alle Paare von Zugriffsrestriktionsausdrücken eines INSEL$^+$–Programms $\mathcal{P}$, die mit der in Definition (5.19) festgelegten Menge $\mathit{AnalysePairs}(\mathcal{P})$ gegeben sind, werden vom INSEL$^+$–Übersetzer anhand des beschriebenen Analyseverfahrens auf Widerspruchsfreiheit überprüft. Stellt der INSEL$^+$–Übersetzer fest, daß ein Paar zu analysierender Zugriffsrestriktionsausdrücke widersprüchlich ist, so wird eine entsprechende Meldung ausgegeben, die u.a. die beiden widersprüchlichen Zugriffsrestriktionsausdrücke in Disjunktiver Normalform anzeigt und Informationen enthält, welche der einzelnen Paare konjunktiver Teilterme der Disjunktiven Normalformen widersprüchlich sind und wodurch die Widersprüche jeweils verursacht

[25] Enthält der Zugriffsrestriktionsausdruck R_2 keine Kontextrestriktionen, so gilt dies für alle konjunktiven Teilterme K_k von $DNF(R_2)$.

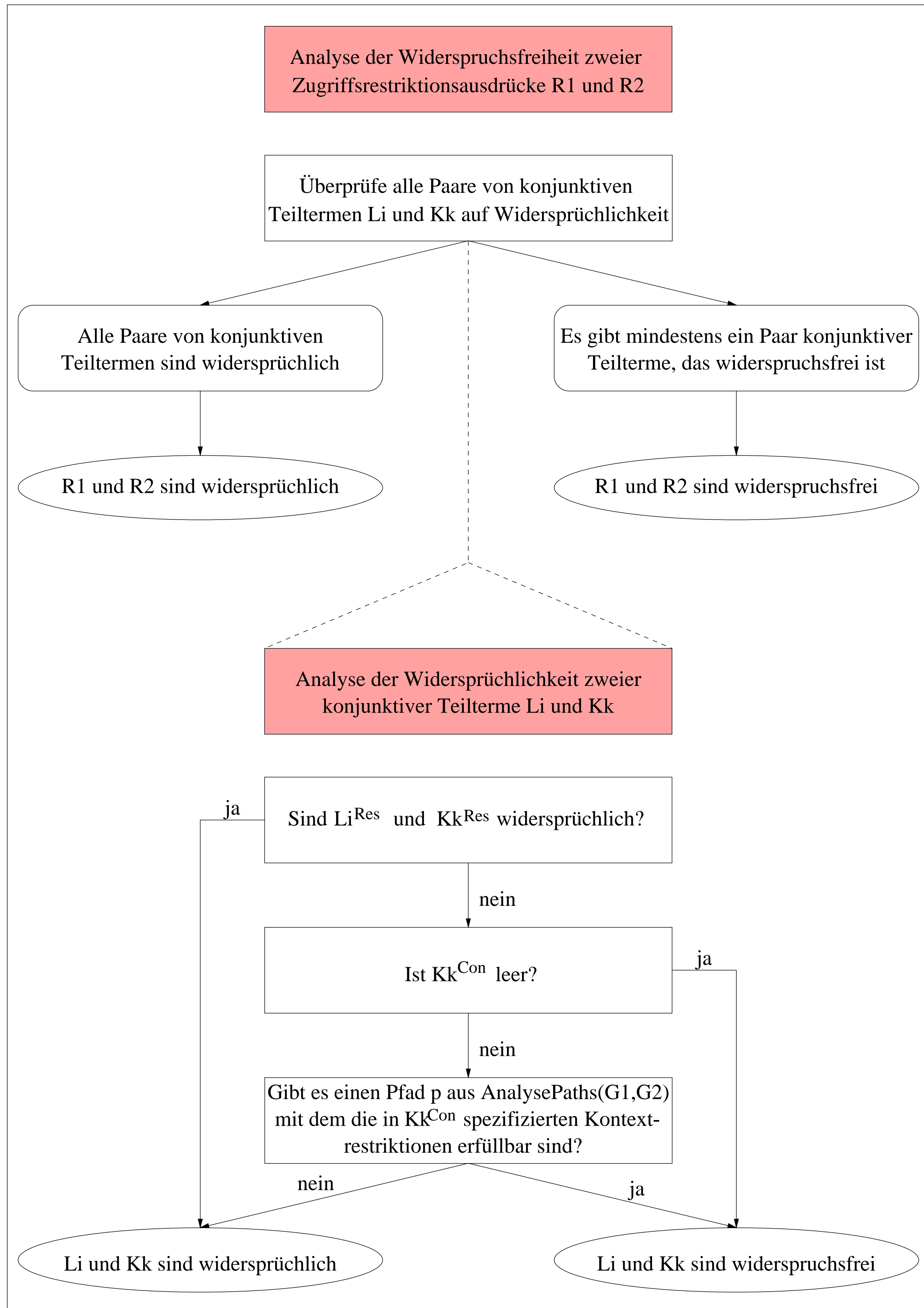

Abbildung 5.9: Analyse der Widerspruchsfreiheit zweier Zugriffsrestriktionsausdrücke

werden. Der Systementwickler hat dann die durch den INSEL$^+$–Übersetzer erkannten Widersprüche durch geeignete Modifikation der Zugriffsrestriktionsausdrücke zu beseitigen. Dabei ist darauf zu achten, daß durch die Änderungen keine neuen Widersprüche verursacht werden. Gemäß Abschnitt 5.2 wird an jedes INSEL$^+$–System die Forderung gestellt, daß das Recht des Systems zumindest potentiell konsistent ist. Dementsprechend aktzeptiert der INSEL$^+$–Übersetzer lediglich INSEL$^+$–Programme, für die gilt, daß alle zu analysierenden Paare von Zugriffsrestriktionsausdrücken widerspruchsfrei sind.

5.3.3 Absolute Konsistenz

In diesem Abschnitt werden die Kriterien und Analysen angegeben, auf deren Basis überprüft wird, ob das Recht eines INSEL$^+$–Systems absolut konsistent ist. Gemäß Abschnitt 5.2 ist das Recht eines INSEL$^+$–Systems absolut konsistent, wenn dessen Konsistenz statisch, d.h. zur Übersetzungszeit des entsprechenden INSEL$^+$–Programms, nachgewiesen werden kann. Im weiteren sind also Kriterien von Interesse, die statisch überprüfbar sind und deren Erfüllung für ein INSEL$^+$–System *gewährleistet*, daß das Recht des Systems konsistent ist.

Wie in Abschnitt 5.2 eingeführt, besagt die Konsistenz des Rechts eines INSEL$^+$–Systems, daß ein Akteur, der das Recht zur Ausführung einer äußeren Operation op einer Komponente k besitzt,

- im Kontext der op–Ausführung das Recht zur Ausführung aller Operationen hat, die im Rahmen der op–Ausführung durch ihn auszuführen sind und
- alle Akteure, die im Kontext der Ausführung von op erzeugt werden, das Recht zur Ausführung aller Operationen haben, die im Kontext ihrer kanonischen Operationen auszuführen sind.

Wenn k nun zugriffskontrolliert ist und op' eine im Kontext der op–Ausführung auszuführende äußere Operation einer Komponente x ist, die ebenfalls zugriffskontrolliert ist, so muß zum statischen Nachweis der Konsistenz des Rechts des Systems überprüft werden, ob aus der Gültigkeit des für op festgelegten Zugriffsrestriktionsausdrucks R^k_{op} stets die Gültigkeit von $R^x_{op'}$ folgt. Dabei sind die eventuell in R^k_{op} und $R^x_{op'}$ spezifizierten Kontextrestriktionen geeignet zu berücksichtigen. Sind G_1 und G_2 die beiden DA–Generatorklassen, deren *erzeuge*–Operationen die Operationen op und op' entsprechen und mit denen somit die Zugriffsrestriktionsausdrücke R^k_{op} und $R^x_{op'}$ festgelegt sind, so spiegeln sich die möglichen Kontexte, in denen die Operation op' bei Ausführung von op aufgerufen werden kann, in dem attributierten statischen Aufrufgraph des entsprechenden INSEL$^+$–Programms in der Menge der Aufrufpfade $AnalysePaths(G_1, G_2)$ (vgl. Definition (5.23)) wider. Die Analyse, ob aus der Gültigkeit von R^k_{op} immer die Gültigkeit von $R^x_{op'}$ folgt, ist somit für den Fall, daß $R^x_{op'}$ Kontextrestriktionen enthält, jeweils bezogen auf einen Aufrufpfad aus $AnalysePaths(G_1, G_2)$ durchzuführen. Dies motiviert den in der folgenden Definition eingeführten Begriff der Konsistenz zweier Zugriffsrestriktionsausdrücke bzgl. eines Aufrufpfades.

Definition 5.25.: Konsistenz zweier Zugriffsrestriktionsausdrücke bzgl. eines Aufrufpfades

Gegeben seien ein INSEL$^+$–Programm $\mathcal{P} = (\mathcal{D}, \delta, \gamma, \iota)$ und der attributierte statische Aufrufgraph $\mathcal{G}(\mathcal{P}) = (V, K, \mu)$ von $\mathcal{P}$. G_1 und G_2 seien zwei zugriffskontrollierte DA–Generatorklassen mit $(G_1, G_2) \in AnalysePairs(\mathcal{P})$, und R_1 und R_2 seien die G_1 bzw. G_2 zugeordneten Zugriffsrestriktionsausdrücke. Weiter sei p ein Aufrufpfad, für den

gilt: $p \in AnalysePaths(G_1, G_2)$ (siehe Definition (5.23)). Für R_1 und R_2 gelte, daß sie in ihrer Disjunktiven Normalform und die konjunktiven Teilterme der Normalform in der Form gemäß (5.22) vorliegen, also:

$$DNF(R_1) = \bigvee_{i=1}^{m_1} \bigwedge_{j=1}^{n_i} L_{ij} \text{ mit } L_i \triangleq \bigwedge_{j=1}^{n_i} L_{ij} = L_i^{Res} \wedge L_i^{Con} \text{ für } i \in \{1, \ldots, m_1\}$$

und

$$DNF(R_2) = \bigvee_{k=1}^{m_2} \bigwedge_{l=1}^{n_k} K_{kl} \text{ mit } K_k \triangleq \bigwedge_{l=1}^{n_k} K_{kl} = K_k^{Res} \wedge K_k^{Con} \text{ für } k \in \{1, \ldots, m_2\}$$

R_1 und R_2 sind konsistent bzgl. des Aufrufpfades $p \overset{\triangle}{\Longleftrightarrow}$ für alle $i \in \{1, \ldots, m_1\}$ gibt es mindestens ein $k \in \{1, \ldots, m_2\}$, so daß gilt:

1. $\omega(L_i^{Res}, a, t) = true \Longrightarrow \omega(K_k^{Res}, a', t') = true$
   für alle Zeitpunkte $t \in \Lambda(y)$ und $t' \geq t$ und alle Akteure a', die zum Zeitpunkt t' im Kontext einer durch einen Akteur a in t initiierten *op*–Ausführung op' auf x aufrufen, wobei x und y zwei zugriffskontrollierte Komponenten sind und $op \in O^E(y)$ die äußere Operation von y ist, für die R_1 festgelegt ist und $op' \in O^E(x)$ die äußere Operation von x ist, für die R_2 festgelegt ist.
2. die in K_k^{Con} spezifizierten Kontextrestriktionen sind mit dem Pfad p erfüllbar.

□

Laut Definition (5.25) sind also zwei Zugriffsrestriktionsausdrücke R_1 und R_2 konsistent bzgl. des Aufrufpfades p, wenn aus der Gültigkeit von R_1 unter Berücksichtigung des durch p festgelegten Kontextes stets die Gültigkeit von R_2 folgt. Aus der Konsistenz von R_1 und R_2 bzgl. des Aufrufpfades p folgt, daß immer dann, wenn ein Akteur das Recht zur Ausführung der *erzeuge*–Operation auf einem Generator der Generatorklasse G_1 hat, auch der Akteur, der gemäß des Pfades p die *erzeuge*–Operation auf einem Generator der Generatorklasse G_2 aufruft, das Recht zur Ausführung dieser *erzeuge*–Operation hat. In der in Bedingung 1. der Definition (5.25) gestellten Forderung, daß die Implikation für alle $t' \geq t$ gültig sein muß, kommt zum Ausdruck, daß die *erzeuge*–Operation auf G_2 im Kontext der Ausführung der kanonischen Operation einer Inkarnation bzgl. G_1 zu einem späteren Zeitpunkt aufgerufen wird als die *erzeuge*–Operation auf G_1, die die Erzeugung dieser Inkarnation bewirkt hat[26]. Dies ist im wesentlichen die Begründung dafür, daß die Möglichkeiten für die Vergabe dynamischer Rechte in INSEL$^+$–Systemen mit absolut konsistentem Recht stark eingeschränkt sind. So darf z.B. der Zugriffsrestriktionsausdruck R_2 in der Regel keine `IN_ACL`–Prädikate enthalten, da sich der Wert dieser Prädikate im allgemeinen zwischen dem Aufruf von *erzeuge* auf G_1 und dem „später“ erfolgenden Aufruf von *erzeuge* auf G_2 ändern kann. Unter Berücksichtigung dieser Überlegungen lassen sich hinreichende, statisch überprüfbare Kriterien für die Konsistenz zweier Zugriffsrestriktionsausdrücke bzgl. eines Aufrufpfades angeben.

5.3.3.1 Kriterien zur Analyse der Konsistenz zweier Zugriffsrestriktionsausdrücke bzgl. eines Aufrufpfades

Es seien zwei Zugriffsrestriktionsausdrücke R_1 und R_2 sowie ein Aufrufpfad p wie in Definition (5.25) gegeben. Zur Analyse, ob R_1 und R_2 konsistent bzgl. p sind, ist zu untersuchen, ob

[26] Vergleiche Fußnote auf S. 190.

es zu jedem konjunktiven Teilterm L_i von $DNF(R_1)$ einen konjunktiven Teilterm K_k aus $DNF(R_2)$ gibt, mit dem die Bedingungen 1. und 2. der Definition (5.25) erfüllt sind. Die Kriterien für die Überprüfung der Gültigkeit der Bedingung 2., d.h. die Eigenschaften, denen der Aufrufpfad p genügen muß, damit mit ihm die in K_k^{Con} spezifizierten Kontextrestriktionen erfüllbar sind, wurden bereits im Abschnitt 5.3.2.2 (vgl. S. 208ff) vollständig angegeben. Im weiteren sind somit lediglich noch Kriterien zu erarbeiten, die hinsichtlich der Gültigkeit der Bedingung 1. zu überprüfen sind. Hierzu wird zunächst der Begriff der Äquivalenz zweier Literale eingeführt.

Definition 5.26.: Äquivalenz zweier Literale

Seien R_1 und R_2 mit ihren Disjunktiven Normalformen $DNF(R_1)$ und $DNF(R_2)$ sowie zwei Literale L_{ij} und K_{kl} wie in Definition (5.25) gegeben. L_{ij} und K_{kl} seien keine Kontextrestriktionen.

Die **Literale L_{ij} und K_{kl} sind äquivalent** $\stackrel{\triangle}{\Longleftrightarrow}$ Für alle Zeitpunkte t und $t' \geq t$ und alle Akteure a gilt:

$$\omega(L_{ij}, a, t) = true \Longrightarrow \omega(K_{kl}, a, t') = true$$

□

Zwei Literale L_{ij} und K_{kl} sind also genau dann äquivalent, wenn aus der Gültigkeit von L_{ij} stets die Gültigkeit von K_{kl} folgt. Aufbauend auf dieser Definition ergibt sich der folgende Satz, auf dessen Basis die Bedingung 1. aus Definition (5.25) überprüft werden kann.

Satz 5.5.: Konsistenz zweier Zugriffsrestriktionsausdrücke bzgl. eines Aufrufpfades

Es seien die Voraussetzungen wie in Definition (5.25) und (5.26) gegeben. Weiter seien

$$L_i^{Res} \triangleq \bigwedge_{j=1}^{r_i} L_{ij} \text{ und } K_k^{Res} \triangleq \bigwedge_{l=1}^{r_k} K_{kl}$$

Dann gilt:

$\omega(L_i^{Res}, a, t) = true \Longrightarrow \omega(K_k^{Res}, a', t') = true \Longleftrightarrow$ Für alle $l \in \{1, \ldots, r_k\}$ gibt es ein $j \in \{1, \ldots, r_i\}$ so, daß gilt: L_{ij} und K_{kl} sind äquivalent.

△

Die Bedingung 1. aus Definition (5.25) ist also genau dann erfüllt, wenn es zu jedem Literal K_{kl} aus K_k^{Res} ein Literal L_{ij} aus L_i^{Res} gibt, so daß L_{ij} und K_{kl} äquivalent sind.

Beweis:

Seien x und y zwei zugriffskontrollierte Komponenten. $op \in O^E(y)$ sei die äußere Operation von y, für die R_1 festgelegt ist, und $op' \in O^E(x)$ sei die äußere Operation von x, für die R_2 festgelegt ist.

"$\Longrightarrow$": Beweis durch Widerspruch

Voraussetzung: Die Bedingung 1. aus Definition (5.25) sei erfüllt, d.h. es gilt:

$$(i) \quad \omega(L_i^{Res}, a, t) = true \Longrightarrow \omega(K_k^{Res}, a', t') = true$$

Da L_i^{Res} und K_k^{Res} definitionsgemäß keine Kontextrestriktionen enthalten, ist der Wert $\omega(K_k^{Res}, a', t')$ unabhängig von den Werten $ActGen_{t'}(a')$, $ActName_{t'}(a')$, $ConGen_{t'}(a')$ und $ConName_{t'}(a')$ des Kontextattributs des Akteurs a'. Da weiter für alle Akteure a', die im Kontext einer durch a initiierten *op*–Ausführung erzeugt werden, gilt: $User_t(a) = User_{t'}(a')$ und $Role_t(a) = Role_{t'}(a')$, kann die Voraussetzung (i) wie folgt vereinfacht werden:

$$(ii) \quad \omega(L_i^{Res}, a, t) = true \Longrightarrow \omega(K_k^{Res}, a, t') = true$$

Annahme: Es gibt ein $l \in \{1, \ldots, r_k\}$, für das es kein $j \in \{1, \ldots, r_i\}$ gibt, so daß L_{ij} und K_{kl} äquivalent sind. Dann kann es einen Zeitpunkt t und einen Akteur a geben, so daß für alle Literale L_{ij} aus L_i^{Res} gilt: $\omega(L_{ij}, a, t) = true$ und damit $\omega(L_i^{Res}, a, t) = true$ aber $\omega(K_k^{Res}, a, t) = false$, da $\omega(K_{kl}, a, t) = false$ gilt. Dies ist jedoch ein Widerspruch zu (ii).

"$\Longleftarrow$": Voraussetzung: Für alle $l \in \{1, \ldots, r_k\}$ gibt es ein $j \in \{1, \ldots, r_i\}$, so daß gilt:

$$(iii) \quad \omega(L_{ij}, a, t) = true \Longrightarrow \omega(K_{kl}, a, t') = true$$

Damit gilt jedoch auch

$$\omega(L_i^{Res}, a, t) = true \Longrightarrow \omega(K_k^{Res}, a, t') = true$$

und damit aufgrund der Tatsache, daß K_k^{Res} keine Kontextrestriktionen enthält, die Behauptung.

$\triangledown$

Im folgenden werden nun hinreichende, statisch überprüfbare Bedingungen angegeben, anhand derer der INSEL$^+$–Übersetzer überprüft, ob zwei Literale äquivalent sind. Auf Basis dieser Kriterien und des Satzes (5.5) wird die Konsistenz zweier Zugriffsrestriktionsausdrücke R_1 und R_2 bzgl. eines Aufrufpfades p vom INSEL$^+$–Übersetzer wie folgt analysiert: nach Umwandlung von R_1 und R_2 in ihre Disjunktiven Normalformen gemäß Definition (5.25) wird für den ersten konjunktiven Teilterm von $DNF(R_1)$ überprüft, ob es zu diesem einen konjunktiven Teilterm aus $DNF(R_2)$ gibt, mit dem die Bedingungen 1. und 2. der Definition (5.25) erfüllt sind. Hierzu werden nacheinander alle konjunktiven Teilterme aus $DNF(R_2)$ anhand der für die Bedingungen 1. und 2. angegebenen Kriterien analysiert, bis ein konjunktiver Teilterm gefunden ist, der diese Bedingungen erfüllt. Anschließend wird analog für den zweiten konjunktiven Teilterm von $DNF(R_1)$ überprüft, ob es zu diesem einen entsprechenden konjunktiven Teilterm aus $DNF(R_2)$ gibt. Die Analyse kann abgebrochen werden, sobald zu einem konjunktiven Teilterm von $DNF(R_1)$ kein konjunktiver Teilterm aus $DNF(R_2)$ gefunden wird, mit dem die Bedingungen 1. und 2. erfüllt sind. In diesem Fall sind R_1 und R_2 nicht konsistent bzgl. des Aufrufpfades p. Gibt es zu jedem konjunktiven Teilterm von $DNF(R_1)$ einen konjunktiven Teilterm aus $DNF(R_2)$, der die Bedingungen 1. und 2. erfüllt, sind R_1 und R_2 konsistent bzgl. des Aufrufpfades p.

In dem folgenden Satz über die Kriterien zur Analyse der Äquivalenz zweier Literale wird zu jeder darin angegebenen Bedingung informell begründet, warum bei Erfüllung der Bedingung die beiden Literale äquivalent sind.

Satz 5.6.: Kriterien für die Äquivalenz zweier Literale

Seien L_{ij} und K_{kl} zwei Literale gemäß der Voraussetzungen der Definition (5.26). `G1` sei der Name der zugriffskontrollierten DA–Generatorklasse, für die der Zugriffsrestriktionsausdruck R_1 festgelegt ist, und `G2` sei der Name der zugriffskontrollierten DA–Generatorklasse, der der Zugriffsrestriktionsausdruck R_2 zugeordnet ist.

Die Literale L_{ij} und K_{kl} sind **äquivalent**, wenn eine der folgenden Bedingungen erfüllt ist:

1. L_{ij} ist `Caller.User = UserId1` und K_{kl} ist `Caller.User = UserId2` oder L_{ij} ist ¬`(Caller.User = UserId1)` und K_{kl} ist ¬`(Caller.User = UserId2)` und es gilt: `UserId1` $=_N$ `UserId2` und `UserId1` bezeichnet eine konstante DE–Inkarnation bzgl. des vordefinierten Generators `UserUidType`.
   Begründung:
   Aus der Annahme $(G_1, G_2) \in \mathit{AnalysePairs}(\mathcal{P})$ folgt, daß `Caller.User` in beiden Literalen stets den gleichen Wert hat. Da `UserId1` und `UserId2` ebenfalls stets den gleichen Wert haben, folgt unmittelbar: $\omega(L_{ij}, a, t) = \mathit{true} \Longrightarrow \omega(K_{kl}, a, t') = \mathit{true}$.
2. L_{ij} ist `Caller.Role = RoleId1` und K_{kl} ist `Caller.Role = RoleId2` oder L_{ij} ist ¬`(Caller.Role = RoleId1)` und K_{kl} ist ¬`(Caller.Role = RoleId2)` und es gilt: `RoleId1` $=_N$ `RoleId2` und `RoleId1` bezeichnet eine konstante DE–Inkarnation bzgl. des vordefinierten Generators `RoleUidType`[27].
   Begründung:
   Aus der Annahme $(G_1, G_2) \in \mathit{AnalysePairs}(\mathcal{P})$ folgt, daß `Caller.Role` in beiden Literalen stets den gleichen Wert hat. Da `RoleId1` und `RoleId2` ebenfalls stets den gleichen Wert haben, folgt unmittelbar: $\omega(L_{ij}, a, t) = \mathit{true} \Longrightarrow \omega(K_{kl}, a, t') = \mathit{true}$.
3. L_{ij} ist `Caller.Role = RoleId1` und K_{kl} ist ¬`(Caller.Role = RoleId2)` und es gilt: `RoleId1` $\neq_N$ `RoleId2` und `RoleId1` und `RoleId2` sind Rollenbezeichner, die in dem Rollen–Part des INSEL$^+$–Programms festgelegt sind.
   Begründung:
   Aus der Annahme $(G_1, G_2) \in \mathit{AnalysePairs}(\mathcal{P})$ folgt, daß `Caller.Role` in beiden Literalen stets den gleichen Wert hat. `RoleId1` und `RoleId2` haben jedoch stets einen unterschiedlichen Wert. Daraus folgt unmittelbar:
   $\omega(L_{ij}, a, t) = \mathit{true} \Longrightarrow \omega(K_{kl}, a, t') = \mathit{true}$.
4. L_{ij} ist `IN_ACL(UserId1, RoleId1, CompId1, OpId1)` und
   K_{kl} ist `IN_ACL(UserId2, RoleId2, CompId2, OpId2)` oder
   L_{ij} ist ¬`(IN_ACL(UserId1, RoleId1, CompId1, OpId1))` und
   K_{kl} ist ¬`(IN_ACL(UserId2, RoleId2, CompId2, OpId2))` und es gilt:
   (a) `G2` ist explizit zugriffskontrollierte DA–Generatorklasse und
      - `UserId1` $=_N$ `UserId2` und
      - `RoleId1` $=_N$ `RoleId2` und
      - `CompId1` $=_N$ `G2` und
      - `CompId2` $=_N$ `THIS` und
      - `OpId1` $=_N$ `Create` und
      - `OpId2` $=_N$ `THIS` oder `OpId2` $=_N$ `Create` und
      - das INSEL$^+$–Programm enthält keinen `ChangeGenACL`–Aufruf auf `G2`, für dessen Parameter `ViewName` gilt: `ViewName` = `Create`

      oder
   (b) `G2` ist explizit zugriffskontrollierte DA–Generatorklasse und
      - `UserId1` $=_N$ `UserId2` und
      - `RoleId1` $=_N$ `RoleId2` und

[27] Hierzu gehören insbesondere alle in dem Rollen–Part des INSEL$^+$–Programms festgelegten Rollenbezeichner.

- CompId1 $\neq_N$ G2 und
- CompId2 $\neq_N$ THIS und
- CompId1 $=_N$ CompId2 und CompId1 bezeichnet eine explizit zugriffskontrollierte DA–Generatorklasse, eine zugriffskontrollierte benannte DA–Inkarnationsklasse, oder eine konstante Zeiger–Inkarnationsklasse, die mit einer explizit zugriffskontrollierten Depot– oder K-Akteur–Generatorklasse qualifiziert ist, und
- OpId1 $=_N$ OpId2 und
- das INSEL^{+}–Programm enthält keinen ChangeGenACL– bzw. ChangeACL–Aufruf auf der mit CompId1 identifizierten Komponente, so daß für den Parameter ViewName dieses Aufrufs gilt: ViewName = OpId1[28]

oder

(c) G2 ist implizit zugriffskontrollierte DA–Generatorklasse und

- UserId1 $=_N$ UserId2 und
- RoleId1 $=_N$ RoleId2 und
- CompId1 ist entweder der Name einer benannten DA–Inkarnationsklasse, die sich auf G3 bezieht, oder der Name einer Zeiger–Inkarnationsklasse, die mit G3 qualifiziert ist, wobei G3 die explizit zugriffskontrollierte DA-Generatorklasse ist, in deren Deklarationsteil G2 enthalten ist, und
- CompId2 $=_N$ THIS und
- OpId1 $=_N$ G2 und
- OpId2 $=_N$ THIS oder OpId2 $=_N$ G2 und
- das INSEL^{+}–Programm enthält keinen ChangeACL–Aufruf auf der mit CompId1 identifizierten Komponente, für dessen Parameter ViewName gilt: ViewName = G2[28]

oder

(d) G2 ist implizit zugriffskontrollierte DA–Generatorklasse und

- UserId1 $=_N$ UserId2 und
- RoleId1 $=_N$ RoleId2 und
- CompId1 ist entweder der Name einer benannten DA–Inkarnationsklasse, die sich auf G3 bezieht, oder der Name einer Zeiger–Inkarnationsklasse, die mit G3 qualifiziert ist, wobei G3 die explizit zugriffskontrollierte DA-Generatorklasse ist, in deren Deklarationsteil G2 enthalten ist, und
- CompId2 $=_N$ THIS und
- OpId1 $=_N$ G4, wobei G4 eine implizit zugriffskontrollierte DA–Generatorklasse ist, die im gleichen Deklarationsteil wie G2 definiert ist, und
- OpId2 $=_N$ G4 und
- das INSEL^{+}–Programm enthält keinen ChangeACL–Aufruf auf der mit CompId1 identifizierten Komponente, für dessen Parameter ViewName gilt: ViewName = G4[28]

oder

[28]Bzw. präziser: ViewName = ViewName1 und OpId1 ist Element des mit ViewName1 bezeichneten Views, d.h. OpId1 $\in$ ViewName.

(e) `G2` ist implizit zugriffskontrollierte DA–Generatorklasse und

- `UserId1` $=_N$ `UserId2` und
- `RoleId1` $=_N$ `RoleId2` und
- `CompId1` ist weder der Name einer benannten DA–Inkarnationsklasse, die sich auf `G3` bezieht, noch der Name einer Zeiger–Inkarnationsklasse, die mit `G3` qualifiziert ist, wobei `G3` die explizit zugriffskontrollierte DA-Generatorklasse ist, in deren Deklarationsteil `G2` enthalten ist, und
- `CompId2` $\neq_N$ `THIS` und
- `CompId1` $=_N$ `CompId2` und `CompId1` bezeichnet eine explizit zugriffskontrollierte DA–Generatorklasse, eine zugriffskontrollierte benannte DA–Inkarnationsklasse, oder eine konstante Zeiger–Inkarnationsklasse, die mit einer explizit zugriffskontrollierten Depot– oder K-Akteur–Generatorklasse qualifiziert ist, und
- `OpId1` $=_N$ `OpId2` und
- das INSEL$^+$–Programm enthält keinen `ChangeGenACL`– bzw. `ChangeACL`–Aufruf auf der mit `CompId1` identifizierten Komponente, so daß für den Parameter `ViewName` dieses Aufrufs gilt: `ViewName` = `OpId1`[28]

Begründung:
Ist (a), (b), (c), (d) oder (e) erfüllt, so identifizieren `CompId1` und `CompId2` zur Laufzeit stets die gleiche zugriffskontrollierte Komponente sowie `OpId1` und `OpId2` die gleiche äußere Operation dieser Komponente. Da Änderungen in dem Zugriffskontrollisteneintrag dieser Operation in der Zugriffskontrolliste der mit `CompId1` identifizierten Komponente aufgrund der fehlenden `ChangeGenACL`– bzw. `ChangeACL`–Aufrufe nicht möglich sind, folgt: $\omega(L_{ij}, a, t) = true \Longrightarrow \omega(K_{kl}, a, t') = true$.

5. L_{ij} ist `ACCESSED(UserId1, CompId1, OpId1)` und
K_{kj} ist `ACCESSED(UserId2, CompId2, OpId2)` und es gilt:

(a) `G2` ist explizit zugriffskontrollierte DA–Generatorklasse und

- `UserId1` $=_N$ `UserId2` und
- `CompId1` $=_N$ `G2` und
- `CompId2` $=_N$ `THIS` und
- `OpId1` $=_N$ `OpId2` oder `OpId2` $=_N$ `ANY`

oder

(b) `G2` ist implizit zugriffskontrollierte DA–Generatorklasse und

- `UserId1` $=_N$ `UserId2` und
- `CompId1` ist entweder der Name einer benannten DA–Inkarnationsklasse, die sich auf `G3` bezieht, oder der Name einer Zeiger–Inkarnationsklasse, die mit `G3` qualifiziert ist, wobei `G3` die explizit zugriffskontrollierte DA-Generatorklasse ist, in deren Deklarationsteil `G2` enthalten ist, und
- `CompId2` $=_N$ `THIS` und
- `OpId1` $=_N$ `OpId2` oder `OpId2` $=_N$ `ANY`

oder

(c) `G2` ist explizit zugriffskontrollierte DA–Generatorklasse und

- `UserId1` $=_N$ `UserId2` und
- `CompId1` $\neq_N$ `G2` und
- `CompId2` $\neq_N$ `THIS` und

- CompId1 $=_N$ CompId2 und CompId1 bezeichnet eine DA–Generatorklasse, eine benannte DA–Inkarnationsklasse, oder eine konstante Zeiger–Inkarnationsklasse, die mit einer Depot– oder K-Akteur–Generatorklasse qualifiziert ist, und
- OpId1 $=_N$ OpId2 oder OpId2 $=_N$ ANY

oder

(d) G2 ist implizit zugriffskontrollierte DA–Generatorklasse und

- UserId1 $=_N$ UserId2 und
- CompId1 ist weder der Name einer benannten DA–Inkarnationsklasse, die sich auf G3 bezieht, noch der Name einer Zeiger–Inkarnationsklasse, die mit G3 qualifiziert ist, wobei G3 die explizit zugriffskontrollierte DA-Generatorklasse ist, in deren Deklarationsteil G2 enthalten ist, und
- CompId2 $\neq_N$ THIS und
- CompId1 $=_N$ CompId2 und CompId1 bezeichnet eine DA–Generatorklasse, eine benannte DA–Inkarnationsklasse, oder eine konstante Zeiger–Inkarnationsklasse, die mit einer Depot– oder K-Akteur–Generatorklasse qualifiziert ist, und
- OpId1 $=_N$ OpId2 oder OpId2 $=_N$ ANY

Begründung:
Gilt (a), (b), (c) oder (d), so identifizieren CompId1 und CompId2 zur Laufzeit stets die gleiche Komponente sowie OpId1 und OpId2 im Fall OpId2 $\neq_N$ ANY die gleiche äußere Operation dieser Komponente. Mit der Semantik des ACCESSED–Prädikats (vgl. Definition (4.10)) folgt: $\omega(L_{ij}, a, t) = true \Longrightarrow \omega(K_{kl}, a, t') = true$.

6. L_{ij} ist ¬(ACCESSED(UserId1, CompId1, OpId1)) und
K_{kj} ist ¬(ACCESSED(UserId2, CompId2, OpId2)) und es gilt:

(a) G2 ist explizit zugriffskontrollierte DA–Generatorklasse und

- UserId1 $=_N$ UserId2 und
- CompId1 $=_N$ G2 und
- CompId2 $=_N$ THIS und
- OpId1 $=_N$ OpId2 oder OpId1 $=_N$ ANY

oder

(b) G2 ist implizit zugriffskontrollierte DA–Generatorklasse und

- UserId1 $=_N$ UserId2 und
- CompId1 ist entweder der Name einer benannten DA–Inkarnationsklasse, die sich auf G3 bezieht, oder der Name einer Zeiger–Inkarnationsklasse, die mit G3 qualifiziert ist, wobei G3 die explizit zugriffskontrollierte DA-Generatorklasse ist, in deren Deklarationsteil G2 enthalten ist, und
- CompId2 $=_N$ THIS und
- OpId1 $=_N$ OpId2 oder OpId1 $=_N$ ANY

oder

(c) G2 ist explizit zugriffskontrollierte DA–Generatorklasse und

- UserId1 $=_N$ UserId2 und
- CompId1 $\neq_N$ G2 und
- CompId2 $\neq_N$ THIS und

- `CompId1` $=_N$ `CompId2` und `CompId1` bezeichnet eine DA–Generatorklasse, eine benannte DA–Inkarnationsklasse, oder eine konstante Zeiger–Inkarnationsklasse, die mit einer Depot– oder K-Akteur–Generatorklasse qualifiziert ist, und
- `OpId1` $=_N$ `OpId2` oder `OpId1` $=_N$ `ANY`

oder

(d) `G2` ist implizit zugriffskontrollierte DA–Generatorklasse und

- `UserId1` $=_N$ `UserId2` und
- `CompId1` ist weder der Name einer benannten DA–Inkarnationsklasse, die sich auf `G3` bezieht, noch der Name einer Zeiger–Inkarnationsklasse, die mit `G3` qualifiziert ist, wobei `G3` die explizit zugriffskontrollierte DA-Generatorklasse ist, in deren Deklarationsteil `G2` enthalten ist, und
- `CompId2` $\neq_N$ `THIS` und
- `CompId1` $=_N$ `CompId2` und `CompId1` bezeichnet eine DA–Generatorklasse, eine benannte DA–Inkarnationsklasse, oder eine konstante Zeiger–Inkarnationsklasse, die mit einer Depot– oder K-Akteur–Generatorklasse qualifiziert ist, und
- `OpId1` $=_N$ `OpId2` oder `OpId1` $=_N$ `ANY`

Begründung:
Analog zur Begründung von 5. folgt mit der Semantik des `ACCESSED`–Prädikats: $\omega(L_{ij}, a, t) = true \Longrightarrow \omega(K_{kl}, a, t') = true$.

7. L_{ij} ist $\mathcal{A} = \mathcal{B}$ und K_{kl} ist $\mathcal{C} = \mathcal{D}$ oder L_{ij} ist $\neg(\mathcal{A} = \mathcal{B})$ und K_{kl} ist $\neg(\mathcal{C} = \mathcal{D})$, wobei $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ und $\mathcal{D}$ Namen konstanter DE–Inkarnationen sind, und es gilt:
$$\mathcal{A} =_N \mathcal{C} \text{ und } \mathcal{B} =_N \mathcal{D} \text{ oder } \mathcal{A} =_N \mathcal{D} \text{ und } \mathcal{B} =_N \mathcal{C}$$
Begründung:
Da die mit $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ und $\mathcal{D}$ identifizierten DE–Inkarnationen konstante Werte haben, folgt mit der üblichen Semantik für die auf den vordefinierten elementaren Datentypen definierten Vergleichsprädikate '=' und '<': $\omega(L_{ij}, a, t) = true \Longrightarrow \omega(K_{kl}, a, t') = true$.
8. L_{ij} ist $\mathcal{A} < \mathcal{B}$ und K_{kl} ist $\mathcal{C} < \mathcal{D}$ oder L_{ij} ist $\neg(\mathcal{A} < \mathcal{B})$ und K_{kl} ist $\neg(\mathcal{C} < \mathcal{D})$, wobei $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ und $\mathcal{D}$ Namen konstanter DE–Inkarnationen sind, und es gilt:
$$\mathcal{A} =_N \mathcal{C} \text{ und } \mathcal{B} =_N \mathcal{D}$$
Begründung:
Analog zu 7.
9. L_{ij} ist $\mathcal{A} < \mathcal{B}$ und K_{kl} ist $\neg(\mathcal{C} = \mathcal{D})$, wobei $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ und $\mathcal{D}$ Namen konstanter DE–Inkarnationen sind, und es gilt:
$$\mathcal{A} =_N \mathcal{C} \text{ und } \mathcal{B} =_N \mathcal{D} \text{ oder } \mathcal{A} =_N \mathcal{D} \text{ und } \mathcal{B} =_N \mathcal{C}$$
Begründung:
Analog zu 7.
10. L_{ij} ist $\mathcal{A} < \mathcal{B}$ und K_{kl} ist $\neg(\mathcal{C} < \mathcal{D})$, wobei $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ und $\mathcal{D}$ Namen konstanter DE–Inkarnationen sind, und es gilt:
$$\mathcal{A} =_N \mathcal{D} \text{ und } \mathcal{B} =_N \mathcal{C}$$
Begründung:
Analog zu 7.
11. L_{ij} ist $\mathcal{A} = \mathcal{B}$ und K_{kl} ist $\neg(\mathcal{C} < \mathcal{D})$, wobei $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ und $\mathcal{D}$ Namen konstanter DE–Inkarnationen sind, und es gilt:
$$\mathcal{A} =_N \mathcal{C} \text{ und } \mathcal{B} =_N \mathcal{D} \text{ oder } \mathcal{A} =_N \mathcal{D} \text{ und } \mathcal{B} =_N \mathcal{C}$$

Begründung:
Analog zu 7.

12. L_{ij} ist ein beliebiges Literal gemäß (5.15) und K_{kl} ist `TRUE`.
Begründung:
Da $\omega(K_{kl}, a, t)$ immer den Wert *true* hat, folgt die Behauptung unmittelbar.

△

Beispiel

Als Beispiel für die Analyse zweier Zugriffsrestriktionsausdrücke auf Konsistenz bzgl. eines Aufrufpfades werden im folgenden zunächst die Zugriffsrestriktionausdrücke der beiden DA–Generatorklassen `KontoAufloesen` und `Aufloesen` aus dem INSEL$^+$–Programm des Kontenverwaltungssystems betrachtet (vgl. auch Beispiel auf Seite 200). Es gilt $(\texttt{KontoAufloesen}, \texttt{Aufloesen}) \in \mathit{AnalysePairs}(KVS)$. Weiter sei der Aufrufpfad $p = (\texttt{KontoAufloesen}, \texttt{Aufloesen})$ mit $p \in \mathit{AnalysePaths}(\texttt{KontoAufloesen}, \texttt{Aufloesen})$ gegeben.

Da der für `Aufloesen` festgelegte Zugriffsrestriktionsausdruck variable DE–Inkarnationen enthält (`Zeit.Stunde` und `Zeit.Wochentag`) sind $R_{\texttt{KontoAufloesen}}$ und $R_{\texttt{Aufloesen}}$ nicht konsistent bzgl. des Aufrufpfades p. In diesem Beispiel wird deshalb im weiteren davon ausgegangen, daß der Zugriffsrestriktionsausdruck von `Aufloesen` keine nicht konstanten DE–Inkarnationen enthält, d.h. es werden aus $R_{\texttt{Aufloesen}}$ alle Literale gestrichen, die den Bezeichner `Zeit.Stunde` oder `Zeit.Wochentag` enthalten. Die beiden im weiteren betrachteten Zugriffsrestriktionsausdrücke lauten damit in Disjunktiver Normalform:

$$
\begin{aligned}
DNF(R_{\texttt{KontoAufloesen}}) = \; & (\texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\
& \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{KontoZeiger}, \texttt{Aufloesen}) \wedge \\
& 8 < \texttt{Zeit.Stunde} \wedge \texttt{Zeit.Stunde} < 17 \wedge \\
& 1 < \texttt{Zeit.Wochentag} \wedge \texttt{Zeit.Wochentag} < 6) \vee \\
& (\texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\
& \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{KontoZeiger}, \texttt{Aufloesen}) \wedge \\
& 8 < \texttt{Zeit.Stunde} \wedge \texttt{Zeit.Stunde} < 17 \wedge \\
& 1 = \texttt{Zeit.Wochentag} \wedge \texttt{Zeit.Wochentag} < 6) \vee \\
& (\texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\
& \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{KontoZeiger}, \texttt{Aufloesen}) \wedge \\
& 8 = \texttt{Zeit.Stunde} \wedge \texttt{Zeit.Stunde} < 17 \wedge \\
& 1 < \texttt{Zeit.Wochentag} \wedge \texttt{Zeit.Wochentag} < 6) \vee \\
& (\texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\
& \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{KontoZeiger}, \texttt{Aufloesen}) \wedge \\
& 8 = \texttt{Zeit.Stunde} \wedge \texttt{Zeit.Stunde} < 17 \wedge \\
& 1 = \texttt{Zeit.Wochentag} \wedge \texttt{Zeit.Wochentag} < 6) \\
DNF(R'_{\texttt{Aufloesen}}) = \; & \texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\
& \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{THIS}, \texttt{THIS}) \wedge \\
& \texttt{Caller.ConGen} = \texttt{KontoAufloesen}
\end{aligned}
$$

Nun ist für den ersten konjunktiven Teilterm L_1 von $DNF(R_{\texttt{KontoAufloesen}})$ zu überprüfen, ob mit dem ersten (und in diesem Fall auch einzigen) konjunktiven Teilterm K_1 von $DNF(R'_{\texttt{Aufloesen}})$ die Bedingungen 1. und 2. der Definition (5.25) erfüllt sind. Hierzu ist zunächst zu untersuchen, ob es zu jedem Literal aus K_1, das keine Kontextrestriktion ist, ein Literal aus L_1 gibt, so daß diese beiden Literale äquivalent sind. Es gilt:

- L_{11} = `Caller.Role = Kundenbetreuer` ist äquivalent zu K_{11} = `Caller.Role = Kundenbetreuer` gemäß Bedingung 2. aus Satz (5.6);

- L_{12} = `IN_ACL(Caller.User, Caller.Role, KontoZeiger, Aufloesen)` ist äquivalent zu K_{12} = `IN_ACL(Caller.User, Caller.Role, THIS, THIS)` gemäß Bedingung 4.(c) aus Satz (5.6), wenn es in dem Kontenverwaltungssystem keinen `ChangeACL`–Aufruf auf dem mit `KontoZeiger` identifizierten Konto–Depot geben kann, so daß für den Parameter `ViewName` des `ChangeACL`–Aufrufs `ViewName` = `Aufloesen` gilt.

Gäbe es in dem Kontenverwaltungssystem keinen derartigen `ChangeACL`–Aufruf, wäre die Bedingung 1. der Definition (5.25) für die konjunktiven Teilterme L_1 und K_1 erfüllt. Da in dem Kontenverwaltungssystem jedoch ein entsprechender `ChangeACL`–Aufruf möglich ist (vgl. Anweisungsteil der Benutzerrepräsentanten–Generatorklasse `Bankleiter`), ist die Bedingung 1. für L_1 und K_1 nicht erfüllt.

Für die Analyse der Gültigkeit der Bedingung 2. ist zu überprüfen, ob die in K_1 spezifizierte Kontextrestriktion `Caller.ConGen = KontoAufloesen` mit dem Aufrufpfad p = (`KontoAufloesen`, `Aufloesen`) erfüllbar ist. Dies ist der Fall, da gilt: $lcg(p)$ = `KontoAufloesen`.

Wie leicht zu sehen ist, sind für die Teilterme L_2, L_3 und L_4 von $DNF(R_{\texttt{KontoAufloesen}})$ die Bedingungen 1. und 2. unter der Voraussetzung, daß das Kontenverwaltungssystem keinen entsprechenden `ChangeACL`–Aufruf bzgl. des mit `KontoZeiger` identifizierten Konto–Depots enthält, ebenfalls mit K_1 erfüllt. $R_{\texttt{KontoAufloesen}}$ und $R'_{\texttt{Aufloesen}}$ wären somit konsistent bzgl. des Aufrufpfades p, wenn die Voraussetzung hinsichtlich des `ChangeACL`–Aufrufs erfüllt wäre.

Als weiteres Beispiel werden die Zugriffsrestriktionsausdrücke der DA–Generatorklassen `KontoVerwalterTyp` und `Einzahlen`, für die (`KontoVerwalterTyp`, `Einzahlen`) $\in$ *AnalysePairs*(KVS) gilt, sowie der Aufrufpfad p' = (`KontoVerwalterTyp`, `Zinsberechnung`, `Zinsberechner`, `Einzahlen`) mit $p' \in$ *AnalysePaths*(`KontoVerwalterTyp`, `Einzahlen`) betrachtet.

$$
\begin{aligned}
DNF(R_{\texttt{KontoVerwalterTyp}}) &= \texttt{Caller.User} = \texttt{System} \\
DNF(R_{\texttt{Einzahlen}}) &= (\texttt{Caller.Role} = \texttt{Kassierer} \wedge \\
&\quad 8 <= \texttt{Zeit.Stunde} \wedge \texttt{Zeit.Stunde} < 17 \wedge \\
&\quad 1 <= \texttt{Zeit.Wochentag} \wedge \texttt{Zeit.Wochentag} < 6) \vee \\
&\quad (\texttt{Caller.ConGen} = \texttt{Ueberweiser}) \vee \\
&\quad (\texttt{Caller.ActorGen} = \texttt{Zinsberechner})
\end{aligned}
$$

Für den einzigen konjunktiven Teilterm von $DNF(R_{\texttt{KontoVerwalterTyp}})$ ist zu überprüfen, ob es einen konjunktiven Teilterm aus $DNF(R_{\texttt{Einzahlen}})$ gibt, so daß die Bedingungen 1. und 2. der Definition (5.25) erfüllt sind. Dies ist für den konjunktiven Teilterm K_3 = `Caller.ActorGen = Zinsberechner` der Fall, da:

- $K_3^{Res} = \emptyset$ gilt (K_3 besteht lediglich aus einer Kontextrestriktion) und somit die Bedingung 1. per se erfüllt ist und

- die Kontextrestriktion `Caller.ActorGen = Zinsberechner` mit dem Aufrufpfad p' erfüllbar ist (Begründung siehe Beispiel auf S. 211).

Daraus folgt, daß $R_{\texttt{KontoVerwalterTyp}}$ und $R_{\texttt{Einzahlen}}$ konsistent bzgl. des Aufrufpfades p' sind.

◇

Auf Basis des Satzes (5.5), der in Satz (5.6) angegebenen hinreichenden Kriterien für die Äquivalenz zweier Literale sowie der in Abschnitt 5.3.2.2 angegebenen Bedingungen für die Erfüllbarkeit einer Kontextrestriktion mit einem Aufrufpfad kann die Konsistenz zweier Zugriffsrestriktionsausdrücke bzgl. eines Aufrufpfades statisch nachgewiesen werden. Für die Analyse der absoluten Konsistenz des Rechts eines INSEL$^+$–Systems ist es nicht erforderlich, jedes einzelne Paar von Zugriffsrestriktionsausdrücken, das sich aus der Menge $AnalysePairs(\mathcal{P})$ ergibt, auf Konsistenz bzgl. aller Aufrufpfade aus der Menge $AnalysePaths(G_1, G_2)$ zu überprüfen. Bei dieser Analyse sind vielmehr lediglich Paare von Zugriffsrestriktionsausdrücken zu berücksichtigen, die sich aus Aufrufpfaden des attributierten statischen Aufrufgraphen ergeben, die entweder von einer Generatorklasse für Benutzerrepräsentanten oder der Generatorklasse der Hauptkomponente ausgehen und vollständige Operationsausführungen widerspiegeln. Dies wird in dem nun folgenden Abschnitt näher erläutert.

5.3.3.2 Analyse der absoluten Konsistenz

Die Konsistenz des Rechts eines INSEL$^+$–Systems besagt informell insbesondere, daß eine einmal angestoßene und begonnene Operationsausführung stets vollständig ausgeführt werden kann und nicht aufgrund eines fehlenden Rechts für eine im Kontext der Operationsausführung auszuführende Operation abgebrochen werden muß. In Abschnitt 5.3.2.2 wurde bereits erläutert, daß Operationsausführungen in einem INSEL$^+$–System ursächlich entweder nur von der Hauptkomponente des Systems oder von einem Benutzer, d.h. von einem Benutzerrepräsentanten aus angestoßen werden können. Dem entsprechend sind für die Analyse der absoluten Konsistenz des Rechts eines INSEL$^+$–Systems alle Aufrufpfade des attributierten statischen Aufrufgraphen, die vollständigen Operationsausführungen entsprechen und entweder von einer Generatorklasse eines Benutzerrepräsentanten oder der Generatorklasse der Hauptkomponente ausgehen, auf „Konsistenz“ zu untersuchen. Der Begriff der „Konsistenz“ eines Aufrufpfades wird in diesem Abschnitt zu einem späteren Zeitpunkt noch formal definiert. Informell besagt er, daß bei Ausführung der Folge von Operationen, die dem Aufrufpfad entspicht, kein Zugriffsverbot auftreten kann. Für die Festlegung der Menge der auf Konsistenz zu untersuchenden Aufrufpfade ist analog zur Definition der Paare auf Widerspruchsfreiheit zu analysierender Zugriffsrestriktionsausdrücke (vgl. Definition (5.19) in Abschnitt 5.3.2.2) die „Trennlinie“, die die Generatorklassen für Benutzerrepräsentanten in dem statischen Aufrufgraphen bilden, zu berücksichtigen. Dies besagt, daß neben den vollständigen Aufrufpfaden, die von einer Generatorklasse eines Benutzerrepräsentanten ausgehen, lediglich noch die Aufrufpfade zu analysieren sind, die von der Generatorklasse der Hauptkomponente ausgehen und keine Generatorklasse eines Benutzerrepräsentanten enthalten. In der folgenden Definition wird nun die Menge der auf Konsistenz zu untersuchenden Aufrufpfade formal präzisiert.

Definition 5.27.: Menge auf Konsistenz zu analysierender Aufrufpfade (*ConsPaths*)

Gegeben seien ein INSEL$^+$–Programm $\mathcal{P} = (\mathcal{D}, \delta, \gamma, \iota)$ und der attributierte statische Aufrufgraph $\mathcal{G}(\mathcal{P}) = (V, K, \mu)$ von $\mathcal{P}$. $\mathcal{G}(\mathcal{P})$ sei zyklenfrei. Die Menge

$$ConsPaths(\mathcal{P}) \subset AP(\mathcal{G}(\mathcal{P}))$$

der auf Konsistenz zu analysierenden Aufrufpfade von $\mathcal{G}(\mathcal{P})$ ist wie folgt definiert:

$$ConsPaths(\mathcal{P}) \triangleq AP(BenRep(\mathcal{P})) \cup AP(HKomp(\mathcal{P}))$$

mit

$$AP(BenRep(\mathcal{P})) \triangleq \{(v_0, v_1, \ldots, v_n) \in AP(\mathcal{G}(\mathcal{P})) \mid kindof(v_0) = BenRep \wedge \nexists\, v \in V : (v_n, v) \in K\}$$

$$AP(HKomp(\mathcal{P})) \triangleq \{(v_0, v_1, \ldots, v_n) \in AP(\mathcal{G}(\mathcal{P})) \mid kindof(v_0) = HKomp \wedge \nexists\, v \in V : (v_n, v) \in K \wedge \forall i \in \{1, \ldots, n-1\} : kindof(v_i) \neq BenRep\}$$

□

Gemäß Definition (5.27) gehören zur Menge $ConsPaths(\mathcal{P})$

- alle Aufrufpfade, die von einer Generatorklasse eines Benutzerrepräsentanten ausgehen und einen Knoten, der keine Nachfolger mehr hat, als Endknoten haben [Menge $AP(BenRep(\mathcal{P}))$] und
- alle Aufrufpfade, die von der Generatorklasse der Hauptkomponente ausgehen und einen Knoten, der keine Nachfolger mehr hat, als Endknoten haben und die entweder keine Generatorklasse eines Benutzerrepräsentanten oder eine solche höchstens als Endknoten enthalten [Menge $AP(HKomp(\mathcal{P}))$].

Die Voraussetzung, daß $\mathcal{G}(\mathcal{P})$ zyklenfrei ist, wird hier lediglich aus Vereinfachungsgründen angenommen. Würde $\mathcal{G}(\mathcal{P})$ zyklische Pfade enthalten, so wären diese im Rahmen der Analyse der absoluten Konsistenz speziell zu berücksichtigen. Darauf wird hier jedoch nicht weiter eingegangen, da durch die Berücksichtigung zyklischer Pfade bei der Beschreibung des Analyseverfahrens keine grundsätzlich neuen Erkenntnisse vermittelt würden und zum anderen dadurch die Beschreibung der Analysen unnötig verkompliziert würde. Ist $p = (v_0, v_1, \ldots, v_n)$ ein Aufrufpfad mit $p \in ConsPaths(\mathcal{P})$, so kann p insbesondere Knoten enthalten, denen ein Zugriffsrestriktionsausdruck zugeordnet ist. Gilt für einen Knoten v_i mit $i \in \{0, \ldots, n\}$: $zra(v_i) = true$, wird der v_i zugeordnete Zugriffsrestriktionsausdruck im weiteren mit R_i bezeichnet[29].

Beispiel

Als Beispiel ist im folgenden die Menge $ConsPaths(KVS)$ des INSEL$^+$–Programms angegeben, das das Kontenverwaltungssystem implementiert (vgl. auch Abbildung 5.5)[30].

$ConsPaths(KVS) =$
{(Bankleiter, SetzeMaxKredit), (Bankleiter, BerechneKredit, LeseMaxKredit),
(Bankleiter, BerechneKredit, ZeigeKontobewegungen, ListeKontoBewegungen),
(Bankleiter, BerechneKredit, LesePersKundendaten),
(Bankleiter, BerechneKredit, LeseKontostand),
(Bankleiter, ErmittleKonto, GibKontoZeiger, SucheKonto, LeseKontonummer),
(Kundenbetreuer, BerechneKredit, LeseMaxKredit),
(Kundenbetreuer, BerechneKredit, ZeigeKontobewegungen, ListeKontoBewegungen),
(Kundenbetreuer, BerechneKredit, LesePersKundendaten),
(Kundenbetreuer, BerechneKredit, LeseKontostand),
(Kundenbetreuer, KontoAufloesen, Aufloesen),
(Kundenbetreuer, AendereKundenDaten), (Kundenbetreuer, LeseKontostand),

[29] Zur Erinnerung: In dem Fall, daß v_i DA–Generatorklasse ist, ist dies der für v_i festgelegte Zugriffsrestriktionsausdruck. Ist v_i benannte DA–Inkarnationsklasse, ist dies der Zugriffsrestriktionsausdruck, der für die DA–Generatorklasse festgelegt ist, auf die sich v_i bezieht.

[30] Anmerkung: In der Menge sind nicht alle Aufrufpfade, die mit der Benutzerverwaltung zusammenhängen, angegeben.

$(\texttt{Kundenbetreuer}, \texttt{KontoEroeffnen}, \texttt{KontoTyp}, \texttt{Kontobewegungsliste}),$
$(\texttt{Kundenbetreuer}, \texttt{ErmittleKonto}, \texttt{GibKontoZeiger}, \texttt{SucheKonto}, \texttt{LeseKontonummer}),$
$(\texttt{Kundenbetreuer}, \texttt{Ueberweiser}, \texttt{Abheben}, \texttt{FuegeEin}),$
$(\texttt{Kundenbetreuer}, \texttt{Ueberweiser}, \texttt{Einzahlen}, \texttt{FuegeEin}),$
$(\texttt{Kunde}, \texttt{ErmittleKonto}, \texttt{GibKontoZeiger}, \texttt{SucheKonto}, \texttt{LeseKontonummer}),$
$(\texttt{Kunde}, \texttt{LeseKontostand}), (\texttt{Kunde}, \texttt{ZeigeKontoauszug}, \texttt{ListeKontoauszug}),$
$(\texttt{Kunde}, \texttt{Abheben}, \texttt{FuegeEin}),$
$(\texttt{Kunde}, \texttt{Ueberweiser}, \texttt{Abheben}, \texttt{FuegeEin}),$
$(\texttt{Kunde}, \texttt{Ueberweiser}, \texttt{Einzahlen}, \texttt{FuegeEin}),$
$(\texttt{Kassierer}, \texttt{ErmittleKonto}, \texttt{GibKontoZeiger}, \texttt{SucheKonto}, \texttt{LeseKontonummer}),$
$(\texttt{Kassierer}, \texttt{Abheben}, \texttt{FuegeEin}), (\texttt{Kassierer}, \texttt{Einzahlen}, \texttt{FuegeEin})\}$

$\bigcup$

$\{(\texttt{KontoVerwaltungsSystem}, \texttt{LoginAtTerminal}, \texttt{CreateUserRepForRole}, \texttt{Bankleiter}),$
$(\texttt{KontoVerwaltungsSystem}, \texttt{LoginAtTerminal}, \texttt{CreateUserRepForRole}, \texttt{Kundenbetreuer}),$
$(\texttt{KontoVerwaltungsSystem}, \texttt{LoginAtTerminal}, \texttt{CreateUserRepForRole}, \texttt{Kunde}),$
$(\texttt{KontoVerwaltungsSystem}, \texttt{LoginAtTerminal}, \texttt{CreateUserRepForRole}, \texttt{Kassierer}),$
$(\texttt{KontoVerwaltungsSystem}, \texttt{LoginAtTerminal}, \texttt{CreateUserRepForRole}, \texttt{Sysadmin}),$
$(\texttt{KontoVerwaltungsSystem}, \texttt{KontoVerwalter}, \texttt{Zinsberechnung}, \texttt{Zinsberechner},$
$\texttt{BerechneZinsen}, \texttt{LeseKontostand}),$
$(\texttt{KontoVerwaltungsSystem}, \texttt{KontoVerwalter}, \texttt{Zinsberechnung}, \texttt{Zinsberechner},$
$\texttt{Abheben}, \texttt{FuegeEin}),$
$(\texttt{KontoVerwaltungsSystem}, \texttt{KontoVerwalter}, \texttt{Zinsberechnung}, \texttt{Zinsberechner},$
$\texttt{Einzahlen}, \texttt{FuegeEin}),$
$(\texttt{KontoVerwaltungsSystem}, \texttt{InitRoles}, \texttt{Insert}),$
$(\texttt{KontoVerwaltungsSystem}, \texttt{InitTerminals}, \texttt{Insert}),$
$(\texttt{KontoVerwaltungsSystem}, \texttt{UserManagement}, \texttt{RoleSet}),$
$(\texttt{KontoVerwaltungsSystem}, \texttt{UserManagement}, \texttt{TerminalNameSet}),$
$(\texttt{KontoVerwaltungsSystem}, \texttt{UserManagement}, \texttt{UserDataManager})\}$

◇

In der folgenden Definition wird nun für Aufrufpfade aus der Menge $ConsPaths(\mathcal{P})$ der Begriff der Konsistenz formal eingeführt.

Definition 5.28.: Konsistenz eines Aufrufpfades $p \in ConsPaths(\mathcal{P})$

Seien $\mathcal{P}$, $\mathcal{G}(\mathcal{P})$ und $ConsPaths(\mathcal{P})$ wie in Definition (5.27) gegeben. Weiter sei $p \in ConsPaths(\mathcal{P})$ ein Aufrufpfad mit $p = (v_0, v_1, \ldots, v_n)$. p ist **konsistent** genau dann, wenn

1. $p \in AP(BenRep(\mathcal{P}))$ und
   (a) $n = 1$ ist, d.h. p ein Pfad der Länge 1 ist oder
   (b) $n > 1$ ist und für alle Knoten v_i mit $i \in \{2, \ldots, n\}$ gilt: $zra(v_i) = false$ oder
   (c) $n > 1$ ist und es mindestens einen Knoten v_i mit $i \in \{2, \ldots, n\}$ und $zra(v_i) = true$ gibt, und es gilt:
      i. $zra(v_1) = true$ und
      ii. für alle R_i mit $i \in \{2, \ldots, n\}$ und $zra(v_i) = true$ gilt:
         $R_1 \wedge \texttt{Caller.Role} = v_0$ und R_i sind konsistent bzgl. des Aufrufpfades $p' = (v_1, \ldots, v_i)$

   oder

i'. $zra(v_1) = false$ und

i". für alle R_i mit $i \in \{2, \ldots, n\}$ und $zra(v_i) = true$ gilt:
$\texttt{Caller.Role} = v_0$ und R_i sind konsistent bzgl. des Aufrufpfades $p' = (v_1, \ldots, v_i)$

2. $p \in AP(HKomp(\mathcal{P}))$ und

   (a) für alle Knoten v_i mit $i \in \{1, \ldots, n\}$ gilt: $zra(v_i) = false$ oder

   (b) für alle R_i mit $i \in \{1, \ldots, n\}$ und $zra(v_i) = true$ gilt:
   $\texttt{Caller.User} = \texttt{System} \wedge \texttt{Caller.Role} = \texttt{SystemRole}$ und R_i sind konsistent bzgl. des Aufrufpfades $p' = (v_0, \ldots, v_i)$

□

Ist ein Aufrufpfad $p \in AP(BenRep(\mathcal{P}))$ mit $p = (v_0, v_1, \ldots, v_n)$ gemäß Definition (5.28) konsistent, so gilt: Wenn ein Benutzerrepräsentant (d.h. ein Benutzer) das Recht zur Ausführung der *erzeuge*–Operation auf einem Generator bzgl. v_1[31] hat, so hat er im Kontext der Ausführung der kanonischen Operation der dadurch erzeugten Inkarnation auch das Recht zur Ausführung aller gemäß des Pfades p potentiell im Kontext dieser Operation auszuführenden Operationen. Sind alle Pfade $p \in AP(BenRep(\mathcal{P}))$, die mit v_0 und v_1 beginnen, also $p = (v_0, v_1, \ldots)$ konsistent, so ist gewährleistet, daß bei Ausführung der kanonischen Operation einer Inkarnation bzgl. v_1 durch einen Benutzerrepräsentanten keine Zugriffsverbote für die im Kontext dieser Operation auszuführenden Operationen auftreten und die kanonische Operation somit stets vollständig ausgeführt werden kann. Für einen von der Generatorklasse der Hauptkomponente ausgehenden konsistenten Aufrufpfad $p \in AP(HKomp(\mathcal{P}))$ gilt analog, daß eine von der Hauptkomponente gemäß des Pfades p angestoßene und begonnene Operationsausführung stets vollständig ausgeführt werden kann und nicht aufgrund eines fehlenden Rechts für eine im Kontext der Operationsausführung auszuführende Operation abgebrochen werden muß. Die Gültigkeit dieser Aussagen wird im folgenden anhand der einzelnen in Definition (5.28) für die Konsistenz eines Aufrufpfades aufgeführten Bedingungen näher begründet und anhand von Beispielen aus der Menge der Aufrufpfade $ConsPaths(KVS)$ des Kontenverwaltungssystems verdeutlicht.

– Bedingung 1.(a)
  In diesem Fall ist p ein von einer Generatorklasse eines Benutzerrepräsentanten ausgehender vollständiger Aufrufpfad der Länge 1. Aufgrund der Pfadlänge von 1 werden bei Ausführung der kanonischen Operation einer Inkarnation bzgl. v_1 keine weiteren DA–Inkarnationen erzeugt, d.h. es werden keine *erzeuge*–Operationen auf DA–Generatoren aufgerufen. v_1 kann zugriffskontrolliert sein oder nicht. Ist für v_1 ein Zugriffsrestriktionsausdruck R_1 festgelegt, so ist dieser bei Aufruf von *erzeuge* auf v_1 auszuwerten. Wird R_1 zu *true* ausgewertet, hat der aufrufende Benutzerrepräsentant das Recht zur Ausführung von *erzeuge* auf v_1. Im Kontext der Ausführung der kanonischen Operation einer Inkarnation bzgl. v_1 sind keine weiteren Zugriffsrestriktionsausdrücke auszuwerten, so daß also auch keine Zugriffsverbote auftreten können.

  Beispiel

  Der Aufrufpfad $(\texttt{Bankleiter}, \texttt{SetzeMaxKredit}) \in ConsPaths(KVS)$ ist konsistent.

◇

[31] O.b.d.A wird hier angenommen, daß v_1 eine DA–Generatorklasse ist.

– Bedingung 1.(b)
In diesem Fall ist p ein von einer Generatorklasse eines Benutzerrepräsentanten ausgehender vollständiger Aufrufpfad der Länge größer 1, in dem neben der Benutzerrepräsentanten–Generatorklasse v_0 höchstens der Knoten v_1 zugriffskontrolliert ist. Die Pfadlänge größer 1 bedeutet, daß im Kontext der Ausführung der kanonischen Operation einer Inkarnation bzgl. v_1 potentiell die *erzeuge*–Operationen auf DA–Generatoren der Generatorklassen aufgerufen werden, die in dem Teilpfad $p'' = (v_2, \dots, v_n)$ enthalten sind. Da die Knoten von p'' nicht zugriffskontrolliert sind und damit für diese *erzeuge*–Operationen keine Zugriffsrestriktionsausdrücke festgelegt sind, können bei Ausführung des Teils der kanonischen Operation einer Inkarnation bzgl. v_1, der dem Pfad p'' entspricht, keine Zugriffsverbote auftreten.

Beispiel

Die Menge $ConsPaths(KVS)$) enthält keinen Aufrufpfad, für den diese Bedingung erfüllt ist. Wäre jedoch die DA–Generatorklasse `ListeKontoauszug` nicht zugriffskontrolliert, so wäre die Bedingung für den Aufrufpfad (`Kunde`, `ZeigeKontoauszug`, `ListeKontoauszug`) erfüllt, der damit konsistent wäre.

◇

– Bedingung 1.(c) mit i. und ii.
In diesem Fall ist p ein von einer Generatorklasse eines Benutzerrepräsentanten ausgehender vollständiger Aufrufpfad der Länge größer 1, dessen Knoten v_1 zugriffskontrolliert ist und in dem für jeden Zugriffsrestriktionsausdruck R_i, der einem zugriffskontrollierten Knoten v_i mit $i \in \{2, \dots, n\}$, von dem es mindestens einen gibt, zugeordnet ist, gilt, daß $R_1 \wedge \texttt{Caller.Role} = v_0$ und R_i konsistent bzgl. des Aufrufpfades $p' = (v_1, \dots, v_i)$ sind. Die Pfadlänge größer 1 bedeutet wiederum, daß im Kontext der Ausführung der kanonischen Operation einer Inkarnation bzgl. v_1 potentiell die *erzeuge*–Operationen auf DA–Generatoren der Generatorklassen aufgerufen werden, die in dem Teilpfad $p'' = (v_2, \dots, v_n)$ enthalten sind. Die Knoten von p'' können zugriffskontrolliert sein. Da für den Zugriffsrestriktionsausdruck R_i eines zugriffskontrollierten Knotens v_i aus p'' gilt, daß $R_1 \wedge \texttt{Caller.Role} = v_0$ und R_i konsistent bzgl. des Aufrufpfades p' sind, folgt mit Definition (5.25), daß R_i im Kontext der Ausführung der kanonischen Operation einer Inkarnation bzgl. v_1 den Wert *true* hat. Da dies analog für die Zugriffsrestriktionsausdrücke aller anderen zugriffskontrollierten Knoten aus p'' gilt, folgt unmittelbar, daß bei Ausführung des Teils der kanonischen Operation einer Inkarnation bzgl. v_1, der dem Pfad p'' entspricht, keine Zugriffsverbote auftreten.
Die UND–Verknüpfung von R_1 mit der Rollenrestriktion $\texttt{Caller.Role} = v_0$ ist erforderlich, um im Rahmen der Analyse der Konsistenz der Zugriffsrestriktionsausdrücke R_1 und R_i bzgl. des Aufrufpfades p' auch die Fälle korrekt zu erfassen, in denen in R_1 diese Rollenrestriktion nicht explizit spezifiziert ist.

Beispiel

Der Aufrufpfad (`Kundenbetreuer`, `KontoAufloesen`, `Aufloesen`) $\in ConsPaths(KVS)$ wäre unter den im Beispiel auf Seite 224 genannten Voraussetzungen, daß

– der Zugriffsrestriktionsausdruck von `Aufloesen` keine variablen DE–Inkarnationen enthielte und
– das Kontenverwaltungssystem keinen `ChangeACL`–Aufruf bzgl. des mit `KontoZeiger` identifizierten Konto–Depots enthielte

konsistent.

◇

– Bedingung 1.(c) mit i'. und ii'.
In diesem Fall ist p ein von einer Generatorklasse eines Benutzerrepräsentanten ausgehender vollständiger Aufrufpfad der Länge größer 1, dessen Knoten v_1 nicht zugriffskontrolliert ist und in dem für jeden Zugriffsrestriktionsausdruck R_i, der einem zugriffskontrollierten Knoten v_i mit $i \in \{2, \ldots, n\}$, von dem es mindestens einen gibt, zugeordnet ist, gilt, daß `Caller.Role` $= v_0$ und R_i konsistent bzgl. des Aufrufpfades $p' = (v_1, \ldots, v_i)$ sind. Dieser Fall unterscheidet sich von der vorhergehenden Bedingung dadurch, daß für v_1 kein Zugriffsrestriktionsausdruck festgelegt ist. Für die Ausführung der *erzeuge*–Operation auf v_1 sind also keine Zugriffsbeschränkungen explizit spezifiziert, obwohl für die gemäß des Teilpfades $p'' = (v_2, \ldots, v_n)$ im Kontext der Ausführung der kanonischen Operation einer Inkarnation bzgl. v_1 potentiell auszuführenden Operationen Zugriffsrestriktionsausdrücke festgelegt sein können. Aus der Bedingung, daß die Rollenrestriktion `Caller.Role` $= v_0$, die im Kontext der durch einen Benutzerrepräsentanten bzgl. v_0 ausgeführten kanonischen Operation einer Inkarnation bzgl. v_1 stets erfüllt ist, und die entsprechenden Zugriffsrestriktionsausdrücke R_i konsistent bzgl. p' sind, folgt, daß bei Ausführung des Teils der kanonischen Operation einer Inkarnation bzgl. v_1, der dem Pfad p'' entspricht, keine Zugriffsverbote auftreten.

Beispiel

Der Aufrufpfad (`Kunde, ErmittleKonto, GibKontoZeiger, SucheKonto, LeseKontonummer`) $\in ConsPaths(KVS)$ ist konsistent, da

- die DA–Generatorklasse `ErmittleKonto` nicht zugriffskontrolliert ist,
- die Rollenrestriktion `Caller.Role = Kunde` und der Zugriffsrestriktionsausdruck `NOT(Caller.Role = SysAdmin)` der K-Order–Generatorklasse `GibKontoZeiger` konsistent bzgl. des Aufrufpfades p' = (`ErmittleKonto`, `GibKontoZeiger`) sind (vgl. Bedingung 3. aus Satz (5.6)),
- die DA–Generatorklasse `SucheKonto` nicht zugriffskontrolliert ist und
- die Rollenrestriktion `Caller.Role = Kunde` und der Zugriffsrestriktionsausdruck `Caller.ActorName = KontoVerwalter` der S-Order–Generatorklasse `LeseKontonummer` konsistent bzgl. des Aufrufpfades p''' = (`ErmittleKonto, GibKontoZeiger, SucheKonto, LeseKontonummer`) sind, weil die Kontextrestriktion `Caller.ActorName = KontoVerwalter` mit p''' erfüllbar ist (vgl. Beispiel auf Seite 212)

◇

– Bedingung 2.(a)
In diesem Fall ist p ein von der Generatorklasse der Hauptkomponente ausgehender vollständiger Aufrufpfad, der keine zugriffskontrollierten Knoten enthält. Da die Knoten von $p'''' = (v_1, \ldots, v_n)$ nicht zugriffskontrolliert sind, können bei Ausführung des Teils der kanonischen Operation der Hauptkomponente, der dem Pfad p'''' entspricht, keine Zugriffsverbote auftreten.

– Bedingung 2.(b)
In diesem Fall ist p ein von der Generatorklasse der Hauptkomponente ausgehender vollständiger Aufrufpfad, dessen Knoten zugriffskontrolliert sein können und in

dem für jeden Zugriffsrestriktionsausdruck R_i, der einem zugriffskontrollierten Knoten v_i mit $i \in \{1, \ldots, n\}$ zugeordnet ist, gilt, daß $R_{hk} = (\texttt{Caller.User} = \texttt{System} \wedge \texttt{Caller.Role} = \texttt{SystemRole})$ und R_i konsistent bzgl. des Aufrufpfades $p' = (v_0, \ldots, v_i)$ sind. Der Zugriffsrestriktionsausdruck R_{hk} hat im Kontext der Ausführung einer Folge von Operationen gemäß des Aufrufpfades p stets den Wert *true*. Aus der Bedingung, daß dieser Zugriffsrestriktionsausdruck und der einem zugriffskontrollierten Knoten v_i aus p zugeordnete Zugriffsrestriktionsausdruck R_i konsistent bzgl. des Aufrufpfades p' sind, folgt, daß die Hauptkomponente im Kontext der Ausführung ihrer kanonischen Operation das Recht zur Ausführung aller gemäß des Pfades p auszuführenden Operationen hat und somit bei Ausführung des Teils der kanonischen Operation der Hauptkomponente, der dem Pfad p entspricht, keine Zugriffsverbote auftreten.

Beispiel

Der Aufrufpfad $p =$ (`KontoVerwaltungsSystem`, `KontoVerwalter, Zinsberechnung, Zinsberechner, Einzahlen, FuegeEin`) $\in ConsPaths(KVS)$ ist konsistent, da

- $R_{hk} = (\texttt{Caller.User} = \texttt{System} \wedge \texttt{Caller.Role} = \texttt{SystemRole})$ und der Zugriffsrestriktionsausdruck `Caller.User = System` der DA–Generatorklasse `KontoVerwalterTyp` konsistent bzgl. des Aufrufpfades $p' =$ (`KontoVerwaltungsSystem`, `KontoVerwalter`) sind (vgl. Bedingung 1. aus Satz (5.6)),
- R_{hk} und der Zugriffsrestriktionsausdruck `Caller.ConName = KontoVerwalter` der DA–Generatorklasse `Zinsberechnung` konsistent bzgl. des Aufrufpfades $p'' =$ (`KontoVerwaltungsSystem`, `KontoVerwalter, Zinsberechnung`) sind, weil die Kontextrestriktion `Caller.ConName = KontoVerwalter` mit p'' erfüllbar ist,
- die DA–Generatorklasse `Zinsberechner` nicht zugriffskontrolliert ist,
- R_{hk} und $R_{\texttt{Einzahlen}}$ konsistent bzgl. des Aufrufpfades $p''' =$ (`KontoVerwaltungsSystem, KontoVerwalter, Zinsberechnung, Zinsberechner, Einzahlen`) sind (Begründung siehe Beispiel auf Seite 224) und
- R_{hk} und $R_{\texttt{FuegeEin}}$ konsistent bzgl. des Aufrufpfades p sind (vgl. Beispiel auf Seite 209).

◇

Wie aus den Bedingungen 1.(c) und 2.(b) der Definition (5.28) ersichtlich ist, werden für die Überprüfung der Konsistenz eines Aufrufpfades $p \in ConsPaths(\mathcal{P})$ im allgemeinen die in Abschnitt 5.3.3.1 angegebenen Kriterien für den statischen Nachweis der Konsistenz zweier Zugriffsrestriktionsausdrücke bzgl. eines Aufrufpfades benötigt. Aus den bisherigen Aussagen dieses Abschnitts läßt sich zusammenfassend der folgende Satz, auf dessen Basis die absolute Konsistenz des Rechts eines INSEL$^+$–Systems überprüft werden kann, ableiten.

Satz 5.7.: Absolute Konsistenz des Rechts eines INSEL$^+$–Systems

Es seien die Voraussetzungen wie in Definition (5.28) gegeben. Dann gilt:

Das Recht des durch $\mathcal{P}$ definierten INSEL$^+$–Systems ist absolut konsistent, wenn für alle $p \in ConsPaths(\mathcal{P})$ gilt: p ist konsistent.

△

Beispiel

Das Recht des Kontenverwaltungssystems ist nicht absolut konsistent, da z.B. der Aufrufpfad (`Kundenbetreuer`, `KontoAufloesen, Aufloesen`) $\in ConsPaths(KVS)$ nicht konsistent ist (vgl. Beispiele auf den Seiten 224 und 230).

◇

Das Verfahren für die Analyse der absoluten Konsistenz des Rechts eines INSEL$^+$–Systems ist in Abbildung 5.10 zusammenfassend dargestellt.

Zunächst werden anhand des attributierten statischen Aufrufgraphen des entsprechenden INSEL$^+$–Programms $\mathcal{P}$ alle Aufrufpfade ermittelt, die zu der Menge $ConsPaths(\mathcal{P})$ gehören. Anschließend werden nacheinander alle Aufrufpfade $p \in ConsPaths(\mathcal{P})$ auf Konsistenz überprüft. Sind alle Pfade aus $ConsPaths(\mathcal{P})$ konsistent, so ist das Recht des durch $\mathcal{P}$ definierten INSEL$^+$–Systems absolut konsistent. Anderenfalls, d.h. gibt es mindestens einen Pfad aus $ConsPaths(\mathcal{P})$, der nicht konsistent ist, ist das Recht des INSEL$^+$–Systems nicht absolut konsistent sondern lediglich potentiell konsistent.

Ein Aufrufpfad $p \in ConsPaths(\mathcal{P})$ wird auf Konsistenz analysiert, indem überprüft wird, ob für p eine der in Definition (5.28) angegebenen Bedingungen erfüllt ist. Ist bei dieser Überprüfung zu untersuchen, ob zwei Zugriffsrestriktionsausdrücke konsistent bzgl. eines Aufrufpfades sind, so erfolgt diese Analyse nach dem hierfür in Abschnitt 5.3.3.1 angegebenen Analyseverfahren (vgl. Seite 218).

Da mit der Analyse der absoluten Konsistenz des Rechts eines INSEL$^+$–Systems das Ziel verfolgt wird, die Konsistenz des Rechts des Systems statisch, d.h. bereits zur Übersetzungszeit des entsprechenden INSEL$^+$–Programms nachzuweisen, sind die Bedingungen und Kriterien, die im Rahmen dieser Analyse überprüft werden, recht restriktiv. Wie insbesondere aus den in Satz (5.6) angegebenen Kriterien für die Äquivalenz zweier Literale ersichtlich ist, sind die Möglichkeiten für die Vergabe dynamischer Rechte in einem INSEL$^+$–System mit absolut konsistentem Recht stark eingeschränkt. Es ist dennoch sinnvoll, Kriterien für den statischen Nachweis der Konsistenz des Rechts eines INSEL$^+$–Systems zu erarbeiten. Zum einen besteht bei erfolgreichem Nachweis der absoluten Konsistenz des Rechts eines INSEL$^+$–Systems bereits zur Übersetzungszeit des entsprechenden INSEL$^+$–Programms die Gewißheit, daß zur Laufzeit des Systems keine Inkonsistenzen in dem Recht und damit auch keine Ausführungsabbrüche begonnener Operationsausführungen aufgrund fehlender Rechte auftreten können. Dies hat insbesondere den Vorteil, daß in dem INSEL$^+$–Programm keine komplexen Maßnahmen zur Behandlung von Zugriffsverboten vorgesehen werden müssen. Zum anderen gibt es Anwendungssysteme, deren Zugriffskontrollpolitik relativ statisch ist und die so konstruiert werden können, daß die Kriterien für die absolute Konsistenz des Rechts erfüllt sind. Zur Unterstützung der systematischen Konstruktion derartiger INSEL$^+$–Systeme lassen sich aus den Kriterien für die absolute Konsistenz Leitlinien ableiten, an denen sich ein Systementwickler orientieren kann, wenn er ein INSEL$^+$–System mit absolut konsistentem Recht entwickeln möchte. Diese Leitlinien werden in dem nun folgenden Abschnitt angegeben.

5.4 Leitlinien für die Konstruktion von INSEL$^+$–Systemen mit absolut konsistentem Recht

Die in dem vorhergehenden Abschnitt angegebenen Bedingungen und Kriterien, die im Rahmen der Analyse der absoluten Konsistenz des Rechts eines INSEL$^+$–Systems überprüft werden, liefern die Basis für die Angabe von Leitlinien für die zielgerichtete und systematische

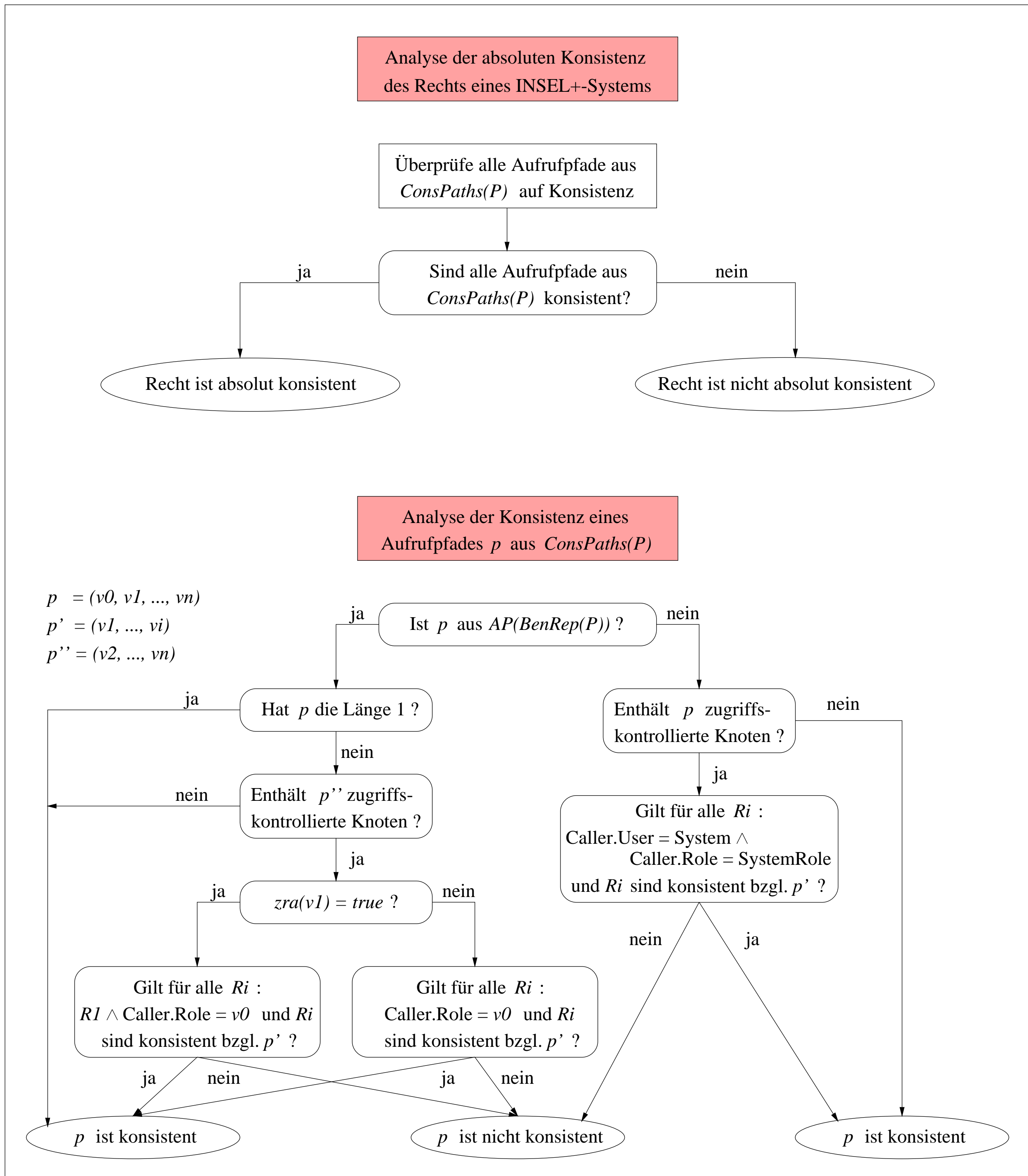

Abbildung 5.10: Analyse der absoluten Konsistenz des Rechts eines INSEL$^+$–Systems

Konstruktion von INSEL$^+$–Systemen mit absolut konsistentem Recht. Gemäß Satz (5.7) ist ein INSEL$^+$–System, dessen Recht absolut konsistent sein soll, so zu konstruieren, daß die von den Generatorklassen der Benutzerrepräsentanten und der Hauptkomponente ausgehenden vollständigen Aufrufpfade konsistent sind. Dem entsprechend ist mit den in Definition

(5.28) angegebenen Bedingungen, die erfüllt sein müssen, damit ein derartiger Aufrufpfad konsistent ist, das Spektrum der Möglichkeiten für die Konstruktion von INSEL$^+$–Systemen mit absolut konsistentem Recht festgelegt. Diese Möglichkeiten und die sich daraus ergebenen Leitlinien werden im folgenden zunächst jeweils unabhängig voneinander erläutert. Bei der Konstruktion eines INSEL$^+$–Systems, dessen Recht absolut konsistent sein soll, sind diese einzelnen Möglichkeiten je nach funktionalen Anforderungen an das System und der zu realisierenden Zugriffskontrollpolitik im allgemeinen in Kombination und aufeinander abgestimmt einzusetzen.

Ausgehend von Definition (5.28) und Satz (5.7) lassen sich folgende Leitlinien für die Konstruktion von INSEL$^+$–Systemen mit absolut konsistentem Recht angeben.

L1. Die äußeren Operationen zugriffskontrollierter Komponenten, die unmittelbar von Benutzerrepräsentanten aus nutzbar sind, sind – soweit dies unter Funktionalitäts- sowie Modularisierungs- und Strukturierungsaspekten möglich und sinnvoll ist – so zu konstruieren, daß bei ihrer Ausführung keine weiteren DA–Inkarnationen erzeugt werden.

Diese Leitlinie ist aus Bedingung 1.(a) der Definition (5.28) abgeleitet. Sie wird in der Regel jedoch nur vereinzelt anwendbar sein, da sie zum einen dem Prinzip einer möglichst weitgehenden Modularisierung und Strukturierung eines INSEL$^+$–Programms entgegenläuft und zum anderen mit derart konstruierten Operationen die Möglichkeiten zur Durchführung komplexer Berechnungen recht eingeschränkt sind. Dies ist darin begründet, daß in derartigen Operationen lediglich Operationen auf DE–Inkarnationen ausgeführt werden können und somit die Nutzung weiterer DA–Inkarnationen sowie die parallele Ausführung von Teilberechnungen durch die Erzeugung entprechender Akteure nicht möglich sind.

Beispiel

Diese Leitlinie hätte zum Beispiel bei der Implementierung der auf den Konto–Depots des Kontenverwaltungssystems definierten Operationen `Einzahlen`, `Abheben`, `ZeigeKontobewegungen` sowie `ZeigeKontoauszug` berücksichtigt werden können. Bei Ausführung dieser Operationen wird jeweils eine Operation auf dem lokal zu jedem Konto–Depot für die Speicherung der Kontobewegungen deklarierten zugriffskontrollierten benannten Depot `Kontobewegungsliste` aufgerufen. Um diesen zusätzlichen Operationsaufruf zu vermeiden, hätte die in der vorliegenden Implementierung (siehe Anhang B) in dem gesonderten Depot `Kontobewegungsliste` definierte Liste, in der die Kontobewegungen gespeichert werden, auch direkt lokal zu dem Konto–Depot deklariert werden können. Die Anweisungen der auf dem Depot `Kontobewegungsliste` definierten Zugriffsoperationen für diese Liste hätten dann direkt in dem Anweisungsteil der o.g. Operationen eines Konto–Depots implementiert werden müssen. Nachteilig hierbei wäre gewesen, daß in den Operationen `Einzahlen` und `Abheben` jeweils identische Anweisungen für das Einfügen eines entsprechenden Eintrags in die Liste der Kontobewegungen erforderlich gewesen wären.

◇

Die folgende Leitlinie ergibt sich aus Bedingung 1.(b) der Definition (5.28).

L2. Nach Möglichkeit sind nur die von den Benutzerrepräsentanten unmittelbar nutzbaren Komponenten als zugriffskontrollierte Komponenten zu konstruieren. Die Anzahl der darüber hinaus als zugriffskontrolliert definierten Komponenten ist – insbesondere unter Einsatz der für die differenzierte Festlegung von Ausführungsumgebungen zur Verfügung stehenden Konzepte – so weit wie möglich zu reduzieren.

Mit dieser Leitlinie wird das Ziel verfolgt, im Kontext der Ausführung der von Benutzerrepräsentanten direkt aufrufbaren Operationen möglichst keine Operationen auf zugriffskontrollierten Komponenten aufzurufen und somit hierbei auch keine Zugriffsrechtsüberprüfungen durch Auswertung von Zugriffsrestriktionsausdrücken vornehmen zu müssen. Um dieses Ziel vor dem Hintergrund der insgesamt für das System zu realisierenden Zugriffskontrollpolitik zu erreichen, sind zum einen das Schachtelungsprinzip und zum anderen die in INSEL$^+$ zur differenzierten Festlegung von Ausführungsumgebungen zur Verfügung stehenden Konzepte (vgl. Abschnitt 4.3) konsequent einzusetzen. Die Komponenten, die für die Erbringung der Funktionalität der äußeren Operationen einer von einem Benutzerrepräsentanten unmittelbar nutzbaren Komponente benötigt werden, sind somit möglichst lokal zu dieser Komponente zu definieren und damit gegenüber dem direkten Zugriff durch Benutzerrepräsentanten zu kapseln. Durch die bei Orientierung an dieser Leitlinie mögliche Reduktion der Anzahl zugriffskontrolliert definierter Komponenten wird gleichzeitig der Aufwand für die zur Laufzeit des Systems durchzuführenden dynamischen Zugriffskontrollen verringert.

Beispiel

Als Beispiel für die Anwendbarkeit dieser Leitlinie kann wiederum das für die Speicherung der Kontobewegungen eines Kontos vorgesehene zugriffskontrollierte benannte Depot `Kontobewegungsliste` betrachtet werden. Dieses Depot wird in der vorliegenden Implementierung bzgl. des explizit zugriffskontrollierten Depot–Generators `KontobewegungsListenTyp` inkarniert, der lokal zur Hauptkomponente des Kontenverwaltungssystems deklariert ist. Da dieser Depot–Generator und die entsprechenden Inkarnationen jedoch nur innerhalb der Konto–Depots nutzbar sein müssen, hätte dieser Generator auch lokal zu dem Depot–Generator der Konten deklariert werden können. In diesem Fall hätte der Generator nicht zugriffskontrolliert konstruiert werden müssen, da er zum einen aufgrund der Kapselung in dem Depot–Generator der Konten nicht unmittelbar von Benutzerrepräsentanten aus nutzbar gewesen wäre und zum anderen die für die Zugriffsoperationen der Kontobewegungslisten–Depots durchzusetzenden Zugriffsbeschränkungen durch Import–/Exportfestlegungen umsetzbar gewesen wären. Als Folge einer derartigen Konstruktion wären bei Ausführung der Operationen `Einzahlen`, `Abheben`, `ZeigeKontobewegungen` sowie `ZeigeKontoauszug` eines Konto–Depots keine weiteren Operationen zugriffskontrollierter Komponenten aufzurufen. Die alternative Implementierung des Depot–Generators `KontobewegungsListenTyp` inkl. der erforderlichen Import– und Exportfestlegungen ist in Abbildung 5.11 ausschnittsweise angegeben.

◇

Die folgende aus Bedingung 1.(c) der Definition (5.28) abgeleitete Leitlinie liefert einen Orientierungsrahmen dafür, wie die Zugriffsrestriktionsausdrücke funktional abhängiger Operationen zu konstruieren sind.

L3. Der Zugriffsrestriktionsausdruck einer von einem Benutzerrepräsentanten unmittelbar aufrufbaren äußeren Operation *op* einer zugriffskontrollierten Komponente und der Zugriffsrestriktionsausdruck einer im Kontext der Ausführung von *op* potentiell auszuführenden äußeren Operation op' einer weiteren zugriffskontrollierten Komponente sind aufeinander abgestimmt so zu konstruieren, daß die Gültigkeit des für *op* festgelegten Zugriffsrestriktionsausdrucks unter Berücksichtigung des jeweiligen Kontextes, in dem op' bei Ausführung von *op* aufgerufen wird, die Gültigkeit des für op' festgelegten Zugriffsrestriktionsausdrucks impliziert.

Dieser Leitlinie entspricht, daß in einem INSEL$^+$–System mit absolut konsistentem Recht das Recht eines Benutzers bzw. des entsprechenden Benutzerrepräsentanten zur Ausführung

```
PROTECTED DEPOT TYPE SPEC KontoTyp (...)
  IMPORT ...;
  EXPORT Kontobewegungsliste.FuegeEin TO Einzahlen, Abheben;
         Kontobewegungsliste.ListeKontobewegungen TO ZeigeKontoBewegungen;
         Kontobewegungsliste.ListeKontoauszug TO ZeigeKontoauszug;
IS
  PROCEDURE TYPE SPEC Einzahlen (...);
  ...
END KontoTyp;

PROTECTED DEPOT TYPE KontoTyp (...)
  ...
IS
  ----
  DEPOT TYPE SPEC KontobewegungsListenTyp
    ...
  IS
    PROCEDURE TYPE SPEC FuegeEin (...);
    ...
  END KontobewegungsListenTyp;
  ...

  DEPOT Kontobewegungsliste : KontobewegungsListenTyp;
  ----
  ...
  PROCEDURE TYPE Einzahlen (...)
    IMPORT Kontobewegungsliste.FuegeEin, ...;
  IS
    ...
  END Einzahlen;
  ...
BEGIN
...
END KontoTyp;
```

Abbildung 5.11: An Leitlinie 2 orientierte alternative Implementierung des Depot–Generators `KontobewegungsListenTyp`

einer Operation *op* das Recht zur Ausführung aller im Kontext von *op* auszuführenden Operationen implizieren muß. Rechtsfestlegungen, die für eine von einem Benutzerrepräsentanten unmittelbar aufrufbare Operation *op* getroffen werden, implizieren somit Rechtsfestlegungen für alle Operationen, die im Kontext einer *op*–Ausführung potentiell aufgerufen werden.

Für die Angabe konkreterer Regeln, die von einem Systementwickler bei der aufeinander abgestimmten Konstruktion von Zugriffsrestriktionsausdrücken zu beachten sind, seien G_1 und G_2 zwei zugriffskontrollierte DA–Generatorklassen, deren zugeordnete Zugriffsrestriktionsausdrücke R_1 und R_2 gemäß Leitlinie L3. aufeinander abgestimmt zu entwickeln sind, d.h.:

– es gibt mindestens eine Benutzerrepräsentanten–Generatorklasse BR, in der ein Erzeu-

gungskonstrukt bzgl. der Generatorklasse G_1 enthalten ist, und

– es gibt in dem statischen Aufrufgraph des INSEL$^+$–Programms $\mathcal{P}$ mindestens einen Aufrufpfad zwischen G_1 und G_2, also $(G_1, G_2) \in AnalysePairs(\mathcal{P})$.

Die Zugriffsrestriktionsausdrücke R_1 und R_2 sind in Disjunktiver Normalform zu entwerfen, da die im folgenden genannten Regeln nur bezogen auf diese Darstellungsform anwendbar sind. R_1 und R_2 sind gemäß Leitlinie L3. so zu konstruieren, daß sie konsistent bzgl. aller möglichen Aufrufpfade zwischen G_1 und G_2 sind. Laut Definition (5.25) muß somit für jeden dieser möglichen Aufrufpfade p gelten, daß es zu jedem konjunktiven Teilterm L_i von R_1 eine konjunktiven Teilterm K_k aus R_2 gibt, so daß die Gültigkeit von L_i unter Berücksichtigung des durch p festgelegten Kontextes die Gültigkeit von K_k impliziert. Der Systementwickler hat also zunächst ausgehend von der zu realisierenden Zugriffskontrollpolitik und den damit für G_1 gegebenen Rechtsfestlegungen den Zugriffsrestriktionsausdruck R_1 in Disjunktiver Normalform zu konstruieren. Anschließend ist zu jedem konjunkiven Teilterm L_i aus R_1 und jedem möglichen Aufrufpfad p zwischen G_1 und G_2 unter Berücksichtigung der mit der Zugriffskontrollpolitik für G_2 gegebenen Rechtsfestlegungen ein konjunktiver Teilterm K_k von R_2 so zu konstruieren, daß aus der Gültigkeit von L_i unter Berücksichtigung des durch p festgelegten Kontextes stets die Gültigkeit von K_k folgt. Gemäß Satz (5.5) ist dazu der konjunktive Teilterm K_k so zu konstruieren, daß es zu jedem Literal aus K_k, das keine Kontextrestriktion ist, in L_i ein äquivalentes Literal gibt. Ggf. sind dabei Literale, die aufgrund der Festlegungen der Zugriffskontrollpolitik in K_k enthalten sein müssen, geeignet in L_i zu übernehmen.

Unter Berücksichtigung der in Satz (5.6) angegebenen Kriterien für die Äquivalenz zweier Literale sind folgende Regeln bei der Konstruktion des konjunktiven Teilterms K_k zu beachten:

L3.1. Der konjunktive Teilterm K_k darf lediglich Kontextrestriktionen enthalten, die mit dem Aufrufpfad p erfüllbar sind. Die gemäß dieser Regel in K_k möglichen Kontextrestriktionen ergeben sich aus den in Abschnitt 5.3.2.2 (vgl. S. 208ff) angegebenen Kriterien für die Erfüllbarkeit einer Kontextrestriktion mit einem Aufrufpfad.

L3.2. Der konjunktive Teilterm K_k darf in Wertvergleichen lediglich die Namen konstanter DE–Inkarnationsklassen enthalten, d.h. in K_k dürfen Wertvergleiche nur über konstanten DE–Inkarnationen durchgeführt werden. Diese Regel ergibt sich aus den Bedingungen 7.–11. des Satzes (5.6) und ist darin begründet, daß sich der Wert variabler DE–Inkarnationen zwischen dem Zeitpunkt der Auswertung von L_i und dem Zeitpunkt der Auswertung von K_k ändern kann und somit bei Nutzung variabler DE–Inkarnationen in K_k nicht gewährleistet werden kann, daß aus der Gültigkeit von L_i stets die Gültigkeit von K_k folgt. Sind gemäß der Zugriffskontrollpolitik die Rechte für die Nutzung von G_2 abhängig von den Werten variabler DE–Inkarnationen zu vergeben, so sind die entsprechenden Wertvergleiche für diese DE–Inkarnationen bereits in L_i zu spezifizieren. Dies hat allerdings zur Konsequenz, daß Rechteänderungen für G_2, die sich zwischen den Zeitpunkten der Auswertung von L_i und K_k durch Werteänderungen dieser DE–Inkarnationen ergeben, nicht unmittelbar wirksam werden, sondern erst bei der nächsten Auswertung von R_1 durchschlagen. Eine derartige Verzögerung der Wirksamkeit dieser Rechteänderungen muß jedoch akzeptiert werden, sofern das Ziel der Konstruktion eines INSEL$^+$–Systems mit absolut konsistentem Recht beibehalten werden soll. Anderenfalls kann für das Recht des Systems lediglich die potentielle Konsistenz gewährleistet werden. Hieraus ergibt sich auch, daß das Recht des Systems im allgemeinen nicht absolut konsistent konstruiert werden kann, wenn das Recht zur

Ausführung der *erzeuge*–Operation auf G_2 von dem Wert eines oder mehrerer Eingabe– bzw. Ein–Ausgabeparameter des Operationsaufrufs abhängig ist, da die entsprechenden Wertvergleiche über diesen Parametern aufgrund der Sichtbarkeitsregeln nur in K_k formulierbar sind und nicht in L_i übernommen werden können.

L3.3. In dem konjunktiven Teilterm K_k dürfen lediglich Wertvergleiche spezifiziert werden, zu denen es in L_i einen äquivalenten Wertvergleich gibt (siehe hierzu die Bedingungen 7.–11. des Satzes (5.6)). Umgekehrt darf es zu jedem in L_i enthaltenen Wertvergleich über konstanten DE–Inkarnationen höchstens einen hierzu äquivalenten Wertvergleich in K_k geben. Zur Berücksichtigung dieser Regeln sind ggf. die konstanten DE–Inkarnationen, die für die gemäß der Zugriffskontrollpolitik in K_k durchzuführenden Wertvergleiche benötigt werden, soweit global zu definieren, daß die entsprechenden Wertvergleiche auch in L_i formulierbar sind.

L3.4. Der konjunktive Teilterm K_k darf lediglich Rollenrestriktionen enthalten, die zu einer Rollenrestriktionen aus L_i bzw. zu der Rollenrestriktion `Caller.Role = RoleId` äquivalent sind, wobei `RoleId` der in dem Rollen–Part des INSEL$^+$–Programms festgelegte Bezeichner der Rolle ist, für die die Benutzerrepäsentanten–Generatorklasse BR definiert wird. Die damit in K_k möglichen Rollenrestriktionen sind durch die Bedingungen 2. und 3. des Satzes (5.6) festgelegt.

L3.5. Der konjunktive Teilterm K_k darf lediglich Benutzerrestriktionen enthalten, die zu einer Benutzerrestriktion aus L_i äquivalent sind. Laut Bedingung 1. des Satzes (5.6) bedeutet dies insbesondere, daß in den Benutzerrestriktionen von K_k lediglich konstante DE–Inkarnationen bzgl. des vordefinierten Generators `UserUidType` genutzt werden dürfen.

L3.6. In dem konjunktiven Teilterm K_k dürfen lediglich `IN_ACL`–Prädikate spezifiziert werden, zu denen es in L_i ein entsprechend äquivalentes `IN_ACL`–Prädikat gibt. Aus Bedingung 4. des Satzes (5.6) ergibt sich die Anforderung, daß Änderungen in dem mit einem `IN_ACL`–Prädikat aus K_k identifizierten Zugriffskontrolleintrag nicht vorgenommen werden dürfen, da nur dann gewährleistet ist, daß sich der Wert des `IN_ACL`–Prädikats zwischen dem Zeitpunkt der Auswertung von L_i und dem Zeitpunkt der Auswertung von K_k nicht verändern kann. Falls es aufgrund der Festlegungen der Zugriffskontrollpolitik erforderlich ist, den entsprechenden Zugriffskontrollisteneintrag ändern zu können, darf das entsprechende `IN_ACL`–Prädikat somit nur in L_i enthalten sein. Dies bedeutet wiederum, daß nur bei Auswertung von L_i überprüft wird, ob ein für die Nutzung von G_2 erforderlicher Zugriffskontrollisteneintrag vorhanden ist. Rechteänderungen, die sich zwischen den Zeitpunkten der Auswertung von L_i und K_k durch Änderungen des Zugriffskontrollisteneintrags ergeben, werden somit erst bei erneutem Aufruf der *erzeuge*–Operation auf G_1 wirksam. Ob dies akzeptabel ist, ist zwischen den Anforderungen der Zugriffskontrollpolitik und dem Ziel der Konstruktion eines Systems mit absolut konsistentem Recht abzuwägen. Muß gemäß der Anforderungen der Zugriffspolitik eine bzgl. der Nutzung von G_2 vorgenommene Rechteänderung zwingend sofort wirksam werden, so kann für das Recht des Systems lediglich die potentielle Konsistenz gewährleistet werden.

L3.7. Der konjunktive Teilterm K_k darf lediglich `ACCESSED`–Prädikate enthalten, die zu einem `ACCESSED`–Prädikat aus L_i äquivalent sind. Die Möglichkeiten, die damit für die Konstruktion von `ACCESSED`–Prädikaten in K_k bestehen, sind durch die Bedingungen 5. und 6. des Satzes (5.6) festgelegt.

Wie aus den Regeln ersichtlich ist, sind zur Erreichung des Ziels der Konstruktion eines INSEL$^+$–Systems mit absolut konsistentem Recht, im allgemeinen Rechtsfestlegungen, die für die Nutzung von G_2 gelten, für G_1 zu übernehmen. Bedingt durch Festlegungen der Zugriffskontrollpolitik für die Nutzung von G_2 ergeben sich somit bei Konstruktion von K_k im allgemeinen Rückwirkungen auf L_i. Diese bestehen darin, daß Literale, mit denen Rechtsfestlegungen für G_2 durchgesetzt werden sollen, geeignet in L_i aufzunehmen sind, sofern dort noch keine entsprechend äquivalenten Literale vorhanden sind. Darf die *erzeuge*–Operation auf G_1 zum Beispiel in einer beliebigen Rolle und die *erzeuge*–Operation auf G_2 nur in einer bestimmten Rolle ausgeführt werden, so ergibt sich bei Konstruktion von K_k die Anforderung, die entsprechende Rollenrestriktion zusätzlich in L_i aufzunehmen. Sollten derartige Rückwirkungen zu einem Widerspruch zu den bestehenden Literalen in L_i führen (zum Beispiel zu einer eventuell vorhandenen Rollenrestriktion), deutet dies daraufhin, daß bereits in der Spezifikation der zu realisierenden Zugriffskontrollpolitik eine Inkonsistenz vorliegt. Dementsprechend ist dann die Zugriffskontrollpolitik zu überprüfen und ggf. zu modifizieren.

Aus der Anforderung, daß es zu jedem Literal aus K_k, das keine Kontextrestriktion ist, in L_i ein äquivalentes Literal geben muß, ergeben sich auch die Einschränkungen hinsichtlich der Konstruktion von INSEL$^+$–Systemen mit absolut konsistentem Recht. Diese Anforderung setzt zum einen voraus, daß das System so konstruierbar ist, daß alle Komponenten, die zur Formulierung entsprechend äquivalenter Literale in L_i benötigt werden, bereits dort sichtbar und bekannt sind. Dies läßt sich jedoch nicht immer erreichen. Zum Beispiel dann nicht, wenn in K_k als Literal ein Wertvergleich über einem Eingabeparameter von G_2 enthalten sein muß, dessen Wert bei Aufruf der *erzeuge*–Operation auf G_1 noch nicht bekannt ist, sondern erst im Laufe einer G_1–Ausführung berechnet wird (vgl. hierzu auch L3.2.). Zum anderen hat die obige Anforderung zur Konsequenz, daß in K_k keine Literale enthalten sein können, deren Wert sich zwischen den Zeitpunkten der Auswertung von L_i und K_k verändern kann. Werden die Rechte für die Nutzung von G_2 gemäß der Zugriffskontrollpolitik dynamisch vergeben, können die zur Durchsetzung dieser Festlegung erforderlichen Literale (z.B. `IN_ACL`–Prädikate oder Wertvergleiche über variablen DE–Inkarnationen) somit nicht in K_k formuliert werden, sondern sind vollständig in L_i zu spezifizieren. Neben dem Problem, daß die dafür benötigten Komponenten (zum Beispiel die zugriffskontrollierte Komponente, deren Zugriffskontrolliste auszuwerten ist) auch hier wiederum bereits in L_i sichtbar und bekannt sein müssen, ist in Kauf zu nehmen, daß entsprechende Rechteänderungen bzgl. G_2 im allgemeinen nicht unmittelbar, sondern erst bei dem nächsten Aufruf der *erzeuge*–Operation auf G_1 und der damit verbundenen Auswertung von L_i wirksam werden. Sollte eine derartige Verzögerung in der Wirksamkeit möglicher Rechteänderungen bzgl. G_2 gemäß der Zugriffskontrollpolitik nicht erlaubt sein, müssen die entsprechenden Literale in K_k spezifiziert werden. Damit ist das Recht des Systems jedoch nicht mehr absolut konsistent sondern lediglich potentiell konsistent.

Beispiel

Als Beispiel für die aufeinander abgestimmte Konstruktion zweier Zugriffsrestriktionsausdrücke gemäß Leitlinie L3. werden im folgenden die Operationen des Kontenverwaltungssystems betrachtet, die die Funktionalität des Auflösens eines Kontos implementieren. Gemäß der Zugriffskontrollpolitik des Kontenverwaltungssystems (vgl. Abschnitt 4.6.1) darf ein Konto nur von einem für die Betreuung des Kontos zuständigen Kundenbetreuer aufgelöst werden. Darüberhinaus besteht die Anforderung, daß Kundenbetreuer lediglich zu den üblichen Schalterzeiten, d.h. werktags zwischen 8.00 und 17.00 Uhr, Operationen des Kontenverwaltungssystems nutzen dürfen.

Für die Umsetzung der Rechtsfestlegung für das Auflösen eines Kontos wird von folgenden Voraussetzungen ausgegangen. Die Funktionalität des Auflösens eines Kontos wird durch

die äußere Operation `KontoAufloesen` des zentralen Kontoverwalters implementiert, die ihrerseits die Operation `Aufloesen` auf dem aufzulösenden Konto aufruft. Zwischen den entsprechenden DA–Generatorklassen, die diese Operationen definieren, existiert dementsprechend genau ein Aufrufpfad p mit $p = (\texttt{KontoAufloesen}, \texttt{Aufloesen})$. Die Operationen `KontoAufloesen` und `Aufloesen` sind potentiell beide unmittelbar von Benutzerrepräsentanten aus aufrufbar. Die Zugriffskontrolliste eines Kontos wird so initialisiert und verwaltet, daß in der Benutzer–Rollen–Liste der Operation `Aufloesen` für die Kundenbetreuer, die für die Betreuung des Kontos zuständig sind, ein entsprechender positiver Benutzer–Rollen–Eintrag vorhanden ist.

Ausgehend von diesen Voraussetzungen wird zunächst der Zugriffsrestriktionsausdruck für die Operation `KontoAufloesen` konstruiert. Gemäß der Festlegungen der Zugriffskontrollpolitik ergibt sich für $R_{\texttt{KontoAufloesen}}$:

$$\begin{aligned} R_{\texttt{KontoAufloesen}} = \;& \texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\ & \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{KontoZeiger}, \texttt{Aufloesen}) \wedge \\ & 8 <= \texttt{Zeit.Stunde} \;\wedge\; \texttt{Zeit.Stunde} < 17 \wedge \\ & 1 <= \texttt{Zeit.Wochentag} \;\wedge\; \texttt{Zeit.Wochentag} < 6 \end{aligned}$$

Dabei ist `KontoZeiger` ein Zeiger, der das aufzulösende Konto identifiziert und der als Eingabeparameter bei Aufruf der Operation `KontoAufloesen` zu übergeben ist. Die Rollenrestriktion (1. Literal) von $R_{\texttt{KontoAufloesen}}$ bewirkt, daß die Operation `KontoAufloesen` grundsätzlich nur von Benutzern, die in der Rolle des Kundenbetreuers agieren, ausgeführt werden kann. Mit dem `IN_ACL`–Prädikat (2. Literal) wird die Forderung umgesetzt, daß ein Konto nur von einem für die Betreuung des Kontos zuständigen Kundenbetreuer aufgelöst werden darf. Dieses `IN_ACL`–Prädikat hat genau dann den Wert *true*, wenn in der Zugriffskontrolliste des über den Eingabeparameter `KontoZeiger` identifizierten aufzulösenden Kontos in der Benutzer–Rollen–Liste der Operation `Aufloesen` ein entsprechender Eintrag für den aufrufenden Kundenbetreuer vorhanden ist. Mit den Wertvergleichen über Komponenten der globalen variablen DE–Inkarnation `Zeit` (3. bis 6. Literal) wird das Recht zur Ausführung der Operation `KontoAufloesen` auf die üblichen Schalterzeiten begrenzt.

Im nächsten Schritt ist nun der Zugriffsrestriktionsausdruck der im Kontext einer `KontoAufloesen`–Ausführung aufzurufenden Operation `Aufloesen` zu konstruieren. Würde diese Konstruktion unabhängig von dem bereits für die Operation `KontoAufloesen` festgelegten Zugriffsrestriktionsausdruck erfolgen, ergäbe sich vermutlich folgender Zugriffsrestriktionsausdruck:

$$\begin{aligned} R'_{\texttt{Aufloesen}} = \;& \texttt{Caller.Role} = \texttt{Kundenbetreuer} \wedge \\ & \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{THIS}, \texttt{THIS}) \wedge \\ & 8 <= \texttt{Zeit.Stunde} \;\wedge\; \texttt{Zeit.Stunde} < 17 \wedge \\ & 1 <= \texttt{Zeit.Wochentag} \;\wedge\; \texttt{Zeit.Wochentag} < 6 \end{aligned}$$

Bei einer derartigen Konstruktion von $R_{\texttt{Aufloesen}}$ wären die Zugriffsrestriktionsausdrücke $R_{\texttt{KontoAufloesen}}$ und $R_{\texttt{Aufloesen}}$ nicht konsistent bzgl. des Aufrufpfades p (vgl. Beispiel auf Seite 224). Um $R_{\texttt{Aufloesen}}$ konsistent zu $R_{\texttt{KontoAufloesen}}$ zu konstruieren, darf $R_{\texttt{Aufloesen}}$ weder die Wertvergleiche über Komponenten der globalen variablen DE–Inkarnation `Zeit` (gemäß Regel L3.2.) noch das `IN_ACL`–Prädikat (gemäß Regel L3.6.) enthalten. Da die mit diesen Literalen für die Operation `Aufloesen` durchzusetzenden Rechtsfestlegungen bereits durch entsprechende Literale in $R_{\texttt{KontoAufloesen}}$ implementiert werden, reicht es aus, in $R_{\texttt{Aufloesen}}$

eine Kontextrestriktion zu spezifizieren, die sicherstellt, daß die Operation `Aufloesen` lediglich im Kontext der Operation `KontoAufloesen` ausgeführt werden kann. Damit ergibt sich für $R_{\texttt{Aufloesen}}$:

$$R_{\texttt{Aufloesen}} \quad = \quad \texttt{Caller.ConGen} = \texttt{KontoAufloesen}$$

Mit dieser Konstruktion von $R_{\texttt{Aufloesen}}$ sind $R_{\texttt{KontoAufloesen}}$ und $R_{\texttt{Aufloesen}}$ konsistent bzgl. des Aufrufpfades p. Änderungen der in den Wertvergleichen verwendeten Komponenten der DE–Inkarnation `Zeit` sowie in dem Zugriffskontrollisteneintrag der Operation `Aufloesen`, die sich zwischen dem Beginn einer Ausführung der Operation `KontoAufloesen` und dem Zeitpunkt des darin stattfindenden Aufrufs der Operation `Aufloesen` ergeben, werden erst bei dem nächsten Aufruf von `KontoAufloesen` wirksam. Die Überprüfung, ob ein Kundenbetreuer das Recht zum Auflösen eines Kontos hat, erfolgt also zum Zeitpunkt des Aufrufs der Operation `KontoAufloesen`. Ist dieses Recht vorhanden, so wird damit auch das Recht impliziert, im Kontext der Ausführung der Operation `KontoAufloesen` die Operation `Aufloesen` auf dem aufzulösenden Konto auszuführen.

◇

Die Leitlinie L3. liefert ausgehend von der Sicht, daß Operationsausführungen von Benutzerrepräsentanten aus angestoßen werden, Regeln für die aufeinander abgestimmte Konstruktion der Zugriffsrestriktionsausdrücke funktional abhängiger Operationen, die unmittelbar bzw. mittelbar von Benutzerrepräsentanten aufrufbar sind. Neben derartigen Operationsausführungen gibt es in einem INSEL$^+$–System Operationsausführungen, die nicht von einem Benutzerrepräsentanten ausgehen, sondern die ursächlich von der Hauptkomponente des Systems initiiert werden. Die folgende Leitlinie, die sich aus Bedingung 2.(b) der Definition (5.28) ergibt, legt fest, wie die Zugriffsrestriktionsausdrücke äußerer Operationen zugriffskontrollierter Komponenten zu konstruieren sind, die von der Hauptkomponente aus aufrufbar sind.

L4. Die Disjunktive Normalform des Zugriffsrestriktionsausdrucks einer von der Hauptkomponente mittel– oder unmittelbar aufrufbaren äußeren Operation *op* einer zugriffskontrollierten Komponente muß für jeden Aufrufpfad p zwischen der Generatorklasse der Hauptkomponente und der Generatorklasse, die die Operation *op* definiert, einen konjunktiven Teilterm enthalten, der mindestens aus einem der folgenden Literale besteht:

- der Benutzerrestriktion `Caller.User = System`,
- der Rollenrestriktion `Caller.Role = SystemRole`,
- Kontextrestriktionen, die mit dem Aufrufpfad p erfüllbar sind.

Andere als diese Literale dürfen in dem konjunktiven Teilterm nicht enthalten sein.

Darf die äußere Operation einer zugriffskontrollierten Komponente gemäß der Zugriffskontrollpolitik nur von der Hauptkomponente ausgeführt werden, so kann dies im einfachsten Fall durch den Zugriffsrestriktionsausdruck `Caller.User = System` implementiert werden. Mit diesem Zugriffsrestriktionsausdruck wird das Recht zur Ausführung der Operation "pauschal" an die Hauptkomponente vergeben, d.h. es besteht unabhängig von dem Kontext, in dem der Operationsaufruf erfolgt. Soll die Hauptkomponente das Recht zur Ausführung der Operation nur innerhalb der Kontexte haben, in denen die Operation von der Hauptkomponente aufgerufen wird, so ist dies in dem Zugriffsrestriktionsausdruck durch entsprechende

Kontextrestriktionen zu spezifizieren. Für jeden der möglichen Kontexte, in denen die Operation von der Hauptkomponente ausgehend aufrufbar ist, ist in dem Zugriffsrestriktionsausdruck ein konjunktiver Teilterm anzugeben, der die entsprechenden Kontextrestriktionen enthält. Im allgemeinen wird die äußere Operation einer zugriffskontrollierten Komponente sowohl von einem Benutzerrepräsentanten als auch von der Hauptkomponente aus aufrufbar sein. Bei der Konstruktion des Zugriffsrestriktionsausdrucks einer derartigen Operation sind dann die Leitlinien L3. (einschließlich der Regeln L3.1. bis L3.7.) und L4. zu berücksichtigen.

Gemäß Leitlinie L4. dürfen in den für die Rechtevergabe an die Hauptkomponente relevanten konjunktiven Teiltermen eines Zugriffsrestriktionsausdrucks keine anderen als die dort angegebenen Literale enthalten sein. Insbesondere können dort keine Literale spezifiziert werden, die eine dynamische Rechtervergabe ermöglichen (z.B. `IN_ACL`–Prädikate). Daraus folgt, daß in INSEL^{+}–Systemen mit absolut konsistentem Recht an die Hauptkomponente lediglich statische Rechte zur Ausführung der äußeren Operationen zugriffskontrollierter Komponenten vergeben werden können.

Beispiel

Der Zugriffsrestriktionsausdruck der DA–Generatorklasse `KontoVerwalterTyp` aus dem Kontenverwaltungssystem, deren *erzeuge*–Operation unmittelbar von der Hauptkomponente aufgerufen wird, besteht aus dem Literal `Caller.User = System` und ist damit gemäß Leitlinie L4. konstruiert. Die Zugriffsrestriktionsausdrücke der DA–Generatorklassen `Zinsberechnung`, `Einzahlen` und `FuegeEin`, deren *erzeuge*–Operationen im Kontext der Ausführung der kanonischen Operation der Hauptkomponente mittelbar aufgerufen werden, enthalten als konjunktiven Teilterm jeweils eine entsprechende Kontextrestriktion und sind damit ebenfalls gemäß Leitlinie L4. konstruiert.

◇

Wie bereits zu Beginn dieses Abschnitts erwähnt, sind die angegebenen Leitlinien bei der Konstruktion eines INSEL^{+}–Systems, dessen Recht absolut konsistent sein soll, aufeinander abgestimmt einzusetzen. Je nach funktionalen Anforderungen an das System und der für dieses zu realisierenden Zugriffskontrollpolitik werden die einzelnen Leitlinien bei der Konstruktion eines Systems unterschiedlich stark anwendbar sein. In dem Konstruktionsprozeß sollten die Leitlinien immer in der Reihenfolge angewendet bzw. berücksichtigt werden, in der sie in diesem Abschnitt angegeben wurden.

Zunächst sollte also gemäß Leitlinie L1. geprüft werden, welche der Operationen, die unmittelbar von den Benutzerrepräsentanten nutzbar sein müssen, so konstruiert werden können, daß bei ihrer Ausführung keine weitere DA–Inkarnationen erzeugt werden und damit auch keine weiteren Operationen zugriffskontrollierter Komponenten aufgerufen werden. Für die Operationen, die nicht derart konstruierbar sind, ist dann gemäß Leitlinie L2. das Ziel zu verfolgen, die für die Erbringung der Funktionalität dieser Operationen benötigten Komponenten möglichst nicht zugriffskontrolliert zu konstruieren, sondern die für die Nutzung dieser Komponenten festgelegten Zugriffsbeschränkungen unter Einsatz des Schachtelungsprinzips und der Konzepte zur differenzierten Festlegung der Ausführungsumgebung durchzusetzen.

Anschließend sind die Zugriffsrestriktionsausdrücke der dann ggf. noch verbleibenden Paare funktional abhängiger Operationen zugriffskontrollierter Komponenten gemäß Leitlinie L3. aufeinander abgestimmt zu konstruieren. Hierbei sind die Konstruktionsregeln L3.1. bis L3.7. zu beachten. Im allgemeinen entscheidet sich bei diesem Konstruktionsschritt, ob das Recht des Systems tatsächlich absolut konsistent konstruiert werden kann. Läßt sich das System nicht mit absolut konsistentem Recht konstruieren, so werden die Ursachen hierfür jedoch erkennbar. Sie können sowohl in funktionalen Anforderungen des Systems als auch in der für

das System zu realisierenden Zugriffskontrollpolitik liegen. Es besteht dann die Möglichkeit, in einem Rückkopplungsprozeß die verursachenden "Sollanforderungen" kritisch zu betrachten und ggf. so zu modifizieren, daß das Recht des Systems absolut konsistent konstruierbar ist.

Im letzten Schritt sind schließlich gemäß Leitlinie L4. die Zugriffsrestriktionausdrücke der äußeren Operationen zugriffskontrollierter Komponenten zu konstruieren bzw. zu erweitern, die unmittelbar oder mittelbar von der Hauptkomponente aus aufrufbar sind.

5.5 Zusammenfassung

In diesem Kapitel wurde das Thema der Konsistenz der in einem INSEL$^+$–System vergebenene Rechte ausführlich behandelt. Der Literaturüberblick in Abschnitt 5.1 zeigte, daß die wenigen in der Literatur vorhandenen sich auf Rechte beziehenden Konsistenzbegriffe überwiegend lediglich objektlokal definiert sind und damit funktionale Abhängigkeiten zwischen Operationen, für deren Nutzung Rechte vergeben werden, nicht berücksichtigen. Derartige Konsistenzbegriffe sind nicht geeignet, Ausführungsabbrüche begonnener Operationsausführungen aufgrund fehlender Rechte zu vermeiden und damit die Voraussetzung für die mit der Wahrnehmung eines Rechts verbundene Leistungsanforderung zu schaffen.

Ausgehend von dieser Erkenntnis wurde in Abschnitt 5.2 zunächst der Begriff des Rechts eines INSEL$^+$–Systems formal präzisiert und anschließend definiert, wann das Recht eines INSEL$^+$–Systems konsistent ist. Der für INSEL$^+$–Systeme eingeführte Konsistenzbegriff ist sehr weitreichend und berücksichtigt insbesondere funktionale Abhängigkeiten zwischen Operationen. Dieser Konsistenzbegriff fordert, daß das Recht zur Ausführung einer Operation *op* auch das Recht zur Ausführung aller Operationen implizieren muß, von denen *op* funktional abhängig ist, die also im Kontext einer *op*–Ausführung auszuführen sind. In einem INSEL$^+$–System mit konsistentem Recht können somit keine Ausführungsabbrüche begonnener Operationsausführungen aufgrund fehlender Rechte auftreten. Damit ist eine wesentliche Voraussetzung für die mit der Wahrnehmung eines Rechts verbundene Leistungsanforderung erfüllt.

Das Recht eines INSEL$^+$–Systems ergibt sich im wesentlichen aus den in dem entsprechenden INSEL$^+$–Programm mit den in Kapitel 4 eingeführten Sprachkonzepten festgelegten Zugriffsbeschränkungen. Damit besteht die Möglichkeit, durch geeignete statische Analysen bereits zur Übersetzungszeit eines INSEL$^+$–Programms Aussagen über die Konsistenz des Rechts des durch dieses Programm definierten INSEL$^+$–Systems zu gewinnen. Die Konsistenz des Rechts eines INSEL$^+$–System kann im allgemeinen jedoch nicht vollständig statisch nachgewiesen werden, da mit den Sprachkonzepten von INSEL$^+$ auch Möglichkeiten zur Vergabe dynamischer Rechte zur Verfügung stehen. Orientiert an den Ergebnissen, die im Rahmen statischer Analysen hinsichtlich einer Konsistenzaussage erzielbar sind, wurden in Abschnitt 5.2 Differenzierungen des Konsistenzbegriffs vorgenommen. Demnach kann das Recht eines INSEL$^+$–Systems absolut konsistent, potentiell konsistent oder absolut inkonsistent sein. Das Recht eines INSEL$^+$–Systems ist absolut konsistent, wenn dessen Konsistenz vollständig statisch nachgewiesen werden kann. Es ist potentiell konsistent, wenn es keine statisch feststellbaren Inkonsistenzen enthält, und absolut inkonsistent, wenn mindestens eine Inkonsistenz statisch nachweisbar ist. An jedes INSEL$^+$–System wird die Anforderung gestellt, daß das Recht des Systems zumindest potentiell konsistent ist. Das Ziel eines Systementwicklers sollte jedoch darin bestehen, durch geeignete Konstruktion möglichst ein System mit absolut konsistentem Recht zu entwickeln.

Die statischen Konsistenzanalysen, die ausgehend von einem INSEL$^+$–Programm eine Aussage darüber liefern, ob das Recht des entsprechenden INSEL$^+$–Systems absolut konsistent, potentiell konsistent oder absolut inkonsistent ist, wurden in Abschnitt 5.3 erklärt. Sie basieren auf dem in Abschnitt 5.3.1 entwickelten attributierten statischen Aufrufgraph des INSEL$^+$–Programms, der potentielle Aufrufbeziehungen und damit funktionale Abhängigkeiten vergröbert auf die in dem Programm definierten Generatoren und Inkarnationen beschreibt. Dieser Aufrufgraph kann nicht nur für Konsistenzanalysen genutzt werden, sondern stellt auch eine geeignete Basis für die Durchführung anderer Analysen – insbesondere hinsichtlich der Gewinnung von Informationen für Realisierungsentscheidungen für Komponenten, wie z.B. die gemeinsame Plazierung von Komponenten aufgrund funktionaler Abhängigkeiten – dar.

Mit dem in Abschnitt 5.3.3 angegebenen Analyseverfahren für die absolute Konsistenz ist es erstmals für den auf funktionale Abhängigkeiten zwischen Operationen bezogenen Konsistenzbegriff möglich, die Konsistenz des Rechts eines Systems vollständig automatisiert statisch nachzuweisen. In Abschnitt 5.4 wurden aus den Kriterien, die im Rahmen dieses Analyseverfahrens überprüft werden, konstruktive Regeln abgeleitet, an denen sich ein Systementwickler orientieren kann, wenn er ein INSEL$^+$–System mit absolut konsistentem Recht entwickeln möchte. Diese Regeln beziehen sich im wesentlichen auf die aufeinander abgestimmte Konstruktion der Zugriffsrestriktionausdrücke funktional abhängiger Operationen. Dabei werden insbesondere die Möglichkeiten zur Vergabe dynamischer Rechte für die Nutzung von Operationen, die im Kontext anderer Operationen aufgerufen werden, eingeschränkt. Inwieweit also ein INSEL$^+$–System so konstruierbar ist, daß das Recht des Systems absolut konsistent ist, hängt im allgemeinen von den funktionalen Anforderungen an das System und der für dieses mit den INSEL$^+$–Konzepten zu realisierenden Zugriffskontrollpolitik ab. Durch Orientierung an den Leitlinien werden jedoch die Ursachen dafür erkennbar, warum sich ein System ggf. nicht mit absolut konsistentem Recht konstruieren laßt. Im Extremfall liegt in der Spezifikation der zu realisierenden Zugriffskontrollpolitik eine Inkonsistenz vor, die durch geeignete Modifikation der Politik zu beheben ist. In anderen Fällen besteht eventuell die Möglichkeit, durch Präzisierungen bzw. geringfügige Änderungen der Zugriffskontrollpolitik doch noch zu erreichen, daß das System mit absolut konsistentem Recht konstruierbar ist.

Die in diesem Kapitel eingeführten Begriffe sowie die erarbeiteten Analyseverfahren und konstruktiven Leitlinien wurden bisher immer auf ein Gesamtsystem bezogen. Es ist jedoch ohne weiteres möglich, diese auch auf Subsysteme eines INSEL$^+$–Systems zu übertragen. Dies bietet dann zum Beispiel die Möglichkeit, ein komplexes INSEL$^+$–System, das nicht komplett als System mit absolut konsistentem Recht konstruierbar ist, so zu konstruieren, daß es aus einer Menge von Subsystemen mit jeweils absolut konsistentem Recht besteht. In einem derartigen System kann dann lediglich bei einem subsystemübergreifenden Zugriff eine Inkonsistenz auftreten.

Mit den in Kapitel 4 eingeführten Sprachkonzepten sowie den in diesem Kapitel angegebenen konstruktiven Leitlinien und statischen Konsistenzanalysen steht ein Instrumentarium zur Verfügung, das die systematische und zielgerichtete Implementierung von Systemen mit konsistent vergebenen Rechten unterstützt. Die weitere Aufgabe besteht nun darin, die konstruierten INSEL$^+$–Systeme auf einer verteilten Hardware–Konfiguration so zur Ausführung zu bringen, daß die mit den Konzepten festgelegten Eigenschaften, zu denen insbesondere die spezifizierten Zugriffsbeschränkungen gehören, realisiert werden. In dem folgenden Kapitel wird ein entsprechender Ansatz für die Realisierung von INSEL$^+$–Systemen vorgestellt.

Kapitel 6

Realisierung von INSEL$^+$–Systemen

In diesem Kapitel wird auf die Realisierung von INSEL$^+$–Systemen auf einer verteilten Hardware–Konfiguration eingegangen. Dabei wird ein sprachbasierter *top–down* Ansatz verfolgt, in dem die für die Realisierung von INSEL$^+$–Systemen benötigten Konzepte und Verfahren ausgehend von der Sprachebene schrittweise über mehrere Realisierungsebenen hinweg angepaßt an die Sprachkonzepte entwickelt und konkretisiert werden.

In dieser Arbeit werden schwerpunktmäßig Konzepte und Maßnahmen zur Realisierung der für die zugriffskontrollierten Komponenten eines INSEL$^+$–Systems festgelegten Zugriffsbeschränkungen angegeben, d.h. Konzepte und Maßnahmen zur Durchsetzung des Rechts eines INSEL$^+$–Systems. Die sonstigen für die Realisierung von INSEL$^+$–Systemen zu lösenden Aufgaben, insbesondere im Bereich der Ressourcenverwaltung, unterscheiden sich nicht von denen, die im Rahmen der Realisierung von INSEL–Systemen zu lösen sind. Konzepte und Verfahren für eine effiziente Realisierung von INSEL–Systemen und die dabei zu lösenden Ressourcenverwaltungsaufgaben wurden bereits in anderen Arbeiten entwickelt (siehe u.a. [EP98], [Gro98], [Win96], [Rad96]). In [EP98] wird die Architektur eines neuen verteilten Betriebssystems als Ausführungsbasis für INSEL–Systeme beschrieben. Diese neuartige Betriebssystem–Infrastruktur ist durch eine strukturierte, sich dynamisch ändernde Menge von Verwaltern (Managern) charakterisiert, die die erforderlichen Realisierungs– und Managementaufgaben kooperativ wahrnehmen. Im Gegensatz zu anderen sprachbasierten Ansätzen (wie zum Beispiel Oberon [WG89, MW95] oder Guide [BBD+91, BLR94, Hag94]) werden dabei alle am Ressourcenmanagement beteiligten Komponenten systematisch in ein Gesamtsystem integriert. In den Arbeiten [Gro98], [Win96], [Rad96] wurden Verfahren für die effiziente und verteilte Ressourcenverwaltung erarbeitet und prototypisch in INSEL–Laufzeitsystemen unter Nutzung existierender Betriebssystem-Infrastrukturen realisiert. Die Schwerpunkte in [Gro98] und [Win96] lagen auf der Entwicklung von Konzepten und Mechanismen für die Speicherverwaltung und für die effiziente Realisierung von Zugriffen auf Komponenten. In [Rad96] wurden hingegen vorrangig Verfahren für die Lastverwaltung erarbeitet.

Die Realisierungskonzepte und –maßnahmen zur Durchsetzung des Rechts eines INSEL$^+$–Systems können nicht unabhängig von den sonstigen Verfahren für die Realisierung des Systems gesehen werden. Dem entsprechend haben die in den bisherigen Arbeiten zur Realisierung von INSEL–Systemen erarbeiteten Konzepte und Verfahren Konsequenzen für die Entwicklung der Maßnahmen zur Durchsetzung des Rechts eines INSEL$^+$–Systems. Sie werden hier deshalb, soweit sie für das grundlegende Verständnis der Realisierung von INSEL$^+$–Systemen und die Einordnung der Maßnahmen zur Rechtsdurchsetzung notwendig sind, erläutert.

Die Realisierung von INSEL$^+$–Systemen wird in dieser Arbeit **konzeptionell** beschrieben. Auf die konkrete Umsetzung, d.h. die Implementierung der angegebenen Realisierungskonzepte und –maßnahmen im Rahmen einer Ausführungsumgebung für INSEL$^+$–Systeme wird in dieser Arbeit nicht eingegangen. Hierzu sei auf die bereits erwähnten Arbeiten verwiesen. Die zur Durchsetzung des Rechts eines INSEL$^+$–Systems erarbeiteten Realisierungskonzepte und –verfahren lassen sich sowohl in die in [Rad96] bzw. [Win96] entwickelten Laufzeitsysteme zur Ausführung von INSEL–Programmen auf einem Cluster von HP-UX Workstations bzw. auf vernetzten Mach3.0–Rechnern als auch in die in [EP98, EPR97] beschriebene INSEL–angepaßte Betriebssystem–Infrastruktur integrieren.

Dieses Kapitel ist folgendermaßen gegliedert. In dem einleitenden Abschnitt 6.1 werden die für das Verständnis der Realisierung von INSEL$^+$–Systemen benötigten Grundlagen vermittelt. Es wird zunächst eine Übersicht über die gestellte Realisierungsaufgabe sowie den zur Lösung dieser Aufgabe verfolgten Realisierungsansatz gegeben. Als Basis für die Realisierung von INSEL$^+$–Systemen wird eine Realisierungsarchitektur vorgestellt, die aus den abstrakten, d.h. mit den Sprachkonzepten festgelegten Eigenschaften eines INSEL$^+$–Systems abgeleitet ist. Diese Realisierungsarchitektur besteht aus einer strukturierten, sich dynamisch ändernden Menge von Verwaltern und wird im weiteren als **Verwalterarchitektur** bezeichnet. Die Verwalterarchitektur ist das Bindeglied zwischen den zu realisierenden Komponenten eines INSEL$^+$–Systems und postulierten Basisdiensten, die von dem jeweils zugrundeliegenden Betriebssystemkern bereitzustellen sind. Die Verwalter stellen somit die Dienste zur Verfügung, die zur Realisierung von INSEL$^+$–Systemen unter Nutzung der postulierten Basisdienste benötigt werden. Gemäß dem verfolgten *top–down* Ansatz sind die Eigenschaften der Verwalter und die von ihnen bereitgestellten Dienste schrittweise zu verfeinern und zu konkretisieren. In dieser Arbeit erfolgt diese Konkretisierung über zwei Realisierungsebenen hinweg. Die erste Realisierungsebene ist charakterisiert durch eine Menge abstrakter Prozessoren und einen abstrakten Speicher, der von allen Prozessoren gemeinsam nutzbar ist. Die zweite Realisierungsebene besteht aus einer Menge untereinander vernetzter Stellenrechner, die jeweils über lokale Prozessoren und lokalen Speicher verfügen. Ein physikalisch gemeinsamer Speicher steht nicht zur Verfügung, d.h. die Prozessoren einer Stelle können lediglich auf den stellenlokalen Speicher zugreifen. Diese Realisierungsebene ist damit der realen Hardware–Konfiguration, auf der INSEL$^+$–Systeme letztlich zu realisieren sind, sehr nahe. Sie abstrahiert lediglich von spezifischen im wesentlichen hardware–orientierten Eigenschaften der Zielebene, wie z.B. der Speichergröße oder dem Netzzugangsprotokoll. In Abschnitt 6.2 werden zunächst die Eigenschaften und Aufgaben der Verwalter für die Realisierungsebene 1 konkretisiert. Diese werden dann in Abschnitt 6.3 den Gegebenheiten der Realisierungsebene 2 entsprechend angepaßt und erweitert.

6.1 Grundlagen

In diesem Abschnitt werden die Grundlagen der Realisierung von INSEL$^+$–Systemen erklärt. Dazu wird zunächst in Abschnitt 6.1.1 angegeben, welche zusätzliche Aufgabe bei der Realisierung von INSEL$^+$–Systemen im Vergleich zur Realisierung von INSEL–Systemen zu lösen ist. Anschließend wird in Abschnitt 6.1.2 der zur Realisierung von INSEL$^+$–Systemen verfolgte *top–down* Ansatz näher erläutert. In Abschnitt 6.1.3 wird schließlich die aus den abstrakten Eigenschaften eines INSEL$^+$–Systems abgeleitete Verwalterarchitektur, die Basis für die Realisierung des Systems ist, angegeben.

6.1.1 Realisierungsaufgabe

Die wesentliche Aufgabe, die im Rahmen der Realisierung von INSEL^{+}–Systemen im Vergleich zur Realisierung von INSEL–Systemen zusätzlich zu lösen ist, besteht darin, das Recht des INSEL^{+}–Systems durchzusetzen, d.h. die für die zugriffskontrollierten Komponenten des Systems festgelegten Zugriffsbeschränkungen zu realisieren. Dazu sind bei Zugriffen von Akteuren auf zugriffskontrollierte Komponenten Zugriffskontrollen durchzuführen. Diese Zugriffskontrollen bestehen darin, bei Aufruf einer äußeren Operation einer zugriffskontrollierten Komponente den für diese Operation festgelegten Zugriffsrestriktionsausdruck auszuwerten. Ist der Wert des Zugriffsrestriktionsausdrucks *false*, so hat der aufrufende Akteur nicht das Recht zur Ausführung der Operation, d.h. der Zugriff ist nicht erlaubt. Anderenfalls ist der Zugriff erlaubt und die Operation kann ausgeführt werden. Für die Durchsetzung des Rechts eines INSEL^{+}–Systems ist also zum einen zu gewährleisten, daß die entsprechenden Zugriffskontrollen durchgeführt werden und nicht umgangen werden können. Eine Umgehung der Zugriffskontrollen wäre zum Beispiel dann möglich, wenn Benutzer einen direkten Zugriff auf den Speicher und damit auch auf die in dem Speicher repräsentierten Komponenten erhalten könnten. Zum anderen müssen die für die Zugriffskontrollen, d.h. die für die Auswertung der Zugriffsrestriktionsausdrücke benötigten Informationen so verwaltet werden, daß sie nicht unautorisiert manipuliert werden können. Für die korrekte Auswertung von Zugriffsrestriktionsausdrücken müssen insbesondere Datenstrukturen angelegt und sicher verwaltet werden, auf deren Basis die speziellen in INSEL^{+} zur Festlegung von Zugriffsrestriktionsausdrücken zur Verfügung stehenden Konzepte realisiert werden können. Dazu gehören:

- Kontextattribut für Akteure
  Für jeden Akteur sind die Informationen zu verwalten, die dem in Definition (4.11) definierten Kontextattribut für Akteure entsprechen. Dazu gehören der Identifikator des Benutzers, in dessen Auftrag der Akteur die kanonische Operation seiner aktuellen Ausführungskomponente ausführt, der Identifikator der Rolle, in der dieser Benutzer agiert, und eindeutige Identifikatoren für den Akteur und seinen Generator sowie seine Ausführungskomponente und den Generator der Ausführungskomponente[1].

- Owner–Attribut für Komponenten
  Jeder Komponente eines INSEL^{+}–Systems ist über das in Definition (4.2) festgelegte Attribut *owner* ein Benutzer als Besitzer fest zugeordnet. Zur Realisierung dieses Attributs ist für jede Komponente der entsprechende Benutzeridentifikator zu verwalten.

- Zugriffskontrollisten
  Gemäß Abschnitt 4.4.3 ist für jede zugriffskontrollierte Komponente implizit eine Zugriffskontrolliste definiert, die geeignet zu realisieren und zu verwalten ist. Für die Auswertung eines `IN_ACL`–Prädikats (siehe Abschnitt 4.4.4.1) ist die Zugriffskontrolliste der Komponente, die in dem Prädikat mit dem Bezeichner <*comp-identifier*> identifiziert wird, auszuwerten. Im Zusammenhang mit der Realisierung von Zugriffskontrollisten sind auch die implizit auf jedem explizit zugriffskontrollierten DA–Generator, jedem zugriffskontrollierten Depot und jedem zugriffskontrollierten K-Akteur definierten äußeren Operationen *change_acl* und *list_acl* zu implementieren.

- Zugriffshistorienlisten
  Um das in Abschnitt 4.4.4.2 definierte Zugriffsrestriktionsprädikat `ACCESSED` zu rea-

[1] Für jede Komponente wird bei ihrer Erarbeitung bzw. Erzeugung ein systemweit eindeutiger Identifikator festgelegt, über den die Komponente in dem realisierten System eindeutig identifizierbar ist (siehe hierzu Abschnitt 6.2.2.1).

lisieren, muß festgehalten werden, welcher Benutzer bereits auf welche Komponenten mit welchen Operationen zugegriffen hat. Für die Repräsentation und Verwaltung dieser Informationen sind mehrere Realisierungsalternativen denkbar, die sich zum einen darin unterscheiden, ob die Informationen benutzerbezogen oder komponentenbezogen verwaltet werden und zum anderen danach differenziert werden können, ob die Informationen zentral oder dezentral erfaßt werden. In dieser Arbeit wird eine dezentrale komponentenbezogene Lösung verfolgt, die dadurch charakterisiert ist, daß nur die Informationen verwaltet werden, die auch tatsächlich für die Auswertung der `ACCESSED`–Prädikate eines Systems benötigt werden. Dazu wird für jede Komponente, die potentiell in einem `ACCESSED`–Prädikat identifiziert wird, eine sogenannte Zugriffshistorienliste verwaltet, in der festgehalten wird, welche Benutzer bereits welche Operationen auf der Komponente ausgeführt haben. Die Komponenten eines Systems, die potentiell in den `ACCESSED`–Prädikaten identifiziert werden können und für die somit eine Zugriffshistorienliste zu führen ist, werden von dem INSEL$^+$–Übersetzer im Rahmen der statischen Anlayse ermittelt.

Bevor im weiteren näher auf den verfolgten Realisierungsansatz eingegangen wird, werden zunächst die Maßnahmen für die Durchführung der Zugriffskontrollen in die allgemeinen Maßnahmen zur Realisierung von INSEL$^+$–Systemen eingeordnet.

Zugriffskontrollen sind bei Zugriffen von Akteuren auf zugriffskontrollierte Komponenten durchzuführen. Die zugriffskontrollierten Komponenten eines Systems lassen sich laut Abschnitt 4.4.1 in zugriffskontrollierte DA–Generatoren und zugriffskontrollierte DA–Inkarnationen unterteilen. Von den zugriffskontrollierten DA–Inkarnationen sind die Depots und die K-Akteure die interessierenden, da lediglich für diese äußere Operationen explizit definiert werden können, für die dann entsprechende Zugriffsrestriktionsausdrücke festgelegt werden können. Ein Zugriff auf einen zugriffskontrollierten DA–Generator entspricht dem Aufruf der Operation *erzeuge* auf dem Generator. Ein Zugriff auf ein zugriffskontrolliertes Depot bzw. einen zugriffskontrollierten K-Akteur wird durch Aufruf einer Zugriffsoperation des Depots bzw. einer Kommunikationsoperation des K-Akteurs initiiert. Als Zugriffsoperationen eines Depots sind den Festlegungen aus Abschnitt 4.4.1 entsprechend lediglich *erzeuge*–Operationen lokaler DA–Generatoren erlaubt. Die Kommunikationsoperationen eines K-Akteurs entsprechen den *erzeuge*–Operationen seiner K-Order–Generatoren. Ein Zugriff auf eine zugriffskontrollierte Komponente wird also stets durch den Aufruf der Operation *erzeuge* auf einem DA–Generator initiiert. Dies bedeutet, daß ein Zugriff auf eine zugriffskontrollierte Komponente mit der Erzeugung einer DA–Inkarnation verbunden ist, wobei die Gültigkeit des für die *erzeuge*–Operation des Generators festgelegten Zugriffsrestriktionsausdrucks Vorbedingung für die Erzeugung der Inkarnation ist. Die Auswertung des Zugriffsrestriktionsausdrucks, d.h. die Zugriffskontrolle ist also vor Erzeugung der Inkarnation durchzuführen. Als Fazit dieser Überlegungen ergibt sich, daß die Maßnahmen zur Durchführung der Zugriffskontrollen in die vorbereitenden Maßnahmen zur Realisierung der Erzeugung von DA–Inkarnationen zu integrieren sind.

Nach dieser Einordnung der Zugriffskontrollen in die insgesamt durchzuführenden Realisierungsmaßnahmen wird im folgenden der Ansatz, nach dem INSEL$^+$–Systeme realisiert werden, erläutert.

6.1.2 Realisierungsansatz

Zur Realisierung von INSEL$^+$–Systemen wird analog zum Realisierungsansatz für INSEL–Systeme ein *top–down* Ansatz verfolgt. Dieser Ansatz ist dadurch charakterisiert, daß die

Realisierung eines Systems ausgehend von der Sprachebene über mehrere Zwischenstufen, die in dieser Arbeit **Realisierungsebenen** genannt werden, nach dem **Prinzip der schrittweisen Konkretisierung** erfolgt. Die verschiedenen Realisierungsebenen ergeben sich dadurch, daß schrittweise Eigenschaften der Hardware–Konfiguration, auf der das System letztlich zu realisieren ist, hinzugenommen werden. Das Prinzip der schrittweisen Konkretisierung besagt, daß beim Übergang von einer Realisierungsebene zur nächsten die Realisierungskonzepte der Ausgangsebene so zu verfeinern und zu ergänzen sind, daß die mit den Sprachkonzepten festgelegten Eigenschaften des Systems unter Berücksichtigung der zusätzlichen Realisierungsrandbedingungen der Zielebene erhalten bleiben. Diese Vorgehensweise zur Realisierung von INSEL$^+$–Systemen orientiert sich damit an dem in [Spi90] eingeführten Grundsatz der **konzeptionellen Konkretisierung** für System–Konstruktionen nach einem *top–down*–Ansatz.

Die Vorteile des verfolgten *top–down* Ansatzes liegen im Vergleich zu den herkömmlichen *bottom–up* Ansätzen im wesentlichen darin, daß die Realisierung nicht vorrangig durch die Eigenschaften der zugrundeliegenden Hardware geleitet wird, sondern von den mit den Sprachkonzepten festgelegten Eigenschaften eines Systems. Dadurch können insbesondere mögliche Alternativen für Realisierungen vor allem im Hinblick auf Effizienzsteigerungen systematisch abgeleitet und durch Bereitstellung eines entsprechenden Spektrums von Realisierungskonzepten und –mechanismen umgesetzt werden. Der verfolgte Ansatz ermöglicht es damit, die Realisierung den jeweiligen Eigenschaften des Systems und seiner Komponenten entsprechend anzupassen. In [Win96] wurden zum Beispiel Alternativen für die Realisierung von passiven INSEL–Inkarnationen und für die Realisierung von Zugriffen auf diese nach dem *top–down* Ansatz abgeleitet und ein entsprechendes Realisierungsinstrumentarium entwickelt. Anhand von Performance–Messungen wurde gezeigt, daß sich durch den systematischen und flexiblen Einsatz dieser Realisierungsalternativen Effizienzsteigerungen erreichen lassen.

In dieser Arbeit werden ausgehend von der Sprachebene zwei Realisierungsebenen betrachtet, die vorrangig unter dem Gesichtspunkt ausgewählt wurden, schrittweise jeweils die Realisierungsrandbedingungen zu erfassen, die wesentliche Konsequenzen bezüglich der zur Durchsetzung des Rechts eines INSEL$^+$–Systems einzusetzenden Sicherheitsmaßnahmen haben. Die Hardware–Konfiguration, auf der INSEL$^+$–Systeme realisiert werden sollen, ist durch eine Menge vernetzter Stellenrechner (kurz: Stellen) charakterisiert. In dem gesamten *top–down* orientierten Realisierungsvorgang ergibt sich ein wesentlicher Konkretisierungsschritt hinsichtlich der zur Durchsetzung des Rechts eines Systems erforderlichen Maßnahmen beim Übergang von einer Realisierungsebene, die von Stellen abstrahiert, zu einer Ebene, in denen die Stellen erfaßt werden. Dementsprechend werden in dieser Arbeit die beiden folgenden Realisierungsebenen betrachtet, deren genauere Charakterisierung in den Abschnitten 6.2 und 6.3 erfolgt.

- Realisierungsebene 1
  Die Realisierungsebene 1 ist gekennzeichnet durch eine Menge abstrakter Prozessoren und einen linear adressierbaren abstrakten Speicher, der von allen Prozessoren gemeinsam nutzbar ist.

- Realisierungsebene 2
  Die Realisierungsebene 2 ist charakterisiert durch eine Menge von Stellen, die durch Nachrichtentransportkanäle untereinander verbunden sind und die jeweils aus einem bzw. mehreren Prozessoren, lokalem Speicher sowie einem lokalen Betriebssystemkern bestehen. Die Prozessoren einer Stelle können direkt lediglich auf den stellenlokalen Speicher zugreifen. Über die Transportkanäle können Nachrichten zwischen den Stellen ausgetauscht werden.

Die wesentlichen Unterschiede zwischen den auf Realisierungsebene 1 bzw. Realisierungsebene 2 zur Durchsetzung des Rechts eines INSEL$^+$–Systems erforderlichen Sicherheitsmaßnahmen ergeben sich insbesondere aus den Eigenschaften der Kanäle, über die die Stellen der Realisierungsebene 2 untereinander vernetzt sind. Die beim Übergang von Realisierungsebene 1 auf Realisierungsebene 2 zu lösenden Probleme zur Durchsetzung des Rechts entsprechen den Problemen, die beim Übergang von einer zentralen stellenbezogenen Realisierung des Systems zu einer verteilten Realisierung auf einer Menge vernetzter Stellen zu lösen sind. Dabei ergeben sich insbesondere Sicherheitsprobleme daraus, daß die Kanäle zwischen den Stellen im allgemeinen unsichere Kanäle in dem Sinne sind, daß sie nicht gegen aktive und passive Angriffe geschützt sind. Geeignete Maßnahmen zum Schutz der sich damit ergebenden Gefährdungen bestehen, wie bereits in Kapitel 2 erläutert, in der wechselseitigen Authentifizierung stellenübergreifend kooperierender Komponenten sowie der Verschlüsselung der zwischen diesen auszutauschenden Nachrichten. Entsprechende Maßnahmen sind also auf Realisierungsebene 2 mit dem Ziel einzusetzen, die Eigenschaften der abstrakten Kanäle zwischen kooperierenden Komponenten der Realisierungsebene 1 mit den unsicheren physikalischen Kanälen, über die die Stellen der Realisierungsebene 2 verbunden sind, zu realisieren.

Gemäß des verfolgten *top–down* Ansatzes besteht die weitere Aufgabe nun darin, zunächst die Realisierungskonzepte und –maßnahmen zur Realisierung von INSEL$^+$–Systemen auf der Realisierungsebene 1 zu erarbeiten. Anschließend sind diese Konzepte und Maßnahmen für die Realisierungsebene 2 anzupassen und um Maßnahmen zur Lösung der sich aus den Eigenschaften dieser Ebene zusätzlich ergebenden Probleme zu ergänzen. Der methodische Vorteil der schrittweisen Konkretisierung über diese beiden Realisierungsebenen besteht darin, daß auf Realisierungsebene 1 zuerst die generellen Maßnahmen zur Realisierung von Zugriffskontrollen eingeführt werden können und anschließend beim Übergang zur Realisierungsebene 2 die mit den unsicheren Kommunikationskanälen hinzukommenden Sicherheitsprobleme gelöst werden können.

Ausgangspunkt für die Entwicklung der Realisierungskonzepte und –maßnahmen der Realisierungsebene 1 ist die bereits erwähnte Verwalterarchitektur. Diese Verwalterarchitektur wird nach dem *top–down* Ansatz aus den Eigenschaften eines INSEL$^+$–Systems der Sprachebene abgeleitet und schrittweise den Gegebenheiten der beiden Realisierungsebenen entsprechend verfeinert und konkretisiert. Die Verwalterarchitektur und ihre Ableitung aus den abstrakten Eigenschaften des Systems wird im folgenden Abschnitt grob erklärt. Eine detailliertere Beschreibung ist zum Beispiel in [Win96] zu finden.

6.1.3 Verwalterarchitektur

Ausgangsbasis für die Ableitung der Verwalterarchitektur ist ein abstraktes INSEL$^+$–System mit seinen Eigenschaften, die sich aus den für die Konstruktion des Systems eingesetzten Konzepten ergeben. Wie in Abschnitt 3.3 erklärt, kann ein INSEL$^+$–System $\mathcal{S}$ der Sprachebene zu einem Zeitpunkt t durch eine Systemkonfiguration $\mathcal{S}_t = (X_t^{DAI}, \delta_t, \alpha_t, \lambda_t, \zeta_t, \varepsilon_t)$ beschrieben werden (vgl. Definition (3.11)). Die Dynamik des Systems $\mathcal{S}$, die sich insbesondere aus der Erzeugung und Auflösung von Komponenten ergibt, läßt sich durch eine Familie solcher Konfigurationen beschreiben. Die mit den Systemkonfigurationen gegebene Beschreibungsform eines INSEL$^+$–Systems ist sehr detailliert. Für die im Zusammenhang mit der Realisierung von INSEL$^+$–Systemen zu lösenden Ressourcenverwaltungsaufgaben ist es sinnvoll, von dieser detaillierten Sicht zu vergröbern, d.h. Komponenten so zu Mengen zusammenzufassen, daß diese Mengen als Einheiten für die Ressourcenverwaltung dienen können. Die Systemkonfigurationen sind also unter dem Gesichtspunkt der Ressourcenverwaltung geeignet zu vergröbern. Aus der Sicht der Ressourcenverwaltung sind die wesentlichen Ereignisse

die Erzeugung und Auflösung von Komponenten, da die Erzeugung bzw. Auflösung einer Komponente mit der Belegung bzw. Freigabe von Ressourcen verbunden ist. Die Komponenten eines Systems, die die Berechnungen des Systems ausführen und dazu Komponenten erzeugen und (implizit) wieder auflösen, sind die Akteure. Es liegt daher nahe, eine Systemkonfiguration so zu vergröbern, daß jedem Akteur die von ihm lebenszeitmäßig abhängigen passiven Komponenten zugeordnet werden und die Relation ε, die die Lebenszeitabhängigkeiten beschreibt, auf die sich damit ergebenden Mengen zu vergröbern. Die Mengen von Komponenten, die aus jeweils einem Akteur zusammen mit den von diesem lebenszeitmäßig abhängigen passiven Komponenten bestehen, werden als **Akteursphären** bezeichnet. Die Menge der Akteursphären eines Systems bildet eine Zerlegung der Menge der Komponenten des Systems, da jede passive Komponente nach dem angegebenen Kriterium eindeutig einem Akteur zugeordnet werden kann. Durch die Vergröberung der Lebenszeit–Relation ε auf die Mengen der Akteursphären ergibt sich eine Baumstruktur über der Menge der Akteursphären. Die Vergröberung der Konfiguration eines Systems zu einer baumstrukturierten Menge von Akteursphären entspricht der in [SEL$^+$96] erklärten Vergröberung eines Systems zur Lebenszeit–orientierten Akteur–Struktur, auf deren präzise Definition an dieser Stelle jedoch verzichtet wird.

Beispiel

In Abbildung 6.1 ist die Vergröberung einer Systemkonfiguration zu einer Menge von Akteursphären anhand eines „Schnappschusses“ des INSEL$^+$–Systems, das die Kontenverwaltung einer Bank realisiert (vgl. Abschnitt 4.6.1 und Anhang B), verdeutlicht. In der oberen Hälfte der Abbildung ist eine Systemkonfiguration des Kontenverwaltungssystems mit den zu dem betrachteten Zeitpunkt existierenden DA–Inkarnationen und den zwischen diesen bestehenden Lebenszeitabhängigkeiten dargestellt. Die Bildung der Akteursphären durch die jeweilige Zusammenfassung eines Akteurs mit den von ihm lebenszeitmäßig abhängigen passiven Inkarnationen ist durch die grau unterlegten Bereiche verdeutlicht. In der unteren Hälfte ist der sich durch die Vergröberung ergebende Baum der Akteursphären dargestellt, wobei von der Zusammensetzung der Sphären aus einzelnen Komponenten abstrahiert wird. Die Sphäre eines Akteurs a wird dabei mit $AS(a)$ bezeichnet. Die Bezeichnungen für die einzelnen Akteure werden aus Platzgründen sinnvoll abgekürzt, wie zum Beispiel `L1` für den Akteur `LoginAtTerminal1`.

◇

Die Vergröberung eines Systems zu einem Baum von Akteursphären liefert die Basis für die Ableitung der Verwalterarchitektur. Die Akteursphären sind die Verwaltungseinheiten. Jeder Akteursphäre wird ein – zunächst noch abstrakter – **Verwalter** zugeordnet, der die Aufgabe hat, die Komponenten der Sphäre zu realisieren und die dafür erforderliche Ressourcenverwaltung in Kooperation mit anderen Verwaltern durchzuführen. Dazu gehört insbesondere die Aufgabe, dem Akteur der Sphäre die Ausführung seiner Berechnungen zu ermöglichen, indem er dem Akteur die hierfür erforderlichen Ressourcen zur Verfügung stellt und diese verwaltet. Die Aufgaben und Eigenschaften der Verwalter bleiben in diesem Sinne zunächst abstrakt[2]; sie werden später gemäß der *top–down* Vorgehensweise schrittweise für die einzelnen Realisierungsebenen konkretisiert.

Ein Verwalter wird mit dem Akteur, dessen Sphäre er zugeordnet ist, erzeugt und wieder aufgelöst. Damit wird die Dynamik des Systems vergröbert auf die Menge der Verwalter übertragen. Ebenso läßt sich die Strukur über der jeweiligen Menge von Akteursphären auf die

[2] Verwalter sind jedoch keine Komponenten eines INSEL$^+$–Systems der Sprachebene, sondern werden dessen Akteursphären in einem ersten Konkretisierungsschritt assoziiert.

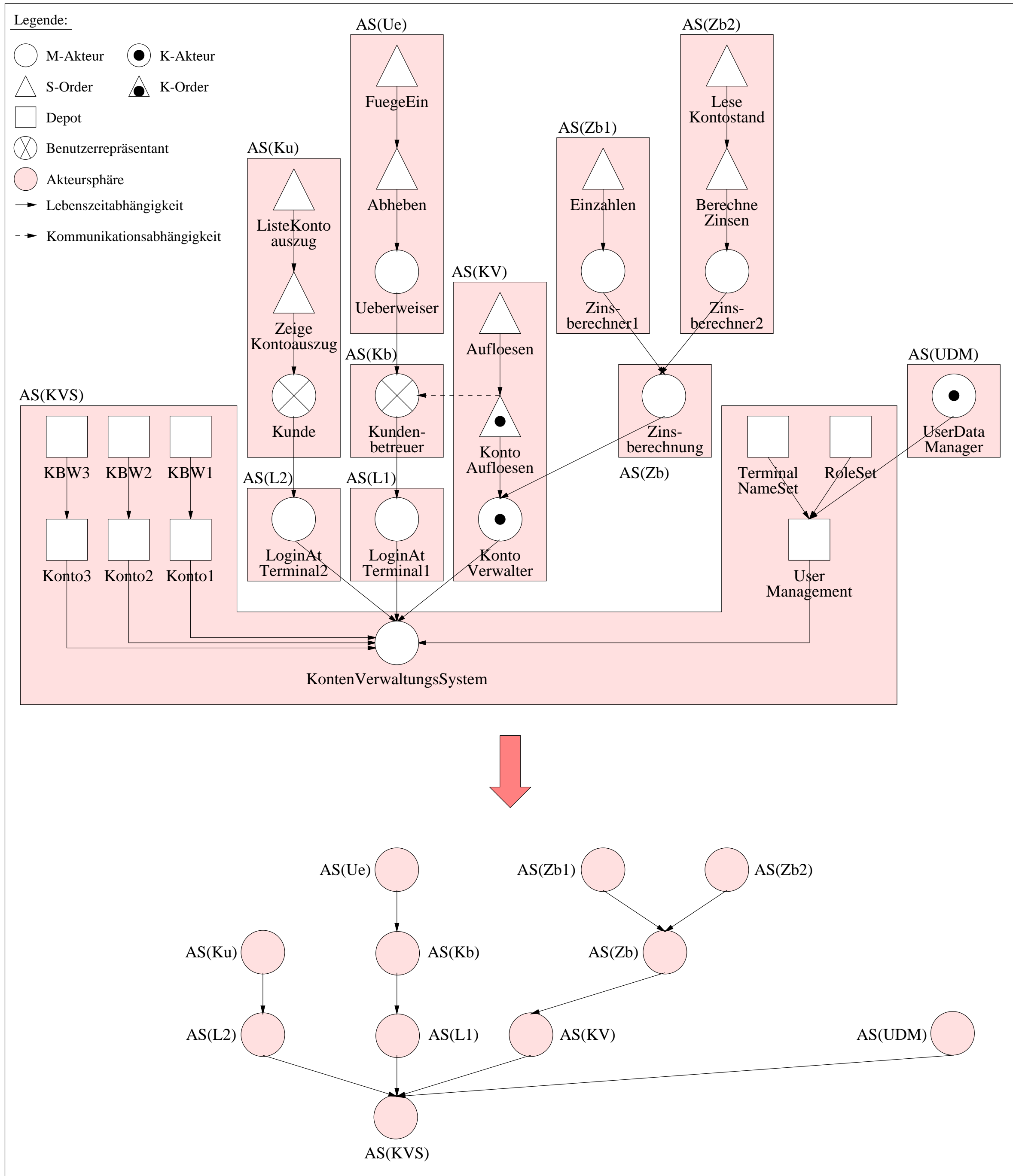

Abbildung 6.1: Vergröberung einer Systemkonfiguration des Kontenverwaltungssystems zu einer Menge von Akteursphären

diese realisierende Menge von Verwaltern übertragen, woraus sich eine Baumstruktur über der Menge der Verwalter ergibt, die für notwendige Kooperationen zwischen den Verwaltern zur Erfüllung ihrer Aufgaben genutzt werden kann. Diese baumstrukturierte Menge untereinander kooperierender Verwalter, die sich der Dynamik des Systems entsprechend entwickelt, wird im weiteren als **Verwalterarchitektur** bezeichnet.

Beispiel

In Abbildung 6.2 ist der sich für die Systemkonfiguration des Kontenverwaltungssystems aus Abbildung 6.1 ergebende Baum von Verwaltern dargestellt. Die Zuordnung je eines Verwalters zu einer Akteursphäre ist durch die Umrahmung des jeweiligen Verwalters und der Sphäre verdeutlicht. Der einer Akteursphäre $AS(a)$ zugeordnete Verwalter wird dabei mit $V(a)$ bezeichnet.

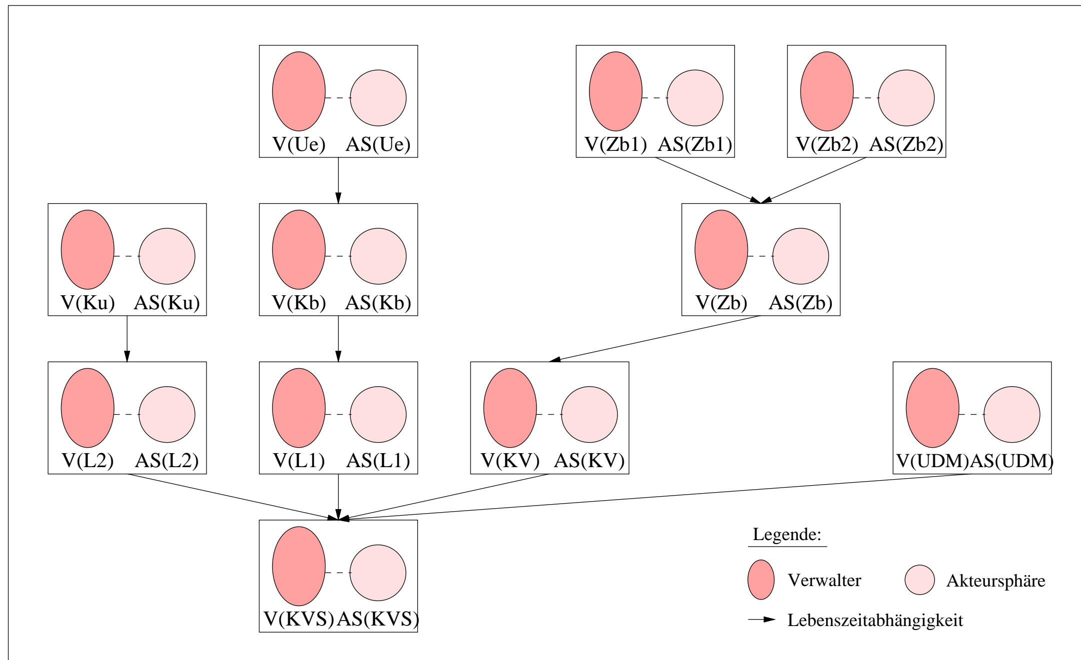

Abbildung 6.2: Verwalterbaum der Systemkonfiguration des Kontenverwaltungssystems aus Abbildung 6.1

◇

Wie oben bereits erwähnt, sind die abstrakten Verwalter dem *top–down* Ansatz entsprechend schrittweise zu konkretisieren und zu realisieren. Dazu gehört zum einen die Konkretisierung der von ihnen durchzuführenden Aufgaben und zum anderen die Konkretisierung der zur Erfüllung dieser Aufgaben eingesetzten Konzepte und Maßnahmen. Diese Konkretisierungen werden im folgenden Abschnitt 6.2 zunächst für die Realisierungsebene 1 vorgenommen. In Abschnitt 6.3 wird dann die Funktionalität der Verwalter den Gegebenheiten der Realisierungsebene 2 entsprechend angepaßt und erweitert. Der Schwerpunkt bei der Erläuterung der Eigenschaften der Verwalter liegt dabei auf den Aufgaben sowie den Konzepten und Maßnahmen, die für die Durchsetzung des Rechts eines INSEL$^+$–Systems benötigt werden.

6.2 Die Realisierungsebene 1

In diesem Abschnitt werden die Aufgaben und Eigenschaften der Verwalter für die Realisierungsebene 1 konkretisiert. Dazu werden zunächst in Abschnitt 6.2.1 die charakteristischen Eigenschaften dieser Realisierungsebene und die auf dieser Ebene zu lösenden Verwaltungs– und Sicherheitsprobleme erläutert. Zur Lösung dieser Probleme wird ein Verwalter in einen Speicherverwalter und einen Akteursphärenverwalter aufgeteilt. Die Akteursphärenverwalter übernehmen zum einen die Aufgabe der Erzeugung sowie der Terminierung und Auflösung von

INSEL$^+$–Inkarnationen. Zum anderen sind sie für die Durchführung von Zugriffskontrollen und die sichere Verwaltung der für diese Kontrollen benötigten Informationen zuständig. Die Beschreibung der Funktionalität und der Schnittstellenoperationen, die die Akteursphärenverwalter zur Verfügung stellen, erfolgt in Abschnitt 6.2.2. Die Zugriffskontrollen werden von den Akteursphärenverwaltern standardmäßig so durchgeführt, daß bei jedem Aufruf einer äußeren Operation einer zugriffskontrollierten Komponente der für die Operation festgelegten Zugriffsrestriktionsausdruck ausgewertet wird. Als Alternative zu diesem Vorgehen und motiviert durch das Ziel, durch flexiblen Einsatz unterschiedlicher Realisierungsmaßnahmen den Aufwand für die durchzuführenden Zugriffskontrollen zu reduzieren, werden in Abschnitt 6.2.3 Tickets eingeführt, deren Besitz zur Ausführung einer äußeren Operation einer zugriffskontrollierten Komponente berechtigt. Es wird erklärt, wie die Funktionalität der Akteursphärenverwalter um Fähigkeiten zur Verwaltung von Tickets, wozu insbesondere das Ausstellen und Löschen von Tickets gehört, erweitert wird.

6.2.1 Charakteristika und Konkretisierung der Verwalter

Die Realisierungsebene 1 ist charakterisiert durch eine endliche Menge P von abstrakten Prozessoren und einem linear adressierbaren abstrakten Speicher S beschränkter Kapazität. Der Speicher S sei für alle Prozessoren aus P gemeinsam mit konstanter Zugriffslatenz nutzbar, d.h. jeder Prozessor $p \in P$ kann auf jede Speicherzelle $s \in S$ mit gleicher Zugriffsdauer zugreifen. Bezüglich ihrer Eigenschaften ist die Realisierungsebene 1 also mit den Eigenschaften von UMA-Multiprozessoren[3] (siehe z.B. [Hwa93]) vergleichbar.

Die gestellte Aufgabe besteht darin, ein abstraktes INSEL$^+$–System mit den Ressourcen dieser Ebene zu realisieren. Dazu sind im wesentlichen die Erzeugung und Auflösung von Komponenten sowie die Zugriffe auf Komponenten zu realisieren. Bei Zugriffen auf zugriffskontrollierte Komponenten ist zu gewährleisten, daß die für das System festgelegten Zugriffsbeschränkungen durchgesetzt werden. Für die Realisierung eines Systems mit den Ressourcen der Realisierungsebene 1 sind eine Reihe von Verwaltungsaufgaben zu lösen, für die geeignete Realisierungskonzepte zu entwickeln sind. Die Vielfalt dieser Verwaltungsaufgaben motiviert eine funktionale Aufteilung der Aufgaben in Prozessorverwaltung, Speicherverwaltung, Komponentenverwaltung sowie die Durchführung von Zugriffskontrollen.

- Prozessorverwaltung
  Die Aufgabe der Prozessorverwaltung besteht darin, die Rechenfähigkeiten der Akteure mit den Prozessoren dieser Ebene zu realisieren. Dazu ist bei Erzeugung eines Akteurs ein Prozessor zu allokieren und mit dessen Auflösung wieder freizugeben.

- Speicherverwaltung
  Der globale Speicher ist durch die Speicherverwaltung so zu verwalten, daß für die Erzeugung von Komponenten Speicher für deren Repräsentation zur Verfügung steht. Mit der Auflösung einer DA–Inkarnation ist der für diese Inkarnation und ihre lokalen N–Komponenten belegte Speicherplatz wieder freizugegeben.

- Komponentenverwaltung
  Die Hauptaufgaben der Komponentenverwaltung liegen in der Erzeugung sowie der Termininierung und Auflösung von Komponenten und der Verwaltung der hierfür erforderlichen Datenstrukturen. Um die in Abschnitt 3.3 erklärten Terminierungs– und

[3] *uniform memory access*

Auflösungsregeln für INSEL$^+$–Systeme korrekt umzusetzen, ist von der Komponentenverwaltung vor der Auflösung einer DA–Inkarnation die Abschlußsynchronisation für π-innere Akteure durchzuführen.

- Durchführung von Zugriffskontrollen
  Vor der Erzeugung einer Inkarnation bzgl. eines zugriffskontrollierten DA–Generators sind Zugriffskontrollen durchzuführen, die in der Auswertung des für diesen Generator festgelegten Zugriffsrestriktionsausdrucks bestehen. Die für die Realisierung dieser Kontrollen benötigten Informationen bzw. die entsprechenden Datenstrukturen sind sicher zu verwalten.

Die Verwaltungsaufgaben sind von den in Abschnitt 6.1 eingeführten Verwaltern zu erledigen, die hierfür geeignet zu konkretisieren sind. Wie bereits erwähnt, ist die Entwicklung von Konzepten für die in den Bereichen der Speicherverwaltung und Prozessorverwaltung anfallenden Aufgaben nicht Ziel dieser Arbeit. Sie werden deshalb postuliert und lediglich soweit erklärt, wie dies für das Verständnis der Realisierungsmaßnahmen in den Aufgabenbereichen Komponentenverwaltung und Zugriffskontrollen notwendig ist. Detailliertere Erläuterungen sind für den Bereich der Prozessorverwaltung in [Rad96] und für den Bereich der Speicherverwaltung in [Win96] und [Gro98] zu finden.

Für die Prozessorverwaltung auf der Realisierungsebene 1 wird ein **globaler Prozessorverwalter** angenommen. Dieser hat die Aufgabe, n virtuelle Prozessoren zur Realisierung der Rechenfähigkeiten von n Akteuren des Systems mit den gegebenen $|P|$ Prozessoren zu realisieren. Der globale Prozessorverwalter bietet Operationen zum Allokieren und Deallokieren von virtuellen Prozessoren an. Die Schedulingstrategie zur Realisierung der virtuellen Prozessoren mit den $|P|$ Prozessoren wird hier nicht näher betrachtet. Es wird lediglich verlangt, daß Informationsflüsse zwischen den Prozessoren über verdeckte Kanäle, wie z.B. die Dauer der Bindung an einen Prozessor oder nicht bereinigte Registerinhalte, ausgeschlossen sind.

Zur Erledigung der Aufgaben der Speicherverwaltung auf der Seite und der Komponentenverwaltung sowie der Durchführung der Zugriffskontrollen auf der anderen Seite wird der einer Akteursphäre zugeordnete Verwalter zu jeweils einem Speicherverwalter und einem Akteursphärenverwalter konkretisiert.

Die **Speicherverwalter** übernehmen die Aufgabe der Verwaltung des globalen Speichers S und bieten Operationen zur Allokation und Deallokation von Haldenspeicher bzw. Kellerspeicher an. Sie kooperieren zur Erfüllung der Speicherverwaltungsaufgabe im wesentlichen entlang der Baumstruktur, die sich durch Übertragung der Struktur der Verwalterarchitektur auf die Menge der Speicherverwalter ergibt. Im wesentlichen besteht die Aufgabe eines Speicherverwalters darin, dem entsprechenden Akteursphärenverwalter den Speicher bereitzustellen, den dieser für die Realisierung der ihm assoziierten Akteursphäre benötigt. Einzelheiten zur Funktionalität der Speicherverwalter und zu den von ihnen durchgeführten Verwaltungsmaßnahmen sind in [Win96] zu finden.

Die **Akteursphärenverwalter** sind für die Erzeugung und die korrekte Auflösung von Komponenten gemäß der INSEL–Auflösungsregeln zuständig. Sie verwalten dazu für jede ihrer Akteursphäre zugeordnete Komponente eine sogenannte Komponentenbeschreibung, auf die später noch genauer eingegangen wird. Da die Maßnahmen zur Durchführung von Zugriffskontrollen, wie in Abschnitt 6.1 erklärt, in die Maßnahmen zur Realisierung der Erzeugung von DA–Inkarnationen einzuordnen sind, ist es naheliegend, die Aufgaben der Zugriffskontrolle und der Verwaltung der hierfür benötigten Informationen ebenfalls den Akteursphärenverwaltern zu übertragen. Der bei einem Zugriff auf eine zugriffskontrollierte Komponente

auszuwertende Zugriffsrestriktionsausdruck wird dann von dem Akteursphärenverwalter ausgewertet, dessen Akteursphäre die zugriffskontrollierte Komponente angehört. Dazu verwaltet ein Akteursphärenverwalter u.a. für jede ihm zugeordnete zugriffskontrollierte Komponente eine Zugriffskontrolliste.

Als Alternative zu diesem Vorgehen könnte zum einen die Aufgabe der Durchführung der Zugriffskontrollen von zusätzlich eingeführten Verwaltern, sogenannten **Zugriffskontrolleuren**, wahrgenommen werden. Zum anderen ergeben sich in diesem Zusammenhang Alternativen bzgl. der Zuordnung von zugriffskontrollierten Komponenten zu Zugriffskontrolleuren und damit für die Anzahl der zu erzeugenden Zugriffskontrolleure.

Der Ansatz, spezielle Zugriffskontrolleure einzuführen, hat vor allem unter dem Gesichtspunkt einer effizienten Realisierung von INSEL$^+$–Systemen einige Nachteile. Diese Nachteile liegen zum einen in dem zusätzlichen Aufwand, der für die Erzeugung und Auflösung der Zugriffskontrolleure zu leisten ist, und zum anderen in dem zusätzlichen Aufwand für die Durchführung der Zugriffskontrollen, der sich aus der zusätzlichen Kooperation zwischen Akteursphärenverwaltern und Zugriffskontrolleuren bei der Erzeugung von Inkarnationen bzgl. zugriffskontrollierter DA–Generatoren ergibt.

Ein Vorteil dieses alternativen Ansatzes könnte unter dem Aspekt der Gewährleistung der Vertrauenswürdigkeit der Komponenten, die Zugriffskontrollen durchführen, gesehen werden. Die **Vertrauenswürdigkeit** einer Komponente besagt, daß die implementierte Ist–Funktionalität der Komponente ihrer spezifizierten Soll–Funktionalität entspricht. Neben Zuverlässigkeitsaspekten beinhaltet die Vertrauenswürdigkeit insbesondere den Schutz der Komponente vor absichtlicher oder unabsichtlicher Manipulation ihrer Funktionalität. Vertrauenswürdige Komponenten sind als Bestandteil der sogenannten Trusted Computing Base (TCB) zu implementieren. Hierbei sind geeignete Maßnahmen zur Gewährleistung der Vertrauenswürdigkeit einzusetzen. Der Aufwand für diese Maßnahmen ist im allgemeinen um so größer, je höher die Anzahl der Komponenten ist, die vertrauenwürdig zu konstruieren sind. Unter diesem Aspekt erscheint es sinnvoll, die Funktionalität der Akteursphärenverwalter und der Zugriffskontrolleure zu trennen und lediglich eine geringe Anzahl von Zugriffskontrolleuren zu erzeugen. Daraus würde sich allerdings lediglich dann ein Vorteil ergeben, wenn für die Akteursphärenverwalter keine entsprechenden Maßnahmen zur Gewährleistung der Vertrauenswürdigkeit einzusetzen wären. Dies ist jedoch nicht der Fall, da die korrekte Funktionalität der Akteursphärenverwalter Voraussetzung für die Realisierung der funktionalen Eigenschaften eines INSEL$^+$–Systems ist. Wird die Vertrauenswürdigkeit der Akteursphärenverwalter nicht gewährleistet, so ist generell in Frage zu stellen, ob das „realisierte“ System die Eigenschaften des „abstrakten“ Systems implementiert, d.h. ob die Ist–Funktionalität des Systems seiner mit den Sprachkonzepten festgelegten Soll–Funktionalität entspricht.

Als Fazit dieser Überlegungen läßt sich festhalten, daß aufgrund der Tatsache, daß für die Akteursphärenverwalter sowieso Maßnahmen zur Gewährleistung ihrer Vertrauenswürdigkeit einzusetzen sind, die genannten Nachteile des Ansatzes der Trennung von Akteursphärenverwaltern und Zugriffskontrolleuren überwiegen. In dem in dieser Arbeit verfolgten Ansatz ist deshalb der einer Akteursphäre zugeordnete Verwalter zum einen Ressourcen–Verwalter und zum anderen Zugriffskontrolleur. Die Akteursphärenverwalter sind somit als Bestandteile der Trusted Computing Base zu konstruieren. Auf die Realisierung der Trusted Computing Base und damit auf Maßnahmen zur Gewährleistung der Vertrauenswürdigkeit wird an dieser Stelle nicht weiter eingegangen, da diese von den Basisschutzmechanismen der zugrundeliegenden Hardware und des eingesetzten Betriebssystemskerns abhängig sind.

Das Spektrum der sich für die Zuordnung von zugriffskontrollierten Komponenten zu Zugriffskontrolleuren bietenden Möglichkeiten wird durch zwei Extreme begrenzt, die im folgenden

kurz erläutert werden.

1. Zuordnung eines „eigenen“ Zugriffskontrolleurs zu jeder zugriffskontrollierten Komponente
   Der einer zugriffskontrollierten Komponente assoziierte Zugriffskontrolleur ist ausschließlich für die Durchführung der Kontrollen bei Zugriffen auf diese Komponente zuständig. Dazu verwaltet er insbesondere eine Zugriffskontrolliste für diese Komponente.

2. Zuordnung aller zugriffskontrollierten Komponenten zu einem zentralen Zugriffskontrolleur
   Der zentrale Zugriffskontrolleur kontrolliert alle Zugriffe auf alle zugriffskontrollierten Komponenten des Systems und verwaltet die dafür benötigten Informationen wie z.B. Zugriffskontrollisten.

Die zweite Möglichkeit entspricht dem bekannten zentralen Referenzmonitor–Konzept (siehe z.B. [GD72]), für das sich jedoch gezeigt hat, daß es für verteilte Systeme wenig geeignet ist [Neu91]. Die erste Alternative zeichnet sich durch die sehr feingranulare Zuordnung von zugriffskontrollierten Komponenten zu Zugriffskontrolleuren aus. Dies führt jedoch im allgemeinen zu einer sehr hohen Anzahl von Zugriffskontrolleuren, wobei ein Großteil der Zugriffskontrolleure in der Regel wenig ausgelastet sein wird. Unter dem bereits angesprochenen Aspekt der Gewährleistung der Vertrauenswürdigkeit der Zugriffskontrolleure und dem dafür zu leistenden Aufwand ist diese Möglichkeit deshalb ebenfalls wenig attraktiv. Zwischen den beiden erläuterten Extremen sind zahlreiche Alternativen denkbar, die sich jeweils darin unterscheiden, welcher Menge von zugriffskontrollierten Komponenten jeweils ein Zugriffskontrolleur zugeordnet wird. Von diesen Möglichkeiten ist die oben beschriebene, nämlich der Menge der zugriffskontrollierten Komponenten, die einer Akteursphäre angehören, jeweils einen Zugriffskontrolleur zuzuordnen, die attraktivste. Dies läßt sich zum einen damit begründen, daß sich bei diesem Vorgehen die Zugriffskontrolleure als Teile der Akteursphärenverwalter systematisch in die *top–down* abgeleitete Verwalterarchitektur für INSEL$^+$–Systeme einordnen lassen. Zum anderen ist an der Realisierung eines Zugriffes auf eine Komponente stets der Akteursphärenverwalter, dessen Akteursphäre diese Komponente zugeordnet ist, beteiligt. Aus der Tatsache, daß dieser Verwalter im Fall eines Zugriffes auf eine zugriffskontrollierte Komponente auch den entsprechenden Zugriffsrestriktionsausdruck auswertet, folgt, daß für den „Anstoß“ der Auswertung des Zugriffsrestriktionsausdrucks, also für die Zugriffskontrolle, kein zusätzlicher Verwalteraufruf notwendig ist. Diese Aussage bezieht sich lediglich auf die Initiierung der Auswertung eines Zugriffsrestriktionsausdrucks, da zur vollständigen Auswertung eines Zugriffsrestriktionsausdrucks unter Umständen Operationen anderer Akteursphärenverwalter aufzurufen sind, um z.B. die Zugriffskontrollisten anderer zugriffskontrollierter Komponenten auszuwerten. Die Akteursphärenverwalter kooperieren also im allgemeinen zur Durchführung der Zugriffskontrollen.

Im folgenden wird die Funktionalität der Akteursphärenverwalter genauer erklärt und zwar schwerpunktmäßig in bezug auf die von ihnen wahrzunehmenden Zugriffskontrollaufgaben. Dazu werden insbesondere Schittstellenoperationen der Akteursphärenverwalter angegeben, die von den mit den virtuellen Prozessoren realisierten Akteuren bzw. von anderen Akteursphärenverwaltern aufgerufen werden können. Die Akteursphärenverwalter werden hier lediglich konzeptionell beschrieben. Auf ihre konkrete Realisierung (d.h. Implementierung) und die in diesem Zusammenhang zu beantwortenden Fragen der Speicherrepräsentation der von ihnen zu verwaltenden Datenstrukturen und der Synchronisation der Zugriffe auf diese sowie auf

die Frage nach der Realisierung von Verwaltern als aktive oder passive Objekte wird in dieser Arbeit nicht eingegangen. Die Vor– und Nachteile der Realisierung von Verwaltern als passive bzw. aktive Objekte werden in [Win96] diskutiert. Dort wird auch eine Implementierung von Akteursphärenverwaltern als passive C++–Objekte ([Str91]) beschrieben. In [Rad96] ist eine Realisierung von Akteursphärenverwaltern als aktive Objekte auf Basis des dort beschriebenen verteilten Thread–Kerns DTK (*Distributed Thread Kernel*) angegeben. Für das Weitere wird lediglich festgelegt, daß die Schnittstellenoperationen der Akteursphärenverwalter mit der Semantik von Prozeduraufrufen ausgeführt werden.

6.2.2 Akteursphärenverwalter

In diesem Abschnitt wird für die einzelnen von den Akteursphärenverwaltern wahrzunehmenden Aufgaben angegeben, welche Funktionalität von den Verwaltern jeweils zur Erfüllung der Aufgabe zur Verfügung gestellt wird. Dazu werden zum einen die jeweiligen Schnittstellenoperationen der Akteursphärenverwalter erklärt und zum anderen die verwendeten Datenstrukturen angegeben. Die von den Akteursphärenverwaltern wahrzunehmenden Aufgaben lassen sich wie folgt untergliedern:

1. Erzeugung von Inkarnationen
2. Terminierung und Auflösung von Inkarnationen
3. Realisierung des Operationen–orientierten Rendezvous (K-Order–Realisierung)
4. Verwaltung der für Zugriffskontrollen benötigten Informationen
5. Durchführung von Zugriffskontrollen

Die in den Punkten 1. bis 3. genannten Aufgaben entsprechen dabei den in [Win96] erläuterten Aufgaben der dort eingeführten Akteursphärenverwalter der Hardware–nahen Ebene. Die zur Erledigung dieser Aufgaben bereitgestellten Funktionalitäten werden deshalb an dieser Stelle nur soweit beschrieben, wie sie für das Verständnis der Maßnahmen zur Durchführung von Zugriffskontrollen nötig sind.

6.2.2.1 Erzeugung von Inkarnationen

Für die Erzeugung von Inkarnationen bietet ein Akteursphärenverwalter Operationen an, die von dem realisierten Akteur, dem der Akteursphärenverwalter zugeordnet ist, aufgerufen werden. Desweiteren stellt er Operationen zur Verfügung, die von anderen Akteursphärenverwaltern im Rahmen der für die Erzeugung von Inkarnationen notwendigen Kooperation zwischen Akteursphärenverwaltern aufgerufen werden. Die generelle Funktionalität dieser Operationen besteht in der Allokation von Speicher zur Realisierung der Speicherfähigkeiten der Inkarnation sowie zusätzlich in der Allokation eines virtuellen Prozessors für den Fall einer Akteur–Erzeugung. Ferner wird für die erzeugte Inkarnation von dem Akteursphärenverwalter, dessen Sphäre die Inkarnation zugeordnet ist, eine **Komponentenbeschreibung** angelegt und verwaltet. Diese enthält u.a. den Identifikator der Inkarnation sowie die Anfangsadresse der Speicherrepräsentantion der Inkarnation. Der Identifikator einer Inkarnation ist systemweit eindeutig und ermöglicht damit die systemweit eindeutige Identifikation der Komponente[4]. Die Komponentenbeschreibung wird zum einen für die korrekte Freigabe

[4] Auf die Struktur der Identifikatoren wird hier nicht weiter eingegangen.

des von der entsprechenden Komponente belegten Speichers bei Auflösung der Komponente benötigt. Zum anderen werden in der Komponentenbeschreibung komponentenspezifische Informationen verwaltet, wie z.B. eine Zugriffskontrolliste oder eine Zugriffshistorienliste (siehe dazu Abschnitt 6.2.2.4).

Die Schnittstellenoperationen eines Akteursphärenverwalters zur Erzeugung von Inkarnationen lassen sich zum einen unterteilen in Operationen zur Erzeugung nicht zugriffskontrollierter Inkarnationen, in Operationen zur Erzeugung zugriffskontrollierter Komponenten und in Operationen, die die Erzeugung einer Komponentenbeschreibung für einen DA–Generator bewirken. Zum anderen lassen sich diese Operationen, wie bereits oben erwähnt, danach klassifizieren, ob sie von dem realisierten Akteur aufgerufen werden, dem der Akteursphärenverwalter zugeordnet ist, oder von anderen Akteursphärenverwaltern. Tabelle 6.1 gibt einen Überblick über die Schnittstellenoperationen zur Erzeugung von Inkarnationen. Die erste Spalte enthält den Bezeichner der jeweiligen Operation; die zweite Spalte gibt an, ob diese Operation von einem Akteur oder einem Akteursphärenverwalter (kurz: ASV) aufgerufen wird; die dritte Spalte beschreibt grob die Funktionalität der Operation und in der vierten Spalte ist schließlich angegeben, zu welcher Art von Operation gemäß obiger Klassifikation die Operation gehört. In der Tabelle wird für das Wort 'zugriffskontrolliert' abkürzend 'zk' geschrieben, 'KOp' steht für 'Kommunikationsoperation' und 'ZOp' steht für 'Zugriffsoperation'.

Operation	**Aufruf durch**	**Erzeugung von**	**Art**
`CreateSOrder`	Akteur	S-Order	nicht zk Komponente
`CreateDepot`	Akteur	Depot	
`CreateDepotDesc`	ASV		
`CreateMActor`	Akteur	M-Akteur	
`CreateMActorDesc`	ASV		
`CreateKActor`	Akteur	K-Akteur	
`CreateKActorDesc`	ASV		
`CreateKOrder`	Akteur	KOp auf K-Akteur (K-Order)	
`CreateDepotOp`	Akteur	ZOp auf Depot	
`CreateDEInc`	Akteur	DE–Inkarnation	
`CreateDEIncDesc`	ASV		
`CreateProtSOrder`	Akteur	zk S-Order	zk Komponente
`CreateProtDepot`	Akteur	zk Depot	
`CreateProtDepotDesc`	ASV		
`CreateProtMActor`	Akteur	zk M-Akteur	
`CreateProtMActorDesc`	ASV		
`CreateProtKActor`	Akteur	zk K-Akteur	
`CreateProtKActorDesc`	ASV		
`CreateProtKOrder`	Akteur	KOp auf zk K-Akteur	
`CreateProtDepotOp`	Akteur	ZOp auf zk Depot	
`CreateGenDesc`	Akteur	Generatorbeschreibung	Generator-Beschreibung
`CreateProtGenDesc`	Akteur	zk Generatorbeschreibung	
`CreateASManager`	ASV	ASV	ASV

Tabelle 6.1: Schnittstellenoperationen der Akteursphärenverwalter zur Erzeugung von Inkarnationen

Für die Erzeugung einer Inkarnation, die nicht der Akteursphäre des Akteurs zugeordnet

wird, der die *erzeuge*–Operation ausführt, werden zwei Operationen benötigt. Inkarnationen, für die dies immer der Fall ist, sind Akteure, da mit der Erzeugung eines Akteurs eine neue Akteursphäre entsteht. Inkarnationen, für die es möglich ist, daß sie nicht der Akteursphäre des erzeugenden Akteurs zugeordnet werden, sind anonyme Depots und anonyme DE–Inkarnationen[5]. Dementsprechend sind in der Tabelle für diese Inkarnationsarten zwei Operationen angegeben (zum Beispiel: `CreateDepot` und `CreateDepotDesc` für die Erzeugung von Depots). Die erste Operation (im Beispiel: `CreateDepot`) wird von dem erzeugenden Akteur auf seinem Akteursphärenverwalter aufgerufen. Dieser Akteursphärenverwalter leitet dann den Auftrag zur Erzeugung der Inkarnation durch Aufruf der zweiten Operation (im Beispiel: `CreateDepotDesc`) an den Akteursphärenverwalter weiter, dessen Sphäre die zu erzeugende Inkarnation zuzuordnen ist. Im Falle der Erzeugung eines Akteurs ist dabei zunächst durch Aufruf der Operation `CreateASManager` ein neuer Akteursphärenverwalter zu erzeugen, auf dem dann anschließend die Operation `CreateMActorDesc` bzw. `CreateKActorDesc` aufgerufen wird. Die Operation `CreateDepotOp` dient zur Realisierung der Ausführung einer Zugriffsoperation eines Depots, die der *erzeuge*–Operation eines lokalen exportierten DA–Generators des Depots entspricht. Ihre Ausführung bewirkt dementsprechend die Erzeugung einer Inkarnation bzgl. dieses Generators.

Die Zusatzfunktionalität der Operationen, die ein Akteur zur Erzeugung einer zugriffskontrollierten Inkarnation auf seinem Akteursphärenverwalter aufruft, besteht darin, daß vor der eigentlichen Erzeugung der Inkarnation zunächst der entsprechende Zugriffsrestriktionsausdruck ausgewertet wird (siehe hierzu Abschnitt 6.2.2.5). Die Operation `CreateProtDepotOp` wird z.B. für die Erzeugung von Inkarnationen, die der Zugriffsoperation eines zugriffskontrollierten Depots entsprechen, benötigt. Ihre Ausführung bewirkt, daß vor Erzeugung der Inkarnation durch Aufruf einer Operation auf dem Akteursphärenverwalter, dessen Sphäre das zugriffskontrollierte Depot zugeordnet ist, der für die Zugriffsoperation festgelegte Zugriffsrestriktionsausdruck ausgewertet wird. Ist das Ergebnis dieser Auswertung positiv, wird die entsprechende Inkarnation erzeugt. Anderenfalls terminiert `CreateProtDepotOp`, ohne daß eine Inkarnation erzeugt wird. In diesem Fall wird die in dem Status–Part des entsprechenden INSEL$^+$–Erzeugungskonstrukts angegebene boolesche Variable (vgl. Abschnitt 4.5) mit dem Wert *false* belegt, was bewirkt, daß die in dem INSEL$^+$–Programm für die Behandlung dieses Zugriffverbots explizit angegebenen Anweisungen ausgeführt werden.

Die Operationen `CreateGenDesc` und `CreateProtGenDesc` dienen zur Erzeugung einer Komponentenbeschreibung für einen nicht zugriffskontrollierten DA–Generator bzw. einen explizit zugriffskontrollierten DA–Generator. Wie später noch genauer zu sehen sein wird, ist es nicht notwendig, für jeden DA–Generator eine Komponentenbeschreibung anzulegen, sondern lediglich für die explizit zugriffskontrollierten DA–Generatoren und die DA–Generatoren, die potentiell in einem `ACCESSED`–Prädikat auftreten. `CreateGenDesc` bzw. `CreateProtGenDesc` werden von einem Akteur bei Erarbeitung der Definition eines solchen DA–Generators auf dem Akteursphärenverwalter aufgerufen, dessen Sphäre dieser DA–Generator zugeordnet wird. Ein DA–Generator wird dabei der Akteursphäre zugeordnet, der die DA–Inkarnation, die den Generator als lokale N–Komponente enthält, angehört.

6.2.2.2 Terminierung und Auflösung von Inkarnationen

Die Akteursphärenverwalter sind für die Umsetzung der in Abschnitt 3.3 festgelegten Terminierungsregel für DA–Inkarnationen und für die Auflösung von Inkarnationen nach den

[5]K-Order bilden unter diesem Aspekt einen Spezialfall. Sie sind während ihrer Ausführung der Akteursphäre des ausführenden K-Akteurs zugeordnet, ansonsten der Akteursphäre des erzeugenden Akteurs. Ihre Erzeugung wird erst unter Punkt 3. „K-Order–Realisierung“ behandelt.

ebenfalls dort angegebenen Auflösungregeln zuständig. Die Terminierungsregel besagt, daß eine DA–Inkarnation erst dann terminieren und aufgelöst werden kann, wenn alle π–inneren Akteure terminiert sind. Die Auflösungsregeln legen fest, daß mit der Auflösung eines Akteurs oder einer Order alle von dieser Inkarnation lebenszeitmäßig abhängigen Inkarnationen aufzulösen sind.

Zur korrekten Umsetzung dieser konzeptionell festgelegten Regeln verwaltet jeder Akteursphärenverwalter die von ihm bei der Erzeugung von Inkarnationen bzw. bei der Erarbeitung von DA–Generatoren angelegten Komponentenbeschreibungen in einem zweidimensionalen **Lebenszeitkeller**. Der Lebenszeitkeller ist zweidimensional in dem Sinne, daß er vertikal und horizontal wachsen kann. In der vertikalen Richtung werden die Komponentenbeschreibung des Akteurs der Akteursphäre und die Beschreibungen der von dem Akteur ausgeführten Order gemäß der LIFO[6]–Semantik verwaltet. Die Komponentenbeschreibungen der von dem Akteur und diesen Ordern lebenszeitmäßig abhängigen Komponenten werden dann ausgehend von der jeweiligen Akteur– bzw. Order–Beschreibung in horizontaler Richtung verwaltet. Hierzu gehören die Komponentenbeschreibungen lebenszeitmäßig abhängiger Depots, Akteure, DE–Inkarnationen und DA–Generatoren. Die Beschreibungen der lebenszeitmäßig abhängigen Akteure sind jedoch nicht vollständige Komponentenbeschreibungen, sondern enthalten lediglich einen Verweis auf den jeweiligen Akteur.

Beispiel

Die Abbildung 6.3 zeigt die Lebenszeitkeller der Akteursphärenverwalter für die in den Abbildungen 6.1 und 6.2 betrachtete Systemkonfiguration des Kontenverwaltungssystems. Die Komponentenbeschreibungen von DE–Inkarnationen und DA–Generatoren sind aus Gründen der Übersichtlichkeit nicht dargestellt. Da alle Konto–Depots lebenszeitmäßig von der Hauptkomponente KVS des Kontenverwaltungssystems abhängig sind, werden ihre Komponentenbeschreibungen in dem Lebenszeitkeller des Akteursphärenverwalters $ASV(KVS)$ horizontal angebunden an die Beschreibung der Hauptkomponente verwaltet. An der Abbildung wird deutlich, daß ein Lebenszeitkeller zwei Arten von Beschreibungen enthalten kann. Zum einen vollständige Komponentenbeschreibungen für alle Arten von Inkarnationen und DA–Generatoren und zum anderen Verweise auf lebenszeitmäßig abhängige Akteure.

◇

Operation	Funktionalität	Aufruf
`InsertEpsRef`	Einfügen eines Akteurverweises in einen Lebenszeitkeller	ASV
`WaitForActors`	Warten auf Terminierung π–innerer Akteure	Akteur
`SignalActorTerm`	Melden der Terminierung eines π–inneren Akteurs	ASV
`Terminate`	Terminierung und ggf. Auflösung einer DA–Inkarnation	Akteur

Tabelle 6.2: Schnittstellenoperationen der Akteursphärenverwalter zur Terminierung und Auflösung von Inkarnationen

Die Schnittstellenoperationen eines Akteursphärenverwalters, die die Terminierung und Auflösung von Inkarnationen sowie die Auflösung von DA–Generator–Beschreibungen betreffen, sind in Tabelle 6.2 angegeben. Die erste Spalte enthält den Bezeichner der jeweiligen

[6] *Last In First Out*

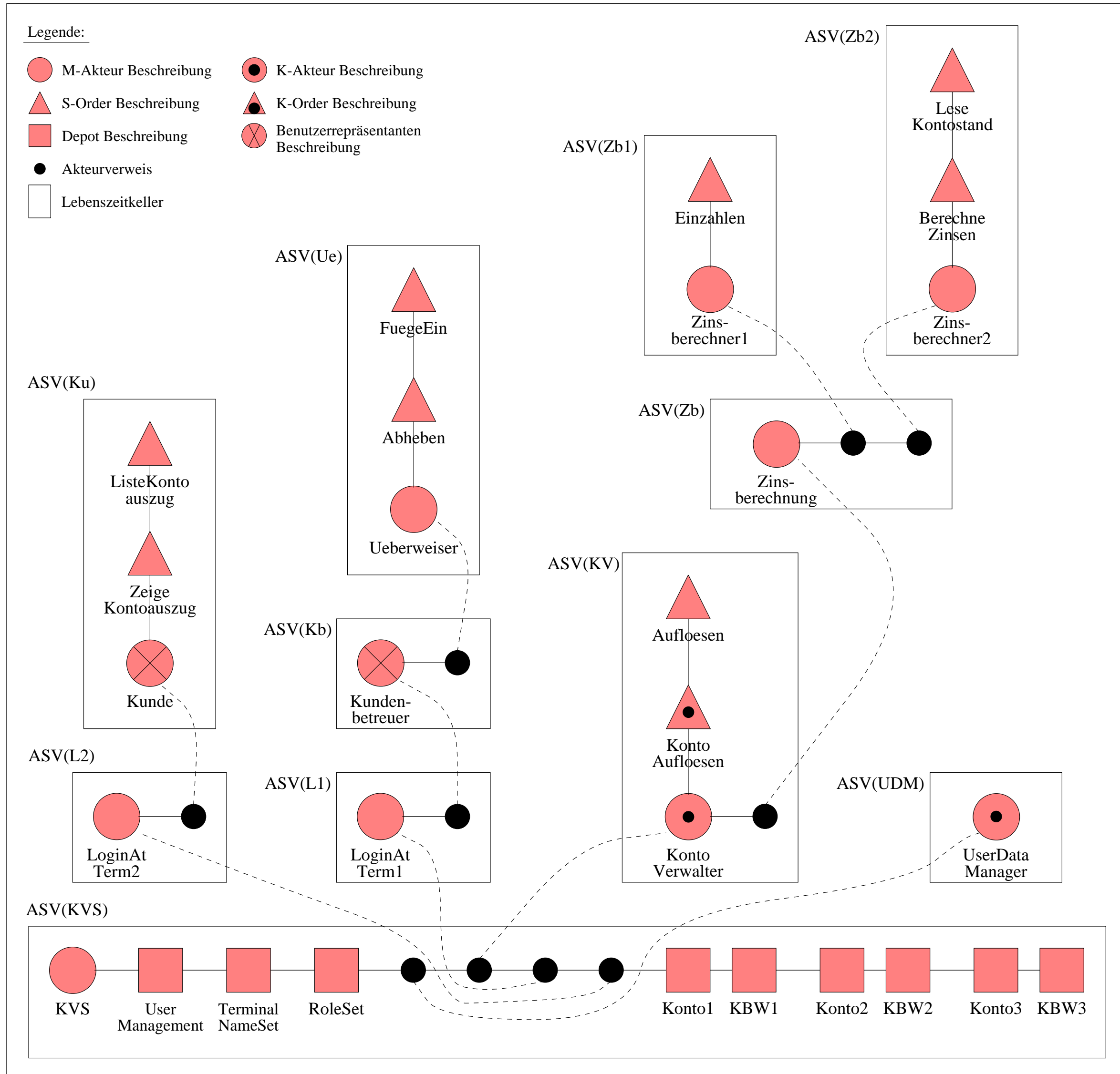

Abbildung 6.3: Lebenszeitkeller der Akteursphärenverwalter des Kontenverwaltungssystems für die Systemkonfiguration aus Abbildung 6.1

Operation; in der zweite Spalte ist die Funktionalität der Operation beschrieben und die dritte Spalte gibt an, ob diese Operation von einem Akteur oder einem Akteursphärenverwalter aufgerufen wird.

Die Operation `InsertEpsRef` wird im Rahmen der Erzeugung eines neues Akteurs[7] A von dem für diesen Akteur erzeugten Akteursphärenverwalter bei Ausführung der Operation `Create[Prot]MActorDesc` bzw. `Create[Prot]KActorDesc` auf dem Akteursphärenverwalter ASV aufgerufen, der die DA–Inkarnation I verwaltet, von der A lebenszeitmäßig abhängig ist. Ihre Ausführung bewirkt, daß ein Verweis auf den Akteur A in dem Teil des Lebenszeitkellers von ASV angelegt wird, der ausgehend von der Komponentenbeschreibung der DA–Inkarnation I horizontal wächst.

[7]Um zu unterscheiden, ob von einer Komponente eines INSEL+–Systems auf der Sprachebene oder von einer realisierten Komponente auf der Realisierungsebene 1 die Rede ist, werden letztere im weiteren mit Großbuchstaben bezeichnet.

Die Operation `WaitForActors` wird von einem Akteur A am Ende der Ausführung der kanonischen Operation einer DA–Inkarnation[8] I auf seinem Akteursphärenverwalter $ASV(A)$ aufgerufen, um auf die Terminierung π–innerer Akteure der DA–Inkarnation I zu warten. Die Akteure, auf deren Terminierung zu warten ist, bzw. Verweise auf diese, sind in den von der Komponentenbeschreibung der DA–Inkarnation I in horizontaler Richtung angebundenen Komponentenbeschreibungen enthalten. Terminiert ein solcher Akteur B, ruft dieser die Operation `Terminate` auf seinem Akteursphärenverwalter $ASV(B)$ auf. Die Ausführung von `Terminate` durch $ASV(B)$ bewirkt u.a., daß die Operation `SignalActorTerm` auf dem Akteursphärenverwalter $ASV(A)$ aufgerufen wird, der die DA–Inkarnation I verwaltet, von der B lebenszeitmäßig abhängig ist. Dieser trägt einen entsprechenden Vermerk über die Terminierung des Akteurs B in den in seinem Lebenszeitkeller enthaltenen Akteurverweis auf B ein. Die Operation `WaitForActors` blockiert den aufrufenden Akteur A solange, bis für alle an die Komponentenbeschreibung von I horizontal angebundenen Akteurverweise ein entsprechender Vermerk vorhanden ist. Anschließend wird die Abschlußsynchronisation durchgeführt, die darin besteht, die Ausgabeparameter der π–inneren Akteure zu übernehmen.

Nach Aufruf und Terminierung von `WaitForActors` am Ende der Ausführung der kanonischen Operation der DA–Inkarnation I wird von dem Akteur A die Operation `Terminate` auf seinem Akteursphärenverwalter aufgerufen. Die Ausführung von `Terminate` bewirkt die Auflösung der DA–Inkarnation I, falls I kein Depot ist, und aller von I lebenszeitmäßig abhängigen Komponenten. Dies sind alle Komponenten, deren Komponentenbeschreibung horizontal an die Komponentenbeschreibung von I angebunden ist. Für den Fall, daß I ein Depot ist, wird I gemäß der konzeptionell festgelegten Regeln nicht aufgelöst, sondern steht in terminiertem Zustand für die Nutzung über seine Zugriffsoperationen zur Verfügung. Bei Auflösung einer Inkarnation wird zunächst der von dieser Inkarnation belegte Speicher deallokiert und zusätzlich der belegte virtuelle Prozessor freigegeben, falls es sich bei der aufzulösenden Inkarnation um einen Akteur handelt. Anschließend wird die Komponentenbeschreibung der Inkarnation gelöscht. Wird ein Akteur aufgelöst, so werden zusätzlich der Akteursphären– und der Speicherverwalter des Akteurs aufgelöst.

6.2.2.3 Realisierung des Operationen–orientierten Rendezvous

Zur Realisierung des Operationen–orientierten Rendezvous–Konzepts und damit zur Realisierung von K-Order sind zum einen Warteräume für K-Order bereitzustellen und zu verwalten und zum anderen Operationen zur geforderten Synchronisation zwischen Auftraggeber und Auftragnehmer zur Verfügung zu stellen. Der Akteursphärenverwalter $ASV(K)$ eines K-Akteurs K stellt dazu einen **Warteraum** für jeden von dem K-Akteur K exportierten K-Order–Generator zur Verfügung, in dem Beschreibungen von K-Order–Aufrufen bzgl. dieses Generators gespeichert werden können. Die Warteräume für die K-Order–Generatoren eines K-Akteurs werden im Rahmen der Erzeugung des K-Akteurs von dem für diesen erzeugten Akteursphärenverwalter bei Ausführung der Operation `Create[Prot]KActorDesc` (siehe Tabelle 6.1) angelegt. In den Warteräumen werden die bei Ausführung von K-Order–Aufrufen bzgl. des K-Akteurs erzeugten K-Order–Beschreibungen solange gespeichert, bis der K-Akteur eine entsprechende Annahme–Anweisung ausführt und die Ausführung der kanonischen Operation der jeweiligen K-Order durch den K-Akteur begonnen werden kann.

Zur Realisierung von K-Order–Aufrufen und –Annahmen und dem damit verbundenen Einfügen bzw. Entnehmen von K-Order–Beschreibungen in die Warteräume bzw. aus den

[8] Dies kann insbesondere auch seine eigene kanonische Operation sein. Ist dies der Fall gilt: $I = A$.

Warteräumen bieten die Akteursphärenverwalter die in Tabelle 6.3 angegebenen Operationen an.

Operation	**Funktionalität**	**Aufruf durch**
`CreateKOrder`	K-Order–Aufruf bzgl. eines nicht zugriffskontrollierten K-Akteurs	Akteur
`CreateProtKOrder`	K-Order–Aufruf bzgl. eines zugriffskontrollierten K-Akteurs	Akteur
`InsertKOrder`	Einfügen einer K-Order Beschreibung in einen Warteraum	ASV
`AcceptKOrder`	Entnehmen einer K-Order Beschreibung aus einem Warteraum	Akteur
`SignalKOrderTerm`	Melden der Terminierung einer K-Order	ASV

Tabelle 6.3: Schnittstellenoperationen der Akteursphärenverwalter zur Realisierung von K-Order–Aufrufen und –Annahmen

Ein Akteur A, der einen K-Order–Aufruf bzgl. eines K-Akteurs K ausführt, ruft dazu in Abhängigkeit davon, ob K zugriffskontrolliert ist oder nicht, entweder die Operation `CreateProtKOrder` oder die Operation `CreateKOrder` auf seinem Akteursphärenverwalter $ASV(A)$ auf. Die Ausführung von `CreateProtKOrder` bewirkt zunächst, daß der für den entsprechenden K-Order–Generator festgelegte Zugriffsrestriktionsausdruck durch Aufruf einer Operation auf dem dem K-Akteur K zugeordneten Akteursphärenverwalter $ASV(K)$ ausgewertet wird (siehe hierzu Abschnitt 6.2.2.5). Ist das Ergebnis dieser Auswertung positiv, so wird analog zur Ausführung von `CreateKOrder` die Operation `InsertKOrder` auf dem Akteursphärenverwalter $ASV(K)$ aufgerufen. Deren Ausführung bewirkt, daß eine Beschreibung des K-Order–Aufrufs in den von $ASV(K)$ für den entspechenden K-Order–Generator verwalteten Warteraum eingefügt wird. Die Operationen `CreateKOrder` bzw. `CreateProtKOrder` blockieren den aufrufenden Akteur A solange, bis die erzeugte K-Order ausgeführt und deren Terminierung durch Aufruf der Operation `SignalKOrderTerm` auf dem Akteursphärenverwalter $ASV(A)$ gemeldet wurde. `SignalKOrderTerm` wird dabei von dem Akteursphärenverwalter $ASV(K)$ des K-Akteurs K aufgerufen und zwar im Rahmen der Ausführung der Operation `Terminate` (siehe Tabelle 6.2), die der K-Akteur K am Ende der K-Order–Ausführung auf $ASV(K)$ aufruft.

Die Ausführung einer K-Order setzt deren Annahme durch den jeweiligen K-Akteur voraus. Zur Annahme einer K-Order ruft der K-Akteur K die Operation `AcceptKOrder` auf seinem Akteursphärenverwalter $ASV(K)$ auf. Die Ausführung dieser Operation bewirkt, daß aus dem Warteraum des entsprechenden K-Order–Generators eine K-Order–Beschreibung gemäß FIFO[9]–Semantik entnommen wird, sofern dieser überhaupt eine K-Order–Beschreibung enthält. Anderenfalls wird der aufrufende K-Akteur solange blockiert, bis eine K-Order–Beschreibung durch Ausführung der Operation `InsertKOrder` in den Warteraum eingefügt wird. Nach Entnahme einer K-Order–Beschreibung aus dem Warteraum wird diese geeignet ergänzt und in den Lebenszeitkeller von $ASV(K)$ eingetragen. Anschließend wird die kanonische Operation der K-Order durch den K-Akteur K, d.h. durch den virtuellen Prozessor, mit dem dieser realisiert ist, ausgeführt. Am Ende der K-Order–Ausführung ruft der K-Akteur K die Operation `Terminate` auf seinem Akteursphärenverwalter $ASV(K)$ auf, in deren Kontext

[9] *First In First Out*

– wie oben beschrieben – die Operation `SignalKOrderTerm` auf dem Akteursphärenverwalter $ASV(A)$ des Akteurs A aufgerufen wird, der den entsprechenden K-Order–Aufruf ausführt.

6.2.2.4 Verwaltung von Zugriffskontrollinformationen

Wie in Abschnitt 6.2.1 erläutert, sind die Akteursphärenverwalter neben den bisher beschriebenen Aufgaben für die zur Durchsetzung des Rechts eines INSEL$^+$–Systems durchzuführenden Zugriffskontrollen und für die sichere Verwaltung der für diese Kontrollen benötigten Informationen zuständig. Im folgenden werden zunächst die Datenstrukturen und Operationen angegeben, die die Akteursphärenverwalter für die Verwaltung der benötigten Zugriffskontrollinformationen bereitstellen. In Abschnitt 6.2.2.5 wird dann beschrieben, wie die Akteursphärenverwalter auf Basis der in diesen Datenstrukturen repräsentierten Informationen die Zugriffskontrollen durchführen.

In Abschnitt 6.1 wurde bereits angegeben, welche Informationen bzw. Datenstrukturen für die korrekte Auswertung von Zugriffsrestriktionsausdrücken, d.h. für die Durchführung von Zugriffskontrollen zu verwalten sind. Dazu gehören

- das Kontextattribut für Akteure,
- das Owner–Attribut für Komponenten,
- Zugriffskontrollisten zur Realisierung des `IN_ACL`–Prädikats und
- Zugriffshistorienlisten zur Realisierung des `ACCESSED`–Prädikats.

Im weiteren werden diese Datenstrukturen und ihre Verwaltung durch die Akteursphärenverwalter im Einzelnen erklärt.

Kontextattribut für Akteure

Ein Akteursphärenverwalter $ASV(A)$ verwaltet für den Akteur A, dessen Sphäre er zugeordnet ist, eine Datenstruktur, in dem die in Definition (4.11) für das Kontextattribut eines Akteurs festgelegten Informationen enthalten sind. Diese Datenstruktur enthält:

(a) den Identifikator des Benutzers, in dessen Auftrag der Akteur die kanonische Operation seiner jeweiligen Ausführungskomponente ausführt;

(b) den Identifikator der Rolle, in der dieser Benutzer agiert;

(c) den Identifikator des Akteurs;

(d) den Identifikator des Generators, bzgl. dem der Akteur inkarniert wurde;

(e) den Identifikator der Ausführungskomponente des Akteurs und

(f) den Identifikator des Generators, bzgl. dem die Ausführungskomponente inkarniert wurde.

Diese Datenstruktur, die im weiteren als **Kontext–Datenstruktur** bezeichnet wird, wird als Bestandteil der Komponentenbeschreibung des Akteurs A verwaltet. Sie wird mit Erzeugung der Komponentenbeschreibung des Akteurs bei Ausführung der Operation `Create[Prot]MActorDesc` bzw. `Create[Prot]KActorDesc` (vgl. Tabelle 6.1) durch den für

den Akteur A erzeugten Akteursphärenverwalter $ASV(A)$ angelegt und initialisiert. Dazu sind bei Aufruf dieser Operationen der Identifikator des dem Akteur A zugeordneten Benutzers (vgl. Definition (4.1)), der Identifikator der ihm zugeordneten Rolle (vgl. Definition (4.3)) sowie der Identifikator des Akteur–Generators, bzgl. dem A zu inkarnieren ist, an den Akteursphärenverwalter $ASV(A)$ zu übergeben.

Handelt es sich bei dem Akteur A nicht um einen Benutzerrepräsentanten, so werden der Benutzeridentifikator und der Rollenidentifikator mit dem Benutzeridentifikator und dem Rollenidentifikator des Akteurs initialisiert, der den Akteur A erzeugt. Wenn A ein Benutzerrepräsentant ist, dann ist der A erzeugende Akteur ein Login–Akteur. In diesem Fall wird der Benutzeridentifikator in der Kontext–Datenstruktur von A mit dem Identifikator des Benutzers initialisiert, für den A erzeugt wird. Der Rollenidentifikator wird mit dem Identifikator der Rolle belegt, in der der Benutzer für die jeweilige Sitzung (*session*) in dem System agiert. Der Benutzeridentifikator und der Rollenidentifikator des Akteurs, der der Hauptkomponente des Systems entspricht, werden mit dem Identifikator des für alle INSEL$^+$–Systeme vordefinierten Benutzers *System* und dem Identifikator der vordefinierten Rolle *SystemRole* initialisiert. Für einen M-Akteur sind der Benutzeridentifikator und der Rollenidentifikator für seine Lebenszeit konstant. Bei einem K-Akteur können sich diese Identifikatoren für die Ausführungsdauern von K–Order ändern. Dies wird weiter unten noch genauer erklärt.

Bei der Erzeugung der Kontext–Datenstruktur eines Akteurs A werden die in der obigen Liste unter (c) und (e) angegebenen Komponenten der Datenstruktur mit dem Identifikator des Akteurs sowie die Elemente (d) und (f) mit dem Identifikator des Generators, bzgl. dem der Akteur inkarniert wird, initialisiert. (c) und (d) bleiben für die Lebenszeit des Akteurs A konstant. (e) und (f) werden mit jeder S-Order– und Depot–Inkarnation, die der Akteur A erzeugt, aktualisiert. Diese Aktualisierung erfolgt im Rahmen der Ausführung der Schnittstellenoperation, die der Akteur A zur Erzeugung einer S-Order– bzw. Depot–Inkarnation auf seinem Akteursphärenverwalter $ASV(A)$ aufruft, also bei Ausführung von `Create[Prot]SOrder` bzw. `Create[Prot]Depot`.

Bei der Annahme einer K-Order durch einen K-Akteur K sind gemäß der in Definition (4.11) für das Kontextattribut eines Akteurs getroffenen Festlegungen neben den Elementen (e) und (f) der Benutzeridentifikator (a) und der Rollenidentifikator (b) der Kontext–Datenstruktur von K zu aktualisieren. (a) bzw. (b) muß dabei der Benutzeridentifikator bzw. der Rollenidentifikator des Akteurs zugewiesen werden, der die entsprechende K-Order erzeugt hat, d.h. der den jeweiligen K-Order–Aufruf ausgeführt hat. Diese Aktualisierung der Kontext–Datenstruktur von K erfolgt im Rahmen der Ausführung der Operation `AcceptKOrder`, die der K-Akteur K zur Annahme einer K-Order auf seinem Akteursphärenverwalter $ASV(K)$ aufruft. Um diese Aktualisierung korrekt durchführen zu können, müssen die in den von $ASV(K)$ verwalteten Warteräumen gespeicherten K-Order–Beschreibungen den Benutzeridentifikator und den Rollenidentifikator des Akteurs enthalten, der die jeweilige K-Order erzeugt hat. Dazu werden diese Identifikatoren von dem Akteursphärenverwalter des erzeugenden Akteurs bei Aufruf der Operation `InsertKOrder` auf $ASV(K)$ als Parameter an $ASV(K)$ übergeben.

Owner–Attribut für Komponenten

Jeder Komponente x eines INSEL$^+$–Systems ist mit dem Attribut $owner(x)$ ein Benutzer als Besitzer fest zugeordnet. Gemäß Definition (4.2) gibt $owner(x)$ den Identifikator des Benutzers an, in dessen Auftrag die Komponente x erzeugt wurde. Das Owner–Attribut einer Komponente wird durch einen Eintrag in der Komponentenbeschreibung der Komponente realisiert. Dieser Eintrag, der als **Owner–Eintrag** bezeichnet wird, wird bei Erzeugung der

Komponentenbeschreibung initialisiert und bleibt für die Lebenszeit der Komponente konstant. Um den Owner–Eintrag korrekt initialisieren zu können, ist bei Aufruf der in Tabelle 6.1 angegebenen Operationen, deren Bezeichner mit `Desc` endet, der Benutzeridentifikator des Akteurs, der die Komponente erzeugt, als Parameter an den Akteursphärenverwalter, der die Komponentenbeschreibung der zu erzeugenden Komponente anlegt, zu übergeben. Dieser Benutzeridentifikator ist Bestandteil der Kontext–Datenstruktur, die von dem Akteursphärenverwalter des erzeugenden Akteurs verwaltet wird. Ein Ausnahmefall liegt lediglich bei der Erzeugung eines Benutzerrepräsentanten vor. In diesem Fall wird der Owner–Eintrag in der Komponentenbeschreibung des Benutzerrepräsentanten nicht mit dem Benutzeridentifikator des erzeugenden Akteurs initialisiert, sondern analog zu dem Benutzeridentifikator in der Kontext–Datenstruktur des Benutzerrepräsentanten mit dem Identifikator des Benutzers, für den der Benutzerrepräsentant erzeugt wird.

Zugriffskontrollisten

Gemäß Abschnitt 4.4.3 ist in INSEL$^+$ für jeden explizit zugriffskontrollierten DA–Generator, jedes zugriffskontrollierte Depot sowie jeden zugriffskontrollierten K-Akteur implizit eine Zugriffskontrolliste definiert. Diese Zugriffskontrollisten werden von den Akteursphärenverwaltern durch geeignete Datenstrukturen realisiert und verwaltet. Die Zugriffskontrolliste einer zugriffskontrollierten Komponente Z wird von dem Akteursphärenverwalter $ASV(Z)$ verwaltet, dessen Sphäre die Komponente Z angehört, der also in seinem Lebenszeitkeller die Komponentenbeschreibung von Z enthält. Die Zugriffskontrolliste der Komponente Z wird als Bestandteil der Komponentenbeschreibung von Z verwaltet, was bedeutet, daß die Komponentenbeschreibung von Z einen Verweis auf die Zugriffskontrolliste von Z enthält. Diese wird mit Erzeugung der Komponentenbeschreibung von Z bei Ausführung der Operation `CreateProtGenDesc`, `CreateProtDepotDesc` bzw. `CreateProtKAkteurDesc` von dem Akteursphärenverwalter $ASV(Z)$ angelegt und gemäß der Angaben im ACL–Part des jeweiligen Generators initialisiert. Auf die konkrete Datenstruktur zur Repräsentation einer Zugriffskontrolliste und deren Initialisierung wird an dieser Stelle nicht weiter eingegangen. Für das Weitere sei lediglich davon ausgegangen, daß die laut Abschnitt 4.4.3 in einer Zugriffskontrolliste enthaltenen Informationen in dieser Datenstruktur geeignet repräsentiert sind.

Die Zugriffskontrolliste einer zugriffskontrollierten Komponente ist immer dann auszuwerten, wenn ein `IN_ACL`–Prädikat (siehe Abschnitt 4.4.4.1) auszuwerten ist, in dem diese zugriffskontrollierte Komponente über den mit dem Parameter <*comp-identifier*> des Prädikats festgelegten Komponentenidentifikator identifiziert wird. Zur Änderung von Einträgen in der Zugriffskontrolliste ist auf jedem explizit zugriffskontrollierten DA–Generator sowie jedem zugriffskontrollierten Depot bzw. K-Akteur implizit die äußere Operation *change_acl* definiert. Weiterhin ist für jede derartige zugriffskontrollierte Komponente die äußere Operation *list_acl* zur Ausgabe des aktuellen Inhalts der Zugriffskontrolliste implizit definiert. Die Akteursphärenverwalter stellen zur Auswertung von Zugriffskontrollisten und zur Realisierung der Operationen *change_acl* und *list_acl* die in Tabelle 6.4 angegebenen Schnittstellenoperationen zur Verfügung.

Die Operation `CheckACL` dient zur Auswertung eines `IN_ACL`–Prädikats. Sie wird von einem Akteursphärenverwalter, der einen Zugriffsrestriktionsausdruck auszuwerten hat, aufgerufen, wenn dieser ein `IN_ACL`–Prädikat enthält, in dem eine zugriffskontrollierte Komponente identifiziert wird, deren Komponentenbeschreibung und damit deren Zugriffskontrolliste dieser Verwalter nicht selbst verwaltet. Wird die Komponentenbeschreibung dieser zugriffskontrollierten Komponente von dem Akteursphärenverwalter selbst verwaltet, so kann er das

Operation	**Funktionalität**	**Aufruf durch**
`CheckACL`	Auswertung einer Zugriffskontrolliste	ASV
`ChangeACLCall`	Änderung einer Zugriffskontrolliste	Akteur
`ChangeACLEntry`		ASV
`ListACLCall`	Ausgabe einer Zugriffskontrolliste	Akteur
`ListACL`		ASV

Tabelle 6.4: Schnittstellenoperationen der Akteursphärenverwalter zur Auswertung von Zugriffskontrollisten und zur Realisierung der Operationen *change_acl* und *list_acl*

`IN_ACL`–Prädikat vollständig autark auswerten. In diesem Fall ist ein `CheckACL`–Aufruf nicht erforderlich. Anderenfalls wird von ihm `CheckACL` auf dem Akteursphärenverwalter aufgerufen, der die Komponentenbeschreibung der zugriffskontrollierten Komponente verwaltet. Als aktuelle Parameter des `CheckACL`–Aufrufs sind der mit dem `IN_ACL`–Prädikat festgelegte Benutzeridentifikator, Rollenidentifikator, Komponentenidentifikator sowie der Operationsidentifikator zu übergeben. Die Ausführung der Operation `CheckACL` bewirkt dann, daß die Zugriffskontrolliste der mit dem Komponentenidentifikator identifizierten Komponente gemäß der in Abschnitt 4.4.4.1 angegebenen Semantik des `IN_ACL`–Prädikats ausgewertet wird. Das Ergebnis dieser Auswertung wird als Rückgabewert bei Terminierung von `CheckACL` an den aufrufenden Akteursphärenverwalter übergeben.

Die von den Akteursphärenverwaltern zur Verfügung gestellten Operationen `ChangeACLCall` und `ChangeACLEntry` dienen zur Realisierung von *change_acl*–Aufrufen[10] auf zugriffskontrollierten Komponenten. Ein Akteur A, der einen *change_acl*–Aufruf auf einer zugriffskontrollierten Komponente Z ausführen möchte, ruft dazu die Operation `ChangeACLCall` auf seinem Akteursphärenverwalter $ASV(A)$ auf. Dabei werden insbesondere die aktuellen Parameter des Aufrufs und damit insbesondere auch der Identifikator der zugriffskontrollierten Komponente Z an $ASV(A)$ übergeben. Wird die Komponentenbeschreibung von Z von $ASV(A)$ selbst verwaltet, so wird von $ASV(A)$ zunächst der für die jeweilige *change_acl*–Operation festgelegte Zugriffsrestriktionsausdruck ausgewertet, d.h. es wird überprüft, ob der aufrufende Akteur A das Recht zur Änderung der Zugriffskontrolliste von Z hat. Ist dies der Fall, so wird die Zugriffskontrolliste von Z entsprechend der in Abschnitt 4.4.3 erklärten Semantik der Operation *change_acl* geändert. Wird die Komponentenbeschreibung von Z nicht von $ASV(A)$ verwaltet, ruft $ASV(A)$ die Operation `ChangeACLEntry` auf dem Akteursphärenverwalter auf, der die Komponentenbeschreibung von Z und damit auch deren Zugriffskontrolliste verwaltet. Dabei werden die aktuellen Parameter des *change_acl*–Aufrufs und das Kontextattribut von A an diesen Akteursphärenverwalter weitergegeben. Die Ausführung von `ChangeACLEntry` bewirkt dann zunächst die Auswertung des entsprechenden Zugriffsrestriktionsausdrucks und anschließend abhängig von dem Ergebnis dieser Auswertung die Änderung der Zugriffskontrolliste von Z entsprechend der Semantik der *change_acl*–Operation.

Analog hierzu werden mit den Operationen `ListACLCall` und `ListACL` der Akteursphärenverwalter Aufrufe der Operation *list_acl* realisiert.

Die bei Ausführung der Operationen `CheckACL`, `ChangeACLCall` und `ChangeACLEntry` sowie `ListACLCall` und `ListACL` auszuführenden Lese– bzw. Schreibeoperationen auf einer

[10] Zur Erinnerung: In INSEL$^+$ steht die Operation *change_acl* für einen explizit zugriffskontrollierten DA–Generator unter dem Bezeichner `ChangeGenACL` und für ein zugriffskontrolliertes Depot bzw. einen zugriffskontrollierten K-Akteur unter dem Bezeichner `ChangeACL` zur Verfügung.

Zugriffskontrolliste sind gemäß der Leser–Schreiber–Semantik zu synchronisieren. Diese Synchronisation erfolgt durch den Erwerb und die Freigabe sogenannter Lese– bzw. Schreib–Sperren, die im folgenden Abschnitt 6.2.2.5 eingeführt werden.

Zugriffshistorienlisten

Zur Realisierung des in Abschnitt 4.4.4.2 definierten `ACCESSED`–Prädikates sind Informationen darüber zu verwalten, welcher Benutzer bereits auf welche Komponenten mit welche Operationen zugegriffen hat. Wie bereits im einleitenden Abschnitt 6.1 dieses Kapitels kurz erläutert, bestehen für die Repräsentation und Verwaltung dieser Informationen mehrere Realisierungsalternativen.

Eine dieser Alternativen besteht darin, für jeden Benutzer des Systems eine Datenstruktur zu verwalten, in der festgehalten wird, auf welchen Komponenten der Benutzer welche Operationen aufgerufen hat. Dieser Ansatz hat jedoch einige offensichtliche Nachteile. Zum einen ist unklar, welcher Akteursphärenverwalter diese Datenstrukturen verwalten soll. Es bietet sich zwar an, diese Datenstrukturen durch den Akteursphärenverwalter des Benutzerdatenverwalters (vgl. Abschnitt 4.2.2) verwalten zu lassen. Diese *zentrale* Lösung ist jedoch aufgrund der Gefahr vom Engpässen nicht akzeptabel. Zum anderen sind zur Aktualisierung dieser Datenstrukturen zusätzliche Verwalteraufrufe notwendig, die sich bei anderen Realisierungsalternativen teilweise einsparen lassen. Einerseits ist bei jedem Zugriff eines Akteurs auf eine Komponente ein zusätzlicher Verwalteraufruf zur Aktualisierung der Datenstruktur des entsprechenden Benutzers durchzuführen, und zwar auf dem Akteursphärenverwalter, der diese Datenstruktur verwaltet. Andererseits sind bei Auflösung einer Komponente die Datenstrukturen aller Benutzer zu aktualisieren, um die darin gespeicherten Informationen von Zugriffen der Benutzer auf die aufzulösende Komponente zu löschen, was wiederum mindestens einen Verwalteraufruf erfordert. Solche zusätzlichen Verwalteraufrufe sollten aus Effizienzgründen jedoch nach Möglichkeit vermieden werden.

Aufgrund der genannten Nachteile wird in dieser Arbeit ein anderer Ansatz verfolgt. Es wird für jede Komponente, die potentiell in einem `ACCESSED`–Prädikat identifiziert wird, eine Datenstruktur verwaltet, die als **Zugriffshistorienliste** bezeichnet wird, und in der festgehalten wird, welche Benutzer bereits welche Operationen auf dieser Komponente aufgerufen haben. Ob eine Komponente potentiell in einem `ACCESSED`–Prädikat identifiziert werden kann, wird von dem INSEL$^+$-Übersetzer im Rahmen einer statischen Analyse auf Basis der in den `ACCESSED`–Prädikaten des entsprechenden INSEL$^+$–Programms für den Parameter <*comp-identifier*> angegebenen Komponentenbezeichnern ermittelt.

Die Zugriffshistorienliste einer potentiell in einem `ACCESSED`–Prädikat identifizierten Komponente Z wird als Bestandteil der Komponentenbeschreibung von Z verwaltet, was bedeutet, daß sie von dem Akteursphärenverwalter $ASV(Z)$ verwaltet wird, der in seinem Lebenszeitkeller die Komponentenbeschreibung von Z enthält. Dazu wird bei Erzeugung der Komponentenbeschreibung von Z ein entsprechender Verweis auf die Zugriffshistorienliste in die Komponentenbeschreibung eingetragen. Analog zur Erklärung der Zugriffskontrollisten wird auch an dieser Stelle nicht näher auf die konkrete Datenstruktur zur Repräsentation einer Zugriffshistorienliste eingegangen.

Die Zugriffshistorienliste einer Komponente ist immer dann zu aktualisieren, wenn ein Akteur eine Operation auf dieser Komponente aufruft[11]. Sie ist auszuwerten, wenn ein `ACCESSED`–

[11] Dies kann auf Basis statisch und ggf. zusätzlich dynamisch gewonnener Analyseinformationen dahingehend optimiert werden, daß die Zugriffshistorienliste nur dann zu aktualisieren ist, wenn der Akteur die jeweilige Operation erstmalig aufruft.

Prädikat auszuwerten ist, in dem diese Komponente über den mit dem Parameter <*comp-identifier*> des Prädikats festgelegten Komponentenidentifikator identifiziert wird. Zur Aktualisierung und Auswertung von Zugriffshistorienlisten bieten die Akteursphärenverwalter die in Tabelle 6.5 angegebenen Operationen an.

Operation	**Funktionalität**	**Aufruf durch**
`CheckAHL`	Auswertung einer Zugriffshistorienliste	ASV
`ChangeAHL`	Änderung einer Zugriffshistorienliste	ASV

Tabelle 6.5: Schnittstellenoperationen der Akteursphärenverwalter zur Auswertung und Änderung von Zugriffshistorienlisten

Die Operation `CheckAHL` dient zur Auswertung eines `ACCESSED`–Prädikats. Sie wird von einem Akteursphärenverwalter aufgerufen, der einen Zugriffsrestriktionsausdruck auszuwerten hat, wenn dieser ein `ACCESSED`–Prädikat enthält, in dem eine Komponente identifiziert wird, deren Komponentenbeschreibung und damit deren Zugriffshistorienliste dieser Verwalter nicht selbst verwaltet. In dem Fall, daß die Zugriffshistorienliste von dem Verwalter selbst verwaltet wird, ist ein `CheckAHL`–Aufruf nicht erforderlich, da das `ACCESSED`–Prädikat dann eigenständig von dem Verwalter auswertbar ist. Im anderen Fall wird von ihm `CheckAHL` auf dem Akteursphärenverwalter aufgerufen, der die Komponentenbeschreibung der Komponente verwaltet. Als aktuelle Parameter des `CheckAHL`–Aufrufs sind der mit dem `ACCESSED`–Prädikat festgelegte Benutzeridentifikator, der Komponentenidentifikator sowie der Operationsidentifikator zu übergeben. Die Ausführung der Operation `CheckAHL` bewirkt dann, daß die Zugriffshistorienliste der mit dem Komponentenidentifikator identifizierten Komponente gemäß der in Abschnitt 4.4.4.2 angegebenen Semantik des `ACCESSED`–Prädikats ausgewertet wird. Das Ergebnis dieser Auswertung wird als Rückgabewert bei Terminierung von `CheckAHL` an den aufrufenden Akteursphärenverwalter übergeben.

Die Zugriffshistorienliste eines DA–Generators, der potentiell in einem `ACCESSED`–Prädikat identifiziert wird, ist dann zu aktualisieren, wenn ein Akteur eine Inkarnation bzgl. dieses Generators erzeugt. Diese Aktualisierung erfolgt durch Ausführung der Operation `ChangeAHL`, die von dem Akteursphärenverwalter des erzeugenden Akteurs bei Ausführung einer der Operationen `Create[Prot]SOrder`, `Create[Prot]Depot`, `Create[Prot]MActor` oder `Create[Prot]KActor` auf dem Akteursphärenverwalter aufgerufen wird, der die Komponentenbeschreibung des Generators und damit auch dessen Zugriffshistorienliste verwaltet. Dabei werden der Identifikator des Generators und der Benutzeridentifikator aus der Kontext–Datenstruktur des erzeugenden Akteurs als Parameter des `ChangeAHL`–Aufrufs an diesen Akteursphärenverwalter übergeben. Die Ausführung von `ChangeAHL` bewirkt dann, daß in der Zugriffshistorienliste des Generators ein Eintrag bestehend aus dem übergebenen Benutzeridentifikator und dem Identifikator der Operation *erzeuge* hinzugefügt wird. Bei Aufruf einer Zugriffsoperation eines Depots, das potentiell in einem `ACCESSED`–Prädikat identifiziert wird, erfolgt der für die Aktualisierung der Zugriffshistorienliste notwendige `ChangeAHL`–Aufruf auf dem Akteursphärenverwalter, der die Komponentenbeschreibung des Depots verwaltet, im Rahmen der Ausführung der Operation `Create[Prot]DepotOp` durch den Akteursphärenverwalter des aufrufenden Akteurs. Die bei Aufruf einer Kommunikationsoperation eines K-Akteurs, der potentiell in einem `ACCESSED`–Prädikat identifiziert wird, erforderliche Aktualisierung seiner Zugriffshistorienliste erfolgt im Rahmen der Ausführung der Operation `InsertKOrder` durch den Akteursphärenverwalter des K-Akteurs. Ein `ChangeAHL`–Aufruf ist für diese Aktualisierung nicht erforderlich.

Für die Synchronisation der bei Ausführung von `CheckAHL` durchzuführenden Lesezugriffe und der bei Ausführung von `ChangeAHL` durchzuführenden Schreibzugriffe auf einer Zugriffshistorienliste gilt das im Abschnitt über die Zugriffskontrollisten Gesagte entsprechend.

Die Vorteile des erklärten Ansatzes der komponentenspezifischen Verwaltung von Zugriffshistorienlisten gegenüber der zu Beginn dieses Abschnitts erläuterten benutzerbezogenen Alternative lassen sich wie folgt zusammenfassen. Zum einen ergibt sich eine klare Zuordnung zwischen Akteursphärenverwaltern und den Zugriffshistorienlisten, die sie zu repräsentieren und zu verwalten haben. Dadurch, daß ein Akteursphärenverwalter lediglich die Zugriffshistorienlisten der Komponenten verwaltet, die der von ihm verwalteten Akteursphäre angehören, werden die für die Realisierung des `ACCESSED`–Prädikats benötigten Datenstrukturen *dezentral* verwaltet. Zum anderen kann mit der Auflösung einer Komponente auch die ggf. für diese Komponente verwaltete Zugriffshistorienliste gelöscht werden. Dies hat insbesondere den Vorteil, daß zum Löschen einer Zugriffshistorienliste kein gesonderter Verwalteraufruf erforderlich ist. Desweiteren ist nicht generell bei Zugriff eines Akteurs auf eine Komponente, für die eine Zugriffshistorienliste verwaltet wird, ein zusätzlicher Verwalteraufruf zur Aktualisierung der Zugriffshistorienliste der Komponente notwendig. So kann die Aktualisierung der Zugriffshistorienliste durch den Akteursphärenverwalter des zugreifenden Akteurs bei Ausführung der Operation erfolgen, die der Akteur zur Realisierung des Zugriffs auf dem Verwalter aufruft, falls die Komponente, auf die zugegriffen wird, der Akteursphäre des Akteurs angehört. Bei Zugriffen auf zugriffskontrollierte Komponenten kann die ggf. notwendige Aktualisierung ihrer Zugriffshistorienliste im Rahmen der Ausführung der im folgenden Abschnitt näher erklärten Operation zur Auswertung des jeweiligen Zugriffsrestriktionsausdrucks erfolgen. Insgesamt ergibt sich, daß bei dem Ansatz der komponentenspezifischen Verwaltung der Zugriffshistorienlisten im allgemeinen deutlich weniger Verwalteraufrufe zur Aktualisierung der Datenstrukturen benötigt werden als bei der benutzerbezogenen Realisierungsalternative.

6.2.2.5 Durchführung von Zugriffskontrollen

In diesem Abschnitt wird erklärt, wie die Akteursphärenverwalter auf Basis der im vorhergehenden Abschnitt angegebenen Datenstrukturen und Operationen die für die Durchsetzung des Rechts eines INSEL$^+$–Systems erforderlichen Zugriffskontrollen durchführen.

Wie bereits in Abschnitt 6.1 erläutert, ist der Zugriff auf eine zugriffskontrollierte Komponente immer mit der Erzeugung einer DA–Inkarnation verbunden, wobei die Gültigkeit des für die *erzeuge*–Operation des Generators festgelegten Zugriffsrestriktionsausdrucks Vorbedingung für die Erzeugung der Inkarnation ist. Die Auswertung des Zugriffsrestriktionsausdrucks ist also vor der „eigentlichen" Erzeugung der Inkarnation. d.h. vor entsprechender Allokation von Speicher und ggf. eines Prozessors zur Realisierung der Inkarnation, durchzuführen. Diesen Überlegungen entsprechend sind die Maßnahmen zur Durchführung der Zugriffskontrollen in die generellen Maßnahmen zur Erzeugung von DA–Inkarnationen eingeordnet.

Zur Erzeugung von Inkarnationen bzgl. eines zugriffskontrollierten DA–Generators stellen die Akteursphärenverwalter spezielle Schnittstellenoperationen zur Verfügung, die bereits in Tabelle 6.1 angegeben wurden. Die Operationen `CreateProtSOrder`, `CreateProtDepot`, `CreateProtMActor` sowie `CreateProtKActor` werden zur Realisierung des Aufrufs der *erzeuge*–Operation auf einem entsprechenden explizit zugriffskontrollierten DA–Generator von dem den Aufruf ausführenden Akteur auf seinem Akteursphärenverwalter aufgerufen. Zur Realisierung des Aufrufs einer Zugriffsoperation eines zugriffskontrollierten Depots bzw. eines Aufrufs einer Kommunikationsoperation eines zugriffskontrollierten K-Akteurs ruft ein Akteur die Operation `CreateProtDepotOp` bzw. `CreateProtKOrder` auf seinem Akteursphären-

verwalter auf. Die allen diesen Operationen gemeinsame Funktionalität besteht darin, daß zunächst die Operation `CheckAccess` (siehe Tabelle 6.6) auf dem Akteursphärenverwalter aufgerufen wird, der die Komponentenbeschreibung des explizit zugriffskontrollierten DA–Generators bzw. des zugriffskontrollierten Depots oder K-Akteurs verwaltet. Die Ausführung von `CheckAccess` bewirkt, daß der für die *erzeuge*–Operation bzw. die Zugriffsoperation oder Kommunikationsoperation festgelegte Zugriffsrestriktionsausdruck ausgewertet wird und das Ergebnis dieser Auswertung als Rückgabewert bei Terminierung von `CheckAccess` an den aufrufenden Akteursphärenverwalter übergeben wird.

Operation	**Funktionalität**	**Aufruf durch**
`CheckAccess`	Auswertung eines Zugriffsrestriktionsausdrucks	ASV

Tabelle 6.6: Schnittstellenoperation der Akteursphärenverwalter zur Auswertung eines Zugriffsrestriktionsausdrucks

Ist das Ergebnis dieser Auswertung positiv, so werden die in Abschnitt 6.2.2.1 erklärten Maßnahmen zur Erzeugung der jeweiligen Inkarnation durchgeführt. Anderenfalls hat der aufrufende Akteur nicht das Recht zur Ausführung der jeweiligen Operation, was bedeutet, daß keine neue Inkarnation erzeugt wird und die in dem INSEL$^+$–Programm im Status–Part des entsprechenden Erzeugungskonstrukts angegebene boolesche Variable (vgl. Abschnitt 4.5) mit dem Wert *false* belegt wird. Wird eine Inkarnation erzeugt, so erhält diese Variable den Wert *true*.

Damit der bei Aufruf einer Operation OP einer zugriffskontrollierten Komponente Z durch Ausführung von `CheckAccess` auszuwertende Zugriffsrestriktionsausdruck vollständig und korrekt ausgewertet werden kann, sind folgende Informationen bei Aufruf von `CheckAccess` als Parameter an den jeweiligen Akteursphärenverwalter zu übergeben:

- der Identifikator der zugriffskontrollierten Komponente Z, für die der Zugriffsrestriktionsausdruck auszuwerten ist;
- der Bezeichner der Operation OP, deren Zugriffsrestriktionsausdruck auszuwerten ist;
- die aktuellen Parameter des OP–Aufrufs, da der Wert des Zugriffsrestriktionsausdrucks von den Werten dieser Parameter abhängig sein kann;
- die Werte der Kontext–Datenstruktur des die Operation OP aufrufenden Akteurs.

Die Auswertung eines Zugriffsrestriktionsausdrucks und damit die Ausführung der Operation `CheckAccess` kann aufgrund der Möglichkeiten für die Konstruktion von Zugriffsrestriktionsausdrücken (siehe hierzu Abschnitt 4.4.4) im allgemeinen recht komplex sein. Im Rahmen der Ausführung von `CheckAccess` können insbesondere Operationen anderer Akteursphärenverwalter aufgerufen werden, um in dem Zugriffsrestriktionsausdruck auftretende `IN_ACL`– und `ACCESSED`–Prädikate auszuwerten. Ist in dem Zugriffsrestriktionsausdruck ein `IN_ACL`–Prädikat enthalten, in dem eine zugriffskontrollierte Komponente identifiziert wird, deren Zugriffskontrolliste nicht von dem `CheckAccess` ausführenden Akteursphärenverwalter verwaltet wird, so ruft dieser zur Auswertung des `IN_ACL`–Prädikats die Operation `CheckACL` auf dem Akteursphärenverwalter auf, der die Komponentenbeschreibung dieser Komponente verwaltet. Analog wird zur Auswertung eines `ACCESSED`–Prädikats, in dem eine Komponente identifiziert wird, deren Zugriffshistorienliste nicht von dem `CheckAccess` ausführenden

Akteursphärenverwalter verwaltet wird, die Operation `CheckAHL` auf dem Akteursphärenverwalter aufgerufen, dessen Sphäre diese Komponente angehört.

Atomarität der Auswertung von Zugriffsrestriktionsausdrücken

In Abschnitt 4.4.4 wurde als Anforderung an die Realisierung gestellt, daß die Auswertung eines Zugriffsrestriktionsausdrucks in dem Sinne atomar sein muß, daß sich nach Auswertung eines Teilausdrucks der Wert dieses Teilausdrucks während der gesamten „restlichen“ Auswertung des Zugriffsrestriktionsausdrucks nicht ändern darf. Diese Forderung besagt im einzelnen folgendes. Ist zur Auswertung eines Zugriffsrestriktionsausdrucks ein `IN_ACL`–Prädikat auszuwerten, in dem eine zugriffskontrollierte Komponente Z identifiziert wird, so darf sich der Wert der Zugriffskontrolliste von Z ab dem Zeitpunkt des Beginns der Auswertung des `IN_ACL`–Prädikats solange nicht ändern, bis die Auswertung des Zugriffsrestriktionsausdrucks abgeschlossen ist. Gleiches gilt für den Wert einer Zugriffshistorienliste einer Komponente, die in einem in dem Zugriffsrestriktionsausdruck enthaltenen `ACCESSED`–Prädikat identifiziert wird, ab dem Zeitpunkt der Auswertung des `ACCESSED`–Prädikats. Tritt in einem Zugriffsrestriktionsausdruck der Bezeichner einer lokalen oder globalen DE–Inkarnation auf, so darf sich der Wert dieser DE–Inkarnation ab dem Zeitpunkt der Auswertung des Teilausdrucks, in dem der Bezeichner dieser DE–Inkarnation enthalten ist, ebenfalls solange nicht ändern, bis die Auswertung des Zugriffsrestriktionsausdrucks abgeschlossen ist.

Zur Erfüllung dieser Anforderung verwalten die Akteursphärenverwalter sogenannte **Leser–Schreiber–Sperren** für die Objekte, auf die bei Auswertung von Zugriffsrestriktionsausdrücken zugegriffen wird und deren Wert sich während dieser Auswertungen potentiell verändern kann. Leser–Schreiber–Sperren (engl.: *Reader–Writer Locks*) dienen im allgemeinen dazu, die Zugriffe auf gemeinsam genutzte Objekte gemäß der Leser–Schreiber Semantik zu synchronisieren. Diese Semantik besagt informell, daß Leseoperationen auf dem Objekt parallel ausgeführt werden können, während Schreibeoperationen untereinander und gegenüber Leseoperationen wechselseitig ausgeschlossen auszuführen sind. Auf einer Leser–Schreiber–Sperre sind zwei Operationen definiert. Die Operation `Acquire` dient zum Erwerb einer Sperre, wobei als Parameter zu übergeben ist, ob eine Lese–Sperre oder eine Schreib–Sperre erworben werden soll. Mit der Operation `Release` kann eine erworbene Sperre wieder freigegeben werden.

Zu den Objekten, denen eine Leser–Schreiber–Sperre zu assoziieren ist, gehören die von den Akteursphärenverwaltern verwalteten Zugriffskontrollisten und Zugriffshistorienlisten sowie die DE–Inkarnationen, die Variablen sind und auf die bei Auswertung eines Zugriffsrestriktionsausdrucks potentiell zugegriffen wird. Die DE–Inkarnationen, auf die bei Auswertung von Zugriffsrestriktionsausdrücken potentiell zugegriffen wird, werden von dem INSEL$^+$–Übersetzer im Rahmen der statischen Analyse auf Basis der in den Zugriffsrestriktionsausdrücken des INSEL$^+$–Programms verwendeten Bezeichner für DE–Inkarnationen ermittelt. Die Leser–Schreiber–Sperren werden von den Akteursphärenverwaltern erzeugt und verwaltet, wobei die Leser–Schreiber–Sperre eines Objekts von dem Akteursphärenverwalter verwaltet wird, der auch das Objekt verwaltet. Die einer Zugriffskontrolliste bzw. Zugriffshistorienliste assoziierte Leser–Schreiber–Sperre wird mit Erzeugung der Zugriffskontrol– bzw. Zugriffshistorienliste angelegt und als Bestandteil der Komponentenbeschreibung der Komponente, der diese Zugriffskontrol– bzw. Zugriffshistorienliste zugeordnet ist, verwaltet. Die Leser–Schreiber–Sperre einer DE–Inkarnation D, auf die potentiell bei Auswertung eines Zugriffsrestriktionsausdrucks zugegriffen wird, wird mit Erzeugung der Komponentenbeschreibung von D bei Ausführung der Operation `CreateDEInc` bzw. `CreateDEIncDesc` durch den Akteursphärenverwalter angelegt, dessen Sphäre D zugeordnet ist.

Die Idee zur Durchsetzung der Forderung nach Atomarität der Auswertung eines Zugriffsrestriktionsausdrucks im oben erklärten Sinn besteht darin, bei Auswertung eines Zugriffsrestriktionsausdrucks jeweils vor Auswertung eines Teilausdrucks Lese–Sperren für die Objekte zu erwerben, auf die bei Auswertung des Teilausdrucks zugegriffen wird. Sind alle benötigten Lese–Sperren vorhanden, kann der entsprechende Teilausdruck ausgewertet werden. Die erworbenen Sperren werden jedoch nicht unmittelbar nach Auswertung des Teilausdrucks freigegeben, sondern erst am Ende der Auswertung des gesamten Zugriffsrestriktionsausdrucks. Wenn gewährleistet ist, daß vor Ausführung von Schreibeoperationen auf diesen Objekten jeweils eine Schreib–Sperre erworben werden muß, so ist mit diesem Verfahren sichergestellt, daß sich die Werte der Objekte, auf die bei Auswertung des Teilausdrucks zugegriffen wird, frühestens dann ändern können, wenn die Auswertung des Zugriffsrestriktionsausdrucks abgeschlossen ist. Das erklärte Verfahren ist mit dem aus dem Bereich der Realisierung von Transaktionen bekannten **Zwei–Phasen Locking** (siehe z.B. [Bac93]) vergleichbar, wobei die Funktionen zur Auswertung der Zugriffsrestriktionsausdrücke hier den Transaktionen entsprechen. Die im allgemeinen mit dem Zwei–Phasen Locking verbundene Gefahr des Auftretens von Deadlocks ist jedoch bei der Auswertung von Zugriffsrestriktionsausdrücken nach diesem Lock–Verfahren nicht vorhanden, da zur Auswertung eines Zugriffsrestriktionsausdrucks lediglich Lese–Sperren erworben werden müssen.

Zum Erwerben bzw. Freigeben von Sperren stellen die Akteursphärenverwalter die in Tabelle 6.7 angegebenen Schnittstellenoperationen zur Verfügung. Diese Operationen werden entweder von einem Akteur oder einem Akteursphärenverwalter aufgerufen.

Operation	**Funktionalität**	**Aufruf durch**
`AcquireLock`	Erwerb einer Lese– oder Schreib–Sperre	ASV und Akteur
`ReleaseLock`	Freigabe einer Lese– oder Schreib–Sperre	ASV und Akteur

Tabelle 6.7: Schnittstellenoperationen der Akteursphärenverwalter zum Erwerb und zur Freigabe von Leser–Schreiber–Sperren

Zum Erwerb einer Lese– oder Schreib–Sperre für ein Objekt ist die Operation `AcquireLock` auf dem Akteursphärenverwalter aufzurufen, der das Objekt und damit auch die dem Objekt assoziierte Leser–Schreiber–Sperre verwaltet. Bei Aufruf von `AcquireLock` auf einem Akteursphärenverwalter sind folgenden Informationen als Parameter zu übergeben:

- der Modus, d.h. ob eine Lese– oder Schreib–Sperre erworben werden soll;
- die Art des Objekts, für das eine Sperre erworben werden soll, also entweder eine DE–Inkarnation, Zugriffskontrolliste oder Zugriffshistorienliste;
- falls eine Sperre für eine DE–Inkarnation erworben werden soll, der Identifikator der DE–Inkarnation;
- falls eine Sperre für eine Zugriffskontrolliste erworben werden soll, der Identifikator der zugriffskontrollierten Komponente, der diese Zugriffskontrolliste zugeordnet ist;
- falls eine Sperre für eine Zugriffshistorienliste erworben werden soll, der Identifikator der Komponente, der diese Zugriffshistorienliste zugeordnet ist.

Die Ausführung von `AcquireLock` bewirkt dann den Aufruf von `Acquire` mit dem angegebenen Modus auf der Leser–Schreiber–Sperre des entsprechenden Objekts. Der Akteur bzw.

Akteursphärenverwalter, der die Operation `AcquireLock` aufruft, bleibt solange blockiert, bis der `Acquire`–Aufruf und damit auch der `AcquireLock`–Aufruf terminiert.

Durch Aufruf der Operation `ReleaseLock` kann eine erworbene Lese– oder Schreib–Sperre wieder freigegeben werden. Analog zu `AcquireLock` sind folgende Informationen bei einem `ReleaseLock`–Aufruf als Parameter zu übergeben:

- der Modus, d.h. ob eine Lese– oder Schreib–Sperre freigegeben werden soll;
- die Art des Objekts, für das die Sperre freigegeben werden soll, also entweder eine DE–Inkarnation, Zugriffskontrolliste oder Zugriffshistorienliste;
- falls eine Sperre für eine DE–Inkarnation freigegeben werden soll, der Identifikator der DE–Inkarnation;
- falls eine Sperre für eine Zugriffskontrolliste freigegeben werden soll, der Identifikator der zugriffskontrollierten Komponente, der diese Zugriffskontrolliste zugeordnet ist;
- falls eine Sperre für eine Zugriffshistorienliste freigegeben werden soll, der Identifikator der Komponente, der diese Zugriffshistorienliste zugeordnet ist.

Die Ausführung von `ReleaseLock` bewirkt den Aufruf der Operation `Release` mit dem angegebenen Modus auf der Leser–Schreiber–Sperre des entsprechenden Objekts. Damit wird die Lese– bzw. Schreib–Sperre freigegeben.

Aufbauend auf den erklärten Operationen zum Erwerb bzw. zur Freigabe von Sperren wird nun genauer angegeben, wie die für die Auswertung von Zugriffsrestriktionsausdrücken geforderte Atomarität realisiert wird.

Bei Auswertung eines Zugriffsrestriktionsausdrucks, d.h. im Rahmen der Ausführung der Operation `CheckAccess`, müssen jeweils vor Auswertung eines Teilausdrucks Lese–Sperren für die Objekte erworben werden, auf die bei Auswertung des Teilausdrucks zugegriffen wird. Ein Teilausdruck, vor dessen Auswertung Lese–Sperren zu erwerben sind, ist entweder ein `IN_ACL`–Prädikat, ein `ACCESSED`–Prädikat oder eine Gleichung bzw. Ungleichung, die Bezeichner lokaler oder globaler variabler DE–Inkarnationen enthält. Vor Auswertung eines `IN_ACL`–Prädikats ist eine Lese–Sperre für die Zugriffskontrolliste der zugriffskontrollierten Komponente zu erwerben, die in dem Prädikat identifiziert wird. Wird die Komponentenbeschreibung dieser Komponente von dem Akteursphärenverwalter verwaltet, der das `IN_ACL`–Prädikat auszuwerten hat, so kann die Lese–Sperre direkt durch Aufruf der Operation `Acquire` auf der der Zugriffskontrolliste assoziierten Leser–Schreiber–Sperre erworben werden. Anderenfalls wird die Operation `AcquireLock` auf dem Akteursphärenverwalter aufgerufen, der die Komponentenbeschreibung dieser Komponente verwaltet. Analog ist vor Auswertung eines `ACCESSED`–Prädikats eine Lese–Sperre für die Zugriffshistorienliste der in dem Prädikat identifizierten Komponente zu erwerben. Ist eine Gleichung oder Ungleichung, die variable DE–Inkarnationen enthält, auszuwerten, so sind für diese zunächst durch `AcquireLock`–Aufrufe auf den Akteursphärenverwaltern, die die Komponentenbeschreibungen dieser DE–Inkarnationen verwalten, Lese–Sperren zu erwerben. Am Ende der Auswertung eines Zugriffsrestriktionsausdrucks, d.h. unmittelbar vor Terminierung der Ausführung von `CheckAccess`, werden alle im Rahmen dieser `CheckAccess`–Ausführung erworbenen Lese–Sperren durch `Release`– bzw. `ReleaseLock`–Aufrufe freigegeben. Dies liefert auch die Begründung dafür, daß der Erwerb einer Lese–Sperre für eine Zugriffskontroll– bzw. Zugriffshistorienliste stets vor dem zur Auswertung des entsprechenden `IN_ACL`– bzw. `ACCESSED`–Prädikats ggf. erforderlichen Aufruf der Operation `CheckACL` bzw. `CheckAHL` erfolgt. Würde die Lese–Sperre erst

zu Beginn der Ausführung der Operation CheckACL bzw. CheckAHL erworben, so würde sie durch den Akteursphärenverwalter erworben, der die entsprechende Zugriffskontroll– bzw. Zugriffshistorienliste verwaltet, und nicht von dem, der den Zugriffsrestriktionsausdruck auszuwerten hat. Dementsprechend müßte die Lese–Sperre bereits am Ende der CheckACL– bzw. CheckAHL–Ausführung wieder freigegeben werden und nicht wie gefordert erst am Ende der Auswertung des Zugriffsrestriktionsausdrucks.

Vor Aufruf einer Schreiboperation auf einem Objekt, auf das in einem Zugriffsrestriktionsausdruck potentiell zugegriffen wird, muß eine Schreib–Sperre für das Objekt erworben werden, die nach Ausführung der Schreiboperation wieder freigegeben wird. Für den Erwerb von Schreib–Sperren für Zugriffskontrollisten werden die von den Akteursphärenverwaltern zur Änderung von Zugriffskontrollisten bereitgestellten Operationen ChangeACLCall und ChangeACLEntry dahingehend erweitert, daß vor Änderung der Zugriffskontrolliste ein Acquire–Aufruf zum Erwerb einer Schreib–Sperre auf der von dem Verwalter lokal verwalteten Leser–Schreiber–Sperre der Zugriffskontrolliste erfolgt. Nach Durchführung der Änderung wird diese Sperre durch Aufruf von Release auf dieser Leser–Schreiber–Sperre freigegeben. Die zur Änderung von Zugriffshistorienlisten zur Verfügung stehende Operation ChangeAHL wird analog erweitert. In den zur Ausgabe einer Zugriffskontrolliste dienenden Operationen ListACLCall bzw. ListACL erfolgt am Beginn ein Acquire–Aufruf zum Erwerb einer Lese–Sperre für die jeweilige Zugriffskontrolliste, die am Ende der Operationsausführung durch einen entsprechenden Release–Aufruf wieder freigegeben wird. Durch die Acquire– und Release–Aufrufe in den Operationen ChangeACLCall, ChangeACLEntry, ListACLCall, ListACL bzw. ChangeAHL und dem Erwerb einer Lese–Sperre vor Aufruf der Operationen CheckACL bzw. CheckAHL wird die in den vorhergehenden Abschnitten geforderte Leser–Schreiber–Synchronisation der Zugriffe auf die von den Akteursphärenverwaltern verwalteten Zugriffskontrollisten und Zugriffshistorienlisten realisiert. Der Erwerb einer Lese–Sperre zu Beginn der Ausführung von CheckACL bzw. CheckAHL ist dabei nicht notwendig, da die benötigte Lese–Sperre – wie oben erläutert – bereits vor Aufruf von CheckACL bzw. CheckAHL von dem aufrufenden Akteursphärenverwalter erworben wurde.

Jeder Schreibzugriff auf eine DE-Inkarnation, auf die bei Auswertung eines Zugriffsrestriktionsausdrucks potentiell zugegriffen wird, wird durch einen AcquireLock–Aufruf zum Erwerb einer Schreib–Sperre und einen ReleaseLock–Aufruf zur Freigabe dieser Sperre geklammert. AcquireLock und ReleaseLock werden dabei von dem den Schreibzugriff ausführenden Akteur auf dem Akteursphärenverwalter aufgerufen, der die Komponentenbeschreibung der DE–Inkarnation verwaltet. Zusätzlich kann auch jeder durch einen Akteur ausgeführte Lesezugriff auf die DE–Inkarnation durch Aufruf der AcquireLock– und ReleaseLock–Operation des Akteursphärenverwalters, dessen Sphäre die DE–Inkarnation zugeordnet ist, geklammert werden. Dies ist jedoch aufgrund der Semantik von INSEL bzw. INSEL^{+}, die weder den wechselseitigen Ausschluß noch die Leser–Schreiber–Synchronisation der Zugriffe auf DE–Inkarnationen festlegt, nicht unbedingt notwendig.

Mit der Erläuterung der Maßnahmen zur Realisierung der für die Auswertung von Zugriffsrestriktionsausdrücken geforderten Atomarität ist vollständig erklärt, wie die Akteursphärenverwalter die Zugriffskontrollen kooperativ durchführen. Das allgemeine Verfahren für die Durchführung der Zugriffskontrollen wird im folgenden beispielhaft anhand des Aufrufs einer Kommunikationsoperation eines zugriffskontrollierten K-Akteurs verdeutlicht.

Beispiel

Als Beispiel wird eine Situation aus dem Kontenverwaltungssystem betrachtet. Es wird angenommen, daß ein Benutzerrepräsentant, der für einen Kundenbetreuer erzeugt wur-

de, die Kommunikationsoperation `KontoAufloesen` auf dem zugriffskontrollierten K-Akteur `KontoVerwalter` aufruft. Der entsprechende K-Order–Aufruf ist in dem INSEL$^+$–Programm des Kontenverwaltungssystems (siehe Anhang B) in der Benutzerrepräsentanten–Generatorklasse `Kundenbetreuer` enthalten:

```
KontoVerwalter.KontoAufloesen(Kontozeiger,ActError) STATUS ErlaubterZugriff;
```

`Kontozeiger` ist ein Zeiger, der auf ein zugriffskontrolliertes Depot zeigt, das Inkarnation bzgl. des Generators `KontoTyp` ist. Er verweist auf das Konto, das durch Ausführung der Operation `KontoAufloesen` aufgelöst werden soll[12]. Vor Ausführung der Operation, d.h. vor Erzeugung der entsprechenden K-Order ist der für die Operation `KontoAufloesen` festgelegte Zugriffsrestriktionsausdruck auszuwerten. Dieser lautet:

$$
\begin{aligned}
R_{\texttt{KontoAufloesen}}: \; & \texttt{Caller.Role} = \texttt{Kundenbetreuer AND} \\
& \texttt{IN_ACL}(\texttt{Caller.User}, \texttt{Caller.Role}, \texttt{KontoZeiger}, \texttt{Aufloesen}) \; \texttt{AND} \\
& \texttt{8} <= \texttt{Zeit.Stunde AND Zeit.Stunde} < \texttt{17 AND} \\
& \texttt{1} <= \texttt{Zeit.Wochentag AND Zeit.Wochentag} < \texttt{6}
\end{aligned}
$$

Zur Auswertung dieses Zugriffsrestriktionsausdrucks ist also neben der Rollenrestriktion und den Wertvergleichen über den Komponenten der globalen DE–Inkarnation `Zeit` die Zugriffskontrollliste des aufzulösenden Konto–Depots auszuwerten, auf das der aktuelle Eingabeparameter `KontoZeiger` des `KontoAufloesen`–Aufrufs verweist. Die Realisierung des Aufrufs der Operation `KontoAufloesen` auf dem K-Akteur `KontoVerwalter` und die hierbei durchzuführende Zugriffskontrolle, die in der Auswertung des für `KontoAufloesen` festgelegten Zugriffsrestriktionsausdrucks besteht, ist für die Realisierungsebene 1 in Abbildung 6.4 verdeutlicht.

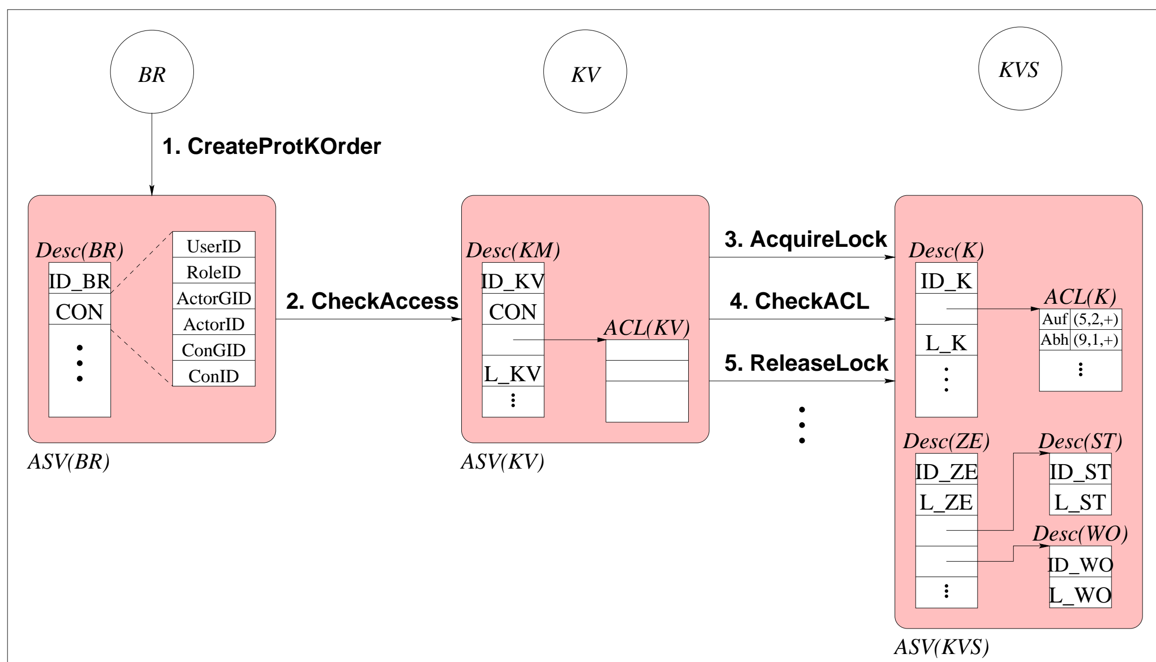

Abbildung 6.4: Durchführung von Zugriffskontrollen auf Realisierungsebene 1

[12]Hierbei handelt es sich um die „logische“ Auflösung des Kontos innerhalb der Anwendung. Das dieses Konto repräsentierende zugriffskontrollierte Depot wird aufgrund seiner Lebenszeitabhängigkeit zur Hauptkomponente des Kontenverwaltungssystems erst mit deren Auflösung „physikalisch“ aufgelöst.

Im linken Teil der Abbildung ist der realisierte Benutzerrepräsentant BR, der den K-Order–Aufruf ausführt, zusammen mit seinem Akteursphärenverwalter $ASV(BR)$ dargestellt. $ASV(BR)$ verwaltet in seinem Lebenszeitkeller die Komponentenbeschreibung von BR, die mit $Desc(BR)$ bezeichnet ist. $Desc(BR)$ enthält neben dem Identifikator `ID_BR` von BR insbesondere die Kontext–Datenstruktur `CON` von BR. Der mittlere Teil der Abbildung zeigt den Akteursphärenverwalter $ASV(KV)$, der dem realisierten K-Akteur `KontoVerwalter` (abgekürzt: KV) zugeordnet ist. $ASV(KV)$ verwaltet in seinem Lebenszeitkeller die Komponentenbeschreibung $Desc(KV)$ von KV, die neben dem Identifikator `ID_KV` und der Kontext–Datenstruktur von KV einen Verweis auf die Zugriffskontrolliste von KV sowie die der Zugriffskontrolliste assoziierte Leser–Schreiber–Sperre `L_KV` enthält[13]. Im rechten Teil der Abbildung ist der Akteursphärenverwalter $ASV(KVS)$ der realisierten Hauptkomponente KVS des Kontenverwaltungssystems dargestellt, von der alle Konto–Depots lebenszeitmäßig abhängig sind. Dementsprechend verwaltet $ASV(KVS)$ in seinem Lebenszeitkeller die Komponentenbeschreibung des Konto–Depots K, auf das der in dem obigen K-Order–Aufruf angegebene Zeiger `Kontozeiger` verweist. Die Komponentenbeschreibung $Desc(K)$ von K enthält neben dem Identifikator `ID_K` des Depots einen Verweis auf die Zugriffskontrolliste von K und die Leser–Schreiber–Sperre `L_K`, die dieser Zugriffskontrolliste zugeordnet ist. Weiterhin verwaltet $ASV(KVS)$ die Komponentenbeschreibung der globalen DE–Inkarnation `Zeit` (abgekürzt: ZE). Da `Zeit` ein Record ist, enthält die Komponentenbeschreibung $Desc(ZE)$ Verweise auf die Komponentenbschreibungen der einzelnen Komponenten des Records (z.B. $Desc(ST)$ für die Komponente `Stunde`), denen jeweils eine eigene Leser–Schreiber–Sperre (z.B. `L_ST`) assoziiert ist.

Der Aufruf der Kommunikationsoperation `KontoAufloesen` auf dem zugriffskontrollierten K-Akteur `KontoVerwalter` wird wie folgt realisiert:

1. Der Benutzerrepräsentant BR ruft die Operation `CreateProtKOrder` auf seinem Akteursphärenverwalter $ASV(BR)$ auf. Dabei werden insbesondere die aktuellen Parameter des K-Order–Aufrufs an $ASV(BR)$ übergeben.

2. Zur Auswertung des für `KontoAufloesen` festgelegten Zugriffsrestriktionsausdrucks ruft $ASV(BR)$ die Operation `CheckAccess` auf dem Akteursphärenverwalter $ASV(KV)$ des K-Akteurs KV auf. Dabei werden die aktuellen Parameter des K-Order–Aufrufs sowie die aktuellen Werte der Kontext–Datenstruktur von BR an $ASV(KV)$ übergeben.
   Im Rahmen der Ausführung von `CheckAccess` durch $ASV(KV)$ wird zunächst die Rollenrestriktion des Zugriffsrestriktionsausdrucks ausgewertet. Dazu wird der übergebene Rollenidentifikator der Kontext–Datenstruktur von BR mit dem Rollenidentifikator der Rolle `Kundenbetreuer` verglichen. Stimmen die beiden Rollenidentifikatoren nicht überein, terminiert der Aufruf von `CheckAccess` mit dem Wert *false*.

3. Sind die Werte der beiden Rollenidentifikatoren gleich, wird das `IN_ACL`–Prädikat des Zugriffsrestriktionsausdrucks ausgewertet. In diesem `IN_ACL`–Prädikat wird das Konto-Depot K identifiziert, was bedeutet, daß die Zugriffskontrolliste des Depots K auszuwerten ist. Dementsprechend ruft $ASV(KV)$ die Operation `AcquireLock` zum Erwerb einer Lese–Sperre für diese Zugriffskontrolliste auf dem Akteursphärenverwalter $ASV(KVS)$ auf, der die Komponentenbeschreibung von K verwaltet. Die Ausführung von `AcquireLock` bewirkt, daß $ASV(KVS)$ die Operation `Acquire` zum Erwerb einer Lese–Sperre auf der der Zugriffskontrolliste assoziierten Leser–Schreiber–Sperre `L_K`

[13] Anmerkung: Da die Rechte zur Nutzung des zentralen Kontoverwalters nicht auf Basis von `IN_ACL`–Prädikaten vergeben werden (vgl. Abschnitt 4.6.1) enthält die Zugriffskontrolliste von KV keine Einträge.

aufruft[14]. Die damit erworbene Lese–Sperre wird mit Terminierung von `AcquireLock` an den aufrufenden Akteursphärenverwalter $ASV(KV)$ übergeben.

4. Nach Erwerb der Lese–Sperre ruft $ASV(KV)$ die Operation `CheckACL` auf $ASV(KVS)$ mit den sich aus den Parametern des `IN_ACL`–Prädikats ergebenden aktuellen Parameterwerten auf. Die Ausführung von `CheckACL` durch $ASV(KVS)$ bewirkt dann die Auswertung der Zugriffskontrolliste von K gemäß der Semantik des `IN_ACL`–Prädikats und die Rückgabe des Ergebnisses dieser Auswertung an $ASV(KV)$.

5. Ist das Ergebnis des `CheckACL`–Aufrufs *false*, wird die für die Zugriffskontrolliste von K erworbene Lese–Sperre durch Aufruf der Operation `ReleaseLock` auf $ASV(KVS)$ freigegeben und die Ausführung von `CheckAccess` terminiert mit *false*.

6. Falls das Ergebnis des `CheckACL`–Aufrufs *true* ist, sind von $ASV(KV)$ als nächstes die beiden Wertvergleiche für die Komponente `Stunde` der globalen DE–Inkarnation `Zeit` und anschließend die Wertvergleiche für die Komponente `Wochentag` durchzuführen. Hierzu ist zunächst jeweils durch Aufruf von `AcquireLock` auf $ASV(KVS)$ eine Lese–Sperre für `Stunde` bzw. `Wochentag` zu erwerben. Anschließend ist der jeweilige Wertvergleich durchzuführen. Ist eine der Ungleichungen nicht erfüllt, so werden alle bis dahin erworbenen Lese–Sperren durch `ReleaseLock`–Aufrufe auf $ASV(KVS)$ freigegeben und die Ausführung von `CheckAccess` terminiert mit dem Wert *false*.

7. Werden alle vier Wertvergleiche zu *true* ausgewertet, werden die erworbenen Lese–Sperren für die Zugriffskontrolliste von K und für die DE–Inkarnationen `Stunde` und `Wochentag` durch `ReleaseLock`–Aufrufe auf $ASV(KVS)$ freigegeben. Anschließend terminiert die Ausführung von `CheckAccess` mit dem Wert *true*.

8. Abhängig von dem Ergebnis des `CheckAccess`–Aufrufs auf $ASV(KV)$ führt $ASV(BR)$ weitere Aktionen durch. Ist das Ergebnis *false*, so wird die in dem Status–Part des K-Order–Aufrufs angegebene boolesche Variable `ErlaubterZugriff` mit dem Wert *false* belegt und die Ausführung von `CreateProtKOrder` terminiert. Anderenfalls wird die boolesche Variable `ErlaubterZugriff` mit dem Wert *true* belegt, und es werden die Maßnahmen zur Erzeugung der entsprechenden K–Order durchgeführt, d.h. $ASV(BR)$ ruft die Operation `InsertKOrder` auf dem Akteursphärenverwalter $ASV(KV)$ auf.

◇

Das Beispiel zeigt, daß für die Auswertung eines Zugriffsrestriktionsausdrucks im allgemeinen nicht die Werte aller Teilterme des Ausdrucks bestimmt werden müssen. Die Auswertung kann beendet werden, sobald der Wert des Ausdrucks feststeht. In dem Beispiel kann die Auswertung für den Fall, daß die Rollenrestriktion nicht erfüllt ist, bereits nach Auswertung dieser Rollenrestriktion abgeschlossen werden, da dann feststeht, daß der Wert des Gesamtausdrucks *false* ist. Analog kann die Auswertung eines Zugriffsrestriktionsausdrucks in Disjunktiver Normalform mit dem Ergebniswert *true* beendet werden, sobald einer der disjunktiven Teilterme zu *true* ausgewertet worden ist.

6.2.2.6 Zusammenfassung

In diesem Abschnitt wurde die Funktionalität der Akteursphärenverwalter beschrieben. Dazu wurden die Schnittstellenoperationen und die Datentrukturen erklärt, die sie zur Erfüllung

[14] Die `Acquire`– und `Release`–Aufrufe auf den den Zugriffskontrollisten assoziierten Leser–Schreiber–Sperren sind in Abbildung 6.4 nicht dargestellt.

der von ihnen wahrzunehmenden Aufgaben der Erzeugung, Terminierung und Auflösung von Inkarnationen, der Realisierung des Operationen–orientierten Rendezvous sowie der Durchführung von Zugriffskontrollen und der Repräsentation sowie Verwaltung der für diese Kontrollen benötigten Informationen zur Verfügung stellen.

Mit den Akteursphärenverwaltern sowie dem kurz angesprochenen globalen Prozessorverwalter und den Speicherverwaltern ist geklärt, wie die zur Realisierung eines INSEL$^+$–Systems auf der Realisierungsebene 1 notwendigen Verwaltungsaufgaben durchgeführt werden. Die Akteure, deren Rechenfähigkeiten mit den virtuellen Prozessoren realisiert werden, rufen zur Erzeugung und Auflösung von Komponenten sowie zur Durchführung von Zugriffen auf Komponenten die Schnittstellenoperationen der Akteursphärenverwalter auf. Diese wiederum nutzen die Speicherverwalter und den globalen Prozessorverwalter, um den für die Realisierung von Komponenten benötigten Speicher und die für die Realisierung von Akteuren zusätzlich benötigten Prozessoren zu allokieren bzw. die für die Repräsentation einer Komponente belegten Ressourcen bei Auflösung der Komponente freizugeben. Die Akteursphärenverwalter führen zudem die zur Durchsetzung des Rechts des INSEL$^+$–Systems notwendigen Zugriffskontrollen eingeordnet in die Maßnahmen zur Erzeugung von DA–Inkarnationen kooperativ durch. Die für diese Zugriffskontrollen benötigten Informationen werden durch die Akteursphärenverwalter gekapselt in geeigneten Datenstrukturen, um die die Komponentenbeschreibungen erweitert werden, repräsentiert und sicher verwaltet. Die Akteursphärenverwalter sind sicherheitsrelevante Komponenten und sind somit als vertrauenswürdige Komponenten der Trusted Computing Base zu konstruieren.

Zum Abschluß dieses Abschnitts folgen noch einige Anmerkungen zur Differenzierung der Fähigkeiten der Akteursphärenverwalter. Die Funktionalität eines Akteursphärenverwalters spiegelt sich u.a. in den von ihm angebotenen Schnittstellenoperationen wider. Die Schnittstellenoperationen der Akteursphärenverwalter sind in den Tabellen 6.1 bis 6.7 angegeben. Wie aus diesen Tabellen ersichtlich ist, ist die Anzahl der insgesamt von den Akteursphärenverwaltern angebotenen Schnittstellenoperationen relativ hoch, was der Vielfalt der von den Akteursphärenverwaltern im allgemeinen durchzuführenden Aufgaben entspricht. Wie im folgenden begründet wird, ist es jedoch nicht notwendig, daß jeder Akteursphärenverwalter alle diese Schnittstellenoperationen anbietet. Die Aufgaben der Akteursphärenverwalter lassen sich grob in zwei Klassen einteilen: zum einen die Aufgaben, die im weitesten Sinne Ressourcenverwaltungsaufgaben sind, d.h. die die Erzeugung und Auflösung von Inkarnationen betreffen, und zum anderen eingeordnet in diese die Aufgaben, die die Durchführung von Zugriffkontrollen betreffen. Die Ressourcenverwaltungsaufgaben sind von allen Akteursphärenverwaltern gleichermaßen wahrzunehmen. Im Hinblick auf die Durchführung der Zugriffskontrollen und die Verwaltung der für diese Kontrollen benötigten Informationen ergeben sich jedoch Unterschiede. Ein Akteursphärenverwalter, dessen Sphäre keine zugriffskontrollierten Komponenten angehören können, braucht zum Beispiel keine Zugriffskontrollisten zu verwalten und muß dementsprechend auch keine Schnittstellenoperationen zur Auswertung, Änderung und Ausgabe von Zugriffskontrollisten anbieten. Ebenso ist für einen Akteursphärenverwalter, der keine zugriffskontrollierten Komponenten zu verwalten hat, die Operation `CheckAccess` zur Auswertung von Zugriffsrestriktionsausdrücken entbehrlich. Analog werden die Operationen zur Auswertung und Aktualisierung von Zugriffshistorienlisten nicht benötigt, wenn einem Akteursphärenverwalter keine Komponenten zugeordnet werden können, für die eine Zugriffshistorienliste zu verwalten ist.

Diese Überlegungen motivieren die Einführung unterschiedlicher **Klassen von Akteursphärenverwaltern**, die sich durch ihre funktionalen und qualitativen Eigenschaften unterscheiden. Die Verwalter sind somit keine uniformen Komponenten, sondern besitzen

abhängig von den Aufgaben, die sie durchzuführen haben, unterschiedliche Fähigkeiten. Die mit Einführung der Verwalterklassen bei Erzeugung eines Akteurs notwendig werdende Entscheidung, welche „Art“ von Akteursphärenverwalter für diesen Akteur und dessen Sphäre zu erzeugen ist, kann automatisiert und damit transparent für den Anwendungsentwickler auf Basis statischer Analysen getroffen werden. So kann zum Beispiel durch eine statische Analyse ermittelt werden, ob der Akteursphäre eines Akteurs potentiell zugriffskontrollierte Komponenten angehören können oder nicht.

Auf die unterschiedlichen Klassen von Akteursphärenverwaltern und die angesprochenen statischen Analysen wird hier nicht weiter eingegangen. Es bleibt lediglich festzuhalten, daß mit der Einführung unterschiedlicher Verwalterklassen Akteursphärenverwalter mit anwendungsangepaßter Granularität und Funktionalität erzeugt werden können, deren Schnittstelle den von ihnen tatsächlich durchzuführenden Aufgaben angeglichen ist.

6.2.3 Tickets

Das bisher erklärte Verfahren für die zur Durchsetzung des Rechts eines INSEL$^+$–Systems durchzuführenden Zugriffskontrollen ist dadurch charakterisiert, daß immer dann, wenn ein Akteur eine äußere Operation einer zugriffskontrollierten Komponente aufruft, der für diese Operation festgelegte Zugriffsrestriktionsausdruck ausgewertet wird. Insbesondere wird auch in dem Fall, daß der Zugriffsrestriktionsausdruck zu *true* ausgewertet wird, der Akteur also das Recht zur Ausführung der Operation hat, bei einem erneuten Aufruf der Operation durch den Akteur der Zugriffsrestriktionsausdruck erneut ausgewertet, um zu überprüfen, ob der Akteur nach wie vor das Recht zur Ausführung der Operation hat. Die Notwendigkeit der erneuten Auswertung des Zugriffsrestriktionsausdrucks ergibt sich im allgemeinen daraus, daß sich der Zustand von Objekten, von denen die Auswertung des Zugriffsrestriktionsausdrucks abhängig ist, zwischen dem Zeitpunkt des ersten Operationsaufrufs und dem Zeitpunkt des erneuten Operationsaufrufs geändert haben kann. Zum Beispiel kann sich der Kontext des Akteurs und dementsprechend der Wert seiner Kontext–Datenstruktur oder die Zugriffskontrolliste der zugriffskontrollierten Komponente geändert haben. Diese Zustandsänderungen können zur Folge haben, daß der Zugriffsrestriktionsausdruck bei dem erneuten Operationsaufruf zu *false* ausgewertet wird und somit der Akteur nicht mehr das Recht zur Ausführung der Operation hat. Hat sich jedoch zwischenzeitlich der Zustand von Objekten, von denen die Auswertung des Zugriffsrestriktionsausdrucks abhängig ist, nicht geändert, so ist die erneute Auswertung des Zugriffsrestriktionsausdrucks nicht erforderlich. Ausgehend von dieser Überlegung wird in diesem Abschnitt das bisher erklärte Verfahren zur Durchführung der Zugriffskontrollen um Maßnahmen ergänzt, die es ermöglichen, einen Zugriffsrestriktionsausdruck erst dann erneut auswerten zu müssen, wenn dies auch tatsächlich notwendig ist. Da die Auswertung eines Zugriffsrestriktionsausdrucks, wie im vorhergehenden Abschnitt bereits erläutert, im allgemeinen recht aufwendig sein kann, kann durch gezielten Einsatz dieser zusätzlichen Maßnahmen der Aufwand für die durchzuführenden Zugriffskontrollen reduziert werden.

Die Grundidee besteht darin, nach positiver Auswertung des Zugriffsrestriktionsausdrucks einer äußeren Operation einer zugriffskontrollierten Komponente an den Akteur, der die Operation aufruft, ein **Ticket**[15] zu vergeben. Sofern dieses Ticket nicht zwischenzeitlich gelöscht wird, berechtigt es den Akteur bei einem erneuten Aufruf der Operation zur Ausführung der Operation, ohne daß der Zugriffsrestriktionsausdruck der Operation erneut ausgewertet

[15]Festlegung: *das* Ticket, -s, -s

werden muß. Ändert sich nach der Vergabe des Tickets der Zustand von Objekten, von denen die Auswertung des Zugriffsrestriktionsausdrucks der Operation abhängig ist, wird das Ticket gelöscht, da in diesem Fall der Zugriffsrestriktionsausdruck bei einem erneuten Aufruf der Operation erneut auszuwerten ist.

Die eingeführten Tickets sind in einigen ihrer Eigenschaften mit Capabilities (siehe Kapitel 2) vergleichbar. So berechtigt das Vorhandensein eines Tickets analog zum Besitz einer Capability zur Ausführung einer Operation, ohne daß weitere Überprüfungen vorgenommen werden. Tickets unterscheiden sich jedoch von Capabilities zum einen darin, daß der Besitz eines Tickets nicht notwendige Voraussetzung für den Zugriff auf eine zugriffskontrollierte Komponente ist. Insbesondere wird ein Ticket nicht zur Identifizierung einer Komponente benutzt. Zum anderen werden Tickets explizit gelöscht, wenn sie ihre Gültigkeit verlieren, was für Capabilities in der Regel nicht der Fall ist. Das in vielen Capability–Systemen bestehende Problem der unkontrollierten Weitergabe von Capabilities wird für die Tickets dadurch gelöst, daß diese durch die vertrauenswürdigen Akteursphärenverwalter verwaltet werden.

Im folgenden wird beschrieben, wie die Funktionalität der im vorhergehenden Abschnitt erklärten Akteursphärenverwalter um Fähigkeiten zur Vergabe, zum Einsatz sowie zum Löschen von Tickets erweitert werden kann. Die vorgeschlagenen Erweiterungen stellen lediglich eine Möglichkeit dar, die Akteursphärenverwalter um die Funktionalität zur Verwaltung von Tickets anzureichern. Auf eine alternative Realisierungsmöglichkeit sowie ihrer Vor– und Nachteile wird im Anschluß an die Beschreibung der hier bevorzugten Alternative eingegangen.

6.2.3.1 Vergabe von Tickets

Tickets, die zur Ausführung der äußeren Operation OP einer zugriffskontrollierten Komponente Z berechtigen, werden von dem Akteursphärenverwalter vergeben, der die Komponentenbeschreibung der Komponente Z verwaltet. Dazu wird die Funktionalität der Schnittstellenoperation `CheckAccess`, die von den Akteursphärenverwaltern zur Auswertung von Zugriffsrestriktionsausdrücken zur Verfügung gestellt wird, entsprechend erweitert. Wie in Abschnitt 6.2.2.5 erklärt wurde, wird die Operation `CheckAccess` im Rahmen der Realisierung des Aufrufs einer Operation OP einer zugriffskontrollierten Komponente Z auf dem Akteursphärenverwalter $ASV(Z)$ aufgerufen, der die Komponentenbeschreibung von Z verwaltet. Der Aufruf von `CheckAccess` erfolgt dabei durch den Akteursphärenverwalter $ASV(A)$ des Akteurs A, der den entsprechenden Operationsaufruf ausführt. Die Ausführung von `CheckAccess` durch $ASV(Z)$ bewirkt, daß der für die Operation OP festgelegte Zugriffsrestriktionsausdruck ausgewertet wird. Ist das Ergebnis dieser Auswertung *true*, so **kann** $ASV(Z)$ ein Ticket ausstellen, das bei Terminierung von `CheckAccess` an den aufrufenden Akteursphärenverwalter $ASV(A)$ übergeben wird und den Akteur A zur Ausführung der Operation OP berechtigt. Die von $ASV(Z)$ zu treffende Entscheidung darüber, ob ein Ticket für die Operation OP an A vergeben wird, ist von zahlreichen Faktoren abhängig, auf die in Abschnitt 6.2.3.5 noch eingegangen wird. Für die weiteren Erklärungen wird davon ausgegangen, daß der Akteursphärenverwalter $ASV(Z)$ die Entscheidung getroffen hat, ein Ticket, das zur Ausführung der Operation OP berechtigt, für den Akteur A auszustellen. Das dann für den Akteur A ausgestellte Ticket hat den in Abbildung 6.5 dargestellten Aufbau.

Neben dem Ticket–Identifikator `TicketId`, über den das Ticket systemweit eindeutig identifizierbar ist, enthält das Ticket den Identifikator der zugriffskontrollierten Komponente Z (`ComponentId`) und den Identifikator der äußeren Operation OP von Z (`OperationId`), zu deren Ausführung das Ticket berechtigt. Der `ActorContext` enthält die aktuellen Werte

TicketId	ComponentId	OperationId	ActorContext

Abbildung 6.5: Aufbau eines Tickets

der Elemente der Kontext–Datenstruktur von A, von denen die Auswertung des Zugriffsrestriktionsausdrucks von OP abhängig ist. Die aktuellen Werte der Elemente der Kontext–Datenstruktur von A werden bei Aufruf der Operation `CheckAccess` an den Akteursphärenverwalter $ASV(Z)$ übergeben. Die Abspeicherung der Werte der Elemente der Kontext–Datenstruktur von A in dem Ticket ermöglicht es, daß der Akteursphärenverwalter $ASV(A)$, an den das Ticket übergeben wird, bei einem erneuten Aufruf der Operation OP durch den Akteur A überprüfen kann, ob die dann aktuellen Werte der Elemente der Kontext–Datenstruktur von A den in dem Ticket angegebenen entsprechen. Nur wenn dies der Fall ist, braucht der Zugriffsrestriktionsausdruck von OP nicht erneut ausgewertet zu werden. Anderenfalls ist das Ticket ungültig und der Zugriffsrestriktionsausdruck von OP muß erneut ausgewertet werden, um zu überprüfen, ob der Akteur A auch in dem „anderen" Kontext, in dem er sich bei dem erneuten OP–Aufruf befindet, das Recht zur Ausführung der Operation OP hat. Ein Ticket ist also immer nur in dem Kontext, der durch den `ActorContext` des Tickets festgelegt wird, gültig. Hierin unterscheiden sich die Tickets deutlich von den klassischen Capabilities, die in der Regel kontextunabhängig ausgestellt werden. Die kontextabhängige Gültigkeit der Tickets bietet den Vorteil, daß Angriffe, die sich aus der unautorisierten Anfertigung von Kopien eines Tickets ergeben, erkannt und abgewehrt werden können, da bei Vorlage eines unautorisiert kopierten Tickets im allgemeinen die Kontextbedingung nicht erfüllt ist.

Es ist unmittelbar einsichtig, daß es genügt, in dem `ActorContext` lediglich die Werte der Elemente der Kontext–Datenstruktur anzugeben, von denen die Auswertung des Zugriffsrestriktionsausdrucks abhängig ist. Enthält der Zugriffsrestriktionsausdruck von OP zum Beispiel nicht den Bezeichner `Caller.Role`, so bedeutet dies, daß der Wert des Zugriffsrestriktionsausdrucks unabhängig von der Rolle ist, in der der aufrufende Akteur A agiert. Dementsprechend braucht der Identifikator der Rolle des Akteurs A, der Bestandteil der Kontext–Datenstruktur von A ist, nicht in dem `ActorContext` des Tickets vermerkt werden.

Nach Ausstellung des Tickets für den Akteur A speichert der Akteursphärenverwalter $ASV(Z)$ Informationen über dieses Ticket in einer Liste ab, die als Bestandteil der Komponentenbeschreibung von Z verwaltet wird. Diese Liste, die im weiteren als **Ticketausstellungsliste** bezeichnet wird, enthält für jede äußere Operation OP von Z einen Eintrag, der auf eine Liste verweist, die Informationen über die von $ASV(Z)$ ausgestellten Tickets, die zur Ausführung von OP auf Z berechtigen, enthält. Zu jedem dieser Tickets ist in dieser Liste ein Eintrag vorhanden, in dem der Identifikator des Tickets und der Identifikator des Akteursphärenverwalters, an den das Ticket übergeben wurde, vermerkt sind. Diese Informationen werden benötigt, um bei einer Zustandsänderung von Objekten, von denen die Auswertung des Zugriffsrestriktionsausdrucks der Operation OP abhängig ist, alle Akteursphärenverwalter, an die Tickets für OP übergeben wurden, darüber zu informieren, daß diese Tickets zu löschen sind (die Einzelheiten zum Löschen von Tickets werden in Abschnitt 6.2.3.3 erklärt).

Das von dem Akteursphärenverwalter $ASV(Z)$ für den Akteur A ausgestellte Ticket wird mit Terminierung des `CheckAccess`–Aufrufs an den Akteursphärenverwalter $ASV(A)$ übergeben. Dieser speichert das Ticket in einer **Ticketliste** ab, die als Bestandteil der Komponentenbeschreibung des Akteurs A verwaltet wird und alle Tickets enthält, die an den Akteur A vergeben wurden und noch nicht gelöscht wurden.

6.2.3.2 Einsatz von Tickets

Ein an einen Akteur A vergebenes Ticket für die äußere Operation OP einer zugriffskontrollierten Komponente Z berechtigt den Akteur A zur Ausführung der Operation OP, ohne daß der Zugriffsrestriktionsausdruck von OP ausgewertet werden muß, wenn die aktuellen Werte der Kontext–Datenstruktur von A den in dem `ActorContext` des Tickets angegebenen Werten entsprechen. Im Rahmen der Realisierung eines OP–Aufrufs auf Z durch A ist also zunächst zu überprüfen, ob der Akteur A bzw. sein Akteursphärenverwalter $ASV(A)$ ein Ticket für die Operation OP besitzt.

Ist die Operation OP die *erzeuge*–Operation auf einem explizit zugriffskontrollierten DA–Generator, ruft der Akteur A zur Realisierung des Aufrufs der *erzeuge*–Operation, wie in Abschnitt 6.2.2 erklärt, abhängig von der Art des Generators eine der Schnittstellenoperationen `CreateProtSOrder`, `CreateProtDepot`, `CreateProtMActor` oder `CreateProtKActor` auf seinem Akteursphärenverwalter $ASV(A)$ auf. Ist die Operation OP hingegen eine Zugriffsoperation eines zugriffskontrollierten Depots bzw. eine Kommunikationsoperation eines zugriffskontrollierten K-Akteurs, ruft der Akteur zur Realisierung des OP–Aufrufs die Schnittstellenoperation `CreateProtDepotOp` bzw. `CreateProtKOrder` auf $ASV(A)$ auf. Die Funktionalität der genannten Schnittstellenoperationen wird dahingehend erweitert, daß vor Aufruf der Operation `CheckAccess` auf dem Akteursphärenverwalter, der die Komponentenbeschreibung des explizit zugriffskontrollierten Generators bzw. des zugriffskontrollierten Depots oder K-Akteurs verwaltet, zunächst überprüft wird, ob in der von $ASV(A)$ verwalteten Ticketliste des Akteurs ein Ticket für die *erzeuge*–Operation bzw. die Zugriffs– oder Kommunikationsoperation vorhanden ist. Ist dies der Fall, wird überprüft, ob die in dem `ActorContext` des Tickets enthaltenen Werte, die den für die Gültigkeit des Tickets notwendigen Kontext festlegen, mit den entsprechenden aktuellen Werten der Kontext–Datenstruktur des Akteurs A übereinstimmen. Stimmen die Werte überein und ist somit das Ticket für den aktuellen Kontext des Akteurs A gültig, wird die Operation `CheckAccess` zur Auswertung des Zugriffsrestriktionsausdrucks nicht aufgerufen, sondern es werden gleich die Maßnahmen zur Erzeugung der jeweiligen DA–Inkarnation durchgeführt. In allen anderen Fällen, d.h. falls in der Ticketliste des Akteurs kein gültiges Ticket vorhanden ist, wird die Operation `CheckAccess` auf dem Akteursphärenverwalter aufgerufen, der die Komponentenbeschreibung des explizit zugriffskontrollierten Generators bzw. des zugriffskontrollierten Depots oder K-Akteurs verwaltet, um den Zugriffsrestriktionsausdruck von OP auszuwerten.

6.2.3.3 Löschen von Tickets

Ein Ticket, das an einen Akteur A für die äußere Operation einer zugriffskontrollierten Komponente vergeben wurde, wird dann ungültig, wenn sich der Zustand von Objekten, von denen die Auswertung des Zugriffsrestriktionsausdrucks dieser Operation abhängig ist, ändert. Die Objekte, von deren Wert der Wert des Zugriffsrestriktionsausdrucks abhängig sein kann und die nicht konstant sind, können neben der Kontext–Datenstruktur des Akteurs A Zugriffskontrollisten, Zugriffshistorienlisten sowie variable DE–Inkarnationen sein. Der Fall, daß sich der Wert der Kontext–Datenstruktur von A ändert, wurde bereits behandelt. Im Fall der Änderung einer Zugriffskontrolliste, einer Zugriffshistorienliste oder einer DE–Inkarnation wird das Ticket gelöscht.

Das generelle Verfahren zum Löschen von Tickets besteht darin, daß im Fall der Änderung des Zustands eines Objekts[16], auf das bei Auswertung von Zugriffsrestriktionsausdrücken

[16] Im weiteren dieses Abschnitts wird unter einem Objekt eine Zugriffskontrolliste, eine Zugriffshistorienliste oder eine variable DE–Inkarnation verstanden.

zugegriffen wird, alle Akteursphärenverwalter über diese Zustandsänderung informiert werden, die einen Zugriffsrestriktionsausdruck ausgewertet haben, dessen Wert von dem Zustand dieses Objekts abhängig ist. Diese Akteursphärenverwalter ermitteln dann anhand der von ihnen verwalteten Ticketausstellungslisten, welche Tickets aufgrund dieser Zustandsänderung zu löschen sind. Stellt ein solcher Akteursphärenverwalter fest, daß ein Ticket zu löschen ist, so ruft er eine spezielle Schnittstellenoperation auf dem Akteursphärenverwalter auf, an den das Ticket vergeben wurde. Die Ausführung dieser Operation bewirkt dann, daß das Ticket aus der von dem Akteursphärenverwalter verwalteten Ticketliste gelöscht wird.

Als Basis für das Löschen von Tickets werden für jedes Objekt, auf das bei Auswertung von Zugriffsrestriktionsausdrücken zugegriffen wird, Informationen darüber verwaltet, welche Akteursphärenverwalter welche Zugriffsrestriktionsausdrücke ausgewertet haben, deren Wert von dem Zustand dieses Objekts abhängig ist. Diese Informationen werden für jedes derartige Objekt von dem für das Objekt zuständigen Akteursphärenverwalter in einer Datenstruktur, die als **Auswertungsliste** bezeichnet wird, gespeichert. Die Auswertungsliste einer DE–Inkarnation wird als Bestandteil der Komponentenbeschreibung der DE–Inkarnation verwaltet, während die einer Zugriffskontrolliste bzw. einer Zugriffshistorienliste zugeordnete Auswertungsliste als Bestandteil der Komponentenbeschreibung der Komponente verwaltet wird, der diese Zugriffskontrolliste bzw. Zugriffshistorienliste zugeordnet ist. Die Auswertungsliste eines Objekts O enthält für jeden Akteursphärenverwalter, der einen Zugriffsrestriktionsausdruck ausgewertet hat, dessen Wert von dem Wert des Objekts O abhängig ist, einen Eintrag, der auf eine Liste von Paaren verweist, die jeweils aus einem Komponentenidentifikator und einem Operationsidentifikator bestehen. Ist ASV ein Akteursphärenverwalter, für den ein Eintrag in der Auswertungsliste des Objekts O existiert, und ist *(ComponentId, OperationId)* ein Paar in der Liste, auf die dieser Eintrag verweist, so besagt dies, daß

- der Akteursphärenverwalter ASV den Zugriffsrestriktionsausdruck der mit *OperationId* identifizierten äußeren Operation der zugriffskontrollierten Komponente, die mit *ComponentId* identifiziert wird, ausgewertet hat und
- die Wert dieses Zugriffsrestriktionsausdrucks von dem Wert des Objekts O abhängig ist.

Die Auswertungsliste eines Objekts O ist also immer dann zu aktualisieren, wenn ein Akteursphärenverwalter einen Zugriffsrestriktionsausdruck auswertet, dessen Wert von dem Wert des Objekts O abhängig ist. Ist O eine Zugriffskontrolliste, so ist die Auswertungsliste von O also dann zu aktualisieren, wenn ein Akteursphärenverwalter ein `IN_ACL`–Prädikat auswertet, in dem die zugriffskontrollierte Komponente identifiziert wird, der die Zugriffskontrolliste O zugeordnet ist. Analog ist bei Auswertung eines `ACCESSED`–Prädikats die Auswertungsliste der Zugriffshistorienliste der Komponente, die in dem Prädikat identifiziert wird, zu aktualisieren. Ist O eine DE–Inkarnation, so ist die O zugeordnete Auswertungsliste zu aktualisieren, wenn ein Akteursphärenverwalter bei Auswertung eines Zugriffsrestriktionsausdrucks auf O zugreift.

Wie in Abschnitt 6.2.2.5 erklärt, muß ein Akteursphärenverwalter bei Auswertung eines Zugriffsrestriktionsausdrucks jeweils vor Auswertung eines Teilausdrucks Lese–Sperren für die Objekte erwerben, auf die bei Auswertung des Teilausdrucks zugegriffen wird. Diese Lese–Sperren werden durch `AcquireLock`–Aufrufe auf den Akteursphärenverwaltern, die die jeweiligen Objekte verwalten, erworben. Ruft also ein Akteursphärenverwalter $ASV1$ auf einem Akteursphärenverwalter $ASV2$ die Operation `AcquireLock` zum Erwerb einer Lese–Sperre für das Objekt O auf, so kann $ASV2$ daraus folgern, daß $ASV1$ einen Zugriffsrestriktionsausdruck auswertet, dessen Wert von dem Wert des Objekts O abhängig ist. Dementsprechend

aktualisiert der Akteursphärenverwalter $ASV2$ am Ende der Ausführung von `AcquireLock` die von ihm verwaltete Auswertungsliste des Objekts O. Um einen entsprechenden Eintrag in der Auswertungsliste vornehmen zu können, sind bei Aufruf der Operation `AcquireLock` folgende zusätzliche Informationen als Parameter von $ASV1$ an $ASV2$ zu übergeben:

- der Identifikator des aufrufenden Akteursphärenverwalters $ASV1$;
- der Identifikator der zugriffskontrollierten Komponente und der Identifikator der äußeren Operation dieser Komponente, deren Zugriffsrestriktionsausdruck der Akteursphärenverwalter $ASV1$ bei Aufruf von `AcquireLock` auswertet.

Bei Änderung eines Objekts O, von dessen Zustand der Wert von Zugriffsrestriktionsausdrücken abhängig ist, wird anhand der Auswertungsliste des Objekts O ermittelt, welche Akteursphärenverwalter über diese Änderung zu informieren sind. Voraussetzung dafür ist jedoch, daß der Akteursphärenverwalter, der die Auswertungsliste des Objekts O verwaltet, Kenntnis davon erlangt, daß sich der Wert von O ändert. Diese Kenntnis erlangt der Akteursphärenverwalter dadurch, daß vor Ausführung einer Schreiboperation auf dem Objekt O eine Schreib–Sperre zu erwerben ist. Zum Erwerb dieser Schreib–Sperre wird die Operation `AcquireLock` auf dem Akteursphärenverwalter aufgerufen, der das Objekt O verwaltet. Anhand dieses `AcquireLock`–Aufrufs erkennt der Akteursphärenverwalter, daß eine Schreiboperation auf dem Objekt O bevorsteht. Nach Erwerb einer Schreib–Sperre durch Aufruf der Operation `Acquire` auf der dem Objekt O assoziierten Leser–Schreiber–Sperre überprüft der Akteursphärenverwalter im Rahmen der Ausführung von `AcquireLock` die Auswertungsliste des Objekts O. Ist diese leer, so terminiert die Ausführung von `AcquireLock` mit der Übergabe der Schreib–Sperre an den aufrufenden Akteur bzw. Akteursphärenverwalter. Anderenfalls wird auf allen Akteursphärenverwaltern, für die ein Eintrag in der Auswertungsliste des Objekts O existiert, die Operation `StartTicketDeletion` (siehe Tabelle 6.8) aufgerufen. Sei im weiteren $ASV1$ ein solcher Akteursphärenverwalter. Dann werden bei Aufruf der Operation `StartTicketDeletion` auf $ASV1$ alle Paare *(ComponentId, OperationId)*, die in der Liste enthalten sind, auf die der Eintrag für $ASV1$ in der Auswertungsliste von O verweist, an $ASV1$ übergeben. Nach Terminierung aller `StartTicketDeletion`–Aufrufe werden alle Einträge aus der Auswertungsliste von O gelöscht. Anschließend terminiert die Ausführung von `AcquireLock` mit der Übergabe der Schreib–Sperre an den aufrufenden Akteur bzw. Akteursphärenverwalter.

Ist das Objekt O eine Zugriffskontrolliste, so werden die Maßnahmen zum Löschen von Tickets nur dann initiiert, wenn die Änderung darin besteht, daß ein positiver Eintrag aus der Zugriffskontrolliste gelöscht werden soll oder ein negativer Eintrag in diese eingefügt werden soll. Nur in diesen Fällen ist es möglich, daß ein Akteur, der ein Ticket für eine Operation besitzt, deren Zugriffsrestriktionsausdruck von dieser Zugriffskontrolliste abhängig ist, aufgrund der Änderung der Zugriffkontrolliste nicht mehr das Recht zur Ausführung der Operation hat.

Operation	**Funktionalität**	**Aufruf durch**
`StartTicketDeletion`	Initiieren des Löschens von Tickets	ASV
`DeleteTicket`	Löschen eines Tickets	ASV

Tabelle 6.8: Schnittstellenoperationen der Akteursphärenverwalter zum Löschen von Tickets

Die Ausführung der Operation `StartTicketDeletion` durch einen Akteursphärenverwalter bewirkt folgendes. Für jedes Paar *(ComponentId, OperationId)*, bestehend aus dem Kom-

ponentenidentifikator *ComponentId* und dem Operationsidentifikator *OperationId*, das der Akteursphärenverwalter mit dem Aufruf von `StartTicketDeletion` erhält, wird überprüft, ob in der Ticketausstellungsliste der mit *ComponentId* identifizierten zugriffskontrollierten Komponente ein Eintrag für die Operation *OperationId* vorhanden ist. Ist kein Eintrag für die Operation *OperationId* vorhanden, so besagt dies, daß keine Tickets, die zur Ausführung von *OperationId* berechtigen, ausgestellt wurden. Anderenfalls verweist der Eintrag auf eine Liste von Paaren der Form *(TicketId, VerwalterId)*. Für jedes dieser Paare wird nun die Operation `DeleteTicket` (siehe Tabelle 6.8) auf dem Akteursphärenverwalter $ASV2$, der mit *VerwalterId* identifiziert wird, aufgerufen. Dabei wird die *TicketId* als Parameter übergeben. Die Ausführung von `DeleteTicket` durch $ASV2$ bewirkt, daß das Ticket mit dem Ticket–Identifikator *TicketId* aus der von $ASV2$ verwalteten Ticketliste gelöscht wird. Sind auf diese Weise alle Tickets, die zur Ausführung von *OperationId* berechtigen, gelöscht, kann der Eintrag für die Operation *OperationId* aus der Ticketausstellungsliste der Komponente *ComponentId* ebenfalls gelöscht werden.

Die Maßnahmen, die zum Löschen von Tickets führen, werden im Rahmen der Ausführung der Operation `AcquireLock` durchgeführt, sofern diese zum Erwerb einer Schreib–Sperre für ein Objekt, von dem die Auswertung von Zugriffsrestriktionsausdrücken abhängig ist, aufgerufen wird. Der Vorteil dieser Vorgehensweise besteht darin, daß damit *vor* Ausführung der Schreiboperation auf dem Objekt und damit *vor* einer möglichen Wertänderung des Objekts alle Tickets gelöscht werden, die zur Ausführung von Operationen berechtigen, deren Zugriffsrestriktionsausdruck von diesem Objekt abhängig ist. Dadurch wird gewährleistet, daß kein Ticket mehr existieren kann, das zur Ausführung einer solchen Operation berechtigt, obwohl der Akteur, für den das Ticket ausgestellt wurde, aufgrund der Wertänderung des Objekts nicht mehr das Recht zur Ausführung der Operation hat. In diesem Sinne werden Rechteänderungen also sofort wirksam. Würden die Maßnahmen zum Löschen der Tickets erst nach der Ausführung der Schreiboperation durchgeführt, z.B. im Rahmen der Ausführung der Operation zur Freigabe der Schreib–Sperre, bestände die Gefahr, daß ein Ticket noch eingesetzt würde, obwohl der entsprechende Akteur nicht mehr zur Ausführung der jeweiligen Operation berechtigt wäre.

Im folgenden wird das erklärte Verfahren zum Löschen von Tickets an einem Beispiel, in dem eine Rechterücknahme durch Entfernen eines Eintrags aus einer Zugriffskontrolliste durchgeführt wird, verdeutlicht.

Beispiel

Als Beispiel wird wiederum eine Situation aus dem Kontenverwaltungssystem betrachtet. Es wird angenommen, daß ein Benutzerrepräsentant BL, der für den Bankleiter erzeugt wurde, die Operation `ChangeACL` auf einem Konto–Depot K aufruft, um einem Kundenbetreuer $KB1$ durch Entfernen des entsprechenden Zugriffskontrollisteneintrags das Recht zur Ausführung der Operation `LeseKontostand` auf dem Konto K zu entziehen. Es wird davon ausgegangen, daß der Kundenbetreuer $KB1$ vor diesem `ChangeACL`–Aufruf das Recht zur Ausführung von `LeseKontostand` auf K hat und ein Benutzerrepräsentant, der für $KB1$ erzeugt wurde, ein Ticket besitzt, das zur Ausführung von `LeseKontostand` auf dem Konto K berechtigt.

Die Maßnahmen, die bei Ausführung des `ChangeACL`–Aufrufs zum Löschen des an den Benutzerrepräsentanten $KB1$ vergebenen Tickets führen, sind in Abbildung 6.6 skizziert. Im oberen Teil der Abbildung ist der realisierte Benutzerrepräsentant BL des Bankleiters zusammen mit seinem Akteursphärenverwalter $ASV(BL)$ dargestellt. Als Bestandteil der Komponentenbeschreibung von BL verwaltet $ASV(BL)$ eine Ticketliste, die ein Ticket enthält, das zur Ausführung der Operation `ChangeACL` (identifiziert mit dem Identifikator `ID_CA`) auf dem

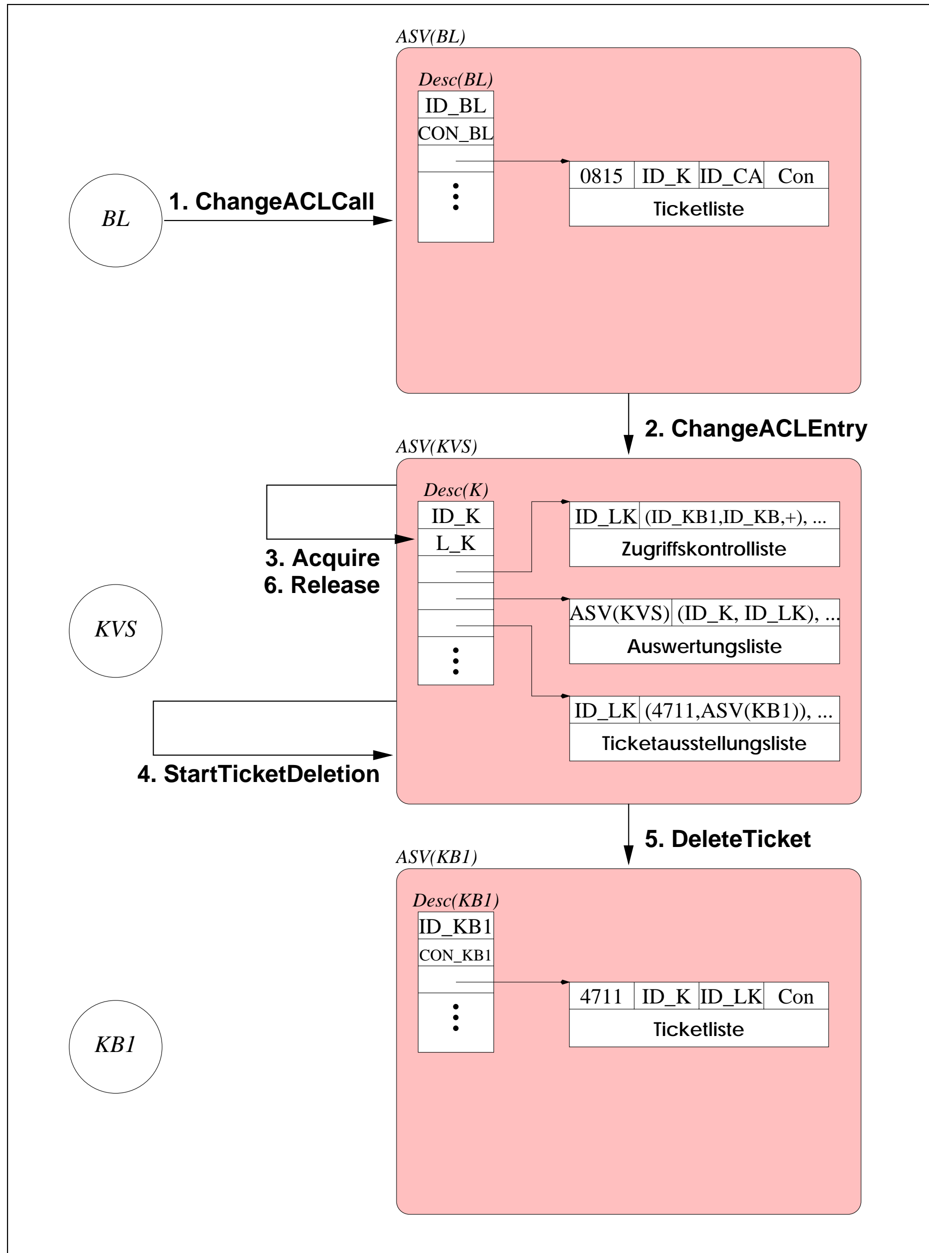

Abbildung 6.6: Beispiel für das Löschen eines Tickets

Konto–Depot K (identifiziert mit dem Identifikator `ID_K`) berechtigt. Der mittlere Teil der Abbildung zeigt den Akteursphärenverwalter $ASV(KVS)$ der realisierten Hauptkomponente KVS, von der alle Konto–Depots lebenszeitmäßig abhängig sind. $ASV(KVS)$ verwaltet die Komponentenbeschreibung des Konto–Depots K. Die Komponentenbeschreibung von K enthält insbesondere Verweise auf die Zugriffskontrolliste von K, die Auswertungsliste dieser Zugriffskontrolliste sowie die Ticketausstellungsliste von K. Die Zugriffskontrolliste enthält in der Benutzer–Rollen–Liste der Operation `LeseKontostand` (identiziert mit dem Identifikator `ID_LK`) einen positiven Benutzer–Rollen–Eintrag bestehend aus dem Identifikator `ID_KB1` des Kundenbetreuers $KB1$ und dem Identifikator `ID_KB` der Rolle Kundenbetreuer. Dies sei der Eintrag, der durch den Aufruf der Operation `ChangeACL` auf dem Konto–Depot K aus der Zugriffskontrolliste von K gelöscht werden soll. Der Eintrag in der dieser Zugriffskontrolliste

zugeordneten Auswertungsliste besagt, daß der Akteursphärenverwalter $ASV(KVS)$ bei Auswertung des Zugriffsrestriktionsausdrucks der Operation `LeseKontostand` des Konto–Depots K auf die Zugriffskontrolliste von K zugegriffen hat. In der Ticketausstellungsliste des Konto–Depots K existiert schließlich ein Eintrag, in dem vermerkt ist, daß $ASV(KVS)$ ein Ticket mit dem Ticket–Identifikator 4711, das zur Ausführung der Operation `LeseKontostand` auf K berechtigt, an den Akteursphärenverwalter $ASV(KB1)$ des Benutzerrepräsentanten des Kundenbetreuers $KB1$ vergeben hat. Im unteren Teil der Abbildung ist dieser Akteursphärenverwalter $ASV(KB1)$ des Benutzerrepräsentanten, der für den Kundenbetreuer $KB1$ erzeugt wurde, skizziert. Die Ticketliste, die von $ASV(KB1)$ als Bestandteil der Komponentenbeschreibung von $KB1$ verwaltet wird, enthält das Ticket mit dem Ticket–Identifikator 4711.

Der Aufruf der Operation `ChangeACL` auf dem Konto–Depot K durch den Benutzerrepräsentanten BL wird in insgesamt sechs Schritten realisiert, die im folgenden angegeben sind.

1. Zur Realisierung des `ChangeACL`–Aufrufs ruft der Benutzerrepräsentant BL die Operation `ChangeACLCall` auf seinem Akteursphärenverwalter $ASV(BL)$ auf. Gemäß dem in Abschnitt 6.2.2.4 Erklärten wird dabei insbesondere der Identifikator des Konto–Depots K an $ASV(BL)$ übergeben. $ASV(BL)$ überprüft zunächst, ob in der von ihm verwalteten Ticketliste ein gültiges Ticket vorhanden ist, das zur Ausführung der Operation `ChangeACL` auf dem Depot K berechtigt. Dies ist gemäß Voraussetzung der Fall.

2. Da der Akteursphärenverwalter $ASV(BL)$ die Komponentenbeschreibung des Konto–Depots K nicht selbst verwaltet, ruft $ASV(BL)$ anschließend die Operation `ChangeACLEntry` auf dem Akteursphärenverwalter $ASV(KVS)$ auf, der die Zugriffskontrolliste von K verwaltet.

3. Im Rahmen der Ausführung der Operation `ChangeACLEntry` ruft $ASV(KVS)$ zunächst zum Erwerb einer Schreib–Sperre die Operation `Acquire` auf der von ihm verwalteten Leser–Schreiber–Sperre `L_K` der Zugriffskontrolliste von K auf.

4. Nach Erwerb der Schreib–Sperre für die Zugriffskontrolliste von K ruft $ASV(KVS)$ auf allen Akteursphärenverwaltern, für die ein Eintrag in der Auswertungsliste dieser Zugriffskontrolliste enthalten ist, die Operation `StartTicketDeletion` auf, um die Verwalter über die bevorstehende Änderung der Zugriffskontrolliste zu informieren. In der hier betrachteten Situation existiert lediglich ein Eintrag in der Auswertungsliste und zwar für $ASV(KVS)$ selbst. Die Operation `StartTicketDeletion` wird also von $ASV(KVS)$ selbst ausgeführt. Die Ausführung von `StartTicketDeletion` durch $ASV(KVS)$ bewirkt, daß die Ticketausstellungsliste des Konto–Depots K nach einem Eintrag für die Operation `LeseKontostand` durchsucht wird. Der Eintrag in der Ticketausstellungsliste für die Operation `LeseKontostand` enhält Informationen über die Tickets, die zu löschen sind. In diesem Fall ist lediglich ein Ticket zu löschen, und zwar das Ticket mit dem Ticket–Identifikator 4711, das an den Akteursphärenverwalter $ASV(KB1)$ vergeben wurde.

5. Zum Löschen dieses Tickets ruft $ASV(KVS)$ die Operation `DeleteTicket` auf dem Akteursphärenverwalter $ASV(KB1)$ auf. Die Ausführung von `DeleteTicket` bewirkt, daß das Ticket mit dem Identifikator 4711 aus der Ticketliste des Benutzerrepräsentanten $KB1$ gelöscht wird.

6. Nach Terminierung des `DeleteTicket`–Aufrufs löscht $ASV(KVS)$ die Einträge für die Operation `LeseKontostand` aus der Ticketausstellungsliste des Konto–Depots K. Damit terminiert die Ausführung von `StartTicketDeletion`. Nach Terminierung der Ope-

ration `StartTicketDeletion` werden alle Einträge aus der Auswertungsliste der Zugriffskontrolliste des Konto–Depots K von $ASV(KVS)$ gelöscht. Anschließend wird die Änderung der Zugriffskontrolliste von K durchgeführt, die hier darin besteht, daß der positive Benutzer–Rollen–Eintrag `(ID_KB1,ID_KB,+)` aus der Benutzer–Rollen–Liste der Operation `LeseKontostand` zu löschen ist. Ist dieser Eintrag gelöscht, wird die unter 3. erworbene Schreib–Sperre durch Aufruf der Operation `Release` auf der Leser–Schreiber–Sperre `L_K` der Zugriffskontrolliste freigegeben. Damit terminieren die Aufrufe der Operationen `ChangeACLEntry` und `ChangeACLCall`, womit der „abstrakte“ Aufruf der Operation `ChangeACL` auf dem Konto–Depot K ausgeführt ist.

◇

Mit dem Beispiel wird die Erklärung der Funktionalität, die die Akteursphärenverwalter zur Verwaltung von Tickets zur Verfügung stellen, abgeschlossen. Bevor im weiteren Kriterien angegeben werden, von denen die Entscheidung über die Vergabe eines Tickets abhängig ist, wird zunächst noch eine alternative Realisierungsmöglichkeit für die Verwaltung von Tickets durch die Akteursphärenverwalter vorgestellt und diskutiert.

6.2.3.4 Realisierungsalternative für die Ticketverwaltung

Das bisher beschriebene Verfahren zur Ticketverwaltung ist dadurch charakterisiert, daß ein Ticket, das einen Akteur zur Ausführung einer äußeren Operation einer zugriffskontrollierten Komponente berechtigt, von dem Akteursphärenverwalter ausgestellt wird, der die zugriffskontrollierte Komponente verwaltet, und von dem Akteursphärenverwalter gespeichert wird, der dem Akteur zugeordnet ist. Alternativ könnte das Ticket auch von dem Akteursphärenverwalter gespeichert werden, der das Ticket ausstellt. In beiden Fällen ist das Ticket bei Zustandsänderungen, die die Ungültigkeit des Tickets zur Folge haben, aus der jeweiligen Ticketliste zu löschen.

Die Speicherung der Tickets durch die Akteursphärenverwalter der Akteure, für die die Tickets ausgestellt wurden, hat den Vorteil, daß ein Akteursphärenverwalter bei einem Aufruf einer äußeren Operation einer zugriffskontrollierten Komponente durch seinen Akteur autonom, also ohne Interaktion mit dem Akteursphärenverwalter der zugriffskontrollierten Komponente, überprüfen kann, ob ein gültiges Ticket für diese Operation vorliegt. Im Fall der Speicherung der Tickets durch die ausstellenden Akteursphärenverwalter ist dies nicht möglich. Hier muß sich der Akteursphärenverwalter des aufrufenden Akteurs an den Akteursphärenverwalter wenden, dem die zugriffskontrollierte Komponente zugeordnet ist. Erst dieser kann dann überpüfen, ob ein gültiges Ticket für den Akteur vorhanden ist.

Die Tickets wurden mit dem Ziel eingeführt, den Aufwand für die bei einem Zugriff auf eine zugriffskontrollierte Komponente durchzuführende Zugriffskontrolle zu reduzieren. Aus diesem Grund sollte bei Vorhandensein eines Tickets nach Möglichkeit jede zusätzliche Verwalterkooperation vermieden werden. Unter diesem Aspekt ist somit das ausführlich beschriebene Verfahren zur Ticketverwaltung der oben angesprochenen Alternative vorzuziehen.

Die Vorteile der alternativen Realisierungsmöglichkeit der Ticketverwaltung liegen in dem geringerem Aufwand, der zum Löschen von Tickets zu leisten ist. Während in der bisher beschriebenen Lösung der Akteursphärenverwalter, der das Ticket ausgestellt hat, zum Löschen des Tickets eine Operation auf dem Akteursphärenverwalter aufrufen muß, an den er das Ticket vergeben hat, kann er das Ticket im Fall der betrachteten Alternative selbst aus seiner Ticketliste löschen.

Ein Ticket wird, wie anhand der im folgenden Abschnitt angegebenen Kriterien für die Ticketvergabe noch deutlich wird, nur dann an einen Akteur vergeben, wenn das Ticket zum einen von dem Akteur potentiell häufig eingesetzt werden kann und zum anderen potentiell eine hohe Gültigkeitsdauer hat. Dies hat zur Folge, daß in einem System die Anzahl der Ticketnutzungen im allgemeinen die Anzahl der Ticketlöschungen deutlich übersteigt. Somit überwiegt der Vorteil des ausführlich vorgestellten Ticketverwaltungsverfahrens, der in dem geringerem Aufwand für die bei einem Operationsaufruf durchzuführende Zugriffskontrolle besteht, gegenüber dem Vorteil des geringeren Aufwands zum Löschen von Tickets bei dem alternativen Verfahren.

6.2.3.5 Kriterien für die Vergabe eines Tickets

In Abschnitt 6.2.3.1 über die Vergabe von Tickets wurde darauf hingewiesen, daß die von einem Akteursphärenverwalter zu treffende Entscheidung darüber, ob ein Ticket, das zur Ausführung der Operation OP einer zugriffskontrollierten Komponente Z berechtigt, an einen Akteur A vergeben wird, von zahlreichen Faktoren abhängig ist. In diesem Abschnitt werden nun die wesentlichen Kriterien angegeben und diskutiert, die die Entscheidung über die Vergabe eines Tickets beeinflussen. Der Schwerpunkt der Erläuterungen liegt darauf, deutlich zu machen, warum die genannten Einflußgrößen bei der Entscheidung über die Vergabe eines Tickets zu berücksichtigen sind und Hinweise darauf zu geben, wie sich der Wert dieser Größen bestimmen läßt. Die Ausarbeitung konkreter Verfahren zur Bewertung der Einflußgrößen ist nicht Bestandteil dieser Arbeit, da hierfür insbesondere Erfahrungswerte benötigt werden, die nur im Rahmen einer – bisher noch nicht erfolgten – Implementierung des Ticket–Konzepts gewonnen werden können.

Die Entscheidung darüber, ob ein Ticket, das zur Ausführung einer Operation OP berechtigt, an einen Akteur A vergeben wird oder nicht, sollte von dem Ziel geleitet sein, das mit der Einführung der Tickets verfolgt wurde, nämlich den Aufwand für die durchzuführenden Zugriffskontrollen zu reduzieren. Dementsprechend ist bei einer solchen Entscheidung der zusätzliche Aufwand, der für die Verwaltung und das eventuelle Löschen des Tickets zu leisten ist, gegenüber der Aufwandsreduzierung abzuwägen, die sich möglicherweise daraus ergibt, daß bei erneuten OP–Aufrufen durch A aufgrund des Vorliegens des Tickets der entsprechende Zugriffsrestriktionsausdruck nicht erneut auszuwerten ist. Die Aufgabe, die die Akteursphärenverwalter bei der Entscheidung über die Vergabe eines Tickets zu lösen haben, besteht also darin, abzuschätzen, ob sich aus der Vergabe des Tickets insgesamt eine Aufwandsreduzierung ergibt. Ob sich durch die Vergabe eines Tickets für eine Operation OP an einen Akteur A insgesamt eine Aufwandsreduzierung ergibt, ist im wesentlichen von drei Einflußgrößen abhängig, die im weiteren näher erläutert werden.

1. Die Anzahl der OP–Aufrufe, die A ausführt[17].
2. Die Komplexität des Zugriffsrestriktionsausdrucks, der für die Operation OP festgelegt ist.
3. Die Änderungshäufigkeit der Objekte, von deren Wert der Wert des Zugriffsrestriktionsausdrucks von OP abhängig ist.

[17] Im weiteren auch als Aufrufhäufigkeit bezeichnet.

Aufrufhäufigkeit

Die Anzahl der OP–Aufrufe durch A ist ein Kriterium, das die Aufwandsreduzierung dadurch beinflußt, daß bei Vorliegen eines Tickets für OP nicht der für OP festgelegte Zugriffsrestriktionsausdruck erneut auszuwerten ist. Ruft der Akteur A die Operation OP zum Beispiel nur einmal auf, so macht es wenig Sinn, an diesen Akteur ein Ticket für OP zu vergeben, da A dieses Ticket überhaupt nicht einsetzen kann. Würde in diesem Fall ein Ticket für OP an A vergeben, so wäre Aufwand für die Ausstellung und Verwaltung des Tickets zu leisten, dem keine Aufwandsersparnis gegenüberstände. Je häufiger der Akteur A jedoch die Operation OP aufruft, desto größer ist die mögliche Aufwandsersparnis, die sich daraus ergibt, daß der Zugriffsrestriktionsausdruck von OP bei Vorliegen eines gültigen Tickets für OP nicht auszuwerten ist. Als Beispiel kann hier ein Editor genannt werden, der durch den Akteur A realisiert wird und der eingegebene Zeichenfolgen in einem Depot speichert, das Operationen zum Lesen und Schreiben von Zeichenfolgen anbietet. In diesem Fall ist es sinnvoll, jeweils ein Ticket auszustellen, das den Akteur A zur Ausführung der Schreib– bzw. der Leseoperation auf dem Depot berechtigt, da davon auszugehen ist, daß diese Operationen häufig von A aufgerufen werden und Rechteänderungen bzgl. des Depots relativ selten auftreten.

Die Anzahl der Aufrufe der äußeren Operation OP einer zugriffskontrollierten Komponente, durch einen Akteur A ist im allgemeinen nur schwer abzuschätzen. Erste Hinweise können im Rahmen einer statischen Analyse gewonnen werden. So kann durch eine derartige Analyse ermittelt werden, ob der Akteur A die Operation OP lediglich einmal oder potentiell mehrfach aufruft. Tritt der Operationsaufruf in einem Schleifenkonstrukt auf, so können möglicherweise weitere Schlußfolgerungen in Bezug auf die Aufrufhäufigkeit gezogen werden. Die im Rahmen einer statischen Analyse ermittelten Informationen über die Aufrufhäufigkeit sind durch Informationen, die zur Laufzeit des Systems gewonnen werden, zu ergänzen. So kann die Anzahl der Aufrufe einer Operation, die ein Akteur innerhalb eines bestimmten Zeitintervalls ausgeführt hat, zum Beispiel dadurch bestimmt werden, daß für jede äußere Operation einer zugriffskontrollierten Komponente von dem Akteursphärenverwalter dieser Komponente für jeden Akteur, der diese Operation aufgerufen hat, ein entsprechender Zähler verwaltet wird.

Komplexität des Zugriffsrestriktionsausdrucks

Die Komplexität des Zugriffsrestriktionsausdrucks, der für die Operation OP festgelegt ist, ist die zweite Größe, die die Entscheidung über die Vergabe eines Tickets für die Operation OP an den Akteur A beeinflußt. Die Komplexität eines Zugriffsrestriktionsausdrucks wird dabei durch die Art und die Anzahl der Objekte festgelegt, von deren Werten der Wert des Zugriffsrestriktionsausdrucks abhängig ist und auf die somit bei Auswertung des Zugriffsrestriktionsausdrucks zugegriffen werden muß. Bezüglich der Art der Objekte wird zum einen unterschieden, ob es sich um DE–Inkarnationen oder um Zugriffskontrollisten oder Zugriffshistorienlisten handelt. Zum anderen wird dahingehend differenziert, ob ein solches Objekt von dem Akteursphärenverwalter verwaltet wird, der für die Auswertung des Zugriffsrestriktionsausdrucks zuständig ist, das Objekt also in diesem Sinne **lokal** zu dem Akteursphärenverwalter ist, oder von einem anderen Akteursphärenverwalter verwaltet wird und somit **global** ist.

Die Komplexität des Zugriffsrestriktionsausdrucks von OP bestimmt also den Aufwand zur Auswertung des Zugriffsrestriktionsausdrucks und damit die Aufwandsreduzierung, die sich nach Vergabe eines Tickets für OP an A daraus ergibt, daß der Zugriffsrestriktionsausdruck von OP bei erneuten OP–Aufrufen durch A nicht erneut auszuwerten ist. Besteht der Zugriffsrestriktionsausdruck von OP zum Beispiel nur aus einer Rollenrestriktion, in der überprüft wird, ob der Rollenidentifikator des OP aufrufenden Akteurs A dem Identifikator einer bestimmten Rolle entspricht, so ist zur Auswertung des Zugriffsrestriktionsausdrucks lediglich

ein einfacher Wertvergleich vorzunehmen. Sind in dem Zugriffsrestriktionsausdruck von OP hingegen mehrere `IN_ACL`– und/oder `ACCESSED`–Prädikate enthalten, so kann die Auswertung des Zugriffsrestriktionsausdrucks aufwendig sein, da zur Auswertung dieser Prädikate die Einträge von Zugriffskontrollisten bzw. Zugriffhistorienlisten überprüft werden müssen, für die zunächst entsprechende Lese–Sperren erworben werden müssen. Werden diese Zugriffskontrollisten bzw. Zugriffhistorienlisten zudem nicht von dem Akteursphärenverwalter verwaltet, der den Zugriffsrestriktionsausdruck auswertet, so sind zur Auswertung dieser Listen Schnittstellenoperationen anderer Akteursphärenverwalter aufzurufen. Enthält der Zugriffsrestriktionsausdruck von OP variable DE–Inkarnationen, die global zu dem Akteursphärenverwalter sind, der den Zugriffsrestriktionsausdruck auswertet, so sind zum Erwerb und zur Freigabe von Lese–Sperren für diese DE–Inkarnationen ebenfalls Schnittstellenoperationen anderer Akteursphärenverwalter aufzurufen.

Die Anzahl und Art der Objekte, von deren Zustand der Wert des Zugriffsrestriktionsausdrucks von OP abhängig ist, bestimmt allerdings nicht nur die Aufwandsreduzierung, die sich daraus ergibt, daß bei Vorliegen eines Tickets für OP dieser Zugriffsrestriktionsausdruck nicht erneut auszuwerten ist, sondern auch den Aufwand, der für die Verwaltung der Informationen zu leisten ist, die für das Löschen vergebener Tickets benötigt werden. Werden Tickets für die Operation OP vergeben, so muß für jedes variable Objekt des Zugriffsrestriktionsausdrucks von OP eine Auswertungsliste verwaltet werden. Je größer die Anzahl der variablen Objekte ist, desto höher ist also auch die Anzahl der zu verwaltenden Auswertungslisten.

Die Komplexität eines Zugriffsrestriktionsausdrucks kann statisch bestimmt werden, indem die einzelnen Objekte, von deren Zustand der Wert des Zugriffsrestriktionsausdrucks abhängig ist, unterschiedlich gewichtet werden. Die Summe der Gewichte der einzelnen Objekte ergibt dann die Komplexität des Zugriffsrestriktionsausdrucks. In Tabelle 6.9 ist angegeben, wie die Objekte abhängig von ihrer Art generell zu gewichten sind.

<table>
<tr><th>Objektart</th><th>Gewichtung</th></tr>
<tr><td>lokale konstante DE–Inkarnation</td><td rowspan="2">sehr niedrig</td></tr>
<tr><td>globale konstante DE–Inkarnation</td></tr>
<tr><td>lokale variable DE–Inkarnation</td><td>niedrig</td></tr>
<tr><td>lokale Zugriffskontrolliste</td><td rowspan="2">mittel</td></tr>
<tr><td>lokale Zugriffshistorienliste</td></tr>
<tr><td>globale variable DE–Inkarnation</td><td>hoch</td></tr>
<tr><td>globale Zugriffskontrolliste</td><td rowspan="2">sehr hoch</td></tr>
<tr><td>globale Zugriffshistorienliste</td></tr>
</table>

Tabelle 6.9: Gewichtung von Objekten, die in Zugriffsrestriktionsausdrücken auftreten können

Ein konstante DE–Inkarnation ist unabhängig davon, ob sie lokal oder global zu einem Akteursphärenverwalter ist, mit einem sehr niedrigen Gewicht zu belegen, da für sie bei Auswertung eines Zugriffsrestriktionsausdrucks im Gegensatz zu einer variablen DE–Inkarnation keine Lese–Sperre zu erwerben ist[18]. Dementsprechend ist eine variable DE–Inkarnation höher zu gewichten. Die unterschiedliche Gewichtung für den lokalen und globalen Fall ist darin begründet, daß in letzterem Fall zum Erwerb und zur Freigabe der jeweiligen Lese–Sperre Operationen eines anderen Akteursphärenverwalters aufgerufen werden müssen. Zugriffskontrol-

[18]Zudem erfolgt ein Zugriff auf eine konstante DE–Inkarnation aufgrund ihrer Repräsentation ohne einen gesonderten Speicherzugriff.

listen und Zugriffshistorienlisten sind jeweils höher als die DE–Inkarnationen zu gewichten, da ihre Auswertung aufwendiger ist als der Zugriff auf eine DE–Inkarnation.

Änderungshäufigkeit der Objekte

Die dritte Größe, die die Entscheidung darüber beeinflußt, ob ein Ticket für die Operation OP an den Akteur A vergeben wird, ist die Änderungshäufigkeit der Objekte, von deren Wert der Wert des Zugriffsrestriktionsausdrucks von OP abhängig ist. Die Änderungshäufigkeit dieser Objekte bestimmt die durchschnittliche Gültigkeitsdauer der Tickets, die zur Ausführung der Operation OP berechtigen, und damit auch den Aufwand, der für das Löschen dieser Tickets bei Wertänderungen dieser Objekte zu leisten ist.

Ist der Wert des Zugriffsrestriktionsausdrucks von OP, abgesehen von den Werten des `Caller`–Attributs, zum Beispiel nur von Konstanten abhängig, so braucht ein für die Operation OP vergebenes Ticket „nie“ gelöscht zu werden, da bzgl. der Operation OP keine Rechteänderungen möglich sind. Das Ticket wird erst dann (implizit) gelöscht, wenn der Akteur, an den es vergeben wurde, aufgelöst wird. In diesem Fall entsteht also kein Aufwand für die Verwaltung der Datenstrukturen, in denen sonst die für das explizite Löschen von Tickets bei Wertänderungen der Objekte benötigten Informationen gehalten werden. Ein Beispiel stellt ein Zugriffsrestriktionsausdruck dar, der lediglich aus einer Rollenrestriktion besteht, die als <*role-identifier*> den Bezeichner einer Rolle enthält.

Ist der Wert des Zugriffsrestriktionsausdrucks von OP jedoch von Objekten abhängig, deren Wert sich häufig ändert, so ist die Gültigkeitsdauer eines für die Operation OP vergebenen Tickets im allgemeinen recht kurz. In diesem Fall entsteht Aufwand dadurch, daß bei jeder Wertänderung eines solchen Objekts alle Tickets für OP, die zwischen dem Zeitpunkt der letzten Wertänderung eines der Objekte und dem Zeitpunkt der aktuellen Wertänderung vergeben wurden, zu löschen sind. Ob sich dieser Aufwand „lohnt“, ist davon abhängig, wie häufig die in diesem Zeitintervall vergebenen Tickets genutzt wurden. Die Anzahl der möglichen Ticketnutzungen ergibt sich aus der ersten bereits diskutierten Einflußgröße, nämlich der Anzahl der OP–Aufrufe, die ein Akteur ausführt.

Wichtige Informationen darüber, ob die Vergabe eines Tickets für die Operation OP unter dem Aspekt der Änderungshäufigkeit der Objekte, von deren Zustand der Wert des Zugriffsrestriktionsausdrucks von OP abhängig ist, sinnvoll ist, können im Rahmen einer statischen Analyse ermittelt werden. So kann statisch festgestellt werden, ob der Wert des Zugriffrestriktionsausdrucks von OP lediglich von konstanten Werten abhängig ist. Ist dies der Fall, so ist nach dem oben Gesagten die Vergabe eines Tickets für die Operation OP stets sinnvoll. Weiterhin kann statisch analysiert werden, ob der Wert des Zugriffsrestriktionsausdrucks von OP neben konstanten Werten lediglich von Zugriffskontrollisten abhängig ist. Unter der Annahme, daß die Einträge in einer Zugriffskontrolliste relativ selten geändert werden, da diese Änderungen explizit von Benutzern vorgenommen werden müssen, kann gefolgert werden, daß die Vergabe eines Tickets für die Operation OP auch in diesem Fall sinnvoll ist. Ein Spezialfall liegt dann vor, wenn der Zugriffsrestriktionsausdruck von OP von den Werten aktueller Parameter der Operation OP abhängig ist. In diesem Fall, der ebenfalls statisch analysiert werden kann, darf kein Ticket für die Operation vergeben werden, da der Zugriffsrestriktionsausdruck bei jedem OP–Aufruf erneut auszuwerten ist. Ist der Wert des Zugriffsrestriktionsausdrucks von OP von Zugriffshistorienlisten und variablen DE–Inkarnationen abhängig, so ist eine Ticketvergabe lediglich dann sinnvoll, wenn die Änderungshäufigkeit jedes dieser Objekte einen gewissen Grenzwert nicht überschreitet. Die Änderungshäufigkeit einer variablen DE–Inkarnation kann als die Anzahl der Schreiboperationen oder als das Verhältnis der Anzahl der Schreib– zu der Anzahl der Leseoperationen (Schreib/Lese–Rate), die jeweils in einem bestimmten Zeitintervall auf dieser DE–Inkarnation ausgeführt werden,

definiert werden. Die Änderungshäufigkeit einer Zugriffshistorienliste ist von der Anzahl der Aufrufe äußerer Operationen auf der Komponente abhängig, der diese Zugriffshistorienliste zugeordnet ist. Erste Informationen über die Anzahl der Schreib– bzw. Leseoperationen, die auf einer DE–Inkarnation ausgeführt werden, bzw. über die Anzahl der äußeren Operationen, die auf einer Komponente aufgerufen werden, können im Rahmen statischer Analysen gewonnen werden. Aus diesen Analysen können im allgemeinen jedoch nur recht grobe Aussagen über die Aufrufhäufigkeit einer solchen Operation abgeleitet werden, wie zum Beispiel: „Die Operation wird lediglich einmal oder potentiell mehrfach aufgerufen". Um genauere Aussagen über die Änderungshäufigkeit eines Objekts machen zu können, sind die im Rahmen der statischen Analyse ermittelten Informationen durch Informationen, die zur Laufzeit des Systems gewonnen werden, zu ergänzen. Für jedes dieser Objekte können zum Beispiel von dem Akteursphärenverwalter, der das Objekt verwaltet, Zähler geführt werden, die immer dann zu inkrementieren sind, wenn eine Schreib– bzw. Leseoperation auf dem Objekt ausgeführt wird. Der aktuelle Wert dieser Zähler bzw. die aus diesen abgeleitete Schreib/Lese–Rate kann im Rahmen der Auswertung eines Zugriffsrestriktionsausdrucks an den den Ausdruck auswertenden Akteursphärenverwalter übergeben werden, der dann diese Information in seine Entscheidung über die Vergabe eines Tickets einfließen lassen kann.

Damit sind die Faktoren, die die Entscheidung über die Vergabe eines Tickets beinflussen, erklärt. Es wurden Hinweise gegeben, wie diese Einflußgrößen bewertet werden können. So wurde zum Beispiel für die Bestimmung der Komplexität eines Zugriffsrestriktionsausdrucks angegeben, wie die einzelnen Objekte, von deren Zustand der Wert des Zugriffsrestriktionsausdrucks abhängig ist, generell zu gewichten sind. Wesentliche Informationen lassen sich durch statische und dynamische Analysen ermitteln und auswerten. Es ist jedoch offen geblieben, welche konkreten Werte für die o.g. Gewichte anzugeben sind und anhand welcher Kriterien aus den Ergebnissen, die die statischen Analysen liefern, ermittelt werden kann, für welche Komponenten bzw. Objekte ergänzende dynamische Analysen durchzuführen sind. Die angegebenen Hinweise zur Bestimmung der Werte der genannten Einflußgrößen sind deshalb in zukünftigen Arbeiten zu konkret anwendbaren Verfahren für die Bewertung dieser Kriterien zu verfeinern und zu präzisieren.

Entscheidungsstrategie für die Ticketvergabe

Die Aufgabe der Akteursphärenverwalter besteht darin, auf Basis der drei erläuterten Einflußgrößen nach Auswertung des Zugriffsrestriktionsausdrucks einer Operation zu entscheiden, ob die Vergabe eines Tickets, an den die Operation aufrufenden Akteur sinnvoll ist. Abgesehen von den erwähnten Sonderfällen, in denen die Entscheidung über die Ticketvergabe bereits statisch getroffen werden kann, ist hierzu von den Akteursphärenverwaltern eine geeignete **Strategie** durchzuführen. Die detaillierte Ausarbeitung einer Strategie, nach der die Akteursphärenverwalter auf Basis der erläuterten Einflußgrößen die Entscheidung über die Vergabe eines Tickets treffen können, geht über den Rahmen dieser Arbeit hinaus. Für die Entwicklung einer solchen Strategie werden Erfahrungswerte benötigt, die es insbesondere ermöglichen, die im Rahmen der Strategie durchzuführenden Aufwandsabschätzungen vorzunehmen. Als Basis hierfür werden u.a. Werte für den Aufwand einer Ticketlöschung (z. B. der hierfür benötigte Zeitbedarf), für die zur Ticketverwaltung erforderlichen Datenstrukturen (Ticketliste, Ticketausstellungsliste, Auswertungsliste) oder für einen Verwalteraufruf benötigt. Diese Erfahrungswerte können lediglich durch eine derzeit noch nicht vorliegende prototypische Realisierung des Ticket–Konzepts innerhalb einer INSEL$^+$–Laufzeitumgebung gewonnen werden.

Als Basis für die Entwicklung einer Strategie für die Ticketvergabe kann allerdings die in Tabelle 6.10 dargestellte Matrix dienen, die im weiteren als **Entscheidungsmatrix** bezeichnet

wird. In der Entscheidungsmatrix ist angegeben, ob die Vergabe eines Tickets abhängig von einem jeweils niedrigen bzw. hohen Wert der drei erläuterten Einflußgrößen sinnvoll ist oder nicht. Der Eintrag '+' in einem Feld der Matrix besagt, daß die Ticketvergabe sinnvoll ist. Im Falle eines '−' sollte kein Ticket vergeben werden. Bei einem '(−)' erscheint eine Ticketvergabe ebenfalls generell nicht sinnvoll; in Ausnahmefällen, die abhängig von den konkreten Werten der Einflußgrößen sind, kann die Vergabe eines Tickets jedoch sinnvoll sein. Ein '?' in einem Matrixfeld besagt, daß keine allgemeine Aussage darüber möglich ist, ob ein Ticket zu vergeben ist oder nicht. In diesen und in allen Fällen, in denen die Einflußgrößen eher mittlere Werte haben und damit wenig differieren, ist die Strategie besonders „gefordert". Hier muß auf Basis der konkreten Werte der Einflußgrößen abgeschätzt werden, ob durch die Vergabe eines Tickets eine Aufwandsreduzierung möglich ist.

		Komplexität			
		niedrig	hoch	niedrig	hoch
Anzahl	niedrig	?	+	−	(−)
OP–**Aufrufe**	hoch	+	+	(−)	?
		niedrig		hoch	
		Änderungshäufigkeit			

Tabelle 6.10: Entscheidungsmatrix für die Vergabe eines Tickets

Die Vergabe eines Tickets in den Fällen, die in der Entscheidungsmatrix mit einem '+' gekennzeichnet sind, ist dadurch motiviert, daß die Änderungshäufigkeit der Objekte, von deren Werten der Wert des entsprechenden Zugriffsrestriktionsausdrucks abhängig ist, niedrig ist. Damit ist auch die Wahrscheinlichkeit, daß ein vergebenes Ticket längere Zeit gültig ist und damit häufig eingesetzt werden kann, relativ hoch. Für den Fall, in dem sowohl die Anzahl der Operationsaufrufe als auch die Komplexität des Zugriffsrestriktionsausdrucks hoch ist, ist die Entscheidung für die Vergabe eines Tickets bei geringer Änderungshäufigkeit der Objekte besonders naheliegend. Hier ist davon auszugehen, daß die Aufwandsersparnis, die sich daraus ergibt, daß der Zugriffsrestriktionsausdruck bei der überwiegenden Anzahl der Operationsaufrufe nicht auszuwerten ist, den zusätzlichen Aufwand, der für das Löschen der Tickets bei Änderungen zu leisten ist, bei weitem überwiegt. Für den Fall, daß alle drei Einflußgrößen einen niedrigen Wert haben, kann eine solche allgemeine Aussage nicht gemacht werden. Hier ist im einzelnen zu überprüfen, ob sich der Aufwand für die Verwaltung des Tickets bei der niedrigen Anzahl der Operationsaufrufe und der niedrigen Komplexität des Zugriffsrestriktionsausdrucks überhaupt lohnt.

Die Fälle, in denen laut der Entscheidungsmatrix die Vergabe eines Tickets nicht sinnvoll erscheint, sind dadurch charakterisiert, daß die Änderungshäufigkeit der relevanten Objekte hoch ist. In diesen Fällen ist die Gültigkeitsdauer eines vergebenen Tickets im allgemeinen relativ kurz, woraus sich die Gefahr ergibt, daß ein Ticket bereits kurz nach seiner Vergabe wieder zu löschen ist. Aus diesem Grund ist in dem Fall, daß sowohl die Anzahl der Operationsaufrufe als auch die Komplexität des Zugriffsrestriktionsausdrucks niedrig ist, davon auszugehen, daß sich durch die Vergabe eines Tickets keine Aufwandsersparnis ergibt. Die Fälle, in denen entweder die Anzahl der Operationsaufrufe oder die Komplexität des Zugriffsrestriktionsausdrucks hoch ist und der Wert der jeweils anderen Größe niedrig ist, sind hingegen etwas differenzierter zu betrachten, obwohl auch hier im allgemeinen davon

ausgegangen werden kann, daß die Gültigkeitsdauer eines Tickets zur Erzielung einer nennenswerten Aufwandsreduzierung zu kurz ist. Für den Fall, daß alle drei Einflußgrößen einen hohen Wert haben, können keine allgemein gültigen Aussagen über die mögliche Aufwandsersparnis durch eine Ticketvergabe abgeleitet werden. Hier können sich zwar bei Vorliegen eines Tickets bereits in relativer kurzer Zeit Aufwandsreduzierungen dadurch ergeben, daß der komplexe Zugriffsrestriktionsausdruck nicht ausgewertet zu werden braucht. Andererseits ist jedoch fraglich, ob die Gültigkeitsdauer des Tickets ausreicht, damit diese Aufwandsreduzierungen unter Berücksichtigung des zusätzlichen Aufwands, der für das Löschen des Tickets zu leisten ist, insgesamt zu einer Aufwandsersparnis führen.

Aus den obigen Erläuterungen der Entscheidungsmatrix folgt, daß der Änderungshäufigkeit der Objekte bei der Entscheidung über die Vergabe eines Tickets größere Bedeutung zukommt als den beiden anderen Einflußgrößen. Sie ist dementsprechend in der zu entwickelnden Strategie am stärksten zu gewichten. Abschließend sei angemerkt, daß die Möglichkeit besteht, die Strategie für die Ticketvergabe adaptiv anzulegen, da nach jedem Löschen eines Tickets bei erneutem Aufruf der Operation, zu deren Ausführung das Ticket berechtigte, erneut über die Vergabe eines Tickets entschieden werden muß. Bei dieser Entscheidung kann die „Richtigkeit“ der zuletzt getroffenen Entscheidungen, die auf Grundlage geeigneter Kriterien, wie zum Beispiel der erreichten Aufwandsersparnis, zu bewerten ist, berücksichtigt werden.

6.2.3.6 Zusammenfassung

Motiviert durch das Ziel, durch flexiblen Einsatz unterschiedlicher Realisierungsalternativen den Aufwand für die bei Zugriffen auf zugriffskontrollierte Komponenten durchzuführenden Zugriffskontrollen zu reduzieren, wurden in diesem Abschnitt Tickets eingeführt und die für ihre Verwaltung eingesetzten Verfahren erklärt. Besitzt ein Akteur bzw. dessen Akteursphärenverwalter ein gültiges Ticket, das ihn zur Ausführung einer äußeren Operation einer zugriffskontrollierten Komponente berechtigt, so braucht bei einem Aufruf dieser Operation durch den Akteur der für die Operation festgelegte Zugriffsrestriktionsausdruck nicht ausgewertet zu werden. Es wurde erklärt, wie die Akteursphärenverwalter um Fähigkeiten zur Verwaltung von Tickets, wozu insbesondere das Ausstellen und Löschen von Tickets gehört, erweitert werden können. Durch die postulierte Vertrauenswürdigkeit der Akteursphärenverwalter ist sichergestellt, daß Tickets nicht unautorisiert ausgestellt oder manipuliert werden können.

Die von einem Akteursphärenverwalter zu treffende Entscheidung über die Vergabe eines Tickets an einen Akteur ist von dem Ziel geleitet, durch die Vergabe des Tickets eine Aufwandsreduzierung zu erreichen. Die wesentlichen der Kriterien, die in diese Entscheidung einfließen, wurden angegeben und diskutiert. Auf Grundlage dieser Kriterien wurden Fälle herausgearbeitet, in denen aufbauend auf statischer Analyseinformation bereits zur Übersetzungszeit entschieden werden kann, ob die Vergabe eines Tickets unter dem Aspekt der Aufwandsreduzierung sinnvoll ist oder nicht. Für die Fälle, in denen dynamisch über die Ticketvergabe zu entscheiden ist, führen die Akteursphärenverwalter eine adaptive Strategie aus, deren wesentliche Parameter vorgestellt wurden. Die Entwicklung konkreter Verfahren zur Bewertung der Kriterien sowie die detaillierte Ausarbeitung der Strategie muß auf Basis einer prototypischen Implementierung des Ticket–Konzepts in weiterführenden Arbeiten erfolgen.

Insgesamt wurde mit den Tickets und den für ihre Verwaltung angegebenen Verfahren die Basis dafür geschaffen, die Zugriffskontrollen unter Ausnutzung statisch und dynamisch gewonnener Analyseinformationen möglichst effizient durchzuführen. Das Ticket–Konzept liefert

somit ein Beispiel für die durch die *top–down* orientierte Vorgehensweise mögliche Entwicklung alternativer Realisierungskonzepte, die eine den jeweiligen Anwendungseigenschaften flexibel anpaßbare Realisierung ermöglichen.

6.3 Realisierungsebene 2

In diesem Abschnitt werden die im vorhergehenden Abschnitt für die Realisierungsebene 1 angegebenen Konzepte und Maßnahmen zur Realisierung von INSEL$^+$–Systemen für die Realisierungsebene 2 konkretisiert. Die Realisierungsebene 2 ist charakterisiert durch eine Menge vernetzter Stellenrechner. In Abschnitt 6.3.1 werden die Eigenschaften der Realisierungsebene 2 näher erläutert und die aufgrund dieser Eigenschaften vorzunehmenden Anpassungen und Erweiterungen der Realisierungskonzepte und –maßnahmen motiviert. Die Notwendigkeit dieser Anpassungen und Erweiterungen ergibt sich vor allem aus dem Vorhandensein physikalisch getrennter Speicher und aus den unsicheren Nachrichtentransportkanälen, über die die Stellenrechner untereinander vernetzt sind. Die Verwalterarchitektur und die grundlegenden Realisierungskonzepte, die für die Realisierungsebene 1 erarbeitet wurden, können jedoch unverändert übernommen werden. Abschnitt 6.3.2 beschreibt die sich aus dieser Übernahme ergebenden Konsequenzen hinsichtlich erforderlicher Konkretisierungen der Verwalter und erklärt insbesondere die Anpassung, die für die Erzeugung anonymer Depots vorzunehmen ist. In Abschnitt 6.3.3 werden die Akteursphärenverwalter dann um Fähigkeiten zur Realisierung stellenübergreifender Zugriffe auf passive Inkarnationen erweitert. Um die Sicherheit der Nachrichtenübertragung im Sinne der vertrauenlichen und authentischen Übertragung von Nachrichten über die unsicheren Transportkanäle zwischen den Stellen zu gewährleisten, werden Maßnahmen zur wechselseitigen Authentifizierung von Akteursphärenverwaltern sowie zur Verschlüsselung der zwischen diesen auszutauschenden Nachrichten eingesetzt. Diese Maßnahmen und die zur wechselseitigen Authentifizierung eingesetzten Authentifizierungsprotokolle werden in Abschnitt 6.3.4 erklärt. Der Abschnitt 6.3.5 betrachtet Optimierungsmöglichkeiten der angegebenen Realisierungsmaßnahmen. Es werden sogenannte Stellvertreter–Verwalter eingeführt, die es ermöglichen, eine Akteursphäre unter Effizienzgesichtspunkten auf mehrere Stellen verteilt zu realisieren. Effizienzgewinne werden hierbei dadurch erreicht, daß die Anzahl stellenübergreifender Aufrufe von Verwalteroperationen bei Vorhandensein lokaler Stellvertreter–Verwalter im allgemeinen reduziert wird.

6.3.1 Charakteristika

Die Realisierungsebene 2 ist charakterisiert durch eine Menge von Stellenrechnern (kurz: Stellen), die durch Nachrichtentransportkanäle untereinander verbunden sind. Über diese Nachrichtentransportkanäle kann jede Stelle mit jeder anderen Stelle Nachrichten austauschen. Eine Stelle besteht aus einem bzw. mehreren physikalischen Prozessoren sowie lokalem Hauptspeicher und lokalem Hintergrundspeicher. Darüberhinaus verfügt jede Stelle über einen **Betriebssystemkern**, der folgende Eigenschaften hat:

1. Er beinhaltet die Implementierung einer **verteilten Prozessorverwaltung**, die von den Betriebssystemkernen aller Stellen zur Virtualisierung der physikalischen Prozessoren der Stellen zur Verfügung gestellt wird. Diese verteilte Prozessorverwaltung realisiert den auf Realisierungsebene 1 postulierten globalen Prozessorverwalter. Die im Rahmen der verteilten Prozessorverwaltung eingesetzten Lastverwaltungsstrategien zur Realisierung der virtuellen Prozessoren mit den physikalischen Prozessoren der Stellen

sind hier nicht weiter von Interesse. Zur Allokation und Deallokation eines virtuellen Prozessors werden vom Betriebssystemkern entsprechende Schnittstellenoperationen bereitgestellt.

2. Er unterstützt einen **virtuellen seitenstrukturierten Adreßraum**, der von den virtuellen Prozessoren, die auf der Stelle realisiert sind, gemeinsam genutzt werden kann.

3. Er implementiert ein **zuverlässiges Nachrichtentransportprotokoll**, das sowohl eine asynchrone als auch synchrone Kommunikation mit den anderen Stellen ermöglicht.

4. Er bietet Möglichkeiten zum Erzeugen, Nutzen und Auflösen von sogenannten **Verwalterobjekten**, mit denen Akteursphärenverwalter und Speicherverwalter realisiert werden können. Für ein Verwalterobjekt können Schnittstellenoperationen definiert werden, die global bekannt gemacht werden und somit von allen Stellen aus aufgerufen werden können. Die Ausführung der Schnittstellenoperationen erfolgt gemäß der Semantik von Prozeduraufrufen. Zur Realisierung stellenübergreifender Aufrufe von Schnittstellenoperationen implementiert der Betriebssystemkern auf Basis der von dem zuverlässigen Nachrichtentransportprotokoll bereitgestellten Primitiven zum Senden und Empfangen von Nachrichten das Konzept des entfernten Prozeduraufrufs (*RPC – Remote Procedure Call*, siehe z.B. [BN84]).

Mit den angegebenen Charakteristika der Realisierungsebene 2 sind die wesentlichen Eigenschaften der Hardware–Konfigurationen, auf denen INSEL$^+$–Systeme realisiert werden sollen, erfaßt. Die Realisierungsebene 2 abstrahiert dabei von den spezifischen physikalischen Eigenschaften einer realen Hardware–Konfiguration wie z.B. der Kapazität der lokalen Hauptspeicher und der Übertragungsrate der Nachrichtentransportkanäle. Die angegebenen Eigenschaften des Betriebssystemkerns stellen „minimale“ Anforderungen an den Betriebssystemkern der realen Stellen. Wie die postulierten Dienste von dem Betriebssystemkern realisiert werden, ist hier nicht weiter von Interesse. In [Win96] wird in Zusammenhang mit der Erläuterung der Realisierung von INSEL–Programmen auf vernetzten Mach3.0–Workstations beschrieben, wie ein existierender Mikrokern (in diesem Fall der Mach3.0–Mikrokern [ABG$^+$86]) um eine Basisschicht, die die geforderten Dienste zur Verfügung stellt, erweitert werden kann. In [Rad96] wurde als Basis für die Realisierung von INSEL–Programmen auf einem Cluster von HP–UX Workstations der verteilte Thread–Kern DTK entwickelt. Der DTK erweitert die gegebene UNIX–Variante HP–UX V9.0 [Hew92] um leichtgewichtige Prozesse zur Realisierung von Akteuren und stellt verschiedene Lastverteilungstrategien zur Verteilung der leichtgewichtigen Prozesse auf die jeweils beteiligten Workstations zur Verfügung. Im Zuge der Arbeiten zur Realisierung einer INSEL–angepaßten Betriebssystem–Infrastruktur wurde der Dycos–Kern (*Dynamic context switching kernel*) (vgl. [Cze97a, Cze97b]) entwickelt. Dycos setzt auf der Sun Ultra–SPARC V9 Hardware–Architektur auf und stellt über eine Basisschicht und der darauf definierten Dycos–Bibliothek geeignete Basisdienste zur Realisierung von INSEL–Systemen zur Verfügung.

Die Aufgabe, die nun gestellt ist, besteht darin, gemäß dem Prinzip der schrittweisen Konkretisierung die für die Realisierungsebene 1 erarbeiteten Konzepte und Maßnahmen zur Realisierung von INSEL$^+$–Systemen für die Realisierungsebene 2 so zu konkretisieren, daß die Eigenschaften des Systems unter Berücksichtigung der zusätzlichen Realisierungsrandbedingungen erhalten bleiben. Dazu sind Anpassungen und Erweiterungen der bisher erarbeiteten Realisierungskonzepte und –maßnahmen, zu denen insbesondere die Akteursphärenverwalter gehören, vorzunehmen, die den speziellen Eigenschaften der Realisierungsebene 2 Rechnung tragen. Die notwendigen Anpassungen und Erweiterungen ergeben sich im wesentlichen aus

zwei Eigenschaften, durch die sich die Realisierungsebene 2 von der Realisierungsebene 1 unterscheidet: dem Vorhandensein physikalisch getrennter Speicher und der Unsicherheit der Nachrichtentransportkanäle zwischen den Stellen.[19]

Physikalisch getrennte Speicher
Nach dem oben Gesagten verfügt jede Stelle über lokalen Speicher, der über den lokalen virtuellen Adreßraum von den virtuellen Prozessoren, die auf der Stelle realisiert sind, nutzbar ist. Damit ist auf der Realisierungsebene 2 kein gemeinsamer Speicher und somit auch kein globaler Adreßraum mehr vorhanden.
Die Komponenten eines INSEL$^+$–Systems sowie die für ihre Realisierung benötigten Verwalter sind also auf den unterschiedlichen Stellen zu realisieren. Für das weitere wird postuliert, daß eine DA–Inkarnation zusammen mit ihren lokalen DE–Komponenten immer vollständig auf einer Stelle repräsentiert wird und ein Verwalter immer auf der Stelle realisiert wird, auf der der Akteur, dem er zugeordnet ist, ausgeführt wird. Aufgrund der Repräsentation der Komponenten auf unterschiedlichen Stellen ist bei Zugriffen auf Komponenten zwischen stellenlokalen und stellenübergreifenden Zugriffen zu unterscheiden. Ein Zugriff einer Komponente K_1 auf eine Komponente K_2 ist **stellenlokal**, wenn die beiden Komponenten K_1 und K_2 auf der gleichen Stelle realisiert sind; anderenfalls ist der Zugriff **stellenübergreifend**.
Aufgrund des Fehlens eines globalen Adreßraums bzw. eines gemeinsamen Speichers sind die Akteursphärenverwalter insbesondere um Fähigkeiten zur Realisierung stellenübergreifender Zugriffe auf passive INSEL$^+$–Inkarnationen zu erweitern.

Unsicherheit der Transportkanäle
Die Stellen sind untereinander über Nachrichtentransportkanäle vernetzt. Diese Nachrichtentransportkanäle sind in dem Sinne unsichere Kanäle, daß sie nicht gegen aktive und passive Angriffe geschützt sind. Dies bedeutet, daß die über diese Kanäle übertragenen Nachrichten abgehört bzw. modifiziert werden können sowie Nachrichten eingeschleust und abgefangen werden können. Das von dem Betriebssystemkern bereitgestellte zuverlässige Nachrichtentransportprotokoll garantiert zwar, daß technisch bedingte Übertragungsfehler erkannt werden und keine Nachrichten verloren gehen. Die Sicherheit der Nachrichtenübertragung im Sinne der vertrauenlichen und authentischen Übertragung von Nachrichten wird durch das Protokoll jedoch nicht gewährleistet. Die Übertragung einer Nachricht ist vertraulich, wenn außer Sender und Empfänger der Nachricht kein unautorisierter Dritter Kenntnis von den in der Nachricht enthaltenen Informationen enthalten kann. Sie ist authentisch, wenn der Sender und der Empfänger einer Nachricht wechselseitig authentifiziert sind und die Nachricht nicht durch Dritte modifiziert werden kann. Die authentische Übertragung von Nachrichten über die Transportkanäle ist wesentliche Voraussetzung zur Realisierung der mit den Sprachkonzepten festgelegten Eigenschaften eines INSEL$^+$–Systems. Dazu gehören zum einen die funktionalen Eigenschaften des Systems und zum anderen die für das System getroffenen Rechtsfestlegungen. Ist die Authentizität der Nachrichtenübertragung nicht gewährleistet, so können zum Beispiel die Parameter, die bei stellenübergreifenden Aufrufen von Schnittstellenoperationen der Akteursphärenverwalter über die Transportkanäle zu übertragen sind, modifiziert werden. Ein Parameter, der für die Durchführung der Zugriffskontrollen wesentlich

[19] Die sich aus dem Vorhandensein unterschiedlicher Speicherressourcen (Haupt– und Hintergrundspeicher) ergebenden Bedrohungen und Sicherheitsprobleme (z.B. Austausch von Festplatten und externes Lesen der darauf gespeicherten Daten oder Aufspielen manipulierter Daten auf die Platte vor Wiedereinbau) werden in dieser Arbeit nicht weiter betrachtet. Sie können im allgemeinen durch eine verschlüsselte Speicherung der Daten auf den Hintergrundspeichermedien inkl. Erstellung von Prüfsummen und digitalen Signaturen gelöst werden.

ist, ist zum Beispiel der aktuelle Wert der Kontext–Datenstruktur eines Akteurs. Wird dieser Parameter während seiner Übertragung modifiziert, so kann dies Verletzungen des Rechts des INSEL$^+$–Systems zur Folge haben mit der Konsequenz, daß das Recht des Systems nicht korrekt durchgesetzt wird.
Für die Realisierungsebene 2 sind also Maßnahmen anzugeben, durch deren Einsatz sowohl die Authentizität als auch die Vertraulichkeit der über die unsicheren Transportkanäle übertragenen Nachrichten gewährleistet werden kann. Wie im weiteren noch deutlich werden wird, werden alle stellenübergreifenden Zugriffe über die Akteursphärenverwalter realisiert. Die zur Gewährleistung der Vertraulichkeit und der Authentizität von Nachrichten eingesetzten Maßnahmen bestehen somit in der wechselseitigen Authentifizierung stellenübergreifend kooperierender Akteursphärenverwalter sowie der Verschlüsselung der zwischen diesen auszutauschenden Nachrichten.

Nach dem bisher Gesagten sind also beim Übergang von der Realisierungsebene 1 zur Realisierungsebene 2 Anpassungen und Erweiterungen der Realisierungskonzepte und –maßnahmen vorzunehmen, die zum einen dem Vorhandensein physikalisch getrennter Speicher und zum anderen der Unsicherheit der Nachrichtentransportkanäle Rechnung tragen. Diese Anpassungen und Erweiterungen sowie die sich jeweils bietenden Alternativen werden im folgenden in mehreren Schritten beschrieben. Zunächst werden die grundlegenden Aufgaben und Eigenschaften der Verwalter, die nahezu unverändert von der Realisierungsebene 1 übernommen werden, konkretisiert. Anschließend werden die Erweiterungen der Akteursphärenverwalter zur Realisierung stellenübergreifender Zugriffe auf passive Inkarnationen erklärt. Darauf folgend werden die Maßnahmen, die zur wechselseitigen Authentifizierung von Akteursphärenverwaltern sowie zur Verschlüsselung von Nachrichten eingesetzt werden, beschrieben. Schließlich werden Optimierungen vorgeschlagen, die zum Ziel haben, die Anzahl stellenübergreifender Aufrufe von Akteursphärenverwalteroperationen zu reduzieren. Diese Optimierungen bestehen zum einen darin, sogenannte Stellvertreter–Verwalter einzuführen, die es ermöglichen, eine Akteursphäre verteilt auf mehreren Stellen zu realisieren. Zum anderen werden Maßnahmen angegeben, um stellenübergreifende Aufrufe zur Auswertung von Zugriffsrestriktionsausdrücken nach Möglichkeit zu vermeiden.

6.3.2 Konkretisierung der Verwalter

Die in Abschnitt 6.2 für die Realisierungsebene 1 angegebenen grundlegenden Konzepte zur Realisierung von INSEL$^+$–Systemen werden für die Realisierungsebene 2 zunächst unverändert übernommen. Zu diesen Konzepten gehören insbesondere die Speicherverwalter, deren Aufgabe in der Verwaltung des insgesamt zur Verfügung stehenden Speichers besteht, sowie die Akteursphärenverwalter, die für die Erzeugung und Auflösung von Inkarnationen sowie für die Durchführung der Zugriffskontrollen und der Verwaltung der für diese Kontrollen benötigten Informationen zuständig sind. Auf die Konkretisierung und Anpassung der Speicherverwalter an die Eigenschaften der Realisierungsebene 2 wird hier nicht weiter eingegangen (siehe dazu [Win96]). Die Akteursphärenverwalter werden auf der Realisierungsebene 2 mit den von dem Betriebssystemkern der Stellen zur Verfügung gestellten Verwalterobjekten realisiert. Ihre in den Abschnitten 6.2.2 und 6.2.3 angegebenen Schnittstellenoperationen werden als Schnittstellenoperationen der entsprechenden Verwalterobjekte definiert. Wie oben bereits erläutert, wird ein Akteursphärenverwalter immer zusammen mit dem Akteur, dessen Sphäre er zugeordnet ist, auf einer Stelle realisiert. Die Akteure werden durch die von den Betriebssystemkernen der Stellen implementierte verteilte Prozessorverwaltung auf die Stellen verteilt, woraus sich eine analoge Verteilung der Akteursphärenverwalter auf die

Stellen ergibt. Daraus folgt, daß die Kooperation zwischen die Akteursphärenverwaltern, die zur Erfüllung der von ihnen wahrzunehmenden Aufgaben notwendig ist, im allgemeinen stellenübergreifend erfolgt. Der stellenübergreifende Aufruf einer Schnittstellenoperation eines Akteursphärenverwalters wird dabei transparent durch das von den Betriebssystemkernen der Stellen implementierte Konzept des entfernten Prozeduraufrufs realisiert.

Aus der unveränderten Übernahme der grundlegenden Realisierungskonzepte der Realisierungsebene 1 folgt insbesondere, daß die Speicherrepräsentation einer Inkarnation von dem Akteursphärenverwalter erzeugt wird, dessen Sphäre die Inkarnation zugeordnet ist. Eine passive Inkarnation wird also im Speicher der Stelle repräsentiert, auf der der Akteur, von dem die Inkarnation lebenszeitmäßig abhängig ist, realisiert ist. Damit ergibt sich zum einen, daß eine Akteursphäre immer komplett auf einer Stelle realisiert wird. Zum anderen hat diese Vorgehensweise zur Folge, daß anonyme Depots und anonyme DE–Inkarnationen im allgemeinen entfernt erzeugt werden und Zugriffe auf diese in der Regel stellenübergreifend sind.

Die Funktionalität der Akteursphärenverwalter, die sie zur Erzeugung von Inkarnationen zur Verfügung stellen, bleibt im wesentlichen erhalten. Anpassungen sind lediglich für die Erzeugung anonymer Depots vorzunehmen. Diese Anpassungen sind notwendig, da ein anonymes Depot im allgemeinen nicht auf der Stelle repräsentiert wird, auf der der erzeugende Akteur realisiert ist. Zur Erzeugung eines anonymen Depots ruft ein Akteur A, wie in Abschnitt 6.2.2.1 erklärt, die Operation `CreateDepot` auf seinem Akteursphärenverwalter $ASV(A)$ auf. Ist das zu erzeugende Depot der Akteursphäre von A zuzuordnen, so wird das Depot lokal von dem Akteursphärenverwalter $ASV(A)$ erzeugt und anschließend wird die kanonische Operation des Depots durch den virtuellen Prozessor, mit dem der Akteur A realisiert ist, ausgeführt. Anderenfalls wird der Auftrag zur Erzeugung des Depots durch Aufruf der Operation `CreateDepotDesc` an den Akteursphärenverwalter $ASV(D)$ weitergeleitet, dessen Sphäre das Depot zuzuordnen ist. $ASV(D)$ kann auf der gleichen Stelle wie $ASV(A)$ realisiert sein. Ist dies der Fall, kann das Depot ebenfalls lokal durch den Akteursphärenverwalter $ASV(D)$ erzeugt werden und die kanonische Operation des Depots durch den virtuellen Prozessor des Akteurs A ausgeführt werden. Ist der Akteursphärenverwalter $ASV(D)$ jedoch auf einer anderen Stelle als $ASV(A)$ realisiert, erfolgt der Aufruf von `CreateDepotDesc` stellenübergreifend. Bei einer stellenübergreifenden Erzeugung des Depots besteht das Problem, daß die kanonische Operation des Depots nicht von dem virtuellen Prozessor, mit dem der Akteur A realisiert ist, ausgeführt werden kann. Dementsprechend ist von dem Akteursphärenverwalter $ASV(D)$ nach Allokation von Speicher zur Repräsentation des Depots sowie der Erzeugung der Komponentenbeschreibung des Depots ein lokaler virtueller Prozessor zu allokieren, der stellvertretend für den virtuellen Prozessor, mit dem der Akteur A realisiert ist, die kanonische Operation des Depots ausführt. Der virtuelle Prozessor des Akteurs A wird solange blockiert, bis die Ausführung der kanonischen Operation des Depots abgeschlossen ist. Für die Ausführung der kanonischen Operation des Depots wird von dem Akteursphärenverwalter $ASV(D)$ ein neuer Lebenszeitkeller angelegt, in den als erstes Element die Komponentenbeschreibung des Depots eingetragen wird. Dieser zusätzliche Lebenszeitkeller ist notwendig, da bei Ausführung der kanonischen Operation des Depots alle Arten von Inkarnationen erzeugt werden können, deren Terminierung und Auflösung nach den für INSEL$^+$–Systeme festgelegten Regeln über diesen Lebenszeitkeller ermöglicht wird. Mit der Terminierung der kanonischen Operation des Depots wird die Komponentenbeschreibung des Depots aus diesem Lebenszeitkeller in den Lebenszeitkeller umgeordnet, den der Akteursphärenverwalter $ASV(D)$ für den Akteur verwaltet, dessen Sphäre er zugeordnet ist. Die Komponentenbeschreibung des Depots wird dabei in horizontaler Richtung an die Komponentenbeschreibung der Inkarnation angebunden, von der das Depot lebenszeitmäßig abhängig ist. Anschließend

wird der für die Ausführung der kanonischen Operation des Depots allokierte virtuelle Prozessor freigegeben sowie der für das Depot angelegte Lebenszeitkeller aufgelöst. Damit terminiert die Ausführung der Verwalteroperationen `CreateDepotDesc` und `CreateDepot`, und die Berechnungen des Akteurs A werden fortgesetzt.

Neben der Erzeugung von Inkarnationen gehört zu den Aufgaben der Akteursphärenverwalter die Umsetzung der für INSEL$^+$–Systeme festgelegten Terminierungs– und Auflösungsregeln, die Realisierung des Operationen–orientierten Rendezvous–Konzepts sowie die Durchführung von Zugriffskontrollen und die Verwaltung der dafür benötigten Informationen. Die in Abschnitt 6.2.2 beschriebene Funktionalität, die die Akteursphärenverwalter zur Erledigung der genannten Aufgaben bereitstellen, kann – ohne Anpassungen vornehmen zu müssen – auf die Realisierungsebene 2 übertragen werden. Der einzige Unterschied besteht darin, daß Aufrufe der Schnittstellenoperationen der Akteursphärenverwalter nun im allgemeinen stellenübergreifend erfolgen.

Aufgrund des auf der Realisierungsebene 1 vorhandenen gemeinsamen Speichers müssen die Akteursphärenverwalter der Realisierungsebene 1 keine Funktionalität zur Realisierung von Zugriffen auf passive Inkarnationen zur Verfügung stellen. Die Akteure, die mit den Prozessoren der Realisierungsebene 1 realisiert sind, können immer direkt auf die Speicherrepräsentationen der Inkarnationen zugreifen. Dies ist auf der Realisierungsebene 2 aufgrund des Fehlens eines gemeinsamen Speichers bei Zugriffen auf entfernt realisierte passive Inkarnationen nicht mehr möglich. Dementsprechend werden die Akteursphärenverwalter um Fähigkeiten zur Realisierung von stellenübergreifenden Zugriffen auf passive Inkarnationen erweitert.

6.3.3 Realisierung stellenübergreifender Zugriffe

In diesem Abschnitt wird erklärt, wie mit Unterstützung der Akteursphärenverwalter stellenübergreifende Zugriffe auf passive Inkarnationen realisiert werden. Zuvor sei noch angemerkt, daß zur Realisierung von Zugriffen auf entfernt realisierte K-Akteure keine Erweiterungen der Akteursphärenverwalter notwendig sind. Ein Zugriff auf einen K-Akteur entspricht der Ausführung einer seiner Kommunikationsoperationen (K-Order). Wie bereits erwähnt, können die Maßnahmen zur Realisierung von K-Order von der Realisierungsebene 1 übernommen werden. Erfolgt der Aufruf einer Kommunikationsoperation stellenübergreifend, so bedeutet dies lediglich, daß der von dem Akteursphärenverwalter des Auftraggebers auf dem Akteursphärenverwalter des jeweiligen K-Akteurs vorgenommene Aufruf der Operation `InsertKOrder` stellenübergreifend ist.

Bei Zugriffen auf passive Inkarnationen ist zwischen Zugriffen auf DE–Inkarnationen und Zugriffen auf Depots zu unterscheiden. Ein Zugriff auf eine DE–Inkarnation entspricht der Ausführung einer Lese– oder Schreiboperation auf der DE-Inkarnation, während ein Zugriff auf ein Depot der Ausführung einer der für das Depot definierten Zugriffsoperationen entspricht. Im folgenden wird zunächst erklärt, wie Zugriffe auf entfernt realisierte DE–Inkarnationen realisiert werden. Anschließend wird dann auf die Möglichkeiten zur Ausführung von Zugriffsoperationen entfernt realisierter Depots eingegangen.

Zugriffe auf DE–Inkarnationen

Eine Lese– bzw. Schreiboperation auf einer DE–Inkarnation K kann nur auf der Stelle ausgeführt werden, in deren Speicher K repräsentiert ist. Ein Akteur A, der auf die DE–Inkarnation K zugreifen will und der auf einer anderen Stelle als K realisiert ist, kann also nicht direkt auf K zugreifen. Um solche entfernten Zugriffe auf DE–Inkarnationen zu realisieren, wird die Schnittstelle der Akteursphärenverwalter erweitert (vgl. Tabelle 6.11). Die

Operationen `GetRemValue` und `GetValue` dienen zur Realisierung der Ausführung einer Lese–Operation und die Operationen `SetRemValue` und `SetValue` zur Realisierung der Ausführung einer Schreiboperation auf einer entfernt repräsentierten DE–Inkarnation.

Ein Akteur A ruft zur Realisierung einer Lese– bzw. Schreiboperation auf einer entfernten DE–Inkarnation K die Operation `GetRemValue` bzw. `SetRemValue` auf seinem Akteursphärenverwalter $ASV(A)$ auf. Dieser ruft dann stellenübergreifend die Operation `GetValue` bzw. `SetValue` auf dem Akteursphärenverwalter auf, der die Komponentenbeschreibung der DE–Inkarnation K verwaltet. Die Ausführung von `GetValue` durch diesen Akteursphärenverwalter bewirkt, daß der aktuelle Wert von K gelesen wird und als Ergebnis von `GetValue` an den aufrufenden Akteursphärenverwalter $ASV(A)$ und damit an den Akteur A übergeben wird. Die Ausführung von `SetValue` bewirkt, daß der DE–Inkarnation K der Wert zugewiesen wird, der als aktueller Parameter von `SetValue` übergeben wird.

Operation	**Funktionalität**	**Aufruf durch**
`GetRemValue`	Lesen des Wertes einer entfernten DE–Inkarnation	Akteur
`GetValue`	Lesen des Wertes einer lokalen DE–Inkarnation	ASV
`SetRemValue`	Schreiben des Wertes einer entfernten DE–Inkarnation	Akteur
`SetValue`	Schreiben des Wertes einer lokalen DE–Inkarnation	ASV
`CreateDepotOp`	Erzeugung einer ZOp eines Depots	Akteur
`ExecDepotOp`	Ausführen einer ZOp, die einer S–Order entspricht, auf einem lokalen Depot	ASV

Tabelle 6.11: Schnittstellenoperationen der Akteursphärenverwalter zur Realisierung stellenübergreifender Zugriffe auf passive Inkarnationen

Zur Ausführung einer Lese– bzw. Schreiboperation auf einer entfernt realisierten DE–Inkarnation sind also nach dem erklärten Verfahren zwei Verwalteraufrufe durchzuführen, wobei der erste Aufruf stellenlokal ist. Alternativ zu diesem Verfahren besteht die Möglichkeit, daß sich ein Akteur zur Ausführung einer Lese– bzw. Schreiboperation auf einer entfernt realisierten DE–Inkarnation unmittelbar an den Akteursphärenverwalter wendet, der die Komponentenbeschreibung der DE–Inkarnation verwaltet, und somit den lokalen Verwalteraufruf einzusparen. Dies würde bedeuten, daß der Akteur direkt mit diesem Akteursphärenverwalter stellenübergreifend kommuniziert. Wie später noch motiviert wird, ist es zur Gewährleistung der Vertraulichkeit und Authentizität von stellenübergreifend auszutauschenden Nachrichten sinnvoll, alle stellenübergreifende Kommunikation lediglich zwischen Akteursphärenverwaltern durchzuführen. Dies ermöglicht es insbesondere, die Objekte, die wechselseitig zu authentifizieren sind, auf die Menge der stellenübergreifend kooperierenden Akteursphärenverwalter zu beschränken. Würde sich ein Akteur zur Realisierung eines Zugriffs auf eine entfernte DE–Inkarnation direkt an den Akteursphärenverwalter dieser DE–Inkarnation wenden, müßten der Akteur und dieser Akteursphärenverwalter wechselseitig authentifiziert werden, was zusätzlichen Aufwand verursachen würde. Dieser Aufwand wird in dem vorgestellten Verfahren durch den lokalen Operationsaufruf, mit dem der Akteur sich zunächst an seinen Akteursphärenverwalter wendet, vermieden.

Nachdem geklärt ist, wie Zugriffe auf entfernt repräsentierte DE–Inkarnationen realisiert werden, werden im folgenden Möglichkeiten zur Ausführung von Zugriffsoperationen entfernt realisierter Depots vorgestellt.

Zugriffe auf Depots

Wie in Kapitel 3 angegeben, entsprechen die Zugriffsoperationen eines Depots den äußeren Operationen der lokalen N–Komponenten des Depots, die mit dem Attribut E definiert sind. Da alle Arten von lokalen N–Komponenten eines Depots mit dem Attribut E definiert werden können, sind unterschiedliche Arten von Zugriffsoperationen eines Depots zu unterscheiden. Zum einen gibt es Zugriffsoperationen, die Lese– bzw. Schreiboperationen auf lokalen DE–Inkarnationen des Depots sind. Zum anderen können Zugriffsoperationen die *erzeuge*–Operationen lokaler DE– und DA–Generatoren des Depots sein. Eine Zugriffsoperation eines entfernt realisierten Depots, die Lese– bzw. Schreiboperation einer lokalen DE–Inkarnation des Depots ist, wird analog wie ein Zugriff auf eine entfernt repräsentierte DE–Inkarnation behandelt und dementsprechend realisiert. Ist die Zugriffsoperation die *erzeuge*–Operation eines lokalen Generators des Depots, hängt die Realisierung eines entfernten Aufrufs dieser Zugriffsoperation von der Art des Generators ab.

Wenn es sich bei dem Generator um einen DE–Generator, einen Akteur–Generator oder um einen Depot–Generator handelt, werden keine besonderen Maßnahmen zur Realisierung eines entfernten Aufrufs der *erzeuge*–Operation dieses Generators ergriffen. Ruft ein Akteur A eine Zugriffsoperation eines Depots auf, die der *erzeuge*–Operation eines solchen Generators entspricht, wird die damit zu erzeugende Inkarnation „wie üblich“ erzeugt, d.h. so, als ob dies die *erzeuge*–Operation auf einem entsprechenden Generator ist, der nicht mit dem Attribut E definiert ist. Im einzelnen bedeutet dies folgendes: Im Fall eines DE–Generators ruft der Akteur A zur Realisierung des Aufrufs die Operation `CreateDEInc` auf seinem Akteursphärenverwalter $ASV(A)$ auf. Entspricht die Zugriffsoperation der *erzeuge*–Operation eines Akteur–Generators, wird die Operation `Create[Prot]MActor` bzw. `Create[Prot]KActor` auf $ASV(A)$ aufgerufen. Ist der Generator ein Depot–Generator, erfolgt die Realisierung des Aufrufs durch Aufruf der Operation `Create[Prot]Depot` auf dem Akteursphärenverwalter $ASV(A)$.

Handelt es sich bei der Zugriffsoperation jedoch um die *erzeuge*–Operation eines lokalen S-Order–Generators des Depots, werden besondere Maßnahmen zur Realisierung eines entfernten Aufrufs und der entfernten Ausführung dieser Operation ergriffen. Dies ist dadurch motiviert, daß die Zugriffsoperationen, die durch S-Order–Generatoren definiert sind, im allgemeinen die Zugriffsoperationen sind, bei deren Ausführung auf lokale DE–Inkarnationen des Depots zugegriffen wird. Würden entfernte Aufrufe dieser Zugriffsoperationen wie übliche S-Order–Aufrufe realisiert, würden die entsprechenden S-Order immer lokal auf der Stelle ausgeführt, auf der der aufrufende Akteur realisiert ist. Dies hätte zur Konsequenz, daß die bei Ausführung der S-Order auf den lokalen DE–Inkarnationen des Depots durchzuführenden Zugriffe stets stellenübergreifend sind. Aus Effizienzgründen ist es daher sinnvoll, die S-Order, die Zugriffsoperationen eines Depots entsprechen, auf der Stelle auszuführen, auf der das Depot realisiert ist, und somit die Zugriffe auf die lokalen DE–Inkarnationen des Depots lokal ausführen zu können. Zur Realisierung stellenübergreifender Aufrufe von Zugriffsoperationen eines Depots, die S-Order–Aufrufen entsprechen, wird die Schnittstelle der Akteursphärenverwalter um die Operation `ExecDepotOp` erweitert (siehe Tabelle 6.11; die ebenfalls in der Tabelle aufgeführte Operation `CreateDepotOp` wurde bereits in Abschnitt 6.2.2.1 für die Akteursphärenverwalter der Realisierungsebene 1 definiert). Ein Akteur A ruft zur Realisierung eines S-Order–Aufrufs, der Aufruf einer Zugriffsoperation eines entfernt realisierten Depots ist, die Operation `CreateDepotOp` auf seinem Akteursphärenverwalter $ASV(A)$ auf. Dieser ruft dann im Rahmen der Ausführung von `CreateDepotOp` stellenübergreifend die Operation `ExecDepotOp` auf dem Akteursphärenverwalter auf, dessen Sphäre das Depot zugeordnet ist und der somit die Depot–Repräsentation erzeugt hat. Analog zu dem bereits erklärten Fall

der entfernten Erzeugung eines anonymen Depots besteht das Problem, daß die Zugriffsoperation, d.h. die kanonische Operation der zu erzeugenden S-Order nicht von dem virtuellen Prozessor, mit dem der Akteur A realisiert ist, ausgeführt werden kann. Dementsprechend wird bei Ausführung der Operation `ExecDepotOp` zunächst ein neuer Lebenszeitkeller angelegt, in den als erstes Element die Komponentenbeschreibung der zu erzeugenden S-Order eingetragen wird. Anschließend wird auf der Stelle, auf der das Depot repräsentiert ist, ein lokaler virtueller Prozessor allokiert, der stellvertretend für den virtuellen Prozessor, mit dem der Akteur A realisiert ist, die kanonische Operation der S-Order ausführt. Die für die korrekte Auflösung der Inkarnationen, die bei Ausführung der S-Order erzeugt werden, benötigten Informationen werden in dem für die S-Order erzeugten Lebenszeitkeller verwaltet. Der virtuelle Prozessor des aufrufenden Akteurs A bleibt solange blockiert, bis die Ausführung der kanonischen Operation der S-Order abgeschlossen ist und damit der erzeugte Lebenszeitkeller aufgelöst sowie der allokierte virtuelle Prozessor freigegeben wird.

Alternativen zur Realisierung von Zugriffen

Mit den erklärten Erweiterungen der Akteursphärenverwalter ist es möglich, Zugriffe auf entfernt repräsentierte passive Inkarnationen zu realisieren. Die Erweiterungen sind dadurch charakterisiert, daß ein Zugriff auf eine passive Inkarnation immer auf der Stelle ausgeführt wird, in deren Speicher die Inkarnation repräsentiert ist. Als Alternative zu diesem Vorgehen besteht die Möglichkeit, die Repräsentation einer passiven Inkarnation auf die Stelle zu migrieren, auf der der aufrufende Akteur realisiert ist, um so den Zugriff lokal ausführen zu können.
Zur **Migration** einer passiven Inkarnation von einer Stelle S_1 auf eine Stelle S_2 ist zum einen die Speicherrepräsentation und zum anderen die Komponentenbeschreibung der Inkarnation von der Stelle S_1 auf die Stelle S_2 zu verschieben. Mit der Komponentenbeschreibung sind insbesondere auch alle Datenstrukturen, die als Bestandteil der Komponentenbeschreibung verwaltet werden, wie zum Beispiel eine Zugriffskontrolliste oder eine Zugriffshistorienliste, auf die Zielstelle zu verschieben. Die Komponentenbeschreibung der zu migrierenden Inkarnation ist auf der Zielstelle geeignet zu verankern, d.h. es muß auf der Zielstelle ein Akteursphärenverwalter vorhanden sein, in dessen Lebenszeitkeller die Komponentenbeschreibung eingeordnet werden kann. Mit den in Abschnitt 6.3.5 eingeführten Stellvertreter–Verwaltern wird hierfür eine Basis geschaffen.

Als weitere Alternative zur Realisierung eines Zugriffs auf eine entfernt repräsentierte passive Inkarnation bietet sich die Möglichkeit, auf der Stelle, auf der der aufrufende Akteur realisiert ist, ein Replikat der Inkarnation anzulegen und somit den Zugriff lokal auszuführen. Die **Replikation** hat gegenüber der Migration generell den Vorteil, daß potentiell auf jeder Stelle ein Replikat einer Inkarnation existieren kann und somit Zugriffe auf die Inkarnation immer lokal und – zumindest Lesezugriffe – parallel ausgeführt werden können. Diesem Vorteil steht der Nachteil gegenüber, daß Maßnahmen zur Konsistenzhaltung der Replikate durchgeführt werden müssen. Der Aufwand, der für die Konsistenzhaltung der Replikate zu leisten ist, hängt dabei von der zugrundeliegenden Kohärenzform ab. Die Kohärenzform legt – grob gesprochen – fest, wann Änderungen der Werte eines Replikats einer Inkarnation in den anderen Replikaten der Inkarnation sichtbar, d.h. lesbar werden. Einen guten Überblick über verschiedene Kohärenzformen liefert zum Beispiel [Mos93].
Zur Replikation von passiven Inkarnationen sind auf den Stellen, auf denen Replikate angelegt werden, Informationen über diese Replikate, wie zum Beispiel Informationen, die für die Konsistenzhaltung der Replikate benötigt werden, zu verwalten. Als Anker für die Verwaltung dieser Informationen können ebenfalls die in Abschnitt 6.3.5 eingeführten Stellvertreter–Verwalter dienen.

Mit den angegebenen Alternativen der Migration und der Replikation sowie der entfernten Ausführung eines Zugriffs stehen insgesamt drei Möglichkeiten zur Realisierung eines Zugriffs auf eine entfernt repräsentierte passive Inkarnation zur Verfügung. Um das sich damit ergebende Spektrum von Realisierungsalternativen für Zugriffe auf passive Inkarnationen unter dem Gesichtspunkt der effizienten Realisierung von Zugriffen ausschöpfen zu können, werden Kriterien und Strategien benötigt, anhand derer entschieden werden kann, auf welche Art ein Zugriff realisiert wird. Kriterien, die diese Entscheidung beeinflussen, sind zum Beispiel die Anzahl konkurrenter Zugreifer auf die jeweilige Inkarnation sowie deren Zugriffsverhalten. Auf diese Kriterien und Strategien wird im weiteren nicht näher eingegangen, da die Möglichkeiten der Migration und Replikation hier lediglich genannt wurden, um das Spektrum von Möglichkeiten, das generell zur Realisierung von Zugriffen auf passive Inkarnationen zur Verfügung steht, aufzuzeigen. Entsprechende Kriterien und Strategien sind in [Win96] zu finden. Ebenso werden in dieser Arbeit keine Maßnahmen zur konkreten Realisierung der Migration bzw. Replikation von passiven Inkarnationen vorgestellt. Es sei lediglich darauf hingewiesen, daß die entsprechenden Realisierungsmaßnahmen stark von den Basismechanismen, die der Betriebssystemkern der Stellen zur Unterstützung verteilt gemeinsamen Speichers (*Distributed Shared Memory*, siehe z.B. [LH89], [NL91]) zur Verfügung stellt, abhängig ist.

Bezüglich der Möglichkeit der Migration bzw. Replikation eines zugriffskontrollierten Depots bzw. eines Depots, für das eine Zugriffshistorienliste verwaltet wird, werden im folgenden noch einige Anmerkungen gemacht. Diese Depots sind besonders hervorzuheben, da für sie sicherheitsrelevante Informationen verwaltet werden, die für die Durchführung von Zugriffskontrollen benötigt werden. Die Migration eines solchen Depots ist unproblematisch, sofern gewährleistet ist, daß die zur Migration des Depots über die Nachrichtentransportkanäle zu sendenden Nachrichten ausreichend geschützt sind. Wie bereits erklärt, wird bei der Migration eines Depots die komplette Komponentenbeschreibung des Depots einschließlich der ggf. für das Depot verwalteten Zugriffskontrolliste bzw. Zugriffshistorienliste auf die Zielstelle verschoben. Diese Listen werden dann von dem Stellvertreter–Verwalter (siehe Abschnitt 6.3.5), in dessen Lebenszeitkeller die Komponentenbeschreibung des Depots auf der Zielstelle eingetragen wird, verwaltet. Der Fall der Replikation eines Depots, für das eine Zugriffskontrolliste und/oder eine Zugriffshistorienliste verwaltet wird, ist differenzierter zu betrachten. Hier besteht die Möglichkeit, diese Listen entweder mit dem Depot zu replizieren, um so bei der Durchführung von Zugriffskontrollen ggf. lokal auf diese Listen zugreifen zu können, oder diese Listen nicht zu replizieren, sondern sie stets von dem Akteursphärenverwalter verwalten zu lassen, der die Originalrepräsentation des Depots erzeugt hat. Im Fall der Replikation der Listen mit dem Depot besteht das Problem, die Inhalte der Listen konsistent zu halten. Änderungen der Einträge dieser Listen können möglicherweise Rechteänderungen zur Folge haben. Um die Forderung nach möglichst sofortiger Wirkung von Rechteänderungen durchzusetzen, sind die Replikate dieser Listen auf jeden Fall streng kohärent zu verwalten.[20] Werden Zugriffskontrollisten und Zugriffshistorienlisten nicht mit den jeweiligen Depots repliziert, sondern stets von dem Akteursphärenverwalter der Originalrepräsentation des Depots verwaltet, besteht das Konsistenzproblem nicht. Dafür muß jedoch bei der Durchführung von Zugriffskontrollen in der Regel entfernt auf diese Listen zugegriffen werden. Dies wirkt sich insbesondere dann nachteilig aus, wenn die Zugriffsoperation eines zugriffskontrollierten Depots aufgrund eines vorhandenen Replikats lokal ausgeführt werden kann, vorher jedoch zur Auswertung des Zugriffsrestriktionsausdrucks der Zugriffsoperation stellenübergreifende Zugriffe zur Überprüfung der Zugriffskontrolliste bzw. der Zugriffshistorienliste des Depots

[20] Strenge Kohärenz, die häufig auch als sequentielle Konsistenz bezeichnet wird, besagt informell, daß ein Lesezugriff jeweils den Wert des zuletzt geschriebenen Schreibzugriffs liefert ([Mos93]).

durchzuführen sind. Um in derartigen Fällen die Anzahl der erforderlichen stellenübergreifenden Auswertungen von Zugriffsrestriktionsausdrücken zu reduzieren, bietet sich in Zusammenhang mit der Replikation zugriffskontrollierter Depots die Nutzung des in Abschnitt 6.2.3 vorgestellten Ticket–Konzepts an. Auf der Realisierungsebene 2 ist somit die Repräsentationsart eines zugriffskontrollierten Depots als weiteres Kriterium in der Ticketvergabestrategie zu berücksichtigen.

Es wird deutlich, daß im Zusammenhang mit der Replikation von Depots, für die eine Zugriffskontrolliste bzw. eine Zugriffshistorienliste verwaltet wird, noch einige Fragen zu klären sind. Insbesondere werden Kriterien und Strategien benötigt, die es ermöglichen, im Fall der Entscheidung für die Replikation eines solchen Depots zu entscheiden, ob die Zugriffskontrolliste bzw. die Zugriffshistorienliste mit dem Depot repliziert werden soll oder nicht und ob ggf. Tickets für die Nutzung des Depots vergeben werden sollen. Die Erarbeitung entsprechender Kriterien und Strategien geht jedoch über den Rahmen dieser Arbeit hinaus. Dementsprechend wird im weiteren davon ausgegangen, daß Depots, für die eine Zugriffskontrolliste bzw. eine Zugriffshistorienliste verwaltet wird, generell nicht repliziert werden.

6.3.4 Wechselseitige Authentifizierung von Akteursphärenverwaltern

In Abschnitt 6.3.1 wurde erläutert, daß die Nachrichtentransportkanäle zwischen den Stellen unsichere Kanäle sind und dementsprechend Maßnahmen einzusetzen sind, die die vertrauliche und authentische Übertragung von Nachrichten über diese Kanäle gewährleisten. Die generellen Maßnahmen, um dieses Ziel zu erreichen, wurden bereits in Kapitel 2 vorgestellt. Sie bestehen in der wechselseitigen Authentifizierung stellenübergreifend kommunizierender Komponenten sowie der Verschlüsselung der zwischen diesen auszutauschenden Nachrichten. Die wechselseitige Authentifizierung sowie der Austausch der zur Verschlüsselung der Nachrichten benötigten Schlüssel erfolgt durch den Einsatz sogenannter Authentifizierungsprotokolle. In diesem Abschnitt wird erklärt, wie die zur Authentifizierung einzusetzenden Verfahren in die bisher erarbeitete Verwalterarchitektur eingeordnet werden können. Die in Kapitel 2 vermittelten Grundlagen zu Authentifizierungs– und Schlüsselaustauschverfahren werden dabei vorausgesetzt.

6.3.4.1 Anforderungen und Schlüsselverwalter

Die Komponenten, zwischen denen Nachrichten stellenübergreifend ausgetauscht werden können, sind die Akteursphärenverwalter. Dies sind die einzigen Komponenten, die unmittelbar stellenübergreifend kooperieren können. Die Kooperation mittels des Operationen–orientierten Rendezvous–Konzepts zwischen Akteuren, die auf unterschiedlichen Stellen realisiert sind, bzw. der Zugriff eines Akteurs auf eine entfernt realisierte Inkarnation erfolgt stets mittelbar unter Zuhilfenahme der Akteursphärenverwalter. Ein Akteur wendet sich zur Ausführung einer Verwalteroperation immer lokal an seinen Akteursphärenverwalter, der dann bei Ausführung der Operation ggf. mit anderen Akteursphärenverwalter stellenübergreifend kooperiert. Im weiteren wird davon ausgegangen, daß der stellenlokale Aufruf einer Schnittstellenoperation eines Akteursphärenverwalters wie ein lokaler Prozeduraufruf realisiert wird und somit sicher ist. Das bedeutet, daß bei einem stellenlokalen Aufruf einer Verwalteroperation keine expliziten Maßnahmen zur wechselseitigen Authentifizierung der aufrufenden Komponente[21] und des aufgerufenen Akteursphärenverwalters durchgeführt werden müssen. Der Vorteil, daß stellenübergreifende Aufrufe von Verwalteroperationen stets von

[21]Dies kann entweder ein Akteur oder ein Akteursphärenverwalter sein.

Akteursphärenverwaltern ausgehen, liegt darin, daß damit die Menge der Komponenten, die explizit wechselseitig zu authentifizieren sind, auf die Menge der stellenübergreifend kooperierenden Akteursphärenverwalter eingeschränkt werden kann. Wäre dies nicht der Fall, so müßten Maßnahmen zur wechselseitigen Authentifizierung einzelner INSEL$^+$–Inkarnationen sowie zum Austausch eines Schlüssels zwischen diesen ergriffen werden, was erheblich mehr Aufwand verursachen würde. Verfahren zum Austausch eines Schlüssels zwischen einzelnen Inkarnationen wurden in [Mar92] entwickelt.

Der Anstoß für die wechselseitige Authentifizierung sowie die Verschlüsselung von Nachrichten bei stellenübergreifenden Zugriffen wird von den Akteursphärenverwaltern durchgeführt. Ihre Funktionalität wird um entsprechende Fähigkeiten erweitert. Dies ist notwendig, da davon ausgegangen wird, daß der Betriebssystemkern der Stellen keine Authentifizierungs– und Verschlüsselungsdienste zur Verfügung stellt (vgl. Abschnitt 6.3.1). Bietet der Betriebssystemkern jedoch Unterstützung für die sichere stellenübergreifende Kommunikation, wie zum Beispiel *Secure Remote Procedure Calls* ([Bir85]), so können diese Dienste genutzt werden.

Zwei Akteursphärenverwalter sind genau dann wechselseitig zu authentifizieren, wenn die beiden Akteursphärenverwalter auf unterschiedlichen Stellen realisiert sind und einer der Akteursphärenverwalter auf dem anderen eine Schnittstellenoperation aufruft. Im Rahmen des Authentifizierungsvorgangs ist ein Schlüssel zwischen den beiden Akteursphärenverwaltern auszutauschen, der u.a. zur Verschlüsselung der Eingabeparameter sowie der Ergebnisparameter der Schnittstellenoperation verwendet werden kann. An das für die wechselseitige Authentifizierung von Akteursphärenverwaltern einzusetzende Authentifizierungsverfahren sind einige Anforderungen zu stellen, die im folgenden kurz erläutert werden.

1. Die Akteursphärenverwalter werden mit den Akteuren dynamisch erzeugt und aufgelöst. Ein Authentifizierungsverfahren, das auf dem Vorhandensein eines Master–Schlüssels zwischen jedem Akteursphärenverwalter und einem Authentifizierungs–Server basiert, ist somit nicht geeignet. Stattdessen müssen die Akteursphärenverwalter in der Lage sein, ihre Authentifizierungsinformation selbst zu generieren und diese unter Verwendung authentischer Basiskanäle an einen bzw. mehrere Authentifizierungs–Server zu übermitteln.

2. Die Anzahl der Akteursphärenverwalter und damit auch die Anzahl wechselseitig zu authentifizierender Akteursphärenverwalter kann groß werden. Statt einer Lösung mit einem zentralen Authentifizierungs–Server ist also ein dezentrales Authentifizierungsverfahren einzusetzen, um die Gefahr des Auftretens von Engpässen zu reduzieren.

3. Die Verschlüsselung der zwischen den Akteursphärenverwaltern auszutauschenden Nachrichten soll möglichst effizient erfolgen. Aufgrund der geringeren Leistung von Public–Key Verfahren (vgl. Kapitel 2) sollte deshalb im Rahmen des Authentifizierungsvorgangs ein Schlüssel für die Anwendung eines Private–Key Verfahrens sicher zwischen den Akteursphärenverwaltern ausgetauscht werden.

Die genannten Anforderungen können durch ein hybrides Authentifizierungs– und Schlüsselaustauschverfahren mit dezentraler Verwaltung der öffentlichen Schlüssel erfüllt werden. Das bedeutet, daß die Authentifizierung zweier Akteursphärenverwalter unter Verwendung eines Public–Key Verfahrens erfolgt und die zwischen den Akteursphärenverwaltern auszutauschenden Nachrichten nach einem Private–Key Verfahren verschlüsselt werden. Der für die Verschlüsselung mit dem Private–Key Verfahren benötigte geheime Schlüssel wird dabei unter

Verwendung der Schlüssel des Public–Key Verfahrens sicher zwischen den jeweiligen Akteursphärenverwaltern ausgetauscht. Jeder Akteursphärenverwalter ist im Besitz eines Schlüsselpaares bestehend aus einem privaten und einem öffentlichen Schlüssel, das er sich unmittelbar nach seiner Erzeugung selbst generiert. Die öffentlichen Schlüssel der Akteursphärenverwalter werden von sogenannten **Schlüsselverwaltern** dezentral verwaltet. Diese stellen Dienste zum Registrieren und zum Erfragen des öffentlichen Schlüssels eines Akteursphärenverwalters zur Verfügung. Neben einem lokalen Schlüsselverwalter auf jeder Stelle gibt es systemweit mehrere globale Schlüsselverwalter. Bei den **globalen Schlüsselverwaltern** sind die öffentlichen Schlüssel der Akteursphärenverwalter verteilt registriert. Das heißt, daß ein globaler Schlüsselverwalter nicht Kenntnis der öffentlichen Schlüssel aller Akteursphärenverwalter hat, sondern lediglich im Besitz der öffentlichen Schlüssel einer Teilmenge der existierenden Akteursphärenverwalter ist. Die **lokalen Schlüsselverwalter** sind die Mittler zwischen den Akteursphärenverwaltern und den globalen Schlüsselverwaltern. Da ihre Schnittstellenoperationen stets lokal von den Akteursphärenverwaltern der jeweiligen Stelle aufgerufen werden können, bilden sie die Basis für die authentische Registrierung und Weitergabe der öffentlichen Schlüssel von Akteursphärenverwaltern sowie deren wechselseitige Authentifizierung. Die Authentizität der öffentlichen Schlüssel beruht somit wesentlich auf der Annahme, daß die stellenlokale Kommunikation sicher ist. Die lokalen Schlüsselverwalter übernehmen ferner die Aufgabe der Zwischenspeicherung öffentlicher Schlüssel von Akteursphärenverwaltern, um die globalen Schlüsselverwalter so weit wie möglich von Schlüsselanfragen zu entlasten. Auf das Verfahren, nach dem die lokalen Schlüsselverwalter entscheiden, bei welchem globalen Schlüsselverwalter der öffentliche Schlüssel eines neu erzeugten Akteursphärenverwalters zu registrieren ist bzw. der öffentliche Schlüssel eines Akteursphärenverwalters zu erfragen ist, wird hier nicht eingegangen. Ein geeignetes Verfahren, das auf einem linearen Hash–Verfahren basiert, wurde in [Wop95] entwickelt. Mit Auflösung eines Akteursphärenverwalter kann auch dessen öffentlicher Schlüssel bei dem für ihn zuständigen globalen Schlüsselverwalter sowie den lokalen Schlüsselverwaltern, die diesen Schlüssel zwischengespeichert haben, gelöscht werden. Die lokalen und globalen Schlüsselverwalter sind analog wie die Akteursphärenverwalter als vertrauenswürdige Komponenten zu konstruieren und somit Bestandteil der Trusted Computing Base.

Zwischen den lokalen und den globalen Schlüsselverwaltern werden öffentliche Schlüssel von Akteursphärenverwaltern in der Regel stellenübergreifend ausgetauscht. Diese Schlüssel müssen authentisch übertragen werden. Es wird also neben der Annahme, daß die stellenlokale Kommunikation zwischen einem Akteursphärenverwalter und einem lokalen Schlüsselverwalter sicher ist, ein authentischer Kommunikationskanal zwischen einem lokalen und einem globalen Schlüsselverwalter benötigt. Dazu wird folgendes als Basis vorausgesetzt:

1. Jeder lokale und globale Schlüsselverwalter besitzt ein Schlüsselpaar, bestehend aus einem privaten und einem öffentlichen Schlüssel.

2. Die globalen Schlüsselverwalter haben Kenntnis des öffentlichen Schlüssels aller lokalen Schlüsselverwalter.

3. Die lokalen Schlüsselverwalter haben Kenntnis des öffentlichen Schlüssels aller globalen Schlüsselverwalter.

Auf die Etablierung dieser Basis wird hier nicht weiter eingegangen. Sie kann durch geeignete Protokolle unter Verwendung von Master–Schlüsseln geschaffen werden (siehe z.B. [Mar92, Wop95]). Aufbauend auf dieser Basis kann der öffentliche Schlüssel eines Akteursphärenverwalters authentisch bei einem globalen Schlüsselverwalter registriert werden sowie

die wechselseitige Authentifizierung zweier Akteursphärenverwalter und die Verteilung eines geheimen Schlüssels zwischen diesen durchgeführt werden. Die dafür benötigten Protokolle werden im folgenden angegeben. Zur Vereinfachung der Protokollbeschreibung wird davon ausgegangen, daß die Schlüsselpaare des Public–Key Verfahrens und damit die öffentlichen Schlüssel der Akteursphärenverwalter nicht erneuert werden. Wäre dies der Fall, müßten neben der Angabe von Schlüsselerneuerungsprotokollen zusätzliche Maßnahmen ergriffen werden, um die Wiedereinspielung von „alten“ nicht mehr gültigen Zertifikaten erkennen zu können. Dies kann entweder durch Zeitstempel, mit denen die Zertifikate versehen werden, oder durch zusätzliche Verwendung von Nonces erreicht werden.

Die Beschreibung der Protokolle erfolgt unabhängig von konkreten Verschlüsselungsverfahren bzw. –algorithmen und den darin verwendeten Schlüssellängen. Es wird lediglich davon ausgegangen, daß die eingesetzten Private–Key und Public–Key Verfahren die in Kapitel 2 erklärten grundlegenden Eigenschaften eines symmetrischen bzw. asymmetrischen Kryptosystems erfüllen. Die Festlegung auf konkrete Algorithmen wird bewußt offen gelassen, da zur Realisierung sicherer Kommunikationskanäle ein Spektrum unterschiedlicher kryptographischer Verfahren und Schlüssellängen zur Verfügung stehen sollte, um die Kanäle flexibel und angepaßt an die unterschiedlichen Sicherheitsanforderungen der Anwendungssysteme konstruieren zu können. Hohe Vertraulichkeits– und Integritätsanforderungen, wie sie u.a. für Systeme aus dem Bankbereich gestellt sind, erfordern den Einsatz starker Kryptographie, also zum Beispiel des symmetrischen IDEA–Algorithmus mit einer Schlüssellänge von 128 Bit und des asymmetrischen RSA–Verfahrens mit mindestens 1024 Bit Schlüsseln. Geringere Vertraulichkeitsanforderungen ermöglichen demgegenüber den Einsatz weniger aufwendigerer und damit in der Regel auch unsicherer Verfahren wie zum Beispiel des DES–Algorithmus mit einem 56 Bit Schlüssel oder des RSA–Verfahrens mit einer Modullänge von lediglich 512 Bit. Als Beispiel für ein System mit relativ geringen Vertraulichkeitsanforderungen kann das in Abschnitt 4.6.2 betrachtete Hausaufgabenverwaltungssystem dienen. Da das eingesetzte Verschlüsselungsverfahren und – insbesondere bei asymmetrischen Kryptosystemen – die Schlüssellänge erhebliche Auswirkungen auf die Leistung der Realisierung haben, können durch die den Sicherheitsanforderungen angepaßte Auswahl Leistungsaspekte stärker berücksichtigt werden. Die Entscheidung, welches Private–Key und welches Public–Key Verfahren jeweils mit welcher Schlüssellänge und ggf. in welchem Betriebsmodus in den Authentifizierungsprotokollen und zur Verschlüsselung eingesetzt wird, kann zur Übersetzungszeit auf Basis geeigneter Annotationen des INSEL$^+$–Programm getroffen werden. Die Festlegung kann entweder systemglobal sein oder spezifisch für einzelne sich in dem INSEL$^+$–Programm durch Operationsaufrufe widerspiegelnde Kommmunikationskanäle erfolgen. Die Annotationskonzepte müssen so gewählt werden, daß der Systementwickler mit ihnen auf hohem Abstraktionsniveau unterschiedliche Vertraulichkeits– und Integritätsanforderungen, zum Beispiel durch Angabe verschiedener Qualitätsstufen wie `keine`, `niedrig`, `mittel` und `hoch`, formulieren kann. Der Systementwickler kann somit – analog zur Festlegung von Zugriffsbeschränkungen – die geforderten Sicherheitseigenschaften der Kommunikationskanäle deklarativ festlegen. Die Realisierung dieser Eigenschaften erfolgt dann automatisiert unter Einsatz geeigneter Verschlüsselungsverfahren. Für die angepaßte Realisierung der Kommunikationskanäle wird ein Spektrum unterschiedlich aufwendiger und sicherer kryptographischer Verfahren benötigt, das zum Beispiel in Form einer Krypto–Bibliothek zur Verfügung gestellt werden kann. Eine derartige Krypto–Bibliothek, in der die wichtigsten symmetrischen und asymmetrischen Verschlüsselungsverfahren inkl. möglicher unterschiedlicher Schlüssellängen und Betriebsmodi in INSEL implementiert sind, wurde in [Rie98] entwickelt. Die Ausarbeitung konkreter Sprach– bzw. Annotationskonzepte zur Festlegung der Sicherheitsanforderungen von Kommunikationskanälen und der in diesem Zusammenhang erforderlichen Kriterien

zur Auswahl geeigneter kryptographischer Verfahren zu einer den Anforderungen angepaßten Realisierung der Kanäle bleibt zukünftigen Arbeiten überlassen.

Die in den folgenden beiden Abschnitten angegebenen Protokolle zur Registrierung des öffentlichen Schlüssels eines Akteursphärenverwalters und zur wechselseitigen Authentifizierung zweier Akteursphärenverwalter werden in einer in der Literatur üblichen Notation beschrieben.

(6.1) Notationen für Authentifizierungsprotokolle

Bei der Beschreibung der Authentifizierungsprotokolle werden folgenden Notationen verwendet:

(a) Ist eine Nachricht N von einem Absender A an einen Empfänger B zu übermitteln, so wird dies wie folgt notiert: $A \longrightarrow B : \; N$.

(b) Ist K eine Komponente, die ein Schlüsselpaar (S_K, P_K) für ein Public–Key Verfahren besitzt, so bezeichnet S_K den privaten Schlüssel und P_K den öffentlichen Schlüssel von K.

(c) Wird eine Nachricht N mit einem Schlüssel S verschlüsselt, so wird für die verschlüsselte Nachricht $\{N\}^S$ geschrieben.

Neben den Nachrichten, die in den einzelnen Protokollschritten zwischen den jeweils beteiligten Komponenten auszutauschen sind, werden auch die entsprechenden Schnittstellenoperationen der Schlüsselverwalter sowie der Akteursphärenverwalter angegeben, deren Aufruf und Ausführung den Austausch dieser Nachrichten bewirkt. Die Nachrichten entsprechen dabei den Eingabe– bzw. Ausgabeparametern dieser Operationen. Die Schnittstellenoperationen und die entsprechenden Nachrichten sind in Tabelle 6.12 zusammengefaßt. Sie werden in den folgenden Abschnitten noch genauer erklärt.

In den Protokollbeschreibungen wird aus Gründen der Übersichtlichkeit auf die Angabe spezieller Fehlerbehandlungsmaßnahmen für die Fälle, in denen ein Protokollschritt zum Beispiel durch ein ungültiges Zertifikat oder eine ungültige Nonce nicht erfolgreich beendet werden kann, verzichtet. Sollte ein Protokollschritt nicht erfolgreich durchgeführt werden können, so terminiert die entsprechende Operation mit einem Fehlerwert und die aufrufende Komponente wiederholt den Operationsaufruf. Speziellere Fehlerbehandlungsmaßnahmen im Rahmen der Ausführung von Authentifizierungsprotokollen werden zum Beispiel in [Wop95] erläutert.

In Kapitel 2 wurde kurz auf formale Methoden zur Verifikation kryptographischer Protokolle eingegangen. In dieser Arbeit wird auf einen formalen Nachweis der Sicherheitseigenschaften der angegebenen Protokolle unter Nutzung dieser Methoden verzichtet, da die Protokolle sich an bekannten „Standardprotokollen“ aus der Literatur orientieren, deren Sicherheitseigenschaften bereits umfassend analysiert und nachgewiesen wurden. Einen guten Überblick hierzu liefert zum Beispiel [Gei95] .

6.3.4.2 Protokoll zur Registrierung eines öffentlichen Schlüssels

Wie oben bereits erwähnt, generiert sich ein Akteursphärenverwalter ASV unmittelbar nach seiner Erzeugung ein Schlüsselpaar (P_{ASV}, S_{ASV}) bestehend aus dem öffentlichen Schlüssel P_{ASV} und dem privaten Schlüssel S_{ASV}. Der öffentliche Schlüssel ist bei dem globalen Schlüsselverwalter GSV_{ASV}, der für die Verwaltung des öffentlichen Schlüssels des Akteursphärenverwalters ASV zuständig ist, zu registrieren. Dazu ruft der Akteursphärenverwalter

Operation	Funktionalität	Aufruf
Lokale Schlüsselverwalter (LSV)		
`LocalRegisterKey`	Weiterleiten des öffentlichen Schlüssels eines ASV zur Registrierung an einen GSV Eingabe: id_{ASV}, P_{ASV} Ausgabe: *flag*	ASV
`LocalGetPublicKey`	Ermitteln des öffentlichen Schlüssels eines ASV Eingabe: id_{ASV} Ausgabe: P_{ASV}	ASV
Globale Schlüsselverwalter (GSV)		
`GlobalRegisterKey`	Registrieren des öffentlichen Schlüssels eines ASV Eingabe: $id_{LSV}, \{id_{ASV}, P_{ASV}\}^{S_{LSV}}$ Ausgabe: *flag*	LSV
`GlobalGetPublicKey`	Ermitteln des öffentlichen Schlüssels eines ASV Eingabe: id_{ASV} Ausgabe: $\{id_{ASV}, P_{ASV}\}^{S_{GSV}}$	LSV
Akteursphärenverwalter (ASV)		
`Authenticate`	1. Schritt der Authentifizierung zweier ASV Eingabe: $\{id_{ASV1}, N_{ASV1}\}^{P_{ASV2}}$ Ausgabe: $\{N_{ASV1}, N_{ASV2}\}^{P_{ASV1}}$	ASV
`ReceiveKey`	2. Schritt der Authentifizierung zweier ASV und Austausch eines geheimen Schlüssels zwischen diesen Eingabe: $\{\{K_{ASV1,ASV2}\}^{S_{ASV1}}, id_{ASV1}, N_{ASV2}\}^{P_{ASV2}}$ Ausgabe: *flag*	ASV

Tabelle 6.12: Schnittstellenoperationen der Schlüsselverwalter und der Akteursphärenverwalter zur Registrierung öffentlicher Schlüssel und zur wechselseitigen Authentifizierung

ASV auf dem lokalen Schlüsselverwalter LSV seiner Stelle die Operation `LocalRegisterKey` auf. Als Eingabeparameter übergibt ASV seinen Identifikator id_{ASV} sowie den öffentlichen Schlüssel P_{ASV} an LSV. In der für die Beschreibung von Authentifizierungsprotokollen üblichen Notation gemäß (6.1) wird dies wie folgt notiert:

$$(1.)\;\; ASV \longrightarrow LSV :\; id_{ASV}, P_{ASV}$$

Der Aufruf von `LocalRegisterKey` erfolgt stellenlokal. Damit wird der öffentliche Schlüssel von ASV also authentisch an den lokalen Schlüsselverwalter LSV übergeben. LSV hat nun die Aufgabe den öffentlichen Schlüssel P_{ASV} authentisch bei dem globalen Schlüsselverwalter GSV_{ASV} zu registrieren. Dies erfolgt durch den im allgemeinen stellenübergreifenden Aufruf der Operation `GlobalRegisterKey` auf dem globalen Schlüsselverwalter GSV_{ASV}. Ist der Aufruf stellenlokal, so können der Identifikator id_{ASV} und der öffentliche Schlüssel P_{ASV} unverschlüsselt an GSV_{ASV} übergeben werden. Anderenfalls ist das Paar (id_{ASV}, P_{ASV}) mit dem privaten Schlüssel S_{LSV} von LSV zu signieren. Mit der Signatur bestätigt LSV, daß P_{ASV} auch wirklich der öffentliche Schlüssel des mit id_{ASV} identifizierten Akteursphärenverwalters ist. Dieses sogenannte Zertifikat wird dann als Eingabeparameter von `GlobalRegisterKey`

an GSV_{ASV} übergeben:

$$(2.)\ \ LSV \longrightarrow GSV_{ASV}:\ id_{LSV}, \{id_{ASV}, P_{ASV}\}^{S_{LSV}}$$

GSV_{ASV} hat laut Voraussetzung Kenntnis des öffentlichen Schlüssels von LSV und kann somit das erhaltene Zertifikat auf seine Authentizität überprüfen. Ist diese Überprüfung erfolgreich, trägt GSV_{ASV} den öffentlichen Schlüssel P_{ASV} in der von ihm verwalteten Liste öffentlicher Schlüssel als den öffentlichen Schlüssel des Akteursphärenverwalters id_{ASV} ein, und anschließend terminiert die Ausführung von `GlobalRegisterKey` und damit auch von `LocalRegisterKey`. Anderenfalls terminiert die Ausführung von `GlobalRegisterKey` mit der Übergabe eines Fehlerwerts an LSV.

6.3.4.3 Wechselseitige Authentifizierung zweier Akteursphärenverwalter

Das Protokoll zur wechselseitigen Authentifizierung von Akteursphärenverwaltern basiert auf dem von Needham und Schroeder entwickelten Protokoll zur wechselseitigen Authentifizierung von Kommunikationspartnern unter Verwendung eines Public–Key Verfahrens ([NS78]). Der Unterschied besteht darin, daß am Ende des Authentifizierungsvorgangs zusätzlich ein geheimer Schlüssel zwischen den Akteursphärenverwaltern ausgetauscht wird. Das folgende Authentifizierungsprotokoll ist immer dann durchzuführen, wenn ein Akteursphärenverwalter stellenübergreifend eine Schnittstellenoperation eines anderen Akteursphärenverwalters aufruft und noch kein geheimer Schlüssel zwischen den beiden Akteursphärenverwaltern ausgetauscht ist.

Seien $ASV1$ und $ASV2$ zwei zu authentifizierende Akteursphärenverwalter, wobei $ASV1$ eine Schnittstellenoperation von $ASV2$ stellenübergreifend aufruft. $ASV1$ benötigt als Basis für die vertrauliche Kommunikation mit $ASV2$ zunächst den öffentlichen Schlüssel von $ASV2$. Dazu wendet sich $ASV1$ durch Aufruf der Operation `LocalGetPublicKey` an seinen lokalen Schlüsselverwalter $LSV1$ und übergibt dabei als Eingabeparameter den Identifikator id_{ASV2} des Akteursphärenverwalters $ASV2$ an $LSV1$.

$$(1.)\ \ ASV1 \longrightarrow LSV1:\ id_{ASV2}$$

$LSV1$ prüft zunächst, ob der öffentliche Schlüssel P_{ASV2} des Akteursphärenverwalters $ASV2$ in der von ihm verwalteten Liste öffentlicher Schlüssel enthalten ist. Ist dies der Fall, übergibt er P_{ASV2} lokal und damit authentisch als Ergebnisparameter von `LocalGetPublicKey` an $ASV1$. Anderenfalls muß sich $LSV1$ den öffentlichen Schlüssel von $ASV2$ von dem globalen Schlüsselverwalter GSV_{ASV2}, der für $ASV2$ zuständig ist, besorgen. Dies erfolgt durch den Aufruf der Operation `GlobalGetPublicKey` auf GSV_{ASV2}. Als Eingabeparameter wird der Identifikator id_{ASV2} übergeben:

$$(2.)\ \ LSV1 \longrightarrow GSV_{ASV2}:\ id_{ASV2}$$

GSV_{ASV2} ermittelt den öffentlichen Schlüssel P_{ASV2} von id_{ASV2} anhand der von ihm verwalteten Liste öffentlicher Schlüssel. Aufgrund der Lebenszeitfestlegungen für INSEL$^+$–Komponenten, die ja auf die Akteursphärenverwalter übertragen werden, ist garantiert, daß der Akteursphärenverwalter $ASV2$ existiert und somit auch der öffentliche Schlüssel P_{ASV2} von $ASV2$ in dieser Liste enthalten ist. GSV_{ASV2} erzeugt unter Verwendung seines privaten Schlüssels $S_{GSV_{ASV2}}$ das Zertifikat $\{id_{ASV2}, P_{ASV2}\}^{S_{GSV_{ASV2}}}$, das dann als Ergebnisparameter von `GlobalGetPublicKey` an $LSV1$ übergeben wird:

$$(3.)\ \ GSV_{ASV2} \longrightarrow LSV_1:\ \{id_{ASV2}, P_{ASV2}\}^{S_{GSV_{ASV2}}}$$

Laut Voraussetzung hat $LSV1$ Kenntnis des öffentlichen Schlüssels von GSV_{ASV2} und kann somit das erhaltene Zertifikat und den darin enthaltenen öffentlichen Schlüssel auf Authentizität überprüfen. Ist P_{ASV2} authentisch, trägt $LSV1$ diesen öffentlichen Schlüssel als den öffentlichen Schlüssel von id_{ASV2} in seine Schlüsselliste ein. Die Speicherung des öffentlichen Schlüssels in der Schlüsselliste von $LSV1$ erfolgt, um weitere Anforderungen des öffentlichen Schlüssels von $ASV2$ lokal erfüllen zu können. Anschließend wird P_{ASV2} lokal und damit authentisch als Ergebnisparameter von `LocalGetPublicKey` an $ASV1$ übergeben.

Um sich gegenüber $ASV2$ zu authentifizieren, ruft $ASV1$ nach Erhalt des öffentlichen Schlüssels von $ASV2$ die Operation `Authenticate` auf $ASV2$ auf. Als Eingabeparameter wird dabei eine mit dem öffentlichen Schlüssel P_{ASV2} von $ASV2$ verschlüsselte Nachricht, bestehend aus dem Identifikator id_{ASV1} von $ASV1$ und einer von $ASV1$ erzeugten Nonce N_{ASV1}, an $ASV2$ übergeben:

$$(4.)\quad ASV1 \longrightarrow ASV2: \ \{id_{ASV1}, N_{ASV1}\}^{P_{ASV2}}$$

$ASV2$ kann die als Eingabeparameter erhaltene verschlüsselte Nachricht mit seinem privaten Schlüssel S_{ASV2} entschlüsseln und somit feststellen, daß ein Authentifizierungswunsch von dem mit id_{ASV1} identifizierten Akteursphärenverwalter vorliegt. Um den Authentifizierungsvorgang fortzusetzen, benötigt $ASV2$ den öffentlichen Schlüssel von id_{ASV1}. Diesen kann er über seinen lokalen Schlüsselverwalter $LSV2$ wie oben beschrieben erhalten:

$$\begin{array}{lll} (5.)\ ASV2 \longrightarrow LSV2 & : & id_{ASV1} \\ (6.)\ LSV2 \longrightarrow GSV_{ASV1} & : & id_{ASV1} \\ (7.)\ GSV_{ASV1} \longrightarrow LSV2 & : & \{id_{ASV1}, P_{ASV1}\}^{S_{GSV_{ASV1}}} \end{array}$$

Nach Erhalt des öffentlichen Schlüssels P_{ASV1} von $ASV1$ verschlüsselt $ASV2$ die mit (4.) erhaltene Nonce N_{ASV1} und eine neu erzeugte Nonce N_{ASV2} mit P_{ASV1}. Mit Übergabe dieser verschlüsselten Nachricht an $ASV1$ terminiert die Ausführung von `Authenticate`.

$$(8.)\quad ASV2 \longrightarrow ASV1: \ \{N_{ASV1}, N_{ASV2}\}^{P_{ASV1}}$$

$ASV1$ entschlüsselt die Nachricht (8.) mit seinem privaten Schlüssel S_{ASV1} und überprüft, ob die darin enthaltene Nonce mit der in (4.) an $ASV2$ übermittelten Nonce übereinstimmt. Ist dies der Fall, ist $ASV2$ gegenüber $ASV1$ authentifiziert, da nur $ASV2$ die Nachricht (4.) entschlüsseln konnte und somit in den Besitz der Nonce N_{ASV1} gelangen konnte. $ASV1$ ist jedoch noch nicht gegenüber $ASV2$ autentifiziert und es ist noch kein geheimer Schlüssel für die Anwendung des Private–Key Verfahrens zwischen $ASV1$ und $ASV2$ ausgetauscht. Um dieses zu erreichen, generiert $ASV1$ einen geheimen Schlüssel $K_{ASV1,ASV2}$, signiert diesen mit seinem privaten Schlüssel S_{ASV1} und verschlüsselt die signierte Nachricht zusammen mit der in (8.) erhaltene Nonce N_{ASV2} mit dem öffentlichen Schlüssel von $ASV2$. Die so signierte und verschlüsselte Nachricht wird durch Aufruf der Operation `ReceiveKey` an $ASV2$ übergeben.

$$(9.)\quad ASV1 \longrightarrow ASV2: \{\{K_{ASV1,ASV2}\}^{S_{ASV1}}, id_{ASV1}, N_{ASV2}\}^{P_{ASV2}}$$

$ASV2$ entschlüsselt die als Eingabeparameter von `ReceiveKey` erhaltene Nachricht zunächst mit seinem privaten Schlüssel S_{ASV2} und überpüft, ob die damit erhaltene Nonce mit der in (8.) an $ASV1$ übermittelten Nonce übereinstimmt. Wenn dies der Fall ist, ist $ASV1$ nun auch gegenüber $ASV2$ authentifiziert. Anschließend wendet $ASV2$ den ihm bekannten öffentlichen Schlüssel P_{ASV1} auf die signierte Nachricht $\{K_{ASV1,ASV2}\}^{S_{ASV1}}$ an und gelangt somit in den Besitz des geheimen Schlüssels $K_{ASV1,ASV2}$. Diesen geheimen Schlüssel trägt

$ASV2$ zusammen mit dem Identifikator id_{ASV1} in eine von ihm verwaltete Liste geheimer Schlüssel ein. Damit terminiert die Ausführung von `ReceiveKey`. Nach Terminierung von `ReceiveKey` trägt $ASV1$ den Schlüssel $K_{ASV1,ASV2}$ zusammen mit dem Identifikator id_{ASV2} in seine Liste geheimer Schlüssel ein.

$ASV1$ und $ASV2$ sind nun wechselseitig authentifiziert und beide sind im Besitz des geheimen Schlüssels $K_{ASV1,ASV2}$, wobei gewährleistet ist, daß lediglich $ASV1$ und $ASV2$ Kenntnis von $K_{ASV1,ASV2}$ haben. Der Schlüssel kann im weiteren für die Verschlüsselung der zwischen $ASV1$ und $ASV2$ stellenübergreifend auszutauschenden Nachrichten genutzt werden.

Wenn $ASV1$ wieder eine Schnittstellenoperation von $ASV2$ aufruft bzw. umgekehrt $ASV2$ eine Schnittstellenoperation von $ASV1$ aufruft, muß nicht erneut das oben angegebene Authentifizierungsprotokoll durchgeführt werden. Die bei jedem erneuten Operationsaufruf notwendige „Reauthentifizierung“ kann implizit durch den bei der erstmaligen Authentifizierung ausgetauschten geheimen Schlüssel sowie die Verwendung von Sequenznummern erfolgen. Als Basis für die Sequenznummern können die zwischen den Akteursphärenverwaltern bei der erstmaligen Authentifizierung ausgetauschten Nonces dienen.

Damit sind die Maßnahmen zur wechselseitigen Authentifizierung von Akteursphärenverwaltern sowie zur Verschlüsselung von Nachrichten vollständig erklärt. Die in Abschnitt 6.3.4.1 an das Authentifizierungsverfahren gestellten Anforderungen sind mit den eingeführten Schlüsselverwaltern, den erklärten funktionalen Erweiterungen der Akteursphärenverwalter und den angegebenen Protokollen erfüllt.

6.3.5 Optimierungen

Der stellenübergreifende Aufruf einer Schnittstellenoperation eines Akteursphärenverwalters ist im Vergleich zu einem lokalen Aufruf teuer. Insbesondere sind die beiden beteiligten Akteursphärenverwalter bei einem stellenübergreifenden Aufruf wechselseitig zu authentifizieren, und die zwischen den Akteursphärenverwaltern auszutauschenden Nachrichten sind zu verschlüsseln. Unter dem Gesichtspunkt der effizienten Realisierung von INSEL$^+$–Systemen sind also Maßnahmen von Interesse, die es ermöglichen, die Anzahl stellenübergreifender Aufrufe von Akteursphärenverwalteroperationen zu reduzieren. In diesem Abschnitt werden entsprechende effizienzsteigernde Maßnahmen angegeben.

Ausgehend von den bisher für die Realisierung von INSEL$^+$–Systemen getroffenen Festlegungen und erarbeiteten Konzepten ergeben sich folgende Ansatzpunkte für die Reduzierung der Anzahl stellenübergreifender Verwalteraufrufe.

1. Depots und DE–Inkarnationen werden auf der Stelle repräsentiert, auf der der Akteur A, von dem sie lebenszeitmäßig abhängig sind, realisiert ist. Im allgemeinen ist A nicht der erzeugende Akteur des Depots bzw. der DE–Inkarnation und auch nicht auf der gleichen Stelle wie dieser realisiert. Zur Erzeugung eines Depots bzw. einer DE–Inkarnation ist also die entsprechende Schnittstellenoperation des Akteursphärenverwalters von A im allgemeinen entfernt aufzurufen. Um ein Depot bzw. eine DE–Inkarnation stets lokal auf der Stelle des Erzeugers erzeugen können und damit den entfernten Akteursphärenverwalteraufruf zu vermeiden, wird auf dieser Stelle ein Anker für die Verwaltung der Komponentenbeschreibung der zu erzeugenden Inkarnation benötigt. Als Anker hierfür wurden in [Win96] sogenannte **Stellvertreter–Verwalter** eingeführt. Die Stellvertreter–Verwalter ermöglichen es, passive Inkarnationen stets lokal erzeugen zu können und damit eine Akteursphäre verteilt auf mehreren Stellen zu realisieren.

Das Konzept der Stellvertreter–Verwalter sowie die zur Integration der Stellvertreter–Verwalter in die bisher erarbeitete Verwalterarchitektur notwendigen Anpassungen der Realisierungskonzepte und –maßnahmen werden im weiteren dieses Abschnitts noch genauer erläutert.

2. Bei einem Zugriff auf eine zugriffskontrollierte Komponente K wird der vor Ausführung des Zugriffs auszuwertende Zugriffsrestriktionsausdruck von dem Akteursphärenverwalter $ASV(K)$ ausgewertet, dessen Sphäre die Komponente K zugeordnet ist. Da dieser Akteursphärenverwalter in der Regel nicht lokal zu dem zugreifenden Akteur ist, erfolgt der entsprechende `CheckAccess`–Aufruf auf dem Akteursphärenverwalter $ASV(K)$ im allgemeinen stellenübergreifend. Wird bei Auswertung des Zugriffsrestriktionsausdrucks jedoch weder auf die Zugriffskontrolliste von K noch auf lokale DE–Inkarnationen von K zugegriffen, so kann der `CheckAccess`–Aufruf auf $ASV(K)$ eingespart werden, indem der Akteursphärenverwalter des zugreifenden Akteurs den Zugriffsrestriktionsausdruck selbst lokal auswertet. Die Auswertung erfolgt dann zu Beginn der Ausführung der Schnittstellenoperation, die der zugreifende Akteur zur Realisierung des Zugriffs auf seinem Akteursphärenverwalter stellenlokal aufruft. Ob ein Zugriffsrestriktionsausdruck von dem Akteursphärenverwalter des zugreifenden Akteurs ohne Unterstützung des Akteursphärenverwalters der Komponente K, auf die zugegriffen werden soll, ausgewertet werden kann, kann statisch durch den INSEL$^+$–Übersetzer analysiert werden. Es ist lediglich zu überprüfen, ob der Zugriffsrestriktionsausdruck ein `IN_ACL`–Prädikat enthält, das als Komponentenidentifikator das Schlüsselwort `THIS` enthält oder ob in dem Zugriffsrestriktionsausdruck Bezeichner lokaler DE–Inkarnationen der Komponente K enthalten sind. Ist eine dieser beiden Bedingungen erfüllt, so ist zur Auswertung des Zugriffsrestriktionsausdrucks die Operation `CheckAccess` auf dem Akteursphärenverwalter aufrufen, dessen Sphäre die Komponente K zugeordnet ist. Anderenfalls kann der Zugriffsrestriktionsausdruck durch den Akteursphärenverwalter des jeweils zugreifenden Akteurs selbst ausgewertet werden.

Im folgenden wird das unter Punkt 1. erwähnte Konzept der Stellvertreter–Verwalter näher erklärt.

Stellervertreter–Verwalter

Die Stellvertreter–Verwalter dienen dazu, passive Inkarnationen, die nicht der Akteursphäre des erzeugenden Akteurs zugeordnet werden, stets stellenlokal erzeugen und verwalten zu können. Stellvertreter–Verwalter werden sowohl für Speicherverwalter als auch für Akteursphärenverwalter erzeugt. Erstere werden benötigt, um Speicher für die Repräsentation einer Inkarnation lokal allokieren zu können, und letztere, um die Komponentenbeschreibung der Inkarnation zu verwalten. Stellvertreter eines Speicherverwalters werden immer zusammen mit Stellvertretern des Akteursphärenverwalters, dem der Speicherverwalter assoziiert ist, erzeugt und aufgelöst. Auf die Stellvertreter von Speicherverwaltern und den sich aus ihrer Existenz ergebenden Konsequenzen für die Speicherverwaltungsmaßnahmen wird im weiteren nicht eingegangen (siehe hierzu [Win96]).

Ein Stellvertreter–Verwalter eines auf einer Stelle S_1 realisierten Akteursphärenverwalters ASV wird genau dann auf einer Stelle S_2 erzeugt, wenn auf S_2 erstmalig eine passive Inkarnation erzeugt werden soll, die der Akteursphäre von ASV zuzuordnen ist. Der erzeugte Stellvertreter–Verwalter stellt die gleichen Schnittstellenoperationen wie der Akteursphärenverwalter ASV zur Verfügung. Die zu erzeugende passive Inkarnation kann dann nach Erzeugung des Stellvertreter–Verwalters stellenlokal durch Aufruf der entsprechenden Schnittstel-

lenoperation des Stellvertreter–Verwalters erzeugt werden. Dabei wird die Komponentenbeschreibung der Inkarnation in den Lebenszeitkeller des Stellvertreter–Verwalters eingetragen.

Mit der Auflösung eines Akteurs bzw. einer Order X sind alle passiven Inkarnationen aufzulösen, die lebenszeitmäßig von X abhängig sind. Bisher wurden die Komponentenbeschreibungen dieser passiven Inkarnationen alle im Lebenszeitkeller des X verwaltenden Akteursphärenverwalters $ASV(X)$ in horizontaler Richtung angebunden an die Komponentenbeschreibung von X verwaltet. Nun werden die Komponentenbeschreibungen der Inkarnationen, die nicht lokal zu X erzeugt werden, in den Lebenszeitkellern der entsprechenden Stellvertreter–Verwalter von $ASV(X)$ verwaltet. Um dennoch mit Auflösung von X alle von X lebenszeitmäßig abhängigen passiven Inkarnationen auflösen zu können, wird als Bestandteil der Komponentenbeschreibung von X eine Liste geführt, die Verweise auf die Stellvertreter–Verwalter enthält, die von X lebenszeitmäßig abhängige Inkarnationen verwalten. Ein Stellvertreter–Verwalter wird in die Stellvertreter–Liste von X eingetragen, wenn auf der Stelle des Stellvertreter–Verwalters erstmalig eine passive Inkarnation erzeugt wird, die lebenszeitmäßig von X abhängig ist. Dieser Eintrag erfolgt im Rahmen der Erzeugung der passiven Inkarnation durch Aufruf der Operation `InsertProxyManager` (siehe Tabelle 6.13) auf dem Akteursphärenverwalter $ASV(X)$. Bei Auflösung von X wird auf allen Stellvertreter–Verwaltern, für die ein Verweis in der Stellvertreter–Liste von X enthalten ist, die Operation `DeleteIncarnations` aufgerufen. Die Ausführung dieser Operation durch diese Stellvertreter–Verwalter bewirkt die Auflösung aller von X lebenszeitabhängigen Inkarnationen, die von dem jeweiligen Stellvertreter–Verwalter verwaltet werden. Damit ist gewährleistet, daß die für INSEL$^+$–Systeme konzeptionell festgelegten Auflösungsregeln auch mit Einführung der Stellvertreter–Verwalter korrekt umgesetzt werden.

Wird ein Akteursphärenverwalter aufgelöst, so können auch sämtliche seiner Stellvertreter–Verwalter aufgelöst werden. Um zu wissen, welche Stellvertreter–Verwalter mit einem Akteursphärenverwalter aufzulösen sind, verwaltet jeder Akteursphärenverwalter eine Liste, die Verweise auf seine Stellvertreter–Verwalter enthält. Ein Stellvertreter–Verwalter trägt sich unmittelbar nach seiner Erzeugung durch Aufruf der Schnittstellenoperation `NewProxyManager` in die Stellvertreter–Liste seines Akteursphärenverwalters ein.

In Tabelle 6.13 sind die erwähnten neuen Schnittstellenoperationen der Akteursphärenverwalter bzw. der Stellvertreter–Verwalter (abgekürzt mit SVV) zusammenfassend angegeben. Die ersten beiden Operationen werden von den Akteursphärenverwaltern bereitgestellt; die dritte Operation ist eine Schnittstellenoperation der Stellvertreter–Verwalter.

Operation	**Funktionalität**	**Aufruf durch**
`NewProxyManager`	Einfügen eines Stellvertreter–Verweises in die Stellvertreter–Liste eines ASV	SVV
`InsertProxyManager`	Einfügen eines Stellvertreter–Verweises in die Stellvertreter–Liste einer Inkarnation	SVV
`DeleteIncarnations`	Auflösen passiver Inkarnationen durch einen Stellvertreter–Verwalter	ASV

Tabelle 6.13: Schnittstellenoperationen der Akteursphärenverwalter zur Realisierung des Konzepts der Stellvertreter–Verwalter

Als Fazit läßt sich festhalten, daß zur lokalen Erzeugung einer passiven Inkarnation I, die le-

benzeitmäßig von einem entfernt realisierten Akteur bzw. einer entfernt realisierten Order X abhängig ist, auf einer Stelle S_1 lediglich dann stellenübergreifende Verwalteraufrufe durchzuführen sind, wenn entweder noch kein entsprechender Stellvertreter–Verwalter auf der Stelle S_1 vorhanden ist oder in der Stellvertreter–Liste des Akteurs bzw. der Order X noch kein Verweis auf diesen Stellvertreter–Verwalter vorhanden ist. Mehrfache Erzeugungen passiver Inkarnationen auf der Stelle S_1, die lebenzeitmäßig von X abhängig sind, können dann ohne stellenübergreifende Kooperation durchgeführt werden. Die mit den Stellvertreter–Verwaltern mögliche stets lokale Erzeugung von Depots und DE–Inkarnationen bietet weiterhin den Vorteil, daß diese Inkarnationen durch ihre erzeugende Komponente lokal – und somit ohne stellenübergreifende Verwalteraufrufe – nutzbar sind. Durch den Einsatz des Konzepts der Stellvertreter–Verwalter wird somit im allgemeinen insgesamt eine Reduzierung der Anzahl stellenübergreifender Aufrufe von Verwalteroperationen und damit eine Effizienzsteigerung erreicht.

Beispiel

Als Beispiel wird die Erzeugung von Konten in dem Kontenverwaltungssystem betrachtet. Ein neues Konto wird als anonymes zugriffskontrolliertes Depot bei Ausführung der Kommunikationsoperation `KontoEroeffnen` durch den K-Akteur `KontoVerwalter` erzeugt. Das Konto–Depot ist lebenszeitmäßig von der Hauptkomponente des Kontenverwaltungssystems abhängig, da sowohl der Depot–Generator `KontoTyp`, bzgl. dem dieses Depot inkarniert wird, als auch der entsprechende Zeiger–Generator `KontoPtrTyp` lokal zur Hauptkomponente definiert sind (vgl. Anhang B). Sind der K-Akteur `KontoVerwalter` (im weiteren abgekürzt mit KV) und die Hauptkomponente (abgekürzt mit KVS) und damit auch die ihnen zugeordneten Akteursphärenverwalter $ASV(KV)$ und $ASV(KVS)$ auf unterschiedlichen Stellen realisiert, ist ohne Nutzung des Stellvertreter–Verwalter Konzepts zur Erzeugung eines neuen Konto–Depots stets ein stellenübergreifender Aufruf der Operation `CreateProtDepotDesc` auf $ASV(KVS)$ durch $ASV(KV)$ erforderlich. Bei Einsatz des Stellvertreter–Verwalter Konzepts sind dagegen lediglich bei der erstmaligen Erzeugung eines Konto–Depots entfernte Verwalteraufrufe durchzuführen. Zum einen wird ein Verweis auf den neuen auf der Stelle von $ASV(KV)$ für $ASV(KVS)$ erzeugten Stellvertreter–Verwalter durch Aufruf der Operation `NewProxyManager` auf $ASV(KVS)$ in dessen Stellvertreter–Liste eingetragen. Zum anderen erfolgt durch Aufruf der Operation `InsertProxyManager` auf $ASV(KVS)$ der Eintrag eines Verweises auf diesen Stellvertreter–Verwalter in die Stellvertreter–Liste der Hauptkomponente[22]. Alle weiteren durch den K-Akteur KV im Rahmen der Ausführung seiner Kommunikationsoperation `KontoEroeffnen` zu erzeugenden Konto–Depots können dann lokal ohne stellenübergreifende Aufrufe erzeugt werden. Dadurch, daß alle Konto–Depots auf der gleichen Stelle wie der K-Akteur KV erzeugt werden, können sie auch lokal durch ihn genutzt werden. Alle Zugriffsoperationen, die der zentrale Kontoverwalter KV auf Konto–Depots aufruft, wie zum Beispiel die Operation `LeseKontonummer` im Rahmen der Ausführung der S-Order `SucheKonto`, können somit lokal durchgeführt werden.

Der Einsatz des Stellvertreter–Verwalter Konzepts ist für dieses Beispiel in Abbildung 6.7 verdeutlicht. In der Abbildung sind die Lebenszeitkeller des Akteursphärenverwalters $ASV(KVS)$ auf der Stelle 1, seines Stellvertreter–Verwalters $ASV(KVS)_Stell$ auf der Stelle 2 sowie des Akteursphärenverwalters $ASV(KV)$ auf der Stelle 2 dargestellt (vgl. hierzu auch Abbildung 6.3). Die Konto–Depots und die zu ihnen lokalen Depots zur Speicherung der Kontobewegungen können aufgrund des auf Stelle 2 vorhandenen Stellvertreter–Verwalters

[22] Als Optimierung bietet es sich hier an, die beiden Stellvertreter–Verweise im Rahmen einer Operation einzutragen und somit lediglich einen stellenübergreifenden Verwalteraufruf durchführen zu müssen.

$ASV(KVS)_Stell$ lokal auf dieser Stelle erzeugt werden. Ihre Komponentenbeschreibungen werden somit im Lebenszeitkeller des Stellvertreter–Verwalters $ASV(KVS)_Stell$ verwaltet. Die Stellvertreter–Liste des Akteursphärenverwalters $ASV(KVS)$ enthält entsprechend dem oben Gesagten einen Verweis auf den Stellvertreter–Verwalter $ASV(KVS)_Stell$. Da die Konto–Depots lebenszeitmäßig von der Hauptkomponente KVS abhängig sind, ist in Stellvertreter–Liste der Hauptkomponente ebenfalls ein Verweis auf den Stellvertreter–Verwalter $ASV(KVS)_Stell$ enthalten.

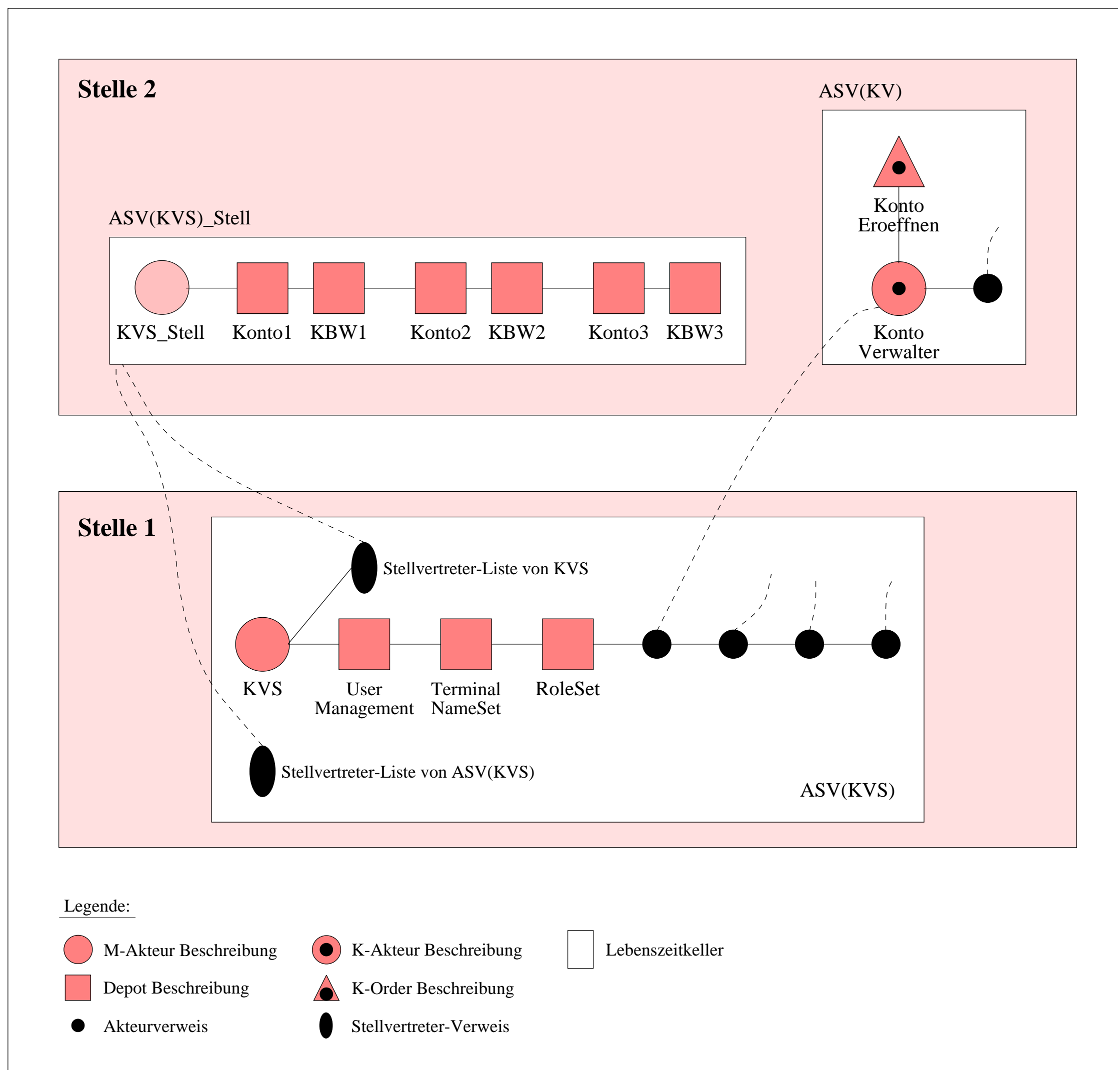

Abbildung 6.7: Beispiel für den Einsatz des Stellvertreter–Verwalter Konzepts

◇

Jeder Stellvertreter–Verwalter generiert analog zu einem Akteursphärenverwalter unmittelbar nach seiner Erzeugung ein Schlüsselpaar und läßt seinen öffentlichen Schlüssel über den jeweiligen lokalen Schlüsselverwalter bei dem für ihn zuständigen globalen Schlüsselverwalter registrieren. Die wechselseitige Authentifizierung und der Austausch eines geheimen Schlüssels bei der stellenübergreifenden Kooperation mit anderen Verwaltern, wozu insbesondere der entsprechende Original–Akteursphärenverwalter gehört, erfolgt dann so wie in Abschnitt 6.3.4 beschrieben.

Mit den Stellvertreter–Verwaltern steht eine geeignete konzeptionelle Basis für die Integration der in Abschnitt 6.3.3 als Alternativen für die Realisierung stellenübergreifender Zugriffe auf passive Inkarnationen angesprochenen Maßnahmen der Migration und Replikation zur Verfügung. Die Stellvertreter–Verwalter bilden die Anker für die bei Einsatz dieser Maßnahmen jeweils lokal benötigten Verwaltungsdatenstrukturen. Für die Migration bedeutet dies, daß die Komponentenbeschreibung einer zu migrierenden Inkarnation auf der Zielstelle in den Lebenszeitkeller des entsprechenden Stellvertreter–Verwalters eingetragen wird. Analog werden bei der Replikation die Beschreibungen der Replikate einschließlich der für die Konsistenzhaltung der Replikate benötigten Informationen in den Lebenszeitkellern der jeweiligen Stellvertreter–Verwalter verwaltet. Der Anlaß für die Erzeugung eines Stellvertreter–Verwalters auf einer Stelle besteht somit nicht nur dann, wenn dort erstmals eine passive Inkarnation erzeugt werden soll, die einer entfernten Akteursphäre angehört, sondern auch im Fall der erstmaligen Migration oder Replikation einer Inkarnation auf dieser Stelle.

6.4 Zusammenfassung

In diesem Kapitel wurden Konzepte und Maßnahmen zur Realisierung von INSEL$^+$–Systemen auf einer Hardware–Basis bestehend aus vernetzten Stellenrechnern angegeben. Der Schwerpunkt lag dabei auf der Erarbeitung der für die Durchsetzung des Rechts eines INSEL$^+$–Systems erforderlichen Realisierungskonzepte und –maßnahmen. Im Gegensatz zu der vielfach bei der Realisierung von Systemen praktizierten *bottom–up* Vorgehensweise wurde ein *top–down* Ansatz verfolgt, nach dem die benötigten Realisierungskonzepte und –verfahren ausgehend von der Sprachebene schrittweise über zwei Realisierungsebenen hinweg entwickelt und konkretisiert wurden. Der wesentliche Vorteil dieser *top–down* orientierten Vorgehensweise besteht darin, daß die Realisierung eines INSEL$^+$–Systems unter Nutzung eines systematisch erarbeiteten Spektrums an Realisierungsalternativen angepaßt an die mit den Sprachkonzepten festgelegten Eigenschaften des Systems erfolgt. Dies ermöglicht es, sowohl die Sicherheitseigenschaften des Systems als auch quantitative Anforderungen, wie zum Beispiel Effizienzanforderungen, adäquat zu berücksichtigen und zu erfüllen.

Die Grundlage für die Realisierung eines INSEL$^+$–Systems bildet die aus der Lebenszeitstruktur des Systems abgeleitete Verwalterarchitektur. Diese Verwalterarchitektur, bestehend aus einer strukturierten, sich dynamisch ändernden Menge untereinander kooperierender Verwalter, wurde in den Abschnitten 6.2 und 6.3 schrittweise den Gegebenheiten der jeweils betrachteten Realisierungsebene entsprechend konkretisiert und angepaßt. Anhand der Übergänge von der programmiersprachlichen Ebene über die Realisierungsebene 1 mit homogenen abstrakten Ressourcen, die von Stellen abstrahieren, zu einem räumlich verteilten System auf der Realisierungsebene 2, die durch vernetzte Stellenrechner chrakterisiert ist, wurden insbesondere die zur Durchsetzung des Rechts eines INSEL$^+$–Systems erforderlichen Konzepte und Verfahren erklärt. Die Aufgabe der zur Durchsetzung des Rechts eines INSEL$^+$–Systems durchzuführenden Zugriffskontrollen sowie der sicheren Verwaltung der für diese Kontrollen benötigten Informationen bzw. Datenstrukturen wird von den Akteursphärenverwaltern wahrgenommen. Mit dem entwickelten Ticket–Konzept steht den Akteursphärenverwaltern neben der Zugriffsrestriktionsausdrucksauswertung eine Realisierungsalternative für die Durchführung der Zugriffskontrollen zur Verfügung. Zur Etablierung vertraulicher und authentischer Kommunikationskanäle zwischen stellenübergreifend kooperierenden Akteursphärenverwaltern wird ein hybrides Authentifizierungs– und Schlüsselaustauschverfahren eingesetzt. Die öffentlichen Schlüssel der Akteursphärenverwalter werden dabei von einer Menge lokaler und globaler Schlüsselverwalter dezentral authentisch verwaltet.

Die Realisierung von INSEL$^+$–Systemen wurde in diesem Kapitel konzeptionell beschrieben. Die vollständige Implementierung der zur Durchsetzung des Rechts eines INSEL$^+$–Systems erarbeiteten Realisierungskonzepte und –maßnahmen sowie deren Integration in eine der vorhandenen INSEL–Ausführungsumgebungen ist auf Basis erster prototypischer Implementierungen von Teilaspekten (siehe [Wop95] für den Bereich der Authentifizierung und [Jac95] für den Bereich der Zugriffskontrolle) in zukünftigen Arbeiten durchzuführen. An den INSEL$^+$–Übersetzer und den zugrundeliegenden Betriebssystemkern sind hohe qualitative Anforderungen gestellt. Sie müssen insbesondere Möglichkeiten zur Konstruktion vertrauenswürdiger Komponenten zur Verfügung stellen.

Dem **INSEL$^+$–Übersetzer** kommt für die Realisierung von INSEL$^+$–Systemen eine zentrale Bedeutung zu. Er führt neben der üblichen syntaktischen und semantischen Analyse eines INSEL$^+$–Programms umfangreiche statische Analysen des Quelltextes durch, um einerseits Aussagen zur Konsistenz des Rechts des entsprechenden INSEL$^+$–Systems zu gewinnen (vgl. Kapitel 5) und zum anderen Informationen für die Auswahl angepaßter Realisierungsalternativen zu erarbeiten. Im Rahmen der semantischen Analyse ist insbesondere die Einhaltung der mit den in Kapitel 4 erklärten Konzepten für Import– und Exportfestlegungen sowie für Operationen–qualifizierte Zeiger statisch festgelegten Zugriffsbeschränkungen vollständig zu überprüfen. Eine wesentliche Aufgabe des INSEL$^+$–Übersetzers besteht in der Erzeugung ausführbaren Maschinencodes. Die korrekte Durchsetzung des Rechts eines INSEL$^+$–Systems beruht dabei wesentlich auf der Erzeugung korrekten Codes durch den INSEL$^+$–Übersetzer. So muß zum Beispiel durch den Übersetzer für jeden Aufruf einer äußeren Operation einer zugriffskontrollierten Komponente korrekter Code zur Durchführung der erforderlichen Zugriffskontrolle generiert werden. Ist dies nicht gewährleistet, kann eine derartige Operation ggf. ohne vorherige bzw. nach nur unvollständiger Zugriffskontrolle ausgeführt werden, was in der Regel eine Rechteverletzung bedeutet. Um die Rechtssicherheit des Gesamtsystems zu gewährleisten, muß der von dem INSEL$^+$–Übersetzer generierte Code vertrauenswürdig sein. Hierzu ist sowohl der INSEL$^+$–Übersetzer als auch der von ihm erzeugte Code gegen Manipulationen besonders zu schützen. Die an den INSEL$^+$–Übersetzer gestellten hohen qualitativen Anforderungen können letztlich nur durch eine formale Verifikation des Übersetzers erfüllt werden. Die hierfür erforderliche Formalisierung und der damit verbundene erhebliche Aufwand gehen weit über den Rahmen dieser Arbeit hinaus und bleiben weiterführenden Arbeiten überlassen.

Kapitel 7

Zusammenfassung und Ausblick

Die Konstruktion sicherer verteilter Rechensysteme ist eine komplexe Aufgabe, die in hohem Maß eine systematische und strukturierte Vorgehensweise erfordert. In dieser Arbeit wurde ein sprachbasierter *top–down* Ansatz zur Konstruktion sicherer verteilter Systeme vorgestellt, mit dem die einleitend formulierten Anforderungen erfüllt werden können. Dieser Konstruktionsansatz ist dadurch charakterisiert, daß alle Eigenschaften eines Systems einschließlich der Sicherheitsanforderungen mit Hilfe progammiersprachlicher Konzepte auf hohem Abstraktionsniveau unabhängig von Realisierungsrandbedingungen festgelegt werden. Das unter Einsatz der Sprachkonzepte konstruierte System und die festgelegten Sicherheitseigenschaften werden automatisiert unter Rückgriff auf ein Spektrum von Sicherheitsmechanismen und –diensten der zugrundeliegenden Ausführungsbasis so realisiert, daß das realisierte System sowohl die funktionalen als auch die Sicherheitsanforderungen durchsetzt.

7.1 Zusammenfassung

Ausgehend von einem Grundlagenteil, der die für diese Arbeit wesentlichen Konzepte und Verfahren aus dem Bereich der sicheren Rechensysteme eingeführt hat, wurde verdeutlicht, daß den für die Implementierung der Systeme genutzten Programmiersprachen eine zentrale Bedeutung in dem Konstruktionsprozeß zukommt. Eine für die Konstruktion sicherer Systeme eingesetzte Programmiersprache muß neben grundlegenden Eigenschaften, die vorrangig die Progammiersicherheit betreffen, Konzepte zur Verfügung stellen, mit denen sich Sicherheitseigenschaften wie zum Beispiel Zugriffsbeschränkungen für die Nutzung von Objekten auf hohem Abstraktionsniveau deklarativ festlegen lassen. Zu den grundlegenden Eigenschaften gehören neben einem Konzept für abstrakte Datentypen und einem strengen Typkonzept u.a. Konzepte zur differenzierten Festlegung von Sichtbarkeitsbereichen und ein Ausnahmebehandlungskonzept. Das Konzept für abstrakte Datentypen sollte die Konstruktion feingranularer Objekte mit differenzierten Operationen ermöglichen. Eine Analyse existierender Programmiersprachen hat gezeigt, daß diese zum einen die genannten grundlegenden Konzepte nur teilweise zur Verfügung stellen und zum anderen keine spezifischen Sprachkonzepte zur deklarativen Festlegung von Sicherheitseigenschaften beinhalten. Vorhandene Ansätze, die sich mit der Bereitstellung spezieller programmiersprachlicher Konzepte zur Realisierung sicherer Systeme beschäftigen, beschränken sich darauf, *low–level* Schutzmechanismen der Hardware– oder Betriebssystemebene, wie zum Beispiel Capabilities oder Zugriffskontrollisten, in eine Programmiersprache zu integrieren.

Die im Rahmen des MoDiS–Projekts entwickelte objekt–basierte Programmiersprache INSEL liefert mit ihrem Konzepterepertoire auf hohem Abstraktionsniveau eine geeignete Ausgangsbasis für die Konstruktion sicherer verteilter Systeme. Sie stellt mit ihren Komponenten– und Strukturierungskonzepten, der strengen Typisierung, den fehlenden Möglichkeiten für direkte Speichermanipulationen sowie den vielfältigen Kapselungs– und Schachtelungsmöglichkeiten zur Festlegung differenzierter Sichtbarkeitsbereiche wichtige für die programmiersprachliche Konstruktion sicherer Systeme benötigte Basiskonzepte bzw. –eigenschaften zur Verfügung. Mit der Sprache INSEL wird die Konstruktion verteilter Anwendungssysteme im Vergleich zu herkömmlichen Ansätzen erheblich vereinfacht, da auf den fehleranfälligen Einsatz von Sprachkonzepten mit niedrigem Abstraktionsniveau verzichtet werden kann. INSEL bietet jedoch keine spezifischen Sprachkonzepte zur expliziten Formulierung von Sicherheitseigenschaften bzw. zur Implementierung anwendungsspezifisch festgelegter Zugriffskontrollpolitiken. Zudem fehlt INSEL bisher ein Ausnahmehandlungskonzept. Entsprechende Konzepte werden von der Sprache INSEL$^+$ als Erweiterung von INSEL bereitgestellt.

INSEL$^+$ erweitert den Konzeptevorrat von INSEL neben einem Benutzer– und Rollenkonzept sowie einem einfachen Ausnahmebehandlungskonzept um zusätzliche Möglichkeiten zur differenzierten Festlegung von Ausführungsumgebungen und um Konzepte zur Konstruktion zugriffskontrollierter Komponenten. Für die Nutzung zugriffskontrollierter Komponenten können komplexe Zugriffsbeschränkungen auf Basis von Zugriffsrestriktionsausdrücken deklarativ festgelegt werden. Die INSEL$^+$–Konzepte unterstützen die Implementierung eines breiten Spektrums anwendungsspezifischer und dynamischer Zugriffskontrollpolitiken. Mit den Möglichkeiten, die zur Definition von Zugriffsrestriktionsausdrücken zur Verfügung stehen, lassen sich komplexe Zugriffsbeschränkungen festlegen, die weit über das hinausgehen, was herkömmlich mit einfachen Zugriffsmatrix–Modellen erfaßbar ist. So können zum Beispiel Zugriffsbeschränkungen auf Basis der Werte globaler und lokaler Variablen oder der Zugriffshistorie sowie des aktuellen Ausführungskontextes von Subjekten formuliert werden. Neben reinen Zugriffskontrollpolitiken lassen sich mit den INSEL$^+$–Konzepten auch Informationsflußbeschränkungen, so wie sie zum Beispiel durch die Multi–Level Sicherheitspolitik des Bell–LaPadula–Modells festgelegt sind, weitestgehend deklarativ implementieren. Hierzu ist lediglich für jedes dieser Politik unterliegende Subjekt und Objekt eine lokale Variable explizit als Label zu definieren, das die Sicherheitsklasse des Subjekts bzw. Objekts angibt. Auf Basis dieser Label sind dann für die äußeren Operationen der Objekte entsprechende Zugriffsrestriktionsaudrücke festzulegen, die die *no write down* und *no read up* Beschränkungen der Politik implementieren.

Aufgrund des hohen Abstraktionsniveaus der INSEL$^+$–Konzepte kann der Systementwickler bei der Festlegung bzw. Implementierung einer Sicherheitspolitik realisierungstechnische Details ausklammern. Damit kann er sich zum einen auf die Formulierung der anwendungsspezifischen Sicherheitsanforderungen konzentrieren und wird zum anderen vom Einsatz realisierungsnaher und an *low–level* Sicherheitsmechanismen orientierten Konzepten entlastet. Sicherheitsprobleme, die in herkömmlichen Ansätzen als Folge eines unsachgemäßen Einsatzes realisierungsnaher Mechanismen auftreten können, werden somit weitestgehend vermieden. Das hohe Abstraktionsniveau der INSEL$^+$–Konzepte liefert gleichzeitig die Basis für das Schließen der semantischen Lücke, die üblicherweise zwischen der Spezifikation von Sicherheitsanforderungen und deren Implementierung besteht. Die Sprachkonzepte von INSEL$^+$ sind so gewählt, daß durch sie insbesondere die systematische Implementierung von Zugriffskontrollpolitiken, die mit dem im Rahmen des SecreDS–Projekt entwickelten Spezifikations– und Modellierungsinstrumentarium (vgl. [Eck93]) formal festgelegt sind, unterstützt wird. Bei kombiniertem Einsatz des SecreDS–Modellierungsinstrumentariums und von INSEL$^+$ als Implementierungssprache kann die angesprochene Lücke unter Nutzung entsprechender

Transformationsregeln deutlich verkleinert werden (siehe [EM97]).

Der in dieser Arbeit vorgestellte sprachbasierte Konstruktionsansatz für sichere Systeme ermöglicht es, bereits zur Übersetzungszeit Aussagen über die Konsistenz der mit den Sprachkonzepten spezifizierten Rechtefestlegungen zu gewinnen. Der für INSEL$^+$–Systeme definierte Konsistenzbegriff ist sehr weitreichend und berücksichtigt im Gegensatz zu anderen Ansätzen, in denen lediglich individuelle objektlokale Eigenschaften erfaßt werden, insbesondere funktionale Abhängigkeiten zwischen Operationen. Die Begriffsdefinition ist von dem Ziel geleitet, daß in einem INSEL$^+$–System mit konistentem Recht keine Ausführungsabbrüche begonnener Operationsausführungen aufgrund fehlender Rechte auftreten können und somit eine wesentliche Voraussetzung für die mit der Wahrnehmung eines Rechts verbundene Leistungsanforderung zu schaffen. Das Recht eines INSEL$^+$–Systems ergibt sich im wesentlichen aus den in dem entsprechenden INSEL$^+$–Programm für die Nutzung der zugriffskontrollierten Komponenten festgelegten Zugriffsbeschränkungen. Da mit den Sprachkonzepten von INSEL$^+$ auch Möglichkeiten für die dynamische Vergabe von Rechten bestehen, kann die Konsistenz des Rechts eines INSEL$^+$–Systems im allgemeinen nicht vollständig statisch nachgewiesen werden. In dieser Arbeit wurden deshalb orientiert an den Ergebnissen, die im Rahmen statischer Analysen hinsichtlich einer Konsistenzaussage erzielbar sind, Differenzierungen des Konsistenzbegriffs vorgenommen. Bei diesen Differenzierungen handelt es sich um die absolute Konsistenz, die potentielle Konsistenz und die absolute Inkonsistenz. Das Recht eines INSEL$^+$–Systems ist absolut konsistent, wenn dessen Konsistenz vollständig statisch nachgewiesen werden kann. Es ist potentiell konsistent, wenn es keine statisch feststellbaren Inkonsistenzen enthält und absolut inkonsistent, wenn mindestens eine Inkonsistenz statisch nachweisbar ist. Die angegebenen statischen Analyseverfahren liefern ausgehend von einem INSEL–Programm Aussagen darüber, ob das Recht des entsprechenden INSEL$^+$–Systems absolut konsistent, potentiell konsistent oder absolut inkonsistent ist. Das Recht eines jeden INSEL$^+$–Systems muß zumindest potentiell konsistent sein. Ist dies nicht der Fall, so sind die vom Übersetzer festgestellten Inkonsistenzen in einem iterativen Prozeß von dem Systementwickler durch Modifikation der formulierten Zugriffsbeschränkungen zu beseitigen.

Besondere Bedeutung kommt der absoluten Konsistenz zu. Mit dem Analyseverfahren für die absolute Konsistenz ist es erstmals für einen funktionale Abhängigkeiten zwischen Operationen erfassenden Konsistenzbegriff möglich, die Konsistenz des Rechts eines Systems vollständig automatisiert statisch nachzuweisen. Aus den Konsistenzkriterien, die im Rahmen dieses Analyseverfahrens überprüft werden, wurden konstruktive Regeln abgeleitet, an denen sich ein Systementwickler hinsichtlich der Konstruktion eines INSEL$^+$–Systems mit absolut konsistentem Recht orientieren kann. In einem INSEL$^+$–System mit absolut konsistentem Recht sind aufgrund der statischen Nachweisbarkeit insbesondere die Möglichkeiten für die Vergabe dynamischer Rechte für die Nutzung von Operationen, die im Kontext anderer Operationen aufgerufen werden, eingeschränkt. Inwieweit ein System so konstruierbar ist, daß das Recht des Systems absolut konsistent ist, hängt somit im allgemeinen von den funktionalen Sollanforderungen an das System und der festgelegten Sicherheitspolitik ab. Durch Orientierung an den konstruktiven Leitlinien werden jedoch die Ursachen dafür erkennbar, warum sich ein System eventuell nicht mit absolut konsistentem Recht konstruieren läßt. Es besteht dann die Möglichkeit, in einem Rückkopplungsprozeß die verursachenden Sollanforderungen kritisch zu betrachten und gegebenenfalls so zu präzisieren bzw. zu modifzieren, daß das Recht des Systems absolut konsistent konstruierbar ist. Sollte das System nicht als Ganzes mit absolut konsistentem Recht konstruierbar sein, können auf Basis der ermittelten Ursachen zumindest einzelne Subsysteme mit absolut konsistentem Recht konstruiert werden. Zugriffsverbote können dann höchstens bei subsystemübergreifenden Zugriffen auftreten. Insgesamt leistet die Arbeit mit dem im Bereich der Konsistenz erarbeiteten Ergebnisse einen

neuen und wesentlichen Beitrag zur Konstruktion von Systemen mit konsistent vergebenen Rechten.

In dem vorgestellten sprachbasierten Ansatz zur Konstruktion sicherer Systeme endet der Konstruktionsprozeß aus Sicht des Systementwicklers mit der Implementierung des Systems als INSEL$^+$–Progamm. Mit diesem INSEL$^+$–Programm sind die Eigenschaften des Systems einschließlich der Sicherheitseigenschaften festgelegt. Die Realisierung des mit den Sprachkonzepten konstruierten Systems erfolgt vollständig automatisiert durch den entwickelten INSEL$^+$–Übersetzer unter Rückgriff auf ein Spektrum von Realisierungskonzepten und –mechanismen, das von der zugrundeliegenden Ausführungsbasis zur Verfügung zu stellen ist. Aufgrund des hohen Abstraktionsniveaus der INSEL$^+$–Sprachkonzepte bestehen im Rahmen der automatisierten Realisierung Freiheitsgrade, die es ermöglichen, die jeweils eingesetzten Realisierungskonzepte und –verfahren angepaßt an die anwendungsspezifischen Anforderungen auszuwählen. Im Gegensatz zu herkömmlichen *bottom–up* orientierten Ansätzen wurde in dieser Arbeit ein *top–down* Ansatz verfolgt, nach dem das benötigte Spektrum an Realisierungskonzepten ausgehend von der Sprachebene schrittweise über zwei Realisierungsebenen hinweg entwickelt und konkretisiert wurde. Der Schwerpunkt lag dabei auf der Erarbeitung der für die Durchsetzung des Rechts eines INSEL$^+$–Systems erforderlichen Verfahren und Mechanismen. Die Aufgabe der hierfür durchzuführenden Zugriffskontrollen einschließlich der sicheren Verwaltung der für diese Kontrollen benötigten Informationen wird durch eine strukturierte, sich dynamisch ändernde Menge kooperierender Verwalter dezentral wahrgenommen. Die Struktur dieser Verwalterarchitektur ist aus den mit den Sprachkonzepten festgelegten Eigenschaften des INSEL$^+$–Systems abgeleitet und ermöglicht damit die systematische Integration der notwendigen Sicherheitsmaßnahmen. Für die Durchführung der erforderlichen Zugriffskontrollen wurden mit der Zugriffsrestriktionsauswertung und dem Ticket–Konzept Alternativen entwickelt, die es ermöglichen, die Zugriffskontrollen effizient und angepaßt an die anwendungsspezifischen Eigenschaften zu realisieren. Neben den neu entwickelten Konzepten werden für die Realisierung der festgelegten Sicherheitseigenschaften auch Standardverfahren und –mechanismen, wie zum Beispiel kryptographische Verfahren und Authentifizierungsverfahren, genutzt. Im Gegensatz zu herkömmlichen Ansätzen werden diese jedoch automatisiert und angepaßt an die Anwendungsanforderungen eingesetzt. So wird zum Beispiel zur Etablierung eines vertraulichen und authentischen Kommunikationskanals zwischen zwei stellenübergreifend kooperierenden Verwaltern ein hybrides Authentifizierungs– und Schlüsselaustauschverfahren verwendet. Die öffentlichen Schlüssel der Verwalter werden dabei von einer Menge lokaler und globaler Schlüsselverwalter dezentral authentisch verwaltet.

Ein wesentlicher Vorteil der sprachbasierten *top–down* orientierten Vorgehensweise liegt darin, daß die Realisierung eines Systems angepaßt an die mit den Sprachkonzepten festgelegten Eigenschaften des Systems und damit orientiert an den anwendungsspezifischen Anforderungen erfolgt. So werden zum Beispiel Zugriffskontrollisten lediglich für die Komponenten eines Systems verwaltet, für die Zugriffsbeschränkungen auf Basis des `IN_ACL`–Prädikats festgelegt sind. Der zu leistende Aufwand für die zur Durchsetzung der Sicherheitsanforderungen benötigten Sicherheitsmaßnahmen kann somit minimiert werden. Dies zeigt sich auch daran, daß in INSEL$^+$ implementierte Anwendungssysteme, für die keine Sicherheitsanforderungen gestellt sind, von dem mit dem Einsatz von Sicherheitsmaßnahmen verbundenen Mehraufwand – abgesehen von längeren Übersetzungszeiten – nicht betroffen sind.

Der in dieser Arbeit vorgestellte sprachbasierte Konstruktionsansatz für sichere Systeme eröffnet aufgrund der dargelegten Vorteile im Vergleich zu herkömmlichen Ansätzen neue Perspektiven für die Konstruktion qualitativ hochwertiger sicherer verteilter Systeme.

7.2 Ausblick

Aus der vorliegenden Arbeit ergeben sich zahlreiche weiterführende Aufgabenstellungen sowohl konzeptioneller als auch praktischer Art.

Im Bereich der Sprachkonzepte werden zusätzliche Konzepte benötigt, um neben Zugriffskontrollpolitiken und einfachen auf Labeln basierenden Informationsflußpolitiken auch weitere Sicherheitseigenschaften deklarativ implementieren zu können. Hierzu gehören zum einen Konzepte zur Formulierung komplexerer Informationsflußpolitiken, wie sie zum Beispiel mit dem im SecreDS–Projekt erarbeiteten Spezifikationsinstrumentarium (vgl. [Eck93]) formulierbar sind, und zum anderen Sprachkonstrukte, mit denen Sicherheitsanforderungen an Kommunikationskanäle festgelegt werden können. Für letztere wurde in Kapitel 6 mit den Annotationskonzepten zur Formulierung unterschiedlicher Authentizitäts–, Vertraulichkeits– und Integritätsanforderungen an Kommunikationskanäle bereits ein erster Ansatz aufgezeigt. Die konkrete Ausarbeitung der entsprechenden Sprach– bzw. Annotationskonzepte und der erforderlichen Kriterien zur Auswahl geeigneter kryptographischer Verfahren zu einer den Anforderungen angepaßten Realisierung der Kommmunikationskanäle bleibt zukünftigen Arbeiten vorbehalten. In diesen ist weiterhin ein umfassendes Ausnahmebehandlungskonzept für INSEL$^+$ zu entwickeln, in das das in Kapitel 4 vorgestellte Konzept zur Signalisierung und Behandlung von Zugriffsverboten zu integrieren ist. Darüberhinaus sind Sprachkonzepte von Interesse, mit denen Zuverlässigkeitsaspekte und weitere qualitative Anforderungen erfaßt werden können. Beispiele hierfür sind Sprachkonstrukte zur Festlegung von Transaktionseigenschaften für Operationen oder zur Formulierung der maximalen Ausführungdauer einer Operation, um Echtzeitanforderungen implementieren zu können.

Eine Fragestellung, die insbesondere für langlebige Systeme relevant ist, betrifft die Möglichkeit, die Sicherheitspolitik dynamisch, d.h. zur Laufzeit des entsprechenden INSEL$^+$–Systems, ohne vollständige Neuübersetzung des INSEL$^+$–Programms zu modifizieren bzw. zu erweitern. Mit den vorhandenen INSEL$^+$–Konzepten ist dies nur sehr eingeschränkt möglich. So können die Zugriffsbeschränkungen zwar auf Basis dynamischer Objekte, wie Zugriffskontrollisten oder Variablen formuliert werden. Eine Änderung des Zugriffsrestriktionsausdrucks einer Operation zur Laufzeit ist jedoch nicht möglich. Hierfür sind in weiterführenden Arbeiten geeignete Lösungen zu entwickeln. Ein Lösungsansatz könnte darin bestehen, die funktionalen Anforderungen und die Sicherheitsanforderungen getrennt voneinander in zwei Teilen zu implementieren. Bei Änderung der Sicherheitsanforderungen zur Laufzeit wäre dann lediglich der Teil, in dem die Sicherheitseigenschaften festgelegt werden, entsprechend zu modifizieren, neu zu übersetzen und dann dynmamisch zu dem laufenden INSEL$^+$–System hinzuzubinden. Von Interesse ist in diesem Zusammenhang auch der Einsatz generischer Konzepte, die es ermöglichen, konkrete Ausprägungen einer Sicherheitspolitik erst zur Laufzeit festlegen zu können, und die damit gleichzeitig die Basis für die Wiederverwendbarkeit implementierter Politiken bilden. Die Erarbeitung entsprechender Konzepte bezogen auf die dynamische Erweiterbarkeit von INSEL–Systemen um funktionale Anforderungen ist Gegenstand aktueller Arbeiten im Rahmen des MoDiS–Projektes.

Für die systematische sprachbasierte Konstruktion sicherer Systeme werden geeignete Werkzeuge benötigt, die den Systementwickler im Konstruktionsprozeß unterstützen. Hierzu gehört insbesondere ein Visualisierungs– und Analyse–Tool, mit dem zum Beispiel neben funktionalen Abhängigkeiten zwischen Komponenten auch bestehende Rechte dargestellt und die Auswirkungen von Rechteänderungen analysiert werden können. Als Ausgangsbasis hierfür kann der im MoDiS–Projekt entwickelte verteilte Interpreter und Debugger DAViT (*Distributed Analyzing and Visualizing Tool*) [Wei97] genutzt werden. Mit DAViT kann die verteilte in-

terpretative Ausführung von INSEL–Programmen mit Hilfe einer graphischen Benutzungsoberfäche, die u.a. die verschiedenen Abhängigkeiten zwischen INSEL–Komponenten visualisiert, überwacht und gesteuert werden. Im Rahmen dieser Arbeit wurde DAViT in einem ersten Schritt um Funktionalitäten zur Ausführung von INSEL$^+$–Programmen und zur Darstellung von zugriffskontrollierten Komponenten erweitert. In weiteren Schritten ist DAViT um Funktionalitäten zur Visualisierung und Analyse der mit den INSEL$^+$–Konzepten spezifizierten Rechtefestlegungen anzureichern.

Das langfristige Ziel besteht darin, eine umfassende graphische Entwicklungsumgebung bereitzustellen, die die Konstruktion sicherer verteilter Systeme nach einem sprachbasierten *top–down* Ansatz in allen Phasen des Konstruktionsprozesses unterstützt. In diesem Zusammenhang sind die in ersten Arbeiten (vgl. [EM97]) angegebenen informellen Regeln zur Transformation eines mit dem SecreDS–Modellierungsinstrumentariums formal spezifizierten Systems und seiner Sicherheitspolitik in ein INSEL$^+$–Programm zu verfeinern und stärker zu formalisieren.

In zukünftigen praktischen Arbeiten sind die in Kapitel 6 erarbeiteten Realisierungskonzepte und –maßnahmen auf der Basis erster prototypischer Implementierungen von Teilaspekten (vgl. [Jac95, Wop95]) vollständig zu implementieren und in die im Rahmen des MoDiS–Projekts entwickelte Ausführungsumgebung und Betriebssystem–Infrastruktur zu integrieren. Hierbei ist insbesondere die von den Akteursphärenverwaltern angewandte Strategie für die Ticketvergabe auf Basis der angegebenen Kriterien detailliert auszuarbeiten und anhand praktisch gewonnener Erfahrungswerte zu optimieren.

Um die Rechtssicherheit eines realisierten INSEL$^+$–Systems gewährleisten zu können, sind an den INSEL$^+$–Übersetzer und die zugrundeliegende Betriebssystem–Infrastruktur hohe qualitative Anforderungen gestellt. So müssen der generierte Code und die Betriebssystemkomponenten vertrauenswürdig und gegen Manipulationen geschützt sein. Dies kann letztlich nur durch eine formale Verifikation des Übersetzers und der sonstigen für die Realisierung eingesetzten Komponenten erreicht werden. Die Bearbeitung dieser umfangreichen Frage– und Aufgabenstellung bleibt langfristig durchzuführenden Arbeiten überlassen.

Weiterführende Arbeiten im Bereich der Konsistenz des Rechts eines INSEL$^+$–Systems betreffen zunächst die vollständige Implementierung der vorgestellten statischen Analyseverfahren innerhalb des INSEL$^+$–Übersetzers. Wesentliche hierfür notwendige Erweiterungen an dem vorhandenen INSEL–Übersetzer wurden im Rahmen dieser Arbeit bereits durchgeführt. In einem weiteren Schritt sind die Ergebnisse der statischen Konsistenzanalysen in DAViT zu visualisieren, so daß dem Systementwickler zum Beispiel Inkonsistenzen und Möglichkeiten zu deren Behebung graphisch veranschaulicht werden. Wie in Kapitel 5 aufgezeigt, ist für ein INSEL–System mit lediglich potentiell konsistentem Recht durch weitergehende statische Analysen zu überprüfen, ob die Voraussetzungen zur Durchführung der Laufzeitmaßnahmen, die das tatsächliche Auftreten von Inkonsistenzen zur Laufzeit vermeiden sollen, erfüllt sind. Die detaillierte Ausarbeitung der hierfür erforderlichen Kriterien sowie die Implementierung der entsprechenden Analyseverfahren und der Laufzeitmaßnahmen muß in zukünftigen Arbeiten erfolgen.

Insgesamt zeigt sich, daß sich aus dem in dieser Arbeit präsentierten sprachbasierten Ansatz und den entwickelten Konzepten und Verfahren zahlreiche weiterführende Frage– und Aufgabenstellungen ergeben, durch deren Bearbeitung weitere Fortschritte hinsichtlich der Konstruktion sicherer verteilter Systeme erzielt werden können.

Anhang A

INSEL$^+$–Syntax

Im folgenden ist die Syntax der Sprache INSEL$^+$ auszugsweise angegeben. Es sind lediglich die Syntaxregeln angegeben, die im Hinblick auf die Syntaxregeln der Sprache INSEL neu sind bzw. modifiziert wurden. Die Syntax der Sprache INSEL ist vollständig in [RW96] enthalten. Die Nummern der einzelnen Regeln beziehen sich auf die in [RW96] eingeführte Numerierung.

Interface–Part

Regel 8 *<interface-part> ::=*
<empty>
| **IS** *<declarative-part>*
<sync-part>
<view-part>
<acl-part>
<access-restriction-part>
END *<opt-identifier>*

Import– und Exportfestlegungen

Regel 10 *<limitation-part> ::=*
<import-part> <export-part>

Die Regeln 118 bis 123 ersetzen die INSEL–Syntaxregel mit der Nummer 11.

Regel 118 *<import-part> ::=*
<empty>
| **IMPORT** *<name-list>* **';'**

Regel 119 *<export-part> ::=*
<empty>
| **EXPORT** *<export-list>* **';'**

Regel 120 *<export-list> ::=*
<export-specification>
| *<export-list>* **';'** *<export-specification>*

Regel 121 *<export-specification> ::=*
<export-name-list>
| *<export-name-list>* **TO** *<identifier-list>*

Regel 122 *<export-name-list> ::=*
<export-name>
| *<export-name-list>* **','** *<export-name>*

Regel 123 *<export-name> ::=*
<identifier>
| *<export-name>* **'.'** *<identifier>*

Operationen–qualifizierte Zeigergeneratoren

Regel 124 *<pointer-type-definition> ::=*
POINTER TYPE *<identifier>* **IS ACCESS** *<name>* *<with-part>*

Regel 125 *<with-part> ::=*
<empty>
| **WITH** *<identifier-list>*

Regel 126 *<pointer-subtype-definition> ::=*
POINTER SUBTYPE *<identifier>* **IS** *<name>* *<with-part>*

Rollen

Regel 127 *<role-part> ::=*
ROLES *<identifier-list>* **';'**

Zugriffskontrollierte DA–Generatoren

Regel 128 *<protected-generator> ::=*
PROTECTED *<da-generator>*

View–Part

Regel 129 *<view-part> ::=*
<empty>
| **VIEWS** *<view-list>* **';'**

Regel 130 *<view-list> ::=*
<view-definition>
| *<view-list>* **';'** *<view-definition>*

Regel 131 *<view-definition> ::=*
<identifier> ':' *<name-list>*
| *<name-list>*

.... **ACL–Part**

Regel 132 *<acl-part> ::=*
<empty>
| **ACL** *<acl-list>* ';'

Regel 133 *<acl-list> ::=*
<acl-entry>
| *<acl-list>* ';' *<acl-entry>*

Regel 134 *<acl-entry> ::=*
<view-name> ':' *<user-role-list>*

Regel 135 *<user-role-list> ::=*
<user-role-entry>
| *<user-role-list>* ',' *<user-role-entry>*

Regel 136 *<user-role-entry> ::=*
'(' *<user-identifier>* ',' *<role-identifier>* ',' *<mode>* ')'

Regel 137 *<user-identifier> ::=*
<name>
| **ALL** `/* steht für -1 */`

Regel 138 *<role-identifier> ::=*
<name>
| **ALL** `/* steht für -1 */`

Regel 139 *<mode> ::=*
'+' | '-'

.... **Zugriffsrestriktions–Part**

Regel 140 *<access-restriction-part> ::=*
ACCESS RESTRICTIONS *<access-restriction-list>* ';'

Regel 141 *<access-restriction-list> ::=*
<access-restriction-entry>
| *<access-restriction-list>* ';' *<access-restriction-entry>*

Regel 142 *<access-restriction-entry> ::=*
<view-name> ':' *<acc-expression>*

Zugriffsrestriktionsausdrücke

Regel 143 *<acc-expression> ::=*
<acc-relation>
| <acc-relation> <logical-operator> <acc-relation>

Regel 144 *<acc-relation> ::=*
<acc-term>
| <acc-term> <relational-operator> <acc-term>

Regel 145 *<acc-term> ::=*
<acc-primary>
| <unary-operator> <acc-primary>

Regel 146 *<acc-primary> ::=*
<literal>
| <caller-attribut>
| <inselp-path>
| <name>
| <user-identifier>
| <role-identifier>
| <acc-predicate>
| '(' <acc-expression> ')'

Regel 147 *<caller-attribut> ::=*
<name>

Regel 148 *<inselp-path> ::=*
<identifier>
| <identifier> '.' <identifier>

Zugriffsrestriktionsprädikate

Regel 149 *<acc-predicate> ::=*
IN_ACL '(' <user-identifier> ',' <role-identifier> ','
<comp-identifier> ',' <this-or-view-name> ')'
| **ACCESSED** '(' <user-identifier> ',' <comp-identifier> ','
<any-or-op-name> ')'

Regel 150 *<comp-identifier> ::=*
<name>
| **THIS**

Regel 151 *<this-or-view-name> ::=*
<view-name>
| **THIS**

Regel 152 *<any-or-op-name> ::=*
<op-name>
| **ANY**

Literale

Regel 90 *<literal> ::=*
<character-literal>
| *<string-literal>*
| *<integer-literal>*
| *<real-literal>*
| *<boolean-literal>*

Operatoren

Regel 95 *<logical-operator> ::=*
AND
| **OR**

Regel 96 *<relational-operator> ::=*
'='
| **'<'**
| **'>'**
| **'/='**
| **'<='**
| **'>='**

Regel 100 *<unary-operator> ::=*
'+'
| **'–'**
| **NOT**

Anhang B

INSEL$^+$–Beispielprogramm: Kontenverwaltungssystem einer Bank

Im folgenden ist das INSEL$^+$–Programm, das das in Abschnitt 4.6.1 informell erläuterte Kontenverwaltungssystem einer Bank implementiert, aufgelistet. Aus Platzgründen sind einige Teile des Programms nur auszugsweise angegeben (jeweils angedeutet durch "...").

```
PROCESS KontenVerwaltungsSystem IS

  -- Rollen des Systems; die Rolle 'SysAdmin' ist vordefiniert
  ROLES
    Kunde,
    Kundenbetreuer,
    Kassierer,
    Bankleiter;

-- Import des vordefinierten K-Akteur Generators 'TermType' und des mit 'TermType'
-- qualifizierten  Zeigergenerators 'TermPointer' fuer abstrakte Terminals
#include "terminal.inp"

-- Import der in INSEL+ global vordefinierten Generatoren und Komponenten
#include "global.inp"

-------------------------------------------------------------------------------------------
-- Benutzerverwaltung und Benutzerzugang                                                --
-------------------------------------------------------------------------------------------

  PROCEDURE TYPE SPEC CreateUserRepForRole (Role: IN string; Term: IN TermPointer);

-- Import der vordefinierten Generatoren fuer die Benutzerverwaltung und den Benutzerzugang
#include "user.inp"

  DEPOT UserManagement : UserManagementType;

  -- Prozedur zum Initialisieren der globalen Rollenmenge
  PROCEDURE TYPE InitRoles IS
  BEGIN
    UserManagement.RoleSet.Insert("Kunde");
    UserManagement.RoleSet.Insert("Kundenbetreuer");
    UserManagement.RoleSet.Insert("Kassierer");
```

```
    UserManagement.RoleSet.Insert("Bankleiter");
    UserManagement.RoleSet.Insert("SysAdmin");
  END InitRoles;

  -- Anzahl der Terminals im System
  NumberOfInitialTerminals : CONSTANT integer := 2;

  -- Prozedur zum Initialisieren der globalen Terminalnamens-Menge
  PROCEDURE TYPE InitTerminals IS
  BEGIN
    UserManagement.TerminalNameSet.Insert("hpspies1");
    UserManagement.TerminalNameSet.Insert("hpspies2");
  END InitTerminals;

---------------------------------------------------------------------------------------------
-- Generatoren und Komponenten fuer die virtuelle Zeit des Kontenverwaltungssystems        --
-- (auszugsweise)                                                                          --
---------------------------------------------------------------------------------------------

  -- Basistypen fuer die virtuelle Zeit
  TYPE SekMinTyp IS 0..60;         -- Sekunde und Minute
  TYPE StdTyp    IS 0..24;         -- Stunde
  TYPE TagTyp    IS 1..31;         -- Tag
  TYPE MonatTyp  IS 1..12;         -- Monat
  TYPE JahrTyp   IS 1900..10000;   -- Jahr
  Type WTagTyp   IS 1..7;          -- Wochentag

  -- Typ fuer das Datum
  TYPE DatumTyp IS RECORD
    Tag       : TagTyp;
    Monat     : MonatTyp;
    Jahr      : JahrTyp;
  END RECORD;

  -- Typ fuer die virtuelle Zeit
  TYPE ZeitTyp IS RECORD
    Datum     : DatumTyp;
    Wochentag : WTagTyp;
    Stunde    : StdTyp;
    Minute    : SekMinTyp;
    Sekunde   : SekMinTyp;
  END RECORD;

  -- globale virtuelle Zeit
  Zeit : ZeitTyp;

  -------------------------------------------------------------------------------------------
  -- Zugriffskontrollierter K-Akteur-Generator fuer die Uhr des Systems                    --
  -------------------------------------------------------------------------------------------
  PROTECTED TASK TYPE SPEC UhrTyp (InitTag   : IN TagTyp;
                                  InitMonat : IN MonatTyp;
                                  InitJahr  : IN JahrTyp;
                                  InitWTag  : IN WTagTyp;
                                  InitStd   : IN StdTyp;
                                  InitMin   : IN SekMinTyp;
                                  InitSek   : IN SekMinTyp;)
    IMPORT ZeitTyp, Zeit, ...;
  IS
    ENTRY TYPE SPEC SetzeZeit (NeueZeit : IN ZeitTyp);
    ...
```

```
    ACCESS RESTRICTIONS
      Create    : Caller.User = System;
      SetzeZeit : Caller.Role = SysAdmin;
      ...
  END UhrTyp;

  PROTECTED TASK TYPE UhrTyp (...)
    ...
  END UhrTyp;

  ErlaubterZugriff : boolean;

  -- die Uhr des Systens
  TASK Uhr : UhrTyp(27, 8, 1966, 6, 16, 0, 0) STATUS ErlaubterZugriff;

---------------------------------------------------------------------------------------------
-- Generatoren und Komponenten der Kontenverwaltung                                        --
---------------------------------------------------------------------------------------------

  -- Daten eines Kunden
  TYPE AllgKundenDatenTyp IS RECORD
    Vorname  : string;
    Nachname : string;
  END RECORD;

  TYPE PersKundenDatenTyp IS RECORD
    GebDatum      : DatumTyp;
    Adresse       : string;
    Gehalt        : real;
    Familienstand : string;
    Beruf         : string;
  END RECORD;

  TYPE KundenDatenTyp IS RECORD
    AllgKundenDaten : AllgKundenDatenTyp;
    PersKundenDaten : PersKundenDatenTyp;
  END RECORD;

  POINTER TYPE KundenDatenPtrTyp IS ACCESS KundenDatenTyp;

  -------------------------------------------------------------------------------------------
  -- Zugriffskontrollierter Depot-Generator fuer Liste der Kontobewegungen eines Kontos    --
  -------------------------------------------------------------------------------------------
  -- Spezifikations-Part
  PROTECTED DEPOT TYPE SPEC KontobewegungsListenTyp
    IMPORT Zeit, DatumTyp, TermPointer;
  IS
    PROCEDURE TYPE SPEC FuegeEin (Zweck: IN string; Betrag: IN real; Kontostand: IN real);
    PROCEDURE TYPE SPEC ListeKontobewegungen (Von, Bis: IN DatumTyp; Term: IN TermPointer);
    PROCEDURE TYPE SPEC ListeKontoauszug (Term: IN TermPointer);

    SYNC READER_WRITER WITH
      READER : ListeKontobewegungen, ListeKontoauszug
      WRITER : FuegeEin;

    ACCESS RESTRICTIONS
      FuegeEin             : Caller.ConGen = KontoTyp.Einzahlen OR
                             Caller.ConGen = KontoTyp.Abheben;
      ListeKontobewegungen : Caller.ConGen = KontoTyp.ZeigeKontobewegungen;
      ListeKontoauszug     : Caller.ConGen = KontoTyp.ZeigeKontoauszug;
```

```
    Create              : Caller.ConGen = KontoTyp;
END KontobewegungsListenTyp;

-- Implementierungs-Part
PROTECTED DEPOT TYPE KontobewegungsListenTyp
  IMPORT Zeit, DatumTyp, TermPointer;
IS
  POINTER TYPE EintragPtrTyp;
  TYPE Eintrag IS RECORD
    LfdNum     : integer;
    Datum      : DatumTyp;
    Zweck      : string;
    Betrag     : real;
    Kontostand : real;
    Next       : EintragPtrTyp;
    Prev       : EintragPtrTyp;
  END RECORD;
  POINTER TYPE EintragPtrTyp IS ACCESS Eintrag;

  Beginn, Ende, LetzterAuszug : EintragPtrTyp := NULL;
  LfdNum : integer := 0;

  -- Zugriffsoperation FuegeEin
  PROCEDURE TYPE FuegeEin (Zweck: IN string; Betrag: IN real; Kontostand: IN real) IS
    NeuerEintrag : EintragPtrTyp;
  BEGIN
    NeuerEintrag := NEW(EintragPtrTyp, Eintrag);
    NeuerEintrag.LfdNum      := LfdNum + 1;
    NeuerEintrag.Datum.Tag   := Zeit.Datum.Tag;
    NeuerEintrag.Datum.Monat := Zeit.Datum.Monat;
    NeuerEintrag.Datum.Jahr  := Zeit.Datum.Jahr;
    NeuerEintrag.Zweck       := Zweck;
    NeuerEintrag.Betrag      := Betrag;
    NeuerEintrag.Kontostand  := Kontostand;
    IF Beginn = NULL THEN
      Beginn := NeuerEintrag;
      Beginn.Prev := NULL;
      LetzterAuszug := Beginn;
    ELSE
      NeuerEintrag.Prev := Ende;
      Ende.Next := NeuerEintrag
    END IF;
    Ende := NeuerEintrag;
  END FuegeEin;

  -- Zugriffsoperation ListeKontobewegungen
  PROCEDURE TYPE ListeKontobewegungen (Von, Bis: IN DatumTyp; Term: IN TermPointer) IS
    ...
  END ListeKontobewegungen;

  -- Zugriffsoperation ListeKontoauszug
  PROCEDURE TYPE ListeKontoauszug (Term: IN TermPointer) IS
    Merke : EintragPtrTyp;
  BEGIN
    IF LetzterAuszug = NULL THEN
      Term.out_str("Es wurden noch keine Buchungen vorgenommen!");
    ELSIF Beginn = Ende THEN
      Term.out_int(Beginn.Datum.Tag);
      ...
      Term.out_real(Beginn.Betrag);
```

```
      Term.out_str("Alter Kontostand:"): Term.out_real(0.0);
      Term.out_str("Neuer Kontostand:"): Term.out_real(Beginn.Kontostand);
    ELSIF LetzterAuszug = Ende THEN
      Term.out_str("Es wurden keine Buchungen vorgenommen!");
    ELSE
      Merke := LetzterAuszug;
      WHILE LetzterAuszug /= NULL LOOP
        Term.out_int(LetzterAuszug.Datum.Tag);
        ...;
        Term.out_real(LetzterAuszug.Betrag);
        LetzterAuszug := LetzterAuszug.Next;
      END LOOP;
      Term.out_str("Alter Kontostand:"): Term.out_real(Merke.Kontostand);
      Term.out_str("Neuer Kontostand:"): Term.out_real(Ende.Kontostand);
      LetzterAuszug := Ende;
    END IF;
  END ListeKontoauszug;

BEGIN
END KontobewegungsListenTyp;

-------------------------------------------------------------------------------------------
-- Zugriffskontrollierter Depot-Generator fuer die Bankkonten                            --
-------------------------------------------------------------------------------------------
-- Spezifikations-Part
PROTECTED DEPOT TYPE SPEC KontoTyp (KontoNummer  : IN integer;
                                    InhaberDaten : IN KundenDatenPtrTyp;
                                    InhaberUid   : IN UserUidType;
                                    BetreuerUid  : IN UserUidType)
  IMPORT ErrorType, UserUidType, KundenDatenPtrTyp, AllgKundenDatenTyp, PersKundenDatenTyp,
         TermPointer, KontobewegungsListenTyp, DatumTyp;
IS
  FUNCTION  TYPE SPEC LeseKontonummer RETURN integer;
  FUNCTION  TYPE SPEC LeseKontostand  RETURN real;
  PROCEDURE TYPE SPEC Einzahlen (Zweck: IN string; Betrag: IN real);
  PROCEDURE TYPE SPEC Abheben (Zweck: IN string; Betrag: IN real; Flag: IN boolean;
                               Error: OUT ErrorType);
  FUNCTION  TYPE SPEC LeseAllgKundenDaten RETURN AllgKundenDatenTyp;
  FUNCTION  TYPE SPEC LesePersKundenDaten RETURN PersKundenDatenTyp;
  PROCEDURE TYPE SPEC AendereKundenDaten (NeueKundenDaten : IN KundenDatenPtrTyp);
  PROCEDURE TYPE SPEC ZeigeKontoauszug (Term: IN TermPointer);
  PROCEDURE TYPE SPEC ZeigeKontobewegungen (Von, Bis: IN DatumTyp; Term: IN TermPointer);
  PROCEDURE TYPE SPEC SetzeMaxKredit (NeuerMaxKredit : IN real);
  FUNCTION  TYPE SPEC LeseMaxKredit RETURN real;
  FUNCTION  TYPE SPEC Aufloesen RETURN ErrorType;

  AnzahlBarAbhebungen : integer := 0;

  SYNC READER_WRITER WITH
    READER : LeseKontonummer, LeseKontostand, LeseAllgKundenDaten, LesePersKundenDaten,
             ZeigeKontoauszug, ZeigeKontobewegungen, LeseMaxKredit;
    WRITER : Einzahlen, Abheben, AendereKundenDaten, SetzeMaxKredit, Aufloesen;

  VIEWS
    KreditBerechnung: LeseKontostand, ZeigeKontobewegungen, LeseAllgKundenDaten,
                      LesePersKundenDaten, LeseMaxKredit;
    LeseKontonummer, LeseKontostand, Einzahlen, Abheben, LeseAllgKundenDaten,
    AendereKundenDaten, ZeigeKontoauszug, SetzeMaxKredit, Aufloesen;
```

```
ACL
  LeseKontostand      : (InhaberUid, Kunde, +), (BetreuerUid, Kundenbetreuer, +);
  Abheben             : (InhaberUid, Kunde, +), (BetreuerUid, Kundenbetreuer, +),
                        (-1, Kassierer, +);
  LeseAllgKundenDaten : (InhaberUid, Kunde, +), (-1, Kundenbetreuer, +),
                        (-1, Kassierer, +), (-1, Bankleiter, +);
  AendereKundenDaten  : (BetreuerUid, Kundenbetreuer, +);
  ZeigeKontoauszug    : (InhaberUid, Kunde, +);
  KreditBerechnung    : (BetreuerUid, Kundenbetreuer, +);
  Aufloesen           : (BetreuerUid, Kundenbetreuer, +);
  ChangeACL           : (InhaberUid, Kunde, +);


ACCESS RESTRICTIONS
  LeseKontonummer     : Caller.ActorName = KontoVerwalter;
  LeseKontostand      : (IN_ACL(Caller.User, Caller.Role, THIS, THIS) AND
                         (Caller.Role = Kunde OR (Caller.Role = Kundenbetreuer AND
                          8 <= Zeit.Stunde AND Zeit.Stunde < 17 AND
                          1 <= Zeit.Wochentag AND Zeit.Wochentag < 6))) OR
                        Caller.Role = Bankleiter OR
                        Caller.ConGen = Zinsberechnung.Zinsberechner.BerechneZinsen;
  Einzahlen           : (Caller.Role = Kassierer AND
                         8 <= Zeit.Stunde AND Zeit.Stunde < 17 AND
                         1 <= Zeit.Wochentag AND Zeit.Wochentag < 6) OR
                        Caller.ConGen = Ueberweiser OR
                        Caller.ActorGen = Zinsberechnung.Zinsberechner;
  Abheben             : (IN_ACL(Caller.User, Caller.Role, THIS, THIS) AND
                         ((Caller.Role = Kunde AND AnzahlBarAbhebungen < 3 AND
                           Betrag <= 1000) OR
                          (Caller.Role = Kunde AND Caller.ConGen = Ueberweiser) OR
                          (Caller.Role = Kundenbetreuer AND Caller.ConGen = Ueberweiser
                           AND 8 <= Zeit.Stunde AND Zeit.Stunde < 17 AND
                           1 <= Zeit.Wochentag AND Zeit.Wochentag < 6) OR
                          (Caller.Role = Kassierer AND
                           8 <= Zeit.Stunde AND Zeit.Stunde < 17 AND
                           1 <= Zeit.Wochentag AND Zeit.Wochentag < 6))) OR
                        Caller.ActorGen = Zinsberechnung.Zinsberechner;
  LeseAllgKundenDaten : IN_ACL(Caller.User, Caller.Role, THIS, THIS);
  AendereKundenDaten  : IN_ACL(Caller.User, Caller.Role, THIS, THIS) AND
                        Caller.Role = Kundenbetreuer AND
                        8 <= Zeit.Stunde AND Zeit.Stunde < 17 AND
                        1 <= Zeit.Wochentag AND Zeit.Wochentag < 6;
  ZeigeKontoauszug    : IN_ACL(Caller.User, Caller.Role, THIS, THIS) AND
                        Caller.Role = Kunde;
  KreditBerechnung    : Caller.ConGen = BerechneKredit AND
                        ((Caller.Role = Kundenbetreuer AND
                          IN_ACL(Caller.User, Caller.Role, THIS, THIS) AND
                          8 <= Zeit.Stunde AND Zeit.Stunde < 17 AND
                          1 <= Zeit.Wochentag AND Zeit.Wochentag < 6) OR
                         Caller.Role = Bankleiter);
  SetzeMaxKredit      : Caller.Role = Bankleiter;
  Aufloesen           : Caller.ConGen = KontoVerwalter.KontoAufloesen AND
                        Caller.Role = Kundenbetreuer AND
                        IN_ACL(Caller.User, Caller.Role, THIS, THIS) AND
                        8 <= Zeit.Stunde AND Zeit.Stunde < 17 AND
                        1 <= Zeit.Wochentag AND Zeit.Wochentag < 6;
  Create              : Caller.ConGen = KontoVerwalter.KontoEroeffnen AND
                        Caller.Role = Kundenbetreuer AND
                        8 <= Zeit.Stunde AND Zeit.Stunde < 17 AND
                        1 <= Zeit.Wochentag AND Zeit.Wochentag < 6;
```

```
    ChangeACL           : (IN_ACL(Caller.User, Caller.Role, THIS, THIS) AND
                           Caller.User = InhaberUid AND Caller.Role = Kunde AND
                           (ViewName = 'LeseKontostand' OR ViewName = 'Abheben' OR
                            ViewName = 'ZeigeKontoauszug')) OR
                          Caller.Role = Bankleiter;
END KontoTyp;

-- Implementierungs-Part des Depot-Generators KontoTyp
PROTECTED DEPOT TYPE KontoTyp (KontoNummer  : IN integer;
                               InhaberDaten : IN KundenDatenPtrTyp;
                               InhaberUid   : IN UserUidType;
                               BetreuerUid  : IN UserUidType)
  IMPORT ErrorType, UserUidType, KundenDatenPtrTyp, AllgKundenDatenTyp, PersKundenDatenTyp,
         TermPointer, KontobewegungsListenTyp, DatumTyp;
IS
  Kontostand : real;
  MaxKredit  : real;

  -- Liste der Kontobewegungen des Kontos
  DEPOT Kontobewegungsliste : KontobewegungsListenTyp;

  -- Zugriffsoperation LeseKontonummer
  FUNCTION TYPE LeseKontonummer RETURN integer
    IMPORT KontoNummer;
  IS
  BEGIN
    RETURN KontoNummer;
  END LeseKontonummer;

  -- Zugriffsoperation LeseKontostand
  FUNCTION TYPE LeseKontostand RETURN real
    IMPORT Kontostand;
  IS
  BEGIN
    RETURN Kontostand;
  END LeseKontostand;

  -- Zugriffsoperation Einzahlen
  PROCEDURE TYPE SPEC Einzahlen (Zweck: IN string; Betrag: IN real)
    IMPORT Kontostand;
  IS
  BEGIN
    IF Betrag > 0.0 THEN
      Kontostand := Kontostand + Betrag;
      Kontobewegungsliste.FuegeEin(Zweck, Betrag, Kontostand);
    END IF;
  END Einzahlen;

  -- Zugriffsoperation Abheben
  PROCEDURE TYPE Abheben (Zweck: IN string; Betrag: IN real; Flag: IN boolean;
                          Error: OUT ErrorType)
    IMPORT Kontostand, MaxKredit, ErrorType;
  IS
    BuchungsBetrag : real;
  BEGIN
    IF Betrag > 0.0 AND Kontostand + MaxKredit >= Betrag THEN
      Kontostand := Kontostand - Betrag;
      BuchungsBetrag := -1.0 * Betrag;
      Kontobewegungsliste.FuegeEin(Zweck, BuchungsBetrag, Kontostand);
      Error := 0;
```

```
    IF Flag THEN  -- Barabhebung
      AnzahlBarAbhebungen := AnzahlBarAbhebungen + 1;
    END IF;
  ELSE
    Error := 1;  -- Betrag nicht gedeckt
  END IF;
END Abheben;

-- Zugriffsoperation LeseAllgKundenDaten
FUNCTION TYPE LeseAllgKundenDaten RETURN AllgKundenDatenTyp
  IMPORT InhaberDaten, AllgKundenDatenTyp;
IS
  AllgKundenDaten :  AllgKundenDatenTyp;
BEGIN
  AllgKundenDaten.Vorname := InhaberDaten.AllgKundenDaten.Vorname;
  AllgKundenDaten.Nachname := InhaberDaten.AllgKundenDaten.Nachname;
  RETURN AllgKundenDaten;
END LeseAllgKundenDaten;

-- Zugriffsoperation LesePersKundenDaten
FUNCTION TYPE LesePersKundenDaten RETURN PersKundenDatenTyp
  IMPORT InhaberDaten, PersKundenDatenTyp;
IS
  PersKundenDaten :  PersKundenDatenTyp;
BEGIN
  PersKundenDaten.GebDatum := InhaberDaten.PersKundenDaten.GebDatum;
  ...
  RETURN PersKundenDaten;
END LesePersKundenDaten;

-- Zugriffsoperation AendereKundenDaten
PROCEDURE TYPE AendereKundenDaten (NeueKundenDaten : IN KundenDatenPtrTyp)
  IMPORT InhaberDaten, KundenDatenPtrTyp;
IS
BEGIN
  IF NeueKundenDaten.AllgKundenDaten.Nachname /= "" THEN
    InhaberDaten.AllgKundenDaten.Nachname := NeueKundenDaten.AllgKundenDaten.Nachname;
  END IF;
  IF NeueKundenDaten.PersKundenDaten.Adresse /= "" THEN
    InhaberDaten.PersKundenDaten.Adresse := NeueKundenDaten.PersKundenDaten.Adresse;
  END IF;
  ...
END AendereKundenDaten;

-- Zugriffsoperation ZeigeKontoauszug
PROCEDURE TYPE ZeigeKontoauszug (Term: IN TermPointer)
  IMPORT TermPointer, Kontobewegungsliste;
IS
BEGIN
  Kontobewegungsliste.ListeKontoauszug(Term);
END ZeigeKontoauszug;

-- Zugriffsoperation ZeigeKontobewegungen
PROCEDURE TYPE ZeigeKontobewegungen (Von, Bis: IN DatumTyp; Term: IN TermPointer)
  IMPORT DatumTyp, TermPointer, Kontobewegungsliste;
IS
BEGIN
  Kontobewegungsliste.ListeKontobewegungen(Von, Bis, Term);
END ZeigeKontobewegungen;
```

```
  -- Zugriffsoperation SetzeMaxKredit
  PROCEDURE TYPE SetzeMaxKredit (NeuerMaxKredit : IN real)
    IMPORT MaxKredit;
  IS
  BEGIN
    IF NeuerMaxKredit > 0.0 THEN
      MaxKredit := NeuerMaxKredit;
    END IF;
  END SetzeMaxKredit;

  -- Zugriffsoperation LeseMaxKredit
  FUNCTION  TYPE LeseMaxKredit RETURN real
    IMPORT MaxKredit;
  IS
  BEGIN
    RETURN MaxKredit;
  END LeseMaxKredit;

  -- Zugriffsoperation Aufloesen
  FUNCTION TYPE Aufloesen RETURN ErrorType
    IMPORT Kontostand, Kontobewegungsliste;
  IS
  BEGIN
    IF Kontostand /= 0.0 THEN
      RETURN 1;    -- Kontostand ungleich 0
    ELSE
      RETURN 0;    -- Konto kann aufgeloest werden
    END IF;
  END Aufloesen;

BEGIN
  Kontostand := 0.0;
  MaxKredit  := 0.0;
END Kontotyp;

-- Zeiger-Generator fuer Zeiger auf Konten
POINTER TYPE KontoPtrTyp IS ACCESS KontoTyp;

-------------------------------------------------------------------------------------------
-- Zugriffskontrollierter K-Akteur-Generator fuer die Kontenverwaltung                   --
-------------------------------------------------------------------------------------------
-- Spezifikations-Part
PROTECTED TASK TYPE SPEC KontoVerwalterTyp
  IMPORT KontoTyp, KontoPtrTyp, KundenDatenPtrTyp, ErrorType, UserUidType, UserManagement,
         ZeitTyp, Zeit, MonatTyp;
IS
  ENTRY TYPE SPEC KontoEroeffnen (KontoInhaber: IN KundenDatenPtrTyp;
                                  KontoNummer : OUT integer);
  ENTRY TYPE SPEC KontoAufloesen (KontoZeiger : IN KontoPtrTyp; Error: OUT ErrorType);
  ENTRY TYPE SPEC GibKontoZeiger (KontoNummer : IN integer; KontoZeiger: OUT KontoPtrTyp;
                                  Error : OUT ErrorType);

  ACCESS RESTRICTIONS
    KontoEroeffnen : (Caller.Role = Kundenbetreuer AND 8 <= Zeit.Stunde AND Zeit.Stunde < 17
                      AND 1 <= Zeit.Wochentag AND Zeit.Wochentag < 6) OR
                     (Caller.User = System AND
                      KontoInhaber.AllgKundenDaten.Nachname = "Zentralkonto");
    KontoAufloesen : Caller.Role = Kundenbetreuer AND
                     IN_ACL(Caller.User, Caller.Role, KontoZeiger, Aufloesen) AND
                     8 <= Zeit.Stunde AND Zeit.Stunde < 17 AND
```

```
                    1 <= Zeit.Wochentag AND Zeit.Wochentag < 6;
    GibKontoZeiger : NOT (Caller.Role = SysAdmin);
    Create         : Caller.User = System;
END KontoVerwalterTyp;

-- Implementierungs-Part des K-Akteur-Generators KontoVerwalterTyp
PROTECTED TASK TYPE KontoVerwalterTyp
  IMPORT KontoTyp, KontoPtrTyp, KundenDatenPtrTyp, ErrorType, UserUidType, UserManagement,
         ZeitTyp, Zeit, MonatTyp;
IS
  POINTER TYPE KontoListenEintragPtrTyp;
  TYPE KontoListenEintragTyp IS RECORD
    Konto : KontoPtrTyp;
    Next  : KontoListenEintragPtrTyp;
    Prev  : KontoListenEintragPtrTyp;
  END RECORD;
  POINTER TYPE KontoListenEintragPtrTyp IS ACCESS KontoListenEintragTyp;

  KontoListenBeginn, KontoListenEnde : KontoListenEintragPtrTyp := NULL;
  AktuelleKontonummer : integer := 100000;
  ErlaubterZugriff : boolean;

  -- Lokale Funktion zum Suchen eines Kontos in der Kontoliste
  FUNCTION TYPE SucheKonto (Kontonummer: IN integer) RETURN KontoPtrTyp
    IMPORT KontoPtrTyp, KontoListenEintragPtrTyp, KontoListenBeginn, ErlaubterZugriff;
  IS
    AktuellerEintrag : KontoListenEintragPtrTyp := KontoListenBeginn;
  BEGIN
    IF AktuellerEintrag = NULL THEN
      RETURN NULL;
    ELSE
      WHILE (AktuellerEintrag.Konto.LeseKontonummer STATUS ErlaubterZugriff /= Kontonummer)
            AND (AktuellerEintrag /= KontoListenEnde) LOOP
        AktuellerEintrag := AktuellerEintrag.Next;
      END LOOP;
      IF ErlaubterZugriff THEN
        IF AktuellerEintrag.Konto.LeseKontonummer STATUS ErlaubterZugriff = Kontonummer THEN
          RETURN AktuellerEintrag.Konto;
        ELSE
          RETURN NULL;
        END IF;
      ELSE
        -- dieser Fall kann gemaess der Rechtsfestlegungen fuer die Zugriffsoperation
        -- "LeseKontonummer" eines Kontos nicht eintreten, da "SucheKonto" nur von dem
        -- benannten K-Akteur "KontoVerwalter"(siehe unten) ausgefuehrt wird
        RETURN NULL;
      END IF;
    END IF;
  END SucheKonto;

  -- Kommunikationsoperation KontoEroeffnen
  ENTRY TYPE KontoEroeffnen (KontoInhaber: IN KundenDatenPtrTyp; KontoNummer : OUT integer;
                             Error : OUT ErrorType)
    IMPORT KontoTyp, KontoPtrTyp, KontoListenEintragPtrTyp, KontoListenEintragTyp,
           KontoListenBeginn, KontoListenEnde, ErlaubterZugriff, ErrorType,
           AktuelleKontonummer, UserManagement, KundenDatenPtrTyp;
  IS
    NeuerEintrag    : KontoListenEintragPtrTyp;
    KontoInhaberUid : UserUidType;
    ActError        : ErrorType;
```

```
BEGIN
  UserManagement.UserDataManager.GetUid(KontoInhaber.AllgKundenDaten.Nachname,
                                         KontoInhaberUid, ActError);
  IF ActError = 1 THEN
    Error := 1;    -- es ist noch keine Kennung fuer den Kunden eingerichtet
  ELSE
    NeuerEintrag := NEW(KontoListenEintragPtrTyp, KontoListenEintragTyp);
    NeuerEintrag.Konto := NEW(KontoPtrTyp, KontoTyp) (AktuelleKontonummer,
       KontoInhaber, KontoInhaberUid, Caller.User) STATUS ErlaubterZugriff;
    IF NOT ErlaubterZugriff THEN
      Error := 2;  -- Zugriffsverletzung bei Ausfuehrung von KontoEroeffnen
    ELSE
      NeuerEintrag.Next := NULL:
      IF KontoListenBeginn = NULL THEN
        KontoListenBeginn := NeuerEintrag;
        KontoListenBeginn.Prev := NULL;
      ELSE
        NeuerEintrag.Prev := KontoListenEnde;
        KontoListenEnde.Next := NeuerEintrag;
      END IF;
      KontoListenEnde := NeuerEintrag;
      KontoNummer := AktuelleKontonummer;
      AktuelleKontonummer := AktuelleKontonummer + 1;
      Error := 0;  -- Konto wurde erzeugt und in Liste eingefuegt
    END IF:
  END IF;
END KontoEroeffnen;

-- Kommunikationsoperation KontoAufloesen
ENTRY TYPE KontoAufloesen (KontoZeiger: IN KontoPtrTyp; Error: OUT ErrorType)
  IMPORT KontoPtrTyp, ErrorType, ErlaubterZugriff, KontoListenBeginn, KontoListenEnde;
IS
BEGIN
  IF KontoZeiger = NULL THEN
    Error := 1;    -- KontoZeiger identifiziert kein Konto
  ELSE
    CASE KontoZeiger.Aufloesen STATUS ErlaubterZugriff IS
      WHEN 0 DO
        IF KontoZeiger = KontoListenBeginn THEN
          KontoListenBeginn := KontoZeiger.Next;
          KontoListenBeginn.Prev := NULL;
        ELSIF KontoZeiger = KontoListenEnde THEN
          KontoListenEnde := KontoZeiger.Prev;
          KontoListenEnde.Next := NULL;
        ELSE
          KontoZeiger.Prev.Next := KontoZeiger.Next;
          KontoZeiger.Next.Prev := KontoZeiger.Prev;
        END IF;
        Error := 0;  -- Konto wurde aufgeloest
      WHEN 1 DO
        Error := 2:  -- Kontostand des aufzuloesenden Kontos ist ungleich 0
      WHEN OTHERS DO
        Error := 3:  -- Interner Fehler
    END CASE;
    IF NOT ErlaubterZugriff THEN
      Error := 4;    -- Zugriffsverletzung bei Ausfuehrung von KontoAufloesen
    END IF;
  END IF;
END KontoAufloesen;
```

```
-- Kommunikationsoperation GibKontoZeiger
ENTRY TYPE GibKontoZeiger (KontoNummer : IN integer; KontoZeiger: OUT KontoPtrTyp;
                           Error : OUT ErrorType)
  IMPORT KontoPtrTyp, ErrorType, SucheKonto;
IS
BEGIN
  KontoZeiger := SucheKonto(KontoNummer);
  IF KontoZeiger /= NULL THEN
    Error := 0;    -- Konto mit dieser Kontonummer vorhanden
  ELSE
    Error := 1;    -- Kein Konto mit dieser Kontonummer vorhanden
  END IF;
END GibKontoZeiger;

------------------------------------------------------------------------------------------
-- Zugriffskontrollierter M-Akteur-Generator fuer die automatische Zinsberechnung      --
-- (auszugsweise)                                                                      --
------------------------------------------------------------------------------------------
PROTECTED PROCESS TYPE SPEC Zinsberechnung
  IMPORT ZeitTyp, Zeit, KontoListenEintragPtrTyp, KontoListenBeginn, MonatTyp,
         KontoPtrTyp, ErrorType;
IS
  ACCESS RESTRICTIONS
    Create : Caller.ConName = KontoVerwalter;
END Zinsberechnung;

PROTECTED PROCESS TYPE Zinsberechnung
  IMPORT ZeitTyp, Zeit, KontoListenEintragPtrTyp, KontoListenBeginn, MonatTyp,
         KontoPtrTyp, ErrorType;
IS
  AlarmierungsZeit : ZeitTyp;
  AktuellerEintrag : KontoListenEintragPtrTyp := KontoListenBeginn;

  ----------------------------------------------------------------------------------------
  -- M-Akteur zur Berechnung und Ueberweisung der monatlichen Zinsen fuer ein Konto    --
  ----------------------------------------------------------------------------------------
  PROCESS TYPE Zinsberechner (KontoZeiger: IN KontoPtrTyp, Monat: IN MonatTyp)
    IMPORT KontoPtrTyp, MonatTyp, KontoListenBeginn, ErrorType;
  IS
    Zweck            : string;
    Zins, SollZinsen : real;
    ErlaubterZugriff : boolean;
    Zentralkonto     : KontoPtrTyp := KontoListenBeginn.Konto;
    ActError         : ErrorType;

    -- Lokale Funktion zur Berechnung der monatlichen Zinsen
    FUNCTION TYPE BerechneZinsen (KontoZeiger: IN KontoPtrTyp, ...) RETURN real
      IMPORT KontoPtrTyp, ...;
    IS
      AktuellerKontostand : real;
    BEGIN
      ...
      AktuellerKontostand := KontoZeiger.LeseKontostand STATUS ErlaubterZugriff;
      ...
    END BerechneZinsen;

  BEGIN
    -- Berechnung der Zinsen fuer den jeweiligen Monat
    Zins := BerechneZinsen(KontoZeiger,...);
    IF Zins /= 0.0 THEN
```

```
    CASE Monat IS
      WHEN 1 DO
        Zweck := "Zinsen fuer Monat Januar";
      WHEN 2 DO
        ...
      WHEN OTHERS DO
        OUTPUT "Interner Fehler!";
    END CASE;
    IF Zins < 0 THEN
      SollZinsen := -1.0 * Zins;
      KontoZeiger.Abheben(Zweck, SollZinsen, FALSE, ActError) STATUS ErlaubterZugriff;
      IF NOT ErlaubterZugriff THEN
        -- dieser Fall kann gemaess der Rechtsfestlegungen fuer die Zugriffsoperation
        -- "Abheben" eines Kontos nicht eintreten!
        OUTPUT "Unerwartete Zugriffsverletzung!";
      ELSIF ActError = 1 THEN
        -- Konto sperren, da der abzubuchende Zinsbetrag nicht gedeckt ist
        ...
      ELSE
        Zentralkonto.Einzahlen(Zweck, SollZinsen) STATUS ErlaubterZugriff;
        ...
      END IF;
    ELSE
      Zentralkonto.Abheben(Zweck, Zins, FALSE, ActError) STATUS ErlaubterZugriff;
      IF NOT ErlaubterZugriff THEN
        -- dieser Fall kann gemaess der Rechtsfestlegungen fuer die Zugriffsoperation
        -- "Abheben" eines Kontos nicht eintreten!
        OUTPUT "Unerwartete Zugriffsverletzung!";
      ELSIF ActError = 1 THEN
        -- das Bankkonto ist nicht gedeckt
        ...
      ELSE
        KontoZeiger.Einzahlen(Zweck, Zins) STATUS ErlaubterZugriff;
          ...
      END IF;
    END IF; END IF;
  END Zinsberechner;

-- Anweisungsteil von Zinsberechnung
BEGIN
  -- Alarmierungszeit auf Ende des aktuellen Monats setzen
  ...
  LOOP
    -- Timer zur Alarmierung am Ende des aktuellen Monats setzen
    ...
    -- Am Monatsende Zinsen fuer alle Konten berechnen
    WHILE AktuellerEintrag /= NULL LOOP
      FORK Zinsberechner(AktuellerEintrag.Konto, AlarmierungsZeit.Datum.Monat);
    END LOOP;
    -- Alarmierungszeit auf Ende des Folgemonats setzen
    AlarmierungsZeit.Datum.Monat := Zeit.Datum.Monat + 1;
    AlarmierungsZeit.Datum.Jahr  := Zeit.Datum.Jahr;
    IF AlarmierungsZeit.Monat = 13 THEN
      AlarmierungsZeit.Datum.Monat := 1;
      AlarmierungsZeit.Datum.Jahr  := Zeit.Datum.Jahr + 1;
    END IF;
  END LOOP;
END Zinsberechnung;
```

```
-- Anweisungsteil von KontoVerwalterTyp
BEGIN
  -- Erzeugen des M-Akteurs fuer die automatische Zinsberechnung am Monatsende
  FORK Zinsberechnung STATUS ErlaubterZugriff;
  IF NOT ErlaubterZugriff THEN
    -- dieser Fall kann gemaess der Rechtsfestlegungen fuer die Erzeugung einer Inkarnation
    -- bzgl. des Generators "Zinsberechnung" nicht eintreten!
    OUTPUT "Unerwartete Zugriffsverletzung!";
  ELSE
    LOOP
      SELECT
        ACCEPT KontoEroeffnen;
      OR
        ACCEPT KontoAufloesen;
      OR
        ACCEPT GibKontoZeiger;
      OR
        TERMINATE;
      END SELECT;
    END LOOP;
  END IF;
END KontoVerwalterTyp;

-- Inkarnierung des Kontoverwalters des Systems
TASK KontoVerwalter : KontoVerwalterTyp STATUS ErlaubterZugriff;

-------------------------------------------------------------------------------------------
-- Zugriffskontrollierter M-Akteur-Generator fuer die Durchfuehrung von Ueberweisungen   --
-------------------------------------------------------------------------------------------

-- Typ fuer Ueberweisungstraeger
TYPE UeberweisungsTraegerTyp IS RECORD
  VonKontoZeiger : KontoPtrTyp;
  AufKontoZeiger : KontoPtrTyp;
  Betrag         : real;
  Zweck          : string;
END RECORD;

-- Spezifikations-Part des M-Akteur-Generators
PROTECTED PROCESS TYPE SPEC Ueberweiser (Ueberweisung: IN UeberweisungsTraegerTyp;
                                         Error: OUT ErrorType)
  IMPORT UeberweisungsTraegerTyp, ErrorType;
IS
  ACCESS RESTRICTIONS
    Create : IN_ACL(Caller.User, Caller.Role, Ueberweisung.VonKontoZeiger, Abheben) AND
             ((Caller.Role = Kunde) OR (Caller.Role = Kundenbetreuer AND
              8 <= Zeit.Stunde AND Zeit.Stunde < 17 AND
              1 <= Zeit.Wochentag AND Zeit.Wochentag < 6));
END Ueberweiser;

-- Implementierungs-Part des M-Akteur-Generators
PROTECTED PROCESS TYPE Ueberweiser (Ueberweisung: IN UeberweisungsTraegerTyp;
                                    Error: OUT ErrorType)
  IMPORT UeberweisungsTraegerTyp, ErrorType;
IS
  ErlaubterZugriff : boolean;
  ActError         : ErrorType;
BEGIN
  Ueberweisung.VonKontoZeiger.Abheben(Ueberweisung.Zweck, Ueberweisung.Betrag, FALSE,
                                      ActError) STATUS ErlaubterZugriff;
```

```
  IF NOT ErlaubterZugriff THEN
    Error := 1;      -- Zugriffsverletzung bei Durchfuehrung der Ueberweisung
  ELSE
    IF ActError = 1 THEN
      Error := 2;    -- zu ueberweisender Betrag nicht gedeckt
    ELSE
      Ueberweisung.AufKontoZeiger.Einzahlen(Ueberweisung.Zweck, Ueberweisung.Betrag)
                                      STATUS ErlaubterZugriff;
      IF NOT ErlaubterZugriff THEN
        -- dieser Fall kann gemaess der Rechtsfestlegungen fuer die Zugriffsoperation
        -- "Einzahlen" eines Kontos nicht eintreten!
        Error := 3;  -- Interner Fehler
      ELSE
        Error := 0;  -- Ueberweisung wurde erfolgreich durchgefuehrt
      END IF;
    END IF;
  END IF;
END Ueberweiser;

-------------------------------------------------------------------------------------------
-- Zugriffskontrollierter PS-Order-Generator fuer die Berechnung des maximalen           --
-- Ueberziehungskredits eines Kontos (auszugsweise)                                      --
-------------------------------------------------------------------------------------------
PROTECTED PROCEDURE TYPE SPEC BerechneKredit (KontoZeiger: IN KontoPtrTyp;
                                              Term       : IN TermPointer)
  IMPORT KontoPtrTyp, TermPointer, DatumTyp, PersKundenDatenTyp;
IS
  ACCESS RESTRICTIONS
    Create : (Caller.Role = Kundenbetreuer AND
              IN_ACL(Caller.User, Caller.Role, KontoZeiger, KreditBerechnung) AND
              8 <= Zeit.Stunde AND Zeit.Stunde < 17 AND
              1 <= Zeit.Wochentag AND Zeit.Wochentag < 6) OR
             Caller.Role = Bankleiter;
END BerechneKredit;

PROTECTED PROCEDURE TYPE BerechneKredit (KontoZeiger: IN KontoPtrTyp;
                                         Term       : IN TermPointer)
  IMPORT KontoPtrTyp, TermPointer, DatumTyp, PersKundenDatenTyp;
IS
  Von, Bis         : DatumTyp;
  PersKundenDaten  : PersKundenDatenTyp;
  Kontostand       : real;
  AlterMaxKredit   : real;
  NeuerMaxKredit   : real;
  ErlaubterZugriff : boolean;
  ...
BEGIN
  ...
  KontoZeiger.ZeigeKontobewegungen(Von, Bis, Term) STATUS ErlaubterZugriff;
  ...
  PersKundenDaten := KontoZeiger.LesePersKundenDaten STATUS ErlaubterZugriff;
  ...
  Kontostand := KontoZeiger.LeseKontostand STATUS ErlaubterZugriff;
  ...
  AlterMaxKredit := KontoZeiger.LeseMaxKredit STATUS ErlaubterZugriff;
  NeuerMaxKredit := ...;
  Term.out_str("Es wurde folgender Ueberziehungskredit ermittelt:");
  Term.out_real(NeuerMaxKredit);
END BerechneKredit;
```

```
  ------------------------------------------------------------------------------------------
  -- Hilfsfunktion zum Ermitteln eines Kontos anhand der Kontonummer                      --
  ------------------------------------------------------------------------------------------
  FUNCTION TYPE ErmittleKonto (Kontonummer: IN integer; Term: IN TermPointer)
                                                                       RETURN KontoPtrTyp
      IMPORT TermPointer, KontoPtrTyp, KontoVerwalter, ErrorType;
    IS
      Kontozeiger      : KontoPtrTyp := NULL;
      ActError         : ErrorType;
      ErlaubterZugriff : boolean;
    BEGIN
      KontoVerwalter.GibKontoZeiger(Kontonummer, Kontozeiger, ActError)
                                                                STATUS ErlaubterZugriff;
      IF NOT ErlaubterZugriff THEN
        Term.out_str("Sie haben keine Rechte zum Durchsuchen der Kontenliste!\n");
        RETURN NULL;
      ELSIF ActError = 1 THEN
        Term.out_str("Sie haben eine ungueltige Kontonummer eingegeben!\n");
        RETURN NULL;
      ELSE
        RETURN Kontozeiger;
      END IF;
    END ErmittleKonto;

--------------------------------------------------------------------------------------------
-- Generatoren fuer die rollenspezifischen Benutzerrepraesentanten                        --
--------------------------------------------------------------------------------------------

  ------------------------------------------------------------------------------------------
  -- Benutzerrepraesentant fuer Benutzer in der Rolle Kunde                               --
  ------------------------------------------------------------------------------------------
  PROTECTED PROCESS TYPE SPEC Kunde (Term: IN TermPointer)
    IMPORT TermPointer, KontoPtrTyp, UeberweisungsTraegerTyp, ErrorType, UserUidType,
           UserManagement, ErmittleKonto;
  IS
    ACCESS RESTRICTIONS
      Create : Caller.ActorGen = UserManagement.LoginAtTerminal AND
               Caller.ConGen   = CreateUserRepForRole;
  END Kunde;

  PROTECTED PROCESS TYPE Kunde (Term: IN TermPointer)
    IMPORT ...;
  IS
    Menueauswahl, Auswahl : integer;
    Kontonummer        : integer;
    EmpfaengerKontoNr  : integer;
    Kontozeiger        : KontoPtrTyp := NULL;
    Kontostand, Betrag : real;
    Ueberweisung       : UeberweisungsTraegerTyp;
    Ueberweisungsfehler: ErrorType;
    ActError           : ErrorType;
    ViewName           : string;
    Kennung            : string;
    BenutzerId         : UserUidType;
    Flag               : boolean;
    KeinFehler         : boolean := TRUE;
    ErlaubterZugriff   : boolean;
```

```
BEGIN
  BLOCK ForExit IS BEGIN
  LOOP
    Term.out_str("-----------------------------------------------------------------\n");
    Term.out_str("Folgende Funktionen stehen Ihnen zur Verfuegung:\n");
    Term.out_str("Kontostand lesen     : 1\n");
    Term.out_str("Kontoauszug ausgeben : 2\n");
    Term.out_str("Geld abheben         : 3\n");
    Term.out_str("Ueberweisen          : 4\n");
    Term.out_str("Rechte aendern       : 5\n");
    Term.out_str("Beenden              : 6\n\n");
    Term.out_str("Waehlen Sie durch Eingabe der jeweiligen Ziffer eine Funktion aus!\n");
    Term.out_str("-----------------------------------------------------------------\n");
    Term.in_int(Menueauswahl);
    CASE Menueauswahl IS
      WHEN 1 DO   -- Kontostand lesen
        Term.out_str("Bitte geben Sie Ihre Kontonummer ein:");
        Term.in_int(Kontonummer);
        Kontozeiger := ErmittleKonto(Kontonummer, Term);
        IF Kontozeiger /= NULL THEN
          Kontostand := Kontozeiger.LeseKontostand STATUS ErlaubterZugriff;
          IF NOT ErlaubterZugriff THEN
            Term.out_str("Sie sind nicht berechtigt, den Kontostand dieses Kontos
                          zu lesen!"\n);
          ELSE
            Term.out_str("Aktueller Kontostand:"); Term.out_real(Kontostand);
          END IF;
        END IF;
      WHEN 2 DO   -- Kontoauszug ausgeben
        Term.out_str("Bitte geben Sie Ihre Kontonummer ein:");
        Term.in_int(Kontonummer);
        Kontozeiger := ErmittleKonto(Kontonummer, Term);
        IF Kontozeiger /= NULL THEN
          Kontozeiger.ZeigeKontoauszug(Term) STATUS ErlaubterZugriff;
          IF NOT ErlaubterZugriff THEN
            Term.out_str("Sie sind nicht berechtigt, sich den Kontoauszug dieses Kontos
                          ausgeben zu lassen!"\n);
          END IF;
        END IF;
      WHEN 3 DO   -- Geld abheben
        Term.out_str("Bitte geben Sie Ihre Kontonummer ein:");
        Term.in_int(Kontonummer);
        Kontozeiger := ErmittleKonto(Kontonummer, Term);
        IF Kontozeiger /= NULL THEN
          Term.out_str("Bitte geben Sie den abzuhebenden Betrag ein:");
          Term.in_real(Betrag);
          IF Betrag < 0.0 THEN
            Term.out_str("Der abzuhebende Betrag muss positiv sein!\n");
          ELSE
            Kontozeiger.Abheben("Barabhebung", Betrag, TRUE, ActError)
                                                          STATUS ErlaubterZugriff;
            IF NOT ErlaubterZugriff THEN
              Term.out_str("Sie sind nicht berechtigt, Geld von diesem Konto abzuheben!\n");
            ELSIF ActError = 1 THEN
              Term.out_str("Der abzuhebende Betrag ist nicht gedeckt!\n");
            ELSE
              Term.out_str("Auszahlung erfolgt!\n");
            END IF;
          END IF;
        END IF;
```

```
    WHEN 4 DO   -- Ueberweisen
      Term.out_str("Bitte geben Sie die fuer die Ueberweisung benoetigten Daten ein:\n");
      Term.out_str("Ihre Kontonummer:"); Term.in_int(Kontonummer);
      Ueberweisung.VonKontoZeiger := ErmittleKonto(Kontonummer, Term);
      IF Ueberweisung.VonKontoZeiger /= NULL THEN
        Term.out_str("Kontonummer des Empfaengers:"); Term.in_int(EmpfaengerKontoNr);
        Ueberweisung.AufKontoZeiger := ErmittleKonto(EmpfaengerKontoNr, Term);
        IF Ueberweisung.AufKontoZeiger /= NULL THEN
          Term.out_str("Betrag:"); Term.in_real(Ueberweisung.Betrag);
          IF Ueberweisung.Betrag < 0.0 THEN
            Term.out_str("Abbruch: Der zu ueberweisende Betrag muss positiv sein!\n");
          ELSE
            Term.out_str("Zweck:"); Term.in_str(Ueberweisung.Zweck);
            FORK Ueberweiser(Ueberweisung, Ueberweisungsfehler) STATUS ErlaubterZugriff;
            IF NOT ErlaubterZugriff THEN
              Term.out_str("Sie sind nicht berechtigt, diese Ueberweisung
                            durchzufuehren!\n");
            END IF;
          END IF;
        END IF;
      END IF;
    WHEN 5 DO   -- Rechte aendern
      Term.out_str("Bitte geben Sie Ihre Kontonummer ein:");
      Term.in_int(Kontonummer);
      Kontozeiger := ErmittleKonto(Kontonummer, Term);
      IF Kontozeiger /= NULL THEN
        Term.out_str("Geben Sie durch Eingabe der Nummer die Operation an, fuer die Sie
                      das Recht an Ihrem Konto aendern moechten!\n");
        Term.out_str("Kontostand lesen     : 1\n");
        Term.out_str("Kontoauszug ausgeben : 2\n");
        Term.out_str("Geld abheben         : 3\n");
        Term.in_int(Auswahl);
        CASE Auswahl IS
          WHEN 1 DO
            ViewName := "LeseKontostand";
          WHEN 2 DO
            ViewName := "ZeigeKontoauszug";
          WHEN 3 DO
            ViewName := "Abheben";
          WHEN OTHERS DO
            KeinFehler := FALSE;
            Term.out_str("Falsche Eingabe!\n\n");
        END CASE;
        IF KeinFehler THEN
          Term.out_str("Geben Sie die Kennung des Kunden ein, fuer den Sie das Recht an
                        der eingegebenen Operation aendern moechten!\n");
          Term.in_str(Kennung);
          UserManagement.UserDataManager.GetUid(Kennung, BenutzerId, ActError);
          IF ActError = 1 THEN
            Term.out_str("Fehler: Einen Kunden mit dieser Kennung gibt es nicht!\n");
            KeinFehler := FALSE;
          ELSE
            Term.out_str("Moechten Sie das Recht an der Operation vergeben (1) oder
                          zuruecknehmen (2)? Bitte geben Sie die jeweilige Nummer an!\n");
            Term.in_int(Auswahl);
            IF Auswahl = 1 THEN
              Flag := TRUE;
            ELSIF Auswahl = 2 THEN
              Flag := FALSE;
```

```
              ELSE
                KeinFehler := FALSE;
                Term.out_str("Falsche Eingabe!\n\n");
              END IF;
              IF KeinFehler THEN
                Kontozeiger.ChangeACL(ViewName, BenutzerId, "Kunde", TRUE, Flag, ActError)
                                                                     STATUS ErlaubterZugriff;
                IF NOT ErlaubterZugriff THEN
                  Term.out_str("Sie sind nicht berechtigt, Rechteaenderungen an diesem
                                Konto vorzunehmen!\n");
                ELSE
                CASE ActError IS
                  WHEN 0 DO
                    Term.out_str("Rechteaenderung erfolgreich durchgefuehrt!"\n);
                  WHEN 2 DO
                    Term.out_str("Das Recht ist bereits vergeben!"\n);
                  WHEN 8 | 9 DO
                    Term.out_str("Das Recht ist nicht vergeben!"\n");
                  WHEN OTHERS DO
                    Term.out_str("Interner Fehler!"\n);
                END CASE;
                END IF;
              END IF;
            END IF;
          END IF;
        END IF;
      WHEN 6 DO
        EXIT ForExit;
      WHEN OTHERS DO
        Term.out_str("Falsche Eingabe!\n\n");
    END CASE;
  END LOOP;
  END ForExit;
END Kunde;

------------------------------------------------------------------------------------------
-- Benutzerrepraesentant fuer Benutzer in der Rolle Kundenbetreuer                      --
------------------------------------------------------------------------------------------
PROTECTED PROCESS TYPE SPEC Kundenbetreuer (Term: IN TermPointer)
  IMPORT TermPointer, KontoPtrTyp, KontoVerwalter, KundenDatenTyp, KundenDatenPtrTyp,
         ErmittleKonto, UeberweisungsTraegerTyp, BerechneKredit, ErrorType;
IS
  ACCESS RESTRICTIONS
    Create : Caller.ActorGen = UserManagement.LoginAtTerminal AND
             Caller.ConGen   = CreateUserRepForRole;
END Kundenbetreuer;

PROTECTED PROCESS TYPE Kundenbetreuer (Term: IN TermPointer)
  IMPORT ...;
IS
  Menueauswahl       : integer;
  KontoInhaber       : KundenDatenPtrTyp;
  Kontonummer        : integer;
  Kontostand         : real;
  Kontozeiger        : KontoPtrTyp := NULL;
  ActError           : ErrorType;
  ErlaubterZugriff   : boolean;
  EmpfaengerKontoNr  : integer;
  Ueberweisung       : UeberweisungsTraegerTyp;
  Ueberweisungsfehler: ErrorType;
```

```
BEGIN
  BLOCK ForExit IS BEGIN
  LOOP
    Term.out_str("-----------------------------------------------------------------\n");
    Term.out_str("Folgende Funktionen stehen Ihnen zur Verfuegung:\n");
    Term.out_str("Konto eroeffnen      : 1\n");
    Term.out_str("Konto aufloesen      : 2\n");
    Term.out_str("Kundendaten aendern  : 3\n");
    Term.out_str("Kontostand lesen     : 4\n");
    Term.out_str("Ueberweisen          : 5\n");
    Term.out_str("Ueberziehungskredit  : 6\n");
    Term.out_str("Beenden              : 7\n\n");
    Term.out_str("Waehlen Sie durch Eingabe der jeweiligen Ziffer eine Funktion aus!\n");
    Term.out_str("-----------------------------------------------------------------\n");
    Term.in_int(Menueauswahl);
    CASE Menueauswahl IS
      WHEN 1 DO   -- Konto eroeffnen
        KontoInhaber := NEW (KundenDatenPtrTyp, KundenDatenTyp);
        Term.out_str("Bitte geben Sie die Daten des Kontoinhabers ein:\n");
        Term.out_str("Nachname:"); Term.in_str(KontoInhaber.AllgKundenDaten.Nachname);
        Term.out_str("Vorname:"); Term.in_str(KontoInhaber.AllgKundenDaten.Vorname);
        Term.out_str("Geburtsdatum:"); Term.in_int(KontoInhaber.PersKundenDaten.GebDatum);
        ...
        KontoVerwalter.KontoEroeffnen(KontoInhaber, Kontonummer, ActError)
                                                          STATUS ErlaubterZugriff;
        IF NOT ErlaubterZugriff THEN
          Term.out_str("Sie sind nicht berechtigt, ein neues Konto zu eroeffnen!\n");
        ELSIF ActError = 1 THEN
          Term.out_str("Fuer den Kunden ist noch keine Kennung eingerichtet! Bitte
                       wenden Sie sich hierzu an den Systemadministrator!\n");
        ELSIF ActError = 2 THEN
          Term.out_str("Bei Ausfuehrung der Operation KontoEroeffnen kam es zu einer
                       unerwarteten Zugriffsverletzung!\n");
        ELSE
          Term.out_str("Das Konto wurde erfolgreich eroeffnet. Die Kontonummer lautet:");
          Term.out_int(Kontonummer);
        END IF;
      WHEN 2 DO   -- Konto aufloesen
        Term.out_str("Bitte geben Sie die Kontonummer des aufzuloesenden Kontos ein:");
        Term.in_int(Kontonummer);
        Kontozeiger := ErmittleKonto(Kontonummer, Term);
        IF Kontozeiger /= NULL THEN
          KontoVerwalter.KontoAufloesen(Kontozeiger, ActError) STATUS ErlaubterZugriff;
          IF NOT ErlaubterZugriff THEN
            Term.out_str("Sie sind nicht berechtigt, dieses Konto aufzuloesen!\n");
          ELSIF ActError = 2 THEN
            Term.out_str("Der Kontostand des aufzuloesenden Kontos ist ungleich 0!\n");
          ELSIF (ActError = 1) OR (ActError = 3) THEN
            Term.out_str("Unerwarteter Fehler bei Ausfuehrung von KontoAufloesen!\n");
          ELSIF ActError = 4 THEN
            Term.out_str("Bei Ausfuehrung der Operation KontoAufloesen kam es zu einer
                         unerwarteten Zugriffsverletzung!\n");
          ELSE
            Term.out_str("Das Konto wurde erfolgreich aufgeloest!\n");
          END IF;
        END IF;
      WHEN 3 DO   -- Kundendaten aendern
        Term.out_str("Bitte geben Sie die Kontonummer des Kontos an, dessen Inhaberdaten
                     Sie aendern moechten:");
```

```
    Term.in_int(Kontonummer);
    Kontozeiger := ErmittleKonto(Kontonummer, Term);
    IF Kontozeiger /= NULL THEN
      -- Eingabe der veraenderten Kontoinhaberdaten
      ...
      -- Aendern der Kontoinhaberdaten
      Kontozeiger.AendereKundenDaten(KontoInhaber) STATUS ErlaubterZugriff;
      IF NOT ErlaubterZugriff THEN
        Term.out_str("Sie sind nicht berechtigt, die Daten des Kontoinhabers
                      zu aendern!\n");
      END IF;
    END IF;
  WHEN 4 DO   -- Kontostand lesen
    Term.out_str("Bitte geben Sie eine Kontonummer ein:");
    Term.in_int(Kontonummer);
    Kontozeiger := ErmittleKonto(Kontonummer, Term);
    IF Kontozeiger /= NULL THEN
      Kontostand := Kontozeiger.LeseKontostand STATUS ErlaubterZugriff;
      IF NOT ErlaubterZugriff THEN
        Term.out_str("Sie sind nicht berechtigt, den Kontostand dieses Kontos
                      zu lesen!"\n);
      ELSE
        Term.out_str("Aktueller Kontostand:"); Term.out_real(Kontostand);
      END IF;
    END IF;
  WHEN 5 DO   -- Ueberweisen
    Term.out_str("Bitte geben Sie die fuer die Ueberweisung benoetigten Daten ein:\n");
    Term.out_str("Kontonummer des Auftraggebers:"); Term.in_int(Kontonummer);
    Ueberweisung.VonKontoZeiger := ErmittleKonto(Kontonummer, Term);
    IF Ueberweisung.VonKontoZeiger /= NULL THEN
      Term.out_str("Kontonummer des Empfaengers:"); Term.in_int(EmpfaengerKontoNr);
      Ueberweisung.AufKontoZeiger := ErmittleKonto(EmpfaengerKontoNr, Term);
      IF Ueberweisung.AufKontoZeiger /= NULL THEN
        Term.out_str("Betrag:"); Term.in_real(Ueberweisung.Betrag);
        IF Ueberweisung.Betrag < 0.0 THEN
          Term.out_str("Abbruch: Der zu ueberweisende Betrag muss positiv sein!\n");
        ELSE
          Term.out_str("Zweck:"); Term.in_str(Ueberweisung.Zweck);
          FORK Ueberweiser(Ueberweisung, Ueberweisungsfehler) STATUS ErlaubterZugriff;
          IF NOT ErlaubterZugriff THEN
            Term.out_str("Sie sind nicht berechtigt, diese Ueberweisung
                          durchzufuehren!\n");
          END IF;
        END IF;
      END IF;
    END IF;
  WHEN 6 DO   -- Ueberziehungskredit berechnen
    Term.out_str("Bitte geben Sie die Nummer des Kontos ein, fuer das Sie den maximalen
                  Ueberziehungskredit berechnen moechten:");
    Term.in_int(Kontonummer);
    Kontozeiger := ErmittleKonto(Kontonummer, Term);
    IF Kontozeiger /= NULL THEN
      BerechneKredit(Kontozeiger, Term) STATUS ErlaubterZugriff;
      IF NOT ErlaubterZugriff THEN
        Term.out_str("Sie sind nicht berechtigt, den Ueberziehungskredit fuer dieses
                      Konto zu berechnen!\n");
      END IF:
    END IF;
  WHEN 7 DO
    EXIT ForExit;
```

```
      WHEN OTHERS DO
        Term.out_str("Falsche Eingabe!\n\n");
    END CASE;
  END LOOP;
  END ForExit;
END Kundenbetreuer;

-------------------------------------------------------------------------------------------
-- Benutzerrepraesentant fuer Benutzer in der Rolle Kassierer                            --
-------------------------------------------------------------------------------------------
PROTECTED PROCESS TYPE SPEC Kassierer (Term: IN TermPointer)
  IMPORT TermPointer, KontoPtrTyp, ErmittleKonto, ErrorType;
IS
  ACCESS RESTRICTIONS
    Create : Caller.ActorGen = UserManagement.LoginAtTerminal AND
             Caller.ConGen   = CreateUserRepForRole;
END Kassierer;

PROTECTED PROCESS TYPE Kassierer (Term: IN TermPointer)
  IMPORT ...;
IS
  Menueauswahl       : integer;
  Kontonummer        : integer;
  Betrag             : real;
  Kontozeiger        : KontoPtrTyp := NULL;
  ActError           : ErrorType;
  ErlaubterZugriff   : boolean;

BEGIN
  BLOCK ForExit IS BEGIN
  LOOP
    Term.out_str("------------------------------------------------------------------\n");
    Term.out_str("Folgende Funktionen stehen Ihnen zur Verfuegung:\n");
    Term.out_str("Betrag auf ein Konto einzahlen : 1\n");
    Term.out_str("Betrag von einem Konto abheben : 2\n");
    Term.out_str("Beenden                        : 3\n\n");
    Term.out_str("Waehlen Sie durch Eingabe der jeweiligen Ziffer eine Funktion aus!\n");
    Term.out_str("------------------------------------------------------------------\n");
    Term.in_int(Menueauswahl);
    CASE Menueauswahl IS
      WHEN 1 DO   -- Betrag einzahlen
        Term.out_str("Bitte geben Sie die Kontonummer ein:");
        Term.in_int(Kontonummer);
        Kontozeiger := ErmittleKonto(Kontonummer, Term);
        IF Kontozeiger /= NULL THEN
          Term.out_str("Bitte geben Sie den eingezahlten Betrag ein:");
          Term.in_real(Betrag);
          IF Betrag < 0.0 THEN
            Term.out_str("Der einzuzahlende Betrag muss positiv sein!\n");
          ELSE
            Kontozeiger.Einzahlen("Bareinzahlung", Betrag) STATUS ErlaubterZugriff;
            IF NOT ErlaubterZugriff THEN
              Term.out_str("Sie sind nicht berechtigt, Geld auf dieses Konto
                            einzuzahlen!\n");
            ELSE
              Term.out_str("Einzahlung ist erfolgt!\n");
            END IF;
          END IF;
        END IF;
```

```
      WHEN 2 DO   -- Betrag abheben
        Term.out_str("Bitte geben Sie die Kontonummer ein:");
        Term.in_int(Kontonummer);
        Kontozeiger := ErmittleKonto(Kontonummer, Term);
        IF Kontozeiger /= NULL THEN
          Term.out_str("Bitte geben Sie den abzuhebenden Betrag ein:");
          Term.in_real(Betrag);
          IF Betrag < 0.0 THEN
            Term.out_str("Der abzuhebende Betrag muss positiv sein!\n");
          ELSE
            Kontozeiger.Abheben("Barabhebung", Betrag, TRUE, ActError)
                                                            STATUS ErlaubterZugriff;
            IF NOT ErlaubterZugriff THEN
              Term.out_str("Sie sind nicht berechtigt, Geld von diesem Konto abzuheben!\n");
            ELSIF ActError = 1 THEN
              Term.out_str("Der abzuhebende Betrag ist nicht gedeckt!\n");
            ELSE
              Term.out_str("Auszahlung ist erfolgt!\n");
            END IF;
          END IF;
        END IF;
      WHEN 3 DO
        EXIT ForExit;
      WHEN OTHERS DO
        Term.out_str("Falsche Eingabe!\n\n");
    END CASE;
  END LOOP;
  END ForExit;
END Kassierer;

------------------------------------------------------------------------------------------
-- Benutzerrepraesentant fuer Benutzer in der Rolle Bankleiter                          --
------------------------------------------------------------------------------------------
PROTECTED PROCESS TYPE SPEC Bankleiter (Term: IN TermPointer)
  IMPORT TermPointer, KontoPtrTyp, ErmittleKonto, BerechneKredit, UserUidType,
         UserManagement, ErrorType;
IS
  ACCESS RESTRICTIONS
    Create : Caller.ActorGen = UserManagement.LoginAtTerminal AND
             Caller.ConGen   = CreateUserRepForRole;
END Bankleiter;

PROTECTED PROCESS TYPE Bankleiter (Term: IN TermPointer)
  IMPORT ...;
IS
  Menueauswahl       : integer;
  Kontonummer        : integer;
  Kontozeiger        : KontoPtrTyp := NULL;
  MaxKredit          : real;
  Auswahl            : integer;
  KeinFehler         : boolean := TRUE;
  ViewName           : string;
  Kennung, Rolle     : string;
  BenutzerId         : UserUidType;
  Mode, Flag         : boolean;
  ActError           : ErrorType;
  ErlaubterZugriff   : boolean;
```

```
BEGIN
  BLOCK ForExit IS BEGIN
  LOOP
    Term.out_str("------------------------------------------------------------------\n");
    Term.out_str("Folgende Funktionen stehen Ihnen zur Verfuegung:\n");
    Term.out_str("Ueberziehungskredit berechnen          : 1\n");
    Term.out_str("Ueberziehungskredit setzen             : 2\n");
    Term.out_str("Rechte an einem Konto aendern          : 3\n");
    Term.out_str("Konto sperren                          : 4\n");
    Term.out_str("Kontobetreuung uebertragen             : 5\n");
    Term.out_str("Kontobetreuung entziehen               : 6\n");
    ...
    Term.out_str("Beenden                                : 99\n\n");
    Term.out_str("Waehlen Sie durch Eingabe der jeweiligen Ziffer eine Funktion aus!\n");
    Term.out_str("------------------------------------------------------------------\n");
    Term.in_int(Menueauswahl);
    CASE Menueauswahl IS
      WHEN 1 DO   -- Ueberziehungskredit berechnen
        Term.out_str("Bitte geben Sie die Nummer des Kontos ein, fuer das Sie den maximalen
                      Ueberziehungskredit berechnen moechten:");
        Term.in_int(Kontonummer);
        Kontozeiger := ErmittleKonto(Kontonummer, Term);
        IF Kontozeiger /= NULL THEN
          BerechneKredit(Kontozeiger, Term) STATUS ErlaubterZugriff;
          IF NOT ErlaubterZugriff THEN
            -- dieser Fall kann gemaess der Rechtsfestlegungen fuer die Operation
            -- "BerechneKredit" nicht eintreten!
            Term.out_str("Unerwartete Zugriffsverletzung: Sie sind nicht berechtigt,
                          den Ueberziehungskredit fuer dieses Konto zu berechnen!\n");
          END IF:
        END IF;
      WHEN 2 DO   -- Ueberziehungskredit setzen
        Term.out_str("Bitte geben Sie die Nummer des Kontos ein, fuer das Sie den maximalen
                      Ueberziehungskredit setzen moechten:");
        Term.in_int(Kontonummer);
        Kontozeiger := ErmittleKonto(Kontonummer, Term);
        IF Kontozeiger /= NULL THEN
          Term.out_str("Bitte geben Sie den Ueberziehungskredit fuer dieses Konto ein:");
          Term.in_real(MaxKredit);
          IF MaxKredit < 0.0 THEN
            Term.out_str("Der Ueberziehungskredit muss positiv sein!\n");
          ELSE
            Kontozeiger.SetzeMaxKredit(MaxKredit) STATUS ErlaubterZugriff;
            IF NOT ErlaubterZugriff THEN
              -- dieser Fall kann gemaess der Rechtsfestlegungen fuer die Operation
              -- "SetzeMaxKredit" eines Kontos nicht eintreten!
              Term.out_str("Unerwartete Zugriffsverletzung: Sie sind nicht berechtigt,
                            den Ueberziehungskredit fuer dieses Konto zu setzen!\n");
            END IF;
          END IF;
        END IF;
      WHEN 3 DO   -- Rechte an einem Konto aendern
        Term.out_str("Bitte geben Sie die Kontonummer ein:");
        Term.in_int(Kontonummer);
        Kontozeiger := ErmittleKonto(Kontonummer, Term);
        IF Kontozeiger /= NULL THEN
          Term.out_str("Geben Sie durch Eingabe der Nummer die Operation/den View an, fuer
                        die/den Sie das Recht an dem Konto aendern moechten!\n");
          Term.out_str("Kontostand lesen     : 1\n");
          Term.out_str("Kontoauszug ausgeben : 2\n");
```

```
Term.out_str("Geld abheben         : 3\n");
Term.out_str("Kredit berechnen     : 4\n");
...
Term.in_int(Auswahl);
CASE Auswahl IS
  WHEN 1 DO
    ViewName := "LeseKontostand";
  WHEN 2 DO
    ViewName := "ZeigeKontoauszug";
  WHEN 3 DO
    ViewName := "Abheben";
  WHEN 4 DO
    ViewName := "KreditBerechnung";
  ...
  WHEN OTHERS DO
    KeinFehler := FALSE;
    Term.out_str("Falsche Eingabe!\n\n");
END CASE;
IF KeinFehler THEN
  Term.out_str("Geben Sie die Benutzer/Rollenkombination ein, fuer die Sie das
              Recht an der eingegebenen Operation/View aendern moechten!\n");
  Term.out_str("Benutzerkennung:"); Term.in_str(Kennung);
  Term.out_str("Rolle:"); Term.in_str(Rolle);
  IF Kennung = "-1" THEN
    BenutzerId := -1;
  ELSE
    UserManagement.UserDataManager.GetUid(Kennung, BenutzerId, ActError);
    IF ActError = 1 THEN
      Term.out_str("Fehler: Einen Benutzer mit dieser Kennung gibt es nicht!\n");
      KeinFehler := FALSE;
    END IF;
  END IF;
  IF (Rolle /= "-1") AND (NOT UserManagement.RoleSet.InSet(Rolle)) THEN
    Term.out_str("Fehler: Eine solche Rolle gibt es nicht!\n");
    KeinFehler := FALSE;
  END IF;
  IF KeinFehler THEN
    Term.out_str("Moechten Sie das Recht an der Operation/View vergeben (1) oder
              zuruecknehmen (2)? Bitte geben Sie die jeweilige Nummer an!\n");
    Term.in_int(Auswahl);
    IF Auswahl = 1 THEN
      Flag := TRUE;
    ELSIF Auswahl = 2 THEN
      Flag := FALSE;
    ELSE
      KeinFehler := FALSE;
      Term.out_str("Falsche Eingabe!\n\n");
    END IF;
    IF KeinFehler THEN
      Term.out_str("Handelt es sich um ein positives (1) oder ein negatives (2)
              Recht? Bitte geben Sie die jeweilige Nummer an!\n");
      Term.in_int(Auswahl);
      IF Auswahl = 1 THEN
        Mode := TRUE;
      ELSIF Auswahl = 2 THEN
        Mode := FALSE;
      ELSE
        KeinFehler := FALSE;
        Term.out_str("Falsche Eingabe!\n\n");
      END IF;
```

```
          IF KeinFehler THEN
            Kontozeiger.ChangeACL(ViewName, BenutzerId, Rolle, Mode, Flag, ActError)
                                                         STATUS ErlaubterZugriff;
            IF NOT ErlaubterZugriff THEN
              -- dieser Fall kann gemaess der Rechtsfestlegungen fuer die Operation
              -- "ChangeACL" auf Konten nicht eintreten!
              Term.out_str("Unerwartete Rechteverletzung!\n");
            ELSE
            CASE ActError IS
              WHEN 0 | 3 | 5 DO
                Term.out_str("Rechteaenderung erfolgreich durchgefuehrt!"\n);
              WHEN 2 | 6 DO
                Term.out_str("Das Recht ist bereits vergeben!"\n);
              WHEN 4 | 7 | 8 | 9 DO
                Term.out_str("Das Recht ist nicht vergeben!"\n");
              WHEN OTHERS DO
                Term.out_str("Interner Fehler!"\n);
            END CASE;
            END IF;
          END IF;
        END IF;
      END IF;
    END IF;
  END IF;
WHEN 4 DO   -- Konto sperren, d. h. allen Benutzern das Recht zum Abheben und allen
            -- Kunden alle weiteren moeglichen Kundenrechte an dem Konto entziehen
  Term.out_str("Bitte geben Sie die Nummer des zu sperrenden Kontos ein:");
  Term.in_int(Kontonummer);
  Kontozeiger := ErmittleKonto(Kontonummer, Term);
  IF Kontozeiger /= NULL THEN
    -- allen Benutzern das Recht zum Abheben entziehen
    Kontozeiger.ChangeACL("Abheben", -1, "-1", FALSE, TRUE, ActError)
                                                         STATUS ErlaubterZugriff;
    IF NOT ErlaubterZugriff THEN
      -- dieser Fall kann gemaess der Rechtsfestlegungen fuer die Operation
      -- "ChangeACL" auf Konten nicht eintreten!
      Term.out_str("Unerwartete Rechteverletzung!\n");
    ELSE
      -- Analyse des Rueckgabewerts "ActError"
      ...
    END IF;
    -- allen Kunden alle weiteren moeglichen Kundenrechte entziehen
    -- a) Recht zum Rechte aendern entziehen
    Kontozeiger.ChangeACL("ChangeACL", -1, "Kunde", FALSE, TRUE, ActError)
                                                         STATUS ErlaubterZugriff;
    IF NOT ErlaubterZugriff THEN
      -- dieser Fall kann gemaess der Rechtsfestlegungen fuer die Operation
      -- "ChangeACL" auf Konten nicht eintreten!
      Term.out_str("Unerwartete Rechteverletzung!\n");
    ELSE
      -- Analyse des Rueckgabewerts "ActError"
      ...
    END IF;
    -- b) Recht zum Lesen des Kontostands entziehen
    Kontozeiger.ChangeACL("LeseKontostand", -1, "Kunde", FALSE, TRUE, ActError)
                                                         STATUS ErlaubterZugriff;
    ...
    -- c) Recht zum Kontoauszug ausgeben entziehen
    Kontozeiger.ChangeACL("ZeigeKontoauszug", -1, "Kunde", FALSE, TRUE, ActError)
                                                         STATUS ErlaubterZugriff;
```

```
    ...
  END IF;
WHEN 5 DO   -- Kontobetreuung uebertragen, d.h. einem Kundenbetreuer alle moeglichen
            -- Kundenbetreuerrechte an einem Konto erteilen
  Term.out_str("Bitte geben Sie die Nummer des Kontos ein, dessen Betreuung Sie
               uebertragen moechten:");
  Term.in_int(Kontonummer);
  Kontozeiger := ErmittleKonto(Kontonummer, Term);
  IF Kontozeiger /= NULL THEN
    Term.out_str("Bitte geben Sie die Kennung des Kundenbetreuers ein, an den Sie
                 die Betreuung uebertragen moechten:");
    Term.in_str(Kennung);
    UserManagement.UserDataManager.GetUid(Kennung, BenutzerId, ActError);
    IF ActError = 1 THEN
      Term.out_str("Fehler: Einen Kundenbetreuer mit dieser Kennung gibt
                   es nicht!\n");
    ELSE
      -- a) Recht zum Lesen des Kontostands vergeben
      KontoZeiger.ChangeACL("LeseKontostand", BenutzerId, "Kundenbetreuer", TRUE,
                            TRUE, ActError) STATUS ErlaubterZugriff;
      IF NOT ErlaubterZugriff THEN
        -- dieser Fall kann gemaess der Rechtsfestlegungen fuer die Operation
        -- "ChangeACL" auf Konten nicht eintreten!
        Term.out_str("Unerwartete Rechteverletzung!\n");
      ELSE
        -- Analyse des Rueckgabewerts "ActError"
        ...
      END IF;
      -- b) Recht zum Aufloesen des Kontos vergeben
      KontoZeiger.ChangeACL("Aufloesen", BenutzerId, "Kundenbetreuer", TRUE,
                            TRUE, ActError) STATUS ErlaubterZugriff;
      ...
      -- c) Recht zur Berechnung des maximalen Ueberziehungskredits vergeben
      KontoZeiger.ChangeACL("Kreditberechnung", BenutzerId, "Kundenbetreuer", TRUE,
                            TRUE, ActError) STATUS ErlaubterZugriff;
      ...
    END IF;
  END IF;
WHEN 6 DO   -- Kontobetreuung entziehen, d.h. einem Kundenbetreuer alle moeglichen
            -- Kundenbetreuerrechte an einem Konto entziehen
  Term.out_str("Bitte geben Sie die Nummer des Kontos ein, dessen Betreuung Sie
               einem Kundenbetreuer entziehen moechten:");
  Term.in_int(Kontonummer);
  Kontozeiger := ErmittleKonto(Kontonummer, Term);
  IF Kontozeiger /= NULL THEN
    Term.out_str("Bitte geben Sie die Kennung des Kundenbetreuers ein, dem Sie
                 die Betreuung des Kontos entziehen moechten:");
    Term.in_str(Kennung);
    UserManagement.UserDataManager.GetUid(Kennung, BenutzerId, ActError);
    IF ActError = 1 THEN
      Term.out_str("Fehler: Einen Kundenbetreuer mit dieser Kennung gibt
                   es nicht!\n");
    ELSE
      -- a) Recht zum Lesen des Kontostands entziehen
      KontoZeiger.ChangeACL("LeseKontostand", BenutzerId, "Kundenbetreuer", TRUE,
                            FALSE, ActError) STATUS ErlaubterZugriff;
      IF NOT ErlaubterZugriff THEN
        -- dieser Fall kann gemaess der Rechtsfestlegungen fuer die Operation
        -- "ChangeACL" auf Konten nicht eintreten!
        Term.out_str("Unerwartete Rechteverletzung!\n");
```

```
              ELSE
                -- Analyse des Rueckgabewerts "ActError"
                ...
              END IF;
              -- b) Recht zum Aufloesen des Kontos vergeben
              KontoZeiger.ChangeACL("Aufloesen", BenutzerId, "Kundenbetreuer", TRUE,
                                    FALSE, ActError) STATUS ErlaubterZugriff;
              ...
              -- c) Recht zur Berechnung des maximalen Ueberziehungskredits vergeben
              KontoZeiger.ChangeACL("Kreditberechnung", BenutzerId, "Kundenbetreuer", TRUE,
                                    FALSE, ActError) STATUS ErlaubterZugriff;
              ...
            END IF;
          END IF;
        ...
        WHEN 99 DO
          EXIT ForExit;
        WHEN OTHERS DO
          Term.out_str("Falsche Eingabe!\n\n");
      END CASE;
    END LOOP;
    END ForExit;
  END Bankleiter;

-----------------------------------------------------------------------------------------------
-- Prozedur zur Erzeugung der rollenspezifischen Benutzerrepraesentanten                     --
-----------------------------------------------------------------------------------------------

  PROCEDURE TYPE CreateUserRepForRole (Role: IN string; Term: IN TermPointer) IS
    ErlaubterZugriff : boolean;
  BEGIN
    IF Role = "Kunde" THEN
      FORK Kunde(Term) STATUS ErlaubterZugriff;
    ELSIF Role = "Kundenbetreuer" THEN
      FORK Kundenbetreuer(Term) STATUS ErlaubterZugriff;
    ELSIF Role = "Kassierer" THEN
      FORK Kassierer(Term) STATUS ErlaubterZugriff;
    ELSIF Role = "Bankleiter" THEN
      FORK Bankleiter(Term) STATUS ErlaubterZugriff;
    ELSIF Role = "SysAdmin" THEN
      FORK UserManagement.SysAdmin(Term) STATUS ErlaubterZugriff;
    ELSE
      Term.out_str("Unbekannte Rolle!\n");
    END IF;
    IF NOT ErlaubterZugriff THEN
      Term.out_str("Zugriffsverletzung bei Erzeugung des Benutzerrepraesentanten!\n")
    END IF;
  END CreateUserRepForRole;

-----------------------------------------------------------------------------------------------
-- Anweisungsteil der Hauptkomponente sowie dort benoetigte lokale Variablen                 --
-----------------------------------------------------------------------------------------------
  ZentralKontoDaten    : KundenDatenPtrTyp := NEW(KundenDatenPtrTyp, KundenDatenTyp);
  ZentralKontoBenutzer : UserNameType;
  ZentralKontoNummer   : integer;
  ActError             : ErrorType;

BEGIN
  IF NOT ErlaubterZugriff THEN
    OUTPUT "Zugriffsverletzung beim Starten/Initialisieren des Systems!";
```

```
  ELSE
    -- Initialisierung der Benutzerverwaltung
    InitRoles;
    InitTerminals;
    UserManagement.CreateRootAccount;
    -- Kennung fuer Benutzung des Zentralkontos der Bank erzeugen
    ZentralKontoBenutzer.Surname := "ZentralKontoBenutzer";
    UserManagement.UserDataManager.NewAccount("Zentralkonto", "Initial", ZentralKontoBenutzer,
                                              "Kunde", ActError);
    IF ActError /= 0 THEN
      OUTPUT "Fehler beim Erzeugen der Zentralkonto-Benutzer-Kennung!";
    ELSE
      -- Zentralkonto der Bank erzeugen
      ZentralKontoDaten.AllgKundenDaten.Nachname := "Zentralkonto";
      KontoVerwalter.KontoEroeffnen(ZentralKontoDaten, ZentralKontoNummer, ActError)
                                                                   STATUS ErlaubterZugriff;
      IF (NOT ErlauberZugriff) OR (ActError /= 0) THEN
        OUTPUT "Fehler/Zugriffsverletzung beim Erzeugen des Zentralkontos!";
      ELSE
        -- Erzeugen der Login-Akteure
        FOR I IN 1..NumberOfInitialTerminals LOOP
          FORK UserManagement.LoginAtTerminal(UserManagement.TerminalNameSet.GetElement(I));
        END LOOP;
      END IF;
    END IF;
  END IF:
END KontenVerwaltungsSystem;
```

Abbildungsverzeichnis

2.1 Systematischer Konstruktionsprozeß für sichere Systeme . . . 23

3.1 Klassifikation der INSEL–Komponenten . . . 40
3.2 INSEL–Implementierung eines Erzeuger–Verbraucher–Systems . . . 46
3.3 Definitions– und Ausführungsstruktur des Erzeuger–Verbraucher–Systems aus Abbildung 3.2 zu einem Zeitpunkt t . . . 51
3.4 Lebenszeitstruktur des Erzeuger–Verbraucher–Systems aus Abbildung 3.2 zum Zeitpunkt t . . . 55
3.5 Ausführungsphasen einer kanonischen Operation . . . 56

4.1 Beispiel für einen Rollen–Part . . . 70
4.2 Ausschnitt aus einer Systemkonfiguration des Kontenverwaltungssystems . . . 72
4.3 INSEL$^+$–Implementierung des Erzeuger–Verbraucher–Systems . . . 77
4.4 INSEL$^+$–Beispielprogramm für Operationen–qualifizierte Zeiger . . . 81
4.5 Klassifikation der INSEL$^+$–Komponenten . . . 83
4.6 Beispiel für einen Spezifikationsteil . . . 84
4.7 Beispiel für einen View–Part . . . 89
4.8 Beispiel für einen ACL–Part . . . 95
4.9 Beispiel für einen `ChangeACL`–Aufruf . . . 98
4.10 Beispiel für einen Zugriffsrestriktions–Part . . . 101
4.11 Auswertung des `IN_ACL`-Prädikats . . . 105
4.12 Beispiel für ein `IN_ACL`–Prädikat . . . 108
4.13 Beispiel für `ACCESSED`–Prädikate . . . 111
4.14 Ausschnitt aus dem INSEL$^+$–Programm des Kontenverwaltungssystems . . . 117
4.15 Beispiel 1 für einen Status–Part . . . 125
4.16 Beispiel 2 für einen Status–Part . . . 126
4.17 Spezifikationsteil des K-Akteur–Generators `KontoVerwalterTyp` . . . 135
4.18 Ausschnitt aus dem Spezifikationsteil des Depot–Generators `KontoTyp` . . . 137
4.19 Implementierung des Hausaufgabenverwaltungssystems unter Verwendung boolescher DE–Inkarnationen . . . 141

4.20 Implementierung des Hausaufgabenverwaltungssystems unter Verwendung des Zugriffsrestriktionsprädikats ACCESSED 142

5.1 Übersicht über die Maßnahmen zur Gewährleistung der Konsistenz des Rechts eines INSEL$^+$–Systems 163

5.2 INSEL$^+$–Beispielprogramm: Erzeuger–Verbraucher–System 165

5.3 Schachtelungsstruktur des INSEL$^+$–Programms aus Abbildung 5.2 169

5.4 Statischer Aufrufgraph des INSEL$^+$–Programms aus Abbildung 5.2 173

5.5 Statischer Aufrufgraph des Kontenverwaltungssystems (Ausschnitt) 174

5.6 Ausschnitt aus dem statischen Aufrufgraph des Kontenverwaltungssystems . 178

5.7 Ausschnitt aus dem statischen Aufrufgraph eines INSEL$^+$–Programms 205

5.8 Ausschnitt aus dem statischen Aufrufgraph des Kontenverwaltungssystems zur Erläuterung der Definition (5.23) 208

5.9 Analyse der Widerspruchsfreiheit zweier Zugriffsrestriktionsausdrücke 214

5.10 Analyse der absoluten Konsistenz des Rechts eines INSEL$^+$–Systems 234

5.11 An Leitlinie 2 orientierte alternative Implementierung des Depot–Generators KontobewegungsListenTyp 237

6.1 Vergröberung einer Systemkonfiguration des Kontenverwaltungssystems zu einer Menge von Akteursphären 254

6.2 Verwalterbaum der Systemkonfiguration des Kontenverwaltungssystems aus Abbildung 6.1 255

6.3 Lebenszeitkeller der Akteursphärenverwalter des Kontenverwaltungssystems für die Systemkonfiguration aus Abbildung 6.1 264

6.4 Durchführung von Zugriffskontrollen auf Realisierungsebene 1 279

6.5 Aufbau eines Tickets 285

6.6 Beispiel für das Löschen eines Tickets 290

6.7 Beispiel für den Einsatz des Stellvertreter–Verwalter Konzepts 322

Tabellenverzeichnis

4.1 Rechte an zugriffskontrollierten Komponenten 86

4.2 INSEL$^+$–Bezeichner der Elemente des `Caller`–Attributs 114

4.3 Kontextrestriktionen 121

6.1 Schnittstellenoperationen der Akteursphärenverwalter zur Erzeugung von Inkarnationen 261

6.2 Schnittstellenoperationen der Akteursphärenverwalter zur Terminierung und Auflösung von Inkarnationen 263

6.3 Schnittstellenoperationen der Akteursphärenverwalter zur Realisierung von K-Order–Aufrufen und –Annahmen 266

6.4 Schnittstellenoperationen der Akteursphärenverwalter zur Auswertung von Zugriffskontrollisten und zur Realisierung der Operationen *change_acl* und *list_acl* 270

6.5 Schnittstellenoperationen der Akteursphärenverwalter zur Auswertung und Änderung von Zugriffshistorienlisten 272

6.6 Schnittstellenoperation der Akteursphärenverwalter zur Auswertung eines Zugriffsrestriktionsausdrucks 274

6.7 Schnittstellenoperationen der Akteursphärenverwalter zum Erwerb und zur Freigabe von Leser–Schreiber–Sperren 276

6.8 Schnittstellenoperationen der Akteursphärenverwalter zum Löschen von Tickets 288

6.9 Gewichtung von Objekten, die in Zugriffsrestriktionsausdrücken auftreten können 295

6.10 Entscheidungsmatrix für die Vergabe eines Tickets 298

6.11 Schnittstellenoperationen der Akteursphärenverwalter zur Realisierung stellenübergreifender Zugriffe auf passive Inkarnationen 306

6.12 Schnittstellenoperationen der Schlüsselverwalter und der Akteursphärenverwalter zur Registrierung öffentlicher Schlüssel und zur wechselseitigen Authentifizierung 315

6.13 Schnittstellenoperationen der Akteursphärenverwalter zur Realisierung des Konzepts der Stellvertreter–Verwalter 320

Literaturverzeichnis

[ABG+86] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation For UNIX Development. Technical report, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, August 1986.

[ABL83] P. Ancilotti, M. Boari, and N. Lijtmaer. Language Features for Access Control. *IEEE Transactions on Software Engineering*, SE-9(1):16–25, January 1983.

[Ada83] Ada. *The Programming Language Ada Reference Manual*, volume 155 of *Lecture Notes in Computer Science*. Springer–Verlag, Berlin, 1983.

[AK93] E. Amann and V. Kessler. Sicherheitsmodelle in Theorie und Praxis. In P.P. Spies, editor, *Euro–ARCH '93*, pages 242–254. Springer–Verlag, October 1993.

[Bac93] Jean Bacon. *Concurrent Systems*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1993. ISBN 0–201–41677–8.

[BAN90] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[Bas93] R. Baskerville. Information Systems Security Design Methods: Implications for Information System Development. *ACM Computing Surveys*, 25(4), December 1993.

[BBD+91] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, and G. Vandome. Architecture and Implementation of GUIDE, an Object–oriented Distributed System. *Computing Systems*, 4(1):31–67, 1991.

[BDS94] BDSG. Bundesdatenschutzgesetz. In *Datenschutzrecht*, Der Bayerische Landesbeauftragte für den Datenschutz, München, 1994.

[Bir85] A.D. Birrell. Secure Communication Using Remote Procedure Calls. *ACM Transactions on Computer Systems*, 3(1):1–14, 1985.

[BJS96] E. Bertino, Sushil Jajodia, and Pierangela Samarati. Supporting multiple access control policies in database systems. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 1996.

[BKLL93] J. Boykin, D. Kirschen, A. Langerman, and S. LoVerso. *Programming under Mach*. Addison–Wesley, 1993.

[BL75] D.E. Bell and L. LaPadula. Secure Computer Systems: Unified Exposition and Multics Interpretation. Technical report, MTR 2997, ESD-TR-75-306, MITRE Corporation, Bedford MA, July 1975.

[BLR94] Roland Balter, Serge Lacourte, and Michel Riveill. The Guide Language. *The Computer Journal*, 37(6):521–530, 1994.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BN89] D.F.C. Brewer and M.J. Nash. The Chinese Wall Security Policy. In *Proceedings of the 1989 IEEE Symposium on Research in Security and Privacy*, pages 206–214, 1989.

[Bru93] H.H. Brueggemann. Prioritäten für eine verteilte, objekt–orientierte Zugriffskontrolle. In G. Weck and P. Horster, editors, *Proceedings der GI–Fachtagung VIS'93, Verläßliche Informationssysteme*, pages 51–66, München, Mai 1993. DuD–Fachbeiträge 16, Vieweg.

[BS92] E. Biham and A. Shamir. Differential Cryptanalysis of the full 16-round DES. In *Lecture Notes in Computer Science: Advances in Cryptology – CRYPTO '92*, pages 487–497. Springer–Verlag, 1992.

[BSI92] Bundesamt für Sicherheit in der Informationstechnik BSI. *IT–Sicherheitshandbuch*. Bundesanzeiger Verlag, 1992.

[CC98] CC. *Common Criteria for Information Technology Security Evaluation*. Version 2.0, 1998.

[CD94] G. Coulouris and J. Dollimore. A security model for cooperative work. Technical Report 674, Department of Computer Science, Queen Mary and Westfield College, University of London, August 1994.

[CFL84] P. Corsini, G. Frosini, and L. Lopriore. Distributing and Revoking Access Authorizations on Abstract Objects: A Capability Approach. *Software – Practice and Experience*, 14(10):931–943, October 1984.

[Cha85] D.L. Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030, 1985.

[CLFL94] J.S. Chase, H.M. Levy, M.J. Feeley, and E.D. Lazowska. Sharing and Protection in a Single Address Space Operating System. *ACM Transactions on Computer Systems*, 12(4):271–307, 1994.

[Com91] Office for Official Publications of the European Communities. *Information Technology Security Evaluation Criteria (ITSEC)*. Catalogue Number CD-71-91-502-EN-C, 1991.

[CW87] D.D. Clark and D.R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of the 1987 IEEE Symposium on Research in Security and Privacy*, pages 184–194, 1987.

[Cze97a] C. Czech. *Architektur und Konzept des Dycos Kerns*. Technical Report TUM-I9717, SFB 342/12/97 A, SFB 342, Technische Universität München, April 1997.

[Cze97b] C. Czech. Dycos – A customizable kernel architecture supporting distributed operating environments. In *Proceedings of the 3rd International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'97)*, pages 203–210, Melbourne, Australia, December 1997.

[DABW96] L. van Doorn, M. Abadi, M. Burrows, and E. Wobber. Secure Network Objects. In *Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 211–221, Oakland, California, May 1996.

[DAH+87] D.E. Denning, S.G. Akl, M. Heckman, T.F. Lunt, M. Morgenstern, P.G. Neumann, and R.R. Schell. Views for Multilevel Database Security. *IEEE Transactions on Software Engineering*, 13(2):129–140, February 1987.

[DBF+93a] A. Dearle, R. di Bona, J. Farrow, F. Henskens, D. Hulse, A. Lindström, S. Norris, J. Rosenberg, and F. Vaughan. Protection in the Grasshopper Operating System. In *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*, Ashville, North Carolina, 1993. IEEE Computer Society.

[DBF+93b] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindström, J. Rosenberg, and F. Vaughan. Grasshopper: An orthogonally persistent operating system. Technical report, Departments of Computer Science, Universities of Adelaide and Sydney, 1993.

[Def85] Department of Defense. *Trusted Computing System Evaluation Criteria.* Technical Report CSC-STD-001-83, December 1985.

[DFW96] D. Dean, E.W. Felton, and D.S. Wallach. Java Security: From Hotjava to Netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 190–200, Oakland, California, May 1996.

[DH66] J.B. Dennies and E.C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, 9(3):143–155, 1966.

[DH76] W. Diffie and M.E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[DS81] D.E. Denning and G.M. Sacco. Timestamps in Key Distribution Protocols. *Communications of the ACM*, 24(8):533, 1981.

[Eck93] C. Eckert. *Konzepte und Verfahren zur Konstruktion sicherer, verteilter Systeme.* Dissertation, Fakultät für Informatik, Technische Universität München, November 1993.

[Eck95] C. Eckert. Matching Security Policies to Application Needs. In J. H.P. Eloff and S.H. von Solms, editors, *Proceedings of the IFIP TC11 11th International Conference on Information Security*, pages 237–254, Cape Town, South Africa, May 1995. Chapman & Hall.

[Eck96] C. Eckert. Issues in the Design of Modern Distributed Computing Environments. In *Proceedings of the 8th IASTED International Conference on Parallel and Distributed Computing and Systems*, Chicago, USA, October 1996.

[Eck98] C. Eckert. *Sichere verteilte Systeme – Konzepte, Modelle und Systemarchitekturen.* Habilitationsschrift, Fakultät für Informatik, Technische Universität München, November 1998.

[ElG85] T. ElGamal. A Public–Key Cryptosystem and a Signature Scheme based on Discrete Logarithms. *IEEE Transactions on Information Theory*, IT-31(4):469–472, 1985.

[EM97] C. Eckert and D. Marek. Developing Secure Applications: A Systematic Approach. In *Proceedings of the IFIP TC11 13th International Conference on Information Security*, pages 267–279, Kopenhagen, Denmark, May 1997. Chapman & Hall.

[EP98] C. Eckert and M. Pizka. Improving Resource Management in Distributed Systems using Language-level Structuring Concepts. *The Journal of Supercomputing*, 13(1):275–298, 1998.

[EPR97] C. Eckert, M. Pizka, and N. Reimer. Konzepte und Verfahren zur Konstruktion heteromorph paralleler Systeme. In *Arbeits– und Ergebnisbericht 1995–1997 des SFB 342: Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen*, pages 209–243. Technische Universität München, Juni 1997.

[EW95] C. Eckert and H.-M. Windisch. A Top–down Driven, Object–based Approach to Application–secific Operating System Design. In M. Theimer and L. Cabrera, editors, *Proceedings of the IEEE International Workshop on Object–Orientation in Operating Systems*, pages 153–156, Lund, Sweden, August 1995.

[Fab74] R.S. Fabry. Capability–based addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[Fel96] Michael B. Feldman. *Software Construction and Data Structures with Ada 95.* Addison–Wesley Publishing Company, Reading, Massachusetts, 1996.

[Fou98] Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design.* O'Reilly & Associates, 1998.

[FR94] W. Fumy and H.P. Ries. *Kryptographie: Entwurf, Einsatz und Analyse von symmetrischen Kryptosystemen.* R. Oldenbourg Verlag, München, 1994.

[Gas88] M. Gasser. *Building a Secure Computer System.* van nostrad Reinhild, New York, 1988.

[GD72] G.S. Graham and P.J. Denning. Protection – Principles and Practice. In *Proceedings of the Spring Joint Computer Conference*, pages 417–429, 1972.

[Gei95] J. Geiger. *Formale Methoden zur Verifikation kryptographischer Protokolle.* Technischer Bericht, Fortgeschrittenenpraktikum, Fakultät für Informatik, Technische Universität München, Dezember 1995.

[Gif82] D.K. Gifford. Cryptographic Sealing for Information Secrecy and Authentication. *Communications of the ACM*, 25(4):274–286, April 1982.

[GM82] J.A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of the 1982 IEEE Symposium on Research in Security and Privacy*, pages 11–20, 1982.

[Gro98] S. Groh. *Ein agentenbasiertes, flexibel anpassungsfähiges Ressourcenmanagement für verteilte, parallele und kooperative Systeme.* Dissertation, Fakultät für Informatik, Technische Universität München, 1998.

[GW76] P.P. Griffiths and B.W. Wade. An Authorization Mechanism for a Relational Database System. *ACM Transactions on Database Systems*, 1(3):242–255, September 1976.

[Hag94] D. Hagimont. Protection in the Guide object–oriented distributed system. In M. Tokoro and E. Pareschi, editors, *Proceedings of the European Conference on Object–Oriented Programming (ECOOP) 1994*, pages 280–298, Bologna, July 1994. Springer Verlag, Lecture Notes in Computer Science, No. 821.

[Hew91] Hewlett–Packard Company. *HP–UX System Security*. Hewlett Packard, Manual Part No. B1862-90009, 2 edition, January 1991.

[Hew92] Hewlett–Packard Company. *HP–UX Reference, Volume 1–3*. Hewlett Packard, 3 edition, January 1992.

[HHM97] D. Hagimont, O. Huet, and J. Mossière. A protection Scheme for a CORBA Environment. In *Proceedings of the ECCOOP'97 Workshop on CORBA, Implementation, Use and Evaluation*, Jyväskyllä, Finland, June 1997.

[HKMR96] D. Hagimont, S. Krakowiak, J. Mossière, and X. Rousset de Pina. A Selective Protection Scheme for the Java Environment. Technical Report RT-Sirac-TR-96, Project Sirac, June 1996.

[HMR96] D. Hagimont, J. Mossière, and X. Rousset de Pina. Hidden Capabilities: Towards a Flexible Protection Utility for the Internet. In *Proceedings of the 7th ACM European SIGOPS Workshop*, Connemare, Ireland, September 1996.

[HMRS96] D. Hagimont, J. Mossière, X. Rousset de Pina, and F. Saunier. Hidden Software Capabilities. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 282–289, Hong–Kong, May 1996. IEEE Computer Society.

[HO90] B. Hailpern and H. Ossher. Extending Objects to Support Multiple Interfaces and Access Control. *IEEE Transactions on Software Engineering*, 16(11):1247–1257, November 1990.

[Hor85] P. Horster. *Kryptologie*. BI Reihe Informatik Nr. 47, Bibliographisches Institut, Mannheim, 1985.

[HS89] A.J. Hurst and A.S.M. Sajeev. A Capability Based Language for Persistent Programming: Implementation Issues. In J. Rosenberg and D. Koch, editors, *Proceedings of the Workshop on Persistent Object Systems*, pages 109–125, Newcastle, Australia, 1989. Springer Verlag.

[HS92] A.J. Hurst and A.S.M. Sajeev. Programming Persistence in $\mathcal{X}$. *IEEE Computer*, 25(9):57–66, 1992.

[HT95] Ralph Holbein and Stephanie Teufel. A Context Authentication Service for Role Based Access Control in Distributed Systems – CARDS. In J. H.P. Eloff and S.H. von Solms, editors, *Proceedings of the IFIP TC11 11th International Conference on Information Security*, pages 270–285, Cape Town, South Africa, May 1995. Chapman & Hall.

[Hu95] Wei Hu. *DCE Security Programming*. O'Reilly & Associates, Inc., 1995.

[Hwa93] Kai Hwang. *Advanced Computer Architecture - Parallelism, Scalability, Programmability.* McGraw-Hill Series in Computer Science, 1993.

[Jac95] H. Jacobsen. *Entwurf und Realisierung eines Konzepts zur dezentralen Zugriffskontrolle für INSEL–Systeme.* Diplomarbeit, Institut für Informatik, Technische Universität München, August 1995.

[JD95] D. Jonscher and K. Dittrich. Argos – A Configurable Access Control System for Interoperable Environments. In D.L. Spooner, S.A. Demurjian, and J.E. Dobson, editors, *Database Security IX – Status and Prospects, Proceedings of the Ninth IFIP WG11.3 Working Conference on Database Security*, pages 43–61, Rensselaerville, New York, USA, August 1995. Chapman & Hall.

[JL78] A.K. Jones and B.H. Liskov. A Language Extension for Expressing Constraints on Data Access. *Communications of the ACM*, 21(5):358–367, May 1978.

[Jon89] Oliver Jones. *Introduction to the X Window System.* Prentice–Hall, Inc., Englewood Cliffs, NJ, 1989.

[KE96] A. Kemper and A. Eickler. *Datenbanksysteme – Eine Einführung.* Oldenbourg Verlag, München, 1996.

[Kel90] U. Kelter. Group-Oriented Discretionary Access Controls for Distributed Structurally Object-oriented Database Systems. In *Proceedings of the European Symposium on Research in Computer Security*, Toulouse, October 1990.

[Ken93] St.T. Kent. Internet Privacy Enhanced Mail. *Communications of the ACM*, 36(8):48–60, 1993.

[KH91] P. Kraaibeek and P. Horster. Integrität in IT–Systemen. In A. Pfitzmann and E. Raubold, editors, *VIS'91 – Verläßliche Informationssysteme*, pages 82–91, Darmstadt, März 1991.

[KNT94] J.T. Kohl, B.C. Neuman, and T.Y. Tso. The Evolution of the Kerberos Authentication System. In *Distributed Open Systems*, pages 78–94. IEEE Computer Society Press, 1994.

[KS78] R.B. Kieburtz and A. Silberschatz. Capability Managers. *IEEE Transactions on Software Engineering*, SE-4(6):467–477, November 1978.

[KS83] R.B. Kieburtz and A. Silberschatz. Access–Right Expressions. *ACM Transactions on Programming Languages and Systems*, 5(1):78–96, January 1983.

[KS91] H.F. Korth and A. Silberschatz. *Database System Concepts.* McGraw Hill, Inc., 2 edition, 1991.

[Kü95] W.E. Kühnhauser. A Paradigm for User–Defined Security Policies. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems.* IEEE Press, 1995.

[KZB+90] P.A. Karger, M.E. Zurko, D.W. Bonin, A.H. Mason, and C.E. Kahn. A VMM Security Kernel for the VAX Architecture. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 2–19, 1990.

[LABW92] B. Lampson, M. Abadi, M. Burrows, and T. Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[Lac91] S. Lacourte. Exceptions in Guide, an Object–Oriented Language for Distributed Applications. In *Proceedings of the European Conference on Object–Oriented Programming (ECOOP) 1991*, Genf, 1991.

[Lam71] B.W. Lampson. Protection. In *Proceedings of the Fifth Annual Princeton Conference on Information Science and Systems*, pages 437–443, 1971.

[Lam73] B.W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, 1973.

[Lau95] B. Lau. A Framework for Access Control Models. In J. H.P. Eloff and S.H. von Solms, editors, *Proceedings of the IFIP TC11 11th International Conference on Information Security*, pages 513–533, Cape Town, South Africa, May 1995. Chapman & Hall.

[LBMC94] C.E. Landwehr, A.R. Bull, J. P. McDermott, and W.S. Choi. A Taxonomy of Computer Program Security Flaws. *ACM Computing Surveys*, 26(3):211–254, Sept. 1994.

[Lev84] H.M. Levy. *Capability–Based Computer Systems*. Digital Press, 1984.

[LH89] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Lin76] T.A. Linden. Operating System Structures to Support Security an Reliable Software. *Computing Surveys*, 8(6):409–445, December 1976.

[Loe91] K. Loepere. *MACH3 Kernel Principles*. Technical Report, Open Software Foundation and Carnegie Mellon University, 1991.

[Lun89] T. Lunt. Access control policies for database systems. In C. Landwehr, editor, *Database Security II – Status and Prospects*, pages 41–52, North Holland, 1989.

[LZ75] B.H. Liskov and S.N. Zilles. Specification techniques for data abstractions. *IEEE Transactions on Software Engineering*, SE-1:7–19, March 1975.

[MA79] J.R. McGraw and G.R. Andrews. Access Control in Parallel Programs. *IEEE Transactions on Software Engineering*, SE-5(1):1–9, January 1979.

[Mar92] D. Marek. *Verteilung kryptographischer Schlüssel in einem konzeptionell strukturierten System*. Diplomarbeit, Fachbereich Informatik, Universität Oldenburg, Juli 1992.

[Mat93] M. Matsui. Linear Cryptanalysis Method for DES Cipher. In *Lecture Notes in Computer Science: Advances in Cryptology – EUROCRYPT '93*, pages 386–397. Springer–Verlag, 1993.

[Mey88] B. Meyer. *Object–oriented Software Construction*. Prentice–Hall International Ltd., 1988.

[Mos93] D. Mosberger. Memory Consistency Models. *Operating Systems Review*, 27(1):18–26, January 1993.

[MT86] S.J. Mullender and A.S. Tanenbaum. The Design of a Capability–Based Distributed Operating System. *The Computer Journal*, 29(4):289–299, 1986.

[MvRT+90] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer Magazine*, 23(5):44–53, May 1990.

[MW95] H. Mössenbö and N. Wirth. *The Programming Language Oberon-2.* Technical Report, ETH Zürich, March 1995.

[Neu91] B.C. Neuman. Protection and Security Issues for Future Systems. In A. Karshmer and J. Nehmer, editors, *Operating Systems of the 90s and Beyond, Proceedings of the International Workshop Dagstuhl Castle, Germany*, pages 184–201. LNCS 563, July 1991.

[NIS01] National Institute of Standards and Technology NIST. FIPS Standard 197. *Federal Register*, 66(235), 06.12.2001.

[NL91] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(8):52–60, August 1991.

[NS78] R.M. Needham and M.D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, 1978.

[OMG91] Object Management Group OMG. The Common Object Request Broker: Architecture and Specification. OMG Document 91.12.1, Revision 1.1, Object Management Group, December 1991.

[Ooi93] J.L. Ooi. *Access Control for an Object–Oriented Distributed Platform.* Master thesis, University of Dublin, Trinity College, Department of Computer Science, August 1993.

[OSF91] Open Software Foundation OSF. *OSF/Motif Programmer's Guide (Release 1.1).* Prentice–Hall, Inc., Englewood Cliffs, NJ, 1991.

[OSF92] Open Software Foundation OSF. *Introduction to OSF DCE.* Prentice–Hall, Inc., Englewood Cliffs, NJ, 1992.

[PE97] M. Pizka and C. Eckert. A Language–based Approach to Construct Structured and Efficient Object–based Distributed Systems. In *Proceedings of the 30th Hawaii International Conference on System Sciences*, Hawaii, USA, January 1997. IEEE.

[PKK+79] G.J. Popek, M. Kampe, C. Kline, A. Stoughton, M. Urban, and E.J. Walton. UCLA Secure Unix. In *AFIPS Conference Proceedings*, pages 355–364, 1979.

[Poh97] N. Pohlmann. *Firewall–Systeme: Sicherheit für Internet und Intranet.* International Thomson Publishing, 1997.

[Qui96] X. Quian. View–Based Access Control with High Assurance. In *Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy*, 1996.

[Rad96] R. Radermacher. *Eine Ausführungsumgebung mit integrierter Lastverteilung für verteilte und parallele Systeme*. Dissertation, Fakultät für Informatik, Technische Universität München, März 1996.

[Rei91] M. Reitenspieß. Verfügbarkeit – eine tragende Säule sicherer Systeme. In A. Pfitzmann and E. Raubold, editors, *VIS'91 – Verläßliche Informationssysteme*, pages 22–44, Darmstadt, März 1991.

[Rie98] T. Rieger. *Konzeption und Implementierung einer Krypto–Bibliothek in INSEL*. Softwareentwicklungsprojekt, Fakultät für Informatik, Technische Universität München, 1998.

[RSA78] R.L. Rivest, A. Shamir, and A. Adleman. A Method for obtaining Digital Signatures and Public Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[RSC92] J. Richardson, P. Schwarz, and L. Cabrera. CACL: Efficient Fine–Grained Protection for Objects. In *Proceedings of the OOPSLA'92 Conference*, pages 263–275, 1992.

[RW96] R. Radermacher and F. Weimer. INSEL Syntax–Bericht. Technischer Bericht TUM-I9617, SFB 342/08/96 A, Technische Universität München, März 1996.

[Sat89] M. Satyanarayanan. Integrating Security in a Large Distributed System. *ACM Transactions on Computer Systems*, 7(3):247–280, August 1989.

[SCFY96] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role–Based Access Control Models. *IEEE Computer*, 29(2):38–47, February 1996.

[Sch95] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 1995.

[SCW85] C. Schaffert, T. Cooper, and C. Wilpolt. Trellis object–based environment: Language reference manual. Technical Report DEC-TR-372, Eastern Research Lab, Digital Equipment Corp., Hudson, MA, November 1985.

[SEL+96] P.P. Spies, C. Eckert, M. Lange, D. Marek, R. Radermacher, F. Weimer, and H.-M. Windisch. Sprachkonzepte zur Konstruktion verteilter Systeme. Technischer Bericht TUM-I9618, SFB 342/09/96 A, Technische Universität München, März 1996.

[Sig97] SigG. Gesetz zur digitalen Signatur (Signaturgesetz – SigG). In *Artikel 3 des Informations– und Kommunikationsdienste–Gesetzes – IuKDG*, pages 1870–1872, Bundesgesetzblatt I, 22.07.1997.

[Sig01] SigG. Gesetz über Rahmenbedingungen für elektronische Signaturen (Signaturgesetz – SigG). In *Bundesgesetzblatt I*, page 876, 16.05.2001.

[SNS88] J.G. Steiner, B.C. Neuman, and J.I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *USENIX Winter Conference*, pages 191–202, February 1988.

[Spa88] E.H. Spafford. The Internet Worm Program: An Analysis. Purdue Technical Report CSD-TR-823, Purdue University, 1988.

[Spa89] E.H. Spafford. The Internet Worm: Crisis and Aftermath. *Communications of the ACM*, 32(6):678, June 1989.

[Spi84] P.P. Spies. Qualifizierte Zugriffsobjekte als Hilfsmittel zur Strukturierung von Systemen. Interner Bericht 11/84/1, Universität Bonn, Bonn, 1984.

[Spi85] P.P. Spies. Datenschutz und Datensicherung im Wandel der Informationstechnologien. In *1. GI-Fachtagung, Datenschutz und Datensicherung im Wandel der Informationstechnologien*, pages 1–25. Springer–Verlag, October 1985.

[Spi90] P.P. Spies. VERITOS-Projektbericht. Interner Bericht SA/90/3, Universität Oldenburg, Oldenburg, Dezember 1990.

[Spi92] P.P. Spies. Pallelverarbeitung mit INSEL. Technischer Bericht, Institut für Informatik, Technische Universität München, München, Mai 1992.

[SS75] J.H. Saltzer and M.D. Schroeder. Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, March 1975.

[SS95] M. Schaefer and G. Smith. Assured discretionary access control for trusted RDBMS. In *Proceedings of the Ninth IFIP WG 11.3 Working Conference on Database Security*, pages 275–289. Chapman & Hall, August 1995.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison–Wesley, Reading, MA, 2nd edition, 1991.

[Sun95] Sun. The Java Language Evironment: A White Paper. Technical report, Sun Microsystems Inc., http://java.sun.com/whitePaper/java-whitepaper-1.html, 1995.

[TMvR86] A.S. Tanenbaum, S.J. Mullender, and R. van Renesse. Using Sparse Capabilities in a Distributed Operating System. In *Proceedings of the 6th International Conference on Distributed Operating Systems*, pages 558–563, May 1986.

[TNT92] M.M. Theimer, D.A. Nichols, and D.B. Terry. Delegation Through Access Control Programs. In *Proceedings of the 12th International Conference on Distributed Systems*, pages 529–536. IEEE Computer Society Press, 1992.

[TW89] P. Terry and S. Wiseman. A "New" Security Policy Model. In *Proceedings of the 1989 IEEE Symposium on Research in Security and Privacy*, pages 215–228, 1989.

[VK83] V. Voydock and S.T. Kent. Security Mechanisms in High–Level Network Protocols. *Computing Surveys*, 15(2):135–171, 1983.

[Weg87] P. Wegner. Dimensions of Object–Based Language Design. In N. Meyrowit, editor, *OOPSLA'87 Conference Proceedings*, pages 168–182, Orlando, Florida, October 1987.

[Wei96] B. Weiser. Implementierung eines Terminal–Servers für INSEL–Systeme. Fortgeschrittenenpraktikum, Lehrstuhl für Systemarchitektur, Technische Universität München, Februar 1996.

[Wei97] F. Weimer. DAViT: Ein System zur interaktiven Ausführung und zur Visualisierung von INSEL–Programmen. Technischer Bericht TUM-I9721, SFB 342/15/97 A, Technische Universität München, April 1997.

[WG89] N. Wirth and J. Gutknecht. The Oberon System. *Software Practice and Experiences*, 19(9), Sept 1989.

[Wie93] M.J. Wiener. Efficient DES Key Search. In *Lecture Notes in Computer Science: Advances in Cryptology – CRYPTO '93*. Springer–Verlag, 1993.

[Wil88] J. Wilson. Views as the Security Objects in a Multilevel Secure Relational Database Management System. In *Proceedings of the 1988 IEEE Symposium on Research in Security and Privacy*, Oakland, California, April 1988.

[Win96] H.-M. Windisch. *Speicherverwaltung für konzeptionell strukturierte Systeme.* Dissertation, Fakultät für Informatik, Technische Universität München, Juli 1996.

[WL93] Th. Woo and S. Lam. Designing a Distributed Authorization Service. Technical Report 93-29, Department of Computer Science, University of Texas at Austin, 1993.

[Wop95] M. Wopkes. *Entwurf und Implementierung eines verteilten Authentifizerungsdienstes unter MACH 3.0.* Diplomarbeit, Institut für Informatik, Technische Universität München, März 1995.