Modellbasierter Entwurf spontaner Komponentensysteme

Christian Salzmann

Institut für Informatik
Technische Universität München

5. August 2002

Institut für Informatik
der Technischen Universität München

Modellbasierter Entwurf spontaner Komponentensysteme

Christian Salzmann

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. Donald A. Kossmann

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Manfred Broy
2. Univ.-Prof. Dr. Uwe Baumgarten

Die Dissertation wurde am 11. März 2002 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 15. Juli 2002 angenommen.

Kurzfassung

Gegenstand dieser Arbeit ist der systematische Entwurf *spontaner Komponentensysteme*, wie sie in den Bereichen Ad-hoc-Netzwerke, Mobile-Computing oder Wide-Area-Computing zum Einsatz kommen. Diese spezielle Klasse verteilter Komponentensysteme zeichnet sich durch einen hohen Grad an Dynamik bezüglich ihrer Komponenten und deren Verbindungsstrukturen aus. Insbesondere muss sich ein solches System permanent während der Laufzeit aus den vorhandenen Komponenten neu komponieren. Generell kann man vier Charakteristika spontaner Komponentensysteme isolieren: *dynamische Verbindungsstrukturen*, *Mobilität* der Komponenten, *Umgebungsprofile*, die die Vererbungsmechanismen einschränken und schließlich *spontane Interaktion* der Komponenten.

Diese vier Charakteristika bewirken in ihren verschiedenen Ausprägungen, dass bei spontanen Komponentensystemen eine invariante logische Architektur nicht auf *eine* technische Architektur, sondern vielmehr auf auf eine Menge von möglichen Realisierungen (so genannte Konfigurationen) abgebildet wird. Zwischen diesen Konfigurationen kann das System zur Laufzeit durch Umkonfiguration, zum Beispiel in Form von Komponentenmigration oder Auf- und Abbau von Kommunikationsverbindungen, wechseln und sich so gegebenenfalls an äußere Ereignisse, wie zum Beispiel dem Auftreten neuer Komponenten oder dem Wechsel der Umgebung, bis hin zum Ausfall von Systemteilen, selbstständig anpassen. Eine architekturelle Beschreibung der zu einer logischen Architektur gehörenden Konfigurationen sowie der jeweils bestehenden Abhängigkeiten ermöglicht die Auswahl der am besten geeigneten Konfiguration im Hinblick auf bestimmte Architekturkriterien, wie beispielsweise Stabilität oder Redundanz.

Bei dem in dieser Arbeit gewählten Ansatz handelt es sich um ein abstraktes, auf formalen Grundlagen basierendes Architekturmodell, welches verschiedene Plattformen und Middlewaretechnologien abdeckt. Hierbei wurden auf die Charakteristika spontaner Komponentensysteme besonderer Wert gelegt. Es wurden insbesondere Ergebnisse aus der Grundlagenforschung, darunter Elemente des FOCUS-Modells und Konzepte des Ambient-Kalküls, aufgegriffen und diese in einer praktikablen, auf aktuelle Anforderungen abgestimmten Form der Systemmodellierung spontaner Komponentensysteme eingesetzt. Ein wichtiger Aspekt ist hierbei die Spezifikation der Architektur solcher Systeme. Insbesondere die invariante Systemstruktur, die so genannte *logische Architektur* spontaner Komponentensysteme, wird eigenständig behandelt und modelliert. Diese wird anschließend auf eine Menge von *Konfigurationen* abgebildet, die den funktionalen Abhängigkeiten in Form von Komponentenbeziehungen Rechnung tragen, und eine mögliche Realisierung der in der logischen Architektur spezifizierten Funktionalität darstellen. Die hierbei gewonnene Information kann anschliessend vom System zur Laufzeit für die autonome Komposition der Systemteile verwendet werden. Durch die Abbildung des Modells kann der Ansatz unter gängigen Middleware-Plattformen für spontane Systeme, zum Beispiel JINI oder .NET, verwendet werden.

Zentrales Ergebnis der Arbeit ist zunächst ein auf formalen Grundlagen abgestütztes Architekturmodell, in dessen Rahmen die notwendigen Abstraktionen zum Entwurf spontaner Komponentensysteme definiert werden. Hierzu zählt insbesondere ein

Dienstbegriff sowie eine Definition von explizit modellierten und adressierbaren Umgebungen, in denen Komponenten und Dienste angesiedelt sind. Das Modell stützt sich auf ein formales Basismodell. Dadurch ist es zum einen möglich, die einzelnen Abstraktionen präzise zu fassen und zum anderen verschiedene Beschreibungsformen für Verhalten konsistent einzubinden. Mittels dieses Modells ist es möglich, die wechselnden Strukturen spontaner Systeme zu einer invarianten Dienstarchitektur in Bezug zu setzten und die einzelnen Konfigurationen nach Gütekriterien wie Stabilität und Effizienz zu bewerten. Zur maschinellen Auswertung von Spezifikationen wurde SADL, eine auf XML basierende Architektur-Beschreibungssprache, entwickelt, die auf Basis des entwickelten Modells im Gegensatz zu existierenden XML-Beschreibungssprachen (z.B. WSDL) eine Beurteilung der architektonischen Gütekriterien zur Laufzeit ermöglicht. Eine Implementierung und Erprobung der vorgestellten Modellierungstechniken in Form einer Modellierungsumgebung und einer Fallstudie aus dem Bereiche Mobile-Computing untermauern die hier vorgebrachten Konzepte.

Danksagung

Sich einem umfangreichen Thema über einen langen Zeitraum zu widmen, wie es bei einer Promotion notwendig ist, ist nur mit der Hilfe und Unterstützung von vielen Seiten möglich. Es ist letztendlich jedoch immer schwer, alle zu nennen, die direkt oder indirekt die Arbeit mit beeinflusst haben.

Zuerst möchte ich mich bei Prof. Dr. Manfred Broy für das Ermöglichen der Arbeit und die begleitende Betreuung bedanken. Dank gilt ebenso Prof. Dr. Uwe Baumgarten für die Übernahme der Zweitbegutachtung.

Bei meinen Kollegen Alexander Pretschner, Wolfgang Schwerin und Dr. Katharina Spies möchte ich mich für viele Diskussionen und Anregungen bedanken, bei Frank Marschall, Robert Sandner und Rupert Stützle für Korrekturen und Hinweise.

Dr. Max Breitling, Alexander Schmidt und Jan Philipps möchte ich für viele anregende Gespräche und Diskussionen danken.

Bei dem dem gesamten Lehrstuhl möchte ich mich für die gute Atmosphäre und die unkomplizierte Zusammenarbeit danken. Ein Umfeld, wie es am Lehrstuhl Broy vorzufinden ist, ist nicht selbstverständlich.

Schließlich möchte ich mich noch bei meiner Familie für die Geduld bedanken, die sie so manches mal beweisen haben.

INHALTSVERZEICHNIS

I Einführung **1**

1 Motivation und Übersicht **3**
1.1 Einleitung 3
1.2 Problembeschreibung 4
1.3 Ansatz 11
1.4 Ziele und Ergebnisse 13
1.5 Aufbau der Arbeit 15

2 Einordnung und verwandte Arbeiten **17**
2.1 Grundlagen 17
2.2 Architektur & Modellierung 28
2.3 Anwendungen 34
2.4 Zusammenfassung 36

3 Entwicklung verteilter Komponentensysteme **37**
3.1 Statische verteilte Komponentensysteme 37
3.1.1 Charakteristika verteilter Systeme 37
3.1.2 Entwicklungsprozesse 39
3.1.3 Standardplattformen 40
3.2 Spontane Komponentensysteme 44
3.2.1 Charakteristika spontaner Komponentensysteme 45
3.2.2 Standardplattformen 49
3.3 Diskussion 55

II Abstraktionen **59**

4 Modellbildung **61**
4.1 Motivation für partiell formalen Ansatz 62
4.2 Zielsetzung 63
4.3 Fallbeispiel: Ad-Hoc Systeme unter JINI 68
4.4 Aufbau und Charakteristika des Modells 72

5 Basismodell **75**

5.1 Grundlagen . . . 76
5.1.1 Ströme . . . 76
5.1.2 Komponenten und deren Komposition . . . 77
5.1.3 Dienste . . . 79
5.1.4 Komponenten-Netzwerke . . . 81
5.2 Spezifikation von Komponentennetzwerken . . . 82
5.2.1 Atomare Spezifikationen . . . 83
5.2.2 Kompositionsspezifikationen . . . 86
5.3 Zusammenfassung . . . 91

6 Engineeringmodell 93
6.1 Konkrete Entitäten . . . 96
6.1.1 Komponenten und deren Dienste . . . 96
6.1.2 Units . . . 102
6.1.3 Bindung . . . 103
6.1.4 Sandboxes und Hostings . . . 104
6.1.5 Konfigurationen und Einsatzumgebungen . . . 106
6.2 Abstrakte Entitäten . . . 108
6.2.1 Freie Dienste . . . 108
6.2.2 Dienstarchitekturen . . . 109
6.3 Entwurfsablauf . . . 110
6.3.1 Spezifikation . . . 110
6.3.2 Deployment Abbildung . . . 112
6.3.3 Auswahl und Realisierung . . . 112
6.4 Zusammenfassung . . . 113

7 Abbildung des Engineeringmodells auf das Basismodell 115
7.1 Motivation . . . 115
7.1.1 Expansionsregeln der Engineeringmodell-Spezifikationen . . . 116
7.2 Komponenten . . . 118
7.2.1 Beispiel: Abbildung einer Engineeringmodell Komponente . . . 124
7.3 Freie Dienste . . . 128
7.4 Bindung . . . 130
7.5 Sandboxes und Hosting . . . 131
7.6 Zusammenfassung . . . 132

III Architektur 135

8 Architekturbegriff spontaner Komponentensysteme 137
8.1 Aufbau einer Architekturabstraktion . . . 138
8.1.1 Logische Dienstebene . . . 139
8.1.2 Technische Konfigurationsebene . . . 141
8.1.3 Implementierungsebene . . . 144
8.2 Architektureigenschaften und Qualitätskriterien . . . 147
8.2.1 Redundanz . . . 148
8.2.2 Breite . . . 148

8.2.3 Flexibilität 149
8.2.4 Relevanz einer Komponente 150
8.2.5 Autarkie einer Komponente 150
8.2.6 Stabilität 151
8.2.7 Anwendungsbeispiel: Architektur berücksichtigender Trading-Service 153
8.3 Zusammenfassung 154

9 Spezifikation und Abbildung 155
9.1 Spezifikation mit SADL 155
9.1.1 Warum XML ? 156
9.1.2 SADL Spezifikation 157
9.1.3 Elemente der SADL 160
9.1.4 Verhaltensspezifikation 166
9.2 Vergleich WSDL – SADL 168
9.3 Abbildung auf Standard Plattformen 170
9.3.1 J2EE: Java & Jini 170
9.4 Zusammenfassung 173

IV Anwendung 175

10 Realisierung 177
10.1 Ablauf des realisierten Entwurfs 177
10.2 Aufbau der Modellierungsumgebung 179
10.2.1 Repository 180
10.2.2 Graphischer Client 181
10.2.3 SADL Client 182
10.2.4 Codegenerator 183
10.3 Funktionalitäten 186
10.3.1 Deployment Abbildung 186
10.3.2 Transaktionsverwaltung 189
10.4 Zusammenfassung 189

11 Fallstudie: Ad-Hoc Systeme 191
11.1 Spezifikation der logischen Architektur 193
11.1.1 Services 193
11.1.2 Service Architectures 200
11.2 Einsatzumgebung 201
11.2.1 Komponenten 202
11.2.2 Sandboxes 205
11.3 Abbildung auf technische Architekturen 206
11.3.1 Eigenschaften der logischen Architektur 206
11.4 Generierung von gekapselten Komponenten 209
11.5 Zusammenfassung 212

V Resümee **213**

12 Ergebnisse und Ausblick **215**
12.1 Zusammenfassung und Ergebnisse 215
12.2 Anwendungsgebiete 217
12.3 Ausblick 218

VI Anhänge **235**

A Zeichenerklärung **237**

B Graphische Notation **239**

C Graphische XML Notation **241**

D Glossar **243**

E SADL Definition **249**

Index **255**

Teil I

Einführung

KAPITEL

EINS

Motivation und Übersicht

1.1 Einleitung

Die rasant wachsende Netzwerkinfrastruktur und die damit verbundene nahezu allgegenwärtige Verfügbarkeit von netzbasierten Software-Systemen verlangen nach einem höheren Grad an Flexibilität und Dynamik der Systeme. Wir erwarten, dass zukünftig Systeme benötigt werden, die nicht nur die technischen Mechanismen verteilter Kommunikation realisieren, sondern auch ihre Adaption und Rekonfiguration während der Laufzeit autonom vornehmen können. Dies zeigt sich beispielsweise in Systemen, die in weiträumigen Netzen wie dem Internet angesiedelt sind und über ein weitaus höheres Maß an Adaptivität und Autonomie verfügen müssen als es bei Systemen in lokalen Netzwerken notwendig ist. Durch den Erfolg von Middleware-Plattformen wie CORBA [Gro98] oder RMI [Inc] existieren mittlerweile Standards, welche die Kommunikationsstrukturen zwischen Instanzen plattformübergreifend festlegen. Allerdings sind diese Standards für spontane, drahtlose Netzwerke sowie Wide-Area-Netze nicht immer ausreichend, da sie vornehmlich für statische, verteilte Systeme konzipiert und nicht für spontane Komponentensysteme ausgelegt sind. Wir charakterisieren die in dieser Arbeit im Mittelpunkt stehenden spontanen Komponentensysteme durch folgende Merkmale, die im Verlauf der Arbeit noch näher beschrieben werden:

Dynamik bezüglich Verbindungen und Komponenten: Sowohl die Kommunikationsverbindungen zwischen den Komponenten des Systems, als auch die Menge der im System integrierten Komponenten ist nicht statisch, sondern während der Laufzeit veränderbar.

Mobilität der Komponenten: Komponenten sind direkt adressierbaren Orten zugeordnet und können zwischen diesen Orten während der Laufzeit des Systems wechseln, ohne das System maßgeblich zu beeinflussen.

Umgebungsprofile: Die Umgebungen der Komponenten können diese in verschiedenster Weise beeinflussen. Beispielsweise bestimmt die Umgebung einer Komponente deren Kommunikationsrechte zu Komponenten anderer Umgebungen.

Spontane Interaktion: Die Interaktionen zwischen Komponenten und deren Komposition zu einem System finden teilweise zur Laufzeit statt und werden autonom vom System vorgenommen.

Neue Middleware-Plattformen wie z.B. JINI [Wal, Edw99] oder .NET [Cor01], aber auch der neue CORBA-Standard 3.0 [Wat99] beinhalten erste Konzepte dieser Art auf einer technischen Ebene. Diese Plattformen sorgen dafür, dass sich die jeweiligen Komponenten automatisch und transparent für Nutzer und Programmierer an Änderungen im Umfeld anpassen. Beispiel für solche teilweise unvorhersehbaren Änderungen sind das Vorhandensein neuer Komponenten und Funktionalitäten im Netz oder Störungen bis hin zum Ausfall eines Netzsegments.

Obwohl diese technischen Infrastrukturen mittlerweile technisch als ausgereift bezeichnet werden können, existieren keine befriedigenden Entwicklungsmethoden und Modellierungstechniken für spontane Komponentensysteme. Die genutzten Methoden und Modelle sind nur bedingt geeignet, da sie meist nur statische Strukturen beinhalten. Die notwendigen Abstraktionen, wie Architekturen und daraus resultierende Methoden, existieren bis dato nur ansatzweise bzw. behandeln lediglich einzelne Aspekte wie z.B. Code-Migration.

Das Ziel dieser Arbeit ist es, Techniken zu entwickeln, die, eingebunden in Methoden und Entwicklungsprozesse, einen systematischen Entwurf spontaner Komponentensysteme ermöglichen. Hierbei wird ein anwendungsgetriebener Ansatz verfolgt, der Top-Down und Bottom-Up Elemente verbindet: Erfahrungen aus der praktischen Realisierung solcher Systeme in Form von Projekten wurden systematisiert und führen zu notwendigen Abstraktionen und Modellen. Diese ermöglichen es, bewährte Architekturkonzepte zu beschreiben und zu verallgemeinern, um damit Systeme effektiver zu entwickeln.

1.2 Problembeschreibung

Heutige Plattformen für die Verteilung von Softwarekomponenten (z.B. CORBA) stellen vornehmlich eine Softwareschicht dar, die es erlaubt, von technologischen Unterschieden der einzelnen Betriebssysteme, wie z.B. Datenformaten oder Adressierungsprotokollen, zu abstrahieren. Dies ist die Grundlage für jegliche verteilte Komponentensysteme, die auf existierenden Systemen realisiert werden.

Die klassische Art der Entwicklung [HOE96] solcher statischer verteilter Systeme besteht in der Spezifikation der Komponenten als Träger der eigentlichen Systemfunktionalität, und dem anschließenden Definieren des sog. *Gluecodes*:

- Festlegen der Komponenten-Schnittstellen
- Definieren der Verbindungsstrukturen (Definition des Gluecodes)
- Erzeugen der Komponentengerüste und Implementierung der Funktionalität (Implementierung des Gluecodes).
- Starten und Binden der Komponenten

Der Gluecode selbst trägt keine eigentliche Funktionalität, sondern stellt die Verbindung zwischen den Komponenten her. Er repräsentiert also die “Verdrahtung” der Komponenten und legt somit fest, welche *Instanz* des Komponententyps *A* eine Methode *m* einer *Instanz* des Komponententyps *B* aufruft. Diese Verbindungsstruktur (Topologie) in Kombination mit der Menge der teilnehmenden Instanzen bezeichnet man

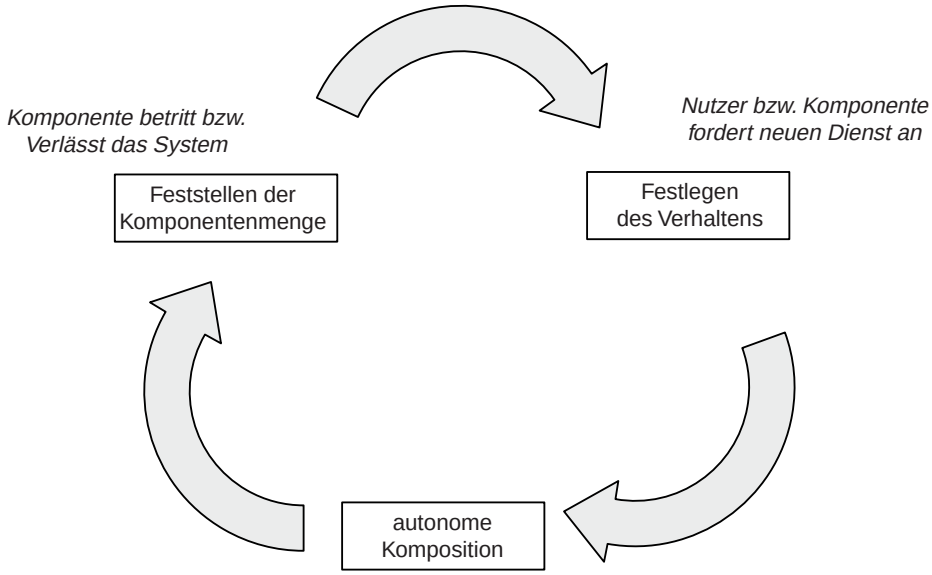


Abbildung 1.1: Der Ablaufzyklus spontaner Systeme, wobei insbesondere die autonome Komposition und deren Kompositionsregeln in dieser Arbeit behandelt werden.

auch als *Konfiguration* eines verteilten Komponentensystems. Neben dem Aufwand einer solchen Konfigurationsspezifikation, die darin besteht den Gluecode zu schreiben, also offensichtlich nichts mit der eigentlichen Systemfunktionalität zu tun hat, leiden die so entstandenen Systeme unter mangelnder Flexibilität und Fehlertoleranz. Stoppt eine der Instanzen, die im Gluecode verwendet wurden, so stoppt meist auch das gesamte System, unabhängig davon, ob eine zweite Instanz derselben Komponentenklasse zugegen ist.

Bezüglich der oben genannten Anwendungsgebiete, wie zum Beispiel mobile Systeme, ergibt sich ein offensichtliches Problem: die Verbindungsstruktur sowie die Menge der teilnehmenden Komponenten wird von vornherein festgelegt und ist über die Laufzeit hinaus vornehmlich fix. Solche Systeme benötigen aber gerade die Eigenschaft, ihre Konfiguration zur Laufzeit autonom und transparent anzupassen.

Neue Middleware-Plattformen, wie beispielsweise JINI oder .NET, konzentrieren sich auf das Konzept der automatisierten Erzeugung des Gluecodes. Dies bedeutet, dass eine Komponente nicht mehr konkret mit den benötigten Instanzen verdrahtet, sondern in das System “hineingeworfen” wird, worauf sich das System bezüglich der Verdrahtung autonom adaptiert. Dies wird dadurch erreicht, dass Komponenten nicht mehr via Instanzbezeichner alloziert werden, sondern via Komponententyp. Die hierbei verwendeten Protokolle (sog. Lookup-Protokolle) finden die benötigte Instanz, anschließend migriert diese zu der betreffenden Umgebung (sog. Lokation) oder wird durch entfernten Aufruf kontaktiert. Das System bzw. die Instanzen passen sich gegenseitig an und kooperieren. Dieser Zyklus (siehe Abbildung 1.1) wiederholt sich, wenn beispielsweise eine neue Komponente das Szenario betritt oder verlässt. Die Schritte sind im

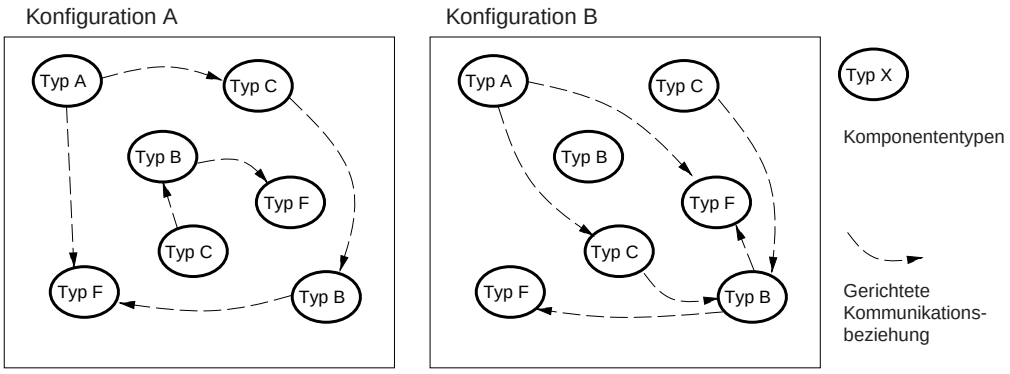


Abbildung 1.2: Zwei mögliche Konfigurationen eines spontanen Systems: Beziehungen zwischen Komponenteninstanzen.

einzelnen:

- Festlegung der Komponentenmenge – Änderung durch Betreten bzw. Verlassen des Systems durch eine Komponente.
- Festlegen der benötigten Funktionalität – Änderung durch neue Anforderungen des Nutzers oder anderer Dienste.
- Autonome Komposition des Systems mittels der vorhandenen Komponenten, unter Berücksichtigung der gegebenen Beschreibungen der einzelnen Systemteile (Komponenten und Dienste).

Man bezeichnet solche sich autonom adaptierenden Systeme auch als *spontane Komponentensysteme* (kurz spontane Systeme). Abbildung 1.2 zeigt zwei mögliche Konfigurationen eines spontanen Systems: Die Beziehungen zwischen den Komponententypen bleiben gleich, obwohl sich die Beziehungen zwischen den Instanzen ändern. Beispiele für solche spontane Middleware sind JINI [Wal], Universal Plug and Play (UPnP) [Con00], Millennium [Cor], .NET [Cor01], GLOBE [vSTKS98] und bedingt CORBA 3.0 [Wat99].

Für den dritten Punkt, die autonome Komposition des Systems, ist die Ausdrucksmächtigkeit des Modells besonders wichtig. Bei der Komposition können nur Kriterien berücksichtigt werden, die durch das Modell ausdrückbar sind und den Komponenten als Meta-Information beigefügt worden sind. Momentane Abstraktionen und Plattformen sind hierbei nur in der Lage, isoliert die Signaturen der Schnittstellen einzelner Komponenten zu beschreiben. Dies bedeutet insbesondere, dass sämtliche Informationen, die die Architektur, also das Zusammenspiel zwischen einer Menge von Komponenten, betreffen, nicht ausdrückbar sind und damit bei der Komposition nicht beachtet werden.

Ziel muss es also sein, eine Abstraktion zu entwickeln, die den Entwurf begleitet und mächtig genug ist, Architektureigenschaften auszudrücken, so dass diese zur Laufzeit bei der autonomen Komposition einfliessen und berücksichtigt werden können.

Charakteristika spontaner Komponentensysteme

Um den Entwurf auf spontane Komponentensysteme auszurichten, müssen zunächst deren Eigenschaften definiert werden. Die folgenden Charakteristika unterscheiden spontane Systeme von herkömmlichen, verteilten Komponentensystemen (siehe hierzu auch Kapitel 3.2 und 4.4):

Dynamisches Verhalten

In statischen verteilten Systemen sind Entwurf und Realisierung meist eng gekoppelt: Das Design bestimmt, welche Instanz welche Rolle innerhalb des Systems einnimmt.

Bei spontanen Systemen sind für einen funktionellen Entwurf mehrere Konfigurationen, d.h. Verbindungsstrukturen zwischen den Komponenteninstanzen möglich. Ereignisse aus der Umgebung können das System veranlassen, von einer möglichen Konfiguration zu einer anderen zu wechseln. Wie die Auswahl aus mehreren möglichen Konfigurationen vom System zu handhaben ist, muss schon in der Entwurfsphase des spontanen Systems berücksichtigt werden.

Mobilität

Verschiedene Lokationen (Wirtsplattformen, Orte) implizieren verschiedene Nebenbedingungen für das System und dessen Konfiguration. Diese können technische Eigenschaften wie Bandbreite beinhalten, aber auch Eigenschaften, die sich aus den Systemanforderungen ergeben, wie z.B. die physische Präsenz des Nutzers. Die Auswirkungen, die die jeweilige Lokation auf die Menge der möglichen Konfigurationen hat, müssen bei spontanen Systemen bereits im Entwurf beachtet werden.

Migration ist die Fähigkeit einer Komponente, ihre Lokation während der Laufzeit des Systems zu ändern. Man unterscheidet hier zwischen *Code-Migration* (sog. schwache Migration) und *Prozessmigration* (auch starke Migration) [FPV98]. Im ersten Fall wird nur der Ausführungscode ohne den momentanen Zustand transportiert (z.B. ein Postscriptdokument zum jeweiligen Postscriptdrucker). Im zweiten Fall wird die Komponente samt Zustand transportiert und nimmt an der Ziellokation ihren Ausführungsfaden dort wieder auf, wo sie ihn unterbrochen hat.

Umgebungsprofile

Verschiedene Lokationen bedeuten nicht nur Auswirkungen durch technische Aspekte wie Netzwerkbandbreite und dadurch verursachte Latenzzeiten. Das schwerwiegendere Problem, welches durch verschiedene Lokationen in Zusammenhang mit großen Netzwerken entsteht, ist das der verschiedenen Rechteräume. Dieses Problem kann z.B. durch Anwendung des Ambient-Kalküls [CG98] verdeutlicht werden: Jeder Prozess ist in eine mobile Umgebung – ein sog. *Ambient* – verpackt, welche die jeweiligen Rechte zur Migration definiert. Die gesamte Kommunikation ist im Ambient-Kalkül via Migration der Prozesse modelliert. Die Migrationsrechte wirken sich indirekt auch

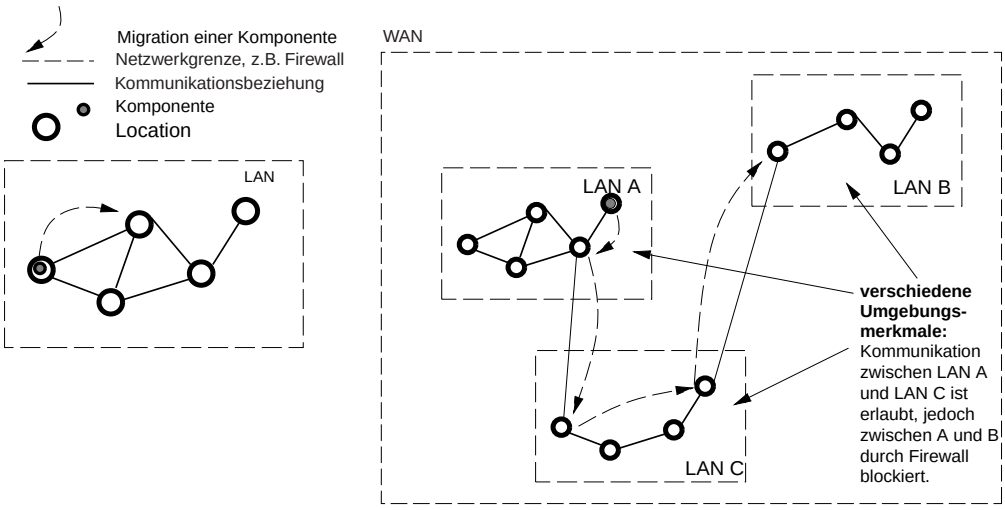


Abbildung 1.3: Verteilung eines Systems über ein sehr großes Netzwerk (WAN) bringt unterschiedlich Umgebungen (Rechte, Autorisierung, Zeit etc.) in unterschiedlichen Netzsegmenten mit sich.

auf die Kommunikation aus. Diese Abstraktion ist gut geeignet, um Kommunikationsrechte in verschiedenen Netzsegmenten von großen Netzen wie z.B. dem Internet zu abstrahieren. Firewalls erlauben bestimmten Prozessen (FTPclient, Browser etc.) bestimmte Kommunikationsarten (FTP, http, etc.), andere werden abgeblockt (Telnet, finger, etc.). Der Ambient-Kalkül wird im Abschnitt 2.1 kurz vorgestellt.

Spontane Interaktion

Komponenteninstanzen spontaner Systeme sind im Gegensatz zu statischen Systemen nicht fest verdrahtet, sondern müssen sich während der Laufzeit finden, auf gegenseitige Eignung prüfen und binden. Dieser Prozess, der im Gegensatz zu statischen verteilten Systemen autonom abläuft, ist anfällig für vielerlei Fehler: die Signaturen der Instanzen müssen kompatibel sein, die Interaktionsformen müssen aufeinander abbildbar sein sowie das Verhalten der Erwartung der Gegenseite entsprechen. Möglichst viele dieser Eigenschaften sollten durch geeignete Spezifikationsformen definiert sein, um zur Laufzeit die Korrektheit der Bindung auf allen Ebenen automatisiert prüfen zu können.

Diese Charakteristika spontaner Systeme müssen in einen systematischen Entwicklungsprozess einbezogen werden. Zumindest in der Spezifikation der Architektur müssen alle Aspekte, die Konfigurationen und deren Auswahl betreffen, beachtet werden. Populäre Entwicklungsmethoden und Modelle wie z.B. der Rational Unified Process (RUP) [Kru00] oder die Real Time Object Oriented Modeling Language (ROOM) [SGW94] beachten nur Teilaspekte. Wichtige Eigenschaften wie z.B. verändernde Lokationen und Rechte oder Transaktionen in sich ändernden Komponentenmengen werden komplett ausser Acht gelassen.

Aus den vier Charakteristika ergeben sich eine Reihe von Konsequenzen, die beim Entwickeln spontaner Komponentensysteme ein Rolle spielen:

Transaktionskontrolle bei sich verändernden Konfigurationen: Wenn ein spontanes System während einer Transaktion eine Umkonfiguration vornimmt, so kann dies zu Interferenzen führen. So kann z.B. das Wechseln von Lokationen und das damit verbundene neue Umgebungsprofil ein Fortführen verhindern oder eine benötigte Instanz nicht mehr vorhanden sein. Im Falle von verschachtelten Transaktionen wird die Abhängigkeit sogar noch ausgeweitet [KM90].

Fehlertoleranz und Fallbackverhalten: Herkömmliche statische Komponentensysteme lehnen sich meist an das Client/Server Prinzip an: Der Berechnungszustand wird zwischen Client- und Serverkomponente verteilt. Der Client enthält die Daten und Operationen, die lokal ausgeführt und gehalten werden können, wobei benötigte Aufrufe entfernt an den Server abgesetzt werden. Dadurch entstehen jedoch Probleme bezüglich der Verlässlichkeit des Systems bei Teilausfällen: Fällt eine Serverkomponente aus, so setzen auch alle damit assoziierten Clientkomponenten aus. Spontane Systeme können hier durch den hohen Grad an Flexibilität helfen [Sal99]. Wird der augenblickliche Zustand der Serverkomponente in die einspringende Ersatzkomponente übernommen, so kann der Fehlerfall sowohl autonom als auch transparent abgefangen werden. Konfigurationswechsel sollten bereits in die Entwicklung eines solchen Systems miteinbezogen oder berücksichtigt werden.

Reflektionsmechanismen: Unter Reflektionsmechanismen versteht man die Fähigkeit von Komponenten, so genannte Meta-Informationen über ihre Struktur und ihr Verhalten mittels so genannter Meta-Object-Protocols (MOP) auszutauschen. Beispiele für populäre MOPs sind die Introspection-Bibliotheken der Programmiersprache JAVA oder die Reflektionsmechanismen in LISP. Wird eine neue Komponente in ein System spontan eingebunden, so muss deren Funktionalität und Struktur gegebenenfalls an einige Systemkomponenten übermittelt werden. Hier bieten sich Reflektionsmechanismen, nach dem Vorbild des Verfahrens von DCOM an.

Sicherheit: Insbesondere bei der autonomen spontanen Interaktion von Komponenten ist die Frage der Sicherheit auf verschiedenen Ebenen zu erörtern. Zwar umfasst der Sicherheitsbegriff eine Vielzahl von Facetten, die nicht im Mittelpunkt dieser Arbeit stehen, jedoch kann durch Kontrollmechanismen die spontane Interaktion überwacht und damit sicherer gestaltet werden. Hierzu ist allerdings die Spezifikation des Verhaltens im Entwurf notwendig, was bei der Mehrzahl der Architekturbeschreibungstechniken nicht möglich ist.

Naming: Mit Naming bezeichnet man die Identifikation der Instanzen durch Bezeichner innerhalb verteilter Systeme. Einerseits müssen die Bezeichner eindeutig sein, andererseits bestimmten Richtlinien des Systems bezüglich des Bezeichneraufbaus für eventuelle Reflektionsmechanismen folgen[1]. Handelt es sich um ein spontanes System mit einer offenen Menge von Instanzen, die eventuell migrieren können, so ergeben sich Probleme hinsichtlich der Eindeutigkeit, als auch der Adressierung – wenn eine Komponente auf Lokation L_1 adressiert wird, muss sie sich dort nicht mehr notwendigerweise aufhalten. Hier existieren Vorarbeiten durch den NomadicPICT-Kalkül

[1] So muss z.B. bei Java-Beans jedes Attribut X durch jeweilige getX und setX Methoden lesbar und schreibbar sein.

[SW98], der die ortsunabhängige Adressierung für mobile Komponenten ermöglicht. Jedoch können in Wide-Area-Netzen zusätzliche Probleme durch die verschiedenen Umgebungsprofile entstehen [vSHHT98], da nicht alle Umgebungen erreichbar sind.

Anwendungen spontaner Komponentensysteme

Anwendungen solcher Systeme finden sich in vielen Gebieten der aktuellen Softwarelandschaft. Einige Beispiele, die im Laufe der Arbeit noch weiter vertieft werden, sind (siehe auch Abbildung 1.4):

- Ad-hoc-Netzwerke beruhen auf einem spontanen, meist drahtlosen Übertragungsmedium. Betritt eine Komponente das abgedeckte Terrain, so müssen die Verbindungsstruktur, aber auch die Software-Struktur auf Applikationsebene dynamisch angepasst werden.
- Mobile Computing bezeichnet zweierlei Typen von Systemen: mobile Geräte, welche sich bei einem Ortswechsel an die neue Umgebung anpassen müssen, sowie mobilen Code. Beide beruhen auf demselben Prinzip: Ein laufendes System muss sich dynamisch an die sich verändernden Umgebungen anpassen.
- Wide-Area-Anwendungen (Wide-Area-Applications) sind verteilte Systeme, welche auf Wide-Area-Networks, z.B. nach ATM- oder X.25-Standard, verteilt sind [Car99a]. Beispiele hierfür sind Internetanwendungen oder auch Systeme, die im Intranet von internationalen, großen Unternehmen verteilt sind. Diese Systeme zeichnen sich durch mehrere Zeit- und Rechte-Zonen aus sowie meist durch ständige Verfügbarkeit. Grundlegende Vorarbeiten hierzu sind in [BC97, Car97] zu finden.

Prinzipiell unterscheiden wir in dieser Arbeit zwischen drei Anwendungsarten von spontanen Komponentensystemen: *dynamische* (oder ad-hoc) Systeme, *mobile* Systeme und schließlich *Wide-Area*-Systeme. Die Vereinigung dieser Systemarten steht im Fokus dieser Arbeit und wird im folgenden als *spontane Komponentensysteme* bezeichnet. Man beachte hierbei, dass die jeweiligen Systemarten keinesfalls disjunkt sind, sondern sich in einigen Punkten überschneiden können (siehe Abbildung 1.4).

Systeme, die in der Lage sind, sich selbst zur Laufzeit dynamisch und transparent an die Umgebung durch Umkonfiguration der eigenen Topologie und Komponentenmenge anzupassen, bezeichnen wir als *dynamische* oder *Ad-Hoc Systeme*.

Eine weitere Anwendung spontaner Systeme findet man bei *mobilen Systemen*. Diese beinhalten zusätzlich zu den Eigenschaften dynamischer Systeme das Konzept von örtlicher Zuordnung. Jede Komponente ist einer Lokation (einer virtuellen oder auch physischen Plattform) zugeordnet. Zwischen diesen Wirtsrechnern (sog. *Lokationen*) können die Komponenten migrieren, d.h. ihren Ort zur Laufzeit wechseln. Die Konfiguration eines mobilen Systems beinhaltet demnach nicht nur die Zuordnung von Instanzen zu Instanzen, sondern zusätzlich auch die von Instanzen nach Lokationen. Beide sind gemäß des Grundkonzeptes von spontanen Systemen über die Laufzeit änderbar.

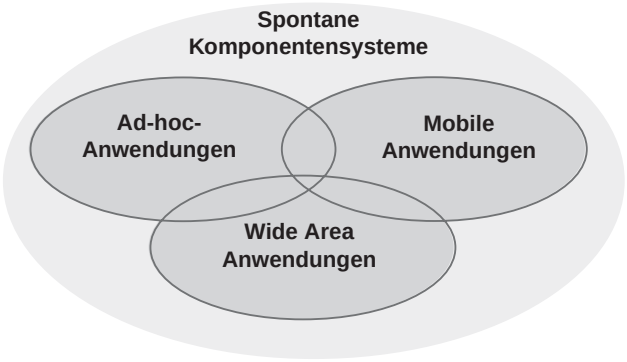


Abbildung 1.4: Klassifikation der Anwendungsarten

Die dritte in dieser Arbeit behandelte Klasse spontaner Systemen sind sog. *Wide-Area-Systeme*. Diese zeichnen sich zusätzlich zu den oben genannten Merkmalen dadurch aus, dass sie über sehr große Netzwerke (Wide-Area-Networks) verteilt sind und somit in unterschiedlichen Netzsegmenten verschiedene *Umgebungsmerkmale* wie Zeit und Kommunikationsrechte (z.B. durch Firewalls bewirkt) gelten (siehe Abbildung 1.3). Diese können sich sehr stark auf die jeweilige Menge an Konfigurationen auswirken, da eine Menge an Konfigurationen technisch möglich wäre, jedoch nur eine Untermenge davon zulässig ist. Ein Beispiel für derartige Systeme sind jegliche Arten von Mobilen Internetanwendungen, die auf GSM oder UMTS basieren und damit mit temporären Netzausfällen rechnen müssen oder Roaming[2] nutzen.

1.3 Ansatz

Ziel dieser Arbeit ist es, Techniken einer systematischen Entwurfsmethode für spontane Komponentensysteme zu entwickeln. Hierbei stehen vor allem die strukturelle Beschreibung und die Architektur im Vordergrund. Grundlage für eine strukturelle Beschreibung ist jedoch eine klare *Modellbildung*, durch die die oben beschriebenen Charakteristika hinreichend ausgedrückt werden können. Die meisten der existierenden Modelle sind hierfür nicht ausreichend geeignet.

In dieser Arbeit stellen wir ein Modell für die Entwicklung spontaner Komponentensysteme vor, das aus einem *formalen Basismodell* und einem intuitiven *Engineeringmodell* besteht. Wir verfolgen hierbei zwei Ebenen der Abstraktion für spontane Systeme gemäß der funktionellen Spezifikation (*Welche Funktionalität erfüllt das System?*) und der Instanziierung (*Welche Instanz übernimmt letztendlich welche Rolle?*). Die Sichtweisen wechseln während des Entwurfsprozesses und bewegen sich von der abstrakteren, funktionellen Dienst-Architektur über die konkrete Konfiguration bis hin zur Implementierung (siehe Abbildungen 1.5 und 1.6).

[2] Mit *Roaming* bezeichnet man das Nutzen eines Mobiltelefons im Netz eines Partnerproviders. Benutzt man ein Mobiltelefon z.B. im Ausland, so kann man die Grundfunktionen wie Telefonie, SMS etc. weiterhin nutzen. Einige Funktionen, die persönliche Informationen benötigen, wie z.B. die Kennung automatisch an die Mailbox zu senden, werden jedoch möglicherweise nicht unterstützt – ein typisches Verhalten für die oben beschriebenen Wide-Area-Systeme.

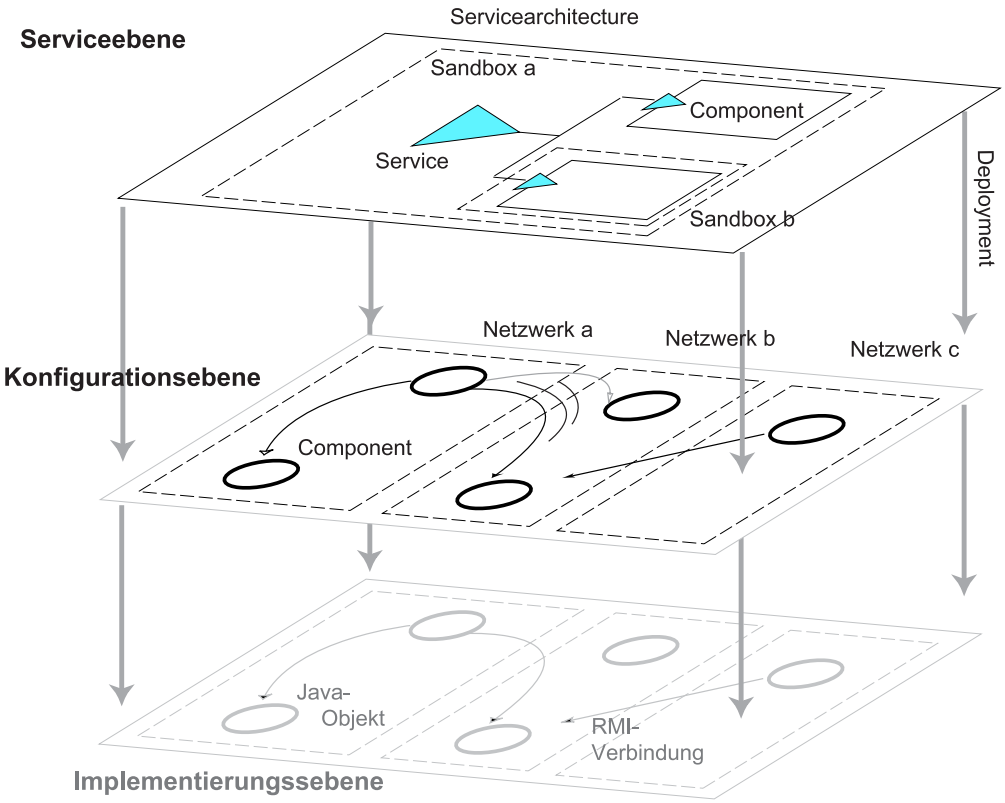


Abbildung 1.5: Problembeschreibung: Statische Funktionsbeschreibung muss auf eine dynamische Konfiguration abgebildet werden, diese wiederum auf eine Implementierung.

Im Detail handelt es sich um die in Abbildung 1.5 gezeigten drei Ebenen:

Abstrakte Dienstebene: Ein Dienst ist eine abstrakte Entität, die ein gewisses Verhalten bezüglich einer Schnittstelle definiert. Hierbei gilt, dass der Dienst nur als Blackbox-Verhalten ohne jegliche Struktureigenschaften definiert ist. Beispiele hierfür sind Jini-Dienste oder allgemein konfigurationsunabhängige Softwaremodule.

Dynamische Konfigurationsebene: Instanzenrollen repräsentieren die statischen Beziehungen innerhalb der Dienste. Abbildung 1.2 zeigt zwei Konfigurationen eines spontanen Systems. Obwohl die Instanzen sich verändern, bleiben die Strukturen zwischen den Typen (also die Erzeuger-/Verbraucherbeziehung) statisch. Diese Beziehungsstruktur wird modelliert, unabhängig von der eigentlichen Bestückung (Deployment) durch Objektinstanzen.

Implementierungsebene: Objektinstanzen bestücken schließlich die Instanzenrollen. Für eine Instanzenrolle kommt im allgemeinen eine Menge von Instanzen in Frage, wobei das Deployment noch durch verschiedene Nebenbedingungen wie z.B. Rechte bzgl. der Lokationen etc. eingeschränkt wird. Die Implementierung ist schließlich abhängig von den verwendeten Technologien und Plattfor-

men, wie zum Beispiel der Implementierungssprache oder Middleware. Da sich die Implementierungsebene von Standard zu Standard stark unterscheidet, steht sie nicht im Zentrum dieser Arbeit. Vielmehr wird die Abbildung auf eine Implementierung, in Java bzw. Jini, exemplarisch aufgezeigt und auf eine andere Implementierung in .NET skizziert.

Auf der statischen, abstrakten Dienstebene kann eine Spezifikation der Dienste analog zu einer Spezifikation statischer, verteilter Systeme entwickelt werden. Die entscheidende Frage, welche Konfigurationen *technisch* möglich sind und welche in der momentanen Umgebung *realisierbar* sind, wird durch die Abbildung auf die technische Ebene gelöst, die eine Menge von möglichen Konfigurationen ergibt. Diese Abbildung beinhaltet zum einen die äußeren Rahmenbedingungen, wie vorhandene Softwarekomponenten und Netzwerkstruktur mit den damit verbundenen Rechten, als auch die zu entwickelnden Systemteile.

Das verwendete *Basismodell* baut auf FOCUS [BS00] auf, einem mathematisch fundierten Systemmodell, welches Nachrichtenaustausch über Kanäle als Interaktionsformen benutzt.

Auf dieser Basis wird ein *Engineeringmodell* definiert, das die charakteristischen Eigenschaften spontaner Komponentensysteme, die im Basismodell nur implizit enthalten sind, explizit ausdrückt. Auf Basis dieses anwendungsnahen Architekturmodells werden Strukturen, Qualitätskriterien und Muster definiert, die sich bei spontanen Systemen anbieten oder bewährt haben [CPV97]. Wie in [SS99] gezeigt wurde, sind klassische Strukturen wie z.B. Client /Server Beziehungen hier nur bedingt geeignet.

1.4 Ziele und Ergebnisse

In der vorliegenden Arbeit wurden Techniken zum modellbasierten Entwurf spontaner Komponentensysteme entwickelt. Der Ablauf des Entwurfs wird in Abbildung 1.6 zusammengefasst.

Hierzu wird zunächst eine Spezifikation des spontanen Systems auf Basis des entwickelten Architekturmodells vorgenommen. In dieser Spezifikation werden Dienste, Komponenten und Sandboxes eingesetzt, um *dynamische Konfigurationen*, *Mobilität* und *Rechtekonzept*, die Charakteristika spontaner Systeme, zu modellieren.

Anschließend wird die Spezifikation zusammen mit Blackboxspezifikationen von existierenden Komponenten (z.B. Legacykomponenten), sowie der Spezifikation der existierenden Netzwerkumgebung (*Einsatzumgebung*) auf die Menge der möglichen Konfigurationen abgebildet (*Deployment*).

Eine gültige Konfiguration wird automatisiert auf Basis von Qualitätskriterien (z.B. Stabilität, Sicherheit etc.) ausgewählt und auf eine Implementierungsplattform (z.B. JINI) in Form von Komponentenskeletten abgebildet.

Eine wesentliche Zielsetzung für die Konzeption und Entwicklung der hier vorgestellten Verfahren zum modellbasierten Entwurf spontaner Komponentensysteme bestand

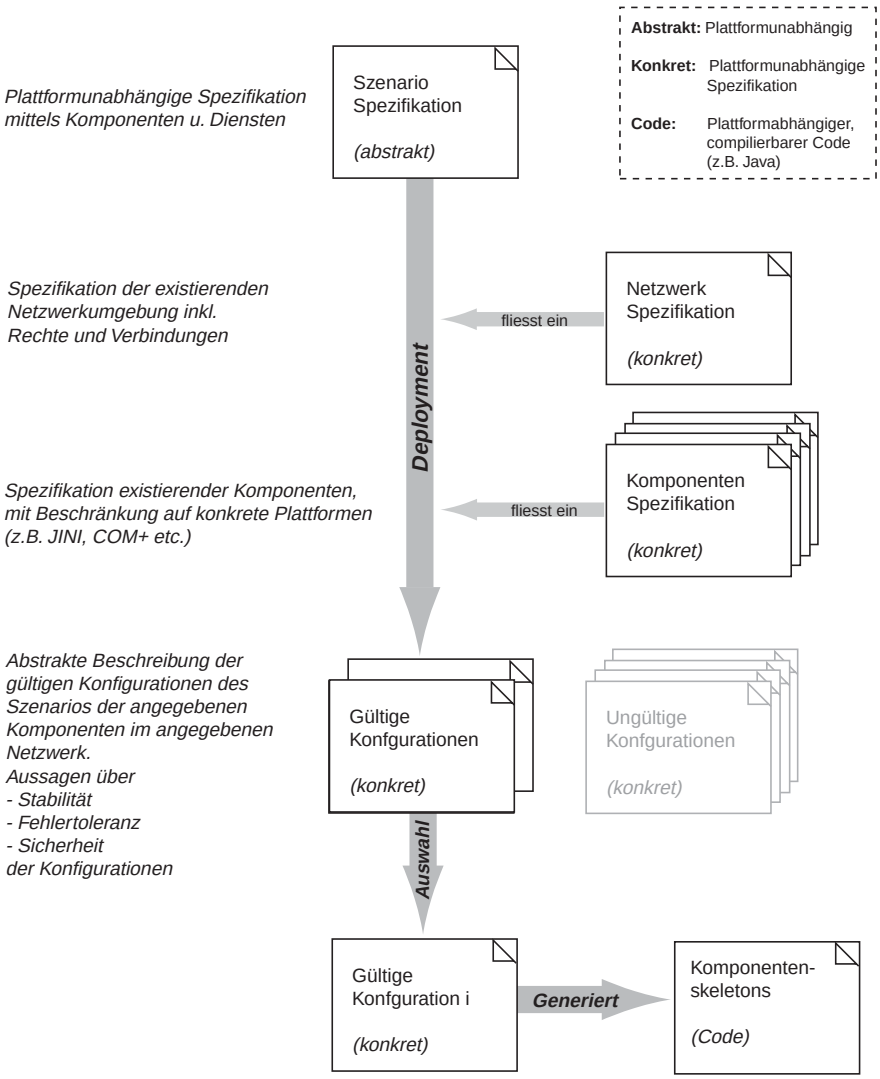


Abbildung 1.6: Ablauf und Ergebnisse der Entwicklung spontaner Komponentensysteme

darin, den gewinnbringenden Einsatz von Ergebnissen der Grundlagenforschung aufzuzeigen. Hierbei wurde besonderer Wert darauf gelegt, die verwendeten Ansätze so aufzuarbeiten, dass sie für Problemstellungen aus der Sicht der Anwendung sowohl geeignet als auch relevant sind.

Zusammenfassend sind die wichtigsten Ergebnisse der Arbeit:

- Es wird ein formal basiertes Architekturmodell entwickelt, das speziell auf die Charakteristika spontaner Komponentensysteme zugeschnitten ist und diese abstrahiert. Im Rahmen des Architekturmodells wurden die folgenden Konzepte definiert und auf eine formale Basis gestellt:
  - Ein **Dienstbegriff**, der Dienste als strukturlose Beschreibung von Funktionalität definiert.

  - Explizite **Umgebungsprofile**, wie sie bei Wide-Area-Computing auftreten, werden in den Entwurf integriert und damit deren Auswirkungen auf die Systemstruktur einbezogen.
  - Es wird eine klare Trennung zwischen invarianter **logischer Architektur** und wechselnden **technischen Architekturen** vorgenommen.

  Das Architekturmodell basiert auf einem formalen Basismodell, das zum einen die präzise Definition der Entitäten und zum anderen die konsistente Einbindung verschiedener Verhaltensbeschreibungen der Dienste ermöglicht.

- Durch eine Entwurfstechnik ist es möglich, die **invarianten Systemstrukturen** (Dienstarchitekturen) spontaner Systeme zu modellieren und diese automatisiert auf die Menge der diese Spezifikation **realisierenden Konfigurationen** abzubilden.

- **Qualitätskriterien und Muster** für die Architektur spontaner Komponentensysteme können auf Basis des Modells definiert werden und von einem System autonom ausgewertet und quantifiziert werden. Hierfür werden einige Beispiele gegeben.

- Zum Modell wurde eine XML-basierte **Beschreibungssprache** entwickelt, die es ermöglicht, Systeme basierend auf existierenden Plattformen (JINI, .NET etc.) auf Basis des Modells und der hier präsentierten Techniken zu spezifizieren und diese Beschreibungen während der Laufzeit zur Komposition auszuwerten, wobei die Architekturkriterien einbezogen werden.

- Die entwickelten Abstraktionen und Techniken wurden in Form von **Entwicklungswerkzeugen** implementiert und getestet. Hierbei wurde als Zielplattform für Ad-Hoc Systeme Java bzw. JINI gewählt.

- Es werden Ergebnisse und Verfahren aus der Grundlagenforschung dermaßen transferiert und verwendet, dass sie praxisrelevant mit den aktuellen Anwendungstechniken verwendbar sind.

1.5 Aufbau der Arbeit

Die Arbeit gliedert sich in fünf Teile: *Einführung*, *Abstraktionen*, *Architektur*, *Anwendung* und *Resümee*. Hierbei ist der Aufbau dermaßen gehalten, dass der Leser einzelne Kapitel oder Teile, die für ihn bereits bekannt sind, überspringen kann. Die wichtigsten Begriffe werden bei der jeweiligen Verwendung noch einmal kurz erklärt oder können im Glossar nachgeschlagen werden.

Leser, die mit der Thematik spontaner Komponentensysteme und deren Modellierung vertraut sind, können beispielsweise den Einführungsteil überspringen und den Schwerpunkt auf die Teile Abstraktionen und Architektur legen.

In der *Einführung* wird neben der Motivation der Arbeit eine Einordnung in verwandte Gebiete und eine Abgrenzung zu anderen Ansätzen vorgenommen. Hier sind vor allem Grundlagenarbeiten wie der π-und der Ambient-Kalkül [Car98, Car99b], der als

Grundlage für das Rechtekonzept bei Wide-Area-Anwendungen dient, zu nennen. Darüberhinaus wird eine Einführung in die wichtigsten Begriffe im Bereich Architektur und Modellierung verteilter Komponentensysteme gegeben, wie zum Beispiel Interface Description Languages, Architekturbeschreibungen oder auch dynamisch änderbare Architekturen. Schließlich werden die wichtigsten Anwendungsfelder spontaner Komponentensysteme, wie zum Beispiel *Mobile Computing* oder *Ubiquitous Computing*, kurz vorgestellt. Der Teil schließt mit einer knappen Einführung in die statischen und dynamische Middleware-Plattformen verteilter Systeme, als da sind CORBA, COM und RMI für die statischen und JINI, UPnP und .NET für die dynamischen Middleware-Plattformen.

Im zweiten Teil – *Abstraktionen* – wird das Systemmodell dieser Arbeit definiert. Hierzu wird zunächst das grundlegende Basismodell, welches in dieser Arbeit zur Anwendung kommt, beschrieben. Anschließend wird das Engineeringmodell beschrieben, welches auf die Charakteristika spontaner Systeme fokussiert. Im letzten Kapitel dieses Teils wird schließlich die Abbildung des Engineeringmodells auf das Basismodell definiert.

Der dritte Teil befasst sich mit der *Architektur* spontaner Komponentensysteme. Hier wird auf Basis des definierten Modells der Architekturbegriff für solche Systeme definiert und darauf aufbauend Qualitätskriterien für spontane Systeme und deren Konfigurationen. Besonderer Wert wird hierbei auf Begriffe wie Stabilität und Sicherheit gelegt. Anschließend werden Strukturen und Stile, die die Qualität steigern, also z.B. hohe Stabilität bieten, in Form von Mustern und Metriken vorgestellt. Aufbauend auf dem im zweiten Teil beschriebenen Engineeringmodell folgt eine knappe, als Beispiel dienende Definitionssprache für die Architektur spontaner Systeme (SADL) und deren Abbildung des Modells auf eine spontane Middleware-Plattform (JINI bzw. Java).

Im vierten Teil wird die praktische *Anwendung* der erzielten Ergebnisse in Form einer prototypischen Implementierung und einer *Fallstudie* gezeigt. Die Implementierung stellt eine Werkzeugunterstützung dar, die es mit der auf dem definierten Modell basierenden Definitionssprache SADL erlaubt, spontane Systeme zu spezifizieren, Konfigurationen zu berechnen und diese auf JINI und .NET in Form von Komponentenskeletten abzubilden. In diesem Abschnitt werden insbesondere auch die gewählten Realisierungen verschiedener Teile des Engineeringmodells erläutert, wie beispielsweise die Deploymentabbildung oder die Transaktionskontrolle. Im ersten hiermit spezifizierten Fallbeispiel wird das in Kapitel vier erwähnte JINI Mobile-Office Szenario modelliert.

Im letzten Teil wird ein *Resumee* gezogen. Hierbei werden die wichtigsten Ergebnisse noch einmal aufgegriffen und deren Vorteile für verschiedenen Anwendungsgebiete diskutiert. Ein Ausblick, der noch offene Punkte und Weiterentwicklungen anspricht, schließt den Abschnitt.

Im Anhang sind die Schemadefinition der Definitionssprache SADL aufgeführt sowie ein Glossar der wichtigsten verwendeten Begriffe.

KAPITEL

ZWEI

Einordnung und verwandte Arbeiten

Spontane Komponentensysteme kombinieren eine hohe Zahl an Eigenschaften, die teilweise aus klassischen Disziplinen der Informatik bekannt sind, teilweise neu sind oder erst in letzter Zeit entwickelt wurden. Beispiel für klassische Disziplinen sind verteilte Systeme und deren Eigenschaften wie Nebenläufigkeit. Beispiele für neue Entwicklungen sind der Ambient-Kalkül oder die Verwebung von Internetdiensten mit verteilten Systemen.

In diesem Kapitel wird ein Überblick über Gebiete gegeben, die der vorliegenden Arbeit als Basis dienen und in deren Umfeld die Ergebnisse anzusiedeln sind. Sie gliedern sich grob in die folgenden drei Gebiete:

Grundlagen: Zunächst wird in den Grundlagen der Ambient-Kalkül und dessen Basis, der π-Kalkül vorgestellt. Anschließend werden Konzepte aus dem Konfigurationsmanagement vorgestellt. Es wird kurz auf auf Reflektionsmechanismen eingegangen, die mittlerweile auch in populären Sprachen wie Java Einzug gehalten haben.

Architektur: Neben den Prozesskalkülen sind Elemente aus Architektur und Modellierung mit den gebräuchlichen Abstraktionen, u.a. Komponenten und Konnektoren, sowie deren Beschreibungstechniken in die Arbeit eingeflossen. Diese werden kurz vorgestellt.

Anwendungsgebiete: Schließlich werden die Anwendungsgebiete beschrieben, für die die Ergebnisse zugeschnitten sind. Darauf folgt ein Abschnitt, der die wichtigsten verwandte Arbeiten diskutiert.

2.1 Grundlagen

Wozu wird ein einheitliches Modell bei der Entwicklung von Software, in diesem Fall für die systematische Entwicklung von spontanen Komponentensystemen, benötigt?

Zunächst besteht der Sinn und Zweck eines solchen Modells in der Schaffung einer *einheitlichen* und *eindeutigen* Basis für die Beschreibung und Überprüfung des Systems. Es sollte daher ein Modell gewählt werden, das die wesentlichen Charakteristika des speziellen Systems hinreichend ausdrücken kann. Bei verteilten Systemen ist dies z.B. Nebenläufigkeit und evtl. auch Heterogenität, bei spontanen Systemen sind dies weitere (siehe unten).

Das Modell muss es ermöglichen, eine *Semantik der Beschreibungstechniken* zu definieren. Beschreibungstechniken sind in der systematischen Entwicklung mittlerer und großer Systeme unumgänglich[1]. Sie ermöglichen es, das System abhängig von verschiedenen Stufen im Entwicklungsprozess aus verschiedenen Perspektiven zu betrachten und somit einzelne Merkmale, wie z.B. Verteilung, Kommunikationsabläufe etc., für den Entwickler transparent zu verdeutlichen. Damit diese Beschreibungstechniken konsistent und korrekt eingesetzt werden können, muss ihre Bedeutung, also die Semantik, eindeutig definiert werden. Hierzu wird wiederum das Modell herangezogen.

Es bietet sich daher an, ein formales Modell als so genanntes *Basismodell* heranzuziehen. Mit Hilfe dieses Basismodells sollen die einzelnen Elemente semantisch fundiert und die Begriffe und Beziehungen eindeutig definiert werden. Ein Beispiel für ein Basismodell ist CSP, das als Grundlage für die Architecture Description Language WRIGHT benutzt wird, um Begriffe und Abläufe eindeutig zu definieren [All97].

π- und Ambient-Kalkül

Eine wichtige Basis für diese Arbeit ist der Ambient-Kalkül, ein von Luca Cardelli entwickelter Ableger des π-Kalküls, der grundlegende Eigenschaften von Wide-Area-Systemen verdeutlicht. In einer Diskussion des Ambient-Kalküls sollen hier die Grundeigenschaften und deren Auswirkungen auf diese Arbeit vorgestellt werden, aber auch die Einschränkungen, die dazu führen, dass der Ambient-Kalkül nicht analog in die Software-Entwicklung übertragen werden kann. Zunächst jedoch soll der π-Kalkül vorgestellt werden, da er die Grundlage des Ambient-Kalküls ist.

Der π-Kalkül [MPW92, Mil91] ist eine idealisierte Programmiersprache, die auf der Kommunikation zwischen nebenläufigen Prozessen basiert. Er ist eine Konsequenz der Communicating Concurrent Sequences (CCS) [Mil89]. Die Kommunikation im π-Kalkül findet über bezeichnete Kanäle statt. Ein im π-Kalkül spezifiziertes System besteht aus einer Menge von nebenläufigen *Prozessen*, die über ungerichtete *Kanäle* asynchron kommunizieren. Die folgende Grammatik definiert den ursprünglichen Kalkül:

P, Q	$::=$	0	nil
		$P \mid Q$	Parallele Komposition von P und Q
		$\bar{c}v$	Ausgabe v auf Kanal c
		$cw.P$	Eingabe auf Kanal c
		new c **in** P	Neuer Kanalbezeichner

[1] Dies ist bei kleinen, prototypischen Systemen anders – hier reicht eine informelle Beschreibung verbunden mit mehreren Wegwerfprototypen oft aus. Daher existieren bis dato auch für spontane Systeme, die sich noch in der Erprobungsphase befinden, keine adäquaten Beschreibungstechniken.

Man beachte hierbei, dass der **new** Operator bezüglich des neuen Bezeichners c soweit wie möglich nach rechts bindet. **new** c **in** $P \mid Q$ sollte also als **new** c **in** $(P \mid Q)$ gelesen werden. Die operationale Semantik, wird durch die folgenden Reduktionsregeln angegeben:

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}l}
\bar{x}y.P \mid x(z).Q & \rightarrow & P \mid [y/z]Q & & & & \text{Komunikation} \\
P \mid R & \rightarrow & Q \mid R & \textit{if } P & \rightarrow & Q & \text{Reduktion unter } \mid \\
(\nu x)P & \rightarrow & (\nu x)Q & \textit{if } P & \rightarrow & Q & \text{Reduktion unter } \nu \\
P & \rightarrow & Q & \textit{if } P \equiv P' & \rightarrow & Q' \equiv Q & \text{strukturelle Kongruenz}
\end{array}$$

Der π-Kalkül wird des Weiteren durch Umbenennungsregeln sowie Konsistenzeigenschaften definiert, die hier jedoch aufgrund des begrenzten Umfangs nicht aufgeführt werden sollen. Vielmehr soll dem Leser ein Eindruck vermittelt werden, für detaillierte Informationen sei auf [MPW92] verwiesen.

Da im π-Kalkül parallel ausführende Prozesse via ungerichtete Kanäle Nachrichten austauschen, können Ein- und Ausgabe auf dem selben Kanal stattfinden. Im folgenden Beispiel wird eine Nachricht a über einen Kanal x gesendet:

$$\bar{x}a \mid xu.\bar{y}u \quad \longrightarrow \quad a/u(\bar{y}u) = \bar{y}a$$

Gelesen von links nach rechts ergibt sich, dass auf dem Kanal x das Datum a gesendet wird, wobei parallel dazu ein Prozess auf diesem Kanal x liest und das Datum auf dem Kanal y unter der Variablen u ausgibt. Wird dieser Ausdruck gemäß den oben definierten Regel aufgelöst, so ergibt sich nach der Substitution von a/u unter Anwendung auf $\bar{y}u$ die Ausgabe des Datums s auf Kanal u.

Prozesse, die in eine Ersetzung eingebunden sind, bezeichnet man als *Redex*, sowie den resultierenden Term als *Kontraktion*. Ein essentieller Unterschied zu bisherigen verteilten Prozess/Kanal Modellen für verteilte Systeme ist die Möglichkeit, auch neue und auch private Kanalbezeichner zu generieren, über Kanäle zu dem Binde-Bereich des Bezeichners zu verschicken und somit neue Kanäle zu schalten. Der Binde-Bereich wird dadurch auf die Zieldomäne erweitert, was man als *Scope Extrusion* bezeichnet. Eventuell auftretende Namenskonflikte werden durch Umbenennung behoben. Im folgenden Beispiel wird der Kanalbezeichner y via Kanal x aus dem Bereich der **new** y **in** Bindung geschickt, die dann verschoben werden muss:

$$\begin{array}{rcl}
(\textbf{new } y \textbf{ in } \bar{x}y \mid yv.P) \mid xu.\bar{u}c & \longrightarrow & \textbf{new } y \textbf{ in } yv.P \mid \bar{y}c \\
& \longrightarrow & \textbf{new } y \textbf{ in } \{c/v\}P
\end{array}$$

Ausschließlich der Kanalbezeichner ermöglicht die Kommunikation über diesen Kanal. Da Kanalbezeichner als Nachrichten via Kanäle verschickt werden können ist somit das dynamische Schalten neuer Kanäle zwischen Prozessen möglich. Dieses Schalten neuer Kanäle in Form des Sendens von Kanalbezeichnern verbunden mit Scope Extrusion ist das prägnanteste Merkmal des π-Kalküls gegenüber Ansätzen wie CCS.

Beim π-Kalkül ist zu beachten, dass im Falle einer Redex-Überlappung eine Reduktion zu verschiedenen Ergebnissen führen kann, da auf einem Kanal mehrere Ausgaben um

die Gunst der Eingabe wetteifern können:

$$\bar{x}a \mid \bar{x}bxu.\bar{y}u \longrightarrow \bar{x}b \mid \bar{y}a$$
$$\text{oder}$$
$$\bar{x}a \mid \bar{x}bxu.\bar{y}u \longrightarrow \bar{x}a \mid \bar{y}b$$

Solche nebenläufigen Wettstreite, bei denen zwei Prozesse um eine Ein- oder Ausgabe wetteifern, bezeichnet man als *plain interferences*. Diese finden statt, wenn ein Prozess mit zwei Partnern auf die gleiche Art nebenläufig interagiert, also einen Dienst anfordert oder anbietet und zwei "willige" Partner sich anbieten. Solche Plain-Interferences sind ausdrücklich erwünscht, um beispielsweise redundante Systemteile zu spezifizieren, die dynamische Lastverteilung realisieren.

Der π-Kalkül wird oft herangezogen, um *Mobilität* zu verdeutlichen. Hierfür ist er jedoch nur bedingt geeignet. Mobilität wird *implizit* durch Erreichbarkeit zwischen Prozessen modelliert, die Überschneidungen im Kanalbezeichnerraum besitzen. Besitzen zwei Prozesse einen gemeinsamen Kanalbezeichner, so sind sie erreichbar und halten sich somit evtl. in der selben Lokation auf. Abbildung 2.1 zeigt die durch den Bereich (scope) des Kanals *u* implizit modellierte Lokation der Prozesse. Durch die folgende Transition "migriert"also der Prozess C auf die gemeinsame Lokation :

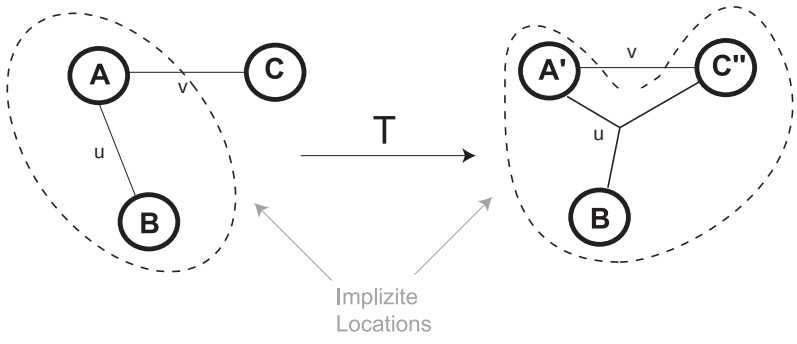


Abbildung 2.1: Implizite Lokationen im π-Kalkül

$$(u)(\bar{v}u.A' \| B) \mid v(z).C' \longrightarrow (u)(A' \mid B \| C'\{u/z\})$$

Fasst man die Eigenschaften und den Fokus des π-Kalküls zusammen, so lässt sich sagen, dass der π-Kalkül seinem Ziel gemäß, nämlich eine exakte Spezifikation nebenläufiger Prozesse und derer dynamischen Kommunikationsstrukturen zu ermöglichen, nur bedingt für spontane System geeignet ist. Jedoch bietet er eine sinnvolle Grundlage, um manche Mechanismen, wie zum Beispiel dynamische Verdrahtung von Komponenten zur Laufzeit, im Kleinen exakt zu spezifizieren. Die Vor- und Nachteile des π-Kalküls, im Kontext spontaner Systeme sind:

+ Dynamische Verdrahtung zur Laufzeit
+ Dynamische Kanalallokation

+ Implizite Prozessmigration

+ Keine explizite Lokation.

- Keine Trennung von logischer (Dienste und funktionale Abhängigkeit) und technischer (Komponenteninstanzen und Konfiguration) Architektur

- Keine Code-Migration (nur implizit mittels Prozessmigration und Zurücksetzen), keine explizite Prozessmigration (da keine explizite Lokationen)

- Keine Umgebungsprofile

- Keine Point-to-point- oder Multicast Kommunikation

- Keine indirekte Adressierung

Um mobile Systeme zu spezifizieren, ist der π-Kalkül aus oben aufgeführten Gründen nur bedingt geeignet. Will man nun mobile Systeme, insbesondere solche in Wide-Area-Netzen, modellieren, so stellen sich zunächst folgende fundamentale Fragen:

- *Welche Systemteile sollen Mobilität besitzen ?*
- *Was ist eine Lokation ?*
- *Was ist Kommunikation ?*

Ein Ansatz, der den π-Kalkül um Antworten auf diese Fragen erweitert, ist der *Ambient-Kalkül*. Hier besitzen so genannte Ambients (eigenständige Umgebungen) Mobilität. Eine Lokation ist ein solcher Ambient und Kommunikation findet lokal (innerhalb eines Ambients) asynchron und ungerichtet statt. Über Ambientgrenzen hinweg findet die Kommunikation jedoch via Migration (Mobilität) der Ambients statt.

Der Ambient-Kalkül ist eine auf dem π-Kalkül aufbauende Abstraktion für verteilte Systeme, die auf weitverzweigten Netzen wie dem Internet angesiedelt sind und über echte *Mobilität* verfügen. Die Grundidee ist hierbei, dass es sich bei weitverzweigten Netzen, sog. Wide-Area-Networks, nicht um ein homogenes Netz mit gleichen Rechten und Eigenschaften handelt, sondern vielmehr um “Flickwerk” von unterschiedlichen lokalen Netzen, die jeweils durch Grenzen wie Firewalls getrennt sind und eigene Rechte und Eigenschaften besitzen. Diesen Rechtezonen wird im Ambient-Kalkül dadurch Rechnung getragen, dass die Kommunikation über Migration stattfindet und die migrierende Einheit nicht nur die physikalische Möglichkeit haben muss, zum Kommunikationspartner zu migrieren, sondern auch alle Rechte, die dafür notwendig sind, die lokal administrierten Netze zu durchqueren.

Im Gegensatz zu klassischen Prozessalgebren, wie zum Beispiel dem π-Kalkül, bei denen die grundlegende Kommunikationsform das Verschicken von Nachrichten ist, baut der Ambient-Kalkül ausschließlich auf dem Prinzip von *Migration* auf. Mit Migration wird das Übersiedeln einer Einheit (z.B. eines Prozesses) auf eine anderen Lokation ermöglicht. Der Ambient-Kalkül ermöglicht also im Gegensatz zum π-Kalkül eine *explizite* Darstellung von Mobilität, durch *explizite* Ortsbestimmung.

Der Ambient-Kalkül gruppiert sich, analog zum π-Kalkül, um nebenläufige Prozesse. Jeder Prozess ist in einem so genannten *Ambient* angesiedelt. Innerhalb eines Ambients, der neben Prozessen auch wiederum Ambients enthalten kann, findet die Kommunikation nicht via Kanäle sondern asynchron und frei statt. Folgende Grammatik definiert die Prozesse und deren lokale Kommunikation:

P, Q	$::=$	0	nil
		$(\nu\, n)\, P$	Neuer Bezeichner n im Bereich P
		$P \mid Q$	Parallele Komposition von P und Q
		$!P$	Replikation
		$M[P]$	Prozess P in Ambient M
		$M.P$	Ausführen einer Capability
		$(n).P$	Lokale Eingabe in P auf *n* gebunden
		$\langle P \rangle$	Lokale asynchrone Ausgabe von P

Die Besonderheit des Ambient-Kalkül ist die Tatsache, dass jeder Prozess innerhalb eines so genannten *Ambients*, also einer *eigenständigen Umgebung*, abläuft. Prozesse innerhalb eines Ambients können unbeschränkt asynchron kommunizieren, für Kommunikation nach aussen setzt der Ambient die Kommunikationsrechte (sog. Capabilities) für die in sich enthaltenen Prozesse und Subambients fest. Diese Rechte richten sich auf die drei Grundprimitive zur Migration des Ambients: Eintreten in einen Ambient, Austreten aus dem Ambient und Auflösen eines Ambients:

M	$::=$	n	Bezeichner
		inM	Eintrittsfähigkeit
		$outM$	Austrittsfähigkeit
		$openM$	Öffnungsfähigkeit
		ε	Leerer Pfad
		$M.M'$	Kompositionspfad

Die Reduktionsregeln für diese Capabilities lauten wie folgt:

$$
\begin{array}{rcll}
n[in\ m.P \mid Q] \mid m[R] & \longrightarrow & m[n[P \mid Q] \mid R]; & \textit{Eintritt von n in m} \\
m[n[out\ m.P \mid Q] \mid R] & \longrightarrow & n[P \mid Q] \mid m[R]; & \textit{Austritt von n aus m} \\
open\ n.P \mid n[Q] & \longrightarrow & P \mid Q; & \textit{Öffnung von n}
\end{array}
$$

Soll zwischen Prozessen, die in verschiedenen Ambients angesiedelt sind, Kommunikation stattfinden, so muss ein Ambient sich selbst oder die Nachricht in den anderen Ambient migrieren. Diese Migration findet mittels der so genannten *Capabilities* (dt. Fähigkeiten) statt. Das Senden einer Nachricht *M* von einem Prozess in Ambient *A* zu einem Prozess in Ambient *B* wird durch "Verpacken" der Nachricht *M* in einem Ambient (Beispiel msg) und dessen Migration nach *B* realisiert:

$$
\begin{array}{ccc}
A[msg[\langle M \rangle \mid outA.inB]] & \mid & B[openmsg.(x).P] \\
\textit{send M: } A \rightarrow B & & \textit{receive x;P}
\end{array}
$$

Aufgrund der Tatsache, dass Capabilities als Nachrichten analog zu dem obigen Muster verschickt werden können, ist es möglich, *Migrationsrechte* und damit *Kommunikationsrechte* in einem verteilten System zu spezifizieren: Nur gewünschte Prozesse

bzw. Ambients, die zuvor die benötigte Erlaubnis in Form der Capability erhalten haben, besitzen das *Recht*, mit einem bestimmten Ambient bzw. Prozess zu kommunizieren. Hiermit ist es möglich, typische Konstrukte von Wide-Area-Anwendungen, z.B. Firewalls und mobiles Verhalten, zu modellieren. Diese Eigenschaft, die für spontane Komponentensysteme und insbesondere für solche in Wide-Area-Netzwerken typisch ist, wird in dem in dieser Arbeit vorgestellten Modell aufgegriffen und in einer anwendungsgetriebenen Form verwendet.

Ferner ermöglicht der Ambient-Kalkül im Gegensatz zum π-Kalkül die explizite Modellierung von Lokationen und Migration: Ein Ambient kann durch den Bezeichner explizit adressiert werden. Jedoch bringt das Fehlen einer Unterscheidung zwischen mobilen Netzwerken und mobilen Komponenten (beides wird durch Ambients abstrahiert) Probleme mit sich, wie sich bei grave interferences (siehe unten) zeigt

Für den Ambient-Kalkül existieren eine Vielzahl von Erweiterungen, wie Typsysteme [CG99], modale Logik [CG00] und Erweiterungen der Sicherheitseigenschaften [LS00].

Ein Problem im Ambient-Kalkül stellen so genannte *grave interferences*, die im Gegensatz zu den bereits oben vorgestellten erwünschten *plain interferences* nur beim Ambient-Kalkül auftreten. Durch das Betreten eines Ambients gibt der eintretende Ambient sämtliche Migrationsrechte an diesen ab. Ein enthaltener Ambient wird bei Migration des enthaltenden Ambients nicht berücksichtigt. Dies kann zu Problemen führen, wenn ein Ambient *A* beispielsweise einen Ambient *B* betritt. Migriert *B* anschließend in eine neue Umgebung, beispielsweise einen Ambient *Z* und tritt Ambient *A* nun wieder aus Ambient *B* aus, so findet er sich ein einer völlig neuen, unbekannten und eventuell ungeeigneten Umgebung wieder, wobei dieser Migrationsschritt nicht zu verhindern war.

Konfigurationsmanagement

Eines der größten Probleme verteilter Systeme ist das konsistente Regeln der Änderungen innerhalb des Systems. Man denke nur an den Ausfall einer Komponente und das Einbinden eines Ersatzes, oder die Erweiterung der Anforderungen eines Systems und das damit verbundene Erweitern des Systems. Dieses Verwalten der verschiedenen *Konfigurationen* wird als *Konfigurationsmanagement* eines Systems verstanden. Eine weitverbreitete Anwendung des Kofigurationmanagements ist das Verwalten von verschiedenen Datendokumenten und von deren Versionen. Das Ändern eines solchen Dokuments zieht evtl. Änderungen an anderen Dokumenten nach sich. Dies bedeutet, dass für eine Versionsänderung auch transitive Abhängigkeiten ausgewertet werden müssen. Systeme, die diese Abhängigkeiten automatisiert verwalten und auflösen, bezeichnet man als *Konfigurationsmanagement-Systeme*. Beispiele für solche Systeme sind das Revision Control System (RCS) [Tic85] oder auch CVS [Fog99]. Der prinzipielle Aufbau eines solchen Systems ist in Abbildung 2.2 dargestellt. Der Fokus dieser Systeme ist die Verwaltung der Abhängigkeiten – nicht jedoch die Umkonfiguration selbst, oder gar eine Umkonfiguration zur Systemlaufzeit. Für diese Arbeit ist daher das sog. *inkrementelle Konfigurationsmanagement* [KM90, Sal99, HMS99] relevant. Hier soll ein System *während der Laufzeit* durch seine Konfigurationseinheit von einer

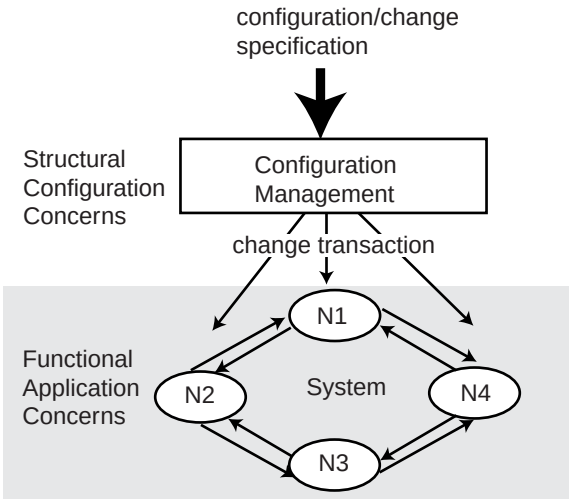


Abbildung 2.2: Systemkonfiguration und Konfigurationsmanagement (aus [KM90])

Konfiguration in die nächste überführt werden. Die Konfigurationseinheit erhält keine konkrete Spezifikation der anvisierten Konfiguration mehr, sondern nur einen Stimulus, der die Umkonfiguration auslöst. Inkrementelles Konfigurationsmanagement ist in Abbildung 2.3 dargestellt. Soll nun ein System durch inkrementelle Umkonfigura-

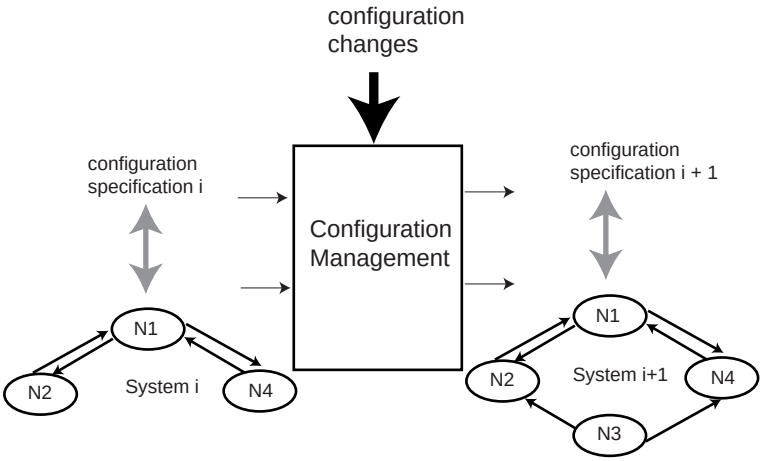


Abbildung 2.3: Inkrementelles Konfigurationsmanagement (aus [KM90])

tion angepasst werden, so müssen insbesondere folgende Eigenschaften sichergestellt werden:

- *Änderungen müssen in adäquater Abstraktion verfasst sein*: Die Codeebene ist zu feingranular, die Ebene ganzer Systemversionen ist trivial.
- *Änderungen müssen das System in einen konsistenten Zustand überführen*: Eine Umkonfiguration darf nicht zu unkontrolliertem Systemverhalten führen; das System muss das gleiche Verhalten wie vor der Umkonfiguration aufweisen (evtl. erweitert).

- *Änderungen sollten den Systemablauf möglichst unbeeinflusst lassen*: Im Optimalfall muss der Systemablauf nicht unterbrochen werden (z.B. durch Herunterfahren des Systems etc.) und die Umkonfiguration ist für Nutzer transparent.

Hieraus ergeben sich fundamentale Eingriffe in den *Systemablauf* und die Transaktionskontrolle: Der Systemablauf muss jederzeit kontrolliert angehalten und wiederaufgenommen werden können, die *Transaktionskontrolle* muss ausfallende Teilnehmer gesondert behandeln. Dieses Problem wird durch das sog. *Evolving Philosophers Problem* beschrieben: Die bekannten "Dining Philosophers" können während des Banketts sterben bzw. neu teilnehmen. Was geschieht mit den Gabeln?

Hier sind vor allem Arbeiten von Kramer und Magee [KM90, MK96] herauszustellen, die sich dieser Problematik angenommen haben. Als Ergebnis ist vor allem DARWIN zu nennen. DARWIN [MDEK95] ist eine Konfigurationssprache für verteilte Systeme, die 1995 von Jeff Kramer und Jeff Magee entwickelt wurde. Hier ist es möglich, Komponenten nicht nur durch die angebotenen Funktionalitäten, sondern auch durch deren benötigten Funktionalitäten zu spezifizieren.

DARWIN bietet eine Abstraktion für die Strukturinformation der Komponenten durch die Abbildung der angebotenen Funktionen auf einen Port und das Binden verschiedener Ports mittels Kanälen. Allerdings ist es hierbei nicht möglich, das Verhalten der Komponente zu spezifizieren. DARWIN beschränkt sich lediglich auf Signaturen.

```
component pipeline (int n) {
    provide input;
    require output;

    array   F[n]: filter;
    forall  k:0..n-1 {
        inst F[k];
        bind F[k].output -- output;
        when k<n-1 bind
            F[k].next -- F[k+1].prev;
        }
    bind
        input -- F[0].prev;
        F[n-1].next -- output;
}
```

Abbildung 2.4: DARWIN-Spezifikation einer Pipeline aus [MK96]

Auch die Bedeutung des Bindens, ausgedrückt durch das Schlüsselwort `bind`, wird nicht definiert. Abbildung 2.4 zeigt die DARWIN -Spezifikation einer Pipeline, die aus n Filtern besteht, die sequentiell verkettet sind. Ein Filter ist eine separat spezifizierte Komponente, die aus einem Eingangs- (*prev*) und zwei Ausgangs-Ports (*next* und *output*) besteht. Ports werden durch den Befehl *Bind* (`Bind Quelle -- Ziel`) gebunden, wobei die beiden Ports typkonsistent sein müssen. Man beachte, dass es sich hierbei um reine Strukturinformationen handelt. Die in Abbildung 2.4 in DARWIN spezifizierte Pipeline hat also folgende Struktur, die in Abbildung 2.5 graphisch dargestellt ist.

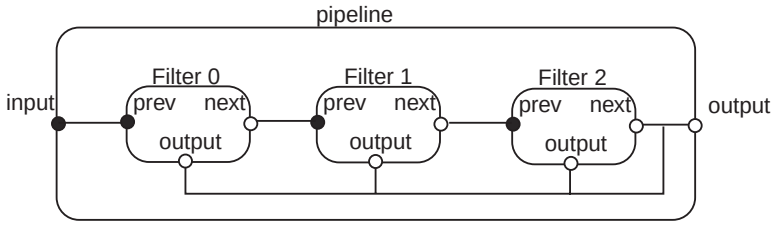


Abbildung 2.5: Die Struktur der in Abbildung 2.4 spezifizierten Pipeline

DARWIN bietet lediglich die Möglichkeit, eine explizite Konfiguration unabhängig von deren Implementierung zu definieren und diese auf eventuelle Inkonsistenzen zu prüfen. Diese Inkonsistenzen sind aber durch die Vernachlässigung der Verhaltensbeschreibung und des mangelnden semantischen Modells auf reine Strukturinformationen beschränkt. DARWIN ist ursprünglich als *Konfigurationsbeschreibungssprache* konzipiert. Neuerdings wird es auch oft als so genannte *Architectural Description Language* (ADL, siehe Abschnitt 2.2) bezeichnet, was es jedoch aufgrund der angegebenen Mängel nicht ist.

FOCUS

FOCUS ist ein am Lehrstuhl für Software & Systems Engineering der TU München entwickeltes mathematisches Systemmodell für verteilte Systeme [BS00, BDD+92]. Hierbei werden verteilte Systeme als Menge von Komponenten dargestellt, die nebenläufig über unidirektionale Kanäle durch Nachrichtenaustausch kommunizieren (siehe Abbildung 2.6). Komponenten, die eigentlichen Träger der Berechnung, werden durch stromverarbeitende Funktionen beschrieben, die den Eingangsstrom von Nachrichten auf den Eingabekanälen auf Ausgangsströme von Nachrichten auf den Ausgangskanälen abbilden. FOCUS stellt damit im Gegensatz zu Prozessalgebren wie CSP [Hoa85], die eine operationelle Semantik bieten, ein Modell auf denotationeller Semantik dar.

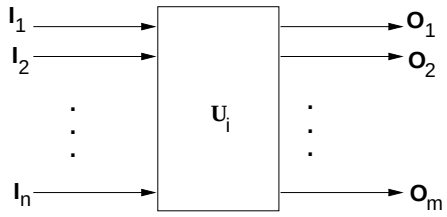


Abbildung 2.6: Eine Komponente (U_i) wird durch die Ein- (I) und Ausgangsströme (O) modelliert.

FOCUS bietet in seiner ursprünglichen Version die Möglichkeit, *statische* verteilte Systeme zu spezifizieren. Mit der Erweiterung DynamicFOCUS [GSB97] ist es möglich, *dynamische* Komponentennetzwerke, also Komponentennetzwerke mit verändernden Kanalzuordnungen, zu spezifizieren. Die Erweiterung verhält sich ähnlich wie die des π-Kalküls zu CCS: Private Kanalbezeichner, also Kanalbezeichner, die nur der besitzenden Komponente bekannt sind, können via Kanäle wie Nachrichten verschickt

werden. Erreicht ein Kanalbezeichner eine Komponente via direkte Verbindung oder verkettete Kanalverbindungen, so hat die Komponente gemäß dem mitgesendeten Zugriffsrecht (?n für Lesen, !n für schreiben und ?!n für Vollzugriff) die Möglichkeit, Nachrichten über den Kanal zu senden. Da für jede Komponente abzählbar unendlich viele private Kanalbezeichner angenommen werden, können unendlich viele neue Kanäle während der Systemlaufzeit aufgebaut werden. Da im Modell die Eigenschaft der *Privacy Preservation* gilt, kann eine Komponente nur auf ihr bekannten Kanälen senden und "vergisst" Kanäle, deren Zugriffsrecht ihr entzogen wurden, sofort wieder. Daher können auch Kanäle wieder abgebaut werden, und Kommunikationskonsistenz ist gewährleistet.

FOCUS wurde in dieser Arbeit als semantisches Basismodell verwendet, um Begriffe eindeutig zu definieren, sowie Verhaltensspezifikationen von Komponenten und Diensten zu ermöglichen.

Eine knappe Einführung in FOCUS wird in Kapitel 5 gegeben, der interessierte Leser sei auf [BS00, BDD+92] verwiesen.

Reflektionsmodelle

Reflektion ist die Eigenschaft von Systemen oder Systemteilen, so genannte *Metainformation* über sich selbst zu verarbeiten. Dies bedeutet, dass ein Systemteil Informationen über seine Struktur, Verhalten oder Repräsentation besitzt, die mit den beschriebenen Instanzen kausal verknüpft ist. Durch die Verarbeitung dieser Metainformation kann ein System Rückschlüsse über sich oder andere Systemteile und deren Beziehungen zueinander treffen. Durch Reflektion werden grundsätzlich zwei Ebenen eines Systems definiert: Die *Domänenebene*, die die eigentlichen verarbeitenden Elemente des Systems enthält und die so genannte *Metaebene*, die die Beschreibung der Elemente der Domänenebene enthält.

Reflektion hatte ihren Ursprung in Arbeiten im Umfeld der funktionalen Programmiersprachen. Hier ist vor allem Lisp und dessen Reflektionsmodell [Smi84] zu nennen. Hier wurde ein Lispsystem entworfen, das Zugriff auf den eigenen Interpreter hatte und somit Entscheidungen über sich und die aktuelle Laufzeitumgebung treffen konnte. Die wichtigsten Arbeiten sind dann im Umfeld von Smalltalk am Xerox PARC [KdRB91, Kic92], sowie am MIT [FSS00a, MN88] entstanden. Es wurden für objektorientierte Sprachen *Meta Object Protocols* (MOP) entwickelt, die mit Hilfe von objektorientierten Terminologien reflektive Informationen über objektorientierte Klassen und Frameworks zur Verfügung stellten. Das klassische Beispiel hierfür ist Smalltalk, das beispielsweise die Klasse einer Instanz wiederum als Objektinstanz im System hält. Diese kann nach objektorientierten Paradigmen im System behandelt und verarbeitet werden.

Heutige Programmiersprachen, wie zum Beispiel Java, greifen dieses Konzept auf und bieten Meta Object Protocols (bei Java werden diese *Introspection* genannt) an. Hierdurch ist es möglich die Metainformation von Objektinstanzen, die mittels dynamischer Klassenallokation zur Laufzeit ins System einbezogen werden, abzufragen und diese Informationen auszuwerten. Ein weiteres Beispiel ist das Component Object Modell (COM) von Microsoft, das via die Schnittstelle *IUnknown*, die von jedem Objekt

implementiert werden muss, die Möglichkeit gibt, eine bis dato unbekannte Objektinstanz auf Vorhandensein einer bestimmten Schnittstellenimplementierung zu prüfen.

All diesen Reflektionsmodellen ist jedoch gemeinsam, dass ein Verhalten auf der Domänenebene lediglich durch eindeutige *Bezeichner* auf der Metaebene beschrieben und identifiziert wird. Sind zwei Bezeichner gleich, so wird das Verhalten ebenfalls als äquivalent betrachtet. Dies ist natürlich nicht notwendigerweise so. Benötigt würde vielmehr eine *semantisch aussagekräftige* Repräsentation des Verhaltens auf der Metaebene, beispielsweise durch formale Beschreibungstechniken.

Für spontane Komponentensysteme sind Reflektionsmodelle durchaus wichtig, da bei dynamischer Umkonfiguration innerhalb eines Systems neue und damit potentiell unbekannte Komponenten in das System eingebunden werden müssen. Da zum Einbinden Information über Verhalten und Struktur der neuen Komponente notwendig sein können, bieten sich hierfür Reflektionsmechnismen an. Dies wird beispielsweise in Systemen, die Komponenten “just-in-time” einbeziehen, wie zum Beispiel bei per ASP-Verfahren (Application Service Providing) bereits praktiziert.

2.2 Architektur & Modellierung

Software-Architektur ist eines der zentralen Themen, die in dieser Arbeit erörtert werden. Im folgenden Abschnitt wird eine knappe Einführung in die gebräuchlichen Abstraktionen bei Architekturspezifikationen gegeben, anschliessend wird im speziellen noch auf einige Ansätze dynamisch änderbarer Architekturmodelle eingegangen. Daraufhin werden die gebräuchlichsten Ansätze zur Beschreibung von mikroskopischen (Komponenten und Interfaces) und makroskopischen Strukturen (Architekturen und Frameworks) vorgestellt.

Architekturbeschreibungen

Mit “Softwarearchitektur” wird das Zerlegen eines Gesamtsystems in Teilsysteme und deren Beziehungen bezeichnet. Architekturbeschreibung beruft sich meist auf das sog. *Komponenten & Konnektoren-Modell*[SG96], das Teilsysteme als Komponenten, also Berechnungseinheiten, und Konnektoren, also Verbindungseinheiten, beschreibt.

Konnektoren sind explizite, semantische Einheiten, die ein Interaktionsmuster zwischen einer Menge von Komponenten repräsentieren. Komponenten sind wiederum die eigentlichen Berechnungseinheiten des verteilten Systems. Hierbei wird meist verlangt, dass ein Konnektor nur zwischen Komponenten existieren kann. Konnektoren können demnach nicht hintereinander geschaltet werden. Die entscheidenden Vorteile dieser Abstraktion sind zum einen die explizite Trennung zwischen Berechnung und Interaktion, zum anderen die intuitive Sichtweise in Bezug auf das heute gängige Implementierungsmodell im objektorientierten Softwareengineering. Obwohl diese Art von Abstraktion sinnvoll und nützlich ist, um technische Strukturen und Interaktionen eines verteilten Komponentensystems, zum Beispiel eines CORBA-Systems, auszudrücken [Tai98], so ist sie doch für frühe Phasen oder spontane Systeme eher

hinderlich. Ein Nachteil dieser Abstraktion, besonders für die hier behandelten spontanen Komponentensysteme, ist gerade die deutliche Nähe zur Implementierung: es werden Abstraktionen benötigt, welche die *technische Realisierung*, d.h. insbesondere die teilnehmenden Komponenten und die Art der Verdrahtung offen lassen. Hierfür ist das Komponenten & Konnektoren-Modell ungeeignet, weil es keine Möglichkeit bietet, zwischen *logischen* und *technischen* Komponenten zu unterscheiden. Diese *logischen* Komponenten müssen auf technische Teilnehmer, die diese Rolle übernehmen könnten, abgebildet werden, und spannen somit einen Raum von möglichen Konfigurationen auf.

Ein entscheidender Nachteil der existierenden Komponenten & Konnektorenmodelle, wie zum Beispiel ACME [GMW97], ist die ausschließliche Konzentration auf *Strukturinformation* und das vollständige Aussparen von Verhaltensinformation. So werden unterschiedliche Typen, z.B. Komponenten und Konnektoren ausschließlich durch Struktureigenschaften bestimmt, ohne Eigenschaften im Verhalten spezifizieren zu können. Eine Instanz, die Signale auf einem Eingabekanal empfängt und diese alternierend auf zwei Ausgabekanälen ausgibt, würde intuitiv als Konnektor eingestuft, da sie keine eigentliche Berechnungsfunktionalität besitzt, sondern lediglich Daten umlenkt. Dieses Beispiel wird jedoch in vielen sog. *Architecture Description Languages* (kurz ADLs) als Komponente interpretiert, da es strukturell zwischen Konnektoren installiert wird [All97]. Darüberhinaus sind die Definitionen dieser Modelle meist informell und weisen Ambiguität auf.

Eine Komponente stellt im allgemeinen eine unabhängige, wiederverwendbare und isolierte Recheneinheit dar [All97]. Komponenten sind also die grundlegenden Berechnungseinheiten, die die eigentliche Systemlogik, also die Prozesse und Daten des Systems, formen. Sie umfassen allerdings *nicht* ihr momentanes Beziehungsgeflecht oder die Kommunikationsform der Daten. Des Weiteren sind Komponenten als unabhängig voneinander zu betrachten [SG96], wodurch die Wiederverwendbarkeit und die Kapselung der Berechnungseinheiten erzwungen wird. Die konkrete Bedeutung des Begriffs “Komponenten” hängt schließlich von der verwendeten Architekturbeschreibungssprache (Architectural Description Language – ADL) und von deren Modell ab. Beispiele hierfür sind WRIGHT [All97], UNICON [Zel96] oder RAPIDE [Luc96].

Ein Konnektor ist eine bestimmte, wiederkehrende Interaktionsform zwischen Komponenten, die gesondert spezifiziert wird [All97]. Konnektoren sind die zweite Aktorengruppe im Architekturmodell. Sie repräsentieren Interaktionsmuster unabhängig von den Teilnehmern (Komponenten) und regeln die jeweilige Kommunikation zwischen den teilnehmenden Komponenten. Wiederum hängt die präzise Definition des Konnektors von der jeweiligen ADL ab. In manchen, wie zum Beispiel WRIGHT [All97], werden Konnektoren lediglich als strukturelle Einheiten ohne jegliche Verhaltensunterschiede zwischen Komponenten und Konnektoren verwendet. Ein Konnektor ist lediglich dadurch definiert, dass er zwischen Komponenten angesiedelt ist.

Eine Lokation repräsentiert die unmittelbare Umgebung (nicht unbedingt den physischen Ort) einer Komponente oder eines Konnektors, was meist ein Wirtsrechner oder eine Wirtsplattform, z.B. eine Virtuelle Maschine oder ein Namensraum, ist. Komponenten, die auf einer Lokation angesiedelt sind, müssen deren Regeln folgen. So kann eine Lokation zum Beispiel ein Netzwerk hinter einer Firewall repräsentieren, wobei

alle IIOP Requests durch diese Firewall abgeblockt werden, was es der Komponente verbietet, mit Komponenten hinter der Firewall mittels IIOP zu kommunizieren und stattdessen nur eine Kommunikationsform mittels HTTP, zum Beispiel SOAP, zulässt.

Ein Architekturstil (engl. Architectural Style) stellt eine Menge von Entwurfsmustern und Idiomen dar, die einen eingeschränkten Entwurfsraum definieren, so zum Beispiel eine bestimmte Menge von Entwurfsalternativen [All97]. Der Designer ist somit in der Lage, verschiedene Strukturen auszudrücken, ohne sich in Implementierungsdetails zu verlieren.

Der Ansatz, Systeme mittels dieser Architekturidiome, Komponenten, Konnektoren und Lokationen zu beschreiben, erweist sich in der Praxis als überaus erfolgreich und beliebt, da er eine Reihe von Vorteilen verspricht: Eine nachvollziebare Abstraktion, eindeutige Designbeschreibungen etc. Nichtsdestoweniger werden von verschiedenen Gruppen durchaus verschiedene Bedeutungen mit den gleichen Begriffen verbunden [Med96]. So definiert WRIGHT [All97] einen Konnektor als rein methodisches Element, das keinerlei Verhaltensunterschiede zu Komponenten aufweist: Ein Konnektor ist etwas, was zwischen einer Menge von Komponenten angesiedelt ist. RAPIDE [Luc96] hingegen wird oft für das Fehlen eines Konnektor-Idioms kritisiert. Der Grund hierfür ist aber vielmehr, dass RAPIDE diesem methodischen Ansatz nicht folgt und daher keinerlei Grund hat, zwischen Komponenten und Konnektoren zu unterscheiden: Ein Konnektor ist ein spezieller Komponententyp. Wir verlangen daher, dass Komponenten nicht nur strukturell differenziert werden können, sondern auch durch ihr Verhalten differenziert werden können, um die Architektur eines Systems beschreiben zu können.

Die meisten Abstraktionen für Softwarearchitektur basieren auf dem vorgestellten Prinzip von Komponenten und Konnektoren. Ein alternatives Modell ist das von Berry und Boudol [BB90] 1990 vorgestellte Modell der Chemical Abstract Machine (CHAM). Unter dieser Abstraktion wird weniger die Struktur und das Verhalten eines System als Ganzes, sondern vielmehr die einer einzelnen Konfiguration desselben beschrieben. Der vorgebrachte Vorteil der CHAM ist die proklamierte Kompositionalität, die es erlaubt, einzelne Teilarchitekturen bequem zu einer Gesammtarchitektur zusammenzufassen.

Dynamisch änderbare Architekturen

Der Bedarf nach ständig verfügbaren Systemen, wie zum Beispiel Flugkontrollsystemen, Internetshops oder Telekommunikations-Switches, wirkt sich auch auf die Spezifikation von Softwarearchitekturen aus. Mit dynamischer Änderung einer Architektur werden die zur Laufzeit stattfindenden Operationen einer auf Komponenten/Konnektoren basierenden Architekturbeschreibung verstanden:

- Hinzufügen von Komponenten
- Entfernen von Komponenten
- Ersetzen von Komponenten

- Strukturelle Rekonfiguration (auf der bestehenden Menge von Komponenten und Konnektoren)

Für dynamisch änderbare Architekturen ist daher das Konzept der Konnektoren besonders wichtig, da Komponenten nur indirekt via einen Konnektor kommunizieren sollen, um bei einer Änderung der Architektur die Eingriffe auf das unmittelbare Umfeld (also nur die Konnektoren) zu begrenzen [OMT98]. Dieses Prinzip ist auch als *abstrakte Kopplung* aus dem Design-Pattern-Umfeld bekannt [G+94]. Es gibt ein große Anzahl von aus verschiedenen Richtungen kommenden Ansätzen, die dynamischen Änderungen der Architektur von Softwaresystemen zu regeln. Hier einige exemplarische Ansätze.

In [PHL97] wurde ein auf Haskel basierendes System vorgestellt, das Module zur Laufzeit auswechseln kann. Das System muss um bestimmte "Sollbruchstellen" herum aufgebaut werden, die von Funktionen gekapselt werden. Dadurch ist ein sicheres und feingranulares Management der Architekturänderungen möglich. Jedoch wird durch die starke Verwebung von Quellcode und Systemstruktur das System relativ unübersichtlich und auf die Zielplattform beschränkt.

Gorlick et al. stellten in [GR91, GQ96] einen auf Datenfluss basierenden Ansatz vor. Hier werden durch so genannte *Weaves*, Netzwerke atomarer nebenläufiger Komponenten, die Transaktionssicherheit geregelt. Eine Laufzeitumgebung sichert ACID-Eigenschaften eines Weaves zu, wobei der Nachrichtenaustausch zwischen den Komponenten durch so genannte Transportdienste asynchron geregelt wird. Die dynamische Umkonfiguration wird durch graphische Werkzeuge vom Programmierer vorgenommen, wobei dieser vollständig für das Change-Management, also zum Beispiel für die Konsistenz und Verfügbarkeit, zuständig ist.

In [GJB96] wird eine Programmiersprache definiert indem Kontrollabschnitte definiert werden, bei denen alle Variablen, die durch eine Strukturänderung beeinflusst würden, bei einer Änderung redefiniert werden müssen. Gupt et al. zeigen, dass solche Kontrollabschnitte nur approximativ definiert werden können.

Die hier vorgestellten Ansätze dynamisch änderbarer Architekturen richten sich vornehmlich alle auf von aussen vorgegebene Änderungen und die damit verbundene Konsistenz innerhalb des Systems, die es zu wahren gilt. Nicht behandelt werden Fragestellungen wie deren Auswirkungen auf den Entwurf, Umgebungsprofile und Lokationen.

CORBA, IDL und OCL

CORBA stellt einen der erfolgreichsten Standards für verteilte Komponentensysteme dar, wobei der Schwerpunkt hier auf Heterogenität der Plattformen und Unterstützung durch so genannte Infrastrukturdienste, wie zum Beispiel Transaction-Service oder Broker-Service, gesetzt wurde. Andere Ansätze, wie zum Beispiel Remote Method Invocation (RMI) [CIP00] oder das Component Object Model (COM) mit dessen Kommunikationsprotokoll DCOM [Box98] stellen vergleichbare Ansätze mit ähnlicher Verbreitung dar, besitzen jedoch nur eine Untermenge an Plattformverfügbarkeit

und Infrastrukturdiensten. Daher wird hier CORBA stellvertretend behandelt – die Eigenschaften und Methoden sind analog übertragbar. CORBA-Systeme werden in en-

```
/* IDL Definition eines Stacks */
module stack_module
{
interface Stack: Finite_Data_Structure, Container

void    push(in Any item);
Any     pop();
boolean empty();
void    clear();
}
```

Abbildung 2.7: Eine IDL Spezifikation des Stacks aus Abbildung 2.8

ger Anlehnung an ihre Implementierungsstruktur entworfen. Hierzu wird die konkrete *technische Architektur*, also die Festlegung der konkreten Verbindungsstruktur des Systems, mittels einer so genannten *Interface Definition Language* (IDL) definiert. Dabei wird durch eine definierte, implementierungunsabhängige Beschreibungssprache die *Schnittstelle* einer jeden Komponente des Systems definiert, nicht jedoch dessen operationelles Verhalten. Eine solche Definition wird schließlich durch einen plattformspezifischen Übersetzer auf die jeweils plattformtypischen Objektsignaturen übersetzt, die so genannten *Stubs* für die Clientseite und *Skeletons* für die Serverseite eines Inteaktionspaares.

Abbildung 2.7 zeigt die IDL-Spezifikation eines Stacks. Man beachte, dass hierdurch nur die Spezifikation der syntaktischen Schnittstelle möglich ist, nicht jedoch Aussagen über das Verhalten möglich sind. Des Weiteren legt man sich schon relativ früh im Entwicklungsprozess auf eine konkrete technische Architektur, also das Schneiden der Komponentengrenzen und deren Abbildung auf eine Plattform fest.

Die Object Constraint Language (OCL) der OMG [WK98] geht hier einen Schritt weiter: es ist möglich, in einer Spezifikation die von aussen beobachtbaren Eigenschaften einer Komponente, wie zum Beispiel Zusicherungen bezüglich einzelner Methoden (Funktionen) einer Komponente, auszudrücken. Dies wird durch Pre- und Post-Conditions, also Prädikate über eine Methode, realisiert. Abbildung 2.8 zeigt eine OCL Spezifikation des in Abbildung 2.7 mit IDL spezifizierten Stacks. Es ist in der OCL nicht möglich, interne Eigenschaften einer Komponente auszudrücken – es werden nur Blackboxeigenschaften berücksichtigt, die auf Methodenaufruf basieren.

Eine Trennung zwischen *logischer* und *technischer* Architektur und deren Abbildung, wie sie für spontane Komponenten notwendig ist, ist bei beiden Ansätzen nicht vorgesehen [Sal00].

OO Modellierung: UML und ROOM

In den letzten Jahren war eine Zunahme an Beachtung und Akzeptanz so genannter Modellierungssprachen zu verzeichnen. Beispiele für solche Modellierungssprachen sind beispielsweise die Realtime Object Oriented Modeling Language (ROOM) oder

```
-- LIFO stack
Stack
-----
::push(OclAny : item): Void
  pre:   -- none
  post:  item = self.top()
         size = size@pre + 1

::pop(): OclAny
  pre:   size >= 1
  post:  -- none

::empty(): Boolean
  pre:   -- none
  post:  result = (size = 0)

::clear(): Void
  pre:   -- none
  post:  self.empty() = true
         size =0
```

Abbildung 2.8: Eine OCL Spezifikation des Stacks aus Abbildung 2.7

die Unified Modeling Language (UML), die einen defacto Standard darstellt. Modellierungssprachen bieten eine Menge an graphischen Notationen, die Attribute und Relationen von Objekten eines objektorientierten Designs beschreiben.

Beide Modellierungssprachen unterscheiden sich jedoch von unserem Ansatz insofern, dass folgende Eigenschaften in bisherigen Modellierungssprachen nur teilweise vorhanden sind oder gänzlich fehlen:

- Sie unterscheiden nicht zwischen logischen Strukturen, die bei einem dynamischen System invariant bleiben, und technischen Konfigurationen, die diese Struktur auf verschiedene Weise realisieren. Es gibt zwar Konstrukte in der UML, die beispielsweise Alternativen und Bedingungen in Diagrammen ausdrücken können, diese beschreiben jedoch beispielsweise Alternativen zwischen zwei Bindungen in einem System, nicht jedoch eine Menge von Systemen, die einer Struktur gehorchen.
- Sie besitzen kein Konzept der direkt explizit modellierten Umgebungsprofile. Zwar existieren beispielsweise in der UML so genannte Deploymentdiagramme, diese dienen aber vielmehr zu Illustration der letztendlichen Installation von Komponenten und deren räumlicher Trennung als der Modellierung von Erreichbarkeit und heterogener Umgebungen.
- Sie besitzen keine formale Fundierung und Sachverhalte sind teilweise nicht eindeutig bestimmbar.

2.3 Anwendungen

Auch wenn die Anwendungsbeispiele in dieser Arbeit projektbedingt aus den Bereichen Mobile Computing stammen bedeutet dies nicht, dass sich die Anwendung der hier vorgebrachten Konzepte und Techniken auf dieses Anwendungsfeld beschränken. Schon bei geringer Abstraktion lassen sich die Gemeinsamkeiten in anderen Bereichen, wie Internet-Anwendungen oder Pervasive Computing sehen. Daher werden in den folgenden Abschnitten weitere Anwendungsfelder gestreift, um die Bedeutung spontaner Komponentensysteme zu illustrieren.

Pervasive und Ubiquitous Computing

Pervasive und Ubiquitous Computing (P&U Computing) bezeichnen eine neue Strömung innerhalb der Anwendung von Netzwerk und Computer. Durch die allgegenwärtige Verfügbarkeit von Netzwerkverbindung und die immer kleiner und leistungsfähiger werdende Hardware ist die Idee des P&U Computing eine *allgegenwärtige, durchdringende*, aber *unaufdringliche* Unterstützung durch Softwaresysteme. Diese Unaufdringlichkeit, welche bedeutet, dass der Nutzer möglichst wenig eingreifen muss, wird als *calm computing* bezeichnet. Der Rechner tritt in den Hintergrund und unterstützt den Nutzer wenn dieser es wünscht, oder sogar ohne dass dieser es merkt[2]. Realisiert

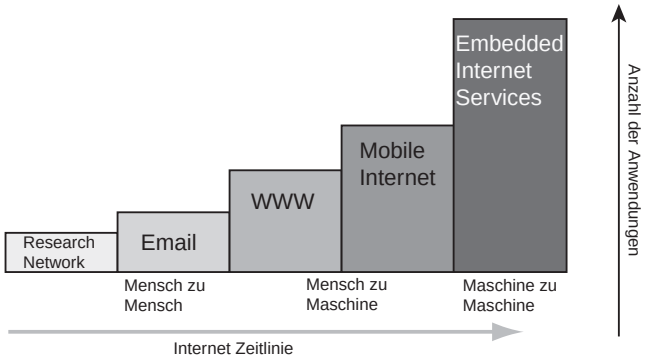


Abbildung 2.9: Entwicklung der Internets bzgl. der Anwendungsarten (aus [Mat00])

wird dies durch die Idee, dass jeder Alltagsgegenstand durch ein eingebettetes System eine eigene Intelligenz besitzt, die darüberhinaus auch an ein (eventuell mobiles) Netzwerk (z.B. das Internet) angebunden ist. Jedes Objekt besitzt dadurch im Netz einen so genannten *Datenschatten*, der eine art Proxy Objekt darstellt, das im Netz die jeweils benötigte Information des zu repräsentierenden Gegenstandes (bzw. der zu repräsentierenden Person) zur Verfügung stellt bzw. einholt. Das Anwendungsgebiet dieser *Embedded-Internet-Services* wird als aussergewöhnlich groß eingeschätzt (siehe Abbildung 2.9). Um jedoch die Unaufdringlichkeit der Datenschatten eines Gegenstandes zu ermöglichen, sind spontane Interaktion der Komponenten notwendig:

[2] Das dies natürlich auf datenschutztechnische und ethische Probleme aufwirft sei hier zwar angemerkt, aber nicht diskutiert.

eine Interaktion mit dem Benutzer muss möglichst vermieden werden. Daher müssen die Internetdienste gegenseitig spontan und autonom interagieren *ohne* zwingende Einbeziehung des Nutzers.

Mobile Computing

Mobilität im Zusammenhang mit Datenverarbeitung nimmt in den letzten Jahren überproportional zu. Das *"überall und jederzeit"* Paradigma, das durch Entwicklungen wie Internet und Miniaturisierung erst ermöglicht wurde, mündet sowohl in vollständigen Rechnern, die portabel und, meist drahtlos, vernetzt sind, als auch in extrem kleinen Spezialgeräten, die beispielsweise Funktionen eines Terminkalenders übernehmen.

Die neuen Strömungen im Mobilfunkbereich, die darauf abzielen, Sprach- und Multimediadienste zu verschmelzen, führen zu vollständigen Miniaturrechnern, die ständig online sind und darüberhinaus im Zusammenhang mit Ortsbestimmung neue Dienste, wie sog. *Location Based Services* [FSS00b] ermöglichen. Mobile Geräte sollen jedoch auch untereinander Daten und Dienste austauschen ohne den Benutzer in technische Einstellungen zu verwickeln. Ein autonomes, spontanes Zusammenspiel mobiler Geräte ist also notwendig. Darüberhinaus bieten die verschiedenen Netzwerkzellen drahtloser Netze, wie beispielsweise GSM, GPRS oder UMTS verschiedene Netzwerkprofile [FMS01], die in die Nutzung eines Dienstes miteinbezogen werden sollen. Will ein Benutzer eines UMTS-Mobiltelefons beispielsweise einen Dienst mit geringen Bandbreiten-Anforderungen, beispielsweise einen SMS-Dienst, nutzen und befindet sich in einer Zelle, die sowohl UMTS als auch das preiswertere GSM bietet, so kann der Dienst vollkommen transparent über das GSM Netz abgewickelt werden. An einem anderen Ort, der nur UMTS bietet, wird dann dieses Netz genutzt.

Im Laufe dieser Arbeit werden verschiedene Beispiele aus dem Mobile-Computing Umfeld gebracht. Als Realisierungsplattform dienen beispielsweise JINI (siehe Abschnitt 3.2.2) oder UPnP [Con00].

Wide-Area-Computing

Die speziellen Eigenschaften von internetbasierten Systemen, wie heterogene Netzwerkumgebungen, Latenzzeitbehandlung und unterschiedliche Soft- und Hardwareteilnehmer werden bei so genannten *Wide-Area-Computing* behandelt. Hier werden Systeme aus wechselnden Komponenten, den so genannten *Webservices* zusammengesetzt, die über Netzwerkgrenzen hinweg kommunizieren. Der Trend geht somit hin zu schlanken Basissystemen, von denen aus der Benutzer auf Webservices zugreift und zeitbasiert die Nutzung der Dienste bezahlt und nicht mehr eine Software pauschal erwirbt. Erste Ausläufer solcher Vertriebsmethoden, wie beispielsweise das sog. Application Service Providing (ASP) existieren bereits, jedoch handelt es sich hierbei um das Mieten von Programmen und nicht wie bei Webservices um das Mieten von Komponenten bzw. Diensten.

Ein entscheidender Punkt bei Webservices ist es, dass die Dienste flexibel sind und jedesmal aufs neue gebunden werden. Das bedeutet, dass ein Kunde zum einen Zeitpunkt

den Dienst des Herstellers *A* nutzt, bei nächsten Mal eine andere Komponente *B* eines anderen Herstellers, die aber denselben Dienst erbringt. Ein Grund könnte zum Beispiel der Preis oder unterschiedliche Qualitätskriterien sein. Entscheidend ist jedoch, dass die Auswahl der Komponente automatisiert und autonom, basierend beispielsweise auf einem Anforderungsprofil, stattfinden soll. Da es sich in einem System erwartungsgemäß um eine sehr große Anzahl von kooperierender Webservices handelt (man denke nur an Standardsoftware wie Officeprodukte) wäre eine ständige Einbindung des Nutzers nicht praktikabel. Dies soll nur in Ausnahmefällen geschehen. Darüberhinaus muss beachtet werden, dass bei Webservices die Tatsache zum Tragen kommt, dass das Internet nicht wie ein LAN eine homogene Netzwerkinfrastruktur besitzt, sondern vielmehr ein Geflecht verschiedener lokaler Netzwerke mit eigenen Profilen ist. Beispiele hierfür sind Firewalls, die manche Kommunikationsarten passieren lassen (z.B. HTTP), andere hingegen blockieren (z.B. CORBA) und lokale Zeitzonen oder lokal administrierte Rechte. Es kann also sein, dass ein Benutzer von einem Terminal *A* aus einen Webservice nutzen kann, nicht hingegen von einem anderen Terminal *B*, das sich in einer anderen Netzwerkumgebung befindet. Es muss daher wie in den oben beschriebenen Fällen eine spontane Interaktion stattfinden, die darüberhinaus auch die Eigenschaften heterogener Netzwerke mit einbezieht.

Beispiele für Plattformen für Wide-Area-Anwendungen sind zum einen die .NET Umgebung [Sur00] zum anderen auch Sun's JXTA Plattform [Gon01].

2.4 Zusammenfassung

In diesem Kapitel wurden die verschiedenen Felder aus Forschung und Anwendung aufgezeigt, die Einfluss auf diese Arbeit haben. Es soll nicht eine allgemein verständliche Einführung in die verschiedenen Themen darstellen, sondern vielmehr eine Sammlung der Themen sein, die der Leser zumindest einordnen können sollte.

KAPITEL

DREI

Entwicklung verteilter Komponentensysteme

Dieses Kapitel gibt einen Überblick über die momentan existierende Plattformen spontaner Komponentensysteme und deren Unterschiede zu herkömmlichen, statischen Komponentensystemen. Ziel ist es, dem Leser die internen Mechanismen und Abläufe, die charakteristisch für solche Systeme sind, zu verdeutlichen, und die daraus resultierenden Unterschiede in der Entwicklung klar zu machen.

Das Kapitel ist wie folgt aufgebaut: zunächst werden die Charakteristika statisch verteilter Systeme beschrieben, sowie geeignete Modelle, Spezifikationstechniken und Entwicklungsprozesse aufgezeigt. Anschließend wird auf die Unterschiede und Charakteristika spontaner Komponentensysteme eingegangen, sowie auf die Unzulänglichkeiten der beschriebenen Entwicklungsmethoden. Abschließend werden die Anforderungen an eine systematische Entwurfsmethode solcher Systeme diskutiert.

3.1 Statische verteilte Komponentensysteme

Statische verteilte Systeme zählen zu den erfolgreichsten Architekturen in der heutigen Softwarelandschaft. Gerade *verteilte Komponentensysteme*, die in dieser Arbeit besondere Beachtung finden, sind stark verbreitet. Man denke nur an Internetanwendungen, Client/Server Systeme oder den normalen Single-User-Arbeitsplatz. In all diesen Systemen werden zumindest Teilkonzepte aus dem Gebiet verteilter Systeme verwendet. Im folgenden Abschnitt werden kurz die zentralen Vorteile, die im weiteren Verlauf auf spontane, verteilte Komponentensysteme übertragbar sind, umrissen. Die Schichtung solcher Systeme ist in der folgenden Abbildung 3.1 abgebildet.

3.1.1 Charakteristika verteilter Systeme

Im folgenden Abschnitt werden kurz die Charakteristika statischer, verteilter Systeme vorgestellt, da diese den Entwurf und die Entwicklungsmethoden solcher Systeme

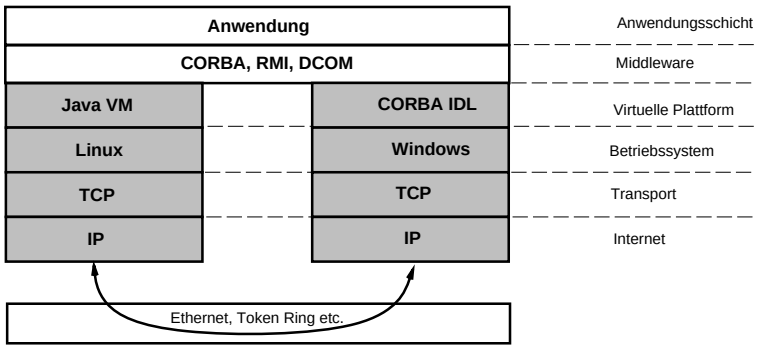


Abbildung 3.1: Schichtenaufbau von statischen Middleware-Plattformen

bestimmen. Die meisten dieser Eigenschaften gelten auch für spontane, verteilte Systeme. Besonderes Augenmerk ist hierbei auf die Eigenschaften von praktischer Relevanz wie z.B. Skalierbarkeit gelegt, da diese die Qualität einer Plattform und ihres Entwurfsprozesses im angewandten Fall bestimmen.

Da der Fokus dieser Arbeit auf spontanen Komponentensystemen liegt, kann hier nur ein knapper Umriss dieser Eigenschaften gebracht werden. Für eine ausführlichere Einführung sei auf [Fle94, MS92] verwiesen. Die Charakteristika, die hier besonders behandelt werden, sind im einzelnen:

Mehrere Komponenten: Verteilte Systeme bestehen aus einer nicht-trivialen Menge von Komponenten. Unter einer Komponente versteht man eine selbstständige, wiederverwendbare Einheit, die über Rechenwerk, Speicher sowie Kommunikationsinfrastruktur verfügt. Eine Komponente kann zum Beispiel ein vollständiger Computer mit Netzwerk sein oder auch eine Softwarekomponente, die über eine Kommunikationsmiddleware wie CORBA (siehe unten) verfügt.

Verbindungsstrukturen: Die Menge an Recheneinheiten kommuniziert über gemeinsame Verbindungsstrukturen. Weitverbreitete Abstraktionen dieser Verbindungsstrukturen sind Kanäle (Pipes) [SG96], die zwischen eine Menge von Recheneinheiten oder gemeinsame Speicherbereiche wie z.B. Tupelspaces [Gel85] geschaltet sind. In der Praxis werden diese Abstraktionen meist durch Object Request Broker (ORB) [HOE96] oder gemeinsame Speichersysteme wie z.B.Harddiskdrives oder Internet URLs realisiert.

Nebenläufigkeit: Verteilte Systeme sind durch die Tatsache gekennzeichnet, dass Operationen sich gegenseitig beeinflussen können und durch einen oder mehrere Ablaufumgebungen (sog. Threads) nebenläufig ausgeführt werden. Das bedeutet insbesondere, dass Eigenschaften wie Koordination und Synchronisation im Entwurfsprozess miteinbezogen werden müssen.

Skalierbarkeit: Eine Kerneigenschaft verteilter Systeme, die zunehmend in den Vordergrund tritt, ist die Skalierbarkeit. Bei der Qualitätsbewertung vieler Middleware-Plattformen wird die Performanz als wichtigstes Qualitätsmerkmal zunehmend von der Skalierbarkeit abgelöst. Unter Skalierbarkeit versteht man die Expansion des Systems in vorhergesehenen Größen, z.B. die Menge der Komponenten, wobei am grundlegenden Aufbau des Systems keine Änderungen vorgenommen werden müssen. Die Namensgebung und das Adressieren von Komponenten ist beispielsweise ein Kernproblem, das die Skalierbarkeit eines Systems bestimmt.

Niedrigere Fehler- und Ausfallwahrscheinlichkeit: Der dezentrale Charakter verteilter Systeme bedeutet auch ein höheres Maß an Fehlertoleranz und Ausfallsicherheit. Fällt eine Komponente aus, so muss dies nicht unbedingt alle Funktionalitäten des Systems lahmlegen. Lediglich jene Komponenten, die funktional abhängig sind, und deren assoziierte Funktionalitäten fallen aus. Diese Abhängigkeiten werden z.B. durch Datenflussanalyse [MJ81] festgestellt und das verteilte System dermaßen strukturiert, dass eine hohe Ausfallsicherheit gewährleistet ist.

Heterogene (Soft- und Hardware-) Komponenten: Ein Aspekt, der zum Beispiel bei der CORBA [Gro98] Architektur im Mittelpunkt steht, ist Heterogenität. Legacy Systeme, d.h. ältere Systeme, die zwar nicht auf dem aktuellen Stand der Technik, jedoch ausgereift und verlässlich sind, werden sehr häufig als eine Komponente in einem verteilten System gekapselt. Daraus ergibt sich ein neues, verteiltes System, das alle hier genannten Vorteile eines verteilten Komponentensystems unter Beibehaltung des verlässlichen Legacy Systems bietet. Hierfür muss jedoch die Heterogenität der alten und neuen Systeme, angefangen von Datenstandardisierung über Unterstützung verschiedenster Betriebssysteme bis hin zur abstrakten, implementierungsunabhängigen Beschreibung durch sog. Interface Definition Languages (IDLs) [HOE96] (siehe auch 2.2), gewährleistet sein.

3.1.2 Entwicklungsprozesse

Der Entwurf und die Modellierung verteilter Komponentensysteme erweist sich aufgrund der oben beschriebenen Charakteristika, wie Nebenläufigkeit oder Heterogenität, als weitaus komplexer als der Entwurf beispielsweise monolithischer Systeme. Dies ist der Grund, warum speziell bei verteilten Systemen, *Entwurfsprozesse*, wie beispielsweise der Rational Unified Process (kurz RUP) [Kru00] oder Catalysis [DW98], verwendet werden. Da keine umfassende Einführung in Entwurfsprozesse für verteilte Systeme im allgemeinen vorgestellt wird, wird hier lediglich exemplarisch der RUP als populäres (nicht unbedingt als bestes) Beispiel umrissen.

Im RUP wird der Entwicklungsprozess der Software in verschiedene *Phasen* (Inception, Elaboration, Construction und Transition) eingeteilt, wobei jede Phase einer Menge von Iterationen zugeordnet ist. Orthogonal zu den Phasen existieren die verschiedenen Workflows, die die jeweiligen Arbeitsarten, wie Requirements Engineering, Testen, Analyse & Design etc., über die Zeitachse der Phasen widerspiegeln. Für den

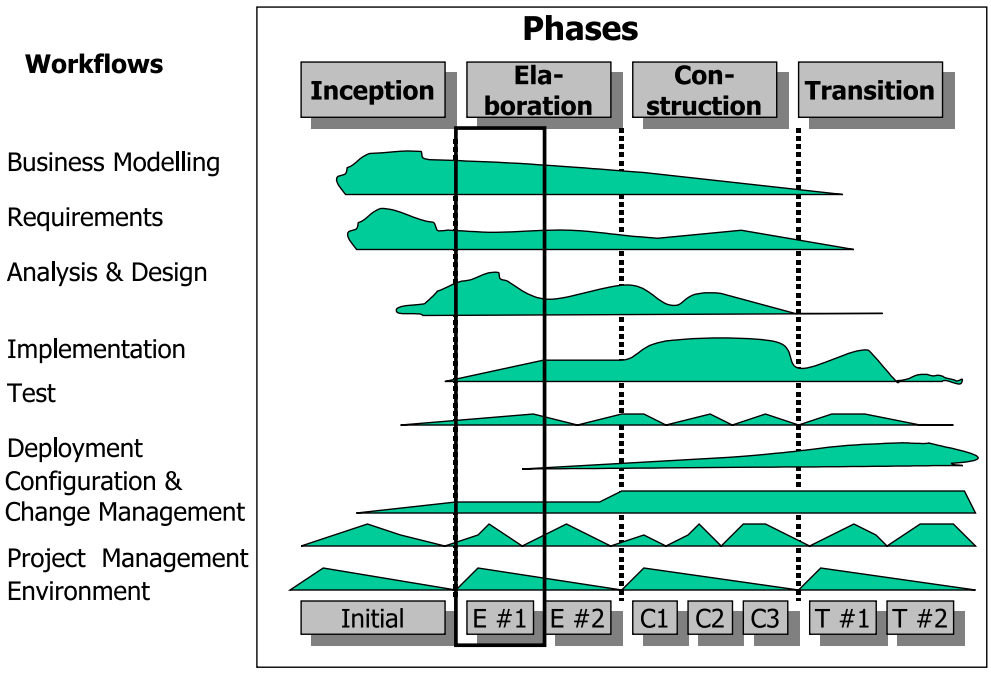


Abbildung 3.2: Grobübersicht der Phasen und einzelnen Workflows des Rational Unified Process

Fokus dieser Arbeit ist Analyse & Design, der Deployment Workflow und der Konfiguration and Change Management Workflow ausschlaggebend. Bedingt spielen natürlich noch andere, wie z.B. Implementierung, eine Rolle. Betrachtet man nun beispielsweise die Übersichtsgrafik 3.2, so sieht man, dass der RUP mit der zeitlichen Phaseneinteilung und der Gewichtung der einzelnen Workflows in den Phasen nur bedingt für spontane Komponentensysteme geeignet ist. Da beispielsweise ein zentraler Aspekt spontaner Komponentensysteme die veränderbare Konfiguration ist, tritt das Konfigurationsmanagement wesentlich früher auf und ist oftmals verwoben mit dem Design- und Analyse-Workflow. Dieser läuft jedoch in statischen Systemen konträr zum Konfigurationsmanagement Workflow. Der Entwurfsprozess für statische Komponentensysteme ist also nur bedingt auf spontane Komponentensysteme anwendbar und bedarf zumindest einer Anpassung in bezug auf die zeitliche Abfolge der verschiedenen Workflows.

Der Leser möge beachten, dass im Rahmen dieser Arbeit die *Techniken* für den Entwurf spontaner Komponentensysteme vorgestellt werden, also die Abstraktionen, Modelle und deren Abbildung auf Plattformen. Eine Einbettung dieser Techniken in einen konkreten Entwurfsprozess, wie hier gerade vorgestellt, würde den Rahmen der Arbeit sprengen und wird daher für zukünftige Arbeiten vermerkt (siehe 12.3).

3.1.3 Standardplattformen

Die grundlegende Aufgabe einer Middleware-Plattform für verteilte Komponenten ist es, die *Transparenz der Verteilung* zu gewährleisten, eine *heterogene Systemstruktur*

zu ermöglichen, sowie die notwendigen Infrastrukturdienste bereitzustellen.

- **Transparenz der Verteilung** sichert, dass die Verteilung auf verschiedene Rechner transparent für die Recheneinheit (Komponente, Rechner etc.) erscheint. Das bedeutet insbesondere, dass sich entfernte und lokale Methodenaufrufe im Code nicht unterscheiden. Die einzelnen Infrastrukturmaßnahmen, wie Marshalling, Naming und Remote Call werden transparent von der Plattform (Middleware) übernommen.
- **Heterogene Systemstruktur** bezeichnet die Struktur eines Systems, das aus heterogenen *Komponentenarten* (z.B. synchrone vs. asynchrone Kommunikation) und *Komponenten-Plattformen* (z.B. Implementierungssprache, Betriebssystem der Laufzeitumgebung etc.) besteht.
- **Infrastrukturdienste** werden benötigt, um ein System zu installieren und zu betreiben. Beispiele hierfür sind Naming- [HOE96], Persistenz- [HOE96] oder Transaktions- [HOE96]Dienste.

Im folgenden Abschnitt werden drei der populärsten Middleware-Plattformen exemplarisch vorgestellt.

Common Object Request Broker Architecture – CORBA

CORBA steht für Common Object Request Broker Architecture und spezifiziert eine Middleware-Architektur, die von der Object Management Group (OMG), einer Non-Profit-Organisation, definiert wurde. Mittlerweile befindet sich CORBA in der Version 3.0. Die OMG definiert nur die Architektur, stellt aber keine Implementierung zur Verfügung, was verschiedenen Firmen, wie zum Beispiel Orbix, IBM, Visigenic etc. überlassen wird.

Transparenz der Verteilung

Für CORBA-Objekte werden entfernte Aufrufe via Interfaces des entfernten Objekts getätigt. Diese sind somit vollkommen transparent für das Objekt selbst. Die Schnittstellen können auf zwei Arten alloziert werden, statisch oder dynamisch. Die statische Allokation, die bei weitem populärer ist, verlangt, die Schnittstellen mittels einer standardisierten Beschreibungssprache, der sog. IDL, zu spezifizieren. Ein Übersetzer generiert aus der Schnittstellenbeschreibung sog. Stubs auf Client-Seite und Skeletons auf Server-Seite, die dann die Abbildung auf das jeweilige Objekt samt Implementierungssprache und Plattform übernehmen. Im Falle der dynamischen Allokation kann das Interface zur Laufzeit alloziert werden. In beiden Fällen ist jedoch zu beachten, dass die Schnittstellen und damit die Objekte über den *Bezeichner*, nicht jedoch über einen *Typ* alloziert werden.

Heterogene Systemstruktur

CORBA stellt einen sowohl plattform- also auch implementierungsunabhängigen Standard dar. Eines der Ziele ist hierbei die Integration von Alt-Systemen, auch Legacy Systemen genannt, mit neuer Technologie. Dies bedeutet, dass CORBA ein Spektrum von Mainframesystemen unter COBOL bis zu Kleinstrechnern unterstützen kann.

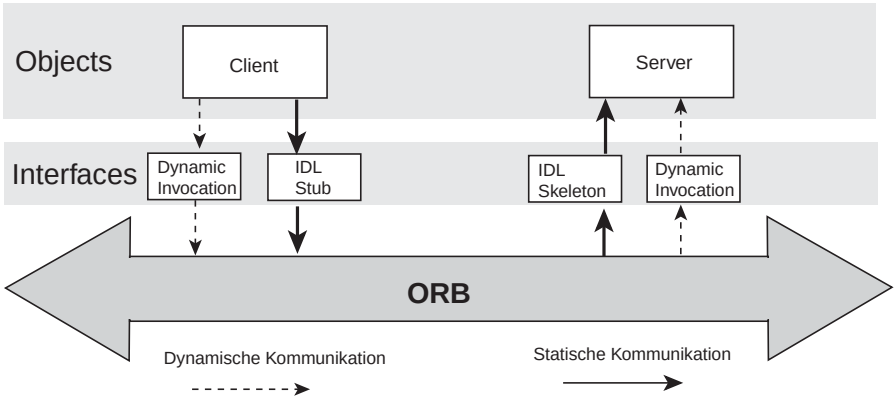


Abbildung 3.3: CORBA Kommunikationsstruktur

Infrastrukturdienste CORBA bietet ein breites Spektrum von Infrastrukturdiesten, die von Naming-Service über Transaction-Service bis zum Security-Service reichen. Ein im Zusammenhang mit spontanen Komponentensystemen bemerkenswerter neuer Dienst ist der in CORBA 3.0 standardisierte *Trading-Service*. Via Trading ist es möglich, die Objekte zur Laufzeit nicht nur bezüglich deren Bezeichner zu allozieren, sondern auch aufgrund deren Typs.

Component Object Model – COM

Das Component Object Model, kurz COM, der Firma Microsoft entstand aus der Vernetzung verschiedener Dokumenttypen heraus. Die damalige Bezeichnung OLE (Object Linking and Embedding) wurde von COM abgelöst, was mit Interneterweitertungen namens ActiveX ergänzt wurde und nun im Rahmen der .NET Initiative als COM+ bezeichnet wird. Der Kommunikationsmechanismus für verteilte Komponentensysteme wird mit DCOM (Distributed Component Object Model) bezeichnet.

COM Komponenten können nur via Interfaces angesprochen werden. Diese Interfaces sind starr festgelegt, also nicht mehr revidierbar, und werden von Microsoft standardisiert und mit einem eindeutigen 16-Byte Bezeichner, der sog. UUID, versehen. Dadurch ist es möglich, Komponentensysteme zu konstruieren, die neue Eigenschaften, die in ein System hereingetragen werden, transparent und autonom vom System einzubinden. Tritt ein neues COM-Objekt (z.B. in Form eines mobilen Gerätes) in das System ein, so wird dieses von den existierenden Objekten via das *IUnknown* Interface, das jedes COM Objekt implementiert, gefragt, ob es bestimmte Interfaces, gekennzeichnet durch die eindeutige UUID, implementiert. Falls ja kann diese neue Funktionalität eingebunden werden. Die Standardisierung der Interfaces ist, da das Verfahren durch eine Institution zentralisiert ist, weitaus effizienter und fortgeschrittener als bei RMI oder CORBA, bei denen die Standardisierung durch Gremien und deren Prozesse von statten geht.

Zu beachten ist, dass Komponenten unter COM nur bezüglich deren Klassenbezeichner und Interfacebezeichner alloziert werden können – nicht jedoch über deren Typ.

Transparenz der Verteilung COM Objekte kommunizieren untereinander via Referenzen auf Interfaces. Im Falle von DCOM kann die Referenz auch entfernt sein. Eine Komponente alloziert ein Objekt durch einen Aufruf bei der zentralen COM-Library (im Microsoft Windows Betriebssystem integriert), durch Angabe des eindeutigen Klassenbezeichners (CLID) sowie des Interfacebezeichners (IFID) des Interfaces via dessen die Kommunikation mit dem Objekt stattfinden soll. Die COM Library er-

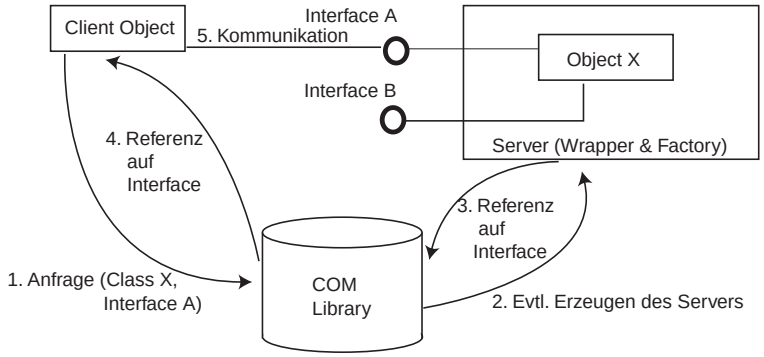


Abbildung 3.4: COM Kommunikationsstruktur

zeugt, wenn nicht schon vorhanden, einen sog. COM-Server für die Klasse, welcher als Wrapper und Factory [G+94, Hel96] für die Interfaces des Objekts fungiert. Soll später das Objekt via ein anderes Interface angesprochen werden, so erzeugt der Server dies transparent. Nach Instanziierung des Objekts und des Interfaces wird die entsprechende Referenz über die COM-Library an das anfragende Objekt übergeben und die Kommunikation zwischen den COM Objekten beginnt. Der Ablauf ist in Abbildung 3.4 dargestellt.

Heterogene Systemstruktur COM ist eine Entwicklung der Firma Microsoft und beschränkt sich daher hauptsächlich[1] auf die Produktfamilie der Microsoftbetriebssysteme, die allerdings von Workstation Systemen (Windows 2000) über PC Systeme (Windows ME) über PDA Plattformen (WindowsCE) bis zu eingebetteten Systemen (embedded Windows) reichen.

Infrastrukturdienste COM selbst stellt lediglich ein Komponentenmodell für Verteilung auf einer Plattform dar. Erweiterungen, wie DCOM (Distributed COM) bieten ein Protokoll für Methodenaufrufe über Rechnergrenzen hinweg. Infrastrukturdienste, wie Transaktionsdienste können aus dem Windows Betriebssystem heraus transparent genutzt werden. Explizite COM Infrastrukturdienste, vergleichbar mit Trading oder Security Diensten aus CORBA, existieren nicht.

[1] Microsoft ist mittlerweile auch bemüht, COM als offenen Standard durchzusetzen. Es existiert auch eine LINUX-Implementierung von COM, die aber nicht vollständig unterstützt wird.

Remote Method Invocation – RMI

Remote Method Invocation (kurz RMI) ist eine Erweiterung der Java Plattform um einen transparenten Verteilungsmechanismus, der sich an RPC (Remote Procedure Calls) anlehnt.

Transparenz der Verteilung RMI Komponenten werden ähnlich wie CORBA Komponenten mittels einer Interface Definition Language (JavaIDL) definiert, die wiederum in Form von Stubs (Client) und Skeletons (Server) Objektgerüste erzeugt, die wiederum aus Komponenten und Methodensignaturen bestehen. Für den dynamischen Laufzeit Aufruf – analog zu dem oben beschriebenen CORBA DII – werden die Interfaces mittels Reflektion durch die Java Introspection Mechanismen alloziert.

Heterogene Systemstruktur RMI baut auf der Java virtuellen Maschine (JVM) auf, die für nahezu jede Hardwareplattform erhältlich ist. Durch diese Indirektionsschicht ist RMI zwar ausschließlich für Java, aber indirekt für nahezu jede Plattform erhältlich. Ein Nachteil, der bei neutralen Standards wie CORBA nicht zu verachten ist, ist die Trennung zwischen Spezifikationsgremium und Hersteller der Implementierung. Durch diese Trennung im Falle von CORBA in OMG (Object Management Group) und verschiedene Hersteller, die die eigentliche Middleware herstellen, sind viele der CORBA Implementierungen nur bedingt kompatibel, da die jeweiligen Hersteller den Standard entweder erweitert oder Unterspezifikationen auf verschiedene Weisen realisiert haben. Im Falle von RMI liegt Spezifikation und Realisierung in einer Hand, was in der Praxis zu einer wesentlich höheren Verlässlichkeit führte, wenn verschiedene heterogene Implementierungen kommunizieren sollen.

Darüberhinaus sind RMI Objekte auch in der Lage mittels IIOP mit CORBA Objekten zu kommunizieren und bieten daher einen hohen Grad an Heterogenität.

Infrastrukturdienste RMI ist sehr stark mit der Java Plattform verwoben, was das Fehlen vieler expliziter Infrastrukturdienste wie Transaktionsdienst, Sicherheit etc. erklärt. Diese werden durch die Java Plattform selbst gehandhabt und machen damit einen expliziten, plattformunabhängigen Infrastrukturdienst obsolet.

3.2 Spontane Komponentensysteme

Im Mittelpunkt dieser Arbeit steht eine spezielle Art von verteilten Komponentensystemen: spontane Komponentensysteme. Diese zeichnen sich zusätzlich zu den Charakteristika statisch verteilter Systeme durch vier Hauptmerkmale aus: *Dynamisches Verhalten, Mobilität, Umgebungsprofile* (Kommunikations- und Migrationsrechte) und *spontane Interaktion*. Spontane Komponentensysteme zielen auf stark verteilte Anwendungen in mittleren und großen Netzwerken ab. Die Schichtung ist in der folgenden Abbildung 3.5 abgebildet.

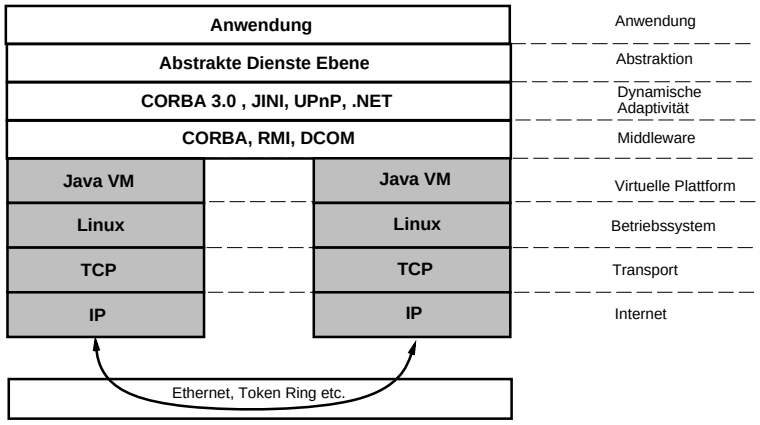


Abbildung 3.5: Schichtenaufbau von spontanen Middleware-Plattformen

3.2.1 Charakteristika spontaner Komponentensysteme

In diesem Abschnitt werden die Charakteristika spontaner Komponentensysteme herausgearbeitet. Zu beachten ist allerdings, dass gemäß der in Kapitel 1.2 gegebenen Definition solcher Systeme, nicht alle der hier aufgeführten Charakteristika gemeinsam auftreten. So kann es spontane Komponentensysteme statischer oder mobiler Art geben. Zu den hier aufgeführten Eigenschaften kommen die eben beschriebenen Eigenschaften verteilter Systeme natürlich noch hinzu, da es sich bei spontanen Komponentensystemen um eine spezielle Art von verteilten Komponentensystemen handelt.

Dynamisches Verhalten: Unter dynamischem Verhalten wird in der vorliegenden Arbeit die Eigenschaft eines Systems verstanden, während der Laufzeit autonom die eigene Struktur zu verändern. Die Anpassung beinhaltet im Modell eines verteilten Systems, das aus Komponenten, Wirtsrechnern und Kanälen existiert, den Aufbau und das Fallen lassen neuer Kanäle sowie das Erzeugen und Zerstören neuer Komponenten bekannten Typs. Man beachte, dass dies insbesondere auch die Eigenschaft der Code-Migration (siehe unten), d.h. das Wechseln eines Wirtsrechners durch eine Komponente, wobei der Zustand *nicht* beibehalten wird, auch unter den Begriff Dynamik fällt.

Ein System wird als *mobil* bezeichnet, wenn in seinem Systemmodell der Begriff der Lokation (oder Wirtsrechner) auftaucht sowie die Fähigkeit, diesen Wirtsrechner während der Laufzeit dynamisch zu wechseln, spezifiziert wird. Bei näherer Betrachtung reicht dies jedoch nicht aus. Es muss zwischen *technischer* und *struktureller Mobilität* [Car99b] unterschieden werde.

Mobilität: Der Begriff *Mobilität* wurde bereits grob umrissen. Nun definieren wir zunächst eine spezielle Form von Mobilität, die *technische* Mobilität. Prinzipiell unterscheidet man zwei Typen von technischer Mobilität: *Code-Migration*, d.h. das Versenden einer ausführbaren Codeeinheit und deren anschließende Initialisierung, sowie *Prozessmigration*, d.h. das Versenden einer Codeeinheit samt ihres Zustandes. Bei

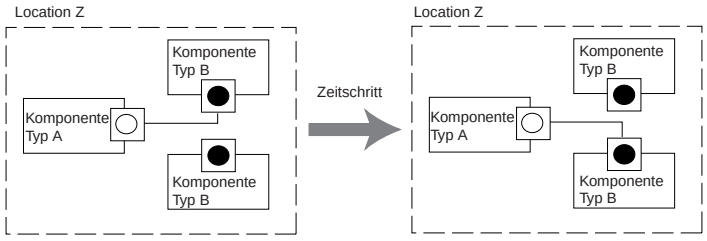


Abbildung 3.6: Dynamisches Verhalten: Umschalten eines Kanals (Ein-/Ausgabe-Ports als schwarze bzw. weisse Kreise)

Prozessmigrationssystemen wird demnach als echte technische Neuerung die Existenz einer mobilen, persistenten Softwareeinheit eingeführt. Bei solchen Systemen ist es nun notwendig, den Zustand der mobilen Einheiten hinreichend zu definieren, wobei das technisch Sinnvolle, mit dem konzeptionell Wünschenswerten abgewägt werden muss.

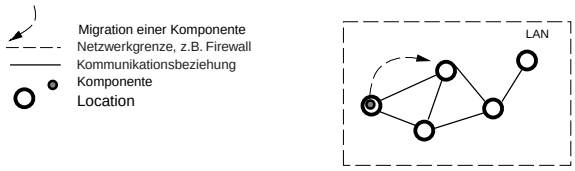


Abbildung 3.7: Mobilität in einem homogenen Netzwerk

Die technische Mobilität, also das Wandern von Code oder Prozessen, impliziert auch das Wandern von *Funktionalitäten*. Damit verändern sich beispielsweise die lokal auf einer Lokation zur Verfügung stehenden Funktionalitäten. Dies bezeichnet man auch als *funktionale Mobilität*. Man beachte hierbei, dass sich funktionale Mobilität auch über mehrere Indirektionsebenen auswirken kann, wenn Funktionen (Dienste) indirekt von einander abhängig sind (siehe hierzu auch Abschnitt 8.1.3).

Aber auch die Domäne, in der die Mobilität stattfindet, hat Einfluss auf das abstrahierte Verhalten. Hierbei unterscheidet man drei Arten von *struktureller Mobilität*: *homogene*, *heterogene* und *mobile Netzwerke*. Im allgemeinen wird Mobilität meist nur auf die technische Mobilität einer Komponente, z.B. von einem Rechner *A* zu einem vernetzten Rechner *B*, zu gelangen, betrachtet, was jedoch nur im ersten Fall, d.h. innerhalb von homogenen Netzwerken, ausreichend ist (siehe Abb 3.7). Dies ist jedoch gerade für mobile Komponenten, die sich in unbekanntes Terrain begeben sollen, unzureichend. In heterogenen Umgebungen (wie Wide-Area-Networks) sind zusätzlich zu den rein technischen Mobilitätseigenschaften (s.o.) noch die strukturellen ausschlaggebend. So sind Wide-Area-Networks, kurz WANs, wie das Internet nicht ein einheitliches großes Netzwerk, sondern vielmehr ein Sammelsurium von vielen, kleinen Netzwerken, die unterschiedliche Profile, wie z.B. Wartung, Zeit, Rechte etc., besitzen und durch Barrieren (z.B. Firewalls) voneinander getrennt sind (siehe Abb.3.8). Zusätzlich zu den reinen technischen Mobilitätseigenschaften kommen somit noch Eigenschaften

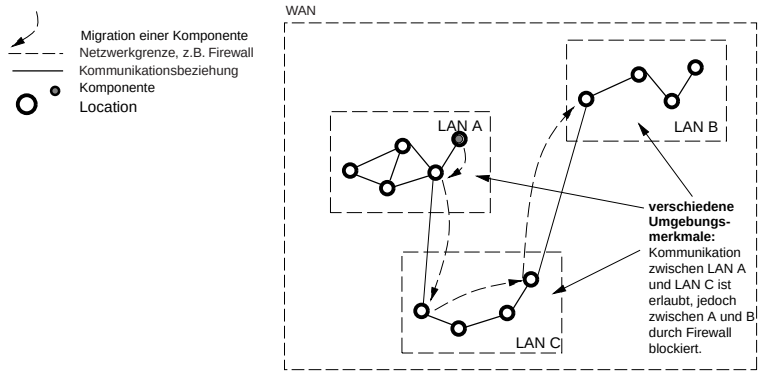


Abbildung 3.8: Mobilität in einem heterogenen Netzwerk

hinzu, die Autorisierung und Adaptivität an fremde Umgebungen betreffen [Car99a].

Die dritte Art struktureller Mobilität stellen mobile Netzwerke dar (siehe Abb. 3.9). Hierbei werden komplette, vernetzte Einheiten dynamisch von einer Konfiguration zu einer anderen verschoben. Ein Beispiel hierfür wäre ein Laptop Computer, der während des Betriebes vom Netz abgeklemmt wird und an einem anderen Netzwerk wieder angeschlossen wird.

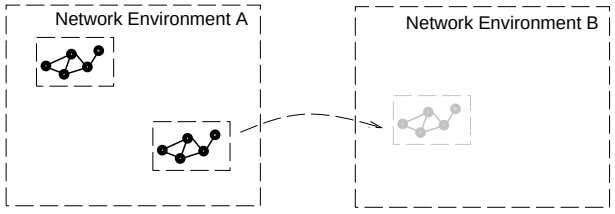


Abbildung 3.9: Mobile Netzwerke

Die drei strukturellen Mobilitätsarten können allerdings nicht separat behandelt werden, da sie sich oft und in vielen Bereichen überschneiden. So können Agenten beispielsweise sowohl statisch realisiert werden und dann mittels Laptop in verschiedene Umgebungen physisch gebracht werden oder als mobiler Agent realisiert zu dieser Lokation per Prozessmigration wandern. Der Effekt bleibt gleich, die Implementierung ist verschieden.

Mobilität muss sinnvoll und dezidiert eingesetzt werden. Der Aufwand, der durch mobile Softwarekomponenten einhergeht, muss bereits im Entwurf wohl begründet sein. In [Mas99] werden beispielsweise Anforderungen an Softwarekomponenten auf die verschiedenen Mobilitätsarten abgebildet, die auch mit verschieden hohem Aufwand verbunden sind. Die Klassifikation, die verwendet wird geht auf [CPV97, FPV98] zurück.

Umgebungsprofile: Wenn spontane Komponentensysteme in weitläufigen Netzwerken, wie z.B. dem Internet oder Intranets großer Unternehmen, angesiedelt sind, unterscheidet sich die Verteilung in einem weiteren Punkt. Im Gegensatz zur Vertei-

lung in lokalen Netzwerken, in denen sich die Verteilung und deren Modellierung ausschließlich auf die technische Erreichbarkeit zwischen Komponenten auf verschiedenen Plattformen beschränkt, kommen nun verschiedene *Umgebungsprofile* zum Tragen.

Betrachtet man als Beispiel für Wide-Area-Netzwerke das Internet, so kann man pauschal sagen, das ein konkretes Internet nicht existiert, sondern vielmehr ein Flickwerk von verschiedensten lokalen Netzwerken, die durch so genannte Gateways, also Öffnungen in andere Netzwerke, verbunden sind. Alle diese lokalen Netzwerke können sich in allen Kriterien unterscheiden. Sowohl gegebene Merkmale (z.B. Ortszeit, Durchsatz), als auch vom lokalen Verwalter gesetzte (z.B. welche Protokolle angeboten werden), können sich unterscheiden. Einen Gateway, der nicht alle Protokolle passieren lässt, bezeichnet man auch als *Firewall*, ein Netzwerk hinter einer Firewall als *Intranet*. Die Menge der ortsabhängigen und lokal administrierten Parameter eines lokalen Netzwerkes bezeichnen wir als *Umgebungsprofil*. Dies beinhaltet insbesondere, welche Protokolle die Firewall passieren dürfen und welche nicht, also welche *Rechte* eine Komponente in dem lokalen Netzwerk hat, mit anderen Komponenten in einem anderen Netzwerk über bestimmter Kommunikationsformen zu kommunizieren. Die Kommunikation selbst kann in Form von Kanälen stattfinden, die die Firewall passieren müssen, oder durch Komponentenmigration, wobei die Komponente die Firewall passieren muss.

Soll nun ein verteiltes System über ein Wide-Area-Network verteilt werden, so müssen die Anforderungen an die Umgebungsprofile aller relevanten Netze schon beim Entwurf des Systems beachtet werden. Es ist nicht ausreichend, wie bei lokalen Netzen, nur die technische Realisierbarkeit (Transparenz der Verteilung, Transaktionssicherheit etc.) zu beachten. Vielmehr müssen sämtliche Umgebungsprofile mit einbezogen werden um zu entscheiden, ob die geplante Kommunikations- und Migrationsstruktur des Systems in dem geplanten Umfeld (d.h. dem Netzwerk aus lokalen Netzwerken) installierbar ist.

Spontane Interaktion: Eines der Kern-Charakteristika der hier behandelten Systeme ist die spontane Interaktion. Mit spontaner Interaktion bezeichnen wir die Tatsache, dass die einzelnen Systemkomponenten unabhängig voneinander entworfen und entwickelt wurden und sich erst zur Laufzeit *finden* (Discovery) und *verbinden* (Bindung). Man beachte, dass es sich hierbei um Vorgänge auf der Applikationsebene handelt, im Gegensatz zu beispielsweise Ad-Hoc-Netzwerken.

Insbesondere das Finden stellt ein komplexes Problem dar: wo bisher ein Softwareingenieur den Entwurf vornimmt und die Entwurfsentscheidungen trifft, müssen diese Entscheidungen nun nach automatisierten Regeln zur Laufzeit vom System vorgenommen werden. Es reicht nicht aus, Komponenten in der erstbesten Form zu verbinden mit dem einzigen Ziel der Kompatibilität und der angebotenen Funktionen. Architektur und Systemstrukturen haben weitreichende Folgen und müssen bei der Auswahl der jeweiligen Komponente miteinbezogen werden, ebenso wie es im herkömmlichen Entwurfsprozess durch einen Menschen vorgenommen wird.

Durch die Tatsache, dass spontane Komponentensysteme keine statische Struktur besitzen, sondern einer kontinuierlichen Umkonfiguration zur Laufzeit unterworfen sind,

ergibt sich die Notwendigkeit die Architektur etwas anders zu betrachten. Das Ziel muss es sein, die *invarianten* Strukturen dieser Umkonfigurationen herauszuarbeiten und diese dann in Bezug zu den jeweiligen Konfigurationen zu bringen. Spontane Komponentensysteme eignen sich somit hervorragend für eine Konkretisierung und Illustration der Vorteile von Architekturbeschreibung: da die Architekturentscheidungen automatisiert zur Laufzeit ausgeführt werden, müssen präzise Beschreibungsformen und Abstraktionen für Architektur und deren Qualitätskriterien existieren, die automatisiert geprüft werden und quantifizierbar sein müssen.

Wir sprechen in diesem Zusammenhang von der invarianten *logischen* Architektur und den verschiedenen *technischen* Architekturen (oder Konfigurationen), zwischen denen durch Umkonfigurationsoperationen gewechselt wird.

3.2.2 Standardplattformen

Plattformen für spontane Systeme benötigen zu den oben unter *statische* verteilte Systemplattformen genannten Eigenschaften, d.h. insbesondere die Transparenz der Verteilung und die Heterogenität des Systems, weitere Eigenschaften, um das spontane und autonome Zusammenspiel verteilter Komponenten zu ermöglichen. Die Kerneigenschaften, die jede der nun im folgenden aufgezähltem Plattformen erfüllen muss, sind:

- **Bekanntmachung**, die Eigenschaft einer Komponente, die eigene Präsenz in einem neuen Netzwerk anzumelden.
- **Discovery**, das Entdecken anderer Komponenten im Netzwerk.
- **Beschreibungs-Austausch**, die Fähigkeit, die eigene Funktionalität die eine Komponente benötigt bzw. anbietet, semantisch zu beschreiben und mit anderen Komponenten auszutauschen.
- **Autonome Komposition**, das eigenständige, dynamische Anpassen auf die sich verändernde Umgebung. Dies beinhaltet dynamisches Verhalten (Aufbauen neuer Kanäle bzw. deren Abbruch) als auch mobiles Verhalten (Migration zu einer Komponente.
- **Interoperabilität**, steht für das Zusammenspiel mit anderen dynamischen Plattformen, ähnlich wie das Zusammenspiel zwischen statischen Plattformen.

Java Intelligent Network Infrastructure – JINI

Die Java Intelligent Network Infrastructure, kurz JINI [Wal, Edw99] wurde von Billy Joy und Jim Waldo entwickelt und 1999 von der Fa. Sun Microsystems freigegeben. Die Wurzeln von JINI entstammen dem akademischen Umfeld [Gel85], nämlich dem Koordinationsmodell für Tuple-Spaces in LINDA, das an der Yale University in den neunziger Jahren entwickelt wurde. JINI ist stark mit der Programmiersprache Java [AaDH00] sowie den dazugehörigen Frameworks und Plattformen wie J2EE [SHM+00] und RMI [CIP00] verwoben.

Im Gegensatz zu manchen Konkurrenten, wie zum Beispiel UPnP (s.u.), stellt es eine echte Middleware dar. Dies zeigt sich in seinem im Vergleich betrachtet großen Umfang, der zum einen für Systementwickler eine echte Entwicklungsplattform mit den dazugehörigen vorgefertigten Routinen und Diensten bedeutet, zum anderen jedoch auch ein relativ hoher Resourcenverbrauch, der nicht zuletzt durch die Java Programmiersprache begründet ist.

In einem JINI System existieren drei verschiedene Akteure: *Clients*, *Dienste* und *Lookup-Services*.

Lookup-Services (oder auch Trading-Services) sind JINI Infrastrukturdienste, die als Schwarzes-Brett bzw. zentrale Drehscheibe zum Auffinden und Anmelden von potentiell kooperierenden Dienstkomponenten dienen. Lookup-Dienste sind untereinander vernetzt und verfügen über identische Informationen.

Dienste sind Softwarekomponenten, die ein spezielles Java Interface implementieren, welches in der JINI Spezifikation definiert ist. Die Kommunikation mit einem JINI Dienst erfolgt stets mittels eines Referenzobjekts (sog. Proxy), das mittels Code-Migration zum Kommunikationspartner übermittelt wird. Dadurch bleibt die eigentliche Kommunikationsbeziehung vollkommen verborgen. Das Kommunikationsinterface zeigt sich für den Client stets gleich: ein lokales Javaobjekt.

Clients sind Softwarekomponenten, welche einen JINI-Dienst nutzen. Sie kommunizieren mit dem Dienst via den JINI Proxy.

JINI bietet *kein eigenständiges Securitymodell*, sondern baut auf dem Securitymodell des RMI-Frameworks auf. Dadurch wird zwar das Sandboxprinzip, das die Sprache Java mit sich bringt, auf verteilte Plattformen erweitert, ist aber dennoch als unzureichend zu bezeichnen[2]. Die Philosophie bei SUN scheint vielmehr zu sein, dass Security nicht durch ein vorgeschriebenes Modell abzudecken ist, sondern individuell für die jeweilige Anwendung vom Entwickler entworfen werden sollte.

Dienst Bekanntmachung Ein neuer Dienst muss sich zunächst in seinem Netzwerk registrieren. Dazu kontaktiert er seinen zuständigen Lookup-Service durch TCP Unicast; falls die Netzadresse nicht bekannt ist, mittels eines UDP Broadcast Aufrufs.

Service Discovery Das *Service Discovery*, also das Auffinden eines potentiell kooperierenden Dienstes, findet in JINI durch den sog. *Lookup-Service* statt. Ein Lookup-Service ist wiederum ein Jini Dienst, der mit einem Naming-Service auf statischen verteilten Middleware Plattformen vergleichbar ist. Jedoch liefert er die Verweise auf einen gesuchten Dienst nicht via dessen Klassenbezeichner, sondern via dessen Typ.

Beschreibungs-Austausch Die Funktionalität eines Dienstes wird unter JINI über dessen Bezeichner ausgedrückt. Es ist nicht möglich, neues Verhalten, beispielsweise

[2] So wird zum Beispiel die Autorisierung eines Dienstes nicht überprüft – jeder kann Dienste anbieten. Da beim Aufruf eines Dienstes Code heruntergeladen und anschließend automatisch ausgeführt wird, sind Viren und anderen Attacken Tür und Tor geöffnet.

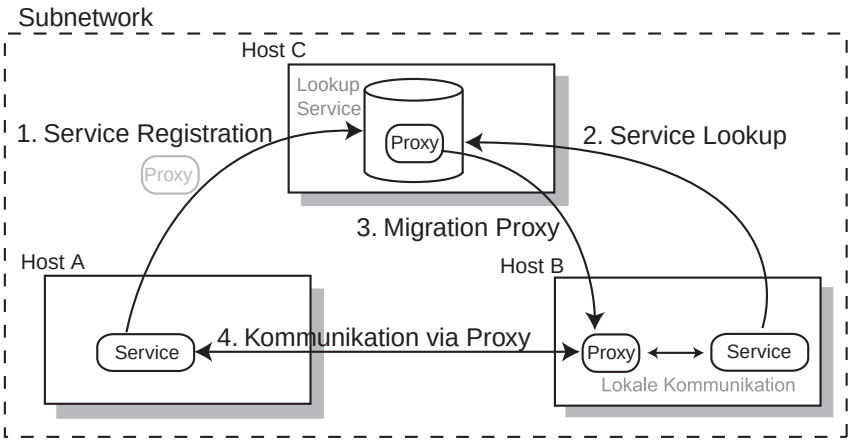


Abbildung 3.10: JINI Service Discovery Protokoll

durch mathematische Beschreibungstechniken, zu beschreiben. Dienste müssen unter JINI daher standardisierte Interfaces (z.B. DienstID = ”Printer”) besitzen. Die Dienste beziehungsweise deren Schnittstellen werden in JINI mittels so genannter Properties beschrieben. Properties können sich zum Beispiel auf die Funktionalität des Dienstes beziehen (z.B. Printer_mode = ”color | bw”), oder auch auf Ortseigenschaften, die bei mobilen Systemen wichtig sind (z.B. Lokation = ”Room1546”).

Die Standardisierung der Schnittstellen befindet sich zum momentanen Zeitpunkt im Aufbau, wobei sich verschiedene Firmen und Organisationen beteiligen. Zum Zeitpunkt dieser Arbeit ist die Standardisierung hauptsächlich auf Geräte (Drucker und Massenspeicher) beschränkt.

Autonome Komposition JINI bietet die Möglichkeit der dynamischen Verdrahtung, d.h. Assoziationen zwischen Objekten können zur Laufzeit verändert bzw. erweitert werden. Darüberhinaus bietet JINI die Möglichkeit der *Code-Migration*. Diese wird z.B. bei der Übertragung des Proxys zum aufrufenden Dienst vom Lookup-Service benutzt. Durch diesen Mechanismus einer Art von Fernsteuerung braucht das Protokoll zwischen dem Proxy und dem Dienst dem nutzenden Dienst nicht bekannt zu sein.

JINI besitzt darüberhinaus das Konzept des *Leasings*, welches insbesondere für mobile Anwendungen wichtig ist: anstatt Verweise auf Objekte auf Dauer zu vergeben, wie es bei objektorientierten Sprachen üblich ist, wird ein Verweis auf eine bestimmte Zeit *geleased*. Ist diese Zeit abgelaufen, so muss der Verweis verlängert werden. Dieses Leasing Konzept ermöglicht ein verteiltes Resourcenmanagement in einem verteilten System, welches über unsichere Verbindungen verfügt. Wird zum Beispiel in einem mobilen System ein Gerät von Hand entfernt – die entsprechenden Dienste sind also nicht mehr verfügbar – so verfallen die entsprechenden Verweise automatisch, da die Leasings nicht erneuert werden.

Interoperabilität JINI ist eng mit der Programmiersprache Java verbunden. Durch die Tatsache, dass Java semi-interpretiert ist und in einer eigenen virtuellen Maschine

abläuft, ist JINI auf allen JAVA Interpretern der zweiten Generation (Java 2.0) lauffähig. Die starke Verbreitung und die verschiedenen Anwendungsplattformen – vom Mainframe bis hin zur Smartcard JVM – ermöglichen äußerst heterogene Systeme.

JINI unterstützt Koordination innerhalb verteilter, spontaner Systeme durch sog. *Distributed Events*, sowie *Transaktionen*.

Universal Plug and Play – UPnP

Die *Universal Plug and Play* (UPnP) [Con00], ein Bestandteil der .NET Technologie von Microsoft, verfolgt einen anderen Ansatz als JINI. Im Gegensatz zu der eng mit der Programmiersprache Java verbundenen Jini-Infrastruktur ist UPnP gänzlich unabhängig von Programmiersprache und Umgebung, sondern setzt auf die standardisierten Internet Protokolle, wie Hypertext Transfer Protocol (HTTP) [FGM+97] und die Extensible Markup Language (XML) [BPSM97a], indem es das beide beinhaltende Simple Object Access Protocol (SOAP) [BEK+00] verwendet. Der Ansatz ist daher von grundauf weniger eine komplette Middleware wie im Falle von Jini, sondern vielmehr eine Protokollerweiterung, die vor allem auf Schlankheit setzt.

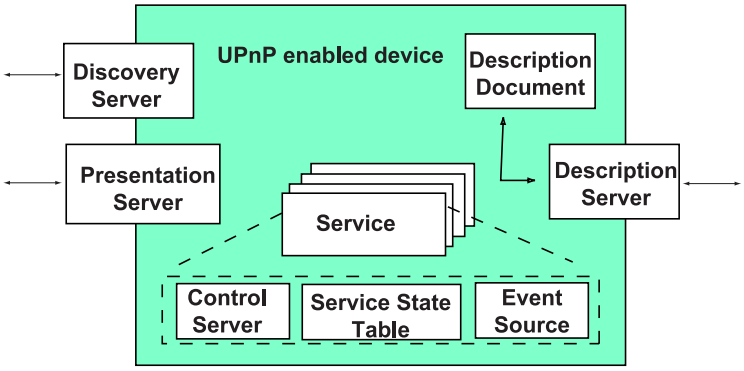


Abbildung 3.11: Aufbau eines UPnP enabled Device

Der zentrale Aktor innerhalb eines UPnP Szenarios ist das sog. *UPnP enabled Device*, also ein UPnP-fähiges Endgerät. Ein solches Device fungiert als Behälter für verschiedene Dienste, die dadurch den Typ des Devices bestimmen. Der Behälter selbst beinhaltet neben der Menge an Diensten die folgenden Komponenten:

Discovery Server – Die Serverkomponente innerhalb des Discovery Protokolls. Im Falle von UPnP wird das Simple Service Discovery Protocol (SSDP) [GCL+99] verwendet.

Presentation Server – stellt eine Nutzerschnittstelle auf XML Basis zur Verfügung, die mittels eines XML Browsers (z.B. Webbrowser) genutzt werden kann.

Description Server – übermittelt das so genannte Description Document, welches das Device beschreibt.

Alle Softwarekomponenten, die mit einem UPnP Device kommunizieren wollen, müssen bestimmte Schnittstellen implementieren, die man als *User Control Point* (UCP) bezeichnet. Grob gesagt handelt es sich hier um die Client-Gegenstücke zu den Serverkomponenten des UPnP Device: Discovery Client, DescriptionClient, Visual Navigation (Presentation Client) und Event Sink. Der so genannte *Rehydrator* ist schließlich eine Brücke[3], welche die APIs der Komponente auf die UPnP Protokolle abbildet.

Dienst Bekanntmachung Bei UPnP beginnt ein neues Gerät, das eine Umgebung betritt mit einem DHCP Broadcast um einen Server zu finden. Anschließend registriert sich das Gerät bei vordefinierten Server durch das Übersenden einer *Bekanntmachungs-Nachricht*, sowie der dem Gerät und seinen Diensten entsprechenden Beschreibung.

Service Discovery Um einen gewünschten Dienst zu finden, wird zunächst ein Server gesucht, der dann die IP Adresse übermittelt. Ist kein Server vorhanden, übernimmt das Device selbst die Serverrolle. Das Suchen nach einem geeigneten Dienst findet selbst wiederum in Form des Simple Service Discovery Protokolls (SSDP) statt, bei welchem ein Universal Resource Locator (URL) zurückgegeben wird, via den dann Beschreibungen des Dienstes auf Basis von XML via HTTP Protokoll angefordert werden können.

Beschreibungs-Austausch Bei der Beschreibung handelt es sich um eine vordefinierte XML Struktur, die grundlegende Eigenschaften, wie Versionsnummer etc. sowie die Schnittstellen der Dienste umfasst. Es handelt sich jedoch um eine isolierte Beschreibung des Gerätes ausserhalb jeglichen Kontextes, ähnlich wie eine WSDL Beschreibung (siehe Abschnitte 9.2).

Autonome Komposition UPnP bietet die Möglichkeit der dynamischen Konfiguration bezüglich der Verbindungsstruktur zwischen den einzelnen Geräten. Es bietet im Gegensatz zu beispielsweise JINI keine Möglichkeit der Code-Migration.

Interoperabilität Generell gilt zu sagen, dass UPnP wesentlich stärker auf Protokollen basiert als auf Middleware. Die jeweiligen Server und Clientkomponenten müssen zu den jeweiligen Protokollen implementiert werden und werden nicht von der Middleware zur Verfügung gestellt. Dies hat den Vorteil, dass Systeme schlanker gehalten werden können, jedoch mit einem höheren Implementierungsaufwand verbunden sind.

.NET: Webservices

Innerhalb des *.NET* Frameworks von Microsoft befinden sich eine Vielzahl von Ansätzen zur Realisierung von Wide-Area-Systemen. Ein Ansatz ist das bereits vorgestellte

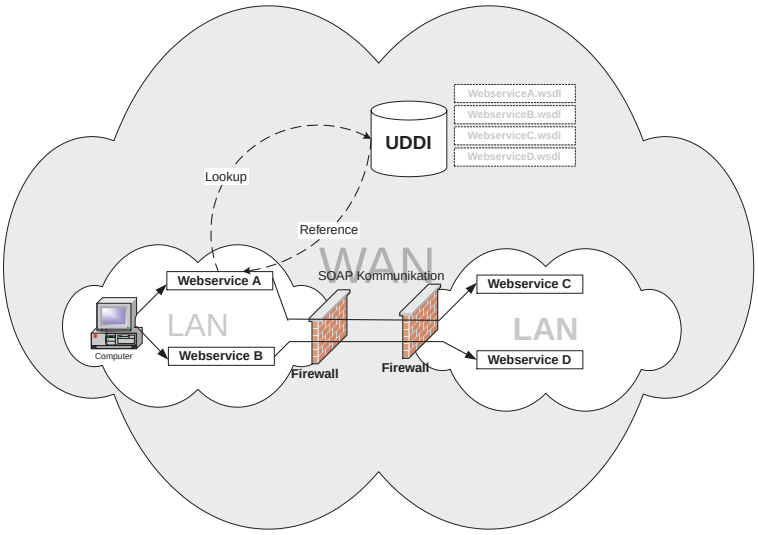


Abbildung 3.12: Webservices unter .NET

Universal Plug and Play (UPnP), das die spontane Interaktion von mobilen Kleinstgeräten unterstützt. Der innerhalb von .NET propagierte Ansatz für Internet-basierte Wide-Area-Anwendungen sind so genannte *Webservices*, eine Weiterentwicklung der Active Server Pages (ASP), die eine Verwebung von lokalen Systemen und Internet basierten Dienste ermöglicht. Nach Planung von Microsoft wird es zukünftig nur noch zwei Anwendungstypen geben:

Lokale Anwendungen, bei denen der Zugriff über einen Browser geschieht und die nur Zugriff auf lokale Resourcen benötigen.

Netzbasierte Anwendungen, die intensiven Gebrauch von Diensten im Internet machen und durch Kombination dieser Dienste eine Vielzahl von neuen Anwendungen möglich machen.

Netzbasierte Anwendungen bestehen aus einem relativ schlanken Gerüst, das benötigte Funktionen in Form von Webservices bei Bedarf aus dem Internet abrufen kann.

Dienst Bekanntmachung Komponenten – unter .NET *Webservices* genannt – werden mittels UDDI (Universal Description, Discovery and Integration) [Con01] vermittelt. UDDI ist ein hierarchisch strukturierter Trading-Service, der analog zu DNS-Servern, im Internet verteilt und via eine URL (Universal Resource Locator) erreichbar ist. Will sich ein neuer Dienst anmelden, so kontaktiert er einen beliebigen UDDI-Server und übermittelt seine Dienstspezifikation in Form eines WSDL (Web Service Description Language) Dokuments. UDDI Server replizieren dann ihre Information und verteilen so die WSDL Information.

[3] Zur Erklärung des Designpatterns Brücke siehe [G+94]

Service Discovery UDDI (Universal Description and Discovery Interface) stellt einen unabhängigen Standard dar, der nicht .NET spezifisch ist. Allerdings findet UDDI in .NET die wichtigste Anwendung. Der Ablauf ist sehr ähnlich zu dem oben beschrieben JINI-Lookupservice. Allerdings wird kein Proxy auf dem UDDI Server gespeichert, wie im Falle des Lookupservice, sondern ein WSDL (Webservice Description Langauge) Dokument. Darüberhinaus sind UDDI Server statisch im Internet verfügbar und unter einheitlichen Adressen erreichbar, analog zu einem DNS (Domain Name Server), der das Naming regelt.

Beschreibungs-Austausch Schnittstellen und nur diese, also nicht wie bereits bei den existierenden Ansätzen bemängelt auch die Verknüpfungen, werden unter .NET durch die Webservice Description Language (WSDL) ausgedrückt. Durch WSDL wird, analog zu einer IDL (siehe Abschnitt 2.2), die Schnittstelle und deren Datentypen auf Basis einer allgemeinen Datenbeschreibungssprache, nämlich XML, definiert. WSDL bietet keine Möglichkeit die Verbindungsstrukturen zwischen den einzelnen Diensten oder deren Verhalten zu definieren.

Auf einen Vergleich zwischen WSDL und der im Rahmen dieser Arbeit entwickelten SADL wird in Abschnitt 9.2 noch einmal gesondert eingegangen.

Autonome Komposition .NET stellt einen sehr weit gefassten Rahmen für zukünftige Anwendungen dar. Bereits jetzt sind Eigenschaften wie dynamische Verbindungsstrukturen und Code-Migration in Form von so genannten ActiveX Controls vorhanden.

Interoperabilität Obwohl .NET ein Framework aus und für die Windows-Welt darstellt, sind die entscheidenden Mechanismen, also UDDI als Discovery Mechanismus, WSDL als Beschreibungssprache und SOAP als Kommunikationsmechanismus, bewusst allgemein gehalten. Durch die ausschließliche Verwendung von allgemeinen, unabhängigen Standards ist es möglich, die hier vorgestellten Mechanismen des .NET Frameworks unter anderen Plattformen und Sprachen zu realisieren und die dadurch entstehenden Webservices in bestehende, heterogene Umgebungen einzubinden.

3.3 Diskussion

Nach der Vorstellung der verschiedenen Plattformen spontaner Komponentensysteme und deren Eigenheiten gegenüber statischen verteilten Systemen wird nun ein knappes Fazit gezogen und der Zusammenhang mit der Stoßrichtung dieser Arbeit erläutert.

Was ist an der Entwicklung spontaner Komponentensysteme speziell?

Die Entwicklung spontaner unterscheidet sich von jenem statischer Komponentensysteme unter anderem darin, dass der Entwickler wenig über die Struktur und einzelnen

Instanzen des Systems zu einem Laufzeitpunkt weiß. Insbesondere ist nicht bekannt, welche Instanzen zu einem Zeitpunkt am System teilnehmen, wie sie verbunden sind und ob die Verbindungsstruktur aufgrund der Umgebungsprofile erlaubt ist.

Die Komposition der einzelnen momentan verfügbaren Systemkomponenten wird vom System *autonom zur Laufzeit* vorgenommen. Daher müssen die Entwurfsentscheidungen, die normalerweise in der Entwurfsphase von den Entwicklern getroffen und informell beispielsweise in Prosa begründet werden, formalisiert und automatisch vom System ausgewertet werden. Es ist daher bei der Entwicklung spontaner Komponentensysteme wichtig, über die Software-Komponenten hinaus begleitend gewisse Abstraktionen und Beschreibungen zu entwickeln, die dann vom System zur Laufzeit ausgewertet werden können, um die richtige Entscheidung zur Komposition zu treffen.

Warum spielt Software-Architektur hier eine besondere Rolle?

Die oben beschriebene Metainformation, die begleitend zum Softwarecode entwickelt werden muss, sollte über die momentan existierende, wie beispielsweise WSDL, hinaus gehen. Die meisten solcher Beschreibungen sind auf den isolierten Softwarebaustein, meist sogar nur dessen Schnittstellen, beschränkt. Es wird aber nicht die Architektur, also das Zusammenwirken der verschiedenen Bausteine und die daraus resultierenden Konsequenzen beschrieben. Betrachtet man zur Komposition jedoch die einzelnen Bausteine nur isoliert, so kann keinerlei Aussage über Architekturkriterien, wie beispielsweise die Stabilität einer Konfiguration etc., getroffen werden.

Betrachten wir ein Beispiel. Angenommen es existieren zwei Softwarekomponenten, *A* und *B*, die beide eine für ein System benötigte Funktionalität f_1 anbieten. Allerdings unterscheiden sich beide Komponenten insofern, dass Komponente *A* neben f_1 noch zwei weitere Funktionen f_n und f_m anbietet, Komponente *B* lediglich eine weitere Funktion f_k anbietet. Betrachtet man nun das System, das eine Komponente mit f_1 benötigt, so sieht man an dessen Architektur, dass die Funktion f_k im System vorkommt, jedoch schon von einer existierenden Komponente versorgt wird. Die Funktionen f_n und f_m kommen im System jedoch nicht vor. Würde man jetzt eine Anfrage mit einer reinen Schnittstellenbeschreibung an die Middleware schicken, also etwa “get component that implements function f_1” so kann das System lediglich beide Komponenten *A* und *B* zurückgeben, da aufgrund der reinen Interface-Beschreibung beide Komponenten gleich geeignet sind. Wäre jedoch eine Beschreibung der Konfiguration existent, so könnte das System aus dieser feststellen, dass Komponente *B* besser geeignet ist, da die Funktion f_k anschließend redundant im System verfügbar wäre und somit die Stabilität erhöhen würde. Diese jeweiligen Auswahlkriterien könnten beim System vorher spezifiziert werden.

Fazit

Obwohl dieses kleine Beispiel natürlich trivial ist, zeigt es, dass spontane Systeme ein sehr gutes Anwendungsfeld für Softwarearchitektur und deren Beschreibungstechniken sind. Da hier viele Kompositions- und Architektur-Entscheidungen automatisiert

getroffen werden müssen, ist eine eindeutige, formalisierte Architekturbeschreibung notwendig.

Ziel dieser Arbeit ist es, Techniken zur Verfügung zu stellen, die, in einen entsprechenden Prozess[4] eingebettet, die entsprechende Beschreibungen erzeugen. Hierbei steht die Architektur im Vordergrund, obgleich auch der sinnvolle Einsatz von Techniken zur Verhaltensbeschreibung aufgezeigt wird.

[4]Dieser Prozess steht allerdings nicht im Rahmen dieser Arbeit.

Teil II

Abstraktionen

KAPITEL

VIER

Modellbildung

Dieses Kapitel behandelt eine der grundlegenden Voraussetzungen für die Beschreibung und Modellierung eines Softwaresystems: die Modellbildung. Durch Modellbildung wird ein Sachverhalt abstrahiert und dieser damit vereinfacht. Auf Basis des Modells werden die Spezifika, die analysiert werden sollen, beispielsweise durch mathematische und logische Mittel ausgedrückt, und deren Konsequenzen abgeleitet. Die Eignung des Modells kann durch zwei Merkmale [Mul94] bestimmt werden:

- **Mächtigkeit:** *Welche Arten von Eigenschaften können ausgedrückt werden ? Welche Fragestellungen können somit beantwortet werden ?*
- **Aufwand:** *Wie aufwändig sind die Lösungen und deren Beschreibung ?*

Beide Merkmale besitzen praktische Relevanz. Zum einen ist die Frage nach lösbaren Problemen – und auch nach den unlösbaren – von großer Wichtigkeit. Zum anderen ist der Aufwand einer Lösung gerade in der praktischen Anwendung von Bedeutung: Ist eine Lösung mit dermaßen hohem Aufwand verbunden, dass sich der Aufwand nicht lohnt, d.h. einen gewissen Tradeoff-Punkt überschreitet, so ist es meist sinnvoll, einen anderen Ansatz zu wählen. Ein Beispiel hierfür ist die Verifikation von Programmen: Ist es oftmals noch möglich, für wohlstrukturierte Systeme mit übersichtlicher Komplexität, wie zum Beispiel eingebettete Systeme oder Netzwerkprotokolle, eine Verifikation durchzuführen, so ist der Aufwand bei den meisten kaufmännischen Software-Systemen nicht mehr vertretbar. Hier ist Testen oder auch bereits das systematische Entwickeln der lohnendere Weg.

Ein weiterer Aspekt ist die Interoperabilität. Die Implementierung eines Systems kann auf verschiedenste Weisen erfolgen, beispielsweise bezüglich der Implementierungssprache, der Plattform des Rechners bis hin zur verwendeten Middlewaretechnologie. Um ein System möglichst unabhängig von diesen Faktoren entwerfen zu können, sollte das Modell, auf dessen Basis das System entworfen wird, möglichst *allgemein*, d.h alle Plattformen abdeckend, sein.

Der Hauptzweck des hier vorgestellten Modells liegt darin, *verschiedene Systemstrukturen* zu einer *Funktionalität* zu gruppieren. Die so definierte Dienstspezifikation, die keine Strukturinformation enthält, wird auf eine Menge von Komponentennetzwerken (Konfigurationen) abgebildet, welche die bis dato fehlende Strukturinformationen beinhalten. Da das Modell demnach vornehmlich von *Strukturen* abstrahiert, kann man es auch als *Architekturmodell* bezeichnen.

4.1 Motivation für partiell formalen Ansatz

Am Anfang der Entwicklung eines Modells stellt sich natürlich die Frage, ob ein formaler oder ein informeller Ansatz gewinnbringender ist. Obwohl hier keine generelle Antwort gegeben werden kann, sollen hier einige der Gründe aufgeführt werden, die zu dem hier gewählten Ansatz eines formalen Basismodells in Kombination mit einem informellen Engineeringmodell geführt haben.

- Informelle Abstraktionen sind schwer genau zu charakterisieren. Insbesondere ist die Abbildung eines informellen Modells auf ein anderes schwierig. Letztendlich wird bei informellen Modellen für deren Abbildung jeweils ein Experte des Modells benötigt. Eine präzise Notation, die einen solchen Experten obsolet macht, ist nicht möglich.
- Informelle Modelle begrenzen die analytischen Möglichkeiten. Insbesondere sind Beweise bestimmter gewünschter Eigenschaften nicht möglich. Ein rein informelles Modell würde diese Möglichkeit also verbauen.
- Eine zumindest semi-formale Abstraktion, zwingt den Nutzer zu einem präzisieren Denken, als es bei einem vollkommen informellen Modell notwendig ist.
- Die meisten existierenden Formalismen für verteilte Systeme, wie zum Beispiel CSP[Hoa85] oder der π-Kalkül[MPW92], beschreiben Systeme mit Basisentitäten, wie Prozesse und Kanäle. Für den praktikablen Einsatz bei der Architekturbeschreibung sind jedoch vordefinierte Strukturen höherer Ordnung, wie Komponenten oder Konnektoren, notwendig. Ein Verfahren, das die Basisentitäten direkt nutzt, ist also nicht praktikabel.
- Die komplette Formalisierung großer Systeme ist nicht nur inpraktikabel, es stellt sich auch die Frage, ob sie notwendig ist. Es handelt sich vielmehr um einzelne Subsysteme innerhalb eines Systems (z.B. die Authentisierung etc.), die gewinnbringend formalisiert werden sollte. Eine “partielle Formalisierung” [JW96], die es erlaubt, bestimmte Teile bei Bedarf zu formalisieren, ist oft gewinnbringender als die komplette.
- Eine formale Basis erlaubt, die Begriffe und Entitäten präzise darzustellen. Informelle Modelle besitzen noch nach langer Anwendung Ambiguitäten, die unter Umständen zu Problemen bei der maschinellen Verarbeitung führen können.
- Formale Modelle dürfen nicht mit formaler Notation gleichgesetzt werden [Par96, Par94]. Formale Modelle helfen, Systeme präzise und redundanzfrei auszudrücken, sind aber schwer zu lesen und nachzuvollziehen. Formale Modelle ohne eine lesbare und nachvollziehbare Notation sind jedoch für die hier adressierten Zwecke (Modellierung) nutzlos.
- Formale Modelle behandeln manche Charakteristika, wie beispielsweise Mobilität, oft implizit, da sie auf Grundeigenschaften zurückgeführt werden. Dies ist sinnvoll, um möglichst wenige Axiome zu besitzen und so die Verifikation einfacher zu halten. Zum Modellieren und entwerfen ist es jedoch von Vorteil solche Eigenschaften explizit darstellen zu können.

- Eine eventuell spätere Erweiterung des Modells um zusätzliche Informationen, die das Verhalten von Schnittstellen und Diensten beschreibt (siehe Abschnitt 12.3), ist nur mit formalen Techniken möglich. Daher sollte diese Option nicht verbaut werden.

Diese Argumente führten zu der Entscheidung, einen partiell formalen Ansatz zu verfolgen: ein Engineeringmodell stellt Strukturen höherer Ordnung explizit zur Verfügung, wie sie für den praktikablen Entwurf benötigt werden. Das Engineeringmodell wird abgebildet auf ein formales Basismodell, was eine Formalisierung von Einzelaspekten bei Bedarf ermöglicht. Dies macht eine praktikable Modellierung mit dem Engineeringmodell und eine formale Spezifikation in bestimmten Bereichen möglich, sofern diese gewünscht bzw. sinnvoll ist. Darüberhinaus ist das Engineeringmodell durch die formale Basis eindeutig und kompakt beschrieben und lässt sich damit eindeutig auf andere Modelle abbilden.

4.2 Zielsetzung

Der Begriff *Software-Architektur* wird weitestgehend als die abstrakte Dekomposition eines Systems in seine Subsysteme oder Komponenten, verbunden mit deren Interaktionsbeziehungen und Rollen, angesehen. Die Architektur eines Systems hat direkten Einfluss auf Qualitätsfaktoren des Systems, wie zum Beispiel dessen Wiederverwendbarkeit, Leistungsfähigkeit oder Verlässlichkeit. Jedoch ist der Grad an Abstraktion der Architektur abhängig vom jeweiligen Entwicklungsstand innerhalb des Entwicklungsprozesses: ist in frühen Phasen die Abstraktion stärker und steht die logische Dekomposition im Vordergrund (z.B. abgeleitet von bestimmten Nutzungsfällen), so wird diese Abstraktion während der Entwicklung zunehmend konkreter, bis sie schließlich in der technischen Implementierung endet. Dort werden schließlich die Subsysteme und Komponenten durch konkrete Code-Objekte (z.B CORBA etc.) repräsentiert sowie deren Beziehungen durch Aggregation und Assoziation. Man spricht in diesem Zusammenhang von *logischer Architektur* in den frühen Phasen und *technischer Architektur* in den letzteren.

Herkömmliche Architekturabstraktionen, wie beispielsweise das Komponenten & Konnektoren-Modell haben in dieser Abbildung von logischen auf technische Architekturen Schwächen. Hier wird durch die Nähe der Abstraktion (Komponenten u. Konnektoren) auf die letztendliche Implementierungsabstraktion (z.B. Softwareobjekte und Objekt-Request-Broker) die technische Architektur schon sehr früh eingeschränkt. Obgleich bei statischen Systemen dies nicht allzu gravierende Auswirkungen hat, ist es bei spontanen Systemen, die im allgemeinen zu einer logischen Architektur eine Vielzahl von technischen Architekturen besitzen (siehe Abbildung 4.1), wichtig, die logische von der technischen Architektur strikt zu trennen und deren Zusammenhang sauber zu fundieren. Gerade die Abbildung von logischer Architektur auf die Menge der passenden technischen Architekturen, die späteren Konfigurationen, muss durch das Architekturmodell abgedeckt sein. Es wird daher eine Modellbildung benötigt, die *logische* Einheiten für frühe Phasen sowie *technische* in den späteren unterstützt. Dar-

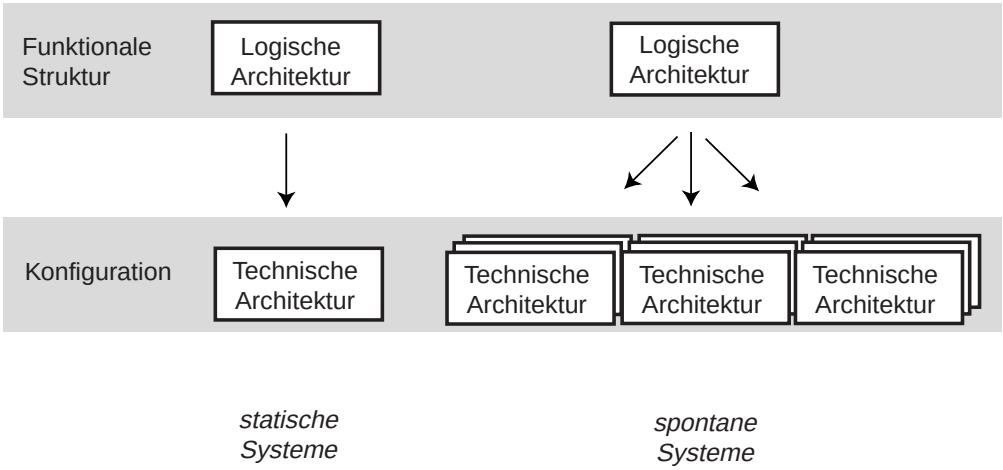


Abbildung 4.1: Abbildung von logischer Architektur auf technische Architektur bei statischen und spontanen Systemen

überhinaus sollte eine eindeutige, formal begründete Abbildung der ersteren auf die letzteren existieren.

Das hier vorgestellte Modell hat daher folgende Kerneigenschaften:

- **Mobiles** und **dynamisches** Verhalten zu modellieren,
- **Umgebungsprofile** in den Entwurf mit einzubeziehen,
- **Spontane Interaktion** sicherer zu gestalten,
- Kombination von **logischen und technischen Architekturelementen** und deren Beziehungen formal zu fundieren und
- **Interoperabilität** auf gängigen Plattformen zu gewährleisten.

Die Eigenschaften wurden bereits teilweise in Abschnitt 3.2 beschrieben. Zur besseren Übersichtlichkeit werden sie hier jedoch noch einmal kurz aufgegriffen.

Mobiles und dynamisches Verhalten

Ein zentraler Aspekt innerhalb der Modellbildung soll das Ausdrücken *mobilen und dynamischen* Verhaltens.

Dynamisches Verhalten, das heist Veränderung der Komponentenmenge und -Verdrahtung, wird in verschiedenen formalen Kalkülen modelliert. Beispiele hierfür sind der π-Kalkül [MPW92] sowie seine verschiedenen Derivate, wie zum Beispiel der Join-Kalkül [Fou98]. Ein anderer, denotationeller Ansatz ist das auch in dieser Arbeit verwendete Systemmodell dynamisches FOCUS [HS96]. Neben dem Aspekt der Dynamik behandeln wir auch den Aspekt der Mobilität. Von *mobilem* Verhalten

spricht man, wenn man sowohl Persistenz der Komponenten, als auch explizite Modellierung der Lokationen, die die Wirtsplattformen der Komponenten darstellen, im Modell vorfindet. Obwohl mobiles Verhalten in manchen Formalismen, wie dem π-Kalkül, implizit durch die Menge der erreichbaren Prozesse vorhanden ist, ist explizites Adressieren einer Lokation nicht vorgesehen: Ein Prozess kann nicht explizit zu einer Lokation *l* migrieren.

Mit der Persistenz von Komponenten wird im Zusammenhang von Mobilität die Eigenschaft bezeichnet, dass es sich bei der Komponente vor und nach dem Migrationsvorgang um dieselbe Instanz handelt. Dies bezeichnet man auch als *Prozessmigration*. Diese stellt im Gegensatz zur Code-Migration sicher, dass der Zustand einer Komponente bei der Migration nicht verloren geht. Mobilität wird durch grundlegende Formalismen wie z.B. dem Ambient-Kalkül [Car97] oder dem verteilten Join-Kalkül [Fou98], die wiederum beide auf den π-Kalkül zurückzuführen sind, modelliert.

Umgebungsprofile

In dem hier angestrebten Modell soll es möglich sein, *Umgebungsprofile* explizit in den Entwurf einzubeziehen. Mit Umgebungsprofilen bezeichnen wir die Abstraktion einer Umgebung, mit der die Einflüsse der Umgebung auf das System und dessen Struktur *explizit* ausgedrückt werden können.

Ein Umgebungsprofil legt beispielsweise die Kommunikationsrechte fest, das bedeutet, ob und wie eine Komponente mit anderen Komponenten kommunizieren kann. Durch Umgebungsprofile sollen Eigenschaften beschrieben werden, wie Firewalls (bestimmte Kommunikationsarten werden abgeblockt) oder Wirtsrechner (engl. Hosts), die Zugriffsrechte (z.B. Massenspeicherzugriff) der darauf angesiedelten Komponenten einschränken. Essentiell hierbei ist, dass ein Umgebungsprofil *explizit* modelliert werden kann und nicht implizit, wie es beispielsweise bei dem π-Kalkül der Fall ist. Gerade im Entwurf ist die explizite Ausdrucksmöglichkeit notwendig, um Eigenschaften zu thematisieren.

Das in diesem Modell verwendete Konzept von Umgebungsprofilen (Sandboxen, siehe Abschnitt 6.1.4) lehnt sich an die im Ambient-Kalkül definierten Ambients an, wobei die grundlegende Kommunikationsform des Ambient-Kalküls die Migration ist. Wir verwenden die Kommunikation via Kanäle, da dies eine geeignetere Abstraktion verteilter Komponentensysteme ist.

Abbildung logischer Architektur auf technische Konfigurationen

Mobiles und dynamisches Verhalten erschweren eine klare Spezifikation, da wechselnde Strukturen im Gegensatz zu statischen kaum ausdrückbar sind. Es ist daher erstrebenswert, die statischen, invarianten Strukturen eines dynamischen Systems zu isolieren und getrennt zu spezifizieren. Diese invarianten Teile des Systems können dann unter Hinzunahme der momentan verfügbaren Komponenten und Umgebungsprofile auf die verschiedensten tatsächlichen Realisierungen abgebildet werden (siehe Abbildung 4.2).

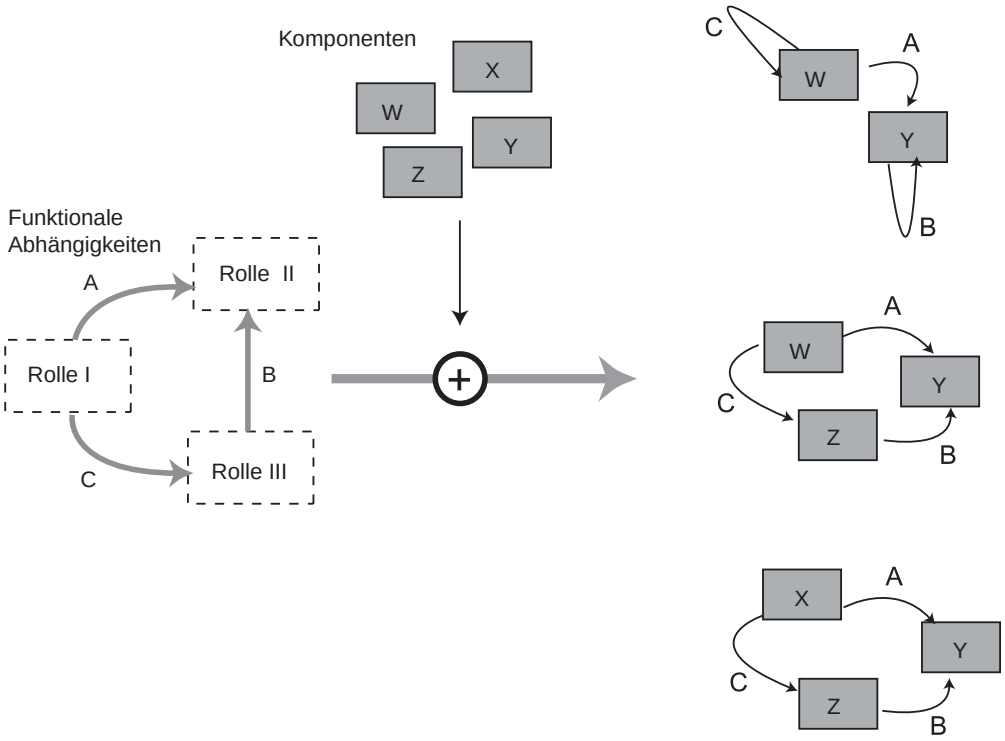


Abbildung 4.2: Statische funktionale Abhängigkeiten (A, B und C) werden durch Hinzunahme von Komponenten (W,X,Y und Z) auf drei Konfigurationen abgebildet.

Diese Trennung zwischen *invarianter logischer Architektur* und der dazugehörigen Menge an *technischen Konfigurationen*, die diese logische Architektur realisieren, ist essentiell für das hier vorgestellte Modell. Es ist möglich, die logische Architektur, die so genannte *Dienstarchitektur*, isoliert zu spezifizieren und somit die invarianten funktionalen Abhängigkeiten zwischen später das System realisierenden Komponenten auszudrücken. Unter Hinzunahme der dem System zu einem bestimmten Zeitpunkt zur Verfügung stehenden Komponenten, von denen bekannt ist, welche Dienste sie anbieten, kann die Dienstarchitektur auf eine Menge von technischen Konfigurationen abgebildet werden, die diese funktionalen Abhängigkeiten erfüllen.

Diese Abbildung kann in Middleware-Plattformen sogar zur Laufzeit automatisiert ausgeführt werden (siehe hierzu auch Abschnitt 8.2.7). Damit wird die Auswahl der jeweiligen Komponenten nicht wie bisher ausschliesslich aufgrund deren Funktionalität getroffen, sondern auch aufgrund ihrer strukturellen Eignung bezüglich der Architektur des Systems. Das bedeutet, dass ein Trading-Service, der das hier beschriebene Modell realisiert, bei Anfrage nach einer Komponente, die eine Funktion *F* anbietet, nicht nur eine beliebige Komponente mit angebotener Funktion *F* zurückgibt. Vielmehr wählt er die Komponente *K* aus, die *F* anbietet und optimal in die Architektur des anfragenden Systems hineinpasst, um beispielsweise Architektur-Qualitätskriterien, wie Stabilität oder Redundanz, zu optimieren. Siehe hierzu auch Abschnitt 8.2.7.

Interoperabilität

Mit *Interoperabilität* wird schließlich die Integration verschiedener Komponententypen innerhalb eines Systems bezeichnet. Da ein System, wie es in dieser Arbeit behandelt wird, als ganzes modelliert werden muss, müssen auch innerhalb des Modells alle enthaltenen Komponententypen und deren Interaktion ausdrückbar und modellierbar sein. Es wird daher eine präzise Notation benötigt, die mächtig genug ist, um:

- Verschiedene Basistypen von Komponenten unabhängig von deren Implementierung auszudrücken.
- Verschiedene technische Plattformen, wie z.B. Middleware etc. und deren Komponenten bzw. Interaktionen, auszudrücken.
- Verschiedene Kommunikationsarten (z.B. lokal oder entfernt, Bus- oder Peer-to-Peer) und -typen (z.B. asynchron oder synchron) auszudrücken.

Man beachte, dass manche Modelle einen systemweiten Kommunikationsmechanismus propagieren, der durch eine Kommunikationskomponente realisiert wird, wie zum Beispiel einen Object Request Broker (ORB) oder einen Kommunikationsbus [Rum96]. Da hier jedoch Interoperabilität und Heterogenität innerhalb des Systems als zentrale Aspekte betrachtet werden sollen, würde ein solcher Ansatz einige Nachteile durch die Vereinheitlichung des Kommunikationsmediums mit sich bringen. Es ist die weit verbreitete Auffassung, dass eine Systemarchitektur die Hauptkomponenten des Systems und deren strukturelle Beziehungen beinhalten soll, um die Gesamtheit des Systems zu beschreiben. Neben diesen strukturellen Informationen ist man aber vor allem auch an der Gesamtfunktionalität des Systems und dessen Interaktionsformen, die zwischen diesen Systemteilen ablaufen, interessiert. Eine Architektursicht sollte darüberhinaus eher das System skizzieren, als die Struktur in voller Detailschärfe wiederzugeben.

Präzision & Einheitlichkeit

Ein Architekturmodell, sollte Eigenschaften wie Struktur und Interaktionsverhalten auf abstrakter Ebene besitzen sowie unabhängig von den jeweiligen Implementierungsdetails, wie zum Beispiel verwendete Middleware oder Implementierungssprache, sein. Ferner sollte die Abstraktion so gewählt sein, dass sie zum einen intuitiv und einfach nachzuvollziehen ist, und damit nicht zu fern von populären Modellierungssprachen wie UML [FS97] oder ROOM [SGW94] ist.

Da Elemente der Softwarearchitektur potentiell wiederverwendbare Elemente eines Softwaresystems sind, muss ein Architekturmodell die Konzepte zur Komposition solcher Elemente unterstützen. Eine Folgerung daraus ist, dass die Elemente eines Architekturmodells, also z.B. die Komponenten, solcherart spezifiziert werden müssen, dass es möglich ist, die Eignung einer Komponente für einen gewissen Kontext eindeutig festzustellen. Das bedeutet insbesondere, dass es notwendig ist, nicht nur die Funktionalität eines Elements zu beschreiben, die es nach aussen *anbietet*, sondern vielmehr

auch diejenige Funktionalität, die das Element von seiner Umgebung *benötigt*. Darüberhinaus muss auch beschrieben werden wie, d.h. durch welche Interaktionsbeziehungen, diese Funktionalität bereitgestellt bzw. benötigt wird.

Kompositionalität bedeutet, dass es möglich sein muss, die gerade beschriebenen Eigenschaften eines Elements aus denen seiner Einzelteile, d.h. Subelemente und deren Beziehungen, abzuleiten.

Die Strukturen in spontanen Komponentensystemen besitzen einen grundlegenden Unterschied zu statischen Komponentensystemen, wie z.B. CORBA oder COM+: Der Zugriff auf einen Kommunikationspartner erfolgt nicht durch dessen Bezeichner, sondern via dessen Typ. Dies bedeutet, dass die grundlegende Kommunikationsstruktur und deren Teilnehmerkreis nicht von vornherein festgelegt ist, sondern lediglich der funktionale Ablauf. Im folgenden wird die konkrete Kommunikationsstruktur als *Konfiguration* (engl. Configuration), der funktionale Abhängigkeit als *Dienstarchitektur* (engl. Service-Architecture) bezeichnet. Es gilt also, dass in spontanen Systemen die Dienstarchitektur fest ist, jedoch durch eine Menge von möglichen Konfigurationen realisiert werden kann. Soll demnach die Architektur eines spontanen Systems festgelegt werden, so müssen beide Ebenen, die Dienstebene und die Konfigurationsebene, beschrieben werden.

Man beachte darüberhinaus, dass die typischen Eigenschaften und Vorteile komponentenorientierter Softwarearchitekturen, als da sind Wiederverwendung, Komposition und Kapselung, auf *beiden* Ebenen gegeben sind: Auf der Dienstebene werden *Dienste* wiederverwendet und komponiert, auf der Konfigurationsebene *Komponenten*.

4.3 Fallbeispiel: Ad-Hoc Systeme unter JINI

Am Lehrstuhl für Software & Systems Engineering der TU München wurde im Rahmen eines mittelgroßen Studentenprojektes ein “Mobile Office” System entwickelt [FGH+99] . Hierbei galt es, durch verschiedene Geräte und Softwaredienste in einem drahtlosen Netzwerk spontan eine Büroumgebung zur Verfügung zu stellen. Es wurden verschiedene Geräte, u.a. ein Faxgerät, ein Mobiltelefon und eine Telefonnummern-Datenbank sowie vorgeschaltete Laptops, die als Proxies agieren, durch ein drahtloses Netzwerk (IGate der Fa. Siemens) und ein darauf realisiertes Jini Netzwerk verbunden. Betritt ein Nutzer mit seinem Laptop und dem entsprechenden Gerät den Raum, so wird der Dienst, welcher von dem Gerät zur Verfügung gestellt wird, spontan erkannt, im Netz publik gemacht und eingebunden. Das Faxgerät befand sich aus Sicherheitsgründen hinter einer Firewall, die lediglich RMI- und HTTP-Aufrufe passieren ließ. Alle anderen Kommunikationsformen waren blockiert, um Fremdnutzung zu verhindern, was bei einem drahtlosen Netzwerk verständlich ist. Das Zulassen von RMI-Aufrufen geht auf die entfernten Aufrufe innerhalb des JINI-Netzwerkes zurück, das in diesem Fall RMI als Kommunikationsprotokoll nutzte. Der HTTP-Aufruf wird benötigt, um GUI (Graphical User Interface) Komponenten zwischen den Geräten zu verschicken: Wird zum Beispiel der Faxdienst von einem Rechner aus aufgerufen, so

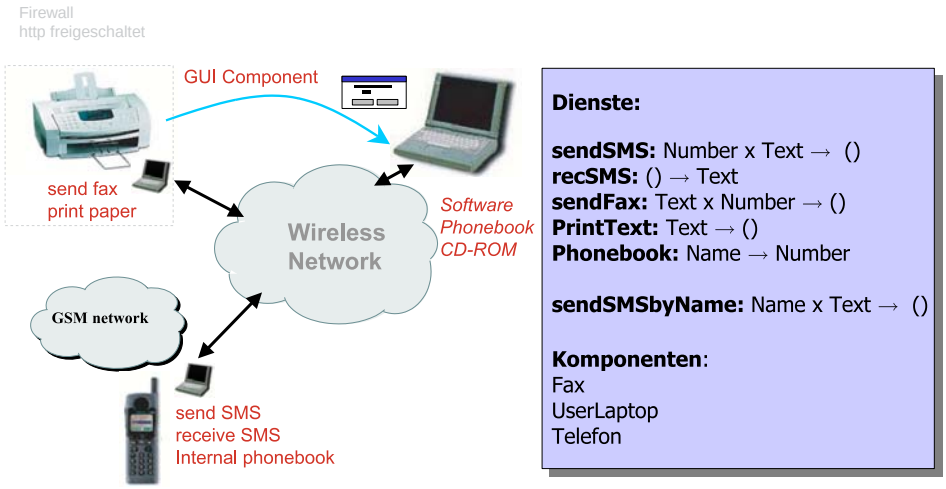


Abbildung 4.3: Beispielszenario “Mobile Office”

wird nach Discovery und Lookup (siehe Kapitel 3.2.2) die GUI Komponente per Code-Migration via HTTP vom Faxdienst auf den aufrufenden Rechner transferiert und dort zur Verfügung gestellt. Das Szenario, das in Abb. 4.3 noch einmal dargestellt ist, beinhaltet also die typischen Charakteristika eines spontanen Systems, wenn auch in einem kleinen übersichtlichen Beispiel:

Dynamische Verbindungsstrukturen: Neue Teilnehmer betreten den Raum und sichten und nutzen Dienste.

Mobilität: GUI Komponenten migrieren zwischen Plattformen während der Laufzeit (Code-Migration).

Umgebungsprofile: Obgleich die meisten Komponenten keinen Einschränkungen bezüglich der Kommunikationsmechanismen unterliegen, kann der Faxdienst nur via RMI oder HTTP kommunizieren. Man beachte hierbei, dass die Komponenten *technisch* in der Lage wären, via andere Protokolle zu kommunizieren: alle Komponenten verfügen über eine Implementierung des IIOP Protokolls. Jedoch wird dieses von der Umgebung (der Firewall) blockiert, sie besitzen also nicht das *Kommunikationsrecht* für IIOP.

Im Szenario ergeben sich bei den Geräten die folgenden möglichen Dienste, wobei hierbei der Begriff “Dienst” im Sinne von “Jini-Service” gebraucht wird:

> **`Phonebook:`** `name:string` → `phonenumber:int`
> Zu einem Namen wird die entsprechende Telefonnummer ausgegeben, sofern sie im Verzeichnis aufgeführt ist.
>
> **`SendSMS:`** `phonenumber:int` × `SMS:string` → ()
> An die übergebene Telefonnummer wird eine, ebenfalls übergebene, SMS (Kurznachricht) gesendet.

`RecSMS:` $() \rightarrow$ `SMS:string`
Es wird geprüft, ob eine Kurznachricht empfangen wurde und ggf. ausgegeben.

`sendFax:` `phonenumber:int` $\times$ `message:string` $\rightarrow ()$
Das übergebene Dokument wird als Faxnachricht an die ebenfalls übergebene Telefonnummer gesendet.

`PrintText:` `text:string` $\rightarrow ()$
Die übergebene Nachricht wird gedruckt.

`SendSMSbyName:` `name:string` $\times$ `SMS:string` $\rightarrow ()$
Die übergebene Nachricht wird als Kurznachricht an den per Namen übergebenen Empfänger gesendet.

Die einzelnen Komponenten – in diesem Fall also die Java Packages – belaufen sich auf folgende:

`Core`: Basisklassen und Basisinterfaces, die bestimmte Standardfunktionalitäten implementieren.

`Core.gui`: Alle wiederverwendbaren GUI Klassen, wie Fenster, Buttons etc.

`sms,phonebook,printer,fax`: Interfaces und Hilfsklassen für die Dienste.

`services.mobilePhone,services.fax, services.phonebook,services.sms`: Die eigentliche Implementierung der Dienstinterfaces.

Daneben existieren die normalen gebräuchlichen Interfaces, die Infrastruktur wie z.B. RMI Aufrufe etc. ermöglichen.

Betrachtet man nun beispielsweise den Dienst *Phonebook*, so kann dieser von verschiedenen Komponenten erfüllt werden, nämlich dem Laptop mit seinem internen CD-ROM, oder dem Mobiltelefon mit dessen internen Telefonbuch. Betrachtet man nun den zusammengesetzten Dienst *SendSMSbyName*, der eine Nachricht an einen Namen schickt, indem zuerst der Name mittels eines Telefonbuchdienstes in eine Telefonnummer umgewandelt wird und dann mittels des Dienstes *SendSMS* an diese die Nachricht geschickt wird, so ergeben sich zwei verschiedene *Konfigurationen* für diesen Dienst:

1. Mobiltelefon (als Dienst SendSMS) nutzt Mobiltelefon (als Dienst “Phonebook”)
2. Mobiltelefon (als Dienst SendSMS) nutzt Laptop (als Dienst “Phonebook”)

Die folgende Abbildung 4.4 verdeutlicht die Projektion des abstrakten Dienstes, der lediglich die funktionalen Abhängigkeiten, nicht jedoch die einzelnen Komponenten, die die Funktion erbringen, spezifiziert und dessen mögliche Konfigurationen auf der

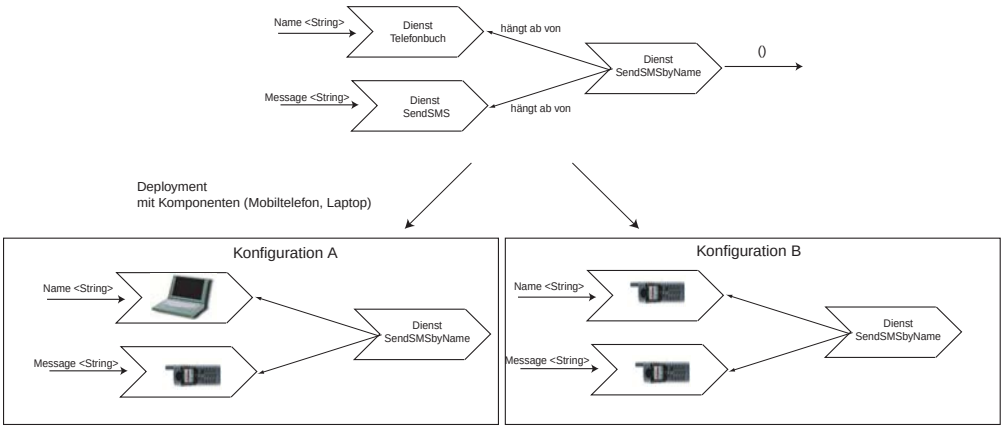


Abbildung 4.4: Die zwei möglichen Konfigurationen des Dienstszenarios “SendSMSbyName”

Menge der beiden Komponenten *Handy* und *Laptop*. Ferner sollte bei spontanen Komponentensystemen auch das bereits erwähnte Rechtekonzept miteinbezogen werden. Wenn also in unserem Beispiel zwei Dienste aufgrund der Netzwerktopologie durch eine Firewall oder ähnliches getrennt sind, kann die Menge der Konfigurationen eingeschränkt werden, da bestimmte Kommunikationsmechanismen, wie zum Beispiel CORBA-Aufrufe, von der Firewall abgeblockt werden, während SOAP Aufrufe via HTTP Protokoll passieren können. Eine Konfiguration mit SOAP Komponenten wäre also möglich, CORBA Komponenten scheiden aus. Es werden also benötigt:

Komponenten mit interner Strukturinformation: Im Sinne von Softwarekomponenten, die durch Verhalten und Struktur beschrieben sind und nach aussen bestimmte Funktionen anbieten und konsumieren. Beispiele hierfür sind die Softwarekomponenten für Mobiltelefon und Laptop, die einen internen Aufbau besitzen (Subkomponenten und deren Verdrahtung).

Komponenten ohne interne Strukturinformation: Diese legen lediglich die funktionale Abhängigkeit fest, *ohne* die interne Struktur, d.h. den Aufbau und die Beteiligten, festzulegen. Beispiel hierfür ist SendSMSbyName, die die Abhängigkeit von SendSMS und Phonebook festlegt ohne einen internen Aufbau festzulegen. Strukturlose Komponenten werden im weiteren als *Dienste* bezeichnet.

Lokationen: Auf diesen müssen alle Komponenten angesiedelt sein. Sie legen auch die Kommunikations- und Migrationsrechte untereinander, d.h. für die jeweilig angesiedelten Komponenten, fest. Lokationen, die Rechte verwalten, werden im weiteren als *Sandboxes* bezeichnet.

Abbildung einer Spezifikation, die strukturlose Komponenten (im weiteren als Dienste bezeichnet) einschließt, auf eine Spezifikation mit ausschließlich Komponenten mit Strukturinformation (siehe Abbildung 4.4). Diese Abbildung wird im weiteren als *Deployment*-Abbildung bezeichnet.

Eine der Aufgaben des Modells ist es nun, die Funktionalitäten und deren Abhängigkeiten durch Dienste zu modellieren und diese dann auf Basis einer Menge spezifi-

zierter Komponenten und einer spezifizierten Lokationen bzw. Umgebungen auf eine Menge von gültigen Konfigurationen abzubilden. Dadurch sind drei der Charakteristika spontaner Komponentensysteme, *dynamische Adaptivität*, *mobiles Verhalten* und *Rechtekonzept* im Modell ausdrückbar. Das vierte Charakteristikum, *spontane Interaktion*, wird durch die Beschreibbarkeit der Dienste erreicht (siehe Abschnitt 6).

4.4 Aufbau und Charakteristika des Modells

Die Architektursicht eines Systems sollte weitaus abstrakter sein als seine Realisierungssicht, wie z.B. eine Schnittstellenspezifikation mittels CORBA IDL [HOE96], um durch Implementierungsdetails nicht die Architektur einzugrenzen. Man beachte allerdings, dass durch eine solche Abstraktion die Genauigkeit nicht notwendigerweise leiden muss [DW98]. Eine Architekturbeschreibung ist nur von Nutzen, wenn sie ausreichend präzise und eindeutig ist. Es ist daher eine formale und eindeutige Basis für das hier vorgestellte Modell erforderlich, welche mächtig genug ist, sowohl Struktur als auch Verhalten auszudrücken. Diese Rolle spielt das Basismodell.

Durch präzise Spezifikation darf die intuitive Form eines Modells allerdings nicht verlorengehen: Ein formales Modell, welches nicht nur durch Automaten zur Analyse und Verifikation herangezogen, sondern auch von Menschen zur systematischen Entwicklung genutzt wird, muss eine einfache und intuitive Systemsicht besitzen.

Aufgrund dieser Ambivalenz, *Präzision* und *Intuition*, wurde im Rahmen dieser Arbeit ein Modell für spontane Komponentensysteme entwickelt, das aus zwei Schichten besteht und damit einen ähnlichen Ansatz verfolgt wie zum Beispiel das WRIGHT [All97] System, das auf CSP [Hoa85] basiert.

Das im Rahmen dieser Arbeit verwendete zweischichtige Modell ist ebenso für statische Komponentensysteme geeignet, bietet aber aufgrund seiner Trennung in logische und technische Abstraktionen die notwendigen Mittel, um spontane Komponentensysteme zu modellieren. Das Modell besteht aus einem Basismodell (FOCUS [BS00]), in dem grundlegende Konzepte, wie Komponenten, Kanäle und Verhalten auf eindeutige Weise definiert sind, und verschiedene Techniken (STDs, MSCs etc.) zur Verhaltensspezifikation ermöglicht. Hier stellen formale Modelle aufgrund ihrer mächtigen Abstraktionsmöglichkeiten und mathematisch nachvollziehbaren Aussagen den besten und ausgereiftesten Ansatz dar. Im Mittelpunkt des hier vorgestellten Ansatzes steht die zweite Schicht, das so genannte *Engineeringmodell*. Dieses bietet, abgestützt auf das Basismodell, die Entitäten spontaner Komponentensysteme explizit an und ist somit für den praktischen Entwurf solcher Systeme intuitiver und gebräuchlicher. Durch die Tatsache, dass es sowohl logische als auch technische Komponenten beinhaltet, kann es den gesamten Modellierungsprozess abdecken, wobei die logischen Komponenten auf technische abbildbar sind [SS01]. Ein informelles Engineeringmodell alleine würde auf den ersten Blick evtl. keine Nachteile besitzen, da der Entwickler mit dem Basismodell nur bei Bedarf, z.B. bei bewusstem Einsatz von Verhaltensbeschreibung, in Kontakt kommt. Betrachtet man derartige Ansätze jedoch näher, so zeigen sich recht bald Unzulänglichkeiten bei *systematischer* Entwicklung, wie sie

vor allem bei mittleren und großen Softwaresystemen notwendig ist. Bekanntes Beispiel ist hier die sehr verbreitete Unified Modeling Language (UML) [FS97], die eine informelle und sehr intuitive Systemsicht ohne formale bzw. präzise Fundierung besitzt. Hier ergaben sich sehr schnell Mängel bei der Definition der Beziehungen, wie z.B. der semantischen Differenzierung zwischen *aggregation* und *composition*, welche nicht eindeutig definiert ist.

Die in Kapitel 3.2 beschriebenen Charakteristika spontaner Komponentensysteme, als da wären dynamisches Verhalten, Mobilität, Umgebungsprofile und spontane Interaktion, werden wie folgt im Engineeringmodell thematisiert:

- **Mobilität** wird durch mobile Softwarekomponenten, die stets einer Sandbox zugeordnet sind, ausgedrückt. Sandboxes sind direkt adressierbar und explizit modelliert. Die eigentliche Mobilität, also der Wechsel, wird durch eine Folge von Schnappschüssen des Systems (verschiedene Konfigurationen) ausgedrückt.

- **Dynamik** und damit auch indirekt Mobilität (Grundlage für Code-Migration) wird durch das explizite Trennen zwischen invarianter *logischer Architektur* und deren zugeordneter Menge von *technischen Architekturen* (Konfigurationen) ausgedrückt. Dadurch ist es möglich, das dynamische Verhalten (die Umkonfigurationen) von der invarianten Funktionalität zu trennen. Die logische, invariante Systemstruktur wird durch eine *Dienstarchitektur* beschrieben.

- **Umgebungsprofile** werden durch *Sandboxes* modelliert. Sandboxes sind spezielle Containerentitäten, die ein Umgebungsprofil (ähnlich zu einem Ambient) repräsentieren. Sandboxes kontrollieren die Kommunikationsverbindungen nach aussen. Dies ist ein Unterschied zum Ambient-Kalkül, der Migration nach aussen kontrolliert, da Migration der grundlegende Kommunikationsmechanismus im Ambient-Kalkül ist. In der hier vorliegenden Arbeit wurden via Kanäle kommunizierende Komponenten als Basisabstraktion verwendet, da dies einer praktikablen Abstraktion im Entwurf näher kommt. Betrachtet man Migration von Komponenten als speziellen Nachrichtentyp, der über Kanäle verschickt wird, so kann die Migration leicht in diesen Ansatz integriert werden.

- **Spontane Interaktion** wird durch die Abbildung einer logischen Architektur auf die Menge der technischen Konfigurationen modelliert. Komponenten greifen nicht via Typen oder Instanzen aufeinander zu, sondern via *Dienste*. Dadurch ist es möglich, eine sehr lose Kopplung zu modellieren, in der die Komponenten weder via Instanzen noch über Typen fest verdrahtet sind.

KAPITEL

FÜNF

Basismodell

Das in dieser Arbeit verwendete Basismodell FOCUS benutzt Relationen zwischen Ein-/Ausgabeströmen, um verteilte Systeme darzustellen [BS00, Bro98b]. In FOCUS wird ein System als Menge von *Komponenten* und *Kanälen* modelliert. Die Komponenten kommunizieren durch asynchronen Austausch von Nachrichten auf den verbindenden Kanälen. Das Verhalten einer Komponente ist durch die Relation zwischen den Historien der Ein- und Ausgangsströme der Komponente auf den jeweiligen Ein- und Ausgabekanälen charakterisiert.

Hier wird dieses Modell als semantische Basis des Architekturmodells dieser Arbeit benutzt. Da es jedoch nicht im Mittelpunkt der Arbeit steht, soll im folgenden nur eine kurze Erläuterung der Konzepte dieses Modells genügen. Für eine ausführliche Erläuterung dieses Systemmodells sei der Leser auf [BS00, GSB97, GS96, Sto99, Bro98b] verwiesen.

In den folgenden Abschnitten definieren wir die Elemente unseres semantischen Modells, also dem Modell, auf das wir syntaktische Ausdrücke (Spezifikationen) abbilden. Als gegeben werden lediglich folgende Mengen vorausgesetzt:

M	Menge der Nachrichten	$\mathbf{T_M}$	Menge der Nachrichtentypen
CH	Menge der Kanäle	$\mathbf{T_{CH}}$	Menge der Kanaltypen

Aufbauend auf diesen Mengen werden folgenden Begriffe eingeführt:

- **Ströme**: Ströme sind endliche oder unendliche Folgen von Nachrichten.
- **Komponenten**: Komponenten werden durch eine Menge von Eingangs- und Ausgangskanälen, sowie einer *Interaktionsfunktion*, die die Stromhistorien der Eingangs- auf die der Ausgangskanäle abbildet, beschrieben.
- **Netzwerke**: Sind via Kanäle kommunizierende Komponenten.
- **Dienste**: Sind ein Teilverhalten einer Komponente, das bezüglich einer Teilsignatur (einem Ausschnitt der Schnittstelle) definiert wird.

Aufbauend auf diesen Aktoren, wird dann eine Spezifikationsart definiert, die es erlaubt *technische*, *logische* und *gemischte Netzwerke* (technische und logische Elemente) zu definieren. Hierbei handelt es sich um die formalen Konstrukte, mit deren Hilfe im späteren Engineeringmodell Konzepte wie *Dienste* (logisches Netzwerk) und *Komponenten* (technisches Netzwerk) fundiert werden.

Wir bauen dabei auf Begriffe auf, wie sie in [Bro98a] definiert sind.

5.1 Grundlagen

5.1.1 Ströme

Wie bereits erwähnt wird in FOCUS ein System als sog. *Komponentennetzwerk* modelliert. Ein solches Komponentennetzwerk besteht aus einer Menge von Komponenten **C**, die durch eine Menge von unidirektionalen Kanälen **CH** miteinander verbunden sind (siehe Abbildung 5.1). Sobald eine Komponente ein Zugriffsrecht, einen sog. *Port*, auf einen Kanal besitzt, ist diese mit dem jeweiligen Kanal verbunden. Ein Port besteht daher aus dem eindeutigen Bezeichner des Kanals ($n \in \mathbf{CH}$) sowie einem Lese- (?) bzw. Schreibrecht (!). Der Port $!n_1$ würde einer empfangenden Komponente als Schreibzugriff auf den Kanal mit dem Bezeichner n_1 ermöglichen, die Komponente besäße demnach nach Empfang des Ports den Kanal n_1 als Ausgangskanal. Mit $!?\mathbf{CH}$ bezeichnen wir die Menge aller Ports. Kanäle werden mittels der Typisierungsfunktion $type : \mathbf{CH} \rightarrow \mathbf{T_{CH}}$ typisiert, wobei $\mathbf{T_{CH}}$ die Menge der möglichen Kanaltypen bezeichnet. Ein Kanaltyp setzt die Art der Nachrichten fest, welche über den Kanal fließen können.

Das Systemverhalten wird durch die Kommunikationshistorien der Kanäle modelliert. Diese werden wiederum durch Nachrichtenströme ausgedrückt. Sei M die Menge aller Nachrichten eines Systems, wobei M^* die Menge der endlichen Sequenzen über M darstellt. Ein *Strom* s_n ist dann die endliche oder unendliche Sequenz von Nachrichten $m_i \in M$.

$$s_n =< m_1, m_2, m_3, \ldots, m_{41}, m_{42}, \ldots >$$

Hierbei sei zu beachten, dass M^* auch die leere Sequenz beinhaltet, welche durch das Symbol $<>$ dargestellt wird. Die Menge der *gezeiteten Ströme* $M^\omega \stackrel{Def}{=} (M^*)^\infty$ ist die Menge der unendlichen Sequenzen von endlichen Sequenzen über M. Dies bedeutet, dass ein Strom eine unendliche Sequenz wohldefinierter Zeitspannen beinhaltet, in denen endliche Sequenzen von Nachrichten enthalten sind:

$$s_m =< (m_1), \ldots, (m_{40}, \ldots, m_{42}), \ldots >$$

Für $i \in \mathbf{N}$ und $x \in M^\omega$ steht $x \downarrow i$ für die Sequenz der ersten i Sequenzen des Stromes x.

In einem folgenden Abschnitt wird die Verhaltensbeschreibung einer Komponente mittels des Verhältnisses der Ein- Ausgabekanalhistorien beschrieben. Die folgenden Definitionen sind dafür Voraussetzung.

Ein *Bündel von Nachrichtenströmen* (engl. *named stream tuple*) ist eine Funktion, die Kanalbezeichner auf gezeitete Nachrichtenströme abbildet. Für $C \subseteq \mathbf{CH}$ sei

$$\overline{C} \ : \ \mathbf{CH} \to M^{\omega}$$

als die Menge der Bündel über der Domäne C definiert.

Für $x \in \overline{C}$ und $C' \subseteq C$ stellt das Bündel $x|_{C'} \in \overline{C'}$ die Einschränkung von x auf die Kanäle von C' dar:

$$\forall\, c \in C' : x\,|_{C'}\,(c) = x(c)$$

Eine Funktion, die das Verhalten einer Komponente repräsentiert, muss kausal sein. Dies bedeutet, dass, falls eine bestimmte Eingabehistorie eine bestimmte Ausgabehistorie bewirkt, so muss eine längere Eingabehistorie kausal sein für mindestens dieselbe Ausgabe. Um diese Anforderung zu formulieren, definiert man folgende Funktion $f\colon \overline{I} \to \overline{O}$. Eine Funktion wird als *time guarded* bezeichnet, wenn und genau wenn für alle Eingabehistorien x, y und für alle $i \in \mathbf{N}$ die folgende Bedingung gilt:

$$x \downarrow i = y \downarrow i \Rightarrow (fx) \downarrow (i+1) = (fy) \downarrow (i+1)$$

Im weiteren Verlauf gehen wir bei allen stromverarbeitenden Funktionen von einer solchen time-guarded Eigenschaft aus, ohne sie explizit zu verlange. Darüberhinaus verlangen wir von einer Verhaltensfunktion die Stetigkeit, um eine intuitive Vorstellung von schrittweiser Berechnung der Ausgaben zu ermöglichen [BDD+92].

5.1.2 Komponenten und deren Komposition

In diesem Abschnitt definieren das mathematische Modell einer Komponente. Wir arbeiten dabei mit getypten Kanälen. Wir nehmen eine Menge $\mathbf{T_{CH}}$ von Sorten oder Typen als gegeben an. Mit **CH** bezeichnen wir die Menge der getypten Kanäle. Weiterhin nehmen wir eine Typzuordnung

$$type : \mathbf{CH} \to \mathbf{T_{CH}}$$

als gegeben an. Zu einer gegebenen Menge **CH** getypter Kanäle können wir nun festlegen, was wir unter einer *Kanal-Belegung* verstehen (sei **M** wieder die Menge aller Nachrichten, dann bezeichnen wir für einen Typ t mit $\|t\|$ die Menge seiner Nachrichten):

$$\overrightarrow{\mathbf{CH}} \stackrel{Def}{=} \left\{ x \in \mathbf{CH} \to (M^*)^{\infty} : \forall\, c \in \mathbf{CH} : x\,(c) \in (\|type\,(c)\|^*)^{\infty} \right\}$$

Eine Kanal-Belegung $x \in \overrightarrow{CH}$ ordnet jedem Kanal $c \in CH$ einen gezeiteten Strom von Elementen aus $type\,(c)$ zu.

Als (syntaktische) Schnittstelle einer Komponente bezeichnen wir die Menge der getypten Ein- und Ausgabe-Kanäle der Komponente:

- (I, O) **(syntaktische) Schnittstelle**

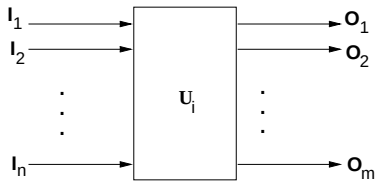


Abbildung 5.1: Das Verhalten einer Komponente wird durch die Relation zwischen Ein- (I) und Ausgabeströme (O) modelliert.

- I Menge getypter **Eingabe-Kanäle**
- O Menge getypter **Ausgabe-Kanäle**

Neben der Schnittstelle einer Komponente modellieren wir ihr Verhalten. Wir beschreiben das Verhalten, indem wir die Historien von Eingabe-Kanälen der Komponente zu Historien ihrer Ausgabekanäle in Beziehung setzen. Eingabe-Historien bilden wir durch Kanal-Belegungen der Eingabe-Kanäle, und Ausgabe-Historien durch Belegungen der Ausgabe-Kanäle ab. Entsprechend modellieren wir das Blackbox-Verhalten einer Komponente durch eine Funktion

$$F : \overrightarrow{I} \rightarrow \wp\left(\overrightarrow{O}\right)$$

Für ein $x \in \overrightarrow{I}$ bezeichnet $F(x)$ die Menge aller Ausgabe-Historien die eine Komponente mit Verhalten F produzieren kann.

Die Menge aller möglichen *Komponenten* über einer Menge von Kanälen sei nun wie folgt definiert: Sei $K \subseteq \mathbf{CH}$ eine Menge getypter Kanäle, dann definieren wir die Menge der Komponenten über dieser Kanalmenge K durch

$$C(K) \stackrel{Def}{=} \left\{(I,O,F) \in \wp(K) \times \wp(K) \times \left(\overrightarrow{\wp(K)} \rightarrow \wp\left(\overrightarrow{\wp(K)}\right)\right) : F \in \vec{I} \rightarrow \wp(\vec{O})\right\}$$

Die Bedingung

$$F \in \vec{I} \rightarrow \wp(\vec{O})$$

in der Definition von $C(\mathbf{CH})$ stellt eine Konsistenzbedingung dar, die wir entsprechend auf die syntaktische Ebene heben können (vgl. Definition der Syntax, Konsistenzbedingungen).

In unserem semantischen Modell entspricht also eine Komponente nicht nur einer Verhaltensfunktionen, sondern erfasst auch die Mengen von Ein- und Ausgabe-Kanälen explizit. Weiterhin definieren wir für eine Komponente $c \in C(\mathbf{CH})$ mit $c = (I,O,F)$,

um uns im weiteren damit auf die Elemente einer Komponente beziehen zu können.

Nun führen wir einen Kompositionsbegriff für Komponenten ein. Dazu definieren wir zunächst einen Verknüpfungsoperator $\oplus$, der für gegebene, disjunkte Kanalmengen

$in.c$	$\stackrel{Def}{=}$	I
$out.c$	$\stackrel{Def}{=}$	O
$behav.c$	$\stackrel{Def}{=}$	F

Tabelle 5.1: Notation für die Bestandteile von Basismodellkomponenten und Netzwerken.

C_1 und C_2 , und Kanal-Belegungen $x \in \overrightarrow{C_1}$ sowie $y \in \overrightarrow{C_2}$ definiert sei durch folgende Gleichungen:

$$(x \oplus y)(c) = x(c) \quad \textit{falls } c \in C_1 \textit{ und}$$
$$(x \oplus y)(c) = y(c) \quad \textit{falls } c \in C_2$$

Der Kompositionsoperator $\otimes$ für Komponenten sei wie folgt definiert:

$$\forall C_1 = (I_1, O_1, F_1), C_2 = (I_2, O_2, F_2) \in C(\mathbf{CH}) : \; C_1 \otimes C_2 = C = (I, O, F)$$

so daß

$$I = (I_1 \cup I_2) \setminus (O_1 \cup O_2) \;\; \textit{und}$$
$$O = (O_1 \cup O_2) \setminus (I_1 \cup I_2) \;\; \textit{und}$$
$$F \in \overrightarrow{I} \to \wp\left(\overrightarrow{O}\right)$$

wobei

$$\forall x \in \overrightarrow{I} : \; F(x) = \{y' \in \overrightarrow{O} : \exists\, y \in \overrightarrow{I_1 \cup I_2 \cup O_1 \cup O_2} :$$
$$y|_O = y' \wedge y|_I = x \wedge \; y|_{O_1} = F_1(y|_{I_1}) \wedge y|_{O_2} = F_2(y|_{I_2})\}$$

Dabei bezeichnen wir mit $x|C$ die Einschränkung der Belegung x auf die Kanäle in C. Zusätzlich zur expliziten Modellierung von Kanälen definieren wir im folgenden noch den Begriff des Komponenten-Netzwerkes, mit dem wir auch die Menge der Komponenten eines Systems im semantischen Modell explizit erfassen.

5.1.3 Dienste

Ein Dienst ist eine Funktionalität, die separat und unabhängig von Komponenten spezifiziert werden kann. Dienste sind also im allgemeinen oft auftretende Funktionalitäten, die durch eine Interaktionsbeschreibung (Blackbox-Verhalten) zu einer Schnittstellen definiert sind, jedoch keinerlei Aussagen über interne die Struktur oder das Glassbox-Verhalten machen.

Ein Dienst im Basismodell stellt ein Teilverhalten F_i' einer Komponente dar, das in Relation zu einer Teilmenge $I_i' \subseteq I, O_i' \subseteq O$ ihrer Signatur I, O, beispielsweise mittels STDs oder MSCs, definiert wird. Hierbei gilt, dass die Teilverhalten sich gegenseitig nicht beeinflussen, also weder gemeinsame Kanäle nutzen, noch Rückkopplungen besitzen. Dadurch werden Probleme der Feature-Interaction ausgeschlossen, da sie für die hier behandelte Anwendungsdomäne unerheblich sind.

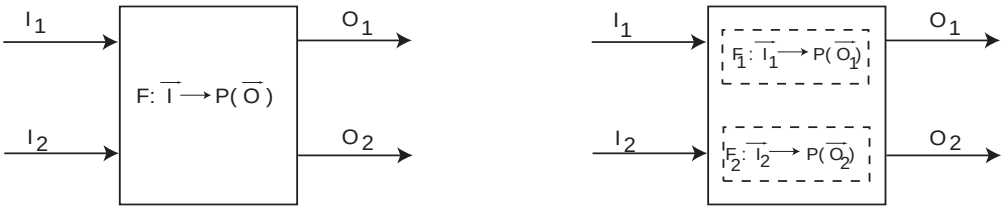


Abbildung 5.2: Zerlegen einer Komponente in Dienste

Eine Komponente mit Verhalten F und der Signatur I, O wird als dienstorientiert bezeichnet, wenn es einen Dienst D mit dem Verhalten F_D bezüglich einer Signatur $I_D \subseteq I$ und $O_D \subseteq O$ gibt, so dass für eine Funktion $F' : \overrightarrow{I'} \rightarrow \wp\left(\overrightarrow{O'}\right)$ mit $I' = I \backslash I_D$ und $O' = O \backslash O_D$ gilt:

$$\begin{aligned} \text{Sei } F_D &: \overrightarrow{I}_D \rightarrow \wp\left(\overrightarrow{O}_D\right), \text{dann} \\ F' &: \overrightarrow{I'} \rightarrow \wp\left(\overrightarrow{O'}\right) \Rightarrow F = F_D \otimes F' \wedge I_D \cap I' = \varnothing \wedge O_D \cap O' = \varnothing \end{aligned}$$

Dies bedeutet insbesondere, dass die Dienste einer Komponente immer disjunkte Kanäle besitzen. Abbildung 5.2 zeigt links eine dienstorientierte Komponente mit dem Verhalten

$$F : \overrightarrow{I} \rightarrow \wp\left(\overrightarrow{O}\right)$$

Dieses lässt sich zerlegen in zwei Teilverhalten F_1 und F_2, die jeweils bezüglich zweier disjunkter Teilsignaturen definiert sind:

$$\begin{aligned} &F_1 : \overrightarrow{I}_1 \rightarrow \wp\left(\overrightarrow{O}_1\right) \text{ und} \\ &F_2 : \overrightarrow{I}_2 \rightarrow \wp\left(\overrightarrow{O}_2\right) \end{aligned}$$

Damit die Komponente dienstorientiert ist, muss gelten, dass $F = F_1 \otimes F_2$.

Dies bedeutet insbesondere, daß wir nur bestimmte Komponentenarten als *dienstorientiert* oder eine *Dienstarchitektur erfüllend* bezeichnen. Nämlich jene Komponenten, deren Verhalten sich in parallelkomponierbare Teilverhalten bezüglich disjunkten Teilsignaturen, zerlegen lässt. Dies schränkt zwar die Menge der möglichen Komponente ein, es werden sogar mögliche Komponentenarten ausgeschlossen, die eventuell eine interessante oder besser geeignete Form besitzen. Jedoch ist diese Form des Modells für die hier behandelte Domäne (spontane Komponentensysteme, wie sie in denen in Abschnitt 2.3 beschriebenen Domänen vorkommen) vollkommen ausreichend. Eine Miteinbeziehung aller Komponentenarten und die damit verbundene Erweiterung des Dienstbegriffes würde einen deutlichen Zuwachs an Komplexität im Modell mit sich bringen, was auf Kosten der Einfachheit und damit der Verständlichkeit gehen würde. Es wäre auch möglich, einen durch mehrere Dienste überlagerten Kanal durch eine Projektion auf mehrere disjunkte Kanal-Belegungen abzubilden, jedoch ist das durch

die vereinfachende Einschränkung auf dienstorientierte Komponenten nicht notwendig. Die Einschränkung tangiert die hier behandelte Anwendungsdomäne keineswegs, da jede Komponente spontaner Komponentensysteme als dienstorientierte Komponente spezifizierbar ist.

Es sei ebenfalls noch einmal darauf hingewiesen, dass ein Dienstbegriff erst dann Sinn macht, wenn er methodisch eingesetzt wird. Dies bedeutet insbesondere, dass die Funktionalität, die ein Dienst repräsentiert, aus der Anwendungsdomäne bestimmt werden muss. Dies ist eine der Hauptaufgaben des Dienstbegriffes des Engineeringmodells in Abschnitt 6.1.1.

5.1.4 Komponenten-Netzwerke

Aufbauend auf dem Begriff der Komponente definieren wir ein Komponenten-Netzwerk als eine Menge von Komponenten. Sei **CH** wieder eine Menge von Kanälen, dann definieren wir entsprechend die Menge $NW(\mathbf{CH})$ der Komponenten-Netzwerke durch

$$NW(\mathbf{CH}) \stackrel{Def}{=} \wp(C(\mathbf{CH}))$$

Wir definieren weiterhin für ein Netzwerk $nw \in NW(\mathbf{CH})$ mit $nw = \{c_1, c_2, ..., c_n\}, n \in \mathbf{N}$:

$$\begin{aligned} comp.nw &\stackrel{Def}{=} \{c_1, c_2, ..., c_n\} \\ in.nw &\stackrel{Def}{=} in.c' \\ out.nw &\stackrel{Def}{=} out.c' \\ behav.nw &\stackrel{Def}{=} behav.c', \\ &\quad wobei\ c' = c_1 \otimes c_2 \otimes ... \otimes c_n \end{aligned}$$

Die Black-Box-Eigenschaften eines Netzwerkes ergeben sich also unmittelbar aus der Komposition der Komponenten des Netzwerkes.

Dies bedeutet insbesondere, dass ein Netzwerk im semantischen Modell durch seine Menge von Komponenten eindeutig charakterisiert wird. Dies geschieht aufgrund der Komposition und der Eigenschaften der Komponenten.

Die Komposition der Netzwerke innerhalb des Basismodells wird durch den Operator $\otimes_{net}$ ausgedrückt, der durch die Vereinigung der Mengen von Komponenten der Netzwerke definiert wird. Die Eingangs- und Ausgangs-Portmenge, sowie die Interaktionsfunktion des komponierten Netzwerkes werden dann durch die oben genannten Anforderungen festgelegt.

$$\otimes_{net} \in NW(\mathbf{CH}) \times NW(\mathbf{CH}) \rightarrow NW(\mathbf{CH})$$

wobei

$$\begin{aligned} &\forall\, nw_1, nw_2 \in NW(\mathbf{CH}) : \\ &nw_1 \otimes_{net} nw_2 \stackrel{Def}{=} nw_1 \cup nw_2 \end{aligned}$$

Die Notation der einzelnen Bestandteile der Netzwerke folgt der oben gegebenen Notation für Komponenten (siehe Tabelle 5.1). Zusätzliche steht *comp.n* für die Menge der Komponenten des Netzwerkes *n*. Mit $NW(\mathbf{CH})$ bezeichnet man nun die Menge aller Netzwerke über **CH**.

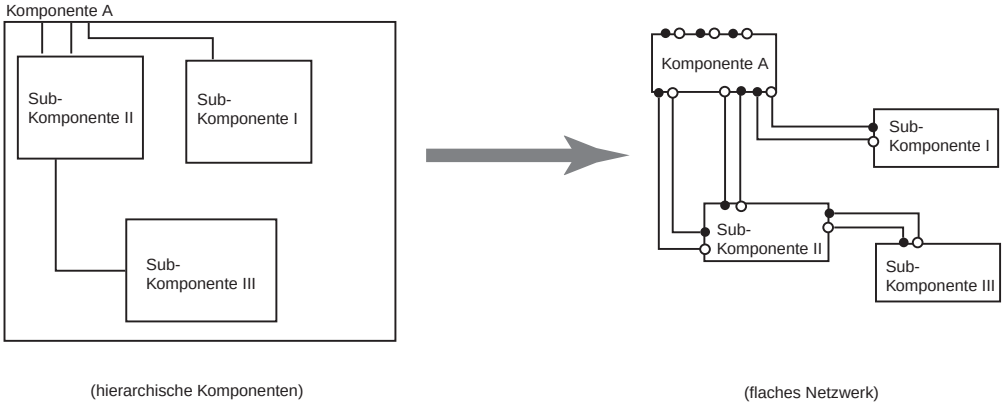


Abbildung 5.3: Eine hierarchische Komponenteabstraktion und deren flaches Pendant im Basismodell (rechts)

Durch die Spezifikation eines Komponentennetzwerkes ist es dann möglich ein flaches Pendant zu hierarchisch verschachtelten Komponenten auszudrücken. Eine Komponente ist dann Subkomponente zu einer anderen, wenn sie nur Kanalverbindungen zu dieser Mutterkomponente oder zu deren Subkomponente besitzt. Abbildung 5.3 illustriert dies an einem Beispiel.

Solche Komponentennetzwerke können dann wiederum als Blackbox und damit als Komponente Betrachtet werden, die gemäß dem oben definierten Dienstbegriff dienstorientiert sein kann. Verschiedene Arten der Spezifikation von Komponentennetzwerken, die in verschiedenen Graden deren Aufbau festlegen werden im folgenden Abschnitt 5.2 behandelt.

5.2 Spezifikation von Komponentennetzwerken

Durch den obigem Komponenten-Begriff haben wir das semantische Modell von FOCUS (vgl. z.B. [Bro98a] und [SS95]) modifiziert. In der obigen Strukturierung modellieren wir die Kanäle einer Komponente explizit, und nicht implizit in der Signatur der Verhaltensfunktion, wie dies in FOCUS der Fall ist. Damit schaffen wir uns die Möglichkeit, zwischen Spezifikationselementen zu unterscheiden, die nur Verhaltens-Eigenschaften spezifizieren, und solchen, die auch strukturelle Eigenschaften festlegen.

Auf syntaktischer Ebene verwenden wir die Ergebnisse aus [SS95] und geben Erweiterungen an, um sowohl logische als auch technische Architektur-Elemente spezifizieren zu können. Durch die Definition einer (semantischen) Abbildung auf das oben definierte semantische Modell geben wir den syntaktischen Ausdrücken eine Bedeutung. Dabei wird der Unterschied zwischen logischer und technischer Sicht deutlich.

Die Semantik der Spezifikationen wird durch die Menge der Komponentennetzwerke bestimmt, die die Spezifikation erfüllen. Diese Menge wird durch drei Prädikate bestimmt:

- **Blackbox-Sicht** (*blackbox*) legt die Signatur der Netzwerke nach aussen fest.
- **Glassbox-Sicht** (*glassbox*) legt die interne Struktur des Netzwerkes fest.
- **Verhaltens-Sicht** (*behavior*) legt das Verhalten des Netzwerkes fest.

Prinzipiell müssen zwei verschiedene Arten von Spezifikationen unterschieden werden: *atomare*, also Netzwerke bestehend aus einer Komponente, und *komponierte* Spezifikationen, die aus mehreren Komponenten bestehen. Im ersteren Fall fällt das Prädikat für die Glassbox-Sicht weg, da das Netzwerk atomar, also aus einem Element bestehend, ist und damit die Glassbox-Sicht trivial ist.

Das Verhaltens-Sicht-Prädikat wird jeweils für die verschiedenen Spezifikationstypen (logisch, technisch oder gemischt) anders definiert und legt somit die möglichen Spielräume fest.

Die einzelnen Definitionen werden in den folgenden Abschnitten erklärt.

5.2.1 Atomare Spezifikationen

Als atomare Spezifikationen bezeichnen wir Spezifikationen, die Netzwerke definieren, die aus genau einer Komponente aufgebaut sind, und die wir auch spezifikatorisch als Einheit (und nicht als logisches Netzwerk) betrachten.

Wir unterscheiden zwei Arten von Basismodellspezifikationen:

- *logische Netzwerk* Spezifikation
- *technische Netzwerk* Spezifikation

Im ersten Fall wird die Menge der die Spezifikation erfüllenden Basismodell-Netzwerke lediglich durch Blackbox-Verhalten und die -Sicht bestimmt, nicht jedoch durch die interne Struktur. Im zweiten Fall wird diese interne Struktur, also eine Glassboxsicht, auch in Betracht gezogen.

Das Schema zur Spezifikation von logischen, atomaren Netzwerken hat folgende Form:

Mit einer solchen Spezifikation führen wir einen Bezeichner für ein Netzwerk ein. Weiterhin definieren wir die endlichen Mengen der getypten Ein- und Ausgabe-Kanäle des Netzwerkes. Sie bilden die syntaktische Schnittstelle. Die Bezeichner $i1, i2, \ldots, in$ und $o1, o2, \ldots on$ stehen als Platzhalter für global eindeutige Kanalbezeichner. Die $I1, \ldots, In$ und $O1, \ldots, On$ sind Platzhalter für Sortenbezeichner. Das Verhalten spezifizieren wir durch ein Prädikat R, dessen Menge der Variablen genau die Kanalbezeichner der Schnittstelle sind. Die Einführung von Bezeichnern (hier A) von Netzwerken

network A
input i1:I1, i2:I2, ..., im:Im
output o1:O1,o2:O2,..., on:On
is logical
R(i1, ..., im, o1, ..., on)
end A

Abbildung 5.4: Schema-Notation einer Spezifikation eines logischen Basismodell-Netzwerkes

erlaubt es uns, Netzwerk-Bezeichner im Sinne einer Typisierung von Teil-Netzwerken im Rahmen der Spezifikation komponierter Netzwerke zu verwenden (siehe unten).

Man beachte, dass *A* ein Platzhalter für einen solchen Netzwerk-Bezeichner ist, der in der konkreten Spezifikation durch einen eindeutigen Bezeichner ersetzt wird. Diese Instanz des Schemas nennen wir *konkrete Spezifikation*.

Die Semantik der Spezifikationen wird, wie bereits oben erläutert, durch die Menge von Komponentennetzwerken definiert, die die Spezifikation erfüllen. Diese Menge wiederum wird durch die drei Konsistenz-Prädikate bestimmt, die aus der Spezifikation abgeleitet werden. Da die Glassbox-Sicht im atomaren Fall trivial ist, definieren wir nun die übrigen beiden. Wir verwenden als Hilfskonstrukt $\overline{c_i} = c_{i1}, \ldots, c_{im}$ und analog $\overline{c_o} = c_{o1}, \ldots, c_{on}$:

Blackbox-Sicht – Blackbox-Sicht drückt die Signatur aus, die die Komponentennetzwerke besitzen müssen. Die Blackbox-Sicht ist folgendermaßen definiert:

$$blackbox \in NW(\mathbf{CH}) \times \mathbf{CH}^m \times \mathbf{CH}^n \rightarrow Bool$$

$$blackbox(nw, \overline{c_i}, \overline{c_o}) \stackrel{Def}{=} in.nw = \{c_{i1}, \ldots, c_{im}\} \wedge out.nw = \{c_{o1}, \ldots, c_{on}\}$$

Verhaltens-Sicht – Die Verhaltens-Sicht beschreibt das Verhalten, das die Komponenten-Netzwerke nach aussen besitzen müssen. Die Verhaltens-Sicht-Definition hängt vom späteren Spezifikationstyp (logisch oder technisch) ab und wird dort jeweils speziell definiert. Das Prädikat hat aber in beiden Fällen die Form:

$$behavior \in NW(\mathbf{CH}) \times \mathbf{CH}^m \times \mathbf{CH}^n \rightarrow Bool$$

Die Menge der Komponentennetzwerke, für die das folgende auf den beiden Prädikaten aufbauende Prädikat *is_A* gilt, bestimmen dann die Semantik der Spezifikation.

$$is\_A \in NW(\mathbf{CH}) \rightarrow Bool$$

$$
\begin{aligned}
is\_A(nw) \quad \stackrel{Def}{=} \quad & \exists\, \overline{c_i}, \overline{c_o} \in \mathbf{CH} : \\
& blackbox(\overline{c_i}, \overline{c_o}) \wedge \\
& behavior(nw, \overline{c_i}, \overline{c_o})
\end{aligned}
$$

Man beachte hierbei, dass *A* einen Platzhalter darstellt, der für jede Spezifikation eines Komponententyps instanziiert wird und somit andere Prädikate zur Folge hat.

Logische Netzwerkspezifikationen, wie die in Abbildung 5.4 aufgeführte Spezifikation des Netzwerkes *A*, stellen ein *logisches Netzwerk* dar. Deren Semantik wird durch die Komponentennetzwerke des Basismodells definiert, die das oben beschriebene Prädikat *is_A* erfüllen (im Sinne der Allgemeinheit wird wieder der Bezeichner *A* als Platzhalter verwendet)[1].

Das Verhaltens-Sicht-Prädikat einer atomaren Netzwerk Spezifikation zur Abbildung einer Schemainstanz *A*, *behavior*, wird nun wie folgt definiert:

$$
\begin{aligned}
behavior(nw, \overline{c_i}, \overline{c_o}) \quad \stackrel{Def}{=} \quad & (\forall\, r \in \overrightarrow{in.nw}, s \in \overrightarrow{out.nw} : s \in behav.nw \Leftrightarrow \\
& R(i_1, \ldots, i_m, o_1, \ldots, on) \wedge \\
& i_1 = r(c_{i1}) \wedge \ldots \wedge i_m = r(c_{im}) \wedge \\
& o_1 = s(c_{o1}) \wedge \ldots \wedge o_n = s(c_{om})
\end{aligned}
$$

Damit ergeben sich die Netzwerk-Eigenschaften direkt aus den Eigenschaften dieser einen Komponente. Eine Komponente bzw. ein Netzwerk muss, um der Spezifikation zu genügen, eine entsprechende Schnittstelle bieten und (bezüglich dieser Schnittstelle) ein Verhalten entsprechend dem Prädikat der Spezifikation. Das bedeutet, dass die Verhaltens-Funktion der Komponente die Belegungen von Ein- und Ausgabe-Kanälen so zueinander in Beziehung setzt, dass das Verhaltens-Prädikat angewandt auf diese Belegungen erfüllt ist.

Technische Basismodell-Spezifikationen werden durch das folgende Schema spezifiziert, das sich lediglich durch das Schlüsselwort technical unterscheidet:

```
network A
input   i1:I1, i2:I2, ..., im:Im
output  o1:O1,o2:O2,..., on:On
is technical
  R(i1, ..., im, o1, ..., on)
end A
```

Abbildung 5.5: Spezifikationsnotation technischer Basismodell-Netzwerke

Die Interpretation erfolgt analog, allerdings ist das Prädikat *is_A* folgendermaßen angepasst:

[1] Man beachte, dass sich die Anzahl der Ein- Ausgabekanäle *m* bzw. *n* nach dem Komponententyp richten. Hier ergeben sie sich aus der Spezifikation aus Abbildung 5.4

$$
\begin{aligned}
is\_A(nw, \overline{c_i}, \overline{c_o}) \quad \stackrel{Def}{=} \quad & \exists\, c \in \mathbf{C}(\mathbf{CH}) : \\
& comp.nw = \{c\} \wedge \\
& blackbox(nw, \overline{c_i}, \overline{c_o}) \wedge \\
& behavior(nw, \overline{c_i}, \overline{c_o})
\end{aligned}
$$

Das bedeutet, dass im Falle eines technischen Netzwerkes, nur Netzwerke erlaubt sind, die aus genau *einer* Basismodell-Komponente bestehen. Damit ist auch die interne Struktur, nämlich die atomare, vorgegeben.

5.2.2 Kompositionsspezifikationen

Im Unterschied zu atomaren Spezifikationen beschreiben wir in Kompositionsspezifikationen Netzwerke auch spezifikatorisch durch Komposition von (Teil-) Spezifikationen. Wir unterscheiden drei Arten kompositorischer Spezifikationen.

- rein technische Spezifikationen,
- rein logische Spezifikationen,
- gemischte Spezifikationen.

Rein technische Spezifikationen definieren nicht nur Anforderungen an die Black-Box-Eigenschaften eines Systems, sondern auch, durch welches (semantische) Komponenten-Netzwerk (Menge und Art von Komponenten, sowie deren Komposition durch gemeinsame Kanäle) diese Eigenschaften zu realisieren sind.

Im Unterschied dazu formulieren wir mit rein logischen Spezifikationen nur Anforderungen an die Black-Box-Eigenschaften eines Systems. Die logische Komposition hilft uns dabei, nur das Problem auf spezifikatorischer Ebene zu strukturieren.

Um wiederum die Menge der Komponentennetzwerke, die die Semantik einer Spezifikation repräsentiert, muss bei Kompositionsspezifikationen im Gegensatz zu atomaren Spezifikationen der interne Aufbau berücksichtigt werden. Dies wird durch die Hinzunahme des Glassbox-Sicht-Prädikats *glassbox*, das den internen Aufbau der Komponentennetzwerke bestimmt. Die Menge der Komponentennetzwerke, die durch eine Kompositionsspezifikation bestimmt werden werden also nicht mehr, wie im atomaren Fall durch zwei Prädikate, Blackbox- und Verhaltens-Sicht, sondern durch drei, Blackbox-, Glassbox- und Verhaltens-Sicht, definiert:

Blackbox-Sicht – Blackbox-Sicht drückt die Signatur aus, die die Komponentennetzwerken besitzen müssen. Die Blackboxkonsistenz wurde bereits oben definiert.

Glassbox-Sicht – legt den internen Aufbau des Netzwerkes fest.

$$
glassbox \in (NW(\mathbf{CH}) \times \mathbf{N}) \quad \rightarrow \quad Bool
$$

$$
\begin{aligned}
glassbox(nw_{comp}, t) \quad \stackrel{Def}{=} \quad & \exists\, nw_1, \ldots, nw_{k+t} \in NW(\mathbf{CH}), \\
& h_1, \ldots, h_p \in \mathbf{CH}, \\
& nw_{comp} = nw_1 \otimes_{net} \ldots \otimes_{net} nw_{(k+t)} \wedge \\
& is\_A1(nw_1, i_{11}, \ldots, i_{1s}, o_{11}, \ldots, o_{1r}) \wedge \\
& \ldots \\
& is\_A(k+t)(nw_{(k+t)}, i_{(k+t)1}, \ldots, i_{(k+t)s}, o_{(k+t)1}, \ldots, o_{(k+t)r}) \wedge \\
& (\forall\, x, y \in 1, \ldots, (k+t) : x \neq y \Rightarrow \\
& comp.nw_x \cap comp.nw_y = \varnothing) \\
& \text{wobei } s, p \in \mathbf{N} \text{ sich aus dem Schema ergeben}
\end{aligned}
$$

Zum Verständnis sei hier gesagt, dass der Parameter $t \in \mathbf{N}$ nur ein Hilfsparameter ist, der die, bei den späteren gemischten Spezifikationen notwendige, Trennung zwischen logischen und technischen Anteilen angibt. Zum besseren Verständnis ist es hilfreich diesen am Anfang auf Null zusetzen und damit wegzulassen.

Verhaltens-Sicht – Die Verhaltens-Sicht beschreibt das Verhalten, das die Komponenten-Netzwerke nach aussen besitzen müssen. Die Verhaltens-Sicht-Definition hängt vom späteren Spezifikationstyp (logisch oder technisch) ab und wird dort jeweils speziell definiert. Das Prädikat hat aber die Form:

$$behavior \in NW(\mathbf{CH}) \times \mathbf{CH}^m \times \mathbf{CH}^n \rightarrow Bool$$

Da wir davon ausgehen müssen, dass wir im Rahmen einer System-Entwicklung manche System-Anteile (lediglich) logisch spezifizieren, andere jedoch (bereits) technisch, integrieren wir beide Sichtweisen in Form gemischter Spezifikationen. Die technischen Anteile legen dabei fest, aus welchen Komponenten ein Teil des gesamten Netzwerkes aufgebaut ist, und welchen Anteil an der Schnittstelle und dem Verhalten des gesamten Netzwerkes diese Komponenten abdecken. Im Unterschied dazu beschreiben wir durch die logischen Teil-Netzwerke die Schnittstellen- und Verhaltens-Anteile des Gesamt-Netzwerkes, für die wir (noch) nicht festlegen, durch welche Menge von "realen" (Implementierungs)-Komponenten diese Anteile erbracht werden sollen. Dementsprechend ergibt sich die Schnittstelle und das Verhalten des spezifizierten Netzwerkes aus der Komposition von sowohl technischen als auch logischen Komponenten. Aussagen über den internen Aufbau des Netzwerkes leiten wir nur aus dem technischen Architektur-Anteil ab.

Technische Netzwerk-Spezifikationen

Spezifikations-Schema für rein technische Architekturen, das bedeutet Architekturen ohne logischen Anteil:

Die A1, ..., Ak sind Platzhalter für Netzwerk-Bezeichner. Die i11, ..., iks; o11, ...,okr sind Platzhalter für Kanal-Bezeichner, die entweder mit den Bezeichnern der

```
network A
input     i1:I1, i2:I2, ..., im:Im
output    o1:O1,o2:O2,..., on:On
internal  h1:H1,h2:H2,..., hj:Hj
is
  technical network
      <o11, ..., o1r> = A1 <i11, ..., i1s>;
                ...
      <ok1, ..., okr> = Ak <ik1, ..., iks>;
end A
```

Abbildung 5.6: Spezifikation von Basismodellkomponenten mit ausschließlich technischen Bestandteilen

Schnittstellen-Kanäle oder mit internen Kanäle belegt werden können. Als interne Kanäle bezeichnen wir dabei solche Kanäle, die nicht in der Schnittstellenbeschreibung auftreten, also die Komponenten des Netzwerkes nur untereinander, und nicht mit der Umgebung des Netzwerkes verbinden. Im Folgenden verwenden wir $\mathsf{h1}, \ldots, \mathsf{hp}$ als Bezeichner für interne Kanäle.

In einer konkreten Netzwerk-Spezifikation ersetzen wir die Platzhalter des Schemas durch konkrete Bezeichner für externe und interne Kanäle. Für die Interpretation einer Spezifikation entsprechend obigem Schema leiten wir zunächst aus den k Gleichungen folgendes Prädikat

$$is\_A \in NW(\mathbf{CH}) \times \mathbf{CH}^m \times \mathbf{CH}^n \longrightarrow Bool$$

ab und definieren es entsprechend dem Schema wie folgt:

$$\begin{aligned} is\_A(nw, \overline{c_i}, \overline{c_o}) \quad \stackrel{Def}{=} \quad & \exists\, nw_{comp} \in NW(\mathbf{CH}) : \\ & blackbox(nw, \overline{c_i}, \overline{c_o}) \wedge \\ & glassbox(nw_{comp}, 0) \wedge \\ & behavior(nw, nw_{comp}) \end{aligned}$$

Hierbei gilt die bereits oben definierte Blackbox- und Glassbox-Sicht, sowie die folgende Definition der Verhaltens-Sicht:

$$behavior(nw, nw_{comp}) \quad \stackrel{Def}{=} \quad nw = nw_{comp}$$

Das bedeutet, dass ein Netzwerk genau dann die Spezifikation erfüllt, wenn wir es in disjunkte Teil-Netzwerke entsprechender Bauart (entsprechend den A1 bis Ak) zerlegen können. Dabei müssen diese Teilnetzwerke entsprechend dem Gleichungssystem der Spezifikation einerseits ihren Beitrag zur Schnittstelle des Netzwerkes leisten, und andererseits über gemeinsame Kanäle mit anderen Teil-Netzwerken verknüpft sein.

Logische Netzwerk-Spezifikationen

Das Spezifikations-Schema für rein logische Architekturen, d.h. Architekturen ohne technischen Anteil, ist analog zu obigem Schema für technische Architekturen aufgebaut, enthält aber das Schlüsselwort *logical network* an Stelle von *technical network*:

network A
input i1:I1, i2:I2, ..., im:Im
output o1:O1,o2:O2,..., on:On
is
logical network
<o11, ..., o1r> = A1 <i11, ..., i1s>;
. . .
<ok1, ..., okr> = Ak <ik1, ..., iks>;
end A

Abbildung 5.7: Spezifikation von Basismodellkomponenten mit ausschließlich logischen Netzwerken

Wieder analog zu oben interpretieren wir die Spezifikation einer logischen Architektur durch ein Prädikat

$$is\_A \in NW(\mathbf{CH}) \times \mathbf{CH}^m \times \mathbf{CH}^n \longrightarrow Bool$$

wobei wir das Verhaltens-Sicht-Prädikat diesmal wie folgt definieren:

$$\begin{aligned} behavior(nw, nw_{comp}) \stackrel{Def}{=} \quad & in.nw = in.nw_{comp} \wedge \\ & out.nw = out.nw_{comp} \wedge \\ & behav.nw = behav.nw_{comp} \end{aligned}$$

Im Unterschied zu einer technischen Spezifikation macht eine rein logische Architektur keine Aussage über die Menge der Komponenten eines Netzwerkes “nw”. Von einem Netzwerk “nw” fordern wir durch eine logische Spezifikation lediglich eine Schnittstellen- und Verhaltens-Eigenschaften, die ein Netzwerk bieten würde, welches entsprechend der Spezifikation aufgebaut wäre. Da wir, im Gegensatz zur technischen Spezifikation, von einem Netzwerk “nw” nicht fordern, dass sich dessen Komponenten-Menge entsprechend der Spezifikation partitionieren lassen muss, ist die spezifizierte Netzwerk-Struktur (gegeben in Form der Gleichungen) nur eine von vielen möglichen Netzwerk-Strukturen, die der Spezifikation genügen.

Spezifikationen mit technischen und logischen Anteilen

Das Schema für Spezifikationen mit technischem und logischem Anteil hat folgende Form:

network A
input i1:I1, i2:I2, ..., im:Im
output o1:O1,o2:O2,..., on:On
internal h1:H1,h2:H2,..., hp:Hp
behavior R(i1, ..., im, o1, ..., on)
is technical network
<o11, ..., o1r> = A1 <i11, ..., i1s>;
...
<ok1, ..., okr> = Ak <ik1, ..., iks>;
is logical network
<o(k+1)1, ..., o(k+1)r> = A(k+1) <i(k+1)1, ..., i(k+1)s>;
...
<o(k+j)1, ..., oljr> = A(k+j) <i(k+j)1, ..., i(k+j)s>;
end A.

Abbildung 5.8: Spezifikation einer Basismodellkomponente mit sowohl logischen als auch technischen Netzwerken

Ergänzend zu dieser Notation der Gleichungen um technische und logische Netzwerke zu definieren können auch Platzhalter verwendet werden. Diese werden durch die Schlüsselwörter *empty* (leer) und *any* (beliebig) vermerkt. Die Platzhalter dienen dazu, eine atomare Komponente oder eine Komponente mit beliebigem technischen oder logischen Netzwerk zu spezifizieren. Wir interpretieren eine Spezifikation mit technischen und logischen Anteilen wieder durch ein Prädikat

$$is\_A \in NW(\mathbf{CH}) \times \mathbf{CH}^m \times \mathbf{CH}^n \longrightarrow Bool$$

welches wir wie folgt definieren:

$$\begin{aligned} is\_A(nw, \overline{c_i}, \overline{c_o}) \quad \stackrel{Def}{=} \quad & \exists\, t \in \mathbf{N}, \\ & nw_{comp} \in NW(\mathbf{CH}) : \\ & blackbox(nw, \overline{c_i}, \overline{c_o}) \wedge \\ & glassbox(nw_{comp}, t) \wedge \\ & behavior(nw, nw_{comp}) \end{aligned}$$

wobei wir das Verhaltens-Sicht-Prädikat diesmal wie folgt definieren:

$$\begin{aligned} behavior(nw, nw_{comp}) \quad \stackrel{Def}{=} \quad & (\forall\, x \in 1, \ldots, k : comp.nw_x \subseteq comp.nw) \wedge \\ & in.nw = in.nw_{comp} \wedge out.nw = out.nw_{comp} \wedge \\ & behav.nw = behav.nw_{comp} \end{aligned}$$

Ein Netzwerk setzt sich wieder aus Teil-Netzwerken zusammen, die den Spezifikationen A1 bis A(k+j) entsprechen. Die Vorstellung, dass wir mit der technischen Architektur eine bestimmte Topologie des Netzwerkes festlegen (und nicht nur Black-Box-Eigenschaften wie die Schnittstelle und das Verhalten des gesamten Netzwerkes)

modellieren wir wie folgt: Für Netzwerke, die für A1 bis Ak, also für technisch interpretierte Spezifikationen eingesetzt werden, legen wir fest, dass diese eine Teilmenge der Komponenten des gesamten Netzwerkes ausmachen. Dies fordern wir nicht für Netzwerke, die für A(k+1) bis A(k+j), also logisch interpretierte Spezifikationen, stehen. Dies entspricht der Vorstellung, dass die logische Architektur keine Aussage über die interne Netzwerk-Struktur macht, sondern lediglich einen Beitrag zu den Black-Box-Eigenschaften (Schnittstelle und Verhalten) des gesamten Netzwerkes liefert.

Alle Teil-Netzwerke stehen wieder für disjunkte Teile des gesamten Netzwerkes. Die Teil-Netzwerke sind entsprechend den Gleichungen der Spezifikation komponiert.

Wie bisher muss ein Netzwerk “nw” in Schnittstelle und diesbezüglichem Verhalten einem Netzwerk entsprechen, welches genau der Spezifikation mit seinen technischen und logischen Anteilen entspricht. Bezüglich der Struktur von “nw” lassen uns die logischen Spezifikations-Anteile aber noch entsprechende Freiheiten. Dies spiegelt sich darin wieder, dass wir analog zur rein logischen Spezifikation auch hier nicht fordern, dass sich die Komponenten-Menge von “nw” entsprechend der Gleichungen für A1 bis A(k+j) partitionieren lassen muss.

5.3 Zusammenfassung

Wir haben in diesem Kapitel zum einen die mathematischen Grundprinzipien des Basismodells erklärt mit denen es uns möglich ist Verhalten und Strukturen auszudrücken. Verhalten wird in Form von Relationen zwischen Ein- und Ausgangskanälen ausgedrückt, wobei sich hierfür noch einige grafische Beschreibungstechniken anbieten, die darauf aufbauen. Als Beispiele hierfür seien MSCs oder STDs genannt, wie sie in [BS00] definiert werden. Die grundlegenden Mechanismen des Basismodells lauten:

Ströme und deren Relationen als mathematisches Mittel des Aufdrückens von Verhalten.

Komponenten und Kanäle als Basisabstraktion verteilter Systeme, wobei Ströme und Relationen als Verhaltensbeschreibung dienen.

Dienste als strukturfreie Spezifikation eines bestimmten Teilverhaltens bezüglich einer Teilsignatur. Beschreibungsmittel hierfür sind wiederum Ströme und Relationen bzw. darauf aufbauende grafische Beschreibungsformen wie STDs oder MSCs.

Komponentennetzwerke als Komposition von Komponenten und des daraus resultierenden Gesammtverhaltens, wobei sowohl Verhalten als auch Struktur durch die Spezifikation definiert werden können.

Zur Spezifikation solcher Komponentennetzwerke haben wir eine Schemanotation definiert, die drei Arten von Spezifikationen unterscheidet:

Technische Netzwerk-Spezifikation beschreibt die Menge an Komponentennetzwerken, die sowohl die Struktur, als auch das Verhalten von aussen (Blackbox) gemäß der Spezifikation besitzen, als auch deren interne Struktur (Glassbox) der Spezifikation gehorcht.

Logische Netzwerk-Spezifikation beschreibt die Menge an Komponentennetzwerken, die zwar die Struktur und das Verhalten von aussen (Blackbox) gemäß der Spezifikation besitzen, jedoch beliebige interne Struktur besitzen können.

Gemischte Netzwerk-Spezifikation beschreibt eine Spezifikation eines komponierten Netzwerkes, das sowohl technische- also auch logische- Netzwerkanteile im oben definierten Sinne besitzt.

KAPITEL

SECHS

Engineeringmodell

Obwohl das formale Basismodell vollständig ist und die hier behandelten Systemarten ausdrücken könnte, stellt sich die Frage, ob eine solche Abstraktion für die praktische Modellierung angebracht ist. Formale Modelle tendieren dazu, die auszudrückenden Sachverhalte auf möglichst wenig exakte Axiome abzubilden, um damit eine effiziente Verifizierung zu erlauben. Dadurch werden viele im praktischen Entwurf auftretenden Entitäten, zum Beispiel ein Ort, abstrahiert und implizit ausgedrückt, beispielsweise durch eine Komponente. Für eine adequate Abstraktion für den praktischen Entwurf ist es aber wichtig, manche oft auftretende Entitäten nicht nur implizit, sondern explizit ausdrückbar und diskutierbar zu machen. Dies ist die Aufgabe des Engineeringmodells.

Im Engineeringmodell werden keine neuen semantischen Konzepte eingeführt, sondern manche für den praktischen Entwurf benötigten Konzepte und Entitäten explizit gemacht, die im Basismodell nur implizit vorhanden sind. Dazu gehören unter anderem:

- **Hierarchische Komponenten** im Gegensatz zu flachen Netzwerken.
- **Explizite Orte und Umgebungsprofile**, statt spezieller Komponententypen.
- **Explizite Kommunikationsarten** (wie RMI, CORBA), statt unterschiedliche Kanaltypen.
- **Explizite Trennung** von Dienst- und Komponentenstruktur.
- **Explizite Zuordnung** von Komponenten zu Orten.

Das Engineeringmodell ist eine Abstraktion, die näher an dem Modell verteilter Softwarekomponenten wie CORBA oder JAVA-RMI ist. Dies trägt der oben, in Kapitel 4, erwähnten Frage nach dem *Aufwand* einer Abstraktion Rechnung. Ist es der Sinn des Basismodells, möglichst alle Sachverhalte präzise und eindeutig zu beschreiben, so ist die Zielsetzung des Engineeringmodells das möglichst effiziente und intuitive Ausdrücken der Sachverhalte ohne aufwendige Abstraktionsschritte.

Im Verlauf dieses Kapitels wird zunächst der Aufbau und das Konzept des Engineeringmodells erläutert, dessen Abbildung auf das formale Basismodell wird im anschließenden Kapitel beschrieben.

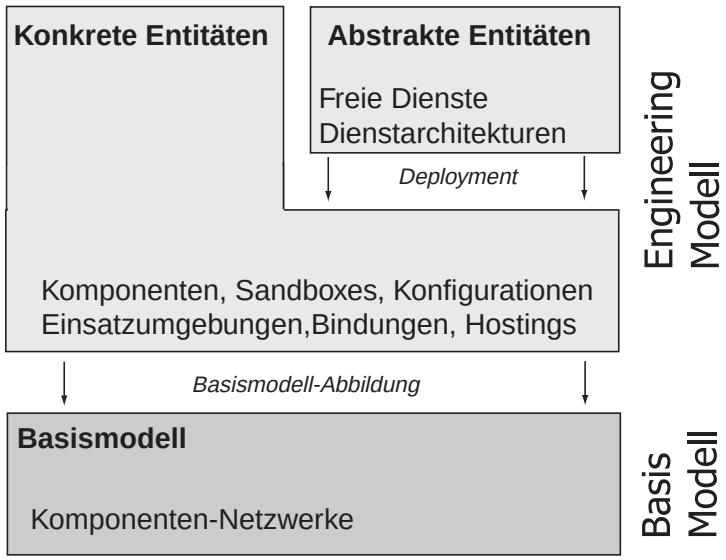


Abbildung 6.1: Abstrakte Entitäten des Engineeringmodells werden auf konkrete Entitäten durch das Deployment abgebildet.

Die Bestandteile des Engineering-Modells lassen sich in zwei Gruppen einteilen: *konkrete Entitäten* sind Entitäten, die bei der Implementierung durch konkrete Codeobjekte ersetzt werden. *Abstrakte Entitäten* sind nur Bestandteile des Systems während den frühen Phasen der Modellierung. Sie werden schon im Engineering-Modell, also vor der Implementierung, durch konkrete Entitäten ersetzt. Beispiele für abstrakte Entitäten sind Dienste oder Dienstarchitekturen, die die logische Architektur eines Szenarios beschreibt. Abstrakte Entitäten werden mittels der *Deployment Abbildung* durch konkrete Entitäten ersetzt. So wird beispielsweise ein Dienst durch eine Komponente ersetzt, die diesen Dienst erbringt. Die Beziehungen zwischen Entitäten, wie zum Beispiel *Bindungs*- (Kommunikationsbeziehungen) und *Hostingrelationen* (Ansiedlung einer Entität auf einer Sandbox) gelten zwischen beiderlei Gruppen, also auch explizit zwischen Diensten (abstrakt) und Komponenten (konkret).

Ein System wird sowohl mittels konkreter Entitäten modelliert, beispielsweise um eine existierende Legacy-Komponente zu repräsentieren, als auch durch abstrakte Entitäten. Die abstrakten Entitäten dienen vornehmlich dazu, *invariante Strukturen* des spontanen Systems, die unabhängig von wechselnden Teilnehmern oder Konfigurationen sind, zu modellieren. Die invarianten Systemstrukturen, sog. *Dienstarchitekturen*, repräsentieren die funktionalen Abhängigkeiten zwischen Rollen des Systems: es wird nicht spezifiziert, *wer* einen Dienst erbringt, sondern nur, dass dieser verfügbar sein muss. Diese absichtliche Unterspezifikation des Systems lässt jene Freiheitsgrade zu, die das spontane System für Umkonfiguration benötigt, wobei jedoch die invariante Dienstarchitektur die Funktionalität definiert. Die Dienstarchitektur besteht sowohl aus konkreten Entitäten, wie Komponenten und Sandboxes, als auch Surrogaten, den freien Diensten, die durch die Deployment-Abbildung durch Komponenten ersetzt werden, um eine Konfiguration zu definieren.

Die Deployment-Abbildung erhält als Parameter eine Menge an Dienstarchitekturen

und eine Einsatzumgebung, welche aus spezifizierten Komponenten und Sandboxes besteht. Beides zusammen wird dann auf die Menge der auf der Basis der Einsatzumgebung *möglichen technischen Konfigurationen* der jeweiligen Dienstarchitekturen abgebildet. Die abstrakten Entitäten der Dienstarchitektur werden dabei entweder durch Komponenten der Einsatzumgebung ersetzt oder aber sie ergeben eine zu implementierende Komponentenbeschreibung.

Auf Basis dieser Konfigurationen zu einer Dienstarchitektur ist es möglich, bestimmte Qualitätseigenschaften einzelner Konfigurationen, wie beispielsweise Stabilität, Sicherheit oder Erweiterbarkeit zu bestimmen.

Die bezüglich solcher Qualitätskriterien geeignetesten Konfigurationen können anschließend automatisiert auf Codeskelette der jeweiligen Plattform abgebildet werden. Dieser Ablauf wurde im Rahmen dieser Arbeit anhand einer Modellierungsumgebung prototypisch realisiert, wobei Komponenten in Form einer Definitionssprache oder mittels graphischen Werkzeugen spezifiziert werden konnten und exemplarisch auf die Plattform Java bzw. JINI abgebildet wurde (siehe hierzu Kapitel 10.2.4).

Die Konzepte und Elemente des Engineeringmodells sind formal durch die des Basismodells beschrieben und erlauben somit eine präzise formale Fundierung, falls dies gewünscht ist. Ein vergleichbarer Ansatz kann in der Sprache WRIGHT [All97] gefunden werden. Hier wurde eine sog. *Architectural Description Language* (kurz ADL) auf dem Basismodell CSP definiert. Obwohl eine ADL auf andere Fragestellungen als die hier behandelte abzielt, nämlich das Darstellen eines Systems als Menge von sog. Komponenten und Konnektoren durch reine Strukturinformation, ist die Schichtung mittels formalen Basismodell ähnlich.

Die Merkmale des hier vorgestellten Engineeringmodells, nämlich Mobilität und Dynamik, separate Spezifikation von logischer und technischer Architektur, sowie deren Zuordnung, wurden bereits erläutert. Diese Merkmale werden im Engineeringmodell folgendermaßen ausgedrückt:

- *Mobilität* und *dynamische Kommunikationsbeziehungen* werden in Form von Konfigurationen als Schnappschüsse (engl. Snapshots) und deren Übergänge modelliert.
- Die *logische Architektur*, welche vornehmlich die funktionalen Abhängigkeiten unabhängig von technischen Strukturen repräsentiert wird durch die so genannten *Dienstarchitekturen* ausgedrückt.
- Die (strukturbestimmende) *technische Architektur* wird durch eine *Konfiguration* ausgedrückt.
- Die *Zuordnung* einer Menge von Möglichen technischen Architekturen zu einer logischen Architektur wird durch die *Deployment Abbildung* eines Dienstarchitekturen und einer Einsatzumgebung (Komponenten- und Netzwerkspezifikationen) auf eine Menge von Konfigurationen hergestellt.

Diese Merkmale oder Eigenschaften werden noch einmal in der Tabelle 6.1 zusammengefasst. Diese Tabelle wird sukzessive in den kommenden Kapiteln erweitert um

Eigenschaft	Abbildung im Engineeringmodell
Mobile Softwarekomponente	Hierarchische Komponente auf Sandbox angesiedelt. Mobilität durch Folge von Konfigurationen
Funktionale Abhängigkeit innerhalb logischer Architektur	Services: Freier Service als Surrogat für eine Komponente oder als Teilfunktionalität der Komponenten (needed bzw. provided).
Rechteumgebungen und damit verbundene Orte (locations).	Sandboxes: Verwalten Kommunikationsrechte zu Komponenten ausserhalb der eigenen Grenzen.
Logische Architekturspezifikation	Servicearchitecture: Szenario aus Komponenten, freien Services und Sandboxes
Technische Architekturspezifikation	Configuration: Szenario ausschliesslich aus Komponenten und Sandboxes

Tabelle 6.1: Die Merkmale spontaner Komponentensysteme und die Abbildung auf ihre Pendants im Engineeringmodell (siehe auch Tabelle 7.1 und 9.2).

die jeweiligen Pendants im Basismodell und in der Realisierungsplattform. In den folgenden Abschnitten werden die einzelnen Elemente des Engineeringmodells erklärt. Hierbei wird unter anderem eine Schemanotation verwendet, die sich folgendermaßen aufbaut:

ElementName : *TypeID*

AttributA `Typ`
AttributB : `<Vektor von Typen>`
AttributC : Funktion DOM $\rightarrow$ RAN

6.1 Konkrete Entitäten

6.1.1 Komponenten und deren Dienste

Ein Dienst ist, neben der Komponente, die zentrale Instanz des Engineeringmodells und tritt sowohl in logischen, als auch technischen Architekturspezifikationen, auf. In einer logischen Architekturspezifikation kann ein Dienst als *freier Dienst* oder in Form eines *Dienstes einer Komponente* (benötigter oder angebotener Dienst) auftreten. In

einer technischen Architektur nur in letzterer Form. Der Komponentenbegriff ist also eng mit dem des *Dienst* verwoben. Zunächst muss jedoch kurz ein Hilfskonstrukt, der so genannte *Calltype* definiert werden. Der eindeutige Calltype identifiziert ein Kommunikationsprotokoll, wie Beispielsweise IIOP (CORBA) oder HTTP.

Definition 6.0: Calltype
Ein Calltype $ct \in \mathbf{CT}_E$ ist ein Typbezeichner, der eine Kommunikationsart eindeutig festlegt.

Ein Dienst ist eine strukturunabhängige Beschreibung eines Verhaltens, das als Teilverhalten einer Komponente auftritt. Das Verhalten wird also lediglich bezüglich einer Signatur spezifiziert und bietet keine internen Abläufe. Um das Verhalten gegen eine Signatur spezifizieren zu können, muss zunächst der Begriff der Signatur (Interface) definiert werden:

Definition 6.1: Interface
Ein Interface $int \in \mathbf{IF}_E$, ist eine Signatur, bestehend aus einem eindeutigen Bezeichner $id \in \mathbf{ID}$Eingabeport (?i) und einem Ausgabeport (!o) , also jeweils einem Kanal mit Zugriffsrecht, sowie einem Calltype, der die Kommunikationsart festlegt.

Zur Definition eines Ports siehe Abschnitt 5.1.1. Als Hilfskonstrukt definieren wir nun den Typ eines Interfaces, indem wir jedem Interface seinen Eingabeporttyp, Ausgabeporttyp und den Calltype zuordnen:

Definition 6.2: Interfacetyp
Ein Interfacetyp $inttype \in \mathbf{TIF}_E$ besteht aus einem eindeutigen Bezeichner $id \in \mathbf{ID}$ einem Eingabeport-Typ und einem Ausgabeport-Typ, sowie einem Calltype.

Welcher Zuschnitt des Verhaltens als Dienst definiert wird hängt sehr stark von der Anwendungsdomäne ab. Es macht beispielsweise Sinn oft wiederkehrende und semantisch sinnvolle Funktionalitäten in Bezug zu einer Anwendungsdomäne als Dienste zu modellieren.

Definition 6.3: Dienst
Ein *Dienst* $s \in \mathbf{S}_E$ ist eine wiederverwendbare, strukturunabhängige Spezifikation eines Teilverhaltens bezüglich einer Signatur (Interface). Ein Dienst besitzt Definitionen der funktionalen Abhängigkeiten, die die Anforderungen an die Umgebung in Form von benötigten Dienste (needed Services) festgelegt.

Eine Dienstspezifikation besteht aus:

- Bezeichner $id \in \mathbf{ID}$
- Signatur (Interface) $int \in \mathbf{IF}_E$
- Blackbox-Verhalten $F : \overrightarrow{?i.int} \rightarrow \overrightarrow{!o.int}$
- Anforderungen (needed Services) $NS \subseteq \mathbf{S}_E$

Die Wiederverwendbarkeit des Dienstes bindet diesen also stark an die Anwendungsdomäne. Tritt eine Funktionalität in einer Domäne, beispielsweise Telefonie, oft auf

und sollte daher als Dienst modelliert werden, so kann dieser Dienst in einer anderen Domäne isoliert auftreten und daher unangebracht sein. Die Weise des Schneidens der Dienste ist also ähnlich wie bei Komponenten stark von der Anwendungsdomäne abhängig.

Ein Dienst besteht aus einer Verhaltensspezifikation, einer Menge an von ihm benötigter Dienste, sowie einer Menge an Interfaces, die zu den jeweiligen Dienst -Nutzern oder -Anbietern gehen. Die Interfaces wiederum legen jeweils die Kommunikationssignatur fest, d.h. einen Ein- und einen Ausgabe-Port, die wiederum die Nachrichtentypen bestimmen.

Analog zum Interfacetyp definieren wir nun den Diensttyp:

Definition 6.4: Diensttyp

Jedem Dienst kann ein eindeutiger *Diensttyp* $ts \in \mathbf{TS}_E$ zugeordnet werden. Der Diensttyp ist definiert durch:

- Bezeichner $id \in \mathbf{ID}$
- Eine Menge an Interfacetypen $int \in \mathbf{IF}_E$
- Blackbox-Verhalten $F : ?\overrightarrow{i.int} \rightarrow !\overrightarrow{o.int}$
- Einer Menge an benötigten Diensttypen $NST \subseteq \mathbf{TS}_E$

Ein Diensttyp sei durch folgendes Schema beschrieben:

Servicetype :*TypeID*

Behavior :	`Specification`
NeededServices	`<Service>`
Interfaces :	`<Interface>`

Die dazugehörigen Interfaces sind für je einen benötigten Dienst (needed Service) bestimmt, plus ein Interface zu dem Dienstnutzer. Sie bestehen jeweils aus Ein- und Ausgabeport, sowie einem Calltype. Der Calltype legt das Kommunikationsprotokoll fest, welches die Schnittstelle verlangt. Dies könnten beispielsweise IIOP, http, oder lokale Aufrufe sein. Anhand eines solchen Calltypes kann eine Verbindung durch eine Sandbox blockiert werden, was einer Firewall entspricht, die ein bestimmtes Protokoll blockiert, andere jedoch passieren lässt.

Interface :*ID*

InputPort :	`Port`
OutputPort	`Port`
Calltype :	`Calltype`

Ein Port setzt wiederum fest, welche Nachrichtentypen (engl. Messagetypes) über ihn fließen können.

Der jeweilige Dienst kann also durch Instanziierung eines Diensttypen mit einem eindeutigen Bezeichner definiert werden.

Ein Dienst kann eine Menge an Diensten benötigen, um die Funktionalität erbringen zu können. Ein Dienst *computeAccountBalance* kann beispielsweise den aktuellen Kontostand eines Kunden zurückgeben. Da für diese Funktionalität keine externe Funktion notwendig ist, gilt, dass die Menge der benötigten Dienste die leer Menge ist. Bei einem Dienst *spellcheck*, welcher auf einen übergebenen Text eine Rechtschreibkorrektur anwendet, kann es sein, dass dieser dazu ein Wörterbuch benötigt. In diesem Falle gilt also, dass die Menge der benötigten Dienste des Dienstes *spellcheck* ein Dienst *dictionary* wäre, der wiederum als Dienst spezifiziert sein muss.

Beispiel: Wir greifen wieder ein Beispiel aus Abschnitt 4.3 auf. Der Dienst Print wird von verschiedenen Geräten angeboten (z.B. Fax, Printer etc.). Er bietet die Möglichkeit, ein Dokument zu drucken, hierbei Einstellungen wie Auflösung etc. (so genanntes "Pagesetup") vorzunehmen und die Anzahl der Kopien festzusetzen. Der Dienst wird folgendermaßen spezifiziert:

Service :*Print*
Behavior : `Specification`
NeededServices `<>`
Interface : `Print_Interface`

Das dazugehörige Interface, welches die Signatur und die Kommunikationstypen (Messagetypes und dazugehörige Parametertyps) festlegt, lautet wie folgt:

Interface :*Print_Interface*
InputPort : `PrintInPort`
OutputPort `PrintOutPort`
Calltype : `RMI`

Port :*PrintInPort*
MesageTypes :
```
<setDocument(String),setCopies(int),
 setPagesetup(String),start(),cancel(),
 pause(),resume()>
```

Port :*PrintOutPort*
MessageTypes : `<start(boolean)>`

Das Verhalten bezüglich dieses Interfaces wird hier in Form eines Zustandsübergangsdiagramms (engl. State Transition Diagram STD) definiert (siehe Abbildung 6.2). Diese Notation , ähnlich eines Automaten, kann automatisiert in die Prädikatenform der im Basismodell verwendeten Verhaltensfunktion *F* umgewandelt werden. Siehe hierzu [BS00].

Während Dienste strukturlose Träger des Verhaltens sind, so werden durch *Komponenten* Dienste gruppiert und mit Strukturinformationen versehen. In Komponenten treten

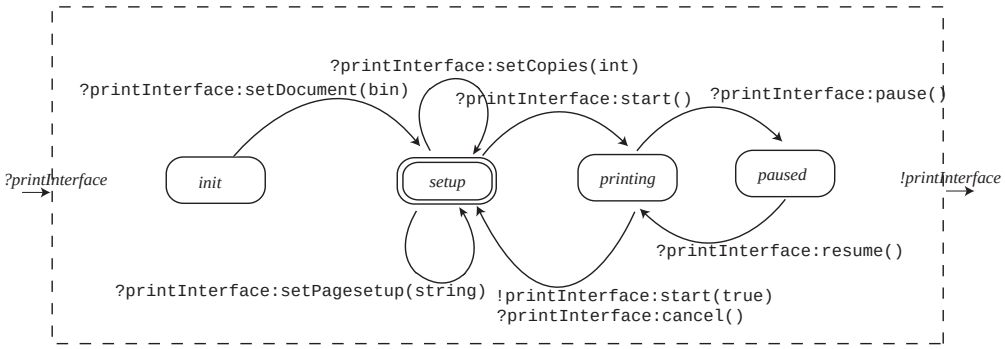


Abbildung 6.2: Das Blackbox-Verhalten des “Print” Dienste

Dienste als *Needed* und *Provided Services* auf, d.h. benötigte und angebotene Funktionalitäten und stellen die einzelnen Teilverhalten einer Komponente dar, die sie nach aussen anbietet oder von der Aussenwelt benötigt. Seit $\mathbf{S}_E$ die Menge aller Dienste:

- $NS \subseteq \mathbf{S}_E$ Die benötigten Dienste (needed services) und
- $PS \subseteq \mathbf{S}_E$ die angebotenen Dienste (provided services),
- $D \in \mathbf{S}_E \rightarrow \wp(\mathbf{S}_E)$ Ihre Abhängigkeitsfunktion (dependency function)

Im allgemeinen bietet und benötigt eine Komponente nicht nur eine, sondern eine Menge von Einzelfunktionalitäten (Dienste) an.

Für jede dieser Funktionalitäten (Dienste) wird nicht nur eine (unidirektionale) Verbindung benötigt, sondern eine sogenannte *Bindung*, bestehend aus einem Eingabe- und einem Ausgabe-Kanal. So benötigt zum Beispiel das Dienst *transmitDataTo*(c_{id}, d), welches Daten d zu einem Empfänger c_{id} übermitteln soll, zwei Ports, nämlich einen Eingabeport um die Anfragen zu empfangen, sowie einen Ausgabeport, der die gewünschten Daten d übermittelt. Zur genaueren Definition der Bindung siehe Abschnitt 6.1.3.

Die Subunits einer Komponente in Verbindung mit deren Bindungsrelationen stellen die Glassboxsicht der Komponente dar. Diese Elemente werden genutzt, um strukturelle Anforderungen an die Realisierung einer Blackbox-Sicht eines Komponententyps zu spezifizieren.[1]

- $SU \subseteq \mathbf{C}_E$ die Menge der Subkomponenten
- *SUB* die Bindungen der Subkomponenten (siehe Abschnitt 6.1.3 *Bindung*)

So bedeutet beispielsweise eine leere Menge von Subunits, dass die Implementierung des Komponententyps durch eine einzige Basismodellkomponente geschehen muss, d.h. durch ein atomares Komponenten-Netzwerk (siehe Abschnitt 5.2.1). Im Falle einer willkürlich angeordneten Menge von Subunits, d.h. einer Menge ohne jegliche

[1] Diese Strukturinformation fehlt dem Dienst. Somit kann ein Dienst als Komponente ohne Strukturinformation betrachtet werden (siehe unten).

Bindungs- oder Hosting-Relationen, drückt man einen Komponententyp mit einem speziellen Blackbox-Verhalten aus, bestehend aus bestimmten Subunits, jedoch ohne jegliche Einschränkung bezüglich deren Komposition.

Definition 6.5: Komponente

Eine *Komponente* $c \in \mathbf{C}_E$ des Engineeringmodells ist eine unabhängige, hierarchisch strukturierte und kompositionale Softwareeinheit, deren Verhalten sich in Dienste zerlegen lässt. Eine Komponentenspezifikation enthält im einzelnen:

- Einen eindeutigen Bezeichner $id \in \mathbf{ID}$
- Angebotenen Diensten (Provided Services)
- Unterkomponenten (Subcomponents)
- Topologie der Unterkomponenten (Subcomponent Bindings)

Eine Engineeringmodell-Komponente entspricht also einem flachen Basismodell-Netzwerk, das zum einen der internen Struktur, die durch die Topologie definiert ist, gehorcht (technisches Netzwerk) und dessen Blackbox-Verhalten sich gemäß der Verhalten der angebotenen Dienste (provided Services) in die entsprechenden Teilverhalten zerlegen lässt. Das Blackbox-Verhalten einer Engineeringmodell-Komponente ist also immer in Teilverhalten (Dienste) zerlegbar, bzw. wird durch Komposition der Teilverhalten (Dienste) definiert. Damit ist der Komponentenbegriff des Engineeringmodells ein *dienstorientierter Komponentenbegriff*, gemäß der Definition in Abschnitt 5.1.3.

Eine Komponente besitzt also eine Menge an benötigten und eine Menge an angebotenen Diensten. Die Dienste selbst werden gesondert spezifiziert und definieren implizit das Verhalten der Komponente. Des Weiteren ergibt sich dadurch eine Dienstabhängigkeitsfunktion der Komponente, die angibt, welche benötigten Dienste von welchem angebotenen induziert werden. Die Strukturinformation der Komponente wird durch eine Menge von Subkomponenten und deren Verbindungen definiert, was dem internen Aufbau der Komponente, also einer Glassboxdefinition, entspricht. Diese Anwendung von Dienste kann somit als eine Erweiterung und konsequente Nutzung des Interface-Konzeptes, wie es zum Beispiel in den Programmiersprache Java [AaDH00] oder Objective-C [Inc92] Anwendung findet, gesehen werden. Jedoch stellen die Interfaces in beiden dieser Fälle nur die angebotenen Signaturen, also in dem hier behandelten Modell die Provided-Dienste, dar und bestehen darüberhinaus nur aus Strukturbeschreibung ohne Verhalten.

Analog zum Dienst definieren wir wieder ein Begriff des Komponententyps:

Definition 6.6: Komponententyp

Einer Komponente $c \in \mathbf{C}_E$ des Engineeringmodells kann ein eindeutiger *Komponententyp* $tc \in \mathbf{TC}_E$ zugeordnet werden. Er besteht aus:

- Einen eindeutigen Bezeichner $id \in \mathbf{ID}$
- Einer Menge angebotener Diensttypen (Provided Services)
- Typen der Unterkomponenten (Subcomponenttypes)
- Topologie der Unterkomponenten (Subcomponent Binding Types)

Zu einer Definition der Abbildung einer Komponente auf das Basismodell siehe Abschnitt 7.2

Um einen Komponententyp kompakt zu spezifizieren, benutzt man die folgende Schemanotation:

Componenttype :*TypeID*
NeededServices <Servicetype>
ProvidedServices : <Servicetype>
SDependency :$PS \rightarrow \mathcal{P}(NS)$
Subcomponents : <componenttype>
SubcomponentBindings : <Bindingtype>

Hierbei sei beachtet, dass die benötigen Dienste (needed Services), sowie die Abhängigkeit zwischen der Diensten (SDependency) bereits implizit durch die Dienste angegeben sind. Sie werden hier nur zur Verdeutlichung noch einmal aufgeführt.

Beispiel: Wir betrachten das Mobiltelefon aus dem Fallbeispiel in Abschnitt 4.3. Das Mobiltelefon bestand aus drei Komponenten: Einer Telefonbuchkomponente, die das interne Telefonbuch regelt, einer SMS Komponente, die den SendSMS-Dienst anbot, und einer Komponente, die den Dienst SendSMSbyName durch Bündelung des Telefonbuchdienstes und des SendSMS Dienstes realisierte. Die Komponente ist graphisch in der Abbildung 6.3 abgebildet und im folgenden Schema definiert:

Componenttype :*Mobiltelefon*
NeededServices : <Phonebook>
ProvidedServices : <SendSMS,Phonebook,SendSMSbyName>
SDependency :(SendSMSbyName;Phonebook)
Subunits :
<InternesTelefonbuch,SMSManager,SMSbyNameManager>
Subunitsbindings :
<(InternesTelefonbuch,Mobiltelefon,Local,Phonebook),
(SMSManager,Mobiltelefon,Local,SendSMS),
(SMSManager,SMSByNameManager,Local,SendSMS),
(SMSbyNameManager,Mobiltelefon,Local,SendSMSbyName),
(Mobiltelefon,SMSByNameManager,Local,Phonebook)>

Man beachte, dass Komponenten des Engineeringmodells lediglich Träger von Strukturinformation sind. Das Verhalten der Komponenten ist implizit durch die Verhalten der angebotenen Dienste definiert.

6.1.2 Units

Da, wie in späteren Abschnitten noch erläutert wird, unter Umständen ein Dienst auch die Rolle eines Platzhalters für eine Komponente (sog. *freie Dienste*) einnehmen kann (siehe Abschnitt 6.2.2) wird zur Vereinfachung der Begriff der *Unit* eingeführt. Units sind die Obermenge von Komponenten und Dienste.

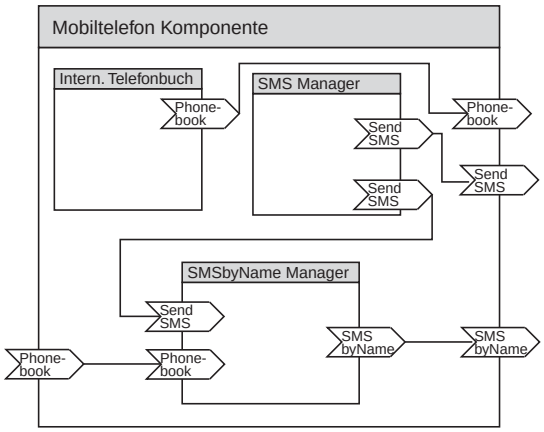


Abbildung 6.3: Struktur der Mobiltelefon-Komponente aus Fallbeispiel 4.3

Definition 6.7: Unit und Unit-Typ
Eine *Unit* $u \in \mathbf{U}_E = \mathbf{S}_E \cup \mathbf{C}_E$ ist ein Oberbegriff für Komponenten und Dienste.

Analog werden die Typen von Komponenten und Diensten zur Menge der Unit-Typen $\mathbf{TU}_E = \mathbf{TS}_E \cup \mathbf{TC}_E$ vereinigt.

6.1.3 Bindung

Eine *Bindung* (engl. Binding) ist der einzige Verbindungs- und Kommunikationsmechanismus im Engineeringmodell. Sollen zwei Komponenten Nachrichten austauschen können, so muss eine Bindung zwischen ihnen existieren. Bindungen existieren zwischen Komponenten nur bezüglich eines Dienstes, wobei der Dienst von der einen Komponente angeboten, und von der anderen benötigt werden muss.

Im Engineeringmodell werden Units bezüglich eines Dienstes gebunden, um die Nachrichten, die für die Nutzung eines Dienstes notwendig sind, auszutauschen. Im Falle von Komponenten muss dies ein Dienst sein, den die Komponente anbietet oder benötigt. Im Falle eines freien Dienste (siehe Abschnitt 6.2.1 und 6.2.2) kann es nur der Dienst selbst sein, bezüglich dessen gebunden wird. Die Bindung beschreibt also – analog zum Basismodell – eine Relation zwischen Units. Hierbei ist die Verbindung bezüglich eines *CallTypes* definiert, der die Kommunikationsform (z.B. RMI,CORBA, Lokal etc.) festlegt.

Definition 6.8: Bindung
Eine Bindung $b \in \mathbf{B}_E$ ist eine gerichtete Verbindung zwischen zwei Diensten, einem angeboten Dienst S_q und einem benötigten Dienst S_s, die einen bestimmten Kommunikationstyp (engl. Calltype) $ct \in \mathbf{CT}_E$ besitzt.

Eine Bindungsspezifikation besitzt demnach folgende Informationen:

- Einen Dienst $s_b \in \mathbf{S}_E$ bezüglich dem gebunden wird (Dienst)
- Eine Unit $src \in \mathbf{U}_E$ mit angebotenem Dienst des Typs $s_b \in \mathbf{TS}_E$ als Quelle (Source),

- Eine Unit $drn \in \mathbf{U}_E$ mit benötigtem Dienst des Typs $s_b \in \mathbf{TS}_E$ als Senke (Drain)
- Einen Kommunikationstyp $ct \in \mathbf{CT}_E$ (Calltype)

Eine Bindung besteht demzufolge aus einer Quelle (Source), einer Senke (Drain) und einem Dienst, bezüglich dessen beide kommunizieren. Darüberhinaus besitzt eine Bindung einen Calltype, der die Kommunikationsart (z.B. das Protokoll) identifiziert. Eine Bindung wird als eigenständige Entität im Modell behandelt:

Binding :*ID*
Source : `Unit`
Drain : `Unit`
CallType : `CallType`
Service : `Service`

Man beachte, dass in der Definition des Dienstes, bezüglich dessen gebunden wird, die Signatur des Dienstes und damit der Bindung enthalten ist. Diese Signatur besteht aus zwei unidirektionalen Ports (In- und Out-Port), die wiederum durch ihren Typ die Nachrichtentypen, die über ihre Kanäle und damit über die Bindung fließen, festlegen. Siehe dazu Abschnitt 6.2.1

Einen Bindungstypen (engl. Bindingtype) definieren wir analog, wobei an die Stelle der jeweiligen Dienste und Units Diensttypen und Unit-Typen treten:

Definition 6.9: Bindungstyp
Ein Bindungstyp $bt \in \mathbf{TB}_E$ ist definiert durch:

- Einen Diensttyp $ts_b \in \mathbf{TS}_E$ bezüglich dem gebunden wird (Diensttyp)
- Einen Unit-Typ $src \in \mathbf{TU}_E$ mit angebotenem Dienst des Typs $s_b \in \mathbf{TS}_E$ als Quelle
- Eine Unit-Typ $drn \in \mathbf{TU}_E$ mit benötigtem Dienst des Typs $s_b \in \mathbf{TS}_E$ als Senke
- Einen Kommunikationstyp $ct \in \mathbf{CT}_E$ (Calltype)

6.1.4 Sandboxes und Hostings

Eine Sandbox ist eine Wirtsplattform für Units, d.h. Komponenten und Dienste, welche die Kommunikationsrechte zu Units auf anderen Sandboxes verwaltet. Eine Sandbox entspricht dem in Abschnitt 3.2.1 beschriebenen *Umgebungsprofil*, das in den Entwurf spontaner Komponentensysteme miteinbezogen werden muss, da es die technische Architektur mitbestimmen kann.

Jede Unit innerhalb eines Systems muss auf einer Sandbox angesiedelt sein, was durch die *hosting* Relation (siehe unten) ausgedrückt wird. Die Menge aller Sandboxes eines Systems wird mit $\mathbf{SBX}_E$ bezeichnet. Sandboxes sind verschachtelt, d.h. Sandboxes können andere Sandboxes beinhalten und bilden somit einen Baum, wobei *ether* die Wurzel-Sandbox bezeichnet (siehe Abbildung 6.4). Man beachte hierbei, dass es sich um zwei verschiedene Relationen für die Schachtelung der Sandboxes und für das Enthalten von Units handelt. Dies ermöglicht es, Units zu modellieren, die in einem Netz-

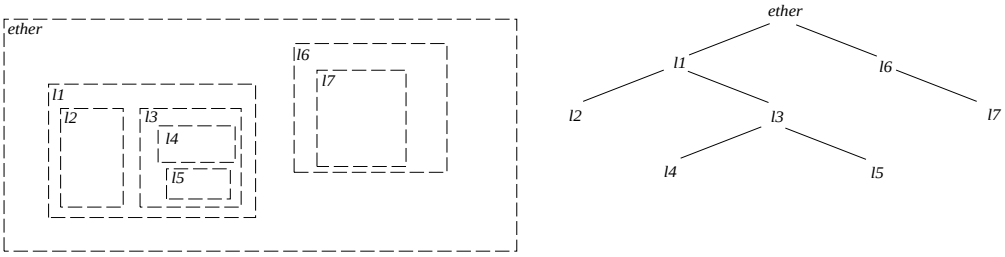


Abbildung 6.4: Verschachtelte Sandboxes

werk angesiedelt sind, das durch einen speziellen Gateway (siehe Glossar), der nur bestimmte Kommunikationstypen passieren lässt, von anderen Netzwerken, und damit anderen Units, abgetrennt ist. Wenn eine Unit nun eine Verbindung zu einer Unit in einer zweiten Sandbox etablieren will, d.h. ein Kanal zwischen beiden geschaltet werden soll, der die jeweiligen Sandboxgrenzen überschreitet, so muss der jeweilige Kanaltyp bei allen durchkreuzenden Sandboxen freigeschaltet sein. Dies bedeutet im Beispiel, dass eine Komponente auf einem Rechner hinter einer Firewall einen http-Request an Komponenten hinter der Firewall schicken kann, jedoch nicht IIOP (CORBA Protokoll) Aufrufe.

Definition 6.10: Sandbox

Eine Sandbox $sbx \in \mathbf{SBX}_E$ ist eine spezielle Komponentenart, die hierarchisch angeordnet ist. Jede Sandbox ist definiert durch:

- Eine übergeordnete Sandbox $parent \in \mathbf{SBX}_E$
- Eine Menge von enthaltenen Sandboxes $sbx_i, \ldots, sbx_k \in \mathbf{SBX}_E$
- Eine Menge von blockierten Bindungs-Typen $b_u, \ldots, b_v \in \mathbf{TB}_E$

Sandboxes repräsentieren die Wirtsplattformen, auf denen Komponenten angesiedelt sein müssen.Jede Unit muss auf genau einer Sandbox angesiedelt sein. Die Sandbox kann wiederum Unter-Sandboxes besitzen und definiert somit eine hierarchische Struktur. Eine Sandbox setzt die Kommunikationsrechte der auf ihr angesiedelten Units fest, indem sie eine Liste von *Blocked-Binding-Types* besitzt, die definiert, welche Bindungs-Typen zwischen Komponenten der Sandbox und Komponenten auf anderen Sandboxes blockiert werden.

Eine Sandbox wird durch das folgende Schema definiert:

Sandbox :*ID*

Parent : `Sandbox`
Children : `<Sandbox>`
BlockedBindingTypes : `<Binding>`

Zusammenfassend kann man sagen, dass im Engineeringmodell die Units Träger der Funktionalität sind, Sandboxes Träger der Kommunikationsrechte innerhalb des Netzes. Beide müssen dementsprechend modelliert werden.

Beispiel: Wir betrachten wieder das Fallbeispiel aus Abschnitt 4.3 und greifen das Faxgerät auf, das hinter einer Firewall gelagert ist, die RMI Aufrufe blockiert.

Sandbox :*Subnet_Fax*

Parent : `Intranet`
Children : `<>`
BlockedBindingTypes : `<(*,*,RMI,*)>`

Man beachte, dass der Asteriskus (*) einen beliebigen Wert repräsentiert. Im obigen Beispiel bedeutet dies, dass sämtliche Bindungen, egal von welcher Komponente sie kommen oder an welche Komponente sie gerichtet werden, blockiert werden, sofern sie vom Calltype "RMI" sind. Auch der Dienst, bezüglich dessen sie gebunden werden, ist irrelevant.

Der oben definierte Bindungs-Typ setzt also insbesondere auch die Kriterien fest, nach denen selektiert werden kann. In der momentanen Realisierung (siehe Abschnitt 10) ist die der Komponententyp der Quelle, sowie der Senke, Calltype und Diensttyp.

Den Sandboxes entsprechend wird eine Ordnung $\geq$ definiert, die angibt ob eine Sandbox alle Bindungen passieren lässt, die eine andere Sandbox passieren lässt:

Definition 6.11: Restriktion $\geq$

Auf der Menge der Sandboxes $\mathbf{SBX}_E$ wird die Ordnungsrelation $\geq$ definiert, die angibt ob eine Sandbox alle Bindungstypen passieren lässt, die eine andere passieren lässt.

$$sbx_i, sbx_k \in \mathbf{SBX}_E : sbx_i \geq sbx_k \quad \Leftrightarrow \quad \neg(\exists\, bt : bt \in BT.sbx_i \wedge bt \notin BT.sbx_k)$$

wobei für $sbx \in \mathbf{SBX}_E$, $BT.sbx \in \mathbf{TB}_E$ die Menge der Blockierten Bindungstypen der Sandbox *sbx* ist.

Die Hosting Relation gibt an, welche Units auf einer Sandbox angesiedelt sind.

Definition 6.12: Hosting

Ein *Hosting* $h \in \mathbf{H}_E$ ist eine Relation, die angibt, welche Units auf einer Sandbox angesiedelt sind. Ein Hosting besteht also aus:

- Einer Sandbox $host \in \mathbf{SBX}_E$
- Einer Menge an Units, $unis = u_1, \ldots, u_k,\ u_i \in \mathbf{U}_E$, die auf der Sandbox *host* angesiedelt sind.

Die Schemanotation einer Hostingrelation ist folgendermaßen definiert:

Hosting :*ID*

Unit : `<Unit>`
Host : `Sandbox`

6.1.5 Konfigurationen und Einsatzumgebungen

Abschließend werden nun noch Einsatzumgebungen und Konfigurationen definiert. Einsatzumgebungen sind ein Hilfskonstrukt, das Komponenten und Sandboxes beinhaltet und somit eine existierende Umgebung, zum Beispiel Legacy Komponenten

und vorhandene Netzwerkinfrastruktur, als eine Engineeringmodell-Spezifikation ausdrückt. Beide werden später für die Abbildung einer logischen auf eine Menge von technischen Architekturen benötigt.

Definition 6.13: Einsatzumgebung

Eine *Einsatzumgebung* (engl. Environment) $eu \in \mathbf{EU}_E$ ist eine Menge von Komponenten- und Sandbox-Spezifikationen existierender Komponenten (z.B. Legacy Komponenten) und Umgebungen (z.B. Netzwerkumgebung) :

$$eu = (C, S),\ \textit{wobei}\ C \subseteq \mathbf{C}_E\ \textit{und}\ S \subseteq \mathbf{SBX}_E$$

C ist hierbei die Menge der Komponenten, *S* die Netzwerkumgebung bestehend aus einer Menge an Sandboxes. Man beachte, dass es sich hierbei nur um Komponenten, d.h. explizit nicht um Dienste handelt.

Environment :*ID*
Components : <Component>
Sandboxes : <Sandbox>

Eine *Konfiguration* ist eine Relation, die eine Einsatzumgebung auf die Einsatzumgebung verbunden mit einer Menge von Bindungs- und Hostingrelationen abbildet. Eine Konfiguration wird durch das folgende Schema beschrieben. Man beachte, dass im Vergleich zu einer Dienstarchitektur (siehe unten) keine Dienste frei als Units auftreten können.

Definition 6.14: Konfiguration

Eine *Konfiguration* ist die Spezifikation einer *technischen Architektur* eines Systems. Eine Konfiguration $k \in \mathbf{K}_E$ besteht aus:

- Einer Menge an Komponenten $Kom \subseteq \mathbf{C}_E$
- Einer Menge an Sandboxen $Snd \subseteq \mathbf{SBX}_E$
- Einer Menge an Bindungen $Bnd \subseteq \mathbf{B}_E$
- Einer Menge an Hostings $Hst \subseteq \mathbf{H}_E$

Eine Konfiguration wird als *gültig* bezeichnet, wenn kein Binding einem Blocked-Binding einer Sandbox entspricht.

Betrachtet man als die vollständige Spezifikation einer Konfiguration, also die direkt durch das Konfiguration-Schema und die indirekt durch die Component-, Sandbox-, Dienst- etc. Schemata spezifizierten Informationen, so stellt eine Konfiguration eine technische Architektur eines Systems dar.

Eine Konfiguration wird durch das folgende Schema definiert:

Configuration :*ID*
Components : <Component>
Sandboxes : <Sandbox>
Bindings : <Binding>
Hostings : <Hosting>

6.2 Abstrakte Entitäten

Technische Architekturspezifikationen (Konfigurationen), die ausschließlich aus konkreten Entitäten bestehen, sind letztendlich das Ziel eines Entwurfs. Um jedoch die verschiedenen dynamischen Eigenschaften in einer Spezifikation auszudrücken, halten wir invariante Strukturen in einer logischen Architektur fest, die dann auf die verschiedenen technischen Architekturen (also Konfigurationen) abgebildet wird. In dieser logischen Architektur werden die Spielräume, die die Unterspezifikation ausdrücken, durch abstrakte Entitäten spezifiziert, die dann durch die spätere Deployment-Abbildung auf konkrete Entitäten in der technischen Architektur, ersetzt werden.

6.2.1 Freie Dienste

Dienste treten im Entwurf in zwei Zusammenhängen auf: Ein Dienst existiert als Teilfunktionalität einer Komponente, die diese anbietet (Provided Service) oder benötigt (Needed Service). Des Weiteren kann ein Dienst in einer logischen Architektur (Dienstarchitektur) als *freier Dienst* auftreten und damit als Platzhalter für spätere Komponenten stehen. Dadurch wird ein gewisser Freiheitsgrad für die Abbildung der logischen auf technische Architekturen spezifiziert.

Der zweite Anwendungsfall von Dienste bei der Modellierung spontaner Komponentensysteme ist deren Nutzung als Komponenten ohne Glassboxsicht. Hierbei werden Dienste neben Komponententypen als Komponentenersatz in der Modellierung verwendet um “Platzhalter” für eine spätere Realisierung zu setzen. Dienste stellen damit lediglich die funktionale Abhängigkeit dar, jedoch keine Strukturinformation: welche Komponente mit welcher Struktur (Glassbox-Sicht) das Dienst beider späteren Realisierung ersetzt, wird nicht festgelegt, sondern bewusst offen gelassen. Es handelt sich somit um eine bewusste *Unterspezifikation*, bei welcher der Funktionsfluss spezifiziert wird, aber nicht seine konkrete Konfiguration.

So war in dem in Abschnitt 4.3 vorgestellten Beispiel bekannt, dass von der Komponente *Mobiltelefon* der Dienst *Phonebook* für die Erbringung des Dienstes *SendSMSbyName* benötigt wurde. Es ist also möglich diesen Dienst als Platzhalter für eine Komponente, die den Dienst erbringt, in diesem Beispiel der CD-ROM Laptop oder sogar das Mobiltelefon selbst, zu behandeln. Es wird also ausgedrückt, welche funktionalen Abhängigkeiten existieren, was benötigt wird und wie es zu erreichen sein muss – jedoch *nicht*, wer letztendlich den Dienst erbringt. Dies ist von Konfiguration zu Konfiguration verschieden, in diesem Fall existieren zwei Konfigurationen.

Dienste sind sowohl unabhängigen Einheiten im Engineeringmodell, als auch *Sichten* auf Komponenten sowie *Ausschnitte* aus deren Verhalten. Daher ist die Semantik eines Dienstes ein Teil der Semantik, nämlich ein weiteres Verhaltensprädikat, der entsprechenden Komponenten des Basismodells, auf die das Dienst abgebildet wird. Die Ein- und Ausgabeports des Dienstes werden direkt auf die Ein- und Ausgabeports der jeweiligen Komponente abgebildet.

Definition 6.15: Freie Dienste

Ein *freier Dienst* $d \in \mathbf{FS}_E \subseteq \mathbf{S}_E$ ist ein angebotener Dienst, der jedoch in keine Komponente eingebunden ist.

6.2.2 Dienstarchitekturen

Dienstarchitekturen dienen dazu, die logische, invariante Struktur eines (Sub-)Systems darzulegen und nicht die konkrete technische Struktur, inklusive Aufbau und Verdrahtung. Die realisierungsunabhängige Funktionale Abhängigkeit zwischen den jeweiligen Diensten legt zwar das Verhalten und die zu erbringenden Funktionalitäten fest, lässt aber offen, welche Komponenten welche Rolle später übernehmen. Dies drücken jeweils die einzelnen Konfigurationen aus. Abbildung 6.5 zeigt ein Beispiel mit Komponenten, die ihre innere Struktur festlegen, sowie Dienste, die nur die nach aussen zugängliche Funktionalität offenlegen.

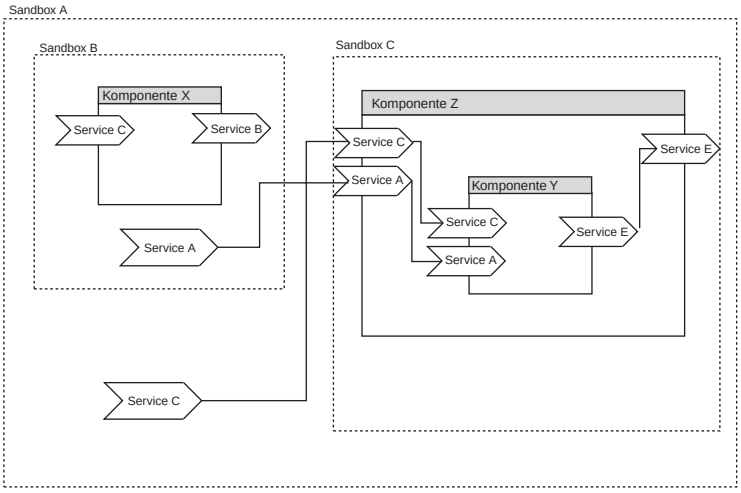


Abbildung 6.5: Eine Dienstarchitektur mit Komponenten, Diensten und Sandboxes

Definition 6.16: Dienstarchitektur

Eine *Dienstarchitektur* (eng. Service-Architecture) $da \in \mathbf{SA}_E$ ist die Beschreibung der logischen invarianten Architektur eines spontanen Systems. Sie unterscheidet sich von einer technischen Architektur (Konfiguration) darin, dass anstatt Komponenten freie Dienste die Funktionalität darstellen. Zusätzlich dazu können auch Komponenten in einer Dienstarchitektur auftreten, um fixe Komponenten zu modellieren, wie beispielsweise Legacy-Komponenten. Eine Dienstarchitektur-Spezifikation besteht also aus:

- Einer Menge an freien Diensten $FD \subseteq \mathbf{FS}_E$
- Einer Menge an Komponenten $Kom \subseteq \mathbf{C}_E$
- Einer Menge an Sandbox-Anforderungen $Snd \subseteq \mathbf{SBX}_E$
- Einer Menge an Bindungen $Bnd \subseteq \mathbf{B}_E$
- Einer Menge an Hostings $Hst \subseteq \mathbf{H}_E$

In der Dienstarchitektur ist es möglich, freie Dienste als Stellvertreter für Komponenten zu verwenden. Dadurch ist es möglich, lediglich die funktionale Abhängigkeit festzusetzen, ohne die Struktur einzuschränken. In der Dienstarchitektur treten konkrete Entitäten, Komponenten und Sandboxes, gemischt mit abstrakten, freien Dienste, auf.

Eine Dienstarchitektur besteht aus den folgenden Bestandteilen:

Service – Architecture :*ID*
Units : `<Units>`
Sandboxes : `<Sandbox>`
Bindings : `<Binding>`
Hostings : `<Hosting>`

Das Verhalten eines Dienstarchitektur ergibt sich aus den einzelnen Diensten.

Die Dienstarchitektur wird im Laufe der Entwicklung mit einer Einsatzumgebung, also einer Menge an Komponenten und Sandboxes, mittels der Deployment-Abbildung auf eine Menge von Konfigurationen (technische Architekturen) abgebildet, die diese invariante logische Architektur erfüllen.

6.3 Entwurfsablauf

Obwohl die Einordnung der hier vorgestellten Techniken in eine Methodik den Rahmen dieser Arbeit bei weitem sprengen würde, soll im folgenden die grobe Anordnung der verschiedenen Spezifikationen kurz angesprochen werden. Abbildung 6.6 zeigt einen möglichen Ablauf der Entwicklung unter Einbeziehung der vorgestellten Techniken. Dieser Ablauf ergab sich aus der Anwendung und Erprobung der Techniken in mehreren Projekten, stellt also einen gewissen Erfahrungswert dar.

Die einzelnen groben Phasen, als da wären Spezifikation, Deployment-Abbildung und Realisierung sollen nun im Weiteren kurz beschrieben werden. Der hier gehandhabte Ablauf wird auch anhand des Beispieles in Kapitel 11 beibehalten.

6.3.1 Spezifikation

Die Spezifikation der Architektur findet statt wie folgt:

Spezifikation der Dienste: hier stehen die Funktionalität der Dienste und die daraus resultierenden funktionalen Abhängigkeiten im Zentrum. Es wird also spezifiziert, welche Funktionalitäten (Dienste) für welchen Dienst benötigt werden und wie diese zusammenspielen. Es wird *nicht* spezifiziert *wer* die jeweiligen Dienste erbringt und wie diese aufgebaut sind.
Durch die Dienstspezifikation werden implizit folgende Bestandteile der Dienste und damit des Systems spezifiziert:

- *Interfaces*: die Schnittstellen und damit die Nachrichtentypen die in Verbindung mit dem Dienst verarbeitet werden.

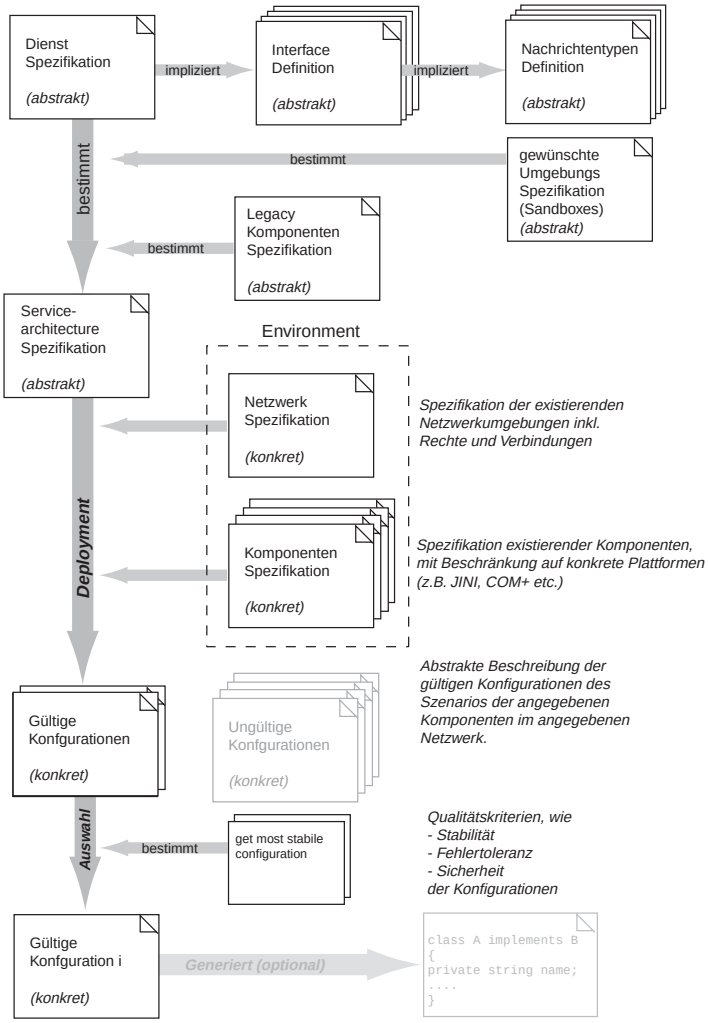


Abbildung 6.6: Ablauf der Entwicklung spontaner Komponentensysteme unter Einbeziehung der hier vorgestellten Techniken

- *Verhalten*: das Blackbox-Verhalten des Dienstes in einer angebrachten Spezifikationsform (in der in Kapitel 10 vorgestellten Realisierung findet dies in Form von Zustands-Übergangs-Diagrammen statt).

Spezifikation von Legacy-Komponenten (optional): falls im Szenario Legacy Komponenten, wie beispielsweise eine Datenbank, existieren, können diese als Komponentenspezifikation in die logische Architektur miteinbezogen werden.

Spezifikation von Umgebungs-Anforderungen (optional): sollen existierende Anforderungen einer Umgebung (Sandbox) mit in die logische Architektur einfließen, so kann diese als abstrakt definierte Sandbox definiert werden.

Dienstarchitektur-Spezifikation: aus den spezifizierten Elementen wird nun die logische, invariante Dienstarchitektur definiert, die hauptsächlich die funktionalen Abhängigkeiten und, bis auf gewollte Legacy-Komponenten, keinerlei Struktu-

rinformation vorgibt.

Die so definierte Dienstarchitektur kann nun kombiniert mit einer Einsatzumgebung durch die Deployment Abbildung, wie folgt, auf eine Menge von technischen Konfigurationen (Konfigurationen) abgebildet werden.

6.3.2 Deployment Abbildung

Durch die Deployment-Abbildung werden zwei Schritte abgearbeitet:

Freie Dienste der Dienstarchitektur werden auf Komponenten der Einsatzumgebung abgebildet, die diesen Dienst anbieten.

Umgebungsanforderungen werden auf Sandboxes der Einsatzumgebung abgebildet, die weniger restringent im Sinne der Definition 6.11 sind.

Wie diese einzelnen Schritte realisiert werden, das heißt wie beispielsweise die Äquivalenz eines freien Dienstes und eines angebotenen Dienstes einer Komponente berechnet werden, hängt von der jeweiligen Realisierung der Deployment-Abbildung ab. In der in Kapitel 10 vorgestellten Realisierung wird diese Äquivalenz beispielsweise einerseits aufgrund des eindeutigen Dienstbezeichners und andererseits aufgrund eines Vergleichs der minimierten Automaten, die das Blackbox-Verhalten des Dienst bestimmen, beurteilt.

Die Abbildung der Umgebungsanforderungen, die durch abstrakte Sandboxes definiert sind, auf Sandboxes der Einsatzumgebung kann beispielsweise durch einfaches Überprüfen der blockierten Bindungstypen erfolgen.

Das Ergebnis ist eine *Menge* von möglichen Konfigurationen, also technischen Architekturen, bestehend aus Komponenten und Sandboxes.

6.3.3 Auswahl und Realisierung

Alle Konfigurationen aus der Menge der möglichen Konfigurationen realisieren die spezifizierte logische Architektur. Zwischen den einzelnen Konfigurationen kann mittels einer endlichen Anzahl von Konfigurationsschritten (siehe Abschnitt 8.1.2) gewechselt werden, ohne den Systemablauf grundlegend zu verändern (siehe hierzu Abschnitt 8.1.3).

Die möglichen Konfigurationen unterscheiden sich jedoch bezüglich architektureller Qualitätskriterien, wie beispielsweise Stabilität, Redundanz oder Resourcenverbrauch (siehe hierzu Abschnitt 8.2). Bezüglich solcher Qualitätskriterien kann nun eine Konfiguration ausgewählt werden und eventuell in Codeskelette, wie sie gemäß der Abbildung des Engineeringmodells definiert ist (siehe Abschnitt 9.3), umgewandelt werden.

Wie man unschwer an den Verweisen erkennen kann, führt dieser Schritt der Auswahl einer Konfiguration und deren Realisierung nun zu dem Begriff der Architektur generell, der im Kapitel 8 besprochen wird. Zunächst wird aber noch der Zusammenhang zwischen Basis- und Engineeringmodell erklärt, indem die Konstrukte des Engineeringmodells auf die Entitäten des Basismodells abgebildet werden.

6.4 Zusammenfassung

In diesem Abschnitt wurde das Engineeringmodell vorgestellt. Das Engineeringmodell stellt eine für den Software-Ingenieur intuitivere Abstraktion dar, die die charakteristischen Entitäten der Anwendungsdomäne explizit modelliert, jedoch auf dem Basismodell aufbaut. Trotzdem ist das Basismodell zur Nutzung des Engineeringmodells explizit nicht zwingend notwendig, sondern optional für Ziele wie Verhaltensbeschreibung, Semantik etc., abrufbar.

Das Engineeringmodell besteht aus zwei Klassen von Entitäten, abstrakte und konkrete Entitäten:

Abstrakte Entitäten: Freie Dienste, Dienstarchitekturen.

Konkrete Entitäten: Komponenten, Sandboxes, (Bindungen), (Hostings).

Abstrakte Entitäten sind Konstrukte, die lediglich in der Abstraktion, also dem Engineeringmodell, existieren, jedoch nicht isomorph auf die Implementierung abgebildet werden. Konkrete Entitäten werden isomorph auf die Implementierung abgebildet. Die hierbei präsentierten Spezifikationstechniken sind lose geordnet, das bedeutet, dass hierbei keine Methodik zur Entwicklung spontaner Komponentensysteme aufgezeigt wird (dies liegt nicht im Fokus dieser Arbeit), sondern vielmehr die Abhängigkeiten und Kausalordnungen der einzelnen Entwicklungsdokumente, wie logische Architekturen, Dienstspezifikationen, technische Konfigurationen und Realisierungen, aufgezeigt werden.

Um bei Bedarf die Zusammenhänge zwischen den einzelnen Spezifikationen der jeweiligen Entitäten präzise auszudrücken sowie den einzelnen Entitäten, abstrakt und konkret, eine semantische Basis zu geben, ist es sinnvoll eine Abbildung auf einen gebräuchlichen Formalismus zu definieren, in dem dann die jeweiligen Kausalitäten formal und präzise ausgedrückt werden können. Daher wird das Engineeringmodell auf das formale Basismodell abgebildet, in dem dann die Zusammenhänge zwischen verschiedenen Spezifikationstechniken, beispielsweise die Verhaltensspezifikationen zweier Dienste in verschiedenen Beschreibungstechniken, wie STDs und MSCs, festzulegen.

Diese Abbildung wird im folgenden Kapitel beschrieben. Hierbei werden auch abstrakte Entitäten, wie freie Dienste, auf das Basismodell abgebildet um zu verdeutlichen, was sich hinter dem Begriff eines freien Dienstes konkret verbirgt.

KAPITEL

SIEBEN

Abbildung des Engineeringmodells auf das Basismodell

Die Abbildung der einzelnen Entitäten, abstrakt und konkret, des Engineeringmodells auf das Basismodell wird in den folgenden Abschnitten erläutert. Hierbei wird jeweils zunächst das Prinzip erklärt und anschließend die wichtigsten Abbildungsschritte formal aufgeführt.

7.1 Motivation

Wie schon erwähnt besteht das hier vorgestellte Modell aus einem anwendungsnahen Engineeringmodell und einem formalen Basismodell. Die Abstraktion des Engineeringmodells ist nahe an der gebräuchlichen Architekturabstraktion von Komponenten und Konnektoren und dessen Spezifikationstechniken lehnen sich an bewährte Beispiele, wie CORBA-IDL oder OCL (siehe Abschnitt 2.2) an.

Die zweite Schicht der Modells besteht aus dem in Kapitel 5 vorgestellten formalen Basismodell, dessen Aufgaben es sind, eine eindeutige Basis aufgrund eines präzisen und bekannten Formalismus bereitzustellen, auf die die Konstrukte des Engineeringmodells abgebildet werden können. Diese Abbildung des Engineeringmodells auf das Basismodell geschieht aus verschiedenen Gründen:

- Um den Entitäten und Spezifikationen des Engineeringmodells eine **eindeutige Bedeutung** zu geben, werden diese durch präzise Konstrukte des Basismodells beschrieben und erhalten somit eine klare Definition. Insbesondere die abstrakten Entitäten, die im späteren Systemcode nicht mehr explizit auftauchen, werden so greifbarer.
- Um das **Verhalten** der Dienste präzise und bequem spezifizieren zu können, sollen Entitäten in Bezug zu einer formalen Basis gesetzt werden. Dadurch können alle Verhaltensbeschreibungstechniken, die auf dieser formalen Basis definiert wurden, wie beispielsweise STDs, MSCs, Prädikate etc., verwendet und in Bezug gesetzt werden.

- Durch die Abbildung des Modells auf eine eindeutige formale Basis können **später Erweiterungen**, die beispielsweise die Verhaltensäquivalenz von Dienste semantisch festlegen (siehe Abschnitt 12.3), in das Modell integriert werden.

Das Ziel ist es, die semantische Bedeutung einer Entität des Engineeringmodells durch eine ihr entsprechende Menge von Basismodell-Netzwerken zu definieren.

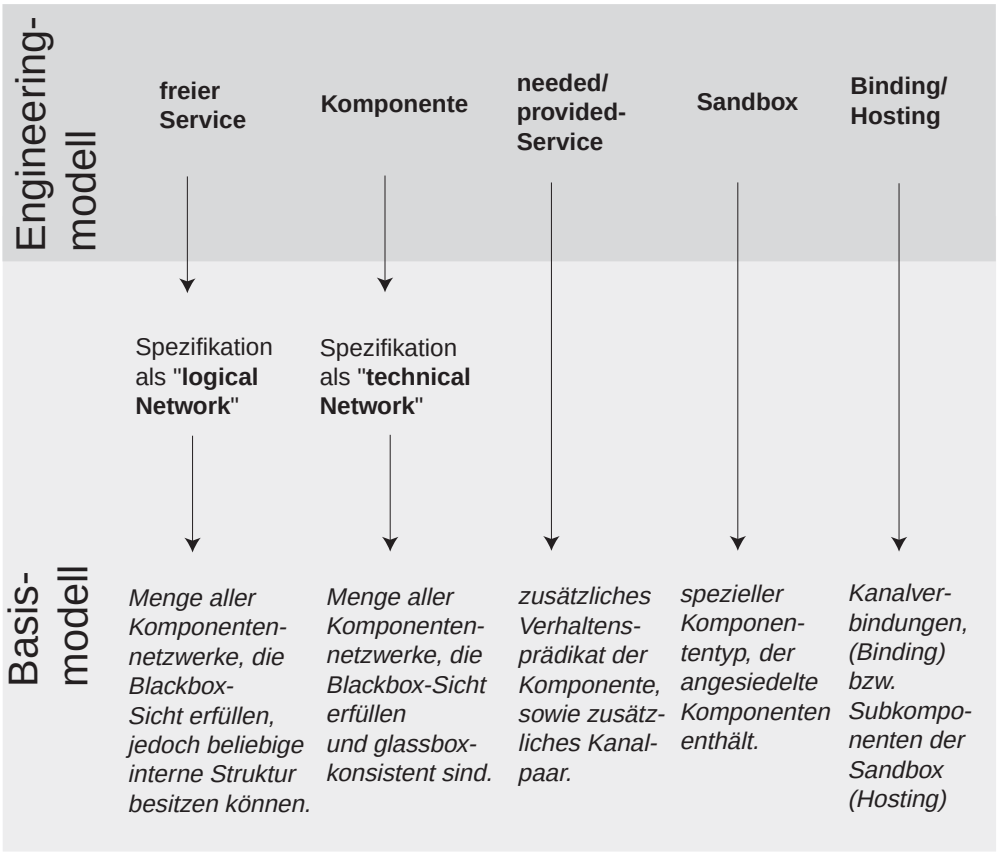


Abbildung 7.1: Entitäten des Engineeringmodells und deren Abbildungen im Basismodell

Hierfür werden die Spezifikationen der Engineeringmodell-Entitäten durch Umformung in Spezifikationsschemata des Basismodells umgewandelt. Diese Spezifikationsschemata definieren anschliessend *Mengen* von Netzwerken, die die Spezifikation erfüllen und somit die Semantik der entsprechenden Engineeringmodell-Entität festlegen (siehe Abbildung 7.1).

Um diese Schemata zu nutzen, müssen also nun die Spezifikationen des Engineeringmodells in die Form der Spezifikations-Schemata umgewandelt werden. Dies geschieht zum einen durch Expansionsregeln, zum anderen durch Abbildungsfunktionen.

7.1.1 Expansionsregeln der Engineeringmodell-Spezifikationen

Zunächst definieren wir jene Makroexpansionsregeln, die der Umwandlung der Tupelschreibweise dienen. Die folgende Expansion zerlegt einen gegebenen Dienst in seine

Verhaltensspezifikation, die Menge seiner benötigten Dienste und die Menge seiner Interfaces, deren Kardinalität definitionsgemäß um eins höher ist als die der benötigten Dienste.

SvcExp

ServiceS
Behavior F_S *Service* $NS_1, \ldots, NS_q$ *Interface* $IF_1, \ldots, IF_{q+1}$

Die Regel *IFExp* (Interface Expansion) zerlegt ein gegebenes Interface in seinen Eingabe- und Ausgabeport sowie seinen Calltype.

IFExp

InterfaceIF
Port I_{IF} *Port* O_{IF} *Calltype* CT_{IF}

Mittels Port-Gruppierung (*PortGrp*) gruppieren wir Tupel der Form Ein- und Ausgabeport sowie Calltype in drei geordnete Mengen von Eingabeports, Ausgabeports und Calltypes.

PortGrp

InputPort $I_1, \ldots, I_n$; *OutputPort* $O_1, \ldots, O_n$; *Calltype* $CT_1, \ldots, CT_n$; $(I_1, O_1, CT_1), \ldots, (I_n, O_n, CT_n)$
$\mathcal{I} =< I_1, \ldots, I_n >$; $\mathcal{O} =< O_1, \ldots, O_n >$; $\mathcal{CT} =< CT_1, \ldots, CT_n >$;

Da die benötigten Dienste weder strukturelle noch verhaltensspezifische Information besitzen, können diese bei der Abbildung auf Basismodell-Netzwerke fallengelassen werden (DropNeededServices – *DrpNS*):

DrpNS

Behavior F_S *Service* $NS_1, \ldots, NS_q$ *Interface* $IF_1, \ldots, IF_{q+1}$
Behavior F_S *Interface* $IF_1, \ldots, IF_{q+1}$

Analog zur ersten Regel können Komponenten mittels Komponentenexpansion (*KompExp*) expandiert werden:

KompExp
Component K
Service $PS_1, \ldots, PS_r$ *Service* $NS_1, \ldots, NS_q$ *Interface* $IF_1, \ldots, IF_{q+1}$ *Component* $S_1, \ldots, S_s$ *Binding* $SUB_1, \ldots, SUB_t$ *Binding* $B_1, \ldots, B_{r+q}$

Abkürzend führen wir die Substitution zweier Mengen, der benötigten und angebotenen Dienste, durch deren Interfaces ein (Interface Substitution – *IFSub*) ein, wobei die Kardinalität der Interfacemenge der Summe der Kardinalität der beiden Dienstmengen entspricht:

IFSub
Dienst $PS_1, \ldots, PS_r$ *Dienst* $NS_1, \ldots, NS_q$
Interface $IF_1, \ldots, IF_{q+r}$

Wir gliedern die Abbildung des Engineeringmodells auf das Basismodell in die folgenden Schritte:

1. Komponenten des Engineeringmodells auf Basismodell-Netzwerke abbilden.
2. Freie Dienste auf Basismodell Netzwerke abbilden.
3. Bindung und Hosting auf Kanäle abbilden.
4. Sandboxes auf Containerkomponenten abbilden.

7.2 Komponenten

Komponenten des Engineeringmodells sind Entitäten, die sowohl funktional als auch strukturell spezifiziert sind. Die funktionale Spezifikation legt das Verhalten in Form von angebotenen und benötigten Dienste fest. Die strukturelle Spezifikation legt fest, wie die Komponente intern aufgebaut ist, also welche Subkomponenten sie enthält und wie diese verbunden sind. Der Nutzen solcher hierarchisch strukturierten Spezifikationstechniken ist bekannt: sie erlauben nicht nur die Beschreibung der Komponenten auf verschiedenen Abstraktionsebenen im Sinne von schrittweiser Verfeinerung [Wir71], sondern darüber hinaus auch ein Kapselung mittels Geheimnisprinzip, um somit eine hochmodulare und entkoppelte Topologie des Systems zu erreichen [Par72].

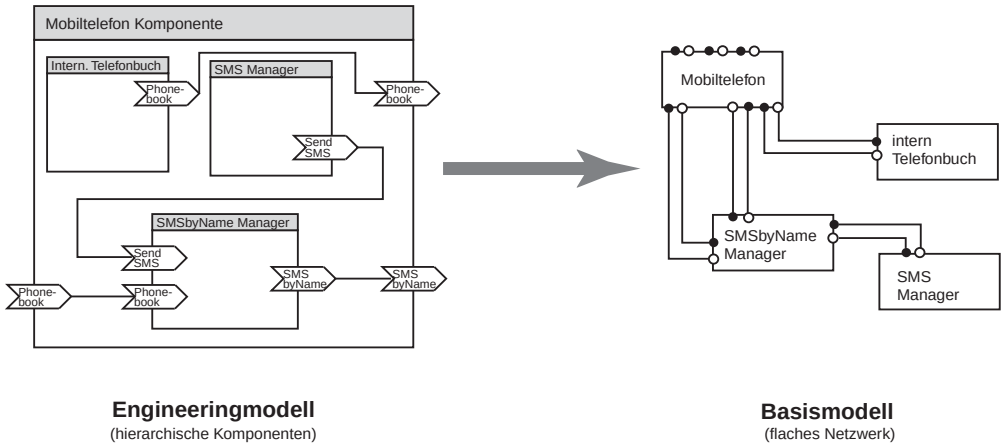


Abbildung 7.2: Engineeringmodell-Komponente und eines ihrer flaches Basismodell Pendants (Ein/Ausgabeports jeweils weiß bzw. schwarz)

Die Signaturen der Bindungsfunktionen eines Komponententyps (siehe oben) setzen somit die Kapselung mittels Geheimnisprinzip fest.

Eine Komponente im Engineeringmodell wurde durch das folgende Schema definiert:

Componenttype : *TypeID*
NeededServices `<Service>`
ProvidedServices : `<Service>`
SDependency : $S_P \rightarrow \mathcal{P}(S_N)$
Subunits : `<unit>`
Subunitsbindings : `<Binding>`

Um die semantische Bedeutung der Spezifikation einer Engineeringmodell-Komponente zu definieren, wird diese nun auf eine Menge flacher Basismodell-Netzwerke abgebildet, die dieser Spezifikation entsprechen (siehe Abschnitt 5.2.2). Dabei wird eine solche Komponente auf ein Netzwerk abgebildet, das ein äquivalentes Blackbox-Verhalten, aber auch eine gleiche interne Struktur besitzt. Diese Glassbox-Sicht wurde im Engineeringmodell durch die Subkomponenten und deren Bindungen definiert. Im Basismodell stehen nur flache Komponentennetzwerke zur Verfügung. Daher wird eine Subkomponente dadurch ausgedrückt, dass Ports von Subkomponenten hierbei entweder an Ports anderer Subkomponenten der Mutterkomponente gebunden sind oder an Ports der Mutterkomponente selbst. In Abbildung 7.2 ist eine Komponente des Engineeringmodells und eines ihrer entsprechenden flachen Basismodellnetzwerke abgebildet.

Bezüglich der Komponententypen kann unterschieden werden zwischen Komponententypen, die über eine leere Menge an Subunits verfügen, sog. *atomaren Komponenten*, und Komponententypen mit einer nicht leeren Menge an Subunits, sog. *komponierten Komponenten*. Man beachte, dass komponierte Komponenten nur ein Spezifikationskonstrukt sind: atomare Komponenten werden auf dementsprechende Komponenten im Basismodell abgebildet, komponierte Komponenten auf ein dementspre-

chendes Komponentennetzwerk im Basismodell. Das Basismodell Komponentennetzwerk besteht aus einem flachen Netzwerk an Basismodellkomponenten, das gemäß der Spezifikation der komponierten Komponenten, also der Subunits, deren Bindung und Hosting, strukturiert ist. Darüberhinaus erfüllt das Komponentennetzwerk die Verhaltensspezifikation des komponierten Komponententyps bezüglich der Ein- und Ausgabeports.

Mit der semantischen Abbildung einer Komponente des Engineeringmodells auf das Basismodell werden mit der Engineeringmodell-Komponente eine Menge von zugehörigen Basismodellnetzwerken assoziiert, die die in der Komponentenspezifikation verlangten Anforderungen bezüglich Verhalten und Struktur erfüllen. Diese Anforderungen sind also insgesamt:

Blackbox-Sicht verlangt, dass die Schnittstellen der Komponente und des Basismodellnetzwerks strukturell und verhaltenstechnisch kompatibel sind.

Glassbox-Sicht verlangt, dass das Netzwerk bezüglich der Struktur der Komponente, d.h. der Bindungen und Hostings der Subunits strukturiert ist. Hierbei müssen zwei Fälle unterschieden werden:

1. Falls die Menge an Subunits leer ist, es sich demnach um eine atomare Komponente handelt, so muss das Netzwerk prinzipiell aus exakt einer Basismodellkomponente bestehen. Die Blackbox-Sicht des Netzwerkes und der Komponente sowie deren Typ impliziert dann die entsprechenden Eigenschaften des Netzwerkes.
2. Falls die Menge von Subunits nicht leer ist, sondern aus einer endlichen Menge von n Subunits besteht, so muss es möglich sein, das entsprechende Netzwerk in eine Menge von n Teilnetzwerken zu zerlegen.

Verhaltens-Sicht verlangt das identische Blackbox-Verhalten der Komponenten und des Netzwerkes.

Es fällt auf, dass diese drei Eigenschaften durch die in Abschnitt 5.2 definierten Spezifikations-Schemata für Basismodell-Netzwerke (technische und logische Netzwerk-Spezifikationen) abgedeckt werden. Eine Spezifikation eines technischen Netzwerkes erfüllt gerade sowohl die verlangte Blackbox- wie Glassbox-Sicht. Um daher die Menge der Basismodell-Netzwerke, die die Semantik einer Engineeringmodell-Komponente festlegen, bequem beschreiben zu können, wird die Spezifikation einer Engineeringmodell-Komponente in die Spezifikation eines technischen Netzwerkes überführt (siehe Abbildung 7.3). Wir bedienen uns dabei zunächst den in Abschnitt 7.1.1 definierten syntaktischen Expansionsregeln.

Eine Komponente wird im weiteren auch durch die folgende Tupelschreibweise dargestellt:

$$component = (id, NS, PS, SU, SUB)$$

Im Weiteren wird für eine Bindung $b \in \mathbf{CB}_E$ die Notation

$$\begin{aligned} drain.b &= u \in \mathbf{U}_E \text{die Senke der Bindung} \\ source.b &= u \in \mathbf{U}_E \text{die Quelle der Bindung} \end{aligned}$$

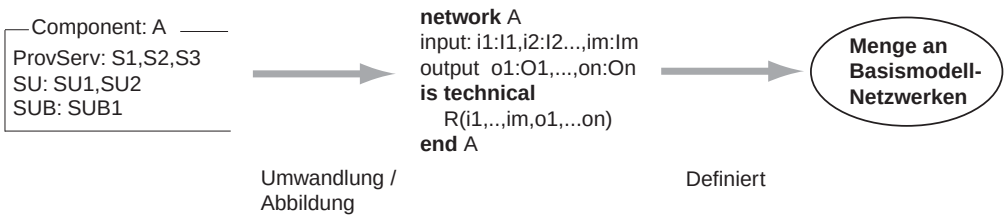


Abbildung 7.3: Prinzipieller Ablauf der Abbildung: Umwandlung einer Engineeringmodell-Spezifikation in eine Basismodell-Netzwerk-Spezifikation, die dann die jeweilige Menge an Netzwerken bestimmt.

benutzt, um die Quelle bzw. Senke einer Bindung zu adressieren. Desweiteren wird analog zu der Notation im Basismodell die Schreibweise $\overline{NS}$ für $(NS_1, \ldots, NS_n)$, also die Vektorisierung, verwendet.

Betrachten wir das Schema einer Komponente K in der entsprechenden Tupelnotation (die Abhängigkeitsfunktion ist implizit in den Diensten gegeben und kann daher weggelassen werden):

$$K = (ID, NS, PS, SU, SUB)$$

$$K = (ID, (NS_1, \ldots, NS_m), (PS_1, \ldots, PS_n), (SU_1, \ldots, SU_o), (SUB_1, \ldots, SUB_p))$$

K wird nun gemäß der Expansionsregeln aus Abschnitt 7.1.1 umgeformt:

$$\begin{aligned}
K \quad &= \quad ((NS_1, \ldots, NS_m), (PS_1, \ldots, PS_n), (SU_1, \ldots, SU_o), (SUB_1, \ldots, SUB_{o+p})) \\
&\xrightarrow{IFSub} \quad ((IF_1, \ldots, IF_{m+n}), (SU_1, \ldots, SU_o), (SUB_1, \ldots, SUB_{o+p})) \\
&\xrightarrow{IFExp} \quad ((I_1, O_1, CT_1), \ldots, (I_{m+n}, O_{m+n}, CT_{m+n}), (SU_1, \ldots, SU_o), \\
&\qquad (SUB_1, \ldots, SUB_{o+p})) \\
&\xrightarrow{PortGrp} \quad (\mathcal{I}, \mathcal{O}, \mathcal{CT}, (SU_1, \ldots, SU_o), (SUB_1, \ldots, SUB_{o+p})) \\
&\qquad mit\ \|\mathcal{I}\| = \|\mathcal{O}\| = \|\mathcal{CT}\| = m + n
\end{aligned}$$

Da es sich bei den Diensten der Komponente um abstrakte Bündelungen von Struktur (Interface) und Verhalten der Komponente handelt, kann die Abbildung der Komponente von der folgenden Definition ausgehen. Sei nun

$$\begin{aligned}
In: SUB \times SU \quad &\longrightarrow \quad < SUB > \\
In(SUB, SU_i) \quad &= \quad < SUB_j, \ldots, SUB_{j+k} > \\
&\qquad , wobei \\
&\qquad \forall j \leq n \leq j + k : drain.SUB_n = SU_i
\end{aligned}$$

Die Funktion *In* filtert also aus einer Menge von Subunit-Bindungen alle jene Bindungen heraus, die in eine gegebene Subunit münden, und gibt diese als Liste aus. Ebenso

definieren wir *Out*:

$$\begin{aligned} Out:\ SUB \times SU &\longrightarrow < SUB > \\ Out(SUB, SU_i) &= < SUB_j, \ldots, SUB_{j+k} > \\ &\quad , wobei \\ &\quad \forall j \leq n \leq j+k : source.SUB_n = SU_i \end{aligned}$$

Sei nun $\mathbf{CB}_E$ die Menge aller Bindungen im Engineeringmodell und **CH** wie gehabt die Menge der Kanalbezeichner. Dann sei

$$F_{Bnd} : \mathbf{CB}_E \longrightarrow \mathbf{CH}$$

die *injektive* Abbildung der Bindungen auf den entsprechenden Kanalbezeichner im Basismodell. Man beachte, dass sowohl die Bindungen in der Spezifikation im Engineeringmodell als Liste als auch die entsprechenden Abbildungen als Kanalbezeichner eine Ordnung besitzen.

$$F_{Bnd}(SUB_1, \ldots, SUB_{o+p}) = \{h_1, \ldots, h_{2(o+p)}\} = H' \text{ mit } h_i \in \mathbf{CH}$$

Ein Bindung wird also jeweils auf zwei Kanalbezeichner, einen Eingabe- bzw. einen Ausgabe-Kanal abgebildet. Wir ordnen nun den "Platzhaltern" der Ein- und Ausgangskanalbezeichnern ijk und ojk die Ein- und Ausgangskanäle zu. Es gilt:

Wenn $Out(SUB, SU_1) = SUB_1, \ldots, SUB_k$
dann sei
$o_i = F_{Bnd}(SUB_i)$

Die Eingabekanäle werden analog mittels der Funktion *In* abgebildet.

Sei nun eine isomorphe Abbildung T definiert, die Ports P aus dem Engineeringmodell einen entsprechenden Kanaltyp C des Basismodells zuordnet, so dass die Nachrichtentypen des Kanaltyps den Messagetypen des Ports entsprechen. Sei $k \in \mathbf{CH}$ ein Kanal und $t_k \in \mathbf{T_{CH}}$ der entsprechende Kanaltyp, so ist $\|t_k\|$ die Menge der Nachrichtentypen, die über den Kanaltyp t_k fließen können. Sei des Weiteren **P** die Menge der Ports und $p \in \mathbf{P}$ ein abzubildender Port mit der Menge der Messagetypen MT_p:

$$\begin{aligned} &T:\ \mathbf{P} \longrightarrow \mathbf{T_{CH}} \\ &\|T(p)\| \equiv MT_p; \end{aligned}$$

Die Funktion T bildet also einen Port-Typ des Engineeringmodells auf einen Kanaltyp des Basismodells dermaßen ab, dass beide Typen die äquivalenten Nachrichten- bzw. Messagetypen besitzen. Für eine Bindung $B \in SUB$ schreiben wir $I.B$ bzw. $O.B$ um den Ein- bzw. Ausgangsport der Quelle der Bindung zu bezeichnen.

Sei die abzubildende Komponente A definiert und besitze die Subkomponenten $SU = (SU_1, \ldots, SU_o)$ sowie die Subkomponenten-Bindungen SUB. Nun kann die der Komponente bezüglich Glassbox- und Blackbox-Sicht entsprechende Menge an Basismodellnetzwerken durch die folgende Spezifikation eines *technischen* Netzwerks ausgedrückt werden:

network A
input i1:$T(I_1)$, i2:$T(I_2)$, ..., im+n:$T(I_{m+n})$
output o1:$T(O_1)$,o2:$T(O_2)$,..., on:$T(O_{m+n})$
internal h1:T($I.SUB_1$),h2:T($O.SUB1$),..., h2(o+p):T($O.SUB_{2(o+p)}$)
is
technical network
$F_{Bnd}(Out(SUB, SU_1)$= $F.SU_1$ ($F_{Bnd}(In(SUB, SU_1))$);
...
$F_{Bnd}(Out(SUB, SU_o)$= $F.SU_o$ ($F_{Bnd}(In(SUB, SU_o))$);
end A

Da das Verhalten in den jeweiligen Diensten definiert wurde, ist das Blackbox-Verhalten der Komponenten lediglich eine Bündelung der angebotenen Dienste. Diese bestimmen indirekt durch deren benötigte Dienste auch das Verhalten auf deren Ein-/Ausgabeports. Das Blackbox-Verhalten einer Komponente entspricht daher der Parallelkomposition der Verhalten ihrer angebotenen Dienste (Provided Services)[1]:

$$F' \equiv F_{PS_1} \otimes \ldots \otimes F_{PS_n}$$

Ebenso entspricht das Verhalten einer Komponente, gemäß der Definition des Schemas eines technischen Netzwerks (siehe Abschnitt 5.2.2), der Komposition ihrer Subkomponenten.

Dies bedeutet, dass letztendlich die “Blätter”, also die Basis-Netzwerke, komponiert werden, was wiederum der Parallelkomposition entspricht. Dies bedeutet insbesondere, dass Komponenten nur aus atomaren Komponenten und Wrapper-Komponenten bestehen. Ein eventuelles Verhalten der Wrapper-Komponente kann als eigene Subkomponente realisiert werden.

Eine Komponentenspezifikation wird auf die Menge aller Basismodell-Netzwerke abgebildet, die sowohl vom externen Verhalten her, als auch bezüglich des internen Aufbaus der Spezifikation entsprechen. Hierbei werden die Dienste, die die Komponente anbietet oder benötigt, auf jeweilige Verhaltensprädikate der Komponente abgebildet.

Eine Komponente wird durch die Spezifikation eines ihr entsprechenden *technischen Netzwerkes* auf die Menge der Basismodell-Netzwerke abgebildet.
Man beachte, dass eine Komponententyp-Spezifikation des Engineeringmodells als eine Menge von Basismodellnetzwerken interpretiert wird, die wiederum aus Basismodellkomponenten bestehen.

Die Bedeutung der Spezifikation ist demnach eine Menge von Basismodellnetzwerken, und nicht ein einzelnes Element, da Spezifikationen im Sinne einer losen Semantik [Wir90] interpretiert werden. Dies bedeutet insbesondere, dass verschiedene Netzwerke, die demselben Engineeringmodell Komponententyp zugeordnet sind, sich in Eigenschaften unterscheiden, die in der Spezifikation nicht explizit definiert werden. Diese Unterspezifikation gibt einen wohl definierten Spielraum, um verschiedene

[1] Man beachte, dass wir durch die Parallelkomposition der Dienste eine interne Beeinflussung der Dienste, also Eigenschaften wie Feature-Interaction, ausschließen.

“Konfigurationen” zu einer Spezifikation zu erlauben, was für unsere Zwecke, nämlich der Modellierung spontaner Komponentensysteme, notwendig ist.

Zur Illustration soll nun ein einfaches Beispiel einer Engineeringmodell-Komponente, ein Drucker mit Steuerkomponente, auf die Menge aller Basismodellnetzwerke abgebildet werden.

7.2.1 Beispiel: Abbildung einer Engineeringmodell Komponente

Wir greifen ein Beispiel aus Abschnitt 4.3 auf, jedoch werden Komponenten und Dienste aus Gründen der Übersichtlichkeit vereinfacht. Das Verhalten der einzelnen Dienste betrachten wir als durch ein STD und seine Prädikatenform gegeben. Ein für dieses Beispiel vereinfachter Drucker bietet einen vereinfachten Print-Dienst an. Die Komponente ist in Abbildung 7.4 graphisch skizziert. Der Drucker sei folgendermassen definiert:

Componenttype :*Printer*
ProvidedServices : `<PrintJob>`
NeededServices `<PrintJob>`
SDependency : `<>`
Subcomponents : `<Spooler,Device>`
SubcomponentBindings : `<Printer_Spooler,Spooler_Device>`

Er bietet die Möglichkeit, ein Dokument zu drucken, hierbei Einstellungen wie Auflösung etc (so genanntes “Pagesetup”) vorzunehmen, die Anzahl der Kopien etc. Der Dienst wird folgendermaßen spezifiziert:

DienstService :*PrintJob*
Behavior : `Specification`
NeededServices `<Print>`
Interface : `PrintJob_Interface,Print_Interface`

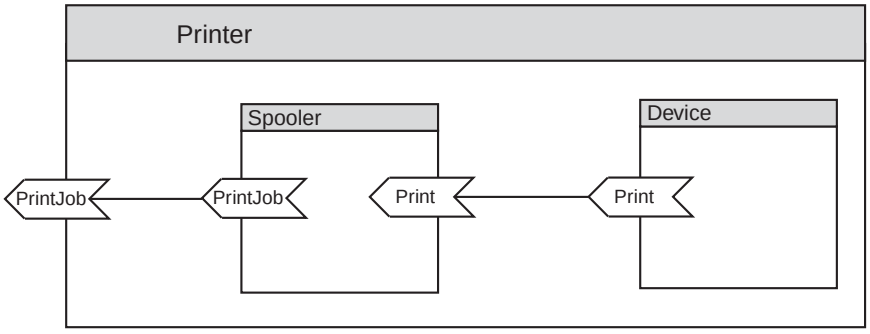


Abbildung 7.4: Beispielkomponente “Printer”

Die dazugehörigen Interfaces, welche die Signatur und die Kommunikationstypen (Messagetypes und dazugehörige Parametertypes) festlegen, lauten wie folgt:

Interface :*PrintJob_Interface*
InputPort : `PrintJobInPort`
OutputPort `PrintJobOutPort`
Calltype : `RMI`

Interface :*Print_Interface*
InputPort : `PrintInPort`
OutputPort `PrintOutPort`
Calltype : `RMI`

Ein Interface setzt sich aus zwei Ports (Ein- bzw. Ausgangsport) zusammen, die wiederum die Menge der Nachrichtentypen und deren Parametertypen bestimmen:

Port :*PrintJobInPort*
MesageTypes :
```
 <setDocument(String),
  start(),cancel(),
  pause(),resume()>
```

Port :*PrintJobOutPort*
MessageTypes : `<start(boolean)>`

Das eigentliche Druckgerät ist die Subkomponente *Device*, die einen Dienst *Print* anbietet, der lediglich von einem gegebenen Dokument einen durch zwei Seitenzahlen vorgegebene Ausschnitt drucken kann. Er kann also insbesondere nicht mehrere Kopien drucken, pausieren etc. Dies wird von der Mutterkomponente *Printer* in Form des Dienstes *PrintJob* realisiert. Der Drucker besteht aus zwei Subkomponenten, einem Spooler und einem Device:

Componenttype :*Spooler*
ProvidedServices : `<PrintJob>`
NeededServices `<Print>`
SDependency : <>
Subcomponents : <>
SubcomponentBindings : <>

Componenttype :*Device*
ProvidedServices : `<Print>`
NeededServices <>
SDependency : <>
Subcomponents : <>
SubcomponentBindings : <>

Der *Print*-Dienst des Druck-Gerätes ist das einfache Übersenden des Dokumentes:

Service :*Print*
Behavior : `Specification`
NeededServices `<>`
Interface : `Print_Interface`

Die Bindung zwischen Mutter- und Sub-Komponente ist durch das folgende Schema definiert:

Binding :*Printer_Spooler*
Source : `Printer`
Drain : `Spooler`
CallType : `local`
Service : `PrintJob`

Binding :*Spooler_Device*
Source : `Spooler`
Drain : `Device`
CallType : `local`
Service : `Print`

Das dazugehörige Interface setzt sich aus den folgenden beiden Ports zusammen:

Port :*PrintInPort*
MesageTypes :
```
<PrintDocumentPages(String,int,int),
 cancel()
```

Port :*PrintOutPort*
MessageTypes : `<PrintDocumentPages(int,int)>`

Dem Leser mag diese Spezifikationstechnik eventuell umständlich und ungewohnt erscheinen, da für eine simple Komponente Information über so viele verschiedene Einheiten verstreut sind. Hierzu muss gesagt werden, dass all diese Einheiten zum einen bei einer werkzeugunterstützten Spezifikation automatisiert generiert werden können, beispielsweise aus einer graphischen Spezifikation. Zum anderen können diese Einheiten wiederverwendet werden, das bedeutet insbesondere bei Diensten, die meist mehrfach auftreten, dass ein Dienst und seine Signaturen nur einmal spezifiziert werden und anschliessend referenziert werden können. Die Komponente besitzt die Tupelschreibweise die folgende Form:

$$K = (\text{“}Printer'', (Print), (PrintJob), (Spooler, Device), (Printer\_Spooler, Spooler\_Device))$$

K wird nun gemäß der Expansionsregeln aus Abschnitt 7.1.1 umgeformt:

$$
\begin{aligned}
K \quad = \quad & (\text{“}\mathit{Printer}'', (\mathit{PrintJob}), (\mathit{PrintJob}), (\mathit{Spooler}, \mathit{Device}), \\
& (\mathit{Printer\_Spooler}, \mathit{Spooler\_Device})) \\
\xrightarrow{IFSub} \quad & ((\mathit{PrintJob\_Interface}, \mathit{Print\_Interface}), (\mathit{Device}), \\
& (\mathit{Printer\_Spooler}, \mathit{Spooler\_Device})) \\
\xrightarrow{IFExp} \quad & ((\mathit{PrintInPort}, \mathit{PrintOutPort}, \text{“}\mathit{RMI}''), \\
& (\mathit{PrintJobInPort}, \mathit{PrintJobOutPort}, \text{“}\mathit{RMI}''), (\mathit{Device}), \\
& (\mathit{Printer\_Spooler}, \mathit{Spooler\_Device})) \\
\xrightarrow{PortGrp} \quad & (\{\mathit{PrintInPort}, \mathit{PrintJobInPort}\}, \{\mathit{PrintOutPort}, \mathit{PrintJobOutPort}\}, \\
& \{\text{“}\mathit{RMI}'', \text{“}\mathit{RMI}''\}), (\mathit{Device}), (\mathit{Printer\_Spooler}, \mathit{Spooler\_Device}))
\end{aligned}
$$

Wir definieren nun die injektive Funktion F_{Bnd}, die eine Bindung auf den entsprechenden Kanalbezeichner des Basismodells abbildet:

$$F_{Bnd}(\mathit{SUB}) = \{(\mathit{ID.SUB} \mid \mathit{In}), (\mathit{ID.SUB} \mid \mathit{Out})\}$$

Wir bilden also eine Bindung auf ein Kanalbezeichnerpaar ab, die durch den eindeutigen Bezeichner der Bindung und jeweils dem Zusatz *In* und *Out* konkateniert werden. Die Subunit-Bindung *Printer_Device* wird also folgendermassen abgebildet:

$$F_{Bnd}(\mathit{Printer\_Device}) = \{\text{“}\mathit{Printer\_DeviceIn}'', \text{“}\mathit{Printer\_DeviceOut}''\}$$

Desweiteren müssen die Ports mittels der Abbildung T auf die entsprechenden Kanaltypen abgebildet werden:

$$
\begin{aligned}
T(\mathit{PrintJobInPort}) \quad &= \quad ?\mathit{PrintJob} \\
& \quad \text{, wobei } \|?\mathit{PrintJob}\| = \\
& \quad \{\mathit{setDocument}, \mathit{start}, \mathit{stop}, \mathit{pause}, \mathit{resume}\} \\
T(\mathit{PrintJobOutPort}) \quad &= \quad !\mathit{PrintJob} \\
& \quad \text{, wobei } \|!\mathit{PrintJob}\| = \{\mathit{start}\} \\
T(\mathit{PrintInPort}) \quad &= \quad ?\mathit{Print} \\
& \quad \text{, wobei } \|?\mathit{Print}\| = \\
& \quad \{\mathit{PrintDocumentPages}(\mathit{int}, \mathit{int.string})\} \\
T(\mathit{PrintOutPort}) \quad &= \quad !\mathit{Print} \\
& \quad \text{, wobei } \|!\mathit{Print}\| = \{\mathit{PrintDocumentPages}(\mathit{int}, \mathit{int})\}
\end{aligned}
$$

Mit den so definierten Kanälen, Kanalbezeichnern und Strukturen kann nun die Komponente auf eine Menge von Basismodellnetzwerken abgebildet werden, die sowohl strukturell als auch verhaltenstechnisch der Komponente entsprechen. Dazu definieren wir nun das entsprechende Schema:

network Printer
input i1:?*PrintJob*
output o1:!*PrintJob*
internal h1:!*Print*,h2:?*Print*
is
technical network
PrintJobOut= $F.Spooler(h1, PrintJobIn)$;
$h1 = F.Device(h2)$;
end Printer

7.3 Freie Dienste

Freie Dienste sind abstrakte Entitäten des Engineeringmodells und werden daher mittels Deploymentabbildung durch Komponenten ersetzt. Hierbei ist zu beachten, dass dies keine injektive Abbildung ist und daher eine Komponente mehrere Dienst-Rollen übernehmen kann, wenn diese die Dienste als " Provided-Services" anbietet (siehe Abschnitt 6.1.1). Für diese Abbildung müssen die Komponenten und Dienste sowohl eine *Blackbox-*, als auch *Verhaltens-Sicht* besitzen:

Blackbox-Sicht: Die Provided- und Needed-Services (angebotenen und benötigten Dienste) einer Komponente sind Sichten auf die Komponente, die einen Ausschnitt aus deren Verhalten bezüglich der Signaturen beinhalten. Folglich muss für die Abbildung eines Dienstes auf die es später realisierende Komponente deren Blackbox-Sicht dem Dienst entsprechen. Dies bedeutet informell, dass die Ein- und Ausgabeports jedes Dienstes einen typkonsistenten Ausschnitt der entsprechenden Portmenge des Komponententyps, modulo Umbenennung, darstellen müssen. Anders ausgedrückt, muss eine totale und injektive Abbildungsfunktion zwischen den Dienst- und Komponentenporttypen existieren. Durch die Injektivität schließen wir aus, dass einem Port mehrere Dienste zugeordnet werden (Mehrfachbelegung – siehe Abbildung 7.5).

Verhaltens-Sicht: Neben der Blackbox-Sicht muss auch die Verhaltens-Sicht zwischen dem Dienst und der ihn realisierenden Komponente gewährleistet sein. Im allgemeinen beschreibt ein Dienst das Verhalten abstrakter als eine Komponente, da er nur einen Teil des Verhaltens bezogen auf eine Untermenge der Ports definiert. Im Unterschied zu einem angebotenen Dienst stellt ein benötigter Dienst nicht ein Teilverhalten der Komponente dar. Wir betrachten einen benötigten Dienst daher als Verhaltens-Anforderungen an die Umwelt. Relativ zu diesem Verhalten der Umwelt wird das eigene Verhalten spezifiziert. Daher wird in diesem Falle keine Verhaltens-Sicht definiert.

Blackbox- und Verhaltens-Sicht werden in den in Abschnitt 5 vorgestellten Spezifikationstechniken durch ein *logical Network* erreicht. Eine solche Spezifikation beinhaltet alle Basismodellnetzwerke, die jeweils die Verhaltens-Sicht erfüllen (also das gleiche Blackbox-Verhalten besitzen) und die identische Blackbox-Struktur besitzen, also

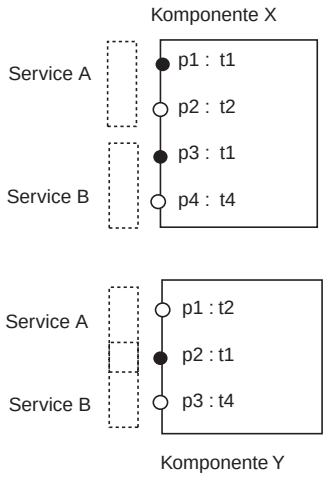


Abbildung 7.5: Mehrfachbelegung eines Ports bei Komponente Y (*p* ist lokaler Portbezeichner, *t* der jeweilige Typ)

identische Porttypen etc. Es wird ausdrücklich nicht eine bestimmte innere Struktur vorgegeben.

Wir betrachten einen Dienst *S*. *S* wird folgendermaßen abgebildet:

$$
\begin{aligned}
S \quad &\overset{SrvExp}{\longrightarrow} \quad (F, (N_1, \ldots, NS_s), (IF_1, \ldots, IF_{s+1})) \\
&\overset{DrpNS}{\longrightarrow} \quad (F, (IF_1, \ldots, IF_{s+1})) \\
&\overset{IFExp}{\longrightarrow} \quad (F, ((I_1, O_1, CT_1), \ldots, (I_{s+1}, O_{s+1}, CT_{s+1}))) \\
&\overset{PrtGrp}{\longrightarrow} \quad (F, \mathcal{I}, \mathcal{O}, \mathcal{CT})
\end{aligned}
$$

Wir bilden damit den Dienst *S* auf die Menge aller Basismodell-Netzwerke ab, die durch die folgende Spezifikation des entsprechenden *logischen Netzwerks* definiert werden:

network S
input i1:T(I_1), i2:T(I_2),..., i(s+1):T(I_{s+1})
output o1:T(O_1), o2:T(O_2),..., o(s+1):T(O_{s+1})
is
logical
F(i1, ..., i(s+1),o1, ...,o(s+1));
end S

Ein freier Dienst wird durch die Spezifikation als *logisches Netzwerk* auf die Menge aller Basismodell Netzwerke abgebildet, die dem spezifizierten Verhalten und der Schnittstelle gehorchen, jedoch beliebige interne Struktur besitzen können.

7.4 Bindung

Die Bindung (engl. Binding) stellt im Engineeringmodell die Relationen zwischen den Units dar.

Binding :*ID*

Source : `Unit`
Drain : `Unit`
CallType : `CallType`
Service : `Service`

Eine Bindung im Engineeringmodell ist als folgender Tupel definiert:

$$BindingB = (ID_{Bnd}, Q, S, CT, SV)$$

Hierbei stellt die Quelle Q und die Senke S jeweils eine Unit dar. Das heißt also eine Komponente oder für den Fall, dass es sich um eine Bindung in einer Dienstarchitektur handelt, ein freier Dienst. Die beiden Units sind gebunden bezüglich eines Dienstes SV, der im Falle eines freien Dienste bei Quelle oder Senke mit diesem übereinstimmen muss. Dieser Dienst bestimmt auch indirekt die Schnittstellensignatur, bezüglich derer die beiden Units gebunden sind.

$$DienstSV = (ID_{Svc}F, IF, NS)$$

F stellt, wie bereits definiert, das Verhalten des Dienstes dar, NS die benötigten Dienste und IF die einzelnen Schnittstellen (engl. Interfaces) , die jeweils noch bezüglich ihrer Nachrichtentypen und deren Parametertypen definiert sind (siehe Abschnitt 6.2.1).

Sei $F : \mathbf{K} \rightarrow NW(\mathbf{CH})$ die Abbildungsfunktion der Engineeringmodellkomponenten auf die Menge der entsprechenden Basismodellnetzwerke (siehe Abschnitt 6.1.1).

Die Bindung B, mit $B = (ID_{Bnd}, Q, S, CT, SV)$ bezüglich des Dienstes $SV = (ID_{Svc}, F, IF, NS)$ wird auf ein Kanalpaar $c_i, c_o \in \mathbf{CH}$ abgebildet, das zwei Komponentennetzwerke $NW_Q = F(Q)$ und $NW_S = F(S)$ verbindet, so dass:

$$\begin{aligned}
Contains_Bnd(B, nw) \in \mathbf{B} \times NW(\mathbf{CH}) \quad &\rightarrow \quad Bool \\
Contains_Bnd(nw, B) \quad &\stackrel{Def}{=} \quad \exists\, c_i, c_o \in nw : \\
&\quad c_i \in in.F(Q) \wedge c_i \in out.F(S) \wedge \\
&\quad c_o \in out.F(Q) \wedge c_o \in in.F(S) \wedge \\
&\quad \|type(c_i)\| = Messagetypes(I) \cup \{CT\} \wedge \\
&\quad \|type(c_o)\| = Messagetypes(O) \cup \{CT\}
\end{aligned}$$

Eine Bindung zwischen zwei Komponenten A und B bezüglich eines Dienstes S unter dem Calltype C wird auf einen Kanalpaar, jeweils Ein- bzw. Ausgangskanal, zwischen den beiden Basismodellnetzwerken der Komponenten A und B abgebildet. Der Typ der jeweiligen Kanäle muss dermaßen beschaffen sein, dass auf ihnen die entsprechenden Nachrichtentypen fliessen. Darüberhinaus wird der Calltype C durch einen zusätzlichen Nachrichtentyp ausgedrückt, der jedoch lediglich zur Identifikation dient.

7.5 Sandboxes und Hosting

Eine Sandbox wird in das Basismodell als spezieller Komponententyp abgebildet, der die angesiedelten Komponenten aggregiert. Eine Sandbox wurde im Engineeringmodell durch das folgende Schema definiert:

Sandbox :*ID*
Parent : `Sandbox`
Children : `<Sandbox>`
BlockedBindingTypes : `<Binding>`

Eine Sandbox wird auf einen speziellen Basismodell-Komponententyp *Sandbox* abgebildet, der die angesiedelten Komponenten als Subkomponenten bezüglich eines bestimmten Kanaltyps *hosts* besitzt. Dies ist in Abbildung 7.6 schematisch abgebildet: Die Abbildung der Sandbox selbst enthält lediglich die Information der Kom-

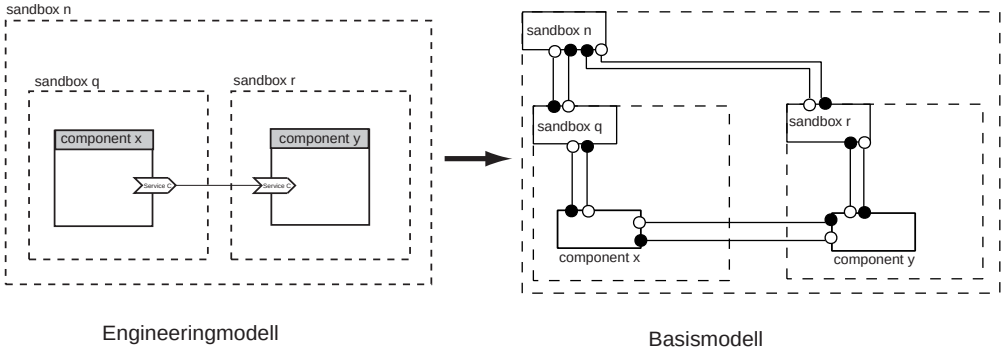


Abbildung 7.6: Die Abbildung einer Sandbox und deren Hostingrelationen auf Komponenten im Basismodell

ponente "Sandbox" – die Information welche Komponenten auf ihr angesiedelt sind. Welche Subkomponentenrelationen also im Basismodell ausgedrückt werden müssen, wird durch die Abbildung der Hosting Relation ausgedrückt. Ein Hosting wurde im Engineeringmodell durch das folgende Schema ausgedrückt:

Hosting :*ID*
Unit : `Unit`
Host : `Sandbox`

Die Hostingfunktion wird folgendermaßen auf die Bindungsfunktion abgebildet, wobei sie auf Komponenten vom Typ Sandbox eingegrenzt wird:

$$contains\_Sandbox : SBX \times NW(\mathbf{CH}) \rightarrow Bool$$

$$\begin{aligned} contains\_Sandbox(sb, nw) \stackrel{Def}{=} \; & (sb.Father \neq \varnothing) \Rightarrow \\ & (\exists\, nw_{father} \subset nw, k_f \in nw : \\ & type(k_f) = hosts \wedge k_f \in in.nw_{father} \wedge \end{aligned}$$

$$
\begin{aligned}
&\mathit{contains\_Sandbox}(\mathit{sb.Father}, \mathit{nw}_{\mathit{father}}))\\
&\wedge\\
&(\mathit{sb.Children} \neq \varnothing) \Rightarrow\\
&(\forall\, c \in \mathit{sb.Children}:\\
&\exists\, \mathit{nw}_{\mathit{child}} \subset \mathit{nw}, k_c \in \mathit{nw}:\\
&\mathit{type}(k_c) = \mathit{hosts} \wedge k_c \in \mathit{in}.\mathit{nw}_{\mathit{child}} \wedge\\
&\mathit{contains\_Sandbox}(c, \mathit{nw}_{\mathit{child}}))
\end{aligned}
$$

Das Prädikat *contains_Sandbox* sagt aus, ob das Basismodell-Pendant der übergebenen Engineeringmodell-Sandbox in dem übergebenen Basismodellnetzwerk *nw* enthalten ist. Dazu wird in der ersten Hälfte die übergeordnete Sandbox (*sb.Father*) überprüft, im zweiten Teil die Menge der enthaltenen Sandboxes (*sb.Children*). Die Hostingrelation wird durch einen speziellen Kanaltyp (*hosts*) ausgedrückt.

Eine Sandbox wird zusammen mit ihren Hosting-Relationen wird auf eine Komponente und eine Menge von Kanälen vom Typ *hosts* abgebildet.

7.6 Zusammenfassung

In diesem Kapitel wurde die Abbildung des informellen Engineeringmodells auf das formale Basismodell gezeigt. Es soll hierbei noch einmal erwähnt sein, das die Zielsetzungen der beiden Modellschichten unterschiedlich sind. So ist die Aufgabe des Engineeringmodells, eine geradlinige Unterstützung der Entwicklung durch geeignete und handhabbare Abstraktionen zu bieten. Die Aufgabe des Basismodells ist es, eine exakte und formal verifizierbare Semantik der einzelnen Entitäten zu definieren. Durch die Abbildung des Engineeringmodells auf das Basismodell werden die Abstraktionen durch bekannte und formale Konstrukte erklärt und die Möglichkeit gegeben, verschiedene Beschreibungsformen in Beziehung zu setzen.

Zusammenfassend lautet die Abbildung grob gesagt:

- **(Freie) Dienste** bestimmen das Blackbox-Verhalten, nicht jedoch die interne Struktur. Sie werden daher durch **logische Netzwerke** spezifiziert, wobei ihr durch Verhaltensbeschreibungen (z.B. STDs) spezifiziertes Verhalten in Form von Prädikaten die Eingangs- auf Ausgangsströme abbildet.
- **Komponenten** bestimmen (indirekt durch ihre angebotenen Dienste) Verhalten und (durch ihre Subkomponenten) die interne Struktur. Sie werden daher durch **technische Netzwerke** spezifiziert. Die jeweiligen angebotenen Dienste und deren benötigte Dienste legen das Blackbox-Verhalten der Komponente als Verhaltens-Prädikate fest.
- **Sandboxes** werden auf spezielle Komponenten des Basismodells, so genannte **Container-Komponenten**, abgebildet.

- **Bindung und Hosting** Relationen werden durch **Kanäle** im Basismodell repräsentiert, wobei das Hosting durch einen gesonderten Kanaltyp zur Sandboxkomponente repräsentiert wird.

Die Merkmale spontaner Komponentensysteme, ihre Repräsentation im Engineeringmodell und die dem entsprechenden Konstrukte im Basismodell werden in der folgenden Tabelle 7.1 zusammengefasst.

Eigenschaft	Abbildung im Basismodell	Abbildung im Engineeringmodell
Mobile Softwarekomponente	Menge der Komponenten-netzwerke, die sowohl black- als auch glass- boxkonsistent gem. der Spezifikation sind. (Technical Network-Spezifikation)	Hierarchische Komponente auf Sandbox angesiedelt. Mobilität durch Folge von Konfigurationen
Funktionale Abhängigkeit innerhalb logischer Architektur	Menge der Komponentennetzwerke, die zwar blackbox-konsistent gem. der Spezifikation sind, jedoch beliebige Struktur besitzen können. (Logical Network-Spezifikation)	Services: Freier Service als Surrogat für eine Komponente oder als Teilfunktionalität der Komponenten (needed bzw. provided).
Rechteumgebungen und damit verbundene Orte (locations).	Spezieller Komponententyp. Angesiedelte Komponenten sind Subkomponenten.	Sandboxes: Verwalten Kommunikationsrechte zu Komponenten ausserhalb der eigenen Grenzen.
Logische Architekturspezifikation	Menge von Komponenten-netzwerken, die Spezifikation erfüllen.	Servicearchitecture: Szenario aus Komponenten, freien Services und Sandboxes
Technische Architekturspezifikation	Komponentennetzwerk (technical Network), das der Spezifikation genügt.	Konfiguration: Szenario ausschliesslich aus Komponenten und Sandboxes

Tabelle 7.1: Die Merkmale spontaner Komponentensysteme und die Abbildung auf ihre Pendants im Engineeringmodell sowie deren Repräsentation im formalen Basismodell. (siehe auch Tabelle 6.1 und 6.1)

Teil III

Architektur

KAPITEL

ACHT

Architekturbegriff spontaner Komponentensysteme

In diesem Kapitel werden der Architekturbegriff spontaner Komponentensysteme und dessen strukturelle Merkmale auf Basis der in dieser Arbeit vorgestellten Abstraktionen und Techniken erklärt.

Die Architektur spontaner Komponentensysteme unterscheidet sich von der statischer Systeme vornehmlich in den ausgeprägten mobilen und dynamischen Strukturen der Systeme. Dieser Dynamik wird in dem hier vorgestellten Modell dadurch Rechnung getragen, dass eine statische logischen Architektur auf eine Menge von technischen Konfigurationen abgebildet wird. Zwischen diesen technischen Konfigurationen kann das System dynamisch wechseln, ohne die Funktionalität zu verändern.

Die Eigenschaften eines Systems und seiner Architektur werden daher auf verschiedenen Ebenen, die jeweils die einzelnen Charakteristika herausmodellieren, gesondert behandelt. Wir stellen im folgenden gemäß dieser Eigenschaften die folgenden Architekturebenen dar:

Die logische Dienstebene, die vornehmlich die invarianten Strukturen eines spontanen Systems modelliert.

Die technische Konfigurationsebene, die die strukturell verschiedenen Konfigurationen, die jedoch alle dieselbe logische Architektur erfüllen, modelliert.

Die Implementierungsebene, die eine Konfiguration konkretisiert und die implementierungsabhängigen Eigenschaften regelt, wie etwa die Transaktionskontrolle beim Wechsel zwischen Konfigurationen derselben logischen Architektur.

Im nächsten Abschnitt wird die Architektur auf abstrakter, statischer Ebene beschrieben und anschließend deren Konfigurationen auf technischer, dynamischer Ebene. Danach wird auf die Übergänge zwischen einzelnen Konfigurationen und die damit einhergehenden Aspekte wie Transaktionskontrolle sowie Stabilität als wichtige Eigenschaften der Architektur spontaner Systeme eingegangen.

8.1 Aufbau einer Architekturabstraktion

Die Architektur eines spontanen Systems besteht aus einer eigenständig zu behandelnden *logischen Architektur* und einer Menge daraus resultierender *technischer Architekturen*, wobei die Menge der technischen Architekturen neben der logischen Architektur auch durch die eingesetzten Komponenten und Netzwerkumgebungen bestimmt wird.

Die logische Architektur wird durch die *Dienstarchitektur* spezifiziert. Diese bestimmt die funktionalen Abhängigkeiten zwischen einzelnen Rollen des Systems, ohne jedoch den Rollen jegliche Strukturinformation, beispielsweise durch Subkomponenten oder Typinformationen, zuzuschreiben. Es wird definiert, welche *Funktionalität* (Dienst) welche anderen Funktionalitäten benötigt und, vorausgesetzt sie sind im System vorhanden, wie sie kommunizieren. Diese Definition ist aber unabhängig davon, welcher Teilnehmer den jeweiligen Dienst letztendlich erbringt. Dieses Konzept der abstrakten Funktionalität ohne Strukturinformation ist bedingt vergleichbar mit Konzepten wie *Contracts* [Szy97] oder *Interfaces*, wie sie in den Programmiersprachen Java oder ObjectiveC zu finden sind. Der Unterschied zu Interfaces ist jedoch, dass Interfaces nur *anbietende* Einheiten sind, also eine Funktion anbieten, wobei nicht im Design festgelegt werden kann, welche Funktionalitäten diese wiederum benötigen. Interfaces sind vielmehr externe Sichten auf Objekte die zunächst kein Verhalten besitzen.

Die resultierende Menge an *technischen Architekturen* wird in Form einer Menge von *Konfigurationen* zu der jeweiligen Dienstarchitektur ausgedrückt. Diese bestehen ausschließlich aus Komponenten und Sandboxes als Aktoren, besitzen also insbesondere nicht mehr das Konzept freier Dienste als Surrogat für Komponenten. Diese werden durch die Deploymentabbildung durch Komponenten ersetzt.

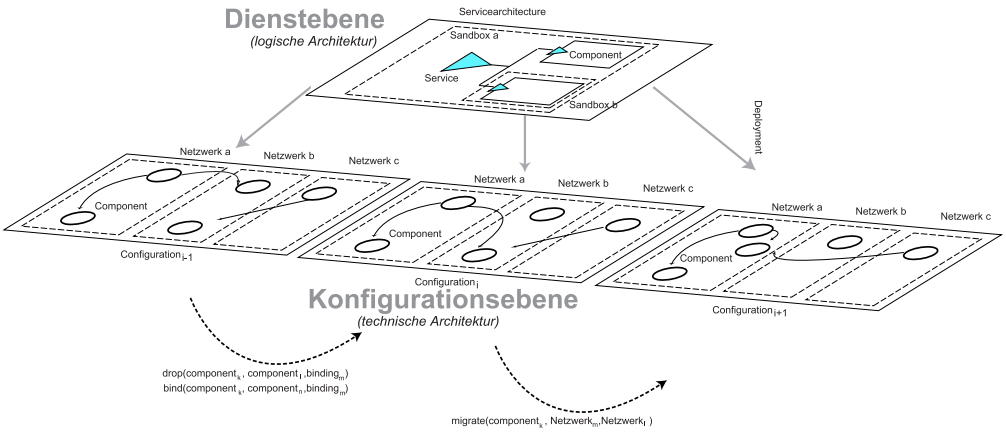


Abbildung 8.1: Aufbau einer Architekturabstraktion: Abbildung der (statischen) Dienstebene auf eine (dynamische) Konfigurationsebene

Zuletzt wird eine technische Konfiguration injektiv auf eine Implementierung abgebildet. Das bedeutet, dass jede Entität einer technischen Konfiguration ein direktes Pendant in der Implementierung, also im Code, besitzt. Die Realisierung muss insbesondere mit den durch die Plattform gegebenen Mitteln die transaktionssichere Umkonfiguration, also das Wechseln zwischen einzelnen Konfigurationen unter Beibehaltung

des aktuellen Berechnungszustandes, sicherstellen.

Wir betrachten also im folgenden drei Ebenen:

Die logische Dienstebene (Serviceebene) legt die funktionalen Eigenschaften des Systems fest, ohne dessen Struktur einzuschränken. Zusammen mit einer Menge von Komponenten und Umgebungen (sogenannten Einsatzumgebungen, engl. Environment) legt eine Dienstarchitektur die Menge der Konfigurationen auf der technischen Konfigurationsebene fest.

Die technische Konfigurationsebene wird durch eine Menge von Konfigurationen bestimmt, die alle eine logische Dienstspezifikation realisieren. Zwischen zwei Konfigurationen kann durch Operationen wie Verbindungsauf- und -abbau sowie Komponentenmigration gewechselt werden, wobei immer noch die invariant logische Architektur beibehalten wird. Eine technische Konfiguration legt über die logische Architektur hinaus Strukturinformation fest, wie beispielsweise, welche Instanz welchen Dienst erbringt oder wie eine bestimmte Instanz aufgebaut ist.

Die Implementierungsebene stellt dann die konkrete Realisierung einer oder mehrerer Konfigurationen einer Dienstarchitektur dar, die darüberhinaus konkrete Realisierungsdetails der jeweiligen Plattform abdecken muss.

Durch das in dieser Arbeit definierte Modell sind wir in der Lage, Entitäten auf allen drei Ebenen zu spezifizieren und deren Zusammenhänge zu definieren. Dies ermöglicht uns insbesondere, aus einer spezifizierten logischen Dienstarchitektur deren technische Architekturen bezüglich einer Einsatzumgebung zu errechnen. Zur Spezifikation und Definition bieten sich folgende Möglichkeiten:

- Die in Abschnitt 6 vorgestellte **Schemadefinition** von Elementen des Engineeringmodells.
- Die XML basierte **Definitionssprache SADL** (Service Architecture Definition Language) (siehe Abschnitt 9.1).
- Eine eingeschränkte **graphische Repräsentation** (siehe Abbildung 6.5).

Wir gehen jetzt in den folgenden Abschnitten auf die einzelnen Ebenen (logische, technische und Implementierung) näher ein und stellen insbesondere die zentralen Informationen vor, die auf der jeweiligen Ebene relevant sind sowie die Entitäten des Modells.

8.1.1 Logische Dienstebene

Auf der logischen Dienstebene werden Szenarien (Systeme) in Form von Dienstarchitekturen spezifiziert. Dies bedeutet, dass insbesondere Funktionen und deren Abhängigkeiten in Zusammenhang mit einer abstrakten Umgebung (z.B. Netzwerk) spezifiziert werden. Strukturinformationen, also insbesondere, welche Instanz oder Klasse

eine Funktion erbringt und wie diese Instanz aufgebaut ist, wird auf dieser Ebene nicht verlangt.

In der Dienstarchitektur stehen folgende Elemente zur Verfügung:

Freie Dienste sind Dienste, die stellvertretend für eine spätere Komponente verwendet werden.

Komponenten, die Dienste anbieten bzw. benötigen. Durch die Verwendung einer Komponente in einer Dienstarchitektur schränkt man die Menge an späteren Konfigurationen ein.

Sandboxes in der Dienstarchitektur legen die Anforderungen an die später konkreten Sandboxes in der technischen Konfiguration fest. Man könnte also von *abstrakten* Sandboxes sprechen, die die Anforderungen an die späteren Sandboxes ausdrücken.

Bindungen stellen die Kommunikationsbeziehung zwischen Komponenten (bzw. Diensten) dar.

Das Ziel einer Dienstarchitektur ist die Spezifikation eines Szenarios, wobei die Abhängigkeit von *Funktionalitäten* definiert wird und die Erreichbarkeit dieser Funktionalitäten. Es wird explizit ausgespart, *woher* sie kommt, das heist von welcher Entität diese Funktionalität erbracht wird.

Wir betrachten nun den zentralen Aspekt, der durch die Dienstarchitektur beschrieben werden soll, die funktionale Abhängigkeit zwischen den Komponenten.

Funktionale Abhängigkeiten

Ein Ziel der Dienstarchitektur ist es, die funktionalen Abhängigkeiten des Szenarios zu spezifizieren, ohne dabei die Menge der möglichen Realisierungen durch Angabe von struktureller Information einzuschränken. Durch die Behandlung von Diensten als Surrogat für eine diesen Dienst erbringende Komponente, wobei über sämtliche weiteren Eigenschaften dieser Komponenten keinerlei Aussage getroffen wird, ist es möglich, Spezifikationen vorzunehmen, die zwar die funktionalen Zusammenhänge darstellen, jedoch nicht aussagen wie die konkrete Struktur später aussieht. Fragen, die bei der Dienstarchitektur im Vordergrund stehen sind:

- Welche Funktionen spielen im Szenario eine Rolle (Dienste) ?
- Welche Dienste werden dadurch indirekt benötigt ?
- Wie hängen diese voneinander ab (Needed/Provided Services) ?
- Welche Netzwerkanforderungen sind gegeben (Sandboxes) ?
- Eventuelle Legacy Komponenten als Komponenten einbinden.

Durch das Erlauben von Diensten als eigenständige Aktoren auf der Dienstebene ist es möglich, die funktionale Abhängigkeit und das Verhalten des Systems zu spezifizieren, ohne die konkrete Struktur vorzugeben. Die freien Dienste werden später durch Komponenten, die diese Dienste anbieten, bei der Deploymentabbildung ersetzt, wobei nicht vorgegeben wird, welche Struktur die Komponenten haben müssen. Es werden also durch diese bewusste Unterspezifikation folgende Freiheitsgrade zugelassen: Dienste können von einer Komponente gebündelt werden, Komponenten können auf verschiedenen Sandboxes angesiedelt sein (solange der Dienst erreichbar ist), Komponenten können verschiedenen internen Aufbau besitzen (solange das Blackbox-Verhalten identisch ist).

8.1.2 Technische Konfigurationsebene

Die Konfigurationsebene stellt ebenso wie die logische Dienstebene eine abstrakte Beschreibung der Architektur des Systems dar. Es handelt sich also ebenso wie auf der logischen Ebene um abstrakte Entitäten, die nicht spezifisch an eine bestimmte Realisierung oder Plattform gebunden sind.

Im Gegensatz zur logischen Ebene, die die invarianten Strukturen ausdrückte, stellt die technische Architekturebene die verschiedenen *Konfigurationen* dar, die die logische Architektur erfüllen. Eine Konfiguration besitzt keine abstrakten Entitäten mehr, sondern Entitäten, die meist injektiv auf die späteren Softwarekomponenten der Implementierung abgebildet werden können. Dadurch besitzt die technische Konfiguration im Gegensatz zur logischen Architektur *Strukturinformation*, die angibt welche Instanz welchen Dienst erbringt bzw. konsumiert und wie die einzelnen Instanzen aufgebaut sind.

Die einzelnen Akteure der technischen Architektur sind:

Komponenten sind die einzigen Entitäten der technischen Architektur (keine freien Dienste mehr). Sie werden später injektiv auf die Implementationen abgebildet.

Sandboxes stellen keine Anforderungen an die Umgebung wie auf der logischen Ebene dar, sondern Spezifikationen der existierenden Umgebungen.

Bindungen sind identisch mit den Kommunikationsbeziehungen der logischen Ebene.

Ein entscheidender Punkt bei spontanen Komponentensystemen ist, wie bereits beschrieben, die Tatsache, dass eine logische Architektur nicht nur auf eine technische Konfiguration abgebildet wird, sondern auf eine Menge. Innerhalb dieser Menge von Konfigurationen, die alle die logische Architektur besitzen kann das System autonom zur Laufzeit von einer Konfiguration auf eine nächste wechseln.

Dieser Übergang zwischen Konfigurationen einer logischen Architektur wird im folgenden Abschnitt näher beschrieben.

Übergänge zwischen Konfigurationen

Wie bereits in Abschnitt 6.2.2 erwähnt, bildet die Deployment-Funktion *Deployment* eine logische Dienstarchitektur zusammen mit einer gegebenen Einsatzumgebung auf eine Menge von Konfigurationen ab. Diese Konfigurationen haben gemein, dass sie die logische Architektur, die durch die Dienstarchitektur beschrieben wurde, mittels Komponenten und Sandboxes aus der Einsatzumgebung realisieren:

$$Deployment : SA \times Env \rightarrow \mathcal{P}(\mathbf{CONF}_E)$$

Hierbei steht *SA* für die Menge aller Dienstarchitekturen, *Env* für die Menge aller Einsatzumgebungen und $\mathcal{P}$ für den Potenzmengenoperator. Betrachtet man nun die Menge der Konfigurationen $Conf_{ij} = \{C_1, \ldots, C_n\} = Deployment(S_i, E_j)$ einer Dienstarchitektur S_i und einer Einsatzumgebung E_j, dann sei nun eine Klassifikation über dem Deployment so definiert, dass innerhalb einer Klasse die Konfigurationen über derselben Menge an Komponenten definiert sind. Sei für $Conf \in \mathbf{CONF}_E : COMP.Conf$ die Menge aller Komponenten der Konfiguration *Conf*, so sei:

$$\begin{aligned} Klasse : Deployment(SA, Env) \quad &\rightarrow \quad \mathbf{K}(Deployment(SA, Env)) = K_1, \ldots, K_n \\ &\quad\ , \textit{ wobei } \forall j \in \{1, \ldots, n\} : \\ \forall Conf_h, Conf_i \in K_j \quad &: \quad COMP.Conf_h = COMP.Conf_i \end{aligned}$$

Die Sandboxes sind per Definition in allen Konfigurationen identisch, wobei nicht genutzte Sandboxes auf der Wurzelsandbox (Ether) angesiedelt sind. Es ist leicht zu sehen, dass sich die Konfigurationen einer Klasse nur in der Kommunikationsstruktur, also den Bindungen und der Ansiedlung, also dem Hosting, unterscheiden. Innerhalb jeder Klasse ist eine *Umkonfiguration* durch die folgenden atomare Operationen darstellbar:

- **move***(c: component, l1: sandbox, l2: sandbox)*
- **bind***(c1: component, c2: component, b: binding)*
- **drop***(c1: component, c2: component, b: binding)*

Der Wechsel zwischen zwei Klassen erfolgt durch die folgenden Operatoren:

- **add***(c: component,l: sandbox)*
- **kill***(c: component)*

Wir definieren nun den Operator $\rightsquigarrow$ als eine endliche Folge von move, bind und drop Operationen. Es gilt, dass innerhalb einer Menge von Konfigurationen zu einer logischen Architektur gilt:

$$\text{Sei} \qquad Conf_{ij} = Deployment(S_i, Env_j);$$

$$\forall C_k, C_h \in Conf_{ij} : C_k \rightsquigarrow C_h;$$

Das bedeutet informell, dass innerhalb einer Klasse von technischen Konfigurationen, die eine gemeinsame logische Architektur realisieren, von einer Konfiguration *i* auf eine andere Konfiguration *j* durch eine endliche Verkettung von Komponentenmigration und Kanalallokation bzw. Deallokation, gewechselt werden kann (siehe Abbildung 8.2).

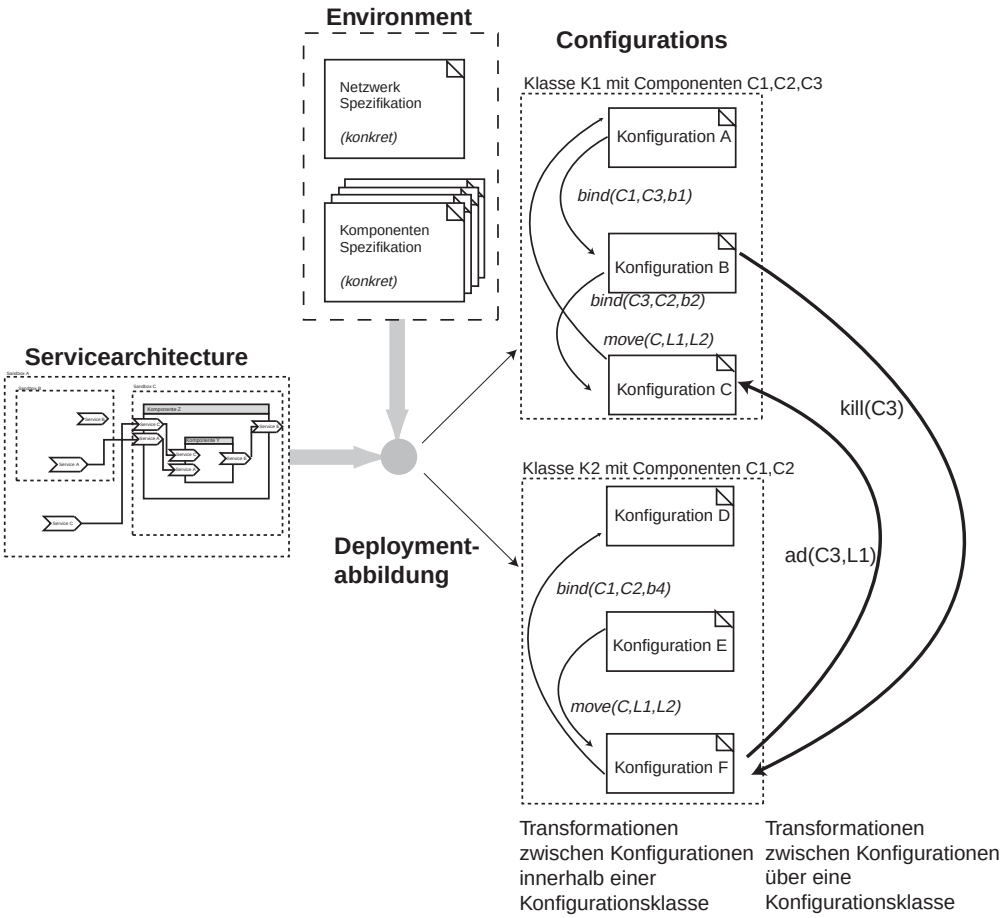


Abbildung 8.2: Transformationen zwischen Konfigurationen

Man beachte hierbei, dass diese Konfigurationswechsel immer noch auf einer Abstrakten Ebene stattfinden und lediglich aussagen, dass beide Konfigurationen dieselbe logische Architektur realisieren und wie der Wechsel durch die Basisoperationen (move, bind, drop) aufgebaut ist. Wie der Wechsel realisiert werden kann, das heist, wie er im laufenden System von statten geht und was dabei beachtet werden muss, wird in folgenden Abschnitt 8.1.3 behandelt.

Es sei hier ebenfalls erwähnt, dass hier von einer statischen Einsatzumgebung, also einer fixen Menge von Komponenten und Sandboxes, ausgegangen wird. Wenn neue Komponenten ins Spiel kommen, zum Beispiel weil ein neues Gerät ins Netzwerk eingebunden wird, so ist neue Deploymentabbildung vorzunehmen.

8.1.3 Implementierungsebene

Die Implementierungsebene gehört nicht mehr zu dem im Rahmen dieser Arbeit entwickelten Modell. Jedoch soll die Abbildung der bis dahin noch abstrakten Entitäten aus der technischen Architekturbeschreibung auf ausführbare Softwarekomponenten und die dabei zu beachtenden Nebenbedingungen gezeigt werden.

Wie bereits im vorherigen Abschnitt erwähnt, wird jede Komponente der technischen Architektur injektiv auf eine Softwarekomponente einer spontanen Plattform, wie Java/Jini oder .NET abgebildet. Die abstrakten Entitäten, wie beispielsweise benötigte oder angebotene Dienste werden gemäß der Abbildung, wie sie in Abschnitt 7 skizziert ist, vorgenommen.

Die einzelnen Akteure der Implementierung seien hier am Beispiel der Implementierungssprache Java gewählt:

Klassen und deren Instanzen stellen die Komponenten dar.

Interfaces stellen die angebotenen Dienste der Klasse dar – Referenzen auf Interfaces die benötigten Dienste.

Netze, sowie Rechner verbunden mit ihren Sicherheitseigenschaften stellen die Sandboxes dar.

RPC (Remote Procedure Call) Mechanismen, wie RMI oder CORBA, stellen die verschiedenen Calltypes der Bindungen dar. Selbstverständlich existiert auch der lokale Aufruf als Bindung mit Calltype “local”.

Wir betrachten im Folgenden speziell die Bedingungen bei Konfigurationsübergängen.

Transaktionssichere Umkonfiguration

Wie bereits erwähnt, gilt es bei der Implementierung die Transaktionssicherheit bei Konfigurationswechseln innerhalb einer logischen Architektur zu gewährleisten. Glücklicherweise existieren hierzu bereits Ansätze aus der Forschung. Wir berufen uns hierbei auf die bereits in Abschnitt 2.1 beschriebenen Arbeiten von Kramer und Magee.

Wir definieren im folgenden ein Verfahren zur Gewährleistung von Transaktionssicherheit bezüglich Diensten, das auf dem hier vorgestellten Modell aufbaut. Hierbei verwenden wir teilweise Begriffe und Methoden aus [KM90], die wir auf die Entitäten unseres Modells angepasst haben. Zunächst definieren wir den Begriff der *Transaktion*, mit dem wir arbeiten:

Definition 8.0: Transaktion

Eine *Transaktion* ist ein Nachrichtenaustausch zwischen zwei Komponenten bezüglich eines Dienstes bzw. einer Bindung. Hierbei gilt:

1. Nur die Senke der Bindung initiiert eine Transaktion.

2. Die Transaktion beeinflusst den internen Zustand der Parteien und besteht aus einer endlichen Sequenz von Nachrichten.

3. Eine Transaktion wird als abschließbar in endlicher Zeit angenommen.

4. Nur die initiierende Komponente, also die Senke der Bindung, kann die Transaktion beenden.

Eine Transaktion nach der obigen Definition ist demnach nicht mit dem klassischen Transaktionsbegriff aus dem Bereich der Datenbanken zu vermischen.

Unter *Transaktionssicherheit* bei einer Umkonfiguration verstehen wir nun im Folgenden, dass eine Transaktion, wie sie hier definiert wurde, durch eine Umkonfiguration im laufenden System, wie sie in Abschnitt 8.1.2 definiert wurde, nicht verletzt wird. Um eine transaktionssichere Umkonfiguration in einem laufenden System zu gewährleisten, müssen verschiedene Eigenschaften sichergestellt werden:

- Änderungen müssen das System in eine die logische Architektur realisierende Konfiguration in endlichen Schritten überführen.
- Die Änderungen sollten den Systemablauf möglichst nicht beeinflussen.

Der erste Punkt ist leicht mit der Zerlegung der Umkonfiguration in atomare Operatoren zu zeigen (siehe vorherigen Abschnitt). Jede der atomaren Operationen ist in endlicher Zeit abschliessbar, also auch eine aus endlich vielen atomaren Operationen zusammengesetzte.

Um, wie im zweiten Punkt gefordert, eine Umkonfiguration sicher zur Laufzeit vornehmen zu können, ohne den Systemablauf erheblich zu stören, müssen Konzepte aus dem Konfigurationsmanagement (siehe Abschnitt 2.1) auf das Engineeringmodell übertragen werden. Dies sind insbesondere der passive und der Ruhe-Zustand einer Komponente.

Definition 8.1: Passive Komponenten

Eine Komponente *K* ist bezüglich eines ihrer angebotenen Dienste *PS passiv* genau dann, wenn

1. Sie nicht in eine Transaktion eines angebotenen Dienstes verwickelt ist.

2. Sie keine neuen Transaktionen eines Dienstes *aktiv*, also ohne Aufforderung durch einen angebotenen Dienst, beginnt.

Dies schließt explizit das Aufnehmen und Abarbeiten von Transaktionen auf benötigten Diensten aus, die für die Erbringung eines angebotenen Dienstes aufgenommen werden. Informell bedeutet es also, dass eine Komponente keine Transaktion anstösst, jedoch von aussen herangetragene Transaktionen abschließt.

Aufbauend auf diesem Begriff des *passiven* Zustands, kann jetzt ein weiterer Zustand definiert werden, der darüberhinaus sicherstellt, dass die Komponente sich nicht inmitten einer Transaktion befindet und weder eine neue Transaktion anstoßen wird noch von einer anderen Komponente im System zu einer solchen Transaktion aufgefordert wird. Dieser Zustand wird im Konfigurationsmanagement als *quiescent* (dt. Ruhezustand) bezeichnet.

Definition 8.2: Ruhezustand einer Komponente
Eine Komponente *K* ist bezüglich eines ihrer angebotenen Dienste *PS* im *Ruhezustand* genau dann, wenn

1. Die Komponente ist passiv bezüglich *PS*
2. Sie nicht auf eine Transaktion eines für diesen Dienst benötigten Dienst *NS* wartet.
3. Der Dienst wurde oder wird nicht von anderen Komponenten in Anspruch genommen.

Um eine Komponente in den Ruhezustand zu versetzten, reicht es aber nicht aus, die den Dienst direkt nutzenden Komponenten in den passiven Zustand zu versetzen, sondern vielmehr die transitive Hülle der Komponenten.

Beispiel: Wir betrachten den *SendSMSbyName* Dienst (siehe Abschnitte 4.3 und 11.1.1). Als Transaktion sehen wir den Nachrichtenaustausch, der bezüglich eines Dienstes notwendig ist. Ein Beispiel wäre beim Dienst *SendSMS* die Nachrichtensequenz, die zum verschicken der SMS (inkl. Quittungsmeldung) notwendig ist. Die Transaktion kann nur von der Komponente angestoßen werden, die diesen Dienst benötigt (also beispielsweise durch den Dienst *SendSMSbyName*). Die Komponente, die *SendSMS* anbietet, wäre diesbezüglich passiv, wenn der Ablauf des Dienstes abgeschlossen ist (also die Quittungsmeldung erfolgte) und sie keine Transaktionen von sich aus anstößt. Die Komponente, die *SMSbyName* anbietet, ist diesbezüglich im Ruhezustand, wenn sie diesbezüglich passiv ist und sie nicht auf das Schließen einer Transaktion der benötigten Dienste, also *SendSMS* und *Phonebook*, wartet.

Wir können nun Bedingungen für die atomaren Konfigurationsoperatoren definieren. Für die Operatoren zum Wechseln einer Konfigurationsklasse *add* und *kill* sind die Bedingungen trivial: eine Komponente kann stets zur Menge dazu genommen werden und besitzt danach zunächst keine Verbindung. Soll eine Komponente entfernt werden, so muss diese isoliert sein, d.h. sie darf keinerlei Bindungen besitzen.

Für die Operationen *bind*, *drop* und *move* lauten die Bedingungen wie folgt:

bind(c1,c2,s): Eine Komponente c_1 kann mit einer Komponente c_2 bezüglich eines Dienstes *s* transaktionskonsistent gebunden werden, wenn die Komponente, die den Dienst *s* benötigt, im Ruhezustand ist.

drop(c1,c2,s): Eine Komponente c_1 kann die Bindung mit einer Komponente c_2 bezüglich eines Dienstes *s* transaktionskonsistent löschen, wenn die den Dienst *s* benötigende Komponente im Ruhezustand ist.

move(c,S1,S2): Eine Komponente kann transaktionskonsistent von einer Sandbox S1 auf eine andere Sandbox S2 migrieren, wenn sie keinerlei Verbindungen mehr besitzt.

Um also eine jede transaktionssichere Umkonfiguration, sei es innerhalb einer Konfigurationsklasse, also durch *drop*, *bind* und *move* Operationen oder von einer Klasse zur anderen, also durch *add* oder *kill*, zu realisieren, wird ein Verfahren benötigt, das eine Menge von Komponenten in den Passiv- oder in den Ruhezustand versetzt.

Dieses Verfahren wird im folgenden Algorithmus angegeben, der eine Komponente K in den Ruhezustand bezüglich Ihres Dienstes S_p überführt, indem alle Komponenten, die den Dienst direkt oder indirekt aufrufen könnten in den passiven Zustand überführt werden.

Algorithmus "Ruhezustand"

Eingabe: Komponente K und Dienst S_p, wobei $S_p \in \mathit{ProvidedServices}(K)$
Ausgabe: System, wobei Komponente K bezüglich S_p im Ruhezustand ist

```
Ruhezustand(Component K, Service S) {
        if ProvidedServices(S) = ∅
          beende S;
        else
          for all s_i ∈ ProvidedServices(S) do
              let k_i = drain.Binding(s_i); // aufrufende Komponente
              Ruhezustand(k_i, s_i);
          loop
}
```

Durch die oben angegebenen Bedingungen und das hier vorgestellte Verfahren ist es also möglich, innerhalb aller Konfigurationen einer Dienstarchitektur und einer Einsatzumgebung, zur Laufzeit zu wechseln und dabei die Systemeigenschaften nur minimal zu stören. Dieses Transaktionsmodell wurde in der Codegenerierung der Entwicklungsumgebung (siehe Abschnitt 10.3.2) realisiert.

8.2 Architektureigenschaften und Qualitätskriterien

Ausgehend von dem Architekturbegriff und dem dieser Arbeit zugrundeliegenden Modell, können nun die verschiedensten Metriken und Qualitätskriterien von Architekturen für spontane Systeme festgelegt werden. Wir stellen hier eine Auswahl vor, wobei darauf hingewiesen wir, dass natürlich auch andere Kriterien denkbar sind und in verschiedenen Anwendungsfällen auch sinnvoll sein können.

Ein entscheidender Unterschied zu statischen Systemen ist die Tatsache, dass eine *logische* Architektur (Dienstarchitektur) auch im Zusammenhang mit der Menge von Komponenten, die sie ausfüllen soll, der Einsatzumgebung, betrachtet werden kann. Dies bedeutet, dass eine Dienstarchitektur für eine Einsatzumgebung *A* durchaus geeignet sein kann, für eine anders aufgebaute Einsatzumgebung *B* jedoch zu eingeschränkt ist.

In den folgenden Abschnitten werden verschiedene Architektureigenschaften definiert, die durch Erfahrungen bei der praktischen Realisierung von spontanen Komponentensystemen [FGH+99] entstanden. Diese können als Entwurfsrichtlinien beziehungsweise Qualitätsmerkmale betrachtet werden. Die Eigenschaften werden zum einen informell beschrieben, um dem Leser ein plastisches Verständnis des Sachverhalts zu ermöglichen, jedoch auch gleichzeitig durch Konstrukte des Engineeringmodells ausgedrückt, was letztendlich als Implementierung bei der Entwicklungsumgebung verwendet wird.

8.2.1 Redundanz

Redundanz spielt eine große Rolle bei spontanen Systemen. Fällt eine Komponente und damit mindestens ein Dienst aus, der im System eingebunden ist, so sollte eine andere Komponente, die diesen Dienst anbietet und noch nicht diesbezüglich gebunden ist, transparent einspringen.

Definition 8.3: Redundanz eines Dienstes
Die *Redundanz Red* eines Dienstes S_i in einer Dienstarchitektur SA_k bezüglich einer Einsatzumgebung (engl. Environment) Env_j ist die Anzahl an überschüssigen angebotenen Diensten dieses Diensttyps:

$$Red(S_i, SA_k, Env_j) \stackrel{Def.}{=} \mid \{ProvidedServices(S_i).Env_j\} \setminus \{NeededServices(S_i).SA_k\} \mid$$

Man beachte, dass die Redundanz *nicht* gleich der überschüssig *verwendbaren* Dienste ist. Liegt die Komponente, die den Dienst anbietet, beispielsweise auf eine Sandbox *B*, die den Calltype zur Sandbox *A* blockiert, auf dem die den Dienst benötigende Komponente hosted, so ist der Dienst zwar vorhanden, aber nicht unbedingt redundant. Nur wenn eine Umkonfiguration existiert, so dass alle Dienste des Systems gebunden sind und diese Komponente eingebunden ist, gilt der Dienst als redundant.

8.2.2 Breite

Die Breite eines Dienstes drückt die Gesamtmenge an benötigten Diensten aus. Da die bloße Betrachtung der direkt benötigten Dienste (needed Services) nicht die wahre Menge, nämlich auch transitiv benötigte Dienste, ausdrückt, existiert der Begriff der *Breite* eines Dienstes.

Definition 8.4: Breite eines Dienstes

Die *Breite* eines Dienstes ist die transitive Hülle der direkt und indirekt benötigten Dienste.

Ein Dienst mit einer geringen Breite lässt sich mit geringerem Aufwand umkonfigurieren, da weniger Komponenten in den Ruhezustand versetzt werden müssen. Dies ist nicht mit der Anzahl der benötigten Dienste zu verwechseln: benötigt ein Dienst S_n eine Reihe von atomaren Diensten $S_1, \ldots, S_k$, und ein Dienst S_m eine Reihe von nicht-atomaren Diensten $S_1, \ldots, S_l$, wobei $S_l \ll S_K$ so kann S_m trotzdem die größere Breite besitzen, weil ein nicht-atomarer benötigter Dienst wiederum eine Reihe von benötigten Diensten besitzt usw. Je breiter ein Dienst ist, desto aufwendiger ist eine

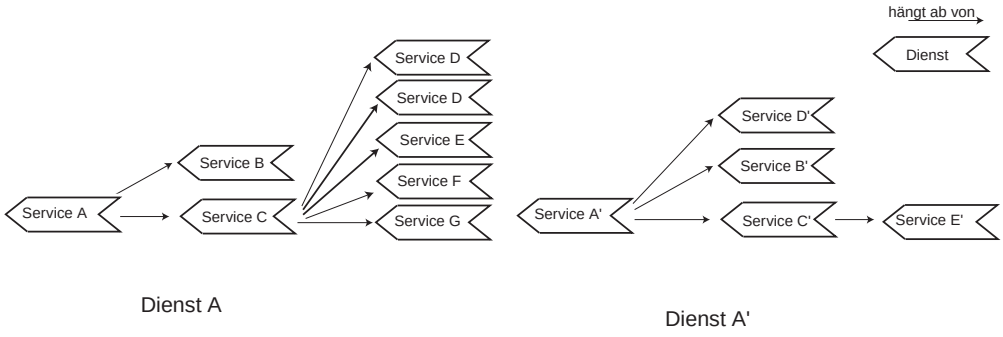


Abbildung 8.3: Die Breite von Dienst A (links) ist trotz der geringeren Anzahl der direkt benötigten Dienste größer als die des Dienstes A' (rechts).

mit ihm verbundene Umkonfiguration, da alle Dienste, von denen er abhängig ist, in den Passiv-Zustand überführt werden müssen.

8.2.3 Flexibilität

Durch die funktionale Abhängigkeit zwischen Diensten kann die *Flexibilität* eines Dienstes bestimmt werden. Bedingt ein Dienst die Komposition einer Menge von Diensten, so ergibt sich dadurch eine Menge von Konfigurationen bei der Deploymentabbildung. Abbildung 8.4 erläutert dies: Der Dienst $S1$ hängt von zwei Diensten, näm-

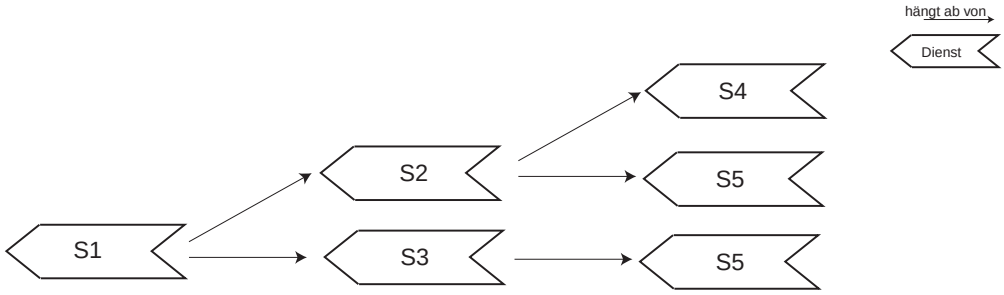


Abbildung 8.4: Flexibilität der Konfigurationen bei Abhängigkeiten auf logischer Dienstebene

lich $S2$ und $S3$ ab. Die Menge der Konfigurationen von $S1$ hängt also ab von der Menge der Konfigurationen von $S2$ und $S3$. Da diese wiederum von Diensten abhängen, kann die Anzahl der Konfigurationen rekursiv heruntergebrochen werden, bis atomare, also unabhängige Dienste, erreicht werden. Die Anzahl der Konfigurationen $\sharp K_{Diensti}$ ist also die Multiplikation der Anzahl der Konfigurationen der benötigten Dienste:

Definition 8.5: Flexibilität eines Dienstes

Die *Flexibilität* eines Dienstes in einer Dienstarchitektur bezüglich einer Einsatzumgebung entspricht der Anzahl der Konfigurationen des Dienstes. Die Flexibilität eines Dienstes $s_i \in \mathbf{S}_E$ bezüglich einer Einsatzumgebung E errechnet sich wie folgt:

$$\sharp K(s_i, E) = \begin{cases} \prod_{s_j \in NS.S_i} \sharp K(s_j, E) & für \ |NS.s_i| \neq 0 \\ |\{c \in C.E \mid \exists ps \in PS.c : type(ps) = type(s_i)\}| & für \ |NS.s_i| = 0 \end{cases}$$

Die Anzahl der Konfigurationen wird also rekursiv durch die Anzahl der Konfigurationen der benötigten Dienste errechnet.

8.2.4 Relevanz einer Komponente

Ist nun eine logische Dienstarchitektur *SA* gegeben sowie ein spezifizierter Komponententyp aus der Einsatzumgebung, so kann dieser Komponententyp eine bestimmte Relevanz für diese Dienstarchitektur besitzen.

Definition 8.6: Relevanz einer Komponente

Die *Relevanz* einer Komponente für eine Dienstarchitektur gibt die Anzahl an Diensten aus der Menge ihrer angebotenen Dienste an, die in der Dienstarchitektur als freie Dienste vorkommen.

Für eine Komponente C und eine logische Dienstarchitektur *SA* gilt:

$$Relevanz(C, SA) \stackrel{Def}{=} |\{s_i \in PS.C \mid \exists s_j \in FS.SA : \ type(s_i) \equiv type(s_j)\}|$$

Es ist leicht zu sehen, dass je höher Relevanz einer Komponente ist, desto stärker ist deren späterer Datenverkehr. Je niedriger sie ist, desto höher kann die Redundanz der späteren Konfiguration sein.

8.2.5 Autarkie einer Komponente

Wie in dem oberen Abschnitt bezüglich der Breite eines Dienstes erläutert, benötigt ein Dienst nicht nur die direkt benötigten Dienste, also die *needed Services* des Dienstes, sondern auch die damit transitiv benötigten Dienste. Eine Komponente wiederum bietet eine Vielzahl von Diensten an, worunter sich auch eine Teilmenge der transitiven Hülle der benötigten Dienste eines Dienstes *S* befinden können. Diese Teilmenge gibt die *Autarkie* einer Komponente an:

Definition 8.7: Autarkie einer Komponente

Die *Autarkie* einer Komponente gibt an, wieviele der für die angebotenen Dienste benötigten Dienst die Komponente selbst anbietet.

Für eine Komponente C mit deren needed Services $NS.C$ und provided Services $PS.C$ gilt:

$$Autarkie(C) \stackrel{Def}{=} \mid \{s \in PS.C \mid \exists\, ns \in NS.C : \; type(s) \equiv type(ns)\} \mid$$

Ein Beispiel hierfür ist in der Fallstudie in Abschnitt 11 zu finden: Das Mobiltelefon bietet den Dienst SendSMSbyName an, der die Dienste SendSMS und Phonebook benötigt. Beide werden jedoch von der Mobiltelefonkomponente angeboten – sie ist also vollständig abgeschlossen.

8.2.6 Stabilität

Stabilität ist eine wichtige Eigenschaft einer logischen Architektur bezüglich einer Einsatzumgebung.

Die folgende Abbildung illustriert die drei Begriffe anhand eines mechanischen Beispiels: ruht das stabile Dreieck optimal, so ist das lokal stabile zwar gegen kleinere Impulse resistent, kann jedoch durch starke umgeworfen werden. Das instabile kann durch kleinste Eingriffe aus dem Gleichgewicht gebracht werden, da es auf dem statisch instabilsten Punkt gelagert ist.

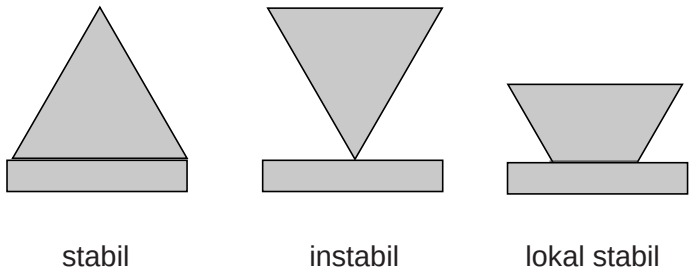


Abbildung 8.5: Stabilität

Definition 8.8: Stabilität einer Konfiguration

Eine Konfiguration K ist *stabil* bezüglich eines Dienstes S_s einer Komponente C_i, wenn es eine Konfiguration K_j in der selben Konfigurationsklasse gibt, die den Dienst S_s von C_i nicht benötigt.

Man beachte den Unterschied zu der oben definierten *Redundanz* eines Dienstes: die Redundanz sagt lediglich aus, dass ein überschüssiger und verwendbarer Dienst vorhanden ist, nicht jedoch, dass dieser durch Umkonfiguration auch verwendbar ist. Die

Stabilität sagt hingegen aus, dass eine solche Umkonfiguration ohne Hinzunahme von neuen Komponenten, die den Dienst anbieten, möglich sein muss.

Folgende Abbildung 8.6 zeigt beispielsweise eine Konfiguration, bestehend aus fünf Komponenten, A, B, C, D und X, die auf drei Sandboxen, U, V und W, angesiedelt sind und bezüglich zwei Diensten via drei Bindungen, s, t und u, gebunden sind. Der Dienst Q ist redundant bezüglich der Komponente A. Angenommen die Bindung s, welche den Dienst Q verbindet, ist vom Calltype “RMI” und dieser werde von Sandbox U blockiert. Die Bindung u hingegen sei vom Calltype “SOAP”, welcher von Sandbox V blockiert wird (siehe Abbildung 8.6). Es wäre also ein Umkonfiguration notwendig, indem Komponente A von Sandbox U nach V migriert. Dies würde jedoch die Bindung u zur Komponente D blockieren. Obwohl der Dienst Q redundant ist, ist die Konfiguration *nicht* stabil.

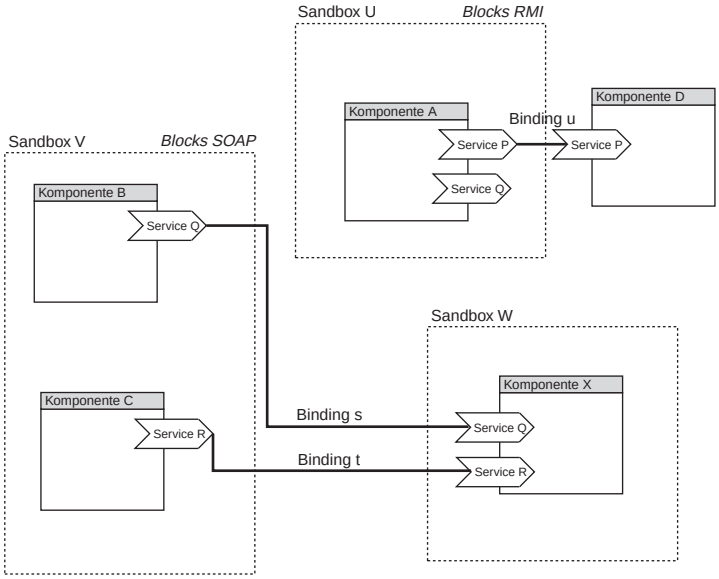


Abbildung 8.6: Beispielkonfiguration (siehe Text)

Wir unterscheiden zwischen den folgenden drei Stufen der Stabilität:

instabil ist eine Konfiguration, wenn sie bezüglich keiner ihrer Dienste stabil ist.

lokal stabil vom Grad n ist eine Konfiguration dann, wenn n ihrer Dienste stabil sind.

stabil ist eine Konfiguration, bestehend aus k Dienste dann, wenn sie lokal stabil vom Grad $n = k$ ist.

Solche Qualitätskriterien, wie beispielsweise die Redundanz eines Dienstes, können in mancherlei Szenarien sinnvoll sein: in mobilen Anwendungen kann ein entfernter Dienst anfällig sein, da er potentiell unterbrochen werden kann. Ist für den Fall eines Verbindungszusammenbruches noch ein redundanter Dienst auf dem Endgerät lokal vorhanden, so steigert dies die Zuverlässigkeit des Systems.

Die hier vorgestellten Architektureigenschaften erheben keinerlei Anspruch auf Vollständigkeit. Vielmehr sollen diese Beispiele erläutern wie sinnvolle Architektureigenschaften und -qualitätskriterien in dem Modell ausgedrückt werden können. Durch diese maschinell auswertbare Beschreibung von Architektureigenschaften kann eine qualitativ bessere autonome Komposition bei spontanen Komponentensystemen erreicht werden, wie das folgende Anwendungsbeispiel zeigt.

8.2.7 Anwendungsbeispiel: Architektur berücksichtigender Trading-Service

Wie in Abschnitt 3.2.2 beschrieben basiert das Auffinden und Binden von spontanen Komponenten auf so genannten *Lookup-* oder *Trading-Services,* bei denen Dienste aufgrund einer Dienstbeschreibung eine Referenz zur einer Komponente erhalten, die diesen Dienst anbietet. Diese Dienstbeschreibung basiert beim momentanen Stand der Technik auf reiner Schnittstellenbeschreibung, wobei lediglich Datentypen verschiedener Plattformen durch Techniken wie XML oder Java plattformübergreigreifend zur Verfügung gestellt werden.

Der Vorteil von der Verwendung von Verhaltensinformation wurde in dieser Arbeit anhand des Verwendens von Automaten als Kontrollinstanz für Aufrufsequenzen von Diensten bereits illustriert. Ein interessanter weiterer Schritt stellt die Verwendung nicht nur von isolierten Metainformation bezüglich der Komponente und deren Diensten dar, sondern auch architekturelle Informationen des momentanen Systems. So wäre es beispielsweise möglich, einen Lookup-Service zu konzipieren, der nicht nur Schnittstellenbeschreibungsdaten verarbeiten kann, sondern auch Architektur-Beschreibung in die Auswahl der zurückzuliefernden Komponente mit einbezieht (siehe Abbildung 8.7). So würde die Anfrage einer Komponente nach einem bestimmten Dienst *S* nicht irgendeine Komponente, die diesen Dienst anbietet, zurückliefern, sondern die Komponente, die in die Architektur des Systems, auf logischer oder technischer Ebene, am besten hineinpasst um Architektureigenschaften, wie sie hier in Abschnitt 8.2 beschrieben sind, zu erfüllen. Beispielsweise wäre eine Anfrage an einen Lookup-Service bezüglich eines Dienstes S, der in dem momentanen Umfeld, in dem er laufen soll, möglichst hohe Relevanz besitzt, nicht möglich. Auch eine permanente Anfrage an den Lookup-Service nach Diensten bzw. Komponenten, die die Redundanz der technischen Architektur erhöhen, ist mit den momentanen Metabeschreibungen der Komponentensysteme nicht ausdrückbar.

Ein solches System ist nur realisierbar, wenn die Beschreibungssprache, die die Metainformation ausdrückt, und das zugrundeliegende Modell Komponenten nicht nur auf Schnittstellenebene beschreiben können, wie dies bei den momentanen Ansätzen wie WSDL [CCMW01a] oder IDL [HOE96] möglich ist, sondern die Systeme auch bezüglich ihrer Architektur beschreiben kann. Mit dem in dieser Arbeit vorgestellten Modell mit seinen auf verbreiteten Technologien aufbauenenden Beschreibungstechniken wie der XML-basierten Definitionssprache SADL wäre ein solcher Lookup-Service realisierbar. Diese Definitionssprache wird im nächsten Kapitel vorgestellt.

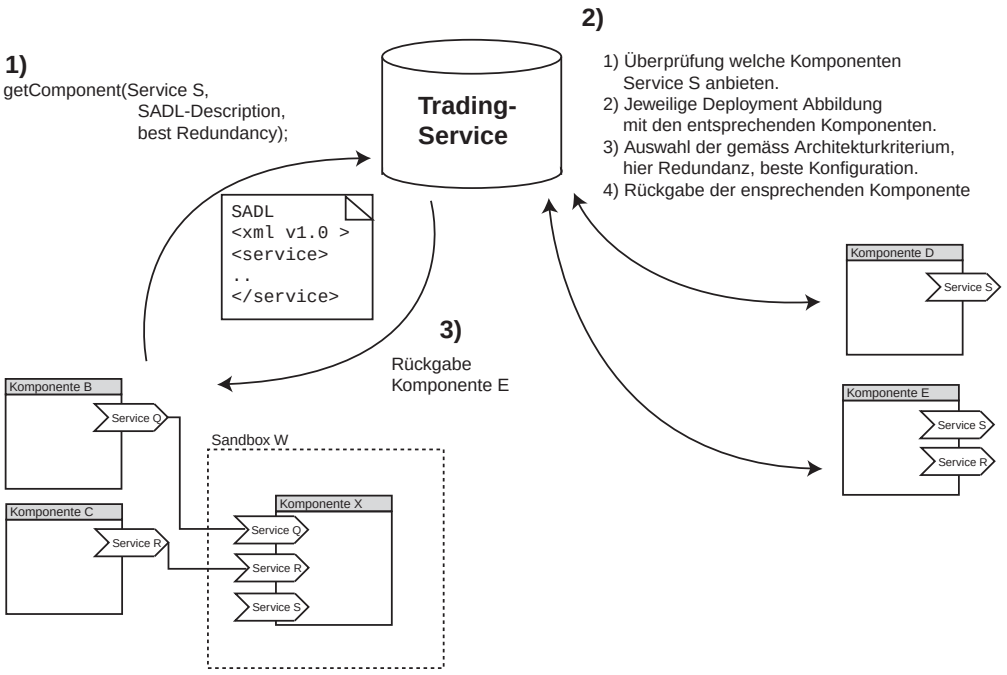


Abbildung 8.7: Ein das Modell und die damit mögliche Architekturbeschreibung auswertender Trading-Service

8.3 Zusammenfassung

In diesem Kapitel wurde gezeigt, wie durch das in der Arbeit vorgestellte Modell Strukturen spontaner Komponentensysteme dermaßen spezifiziert werden können, dass bestimmte gewünschte Systemeigenschaften definiert, gemessen und sichergestellt werden können. Da die Architektur spontaner Systeme ständigen Änderungen unterworfen ist und man zwischen invarianten und variablen Teilen unterscheiden muss, bietet sich eine Aufteilung der Architekturabstraktion in logische, invariante Teile und technische, dynamische Anteile an.

Die Anwendung solcher Architekturbeschreibung ist jedoch nicht nur auf den Entwurf, obgleich dies der Schwerpunkt dieser Arbeit ist, beschränkt. So sind Eigenschaften wie Varianz und Stabilität gewünschte Eigenschaften von spontanen Systemen, die beispielsweise durch eine Umkonfiguration durch Auftreten mobiler Geräte beibehalten werden sollen. Um solche Eigenschaften jedoch in einem spontanen und damit autonomen System messen und beibehalten zu können, müssen sie mit adäquaten Mitteln beschreibbar sein. Dies ist mit herkömmlichen Techniken, wie beispielsweise .NET oder Java/Jini nicht möglich, da nur isolierte Komponentenbeschreibungssprachen vorgesehen sind, nicht jedoch eine Beschreibung des Gesamtsystems, die eine Messung von Architekturkriterien und Qualitätsmerkmalen, wie sie hier vorgestellt wurden, möglich macht (siehe hierzu auch Abschnitt 12.3).

KAPITEL

NEUN

Spezifikation und Abbildung

Im Rahmen dieser Arbeit wurde das Engineeringmodell in zwei Formen prototypisch realisiert: einer Repository-Implementierung, die es ermöglicht, mit Werkzeugen auf Entitäten des Modells zuzugreifen und zu Verarbeiten und einer Beschreibungssprache auf XML Basis, die eine Festlegung des Engineeringmodells bezüglich Entitäten und deren Abbildung illustriert. In diesem Kapitel wird diese Definitionssprache (SADL - Service Architecture Definition Language) erläutert.

Anschließend wird die Abbildung des Engineeringmodells (und damit die Definitionssprache) auf einer Standardplattformen (Java bzw. JINI) definiert. Dadurch ist es möglich, mittels der Definitionssprache SADL spontane Komponentensysteme abstrakt auf Basis des Engineeringmodells zu spezifizieren, auf Basis dieser Spezifikation architekturelle Eigenschaften zu berechnen und Konfigurationen automatisiert zu generieren. Diese können dann von einem Werkzeug (siehe Abschnitt 10) in Codeskelette für Standardplattformen (z.B. JINI bzw. .NET) übersetzt werden.

Diese Entwicklungsumgebung wird dann exemplarisch in Kapitel 10 vorgestellt.

9.1 Spezifikation mit SADL

SADL (Service Architecture Definition Language) stellt eine Definitionssprache für das Engineeringmodell dar, die es ermöglicht, Spezifikationen in lesbarer, textueller Form anzugeben. Sie dient damit vornehmlich dazu, funktionale Abhängigkeiten zwischen Akteuren und Rollen festzulegen. Diese wiederum werden später durch ein Werkzeug auf eine Systemkonfigurationen gemäß des in den vorigen Kapiteln definierten Modells abgebildet. Im Gegensatz zu anderen Wide-Area-Computing Definitionssprachen, wie zum Beispiel der WSDL (Web Service Description Language) [CCMW01a] werden in diesem Ansatz gemäß des vorgestellten Modells die invarianten funktionalen Abhängigkeiten eines Systems spezifiziert (in Form einer Dienstarchitektur) und dann in Zusammenhang mit gegebenen Umgebungsspezifikationen auf eine Menge von Konfigurationen abgebildet.

SADL bietet auch die Möglichkeit, Verhaltensspezifikationen in die Systemdefinition zu integrieren. Dazu können verschiedene Beschreibungstechniken, wie z.B. STDs (State Transition Diagramms) oder auch MSCs (Message Sequence Charts) verwendet

werden, oder eine pragmatische Lösung durch Gebrauch von standardisieren Identifikatoren für Standard-Verhalten der Dienste, wie es z.B bei DCOM [Box98] praktiziert wird, eingesetzt werden.

In diesem Kapitel wird die Modellierungssprache knapp durch eine Reihe anschaulicher Beispiele angegeben. Der Aufbau des Engineeringmodells wird als bekannt vorausgesetzt. Eine ausführliche Syntaxdefinition in Form eines XML-Schemas befindet sich im Anhang.

9.1.1 Warum XML ?

SADL ist als XML (eXtensible Markup Language) Dialekt konzipiert und wird durch ein XML-Schema definiert. Der Grund für die Nutzung von XML im Gegensatz zur Definition einer Sprache auf herkömmlichen Wege (Übersetzerbau mittels Lex und Yacc) soll hier kurz umrissen werden.

XML wird dieser Tage als eine der am wichtigsten eingeschätzten Technologien betrachtet. Vieles der Einschätzungen sind Überschätzungen. Technisch betrachtet ist XML lediglich eine Notationsform für Baumstrukturen und eine Möglichkeit, Grammatik-Bäume zu beschreiben, vergleichbar mit einer BNF (Backus Naur Form). Allerdings stellt XML einen Standard dar, der als universelles Datenaustauschformat genutzt werden kann, und aufgrund seiner bewusst einfach gehaltenen Basisstruktur elegant erweitert werden kann. Verwendet man einen XML-Dialekt, der durch DTDs (Data-Type-Definitions) oder XML Schemas definiert wird, kann jeder XML Parser zum Parsen der definierten Sprache genutzt werden[1]. Es existieren mittlerweile ein Vielzahl von XML Dialekten, wie beispielsweise die WSDL (Web Service Description Language) [CCMW01b] oder WML (Wireless Markup Language) [Ltd98]. XML selbst entstand aus der SGML (Standardized General Markup Language) heraus und ist daher (wie auch an der Syntax erkennbar) eng mit HTML (Hypertext Markup Language), einem weiteren SGML Spross, verwandt. XML ist im Gegensatz zu HTML eine *Strukturdefinitionssprache*, die es ermöglicht, hierarchische Baumstrukturen von Dokumenten zu definieren. Neben dem Vorteil, einen offenen Standard zu verwenden, ergeben sich eine Vielzahl weiterer Vorteile:

XML besitzt eine lesbare Syntax. Im Gegensatz zu vielen anderen Datenformaten kann XML auf normaler Textzeichenbasis (Unicode) dargestellt werden. Dies hat zum einen den Vorteil, dass sowohl eine Maschine als auch der Mensch denselben Datensatz ohne Werkzeuge lesen und verstehen können. Zum anderen ist es dadurch möglich, dass alle Softwarewerkzeuge, die auf Textbasis arbeiten, wie zum Beispiel das Versionsverwaltungssystem CVS, ohne Änderungen einsetzbar sind.

XML ist mächtig genug, um objektorientierte Konzepte wie Vererbung und Polymorphie auszudrücken. Gerade in dem hier behandelten Fall, in dem ein komponentenorientiertes Modell in einer Sprachgrammatik ausgedrückt werden soll, ist es bequem für den Nutzer der Sprache, wenn die objektorientierten Konzepte strukturerhaltend in der Sprache wiederzufinden sind.

[1] XML Parser sind damit neben GUIs das zweite Beispiel für nutzbringende Komponententechnologie.

XML erspart das Bauen eines Parsers. Da die Syntax von XML definiert ist und verschiedenste XML Parser existieren, ist es nicht nötig, einen eigenen Parser zu erstellen. Die Definition der Sprachgrammatik in Form eines DTD (Document Type Definition), oder wie in diesem Fall eines XML-Schemas, reicht aus, um jedem XML Parsers die notwendigen Informationen bereitzustellen, um das jeweilige Sprachdokument zu parsen.

XML ist weitestgehend standardisiert und akzeptiert. XML und die in deren Umfeld definierten Modelle setzen sich zunehmend bei der Integration von Systemen durch. XML wird von vielen Herstellern als Datenformat verwendet. Da XML grundsätzlich ein Datenformat beschreibt, das auch in lesbarer Form dargestellt werden kann, kann eine Sprachdefinition auch als Definition des Austauschformates zwischen Systemen benutzt werden, was im Falle der hier vorgestellten Entwicklungsumgebung verfolgt wurde.

Durch die Verknüpfung zum Internet ist es einfach, Sprachdefinitionen zur Verfügung zu stellen. Durch die einfache Angabe der URL im Dokumentenkopf kann der jeweilige XML Parser auf die notwendigen Informationen via Internet zugreifen. Eine Installation ist nicht notwendig.

Ein letzter Grund ist auch die Tatsache, dass existierende Ansätze zur Beschreibung von Webservices als XML Dialekt definiert sind, diese jedoch kein Basismodell besitzen, wie es in dem hier propagierten Ansatz gemacht wurde. Ein Beispiel hierfür ist die *Webservice Definition Language* (WSDL), die unter anderem zur Beschreibung von Webservices im .NET Framework verwendet wird. SADL lässt sich nun aufgrund der selben verwendeten Techniken sehr gut mit WSDL vergleichen und erlaubt dadurch eine griffige Illustration der Vorteile des hier gewählten Ansatzes durch das Engineering- und Basismodell. Für eine Gegenüberstellung von SADL und WSDL siehe Abschnitt 9.2

9.1.2 SADL Spezifikation

Innerhalb dieser Arbeit kann keine Einführung in XML und die hier verwendeten XML Konzepte, wie Namespaces, Schemas oder XPointer gegeben werden. Der Leser sei bei Bedarf an die jeweiligen Dokumente des W3C [BPSM97b] verwiesen.

Durch SADL werden in einem *Projektdokument* folgende Elemente definiert:

`Service`: Eine Dienstinstanz wird durch die Angabe des Typs sowie ihrer Ansiedlung auf einer Sandbox auch ein Hosting definiert.

`Servicetype`: Ein Diensttyp bestimmt dessen Interface, die benötigten Dienste (needed Services) sowie eine Verhaltensspezifikation.

`Component`: Eine Komponenteninstanz wird durch ihren Typ sowie die Ansiedlung in Form eines Hostings definiert.

`Componenttype`: Der Typ einer Komponente wird durch Verweise auf die angebotenen und benötigten Dienste sowie das Interface der Komponente und deren

Verhalten in Form einer Spezifikation bestimmt. Im Gegensatz zum Dienst besitzt der Komponententyp auch Strukturinformationen in Form von Verweisen auf die Definition der Subkomponenten sowie deren Bindings.

`Sandbox`: Eine Sandbox wird durch die Mutter-Sandbox, die enthaltenen Sandboxes (jeweils Verweise auf deren Definitionen) sowie eine Liste von zu blockierenden Bindingtypen definiert. Da ein Bindingtyp (siehe unten) durch Quelle, Ziel und Calltype definiert ist, ist es möglich, Kommunikationen bezüglich dieser Kriterien auszufiltern. Ferner besitzt eine Sandbox noch Verweise auf die auf ihr angesiedelten Komponenten und Dienstinstanzen.

`Environment`: Ein Environment (Einsatzumgebung) definiert abstrakt eine Umgebung, die aus Komponenten und Netzwerkumgebungen existiert. Man beachte, dass keine Dienste frei, das heist ausserhalb einer Komponente, vorkommen.

`Servicearchitecture`: Eine Servicearchitecture ist schließlich das Ziel der Spezifikation. Sie ist eine Beschreibung der invarianten funktionalen Abhängigkeiten verschiedener Instanzen innerhalb einer Netzungebung, unabhängig von der jeweiligen Konfiguration. Sie besteht aus einer Menge von Komponenteninstanzen, freien Dienstinstanzen, Sandboxes sowie den dazugehörigen Binding- und Hosting-Beziehungen.

Zusätzlich werden indirekt folgende Subelemente entsprechend dem Engineeringmodell definiert:

- `binding` wird durch ein Tupel zweier Units (Component oder Service) sowie dem Calltype, über den die Kommunikation stattfindet, und eines Dienstes via dessen das Binding gesetzt ist, definiert.
- `hosting` wird analog durch ein Tupel, bestehend aus einer Unit und einer Sandbox, definiert.
- `messagetype` setzt die Nachrichtentypen fest, die über einen Kanal fließen. Sie besitzen bestimmte Parameter, die zu den jeweiligen Nachrichten gehören.
- `calltype` legt das Protokoll fest, mittels dessen die Nachrichten über einen Kanal fließen. Beispiel: CORBA-IIOP
- `behavior` enthält, wie im Engineeringmodell definiert, die Verhaltensinformation eines Dienstes. Im Falle von SADL spezifiziert dieses Feld *Interaktionsinformationen* bezüglich des Dienste. Diese wird in Form eines Zustands-Übergangs-Diagramms (STD) in XML Notation zur Verfügung gestellt (per URL). Siehe hierzu Kapitel 10.2.4.
- `interface` definiert die Signatur einer Komponente bzw. eines Dienstes. Es dient als Andockpunkt für spätere Binding-Relationen und wird durch die entsprechenden Messagetypes und den Calltype definiert.

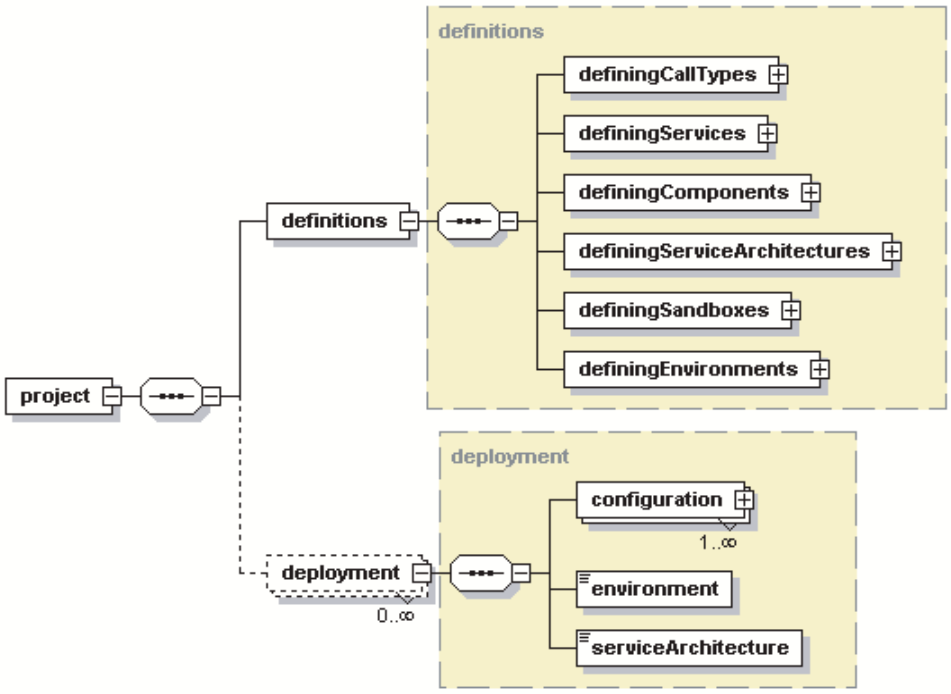


Abbildung 9.1: Aufbau eines SADL Spezifikationsdokumentes

SADL definiert mittels XML Components, Component-Types, Services, Service-Types, Sandboxes, Servicearchitectures, Environments, Configurations und die damit verbundenen Elemente wie Bindings etc. Jedes SADL Dokument ist nach dem folgenden Schema aufgebaut: Ein Dokument beginnt mit dem Element `definitions` als Wurzel. Innerhalb der definitions werden in den jeweiligen Blöcken Servicearchitectures, Environments, Components, Component-Types, Services, Service-Types und Sandboxes definiert. Die jeweilige Definitionssyntax folgt in den nächsten Abschnitten. Diese Struktur findet sich in den Spezifikationsdokumenten in der folgenden XML-Notation wieder:

```
<?xml version="1.0" encoding="UTF-8"?>
<sadl:project name="SADL_Beispiel">
 <sadl:definitions>
  <sadl:definingServices>
    <sadl:service>...</sadl:service>
                    ...
  </sadl:definingServices>

  <sadl:definingServiceArchitectures>
    <sadl:serviceArchitecture>...</sadl:serviceArchitecture>
                    ...
  </sadl:definingServiceArchitectures>

  <sadl:definingCallTypes>
    <sadl:callType>...</sadl:callType>
                  ...
```

```
  </sadl:definingCallTypes>

  <sadl:definingComponents>
    <sadl:component>...</sadl:component>
                    ...
  </sadl:definingComponents>

  <sadl:definingEnvironments>
    <sadl:service>...</sadl:service>
                  ...
    <sadl:environment>...</sadl:environment>
                       ...
  </sadl:definingEnvrionments>

  <sadl:definingSandboxes>
    <sadl:sandbox>...</sadl:sandbox>
                  ...
  </sadl:definingSandboxes>
 </sadl:definitions>

 <sadl:deployment name="delpoyment_one">
   <sadl:configuration>
    ...
   </sadl:configuration>
    ...
 </sadl:deployment>

</sadl:project>
```

9.1.3 Elemente der SADL

Da SADL als XML Schema definiert ist, werden manche Elemente, die nur auf der Schemaebene als abstrakte Elemente existieren, als Schemadefinition vorgestellt. Jene Elemente, die vom Nutzer dann als SADL Spezifikation definiert werden müssen, wie zum Beispiel Komponenten, werde als Beispiele vorgestellt.

Da es sich bei SADL um eine praktikable Definitionssprache handelt, wurden hier die Entitäten des Engineeringmodells um Typen erweitert. Dies geschah aufgrund des effizienteren Definieren, das Typen aufgrund der Referenzierbarkeit erlauben.

Units: Komponenten und Dienste

Units wurden im SADL-Schema als abstraktes Element definiert, was bedeutet, dass eine Unit niemals explizit instanziiert wird, sondern nur in Form von abgeleiteten Elementen – Komponente oder Dienst – existiert. Komponenten werde durch das folgende Schema (siehe Abbildung 9.2) definiert. Beispielsweise bezeichnet

```
<?xml version="1.0"?>
<component xmlns="http://www4.in.tum.de/sadl"
 name="Mobiltelefon">
  <providedServices>
         sendSMS,phonebook,sendSMSbyName
```

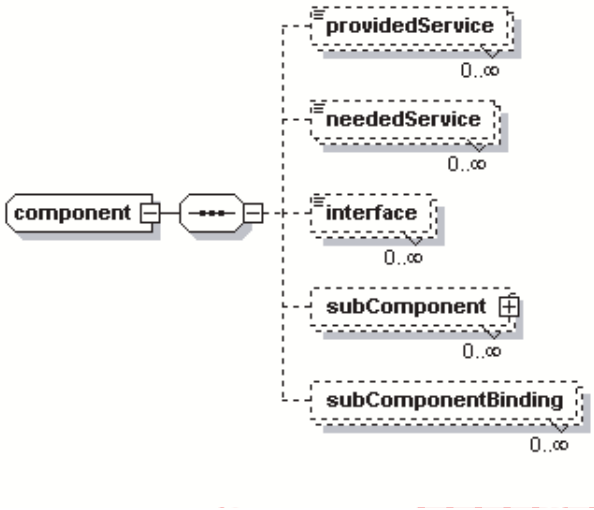


Abbildung 9.2: Aufbau einer SADL-Spezifikation einer Komponente

```
  </providedServices>
  <neededServices>
         phonebook
  </neededServices>
  <subcomponents>
     smsManager,smsByNameManager
  </subComponents>
  <subcomponentBindings>
    <binding source="S35" drain="smsByNameManager"
     service="phonebook" calltype="local/>
     ...
  </subComponentBindings>
</componentType>
```

einen Komponententyp `Mobiltelefon`, der über die Dienste `SendSMS`, `SendSMSbyName` und `Phonebook` verfügt. Eine Komponenteninstanz `S35` vom Typ `Mobiltelefon` wird von der folgenden SADL-Definition bestimmt:

```
<?xml version="1.0"?> <componentInstance
xmlns="http://www4.in.tum.de/sadl" name="S35">
   <type> Mobiltelefon </type>
</componentInstance>
```

Die angebotenen und benötigten Dienste einer Komponente werden durch *needed* und *provided Services* adressiert. Innerhalb der Komponente können sie auch nur Referenziert werden, sofern sie an anderer Stelle definiert wurden.

Dienste können gemäß dem Engineeringmodell in Dienstarchitekturen entweder als freie Teilnehmer oder als Teil von Komponenten auftreten. Hierzu werden innerhalb des ”defining-services”-Bereichs Dienste und Diensttypen wie folgt definiert:

```
<?xml version="1.0"?>
<service xmlns="http://www4.in.tum.de/sadl" name="sendSMSbyName">
```

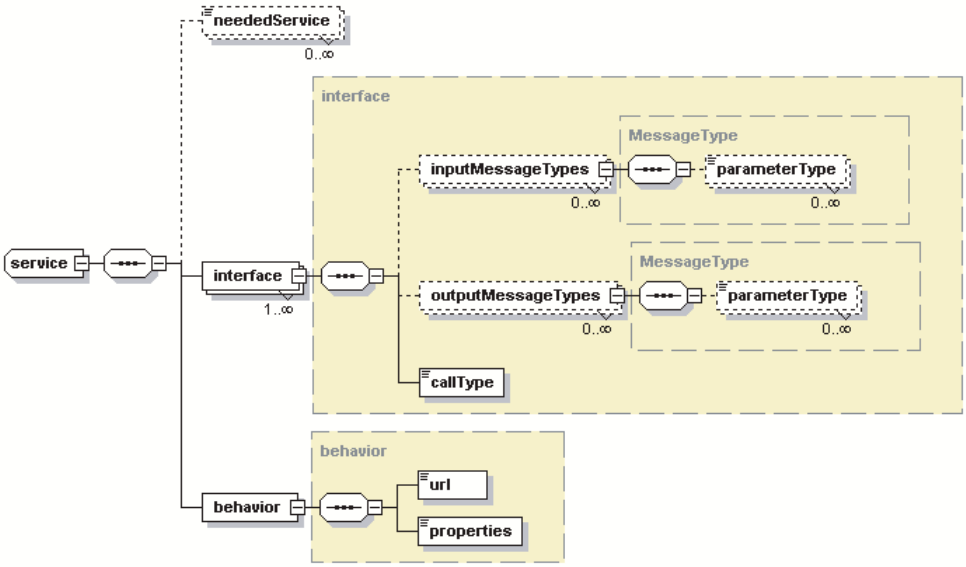


Abbildung 9.3: SADL Struktur einer Dienst-Spezifikation

```
<neededServices>
  sendSMS,Phonebook
</neededServices>
<interface name="sendSMSbyName_IF">
   <inputMessageType>
    sendSMSbyName(name:string,message:string)
   </inputMessageType>
   <outputMessageType>
    sendSMSbyName(success:bool)
   </outputMessageType>
   <callType>
    RMI
   </callType>
</interface>
<interface name="sendSMS_IF">
   <inputMessageType>
      sendSMS(number:int,message:string)
   </inputMessageType>
   <outputMessageType>
      sendSMS(success:bool)
   </outputMessageType>
   <callType>
    RMI
   </callType>
</interface>
<interface name="phonebook_IF">
   <inputMessageType>
      getPhoneNumber(name:string)
   </inputMessageType>
   <outputMessageType>
      getPhoneNumber(number:int)
   </outputMessageType>
```

```
      <callType>
       RMI
      </callType>
   </interface>
   <behaviour>
     <url>
     http://www4.in.tum.de/dante/Examples/MO/SMSbName.beh
     </url>
     <properties>
     http://www4.in.tum.de/dante/Examples/MO/SMSbName.prop
     </properties>
   </behaviour>
</service>
```

Ein Dienst besitzt für jede potentielle Bindung ein *Interface*. Ein Interface setzt sich aus einem Calltype sowie *Input-* und *Output-Messagetypes* zusammen. Ein Messagetype kann wiederum Parametertypes besitzen.

Sandboxes

Eine Sandbox ist die Wirtsplattform, die Komponenten und Dienste beinhaltet. Sie beinhaltet eine Referenz auf die Muttersandbox und eine Menge an enthaltenen Sandboxes. Die wichtigste Eigenschaft der Sandbox ist die Eigenschaft, Kommunikationsverbindungen der enthaltenen Komponenten und Dienste nach aussen abzublocken. Eine Sandbox enthält daher eine veränderbare Liste von zu blockierenden Bindungstypen. Es ist also möglich, alle Kanäle, die zum Beispiel CORBA Verbindungen repräsentieren, zu blockieren, oder alle Kanäle von oder zu einer bestimmten Komponenteninstanz. Der XML Code hat dementsprechend die folgende Form:

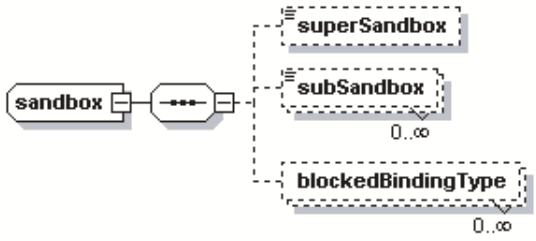


Abbildung 9.4: Aufbau der SADL Spezifikation einer Sandbox

```
<?xml version="1.0"?>
<sandbox name = "sunbroy78">
  <superSanbox>
    ether
  </superSandbox>

  <subSandboxes>
    JavaVM1
  </subsandbox>
```

```
  <blockedBindingTypes>
    <binding>
      <provider> * </provider>
      <consumer> * </consumer>
      <service> * </service>
      <callType> RMI </callType>
    </binding>
  </blockedBindingTypes>
</sandbox>

<sandbox name = "JavaVM1">

 <superSanbox>
  Sunbroy78
 </superSandbox>

 <subSandboxes>
 </subsandboxes>

 <blockedBindingTypes>
 </blockedBindingTypes>
</sandbox>
```

In dieser Beispieldefinition wurde eine Sandbox *Sunbroy78*, also ein Rechner, definiert, der eine Subsandbox *JavaVM*, eine Java Virtuelle Maschine, beinhaltet. Die Rechtekonfiguration der Subsandbox besagt, dass lediglich Kanäle von der SiemensS35 Komponente zu der Phonebookkomponente unabhängig vom Calltype geblockt werden. Die Rechtekonfiguration der Sandbox *Sunbroy78* setzt hingegen fest, dass sämtliche Kanäle unabhängig vom Ursprung oder Ziel, die das Kommunikationsmedium RMI nutzen, geblockt werden.

Binding & Hosting

In der Spezifikation werden sowohl Komponenten als auch Dienste *gebunden*, wenn ein angebotener Dienst einen benötigten Dienst ausfüllt. Das Binding wird durch ein geordnetes Tupel bestehend aus zwei Units (*Source* und *Drain*) sowie den Dienst, bezüglich dessen die Bindung stattfindet, beschrieben. Im folgenden Beispiel wird ein Binding zwischen zwei Telefonkomponenten, wie sie oben spezifiziert wurden, definiert, wobei die Komponenten bezüglich des Dienstes *SendSMS* gebunden sind. Andere Dienste könnten also von anderen Komponenten konsumiert bzw. produziert werden. Der Calltype gibt *RMI* an, beide Komponenten kommunizieren bezüglich des Dienstes *SendSMS* also via RMI (Remote Method Invocation).

```
<?xml version="1.0"?>
<binding name = "b1c23h42">
   <source> "TelefonA" </source>
   <drain> "TelefonB" </drain>
   <service> "sendSMS" </service>
   <callType> "RMI" </callType>
</binding>
```

Im Falle der Bindung mit einem freien Dienst ist der Dienst, bezüglich dessen gebunden wird, natürlich der triviale Fall, nämlich dieser Diensttyp des freien Dienstes. Im folgenden Beispiel *muss* der Dienst, bezüglich dem gebunden wird, *SendSMS* sein, da ein Kommunikationspartner ein freier Dienst vom Typ *SendSMS* ist:

```
<?xml version="1.0"?>
<binding name = "x34k56bc">
   <source> "TelefonA" </source>
   <drain> "sendSMS" </drain>
   <service> "sendSMS" </service>
   <callType> "" </callType>
</binding>
```

Ein *Hosting* setzt wie im Kapitel 6 beschrieben die Zuordnung der Units auf die Sandboxes fest. Die Konsistenzbedingung verlangt, dass jede Unit auf *genau einer* Sandbox angesiedelt ist. Hier ist zu beachten, dass die Ansiedlung im Falle einer verschachtelten Sandboxhierarchie nur die *unterste* Sandbox betrifft: Eine Sandbox, die eine weitere Sandbox enthält, hat nicht deren Units auf sich angesiedelt. Ein Hosting wird durch wie folgt definiert:

```
<?xml version="1.0"?>
<hosting host=sunbroy78 units=S35,S36>
</hosting>
```

zu beachten ist hierbei, dass Binding- und Hostingrelationen separat, also nicht innerhalb der Komponenteninstanz oder der Sandboxinstanz, definiert werden, um eine größtmögliche Modularität zu erhalten. Um später aus einer gegebenen Dienstarchitektur- und Einsatzumgebung-Definition die zugehörigen Konfigurationen zu bestimmen, ist lediglich die Berechnung und Erzeugung der Binding- und Hostingrelationen notwendig. Die davon unabhängigen Komponenten und Sandboxdefinitionen können beibehalten werden.

Dienstarchitektur & Einsatzumgebung

Dienste können, wie bereits oben erwähnt, auch ohne die Einbettung in eine Komponente als sog. freie Dienste in einer Dienstarchitektur-Spezifikation verwendet werden. Komponenten können hierbei optional zur Modellierung von Legacy Komponenten mit einbezogen werden.

Die SADL-Spezifikation für eine Dienstarchitektur ist also eine Menge von Komponenten, eine Menge freier Dienste, eine Menge an Sandbox-Spezifikationen sowie dazugehörige Binding und Hosting Relationen. Der Aufbau einer SADL-Dienstarchitektur ist in Abbildung 9.5 dargestellt.

Eine Einsatzumgebung (engl. Environment) stellt sich als zwei Mengen von Komponenten und Sandboxes dar. Dieses einfache Konstrukt ist in Abbildung 9.6 dargestellt. Man beachte hierbei, dass es sich um Sandbox-Instanzen handelt, im Gegensatz zu den Sandboxen in einer Servicearchitecture, die ja abstrakte Anforderungen an die spätere Sandboxes spezifizieren.

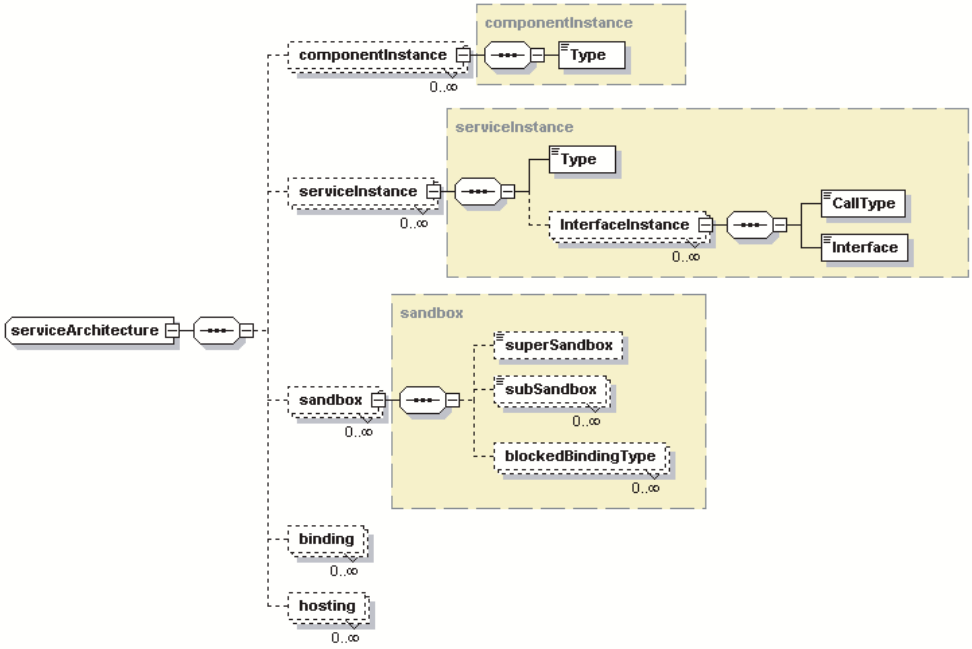


Abbildung 9.5: Aufbau einer SADL Dienst-Architecture Spezifikation

9.1.4 Verhaltensspezifikation

Im Engineeringmodell besitzt ein Dienst ein bestimmtes Verhalten, das durch eine Verhaltensspezifikation gemäß dem Basismodell angegeben wird. Welcher Grad an Verhaltensbeschreibung sowie welche Spezifikationstechnik verwendet wird, ist nicht vorgeschrieben. FOCUS bietet hierfür eine Reihe von Spezifikationstechniken an, die hier nicht näher beschrieben werden sollen. In der hier beschriebenen Realisierung des Engineeringmodells wird das *Blackbox-Verhalten* eines Dienstes mit *endlichen Automaten* spezifiziert. Hierfür werden als graphische Notation State Transition Diagramms (STDs) verwendet. Obgleich natürlich auch andere Techniken, wie beispielsweise MSCs oder reine Prädikate als Beschreibungen möglich wären, wurden hier Automaten gewählt. Da Automaten als Graphen dargestellt und spezifiziert werden können, bietet sich diese Abstraktion für ein graphisches Spezifikationswerkzeug an. In der Werkzeugumgebung können die Automaten jedoch – wie andere Spezifikationstechniken, die eventuell noch folgen – auf eine Reihe von Prädikaten abgebildet werden. Dazu wird das Verfahren nach [BS00] verwendet.

Für endliche Automaten existieren verschiedene Definitionen. Für unsere Zwecke ist die folgende geeignet:

Ein deterministischer endlicher Automat D ist ein Tupel $D = (E, Z, \delta, F, z_0)$, wobei:

- die endliche Menge E das *Eingabealphabet* ist,
- die endliche Menge Z das *Zustandsalphabet* ist,

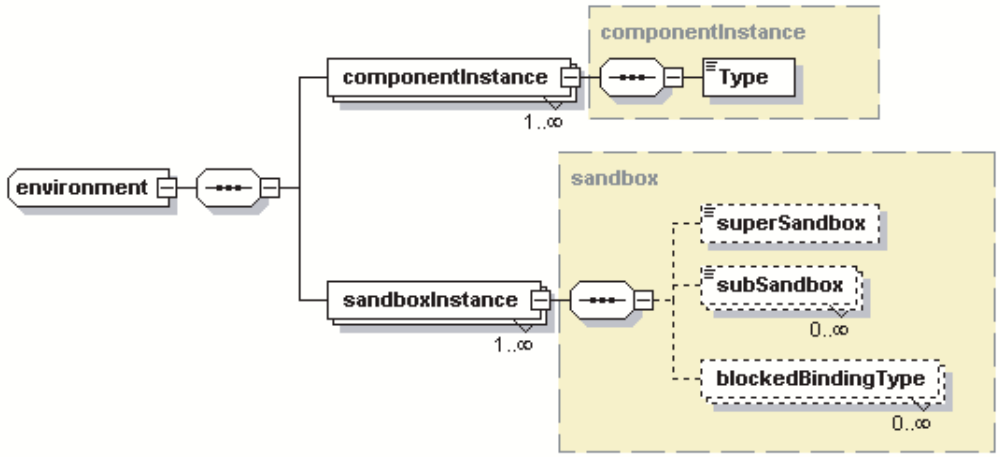


Abbildung 9.6: Struktur einen SADL Environment-Spezifikation

- die totale Funktion $\delta : Z \times E \rightarrow Z$ die *Zustandsübergangsfunktion* genannt wird,
- die Teilmenge $F \subseteq Z$ die Menge der *akzeptiertenden* oder finalen Zustände ist,
- der Zustand $z_0 \in Z$ der *Startzustand* ist.

Das Konzept eines endlichen Automaten ist weitgehend bekannt, ansonsten sei der Leser auf [Rum96]verwiesen. Die hier verwendeten Zustands-Übergangs-Diagramme (engl. State-Transition-Diagrams, STDs), folgen der in [BS00] beschriebenen Notation. Diese Diagramme werden anschliessend durch ein ebenfalls in [BS00] beschriebenes Verfahren auf jene prädikatenlogichen Ausdrücke abgebildet, die im Engineeringmodell und im Basismodell das Verhalten eines Dienstes beschreiben.

In der hier behandelten Realisierung (SADL) wird das Verhalten eines Dienstes, angegeben durch die URL im Feld `behavior`, durch ein STD in XML Notation angegeben. Für die Interaktion des hier verwendeten Beispiels eines “Print” Dienstes und einer Dienstnehmers sei folgender Automat definiert:

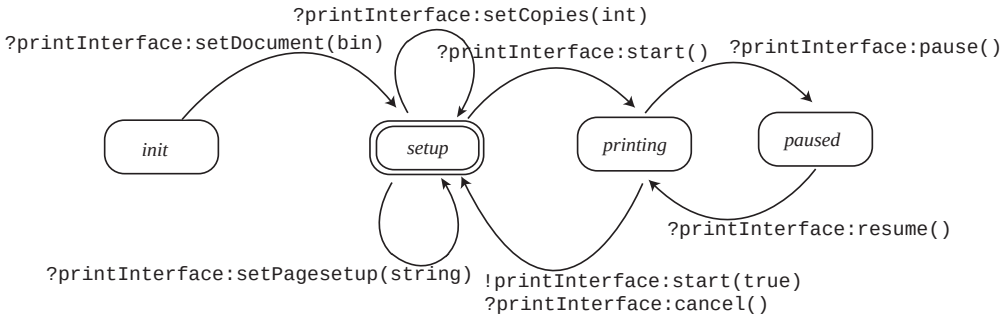


Abbildung 9.7: Der Automat des “Print” Dienste

Diese Verhaltensinformation wird später von der Werkzeugumgebung verwendet um

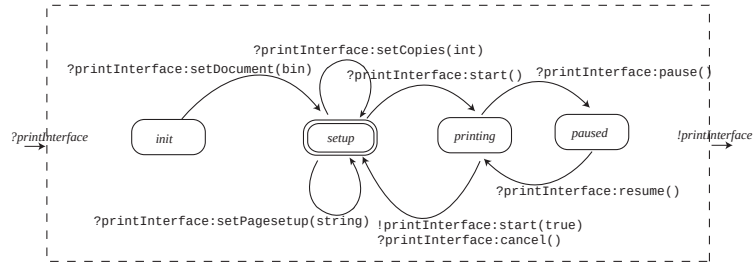
Mechanismen in die letztendlichen Codeskelette, die diesen Ablauf überwachen, einzuweben (siehe Abschnitt 10.2.4).

Das STD selbst besitzt das folgende Schema:

[STD :*Print*]

local $k \in \mathbf{N}$

init $k = 0 \wedge$ *device* $= 0$



device	*?printInterface*	*!printInterface*	k'
0	*setDocument(bin)*	-	1
0	*setCopies(int)*	-	1
0	*start()*	-	2
0	*pause()*	-	3
0	*resume()*	-	2
0	*cancel()*	-	1
1	-	*start(true)*	1
1	*setPagesetup(string)*	-	

9.2 Vergleich WSDL – SADL

In diesem Abschnitt soll kurz ein Vergleich zwischen SADL und der "Web Service Description Language (WSDL)" gezogen werden. Die WSDL ist ein zur Standardisierung angemeldeter Vorschlag des W3C, der maßgeblich von einem Zusammenschluss zwischen den Firmen IBM und Microsoft entwickelt wurde [CCMW01a].

WSDL soll die Möglichkeit bieten, sog. Web-Services – also Dienste im Internet – abstrakt zu spezifizieren, um somit Metainformation zur Verfügung zustellen, aufgrund derer Dienste beim jeweiligen Trading-Service – in diesem Falle handelt es sich um den UDDI-Dienst (siehe 3.2.2) – Dienste auswählen zu können. Da Webservices auf den verschiedensten Plattformen und Implementierungssprachen aufsetzen können, galt es in der WSDL zunächst eine Abstraktion zu definieren, die von Schnittstellen und den damit verbundenen Datentypen abstrahiert. Das folgende Beispiel zeigt eine WSDL Spezifikation eines sehr einfachen Webservices, der nur eine Operation (Methode) *getLastTradePrice* besitzt und auf diese Anfrage den Preis in Form eine Fließkommazahl zurückgibt. Um eine kompakte Form und Lesbarkeit zu ermöglichen, was bei XML nicht immer gegeben ist, verwenden wir ein vereinfachtes Format, bei der Einzelheiten wie Namespaces etc. weggelassen wurden

```
<?xml version="1.0"?>
<definitions name="StockQuote"
...
    <types>
          ...
           <element name="TradePriceRequest">
              <complexType>
                  <all>
                      <element name="tickerSymbol" type="string"/>
                  </all>
              </complexType>
           </element>
           ...
    </types>

    <message name="GetLastTradePriceInput">
        <part name="body" element="xsd1:TradePriceRequest"/>
    </message>
    ...

    <portType name="StockQuotePortType">
        <operation name="GetLastTradePrice">
           <input message="tns:GetLastTradePriceInput"/>
           <output message="tns:GetLastTradePriceOutput"/>
        </operation>
    </portType>

    <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
        <soap:binding style="document" transport="http://..."/>
        <operation name="GetLastTradePrice">
           <soap:operation soapAction="http://example.com/TradePrice"/>
           <input>
               <soap:body use="literal"/>
           </input>
           <output>
               <soap:body use="literal"/>
           </output>
        </operation>
    </binding>

    <service name="StockQuoteService">
        <documentation>My first service</documentation>
        <port name="StockQuotePort" binding="tns:StockQuoteBinding">
           <soap:address location="http://example.com/stockquote"/>
        </port>
    </service>

</definitions>
```

Man sieht an diesem Beispiel, dass ein Dienst in WSDL als Netzknoten (Dienst) definiert ist, der bestimmte Schnittstellen (Ports) besitzt, über die Nachrichten (Messages) eines gewissen Typs (Type) fliessen. Ein WSDL-Binding (nicht zu verwechseln mit einem Binding im Engineeringmodell) ist eine Festlegung eines WSDL-Ports auf ein bestimmten Protokolltyp, beispielsweise SOAP. Ein WSDL-Binding ist also mit einem Protokoll-Adapter vergleichbar. Es ist weder möglich zu definieren, wie dieser Nachrichtenfluss definiert ist, noch eine Architektur, also beispielsweise ein Netzwerk, zu

Merkmal	SADL	WSDL
Datentyp-Abstraktion	XML	XML
Formale Basis	FOCUS	–
Verhaltensbeschreibung	möglich, z.B. STDs etc.	nicht möglich
Komponenten	Components	–
Dienste	Freie Dienste	Dienst (=Signatur)
Umgebungen	Sandbox	–
Kommunikationsarten	Binding	Bindung
Architektur	Service-Architecture, Configuration	–

Tabelle 9.1: Gegenüberstellung von WSDL und SADL

definieren. Eine Trennung zwischen logischer und technischer Architektur ist daher ebenfalls nicht vorgesehen. Die Unterschiede zwischen WSDL und SADL werden in der folgenden Tabelle 9.1 noch einmal gegenübergestellt.

Zusammenfassend kann man also sagen, dass WSDL in seiner jetzigen Form lediglich eine Interface Definition Language (IDL - siehe Abschnitt 2.2) für Internetdienste ist und nur in der Datenabstraktion mit SADL vergleichbar ist. Als Auswahlkriterium für eine Anfrage bei einem Tradingservice ist unter WSDL nur die abstrakte Beschreibung einer Schnittstelle möglich. Architekturelle Kriterien, wie sie in Abschnitt 8.2 beschrieben und in SADL ausdrückbar sind, sind unter WSDL nicht möglich.

9.3 Abbildung auf Standard Plattformen

Technische Architekturen (Konfigurationen) des Engineeringmodells werden bei der Entwicklung schließlich auf Codeskelette abgebildet. Codeskelette sind Objekte und Klassen, die zwar die konkrete Struktur, also Bezeichner und Methodenrümpfe, besitzen, jedoch die Methodenkörper aussparen. Dies ist ein gebräuchliches Verfahren bei werkzeugnterstütztem Softwareengineering [HSSS96].

In den folgenden zwei Abschnitten soll das grundlegende Mapping auf die im Rahmen dieser Arbeit gewählten Zielplattform(Java/JINI) skizziert werden. Die Abbildung auf anderer Plattformen, z.B. C♯ bzw. .NET, ist analog. Für eine vollständige Abbildung sei der Leser auf die Implementierung der Werkzeugumgebung (siehe Kapitel 10) verwiesen.

9.3.1 J2EE: Java & Jini

Jini ist das spontane Komponentenframework von SUN, das auf Java und dessen Verteilungsmechanismus RMI (Remote Method Invocation) aufbaut. Es wurde bereits in Kapitel 3.2.2 beschrieben.

Ziel ist es, jene Informationen, die im Modell spezifiziert wurden, auf die Java Plattform abzubilden und zu nutzen.

Dienste innerhalb einer Komponente, also needed und provided Dienste, werden auf Java-Interfaces abgebildet. Freie Dienste sind abstrakte Elemente und treten bei technischen Architekturen nicht mehr auf. Ein Java-Interface ist folgendermaßen aufgebaut:

$$JavaInterface \quad := \quad (name, Method_1, ..., Method_n);$$

wobei

$$Method_i \quad := \quad (name, ReturnType, ParameterType_1, ..., ParameterType_k);$$

Ein Dienst *S*, definiert als

$$S = (NS, IF, BB);$$

wobei *NS* die Menge der needed Services ist und *IF* das Interface bestehend aus Ein- und Ausgabenachrichten (*I* und *O*) die aus den jeweiligen Messagetypes $MT_1, \ldots, MT_i$ und den jeweiligen Parametertypes $P_1^{MT_j} \ldots P_m^{MT_j}$ bestehen. Das Blackbox-Verhalten BB sei als FSM in XML spezifiziert.

Ein Dienst *S* wird durch die Abbildung $F : S \rightarrow JI$ nun folgendermaßen auf ein Java-Interface *JI* abgebildet:

$NS_i \in NS$	$\longrightarrow$	Verweis auf eine Instanz ein Objekt, das das Java-Interface $JI_i = F(NS_i)$ implementiert
$MT_i \in I \cup MT_i \in O$	$\longrightarrow$	$Method_i$, wobei $Method_i := (MT_i, PT^{MT_i \in O}, PT_1^{MT_i}, ..., PT_m^{MT_i})$

Der jeweilige Calltype des Dienstes bewirkt, dass das Interface über bestimmte Mechanismen aufgerufen wird. Diese Mechanismen sind für den Calltype und die jeweilige Zielsprache definiert. Beispielsweise muss ein Javainterface beim Calltype *RMI* via RMI aufgerufen werden, was bestimmte Infrastruktur-Interfaces wie Naming-Services (JNDI) etc. verlangt.

Beispiel:
Der folgende in SADL spezifizierte Dienst:

```
<?xml version="1.0"?>
<service xmlns="http://www4.in.tum.de/despiau/sadl">
<service name="Print">
  <interface>
     <inputMessageType name="setDocument">
      <ParameterType name="Document">String</ParameterType>
     </inputMessageType>
     <inputMessageType name="setCopies">
      <ParameterType name="copies">int</ParameterType>
     </inputMessageType>
     <inputessageType name="startJob">
     </inputMessageType>
     <inputMessageType name="pauseJob">
     </inputMessageType>
     <inputMessageType name="resumeJob">
     </inputMessageType>
     <inputMessageType name="cancelJob">
     </inputMessageType>
```

```
     <outputMessageType name="startJob">
      <ParameterType name="success">boolean</ParameterType>
     </outputMessageType>
    <CallType>RMI</CallType>
  </interface>
  <behavior>
    <url>PrintUrl</url>
    <properties/>
  </behavior>
</service>
```

wird gemäß der oben definierten Zuordnung auf das folgende Javainterface abgebildet:

```
import java.rmi.Remote
import java rmi.RemoteException;
import java.util.List;

public interface Print {
// Generated automaticall by DANTE
// Interface withoutBB behavior

public void setDocument(string Document);
public boolean startJob;
public void pauseJob();
public void resumeJob();
public void cancelJob();

}
```

Angebotene *Dienste* werden als Java-Interfaces realisiert, wobei deren Verhalten als Schnittstellen Überwachen eingesetzt wird (siehe hierzu Abschnitt 10.2.4 und 11.4). Darauf wird im folgenden Absatz (Komponente) noch einmal eingegangen.

Eine *Komponente* wird nun auf ein Java-Objekt (Komponententyp auf Klasse) abgebildet, das dem internen Aufbau entspricht und welches die Provided-Services als Interfaces implementiert sowie Referenzen auf Objekte hält, welche die Needed-Services als Interface implementieren. Diese Referenzen referenzieren die Objekte jeweils als Interface, nicht als die jeweilige Klasse des Objekts. Durch die Verwendung von formalen Techniken im Modell, die zur Beschreibung des Verhaltens der Komponenten und Dienste genutzt werden können, ist es nun möglich, Verhaltensinformation als Überwachung in die Abbildung auf den Code einzuweben. In der hier vorgestellten Implementierung von SADL und dem DANTE Werkzeug wurden State-Transition-Diagrams (STDs) als Spezifikationstechnik des Blackbox-Verhaltens gewählt (siehe hierzu 9.1.4). Eine Komponente wird nicht nur auf ein Codeskelett, welches die Klassen und Methodensignaturen vorgibt, abgebildet. Vielmehr wird dieses Codeskelett in eine Containerklasse eingebettet, welche die nach aussen anzubietenden Interfaces (also die Dienste) gemäß des spezifizierten Blackbox-Verhaltens überwacht. Verletzt die kooperierende den Dienst nutzende Komponente dieses Verhalten, indem sie eine ungültige Aufrufsequenz nutzen will, so wird der Vorgang unterbrochen. Eine prototypische Implementierung eines Codegenerators wird in Abschnitt 10.2.4 beschrieben.

Diese Überwachung der gültigen Nutzungssequenzen ist bei spontan kooperierenden Komponenten von großem Interesse in der praktischen Anwendung: Bei einer ungül-

tigen Aufrufsequenz kann die das Blackbox-Verhalten verletzende Komponente ermittelt werden und damit die Verschuldung in eventuelle Billingmodelle einbezogen werden.

Sandboxes werden auf existierende Umgebungen, die die Interaktionsform bestimmen, abgebildet. Beispiele hierfür sind Java Virtuelle Maschinen (JVM), Rechner oder Subnetze. Alle diese Umgebungen können die Kommunikationsform bestimmen und sind dementsprechend in der Entwurfsphase beispielsweise in SADL entsprechend zu spezifizieren.

Binding & Hosting sind die jeweiligen Zuordnungen unter Komponenten, also Java Objekten und Sandboxes, also Rechner, Netze etc. (Hosting)

9.4 Zusammenfassung

In diesem Abschnitt wurde eine Spezifikationssprache (SADL) definiert, die es ermöglicht, Spezifikationen gemäß dem definierten Engineeringmodell textuell zu beschreiben. Dies erfolgt mit Hilfe einer XML Definition. Dem ein oder anderen Leser mag diese Art von Spezifikationssprache umständlich zu lesen und nur schwer handhabbar erscheinen. Dies ist zum einen richtig, zum anderen werden die Gründe, die für diesen Weg sprechen (siehe auch Abschnitt 9.1.1), klarer, wenn man noch einmal die Aufgaben dieser Beschreibungsform betrachtet:

- Ein Ausauschformat zwischen Werkzeugen. Daher sollte das Format von möglichst vielen Plattformen unterstützt werden.
- Ein Beschreibungsformat, das vom System zur Laufzeit ausgewertet werden soll und von allen Plattformen unterstützt werden soll.
- Es sollte eine weit verbreitete Datenabstraktion besitzen.
- Es sollte auch für den Menschen lesbar sein.
- Die benötigten Werkzeuge (parser, Scanner etc.) sollten leicht zu implementieren sein.

Der Vorteil eines formalisieren Modells hinter der Beschreibungssprache zeigt sich bei einem Vergleich mit herkömmlichen Ansätzen wie WSDL: hier ist nur eine datenabstrakte Beschreibung der Schnittstellen möglich, jedoch kein Verhalten und keine Architektur. Daher ist auch eine Trennung zwischen logischer Architektur und technischen Konfigurationen vorgesehen. Durch eine Übersendung einer SADL Beschreibung des momentanen Systems und des gewünschten Dienstes an einen Tradingservice anstelle einer WSDL Beschreibung wäre es möglich, nicht nur eine funktional passende Komponente auszuwählen, wie es beispielsweise bei UDDI der Fall ist, sondern die Komponente, die aufgrund architektureller Kriterien wie beispielsweise Stabilität am besten in das System passt.

Abschliessend wurde SADL, stellvertretend für das Engineeringmodell, auf eine Beispielplattform, in diesem Fall Java in Kombination mit Jini, abgebildet. Die komplette

Abbildungskette, von Basismodell über Engineeringmodell bis hin zur Implementierungsplattform, ist in der folgenden Tabelle zusammengefasst.

Eigenschaft	Abbildung im Basismodell	Abbildung im Engineeringmodell	Java/Jini (Beispielplattform)
Mobile Softwarekomponente	Menge der Komponenten-netzwerke, die sowohl black- als auch glass- boxkonsistent gem. der Spezifikation sind. (Technical Network-Spezifikation)	Hierarchische Komponente auf Sandbox angesiedelt. Mobilität durch Folge von Konfigurationen	Um Kontroll-Wrapper Erweiterte Jini-Komponente (nur Code-Migration)
Funktionale Abhängigkeit innerhalb logischer Architektur	Menge der Komponentennetzwerke, die zwar blackbox-konsistent gem. der Spezifikation sind, jedoch beliebige Struktur besitzen können. (Logical Network-Spezifikation)	Services: Freier Service als Surrogat für eine Komponente oder als Teilfunktionalität der Komponenten (needed bzw. provided).	Provided-Service: Interface Needed-Service: Referenz auf Interface.
Rechteumgebungen und damit verbundene Orte (locations).	Spezieller Komponententyp. Angesieldete Komponenten sind Subkomponenten.	Sandboxes: Verwalten Kommunikationsrechte zu Komponenten ausserhalb der eigenen Grenzen.	Java Virtuelle Maschine, Recherplattform, durch Firewall begrenztes Netzwerk etc.
Logische Architekturspezifikation	Menge von Komponenten-netzwerken, die Spezifikation erfüllen.	Servicearchitecture: Szenario aus Komponenten, freien Services und Sandboxes	-
Technische Architekturspezifikation	Komponentennetzwerk (technical Network), das der Spezifikation genügt.	Configuration: Szenario ausschliesslich aus Komponenten und Sandboxes	Jini-Code der entsprechend angesiedelten Komponenten

Tabelle 9.2: Die Merkmale spontaner Komponentensysteme und die Abbildung auf ihre Pendants im Engineeringmodell, deren Repräsentation im formalen Basismodell und die letztendliche Abbildung auf Java (bzw. Jini) (siehe auch Tabelle 6.1 und 7.1)

Teil IV

Anwendung

KAPITEL

ZEHN

Realisierung

Bei den in dieser Arbeit vorgestellten Techniken zur Modellierung und Architekturspezifikation ist die maschinelle Unterstützung der Spezifikation eine logische Folge. Gerade die Zahl der Konfigurationen zu einer Dienstarchitektur kann sehr schnell mit der Anzahl der Komponenten steigen und macht daher die automatisierte Abbildung und Konsistenzprüfung durch ein Werkzeug sinnvoll.

Daher wurden die hier vorgestellten Techniken in Form einer Modellierungsumgebung prototypisch realisiert. Dies geschah zum einen als Instanziierung des Modells, was dadurch eine Konkretisierung beispielsweise bei der Deploymentabbildung erfährt.

Zum anderen wurde das System aber auch realisiert, um die Praktikabilität des Ansatzes unter realen Bedingungen (d.h. existierende Plattformen und realistische Szenarien) zu illustrieren.

Im folgenden Abschnitt wird der Ablauf des Entwurfs und die Zusammenhänge der einzelnen Ergebnisse erläutert. Darauf folgt die technische Beschreibung der Werkzeugumgebung und die Abbildung der Entwurfsphasen auf die einzelnen Module des Systems. Anschließend werden die einzelnen Module, ihr Aufbau und ihre Produkte genauer beschrieben. Schließlich werden die implementierten Funktionalitäten des Systems sowie noch nicht realisierte aber mögliche Ergänzungen, beschrieben.

Im nächsten Kapitel wird dann eine Fallstudien aus dem Bereich ad-hoc Systeme (JINI) vorgestellt.

10.1 Ablauf des realisierten Entwurfs

Wie bereits in Kapitel 8 beschrieben, teilt sich der Entwurf – und hierbei wird in dieser Arbeit verstärkt der Entwurf der Architektur behandelt – in drei Ebenen auf:

- Spezifikation der invarianten Abläufe zwischen (wechselnden) Teilnehmern (**Dienstarchitektur**)
- Abbildung in Kombination mit Komponenteninstanzen und Umgebungsprofilen (Sandboxes) auf technische **Konfigurationen**.

- Auswahl einer Konfiguration und ihre Abbildung auf eine **Plattform** in Gestalt von Codeskeletten.

Diese drei Schritte werden durch die Modellierungsumgebung DANTE abgedeckt. Innerhalb des Werkzeuges kann die logische Architektur entweder durch SADL Spezifikationen textuell oder durch Editoren graphisch spezifiziert werden. Die so spezifizierten Akteure (Komponenten, Dienste, Sandboxes etc.) werden in einer Datenbank abgelegt und werden durch ein Repository transparent als Objekte bereitgestellt.

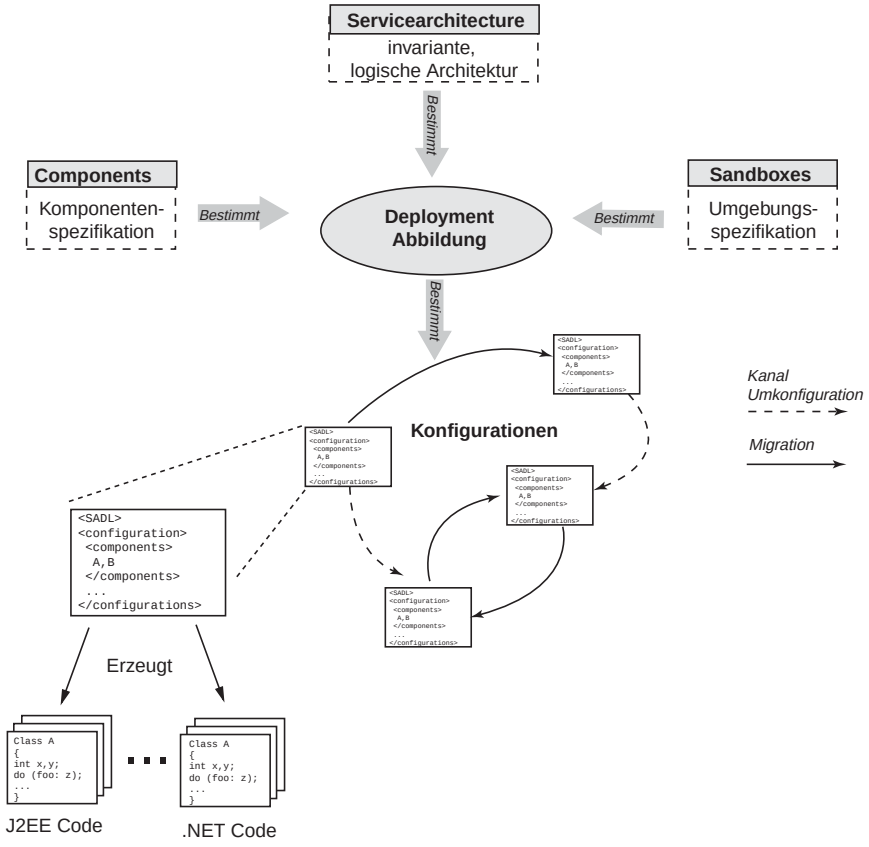


Abbildung 10.1: Zusammenhang zwischen der statischen, logischen Architektur und den einzelnen technischen Architekturausprägungen (Konfigurationen)

Das Werkzeug stellt auf der Basis eines so spezifizierten Projektes eine Reihe von Funktionen zur Verfügung:

- Die *Deployment-Abbildung* errechnet zu einer spezifizierten Dienstarchitektur und einer spezifizierten Einsatzumgebung die Menge der möglichen Konfigurationen und legt sie im Repository ab.
- Ein *Codegenerator* kann Konfigurationen aus dem Repository auf Codeskelette (analog zu einem IDL Compiler) der gewünschten Zielplattform (Java/JINI) abbilden.

- Eine *Transaktionsverwaltung* kann Dienste (bzw. Interfaces im Codeskelett) und deren direkt bzw. indirekt abhängige Methoden kapseln und dadurch Transaktionen bezüglich Umkonfigurationen absichern.

10.2 Aufbau der Modellierungsumgebung

Die Modellierungsumgebung setzt sich aus verschiedenen Clients zusammen, die sich um ein zentrales Repository gruppieren. Das Repository selbst verwaltet Spezifikationen innerhalb des Engineeringmodells und stellt damit eine Instanziierung des Engineeringmodells dar. Programme können dieses Repository nutzen, um Entitäten zu spezifizieren, diese zu Verwalten und bestimmte Konsistenzprüfungen auf Spezifikationen innerhalb des Repositories ausführen zu lassen. Das Repository selber bietet verschiedene Funktionen an, um strukturelle Eigenschaften sowohl logischer Architekturspezifikationen (Dienstarchitekturen), als auch technischer (Konfigurationen), zu überprüfen.

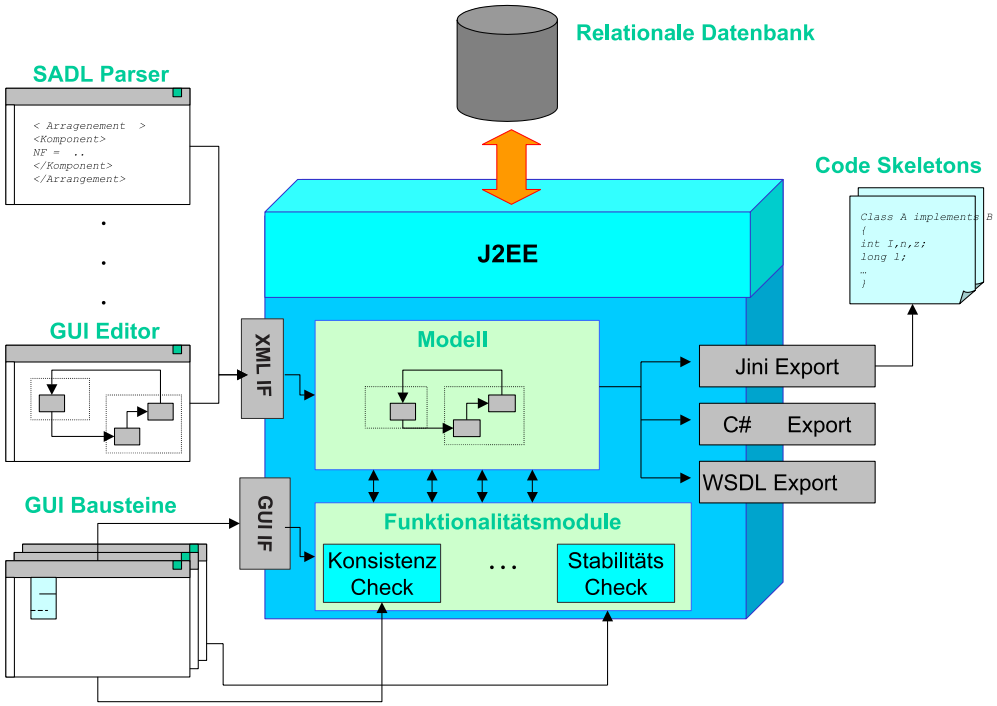


Abbildung 10.2: Grobübersicht der DANTE-Umgebung

Verschiedene Clients ermöglichen es, Spezifikationen in das Repository einzuspielen oder aus dem Repository heraus Code zu erzeugen.

Zum Zeitpunkt dieser Arbeit bietet die DANTE Umgebung folgende Clients:

SADL Client Der SADL Client ermöglicht es, SADL Spezifikationen textuell zu verfassen und diese auf Konsistenz zu überprüfen. Anschließend kann eine konsistente SADL Spezifikation in das Repository eingespielt werden.

Graphischen Client Ein graphischer Client ermöglicht eine dialogorientierte Spezifikation direkt auf dem Repository. Darüberhinaus kann eine Spezifikation als graphische Repräsentation gemäß der in dieser Arbeit verwendeten Notation ausgegeben werden. Ferner dient der graphische DANTE Client als Browser um in den verschiedenen Spezifikationen im Repository zu Browsen.

Java Codegenerator Der Codegenerator bildet Spezifikationen des Engineeringmodells aus dem Repository in Codeskelette ab, wie es in Abschnitt 9.3 beschrieben ist.

10.2.1 Repository

Das Repository stellt durch seine Instanziierung des Engineeringmodells den Mittelpunkt der Modellierungsumgebung dar. Dadurch ist es möglich, Spezifikationen gemäß dem Engineeringmodell abzulegen und maschinell zu verarbeiten. Ein entscheidender Faktor hierbei ist, wie das Engineeringmodell realisiert wird, also auf welche Weise Typen des Engineeringmodells auf Datentypen abgebildet werden, da dies die Attribute bestimmt, auf deren Basis Entscheidungen im System getroffen werden. Das

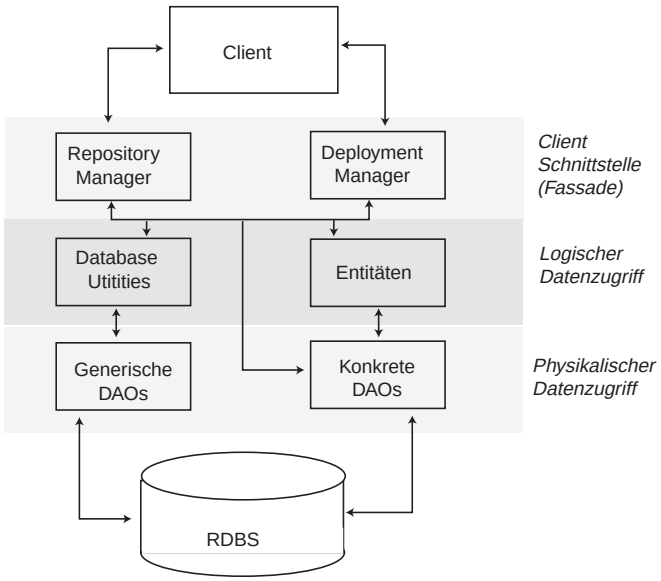


Abbildung 10.3: Die Zugriffsschichten des Repositories

Repository wurde als Serveraplikation mittels Java Enterprise Beans realisiert [Got01]. Dieser Ansatz bietet den Vorteil, dass die Skalierung bezüglich der Anzahl der Clients vom System abgenommen wird. Da die Modellierungsumgebung schon beim momentanen Stand bis zu vier Clients pro Nutzer umfassen kann, ist diese Eigenschaft von hohem Nutzen. Die Spezifikationsdaten werden letztendlich transparent in einer relationalen Datenbank im Hintergrund gehalten. Clients greifen nur indirekt über die *Repositorymanager* Komponente auf die Daten zu, die wiederum über eine Indirektionsebene vom physikalischen Datenzugriff getrennt ist (siehe Abbildung 10.3). Dies

ist notwendig, um die Zugriffslogik von den proprietären Zugriffsmechanismen der verschiedenen Datenbankprodukte zu kapseln.

10.2.2 Graphischer Client

Der graphische Client ist die Hauptnutzungsschnittstelle für das Repository. Er ist als Java-Client realisiert und bietet die folgenden Funktionalitäten:

- **Browsen** im Repository.
- **Dialogbasierte Spezifikation** von allen Entitäten des Engineeringmodells.
- **Graphische Ausgabe** von Spezifikationen.
- **Graphische Spezifikation** von State-Transition-Diagramms (STDs) für die Blackbox Spezifikation von Dienste.

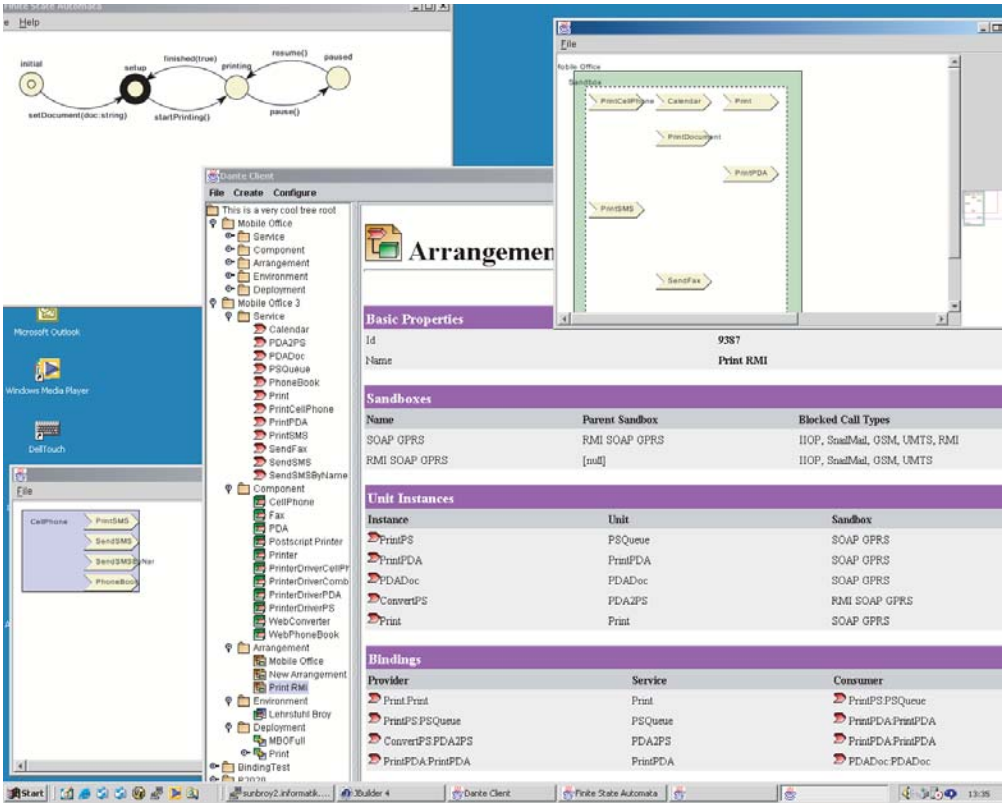


Abbildung 10.4: Der Client mit graphischer Ausgabe und STD Editor

Im Repository sind Spezifikationen als *Projekte* zusammengefasst, die einem Benutzer zugeordnet sind. Ein Projekt besteht aus einer Anzahl von Components, Dienste, Dienstarchitekturen, Einsatzumgebungen und Deployments. Diese bestehen wiederum gemäß ihrer Definition im Engineeringmodell aus verschiedenen Entitäten. Um eine

bequeme und einsichtige Nutzung und eine effiziente Nutzungsstruktur zu gewährleisten, bietet der graphische Client eine Browserfunktionalität in Form eines Projektbaumes, wie er aus vielerlei Anwendungen bekannt ist, an. Abbildung 10.4 zeigt diesen Ausschnitt des graphischen Clients.

Innerhalb des Projektbaumes kann eine Entität ausgewählt oder hinzugefügt werden und durch Dialoge spezifiziert werden.

Eine Entität kann darüberhinaus graphisch angezeigt werden. Hierbei wird die in dieser Arbeit verwendete graphische Notation benutzt, d.h. Dienste werden durch Pfeile, Komponenten durch Boxen und Sandboxes durch gestrichelte Rechtecke dargestellt. Im momentanen Stadium handelt es sich hierbei jedoch um eine graphische Ausgabe, das bedeutet, dass die graphische Repräsentation nicht aktiv zur Spezifikation genutzt werden kann (z.B. durch Verschieben einer Entität etc.).

Eine weitere Funktionalität im graphischen Client ist ein graphischer Editor zum Spezifizieren des Blackbox-Verhalten der Dienste mit Zustands-Übergangs-Diagrammen (engl. State Transition Diagramms, kurz STDs). Diese haben den Vorteil, dass sie bequem als Graph definiert werden können, wobei das System sie in Prädikatenform konform zum Systemmodell nach [BS00] umwandeln kann.

Abbildung 10.4 zeigt eine Beispielssitzung des graphischen Clients, wobei der Projektbaum, die Beschreibung einer Komponente, die graphische Ausgabe und der STD Editor zu sehen ist.

10.2.3 SADL Client

Der SADL Client ist ein .NET basierter Client, der es erlaubt, SADL Spezifikationen zu erstellen, editieren und in das Repository einzuspielen. Da XML basierte Spezifikationssprachen für den Nutzer eine ungewohnte oder auch umständliche Syntax verwenden, wurde insbesondere das Einfügen der jeweiligen Spezifikations-Gerüste automatisiert. Dadurch muss der Nutzer die genaue Syntax, insbesondere die große Menge an so genannten XML-Tags nicht kennen. Zur Lesbarkeit und besseren Erstellung von SADL- und damit XML-Dokumenten unterstützt der SADL Client die Erstellung der Spezifikationen im weiteren durch eine Vielzahl von Makros und graphischen Editierungsfunktionen.

SADL kann wie im Kapitel 9.1 beschrieben alle Spezifikationstypen, also auch die Ergebnisse wie Konfigurationen, auszudrücken. Daher ist es entscheidend, dass nur konsistente Dokumente in das Repository eingespielt werden.

Vor dem Aufspielen der Spezifikationen in das Repository werden die in XML nicht ausdrückbaren Konsistenzbedingungen des SADL Dokuments überprüft. Diese Validierung der SADL Spezifikation erfolgt in zwei Schritten:

Syntaktische Validierung: Hier wird durch den XML Parser die syntaktische Korrektheit der Spezifikation zu der SADL Definition in Form ihres XML Schemas überprüft. Im Schema sind die Struktur der Spezifikation und die Datentypen von SADL definiert. Die syntaktische Spezifikation überprüft also die Korrektheit der Elemente der Spezifikation, ihre Ordnung und Kardinalitäten. Es ist

nicht möglich, Referenzen auf deren Typkorrektheit zu prüfen, da alle Referenzen vom Typ `IDREF` sind, also der Wertebereich einer Referenz nicht einschränkbar ist.

Konsistenz Validierung: Die durch XML nicht ausdrückbaren Konsistenzbedingungen in SADL, wie zum Beispiel die oben erwähnte Überprüfung der Referenzen auf Typkorrektheit, werden zusätzlich nach dem Parsen des SADL Dokuments durch den Client vorgenommen.

Innerhalb der Konsistenz Validierung werden die folgenden Aspekte auf Konsistenz überprüft:

- *Referenzen* werden auf deren Existenz und Typkorrektheit geprüft, da XML nur einen generischen Zeigertyp besitzt, dessen Wertebereich nicht beschränkt werden kann.
- *Dienste* werden innerhalb von Komponenten insofern auf Konsistenz geprüft, ob beispielsweise eine Komponente einen Diensttyp als Needed-Dienst mehrfach benötigt.
- *Komponenten* werden auf Dienstkonsistenz (siehe oben) sowie auf strukturelle Konsistenz bezüglich ihrer Subkomponenten geprüft. Dies beinhaltet insbesondere, ob die Bindungen der Subkomponente “innerhalb” der Komponente liegen (siehe Definition des Engineeringmodells).
- *Dienstarchitekturen* werden daraufhin geprüft, ob Dienste bzw. Komponenten auf mehreren oder keiner Sandboxes angesiedelt sind. Auch die Bindungen werden ähnlich wie die Bindungen der Subkomponenten bei Komponenten auf Konsistenz und Erreichbarkeit geprüft.
- *Einsatzumgebungen* werden ebenfalls, analog zu den Dienstarchitekturen, auf Konsistenz geprüft.

Es ist leicht zu sehen, dass die Konsistenzprüfungen mit zunehmender Anzahl von Komponenten und Dienste exponentiell in ihrer Komplexität steigen.

10.2.4 Codegenerator

Der Codegenerator ist ein Bestandteil des Systems, der Spezifikationen aus dem Engineeringmodell heraus in Codeskelette einer Zielsprache umwandelt. Momentan wird hierbei die Sprache Java unterstützt und es werden aus Komponenten des Engineeringmodells Java-Objekte mit Jini-Unterstützung erzeugt.

Der Codegenerator erstellt allerdings nicht die eigentliche Funktionalität der Komponenten, sondern erzeugt deren Gerüst, ein so genanntes *Codeskelett*, wie es auch bei Interface Definition Languages (IDL) wie beispielsweise die CORBA oder RMI IDL

erfolgt. Das somit erzeugte Komponentenmodell ist in Abbildung 10.5 dargestellt. Eine Containerkomponente umschließt die eigentliche Komponente. Die Containerkomponente besitzt verschiedene Proxy-Objekte, die die Kommunikation nach aussen regeln. Nachrichten werden von den Proxies über die Verhaltens-Überwachung oder die Konfigurations-Management-Steuerung geschickt. Diese Komponenten regeln jeweils die Bedingungen für eine geplante Umkonfiguration – siehe Abschnitt 8.1.3 – oder prüfen ob das spezifizierte Blackbox-Verhalten eingehalten wird.

Ein Codeskelett für eine Komponente (in diesem Fall die *Printer* Komponente aus dem Beispiel 9.3.1) hat die folgende Form (die plattformspezifischen Methoden und Attribute, die Jini verlangt, wurden zur besseren Lesbarkeit hier weggelassen):

```
class Faxgeraet extends ... implements Print,Fax {
// This code is automatically generated by the
// DANTE Environment (c) TUM 2001

    // PRINT INTERFACE
    void setDocument(stream document) {
        if (STD_Print("setDocument")
            setDocument_Code(document);
        else
            generateError("setDocument(document)");
    }
    void setCopies(int copies){
        if (STD_Print("setCopies")
            setCopies_Code(copies);
        else
            generateError("setCopies(copies)");
    }
    bool startPrinting(){
        if (STD_Print("startPrinting")
            return startPrinting_Code();
        else
            {generateError("setCopies(copies)");
            return false;}
    ...

    // FAX INTERFACE

    ...

    // CODE TO IMPLEMENT
    void setDocument_Code(stream document) {
        // implement your code here ...   }

    void setCopies_Code(int copies){
    // implement your code here ...   }

    ...
}
```

Man beachte, dass die eigentliche Komponente durch eine so genannte Wrapper-Komponente gekapselt ist, die jeden Aufruf, der an die Komponente gerichtet wird, auf seine Korrektheit bezüglich des spezifizierten Blackbox-Verhalten überprüft. Hierzu wird aus der Verhaltensbeschreibung eines Dienstes, die im Repository als XML

Spezifikation eines STDs vorliegt, eine Methode generiert, die diesen Automaten realisiert. Da jeder Dienst auf ein Java Interface abgebildet wird (also auf eine Reihe von Methoden, die im Dienst durch dessen Messagetypes spezifiziert wurden) existiert für jedes Interface mit Namen *Name* eine Methode `STD_Name(String Method)`, die den Wert `true` zurückgibt, falls die Methode *Method* im Verhalten im momentanen Zustand erlaubt ist. Im positiven Fall wird die Methode tatsächlich aufgerufen, anderenfalls ein Fehler generiert.

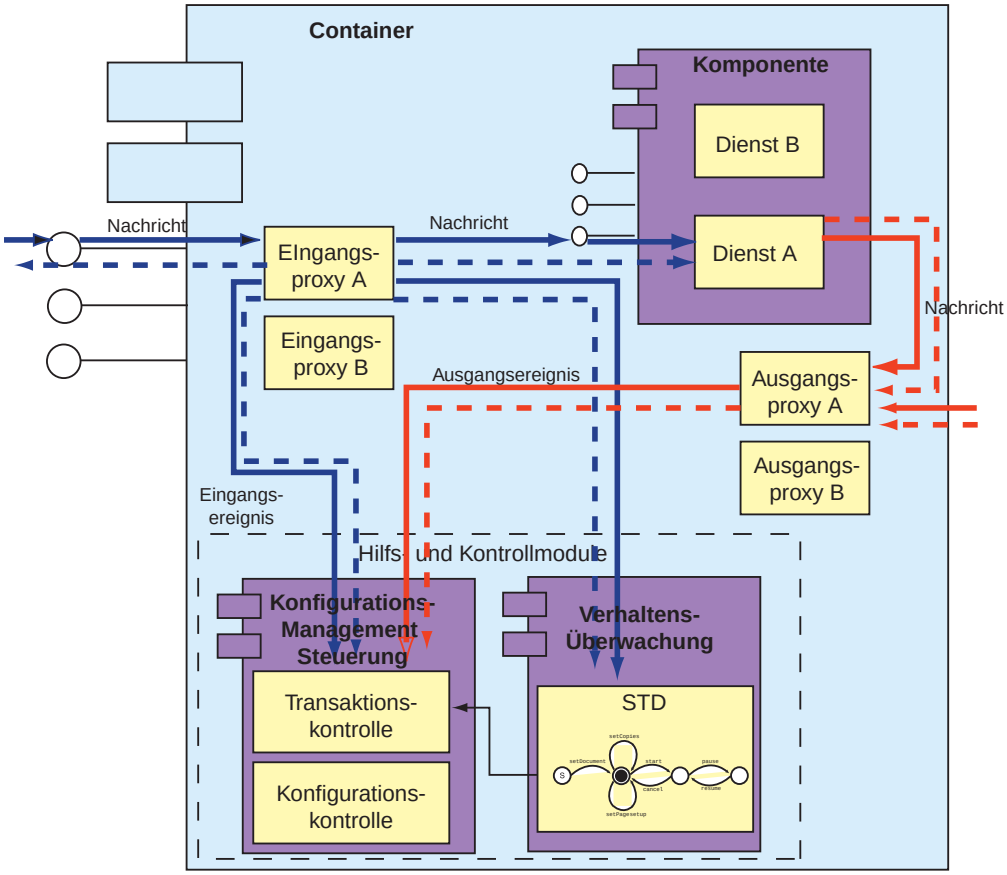


Abbildung 10.5: Aufbau des generierten Komponententyps: Objektskelett (Methodenrümpfe), das von einem Container umgeben ist, der die Aufrufreihenfolge gemäß der Blackbox-Spezifikation überwacht und Transaktionssicherheit regelt.

Eine solche Überprüfung auf korrekte Aufrufsequenzen ist für die spontane Interaktion von Komponenten von großem Nutzen.

- Eine ungültige Aufrufsequenz kann vor dem Aufruf **abgeblockt** werden, was die Stabilität des Systems erhöht.
- Der **Verursacher** einer ungültigen Aufrufsequenz kann automatisiert zur Laufzeit festgestellt werden. Dies hat insbesondere für das Billing, also die kommerzielle Abrechnung der Nutzung des Dienstes, Auswirkungen. Bei einem aus mehreren Komponenten zusammengesetzten Dienst, der nicht korrekt arbeitet,

kann die die Interaktionssequenzen verletzende Komponente zur Laufzeit festgestellt werden.

Man beachte, dass die hier realisierte Kontrolle der Aufrufsequenzen bezüglich des Blackbox-Verhaltens nur eine Möglichkeit darstellt, die aufgrund des relativ geringen Implementierungsaufwands gewählt wurde.

10.3 Funktionalitäten

Durch die Realisierung des Engineeringmodells in Form des Repositories und der Spezifikationswerkzeuge in Form des SADL und des graphischen Clients ist es möglich, die verschiedenen Tests und Funktionalitäten, die in dieser Arbeit beschrieben wurden, zu automatisieren. Exemplarisch werden nun zwei vorgestellt, die in der DANTE Umgebung realisiert wurden.

10.3.1 Deployment Abbildung

Die Deployment Abbildung wurde im Kapitel 6.2.2 als eine Funktion definiert, die eine logische Dienstarchitektur zusammen mit einer Einsatzumgebung, also einer Menge an Komponenten und Sandboxes, auf eine Menge von technischen Konfigurationen abbildet, die diese Dienstarchitektur realisieren und aufgrund der Sandboxes möglich sind. Diese Abbildung basiert letztendlich auf dem Begriff der Verhaltensäquivalenz, also welche Komponenten welche freien Dienste in der Dienstarchitektur ersetzen können, also welche Provided-Services einer Komponente verhaltensäquivalent zu einem freien Dienst in der Dienstarchitektur sind.

Verhaltensäquivalenz

Die Definition der Äquivalenz zweier Komponenten bzw. eines Dienstes und einer Komponente ist für das Deployment entscheidend. Es gibt eine Vielzahl verschiedener Möglichkeiten, die Äquivalenz zu definieren, wobei darauf hingewiesen sei, dass diese Definitionen keinen Anspruch darauf erheben, die *tatsächliche Verhaltensäquivalenz* zu definieren. Die tatsächliche Äquivalenz des Glassbox-Verhaltens ist meist zu aufwendig in der Prüfung, gerade wenn die Prüfung zur Laufzeit stattfinden soll. Zwei Möglichkeiten die Verhaltensäquivalenz praktikabel zu behandeln sind:

- Äquivalenz des Blackbox-Verhaltens kann, wenn diese als endliche Automaten definiert sind, durch Minimierung der Automaten und anschließenden Vergleich geprüft werden. Für jeden endlichen Automaten existiert ein minimierter endlicher Automat, der bis auf Isomorphie eindeutig ist. Für der Fall von standardisierten Alphabeten ist also der minimierte Automat eindeutig.

- Standardisierte Schnittstellen, denen aufgrund des syntaktischen Bezeichners ein implizites Verhalten zugeordnet wird. Dieses triviale Verfahren ist das gebräuchlichste. Beispiele sind DCOM, Jini etc. Hier werden von Standardsierungsgremien feste Schnittstellen mit fixem Verhalten definiert, die ausschließlich über den Bezeichner erkannt werden.

Zunächst benötigen wir einen Algorithmus, der zu einem bereinigten (d.h. nicht erreichbare Zustände sind bereits entfernt), deterministischen endlichen Automaten (siehe Abschnitt 9.1.4) den entsprechenden minimierten Automaten ausgibt, der dann bis auf Isomorphie Abbildungen der Zustands- und Eingabebezeichner eindeutig ist. Wir verwenden hierzu eine Abwandlung des Algorithmus von Hopcroft/Ullman (siehe [HU79]):

Algorithmus "Minimierung endlicher Automaten"

Eingabe: Sei $D = (E, Z, \delta, F, z_0)$ ein bereinigter, deterministischer, endlicher Automat.
Ausgabe: Menge M der Zustände, die zu verschmelzen sind um D_{min} zu erhalten.

Stelle Tabelle T aller Zustandspaare $p = \{z, z'\}$ mit $z \neq z'$ von M auf.
Markiere alle Paare $\{z, z'\}$ mit $z \in F$ und $\neg(z' \in F)$
Markiere alle Paare $\{z, z'\}$ mit $\neg(z \in F)$ und $z' \in F$
for all $p \in T$: $nichtMarkiert(p)$ **do**
    do loop
        for all $a \in E$ **do**
            if $markiert(\{\delta(z,a), \delta(z',a)\})$
              markiere {z,z'}
        od
    until keine Änderungen in T
od
for all $p \in T$ **do**
    if $\neg(markiert(p))$
      $M := M \cup p$
od

Wir definieren nun das Äquivalenzprädikat *equiv* zweier Dienste S_i und S_j bezüglich deren Blackbox-Verhalten, gegeben als zwei deterministische endliche Automaten D_i und D_j:

$$S_i \; equiv \; S_j \iff D_i^{min} \equiv D_j^{min};$$

wobei D^{min} jeweils die minimierten Automaten sind und $\equiv$ die Gleichheit modulo einer Isomorphie, also Umbenennung der Zustände und Aktionen sei.

Deployment-Algorithmus

Wir definieren zunächst einige Mengen und Operationen, die die Abarbeitung steuern. Hierbei handelt es sich um die Menge der Elemente der Dienstarchitektur sowie eine Ordnung auf derselben.

Definition 10.0: Elemente

Sei eine Dienstarchitektur *SA* gegeben, so definieren wir:

$$Elements_{SA} = Dienste.SA \cup Components.SA \cup Sandboxes.SA$$

$Elements_{SA}$ ist also die Menge aller Dienste, Komponenten und Sandboxes, die in der Dienstarchitektur auftraten.

Mit dem folgende Hilfskonstrukt definieren wir eine Ordnung auf den Elementen, die die Abarbeitung regelt.

Definition 10.1: Ordnung auf den Elementen

Wir definieren auf der Menge $Elements_{SA}$ die folgende Ordnung $\leq$, so dass gilt:

$$\forall\, b \in Sandboxes.SA, c \in Components.SA, s \in Dienste.SA : b \leq c \leq s$$

Dienste sind also oberste Elemente, gefolgt von Komponenten, die wiederum die Sandboxes nach sich ziehen.

Nun kann der Deployment-Algorithmus definiert werden, der eine Einsatzumgebung und eine Dienstarchitektur in Form einer Menge an Komponenten, Diensten und Sandboxes als Eingabe erhält. Die Ausgabe des Algorithmus ist eine Menge von Zuordnungen, die den jeweiligen Diensten eine entsprechende Komponente zuteilt.

Algorithmus "Deploymentabbildung"

Eingabe: Mengen von Komponenten, Dienste und Sandboxes $Elements_{SA}, Elements_{Env}$
Ausgabe: Menge von Zuordnungen

do loop
  $s_{min} := Min\{E'_{SA} \mid \leq\}$
  $E_{map} := \{e \mid e \in E'_{ENV} \wedge map(s_{min}) = e\}$
  for all $e \in E_{map}$ **do**
    $t_e = (E'_{SA} \backslash \{s_{min}\}, E'_{ENV} \backslash \{e\}, G \cup (s_{min}, e))$;
  od
until $E'_{SA} = \varnothing$

Jedes Tupel t_e enthält eine (partielle oder vollständige) Konfiguration G_e. Die Menge aller Konfigurationen **G** wird durch den Algorithmus rekursiv erzeugt.

Man beachte, dass das System bei dem hier vorgestellten Deploymentalgorithmus die Zuordnung von Dienste auf Dienste letztendlich immer noch primär via den Identifikator vornimmt. Der Identifikator muss daher eindeutig sein. Allerdings erkennt das System durch die Überprüfung der Äquivalenz der Dienste, die implizit in der *map* Abbildung vorgenommen wird, eine eventuelle falsche Zuordnung eines Dienstes, der zwar den Bezeichner eines Komponententyps *x* besitzt, jedoch nicht dessen korrekte Verhaltensspezifikation in Form des Automaten. Es ließe sich ebenfalls eine primäre

Auswahl bezüglich des Verhaltens realisieren, jedoch müssen dazu das Alphabet und die Zustandsbezeichner eindeutig sein.

Des Weiteren sei darauf hingewiesen, dass dies natürlich nur *eine* Form der Realisierung der Deploymentabbildung des Engineeringmodells ist. Da die Deploymentabbildung von der Zuordnung von Dienste auf Komponenten und damit von der Äquivalenzprüfung bzw. dem Äquivalenzbegriff von Diensten abhängig ist, ergeben sich eine Vielzahl von Realisierungsformen, die von der trivialen Bezeichnerzuordnung bis hin zum semantischen Vergleich führen können. Die hier vorgestellte Äquivalenzprüfung ist nicht zuletzt aufgrund ihres relativ geringen Realisierungsaufwandes gewählt worden, da auf viele bekannte Ergebnisse aus der Automatentheorie zurückgegriffen werden konnte. Darüberhinaus eignen sich Automaten als Spezifikationstechnik für graphische Systeme gut, da sie als Graph darstellbar sind. Es sei jedoch darauf hingewiesen, dass, falls das Repository um weitere Verhaltensspezifikationsformen erweitert würde, sich selbstverständlich auch andere Äquivalenztests realisieren ließen.

10.3.2 Transaktionsverwaltung

Um eine Umkonfiguration zur Laufzeit, wie in Abschnitt 8.1.3 beschrieben, zuzulassen, müssen die erzeugten Komponenten über ein entsprechendes Transaktionskonzept verfügen. Dies bedeutet, dass eine generierte Komponente über einen Transaktionsmechanismus verfügen muss, der bei Umkonfiguration eines Dienstes *S* einer Komponente *K* sämtliche notwendigen Konfigurationsbedingungen (siehe Abschnitt 8.1.3), wie beispielsweise den Ruhezustand der transitiven Hülle der abhängigen Dienste, transparent und automatisiert vornehmen muss, bevor eine kontrolliert ablaufende Umkonfiguration vorgenommen wird.

Eine Realisierung des Transaktionskonzeptes des hier vorgestellten Modells für Java/Jini Komponenten kann in [Sch01] gefunden werden.

10.4 Zusammenfassung

In diesem Kapitel wurde eine Realisierung der hier vorgestellten Techniken präsentiert, die den Entwurf und die Spezifikation spontaner Komponentensysteme durch Werkzeuge unterstützt. Hierbei wurde das Engineeringmodell in Form eines Repositories und verschiedener Editoren realisiert, die den Entwurf graphisch und textuell ermöglichen und das Deployment und die Codegenerierung weitgehend automatisieren. Hierbei wurde auch darauf hingewiesen, das Teile des abstrakten Engineeringmodells bei der Realisierung konkretisiert werden müssen und damit die Handhabung und den Nutzen des Modells beeinflussen. Beispiele sind hierfür der letztendliche Deployment-Algorithmus und der damit verbundene Verhaltens-Äquivalenz-Begriff oder die Transaktionsverwaltung. Die hier vorgestellte Realisierung kann daher nur ein Vorschlag von mehreren Möglichkeiten der Instanziierung des hier beschriebenen Modells sein, der auch keinerlei Anspruch auf das Optimum erhebt.

Es soll auch darauf hingewiesen werden, dass diese Anwendung des Modells und der damit verbundenen Techniken im Entwurf nur eine Möglichkeit der Anwendung

ist. Gerade spontane Interaktion, die automatisierter Auswertung von Systembeschreibungen zur autonomen Komposition von Systemteilen bedarf, bietet ein grosses Anwendungsfeld zur Auswertung der hier verwendeten Spezifikationen. Ein weiteres Beispiel, das nur kurz angerissen werden soll, ist die Verwendung von Architekturbeschreibungen bei der Auswahl von Systemkomponenten innerhalb des Trading-Dienste (oder Lookup-Service etc.), die bis dato nur die Beschreibung einzelner Komponenten, nicht aber die der Architektur, auswerten. Siehe hierzu auch Abschnitt 8.2.7.

KAPITEL

ELF

Fallstudie: Ad-Hoc Systeme

Im folgenden soll nun ein Beispiel vorgebracht werden, das eine Spezifikation eines spontanen Ad-Hoc Systems darstellt. Es sei darauf hingewiesen, dass die hier verwendete Spezifikationsform dem einen oder anderen Leser als lang und umständlich erscheinen mag. Dies resultiert zum einen aus der Komplexität spontaner Systeme, zu anderen jedoch aus der Tatsache, dass viele Systemteile isoliert spezifiziert werden und bei werkzeugunterstütztem Entwurf, der das Ziel des hier vorgebrachten Ansatzes ist, referenziert werden können. Da dies bei schriftlicher Form zu Unübersichtlichkeit führt, sei auf die in Kapitel 10 beschriebene Werkzeugunterstützung verwiesen.

Das hier vorgestellte Szenario entspricht dem im Abschnitt 4.3 beschriebenen Szenario, das ein Ausschnitt des hier behandelten ist. Es sei darauf hingewiesen, dass es sich hierbei natürlich um ein absichtlich einfach gehaltenes Beispiel handelt, da ein Beispiel von hoher Komplexität einen zu hohen Umfang hätte, um in schriftlicher Form einen anschaulichen Nutzen zu bringen. Hierzu sei der Leser auf die Werkzeugunterstützung in Kapitel 10 verwiesen. Auch kann nicht jede Entität vollständig spezifiziert werden, da dies zu einem zu hohen Umfang führen würde. Stattdessen werden nur einige Entitäten exemplarisch spezifiziert, wobei bei anderen analog verfahren werden würde.

Eine Reihe von mobilen Endgeräten sollen bei Zusammentreffen spontan interagieren. Es existiert eine drahtlose Netzumgebung, über die verschiedene Komponenten spontan interagieren und folgende Dienste bzw. Dienstszenarien realisieren sollen:

Print: Der Print-Service stellt einen Dienst dar, der Dokumente in einem bestimmten Format (Postscript) auf einem Gerät ausdrucken kann. Dieses Gerät kann ein bestimmter Drucker sein oder aber auch ein Kombi-Gerät, wie beispielsweise ein Fax-Print Gerät, wie es in dem Szenario vorkommt. Der Druckdienst verfügt über verschiedene Einstellungen, so zum Beispiel die Anzahl der Kopien, die gedruckt werden sollen, oder die Wahl zwischen Duplex- (beidseitigen) oder Simplex-Druck (einseitigen) etc. Solche Einstellungen müssen natürlich vor dem Starten des jeweiligen Druckauftrags vorgenommen werden, ganz wie man es von einem handelsüblichen Druckertreiber kennt.

Calendar: Der Kalenderdienst stellt die Funktionalität eines elektronischen Terminplaners zur Verfügung. Auf Anfrage eines Datums wird die Menge der für diesen Tag geplanten Termine zurückgegeben. Man kann solche Termine auch unter

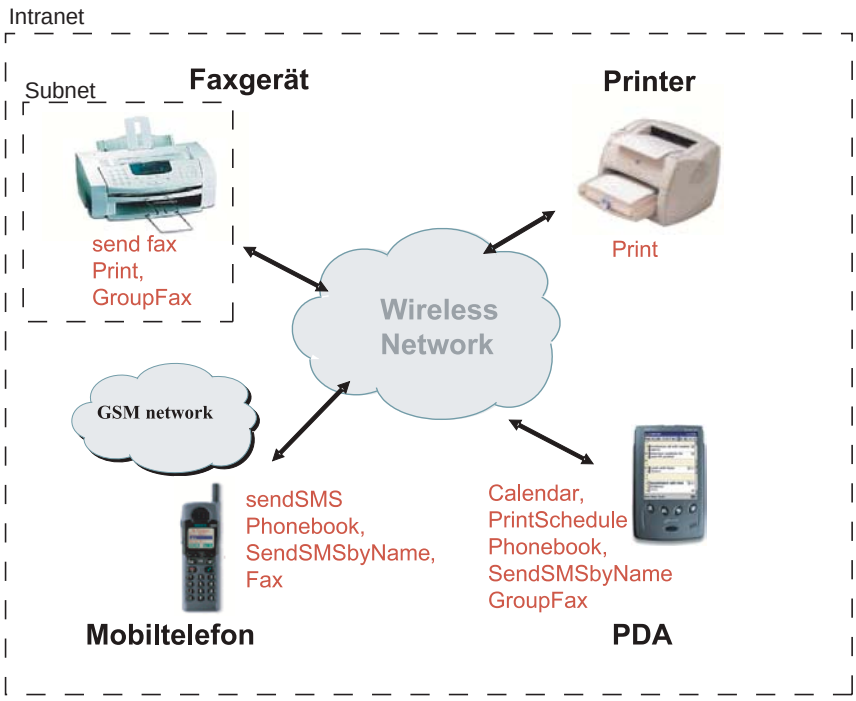


Abbildung 11.1: Anwendungsbeispiel: Spontan interagierende Geräte und ihre Dienste in einem Ad-Hoc Netzwerk

Angabe von Datum, Name und Dauer eintragen, wobei eine positive bzw. bei Überschneidung mit bereits existierenden Terminen, negative Bestätigung zurückgegeben wird. Um sich zu identifizieren, muss sich der Benutzer vor Nutzung des Terminkalenders erst mit einer Kennung anmelden.

Phonebook: Der Telefonbuchdienst bietet eine interne Datenbank von Namen und den dazugehörigen Telefonnummern an. Auf Namensanfrage wird die Telefonnummer zurückgeliefert. Ein Telefonbuchdienst wird von vielerlei Komponenten zur Verfügung gestellt. Beispiele hierfür sind Mobiltelefone, die eine interne Nummernverwaltung besitzen, PDAs, die ein elektronisches Adressbuch bieten etc.

SendSMS: Der Short-Messaging-Service (SMS) des GSM Standard für Mobiltelefone bietet einem Nutzer die Möglichkeit, mit dem Gerät eine asynchrone Kommunikation, ähnlich einer Email, zu nutzen. Der Nutzer kann eine kurze Textnachricht von ca. 120 Zeichen abfassen und diese an ein GSM-fähiges Endgerät, das durch eine Telefonnummer identifiziert ist, absenden.

SendFax: Dieser Dienst realisiert die Funktionalität, ein Dokument, das entweder in dem faxspezifischen Datenformat oder im ASCII Datenformat (Textdokument) vorliegt, an ein faxfähiges Endgerät zu senden. Das Endgerät wird über dessen Telefonnummer adressiert. Man beachte, dass dieser Dienst nicht nur von typischen Faxgeräten, sondern beispielsweise auch von GSM-basierten Mobiltelefonen realisiert wird.

SendSMSbyName: Der SendSMSbyName-Dienst ist eine komfortablere Version des SendSMS Dienstes, auf dem er aufbaut. Der SendSMS-Dienst sendet eine Nachricht an ein Endgerät, identifiziert über eine Telefonnummer. Der SendSMSbyName-Dienst hingegen ist eine Person-zu-Person Verbindung, die eine Nachricht an eine über ihren Namen identifizierte Person sendet. Der SendSMSbyName-Dienst benötigt einen SendSMS-Dienst sowie einen Phonebook-Dienst, der die Abbildung von Name auf Nummer erbringt.

GroupFax: Ein spezieller Gruppen-Faxdienst hat die Funktion, einer Gruppe von Teilnehmern ein gemeinsames Fax Dokument zu schicken und anschließend an die Adressaten eine SMS-Nachricht mit dem Sendebericht zu schicken (Erfolg, Misserfolg, kein Papier etc.). Als Eingabe erhält der Dienst die Namen der Teilnehmer sowie das zu sendende Dokument. Der Gruppen-Faxdienst versucht nun via einen Fax-Dienst das Dokument zu verschicken, und die Empfänger anschließend via einen SendSMSbyName-Dienst über Erfolg bzw. Misserfolg zu benachrichtigen.

PrintSchedule: bietet die Möglichkeit, für einen Kalender die Tagestermine eines Tages in Form eines Tagesplans auf einem Drucker auszudrucken. Dafür wird sowohl ein Druckdienst als auch ein Kalenderdienst benötigt. Nach erfolgreichem Druckvorgang wird eine Quittungsmeldung über den Erfolg bzw. Misserfolg zurückgegeben.

In dem zu spezifizierenden Szenario soll nun das Zusammenspiel dieser verschiedenen Dienste modelliert werden. Dazu werden zunächst die Dienste gemäß dem Engineeringmodell spezifiziert, anschließend wird das Zusammenspiel in Form der logischen Servicearchitecture definiert.

Eine Reihe von Komponenten, in diesem Beispiel mobile, drahtlos vernetzte Geräte, bilden eine Einsatzumgebung, mit der eine Reihe von technischen Konfigurationen realisiert werden kann. Anschließend werden die verschiedenen Konfigurationen und auch die logische Architektur bezüglich verschiedener Architektureigenschaften bewertet.

11.1 Spezifikation der logischen Architektur

Zuerst werden die notwendigen Bestandteile einer logischen Architektur spezifiziert. Anschliessend werden unter deren Zuhilfenahme die jeweiligen Servicearchitectures für einzelne Dienst-Szenarien definiert.

11.1.1 Services

Die Dienste werden in der in Kapitel 6 vorgestellten Schemanotation angegeben. Aufgrund des Umfangs der einzelnen Spezifikationen wird nur knapp auf die genauen Eigenschaften der Dienste eingegangen. Das Verhalten wird nur exemplarisch anhand zweier Beispiele spezifiziert.

Print

Der Print-Service bietet die Möglichkeit, Dokumente zu drucken. Vor dem Druckvorgang ist es notwendig, das zu druckende Dokument festzulegen und auf den Druckdienst zu überspielen, um Übertragungsfehler beim mechanischen Druckvorgang zu vermeiden. Anschließend können Druckparameter, wie Format, Qualität etc. festgelegt werden sowie die Anzahl der zu druckenden Kopien. Ist der Druckvorgang dann einmal gestartet, kann er entweder pausiert werden, abgebrochen, oder im pausierten Zustand wieder angefahren werden. Bei erfolgreichem Drucken wird eine Erfolgsmeldung, im Misserfolgsfall eine Fehlermeldung an den Auftraggeber gesendet.

Service :*Print*
Behavior : `Specification`
NeededServices `<>`
Interface : `Print_Interface`

Das dazugehörige Interface, welches die Signatur und die Kommunikationstypen (Messagetypes und dazugehörige Parametertyps) festlegt lautet wie folgt:

Interface :*Print_Interface*
InputMessageTypes : `<setDocument(String),setCopies(int),`
`setPagesetup(string),start(),cancel(),`
`pause(),resume()>`
OutputMessageTypes`<start(boolean)>`
Calltype : `RMI`

Das Blackboxverhalten des Druckdienstes wird durch das folgende State-Transition-Diagram ausgedrückt: Dieses Diagramm wird beispielsweise in der Werkzeugumge-

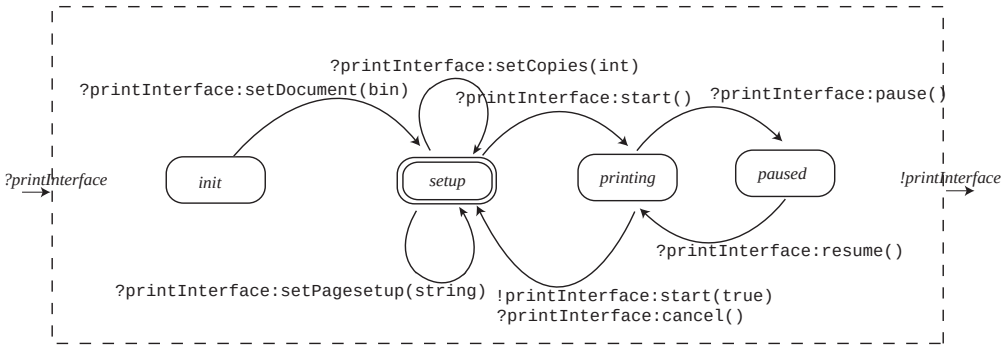


Abbildung 11.2: Das Blackboxverhalten des “Print” Services

bung in Prädikate umgewandelt und im Repository abgelegt. Bei der Codegenerierung wird diese Information dann verwendet,l um eine Wrapperkomponente zu generieren, die ungültige Aufrufsequenzen abfängt um somit Fehlinteraktionen bei den spontanen Komponenten zu erkennen und den Verursacher für Fehlverhalten des Systems zu bestimmen.

Calendar

Der Kalenderdienst ermöglicht – wie bereits erwähnt – das Eintragen und Abfragen der Tagesplanung. Es wird dazu kein weiterer Dienst benötigt und der Kalenderdienst besitzt somit ein Interface, bestehend aus einem Ein- und einem Ausgabe-Port.

Service :*Calendar*
Behavior : `Specification`
NeededServices <>
Interface : `Calendar_Interface`

Zunächst muss sich der Nutzer des Kalenderdienstes mittels einer Kennung und eines Passwords identifizieren. Anschliessend kann er zu einem beliebigen Datum einen Eintrag vermerken oder sich den Tagesplan zu einem beliebigen Tag ausgeben lassen. Eine Abmeldefunktion schliesst die Sitzung. Das zugehörige Interface ist einfach gehalten:

Interface :*Calendar_Interface*
InputMessageTypes :
`<login(name:string,password:string),`
`getDay(date:int),logout(),`
`setDate(date:int,name:string,time:int)`
OutputMessageTypes :
`<login(access:boolean),getDay(schedule:string),`
`setDate(success:boolean),logout(success:boolean)>`

Das Blackboxverhalten des Dienstes wird aus Platzgründen hier nicht aufgeführt, ist jedoch leicht nachvollziehbar.

Phonebook

Der Phonebookdienst stellt eine Möglichkeit dar, Daten für einen bestimmten Personenkreis zu verwalten. Er gibt nach Identifikation des Nutzers durch eine Anmeldung zu einem Namen die entsprechende Telefonnummer aus. Natürlich existiert auch eine Abmeldefunktion.

Service :*Phonebook*
Behavior : `Phonebook_Specification`
NeededServices <>
Interface : `Phonebook_Interface`

Die entsprechenden Messagetypes sind in der Spezifikation des Interfaces definiert. Es existieren drei Eingabe-Nachrichtentypen, die die Anmeldung, Abmeldung und die Anfrage nach einer Nummer zu einem Namen abwickeln. Als Ausgabenachrichten existiert die Antwort auf die Anmeldung mit positiver (Zugang) oder negativer Reaktion (Verweigerung) sowie die Rückgabe der Anfrage und die Quittung der Abmeldung.

Interface :*Phonebook_Interface*

InputMessageTypes :
`<login(name:string,password:string),`
`getNumber(Name:String),logout()`
OutputMessageTypes :
`<login(access:boolean),getNumber(number:int),`
`logout(success:boolean)>`

Ein solcher Dienst kann von vielerlei Komponenten realisiert werden. Beispiele hierfür sind Mobiltelefone, PDAs oder auch Webservices, die eine persönliche Umgebung im Internet anbieten, wie beispielsweise der .NET Webservice *myAdresses*.

SendSMS

Der SendSMS-Dienst bietet die Möglichkeit, eine aus einer Zeichenkette bestehende Kurznachricht an einen Empfänger über den GSM-basierten SMS-Dienst an ein GSM-Endgerät zu schicken. Das Empfangsgerät wird hierbei durch eine Telefonnummer identifiziert.

Service :*SendSMS*

Behavior : `Specification_PrintSMS`
NeededServices <>
Interface : `SendSMS_Interface`

Das Interface ist durch eine Menge von Ein- und Ausgabenachrichten definiert, wobei nur ein Nachrichtentyp existiert, der die Nummer und die Nachricht als Eingabe akzeptiert und mit einer booleschen Erfolgsmeldung quittiert.

Interface :*SendSMS_Interface*

InputMessageTypes :
`<sendSMS(Number:int,Message:String)`
OutputMessageTypes :
`<sendSMS(success:boolean)>`

Ein solcher Dienst wird beispielsweise von GSM-basierten Mobiltelefonen, aber auch von Web-Diensten angeboten.

SendSMSbyName

Bisher wurden nur atomare Dienste, also Dienste, die keine weiteren Dienste benötigten, vorgestellt. Nun wird ein Dienst definiert, der wiederum eine Reihe von weiteren Diensten impliziert.

Der SendSMSbyName-Dienst bietet eine komfortablere Möglichkeit SMS-Nachrichten zu versenden als der reine SendSMS-Dienst. Hier ist es nicht mehr

notwendig, die Nummer des Empfängers zu kennen. Die Nachrichten können an einen Namen geschickt werden. Der Dienst nutzt intern einen Phonebook-Dienst, wie er oben definiert wurde, und schlägt unter dem angegebenen Namen die dazugehörige Nummer nach. Anschliessend nutzt er diese Nummer, um einen SendSMS-Dienst, wie oben definiert, mit den Parametern Nummer und Nachricht aufzurufen. Man beachte, dass die konkrete Bindung, also die konkrete Komponente, die diese Dienste (SendSMS und Phonebook) erbringt, nicht festgelegt ist und zur Laufzeit wechseln kann.

Der Dienst SendSMSbyName benötigt also zwei Dienste: SendSMS und Phonebook. Daher besitzt er drei Interfaces, jeweils ein Interface zu den benötigten Diensten und ein Interface für sich selbst bzw. den Dienstnutzer.

Service :*SendSMSbyName*
Behavior : `Specification_SendSMSbyName`
NeededServices `<SendSMS,Phonebook>`
Interface : `SendSMSbyName_Interface,`
`SendSMS_Interface,Phonebook_Interface`

Die drei Interfaces werden wie folgt definiert. Das erste Interface ist das SendSMS Interface, wie es schon oben bei dem jeweiligen Dienst definiert wurde.

Interface :*SendSMS_Interface*
InputMessageTypes :
`<sendSMS(Number:int,Message:String)`
OutputMessageTypes :
`<sendSMS(success:boolean)>`

Das Phonebook-Interface ist ebenfalls bekannt, wir führen es der Vollständigkeit halber jedoch noch einmal auf:

Interface :*Phonebook_Interface*
InputMessageTypes :
`<login(name:string,password:string),`
`getNumber(Name:String),logout()`
OutputMessageTypes :
`<login(access:boolean),getNumber(number:int),`
`logout(success:boolean)>`

Das Interface des eigentlichen SendSMSbyName-Dienstes besitzt nun die folgende Form, wobei wie oben beschrieben der Dienst als Eingabenachricht eine Nachricht und einen Namen akzeptiert und als Ausgabe eine boolesche Quittungsmeldung liefert.

Interface :*SendSMSbyName$_I$nterface*
InputMessageTypes :
`<sendMessage(Message:String,Name:String)>`
OutputMessageTypes :
`<sendMessage(Success:boolean)>`

Es sei hier nochmals darauf hingewiesen, dass im Unterschied zu einer Komponentendefinition hier nicht festgelegt wurde, welcher Teilnehmer welchen Dienst erbringt, sondern nur eine Spezifikation, welche Teilnehmerrollen existieren werden und welche Nachrichten zwischen ihnen fliessen werden.

SendFax

Bei dem SendFax-Dienst handelt es sich wieder um einen atomaren Dienst, der von einer Komponente angeboten wird, die die Möglichkeit bietet, eine Faxnachricht an eine gegebene Telefonnummer zu senden.

Service :*SendFax*
Behavior : `Specification_SendFax`
NeededServices <>
Interface : `SendFax_Interface`

Das Interface ist ähnlich dem des SendSMS Dienstes, wobei anstelle der Textnachricht eine Fax-Nachricht im Binärformat übergeben wird. Als Erfolgs- bzw. Misserfolgsmeldung wird wieder ein boolescher Wert zurückgegeben.

Interface :*SendFax_Interface*
InputMessageTypes :
`<sendFax(Number:int,Fax:binary)`
OutputMessageTypes :
`<sendFax(success:boolean)>`

GroupFax

Schließlich spezifizieren wir einen weiteren Dienst, der von anderen Diensten abhängt. Der GroupFax-Dienst bietet die Möglichkeit an, eine Reihe von Adressaten eine einheitliche Fax-Nachricht zu senden (beispielsweise an die Büro-Nummer) und der Person eine Meldung über die erfolgreiche Fax-Sendung als SMS Nachricht zu verschicken.

Der Dienst benötigt also zwei Dienste, nämlich den Fax-Dienst, um Faxe zu verschicken sowie den SendSMSbyName-Dienst, um die SMS-Nachrichten zu senden. Daraus ergibt sich, dass er drei Interfaces besitzt, jeweils eins zu den Diensten und eins für den Dienstnutzer.

Service :*GroupFax*
Behavior : `Specification_GroupFax`
NeededServices `<Fax,SendSMSbyName>`
Interface : `SendSMSbyName_Interface,Fax_Interface,`
`GroupFax_Interface`

Die Interfaces zu den benötigten Diensten, *SendSMSbyName_Interface* und *Fax_Interface* sind bereits definiert und werden daher nicht mehr aufgeführt. Es sei

lediglich darauf hingewiesen, dass Interfaces, abhängig davon, ob sie den jeweiligen Dienst anbieten oder benötigen, gepolt sind, also welcher Port Ein- bzw. Ausgabe-Port ist. Aber dies ist nur eine Frage der Spezifikationstechnik und erspart zusätzliche Spezifikation. Das Interface des GroupFax-Dienstes selbst wird wie folgt definiert:

Interface :*GroupFax_Interface*

InputMessageTypes :

```
 <setDocument(Fax:binary),setNames(Names:String[]),
```

OutputMessageTypes :

```
 <finished()>
```

Das Interface bietet die Möglichkeit, eine Faxnachricht einzustellen sowie ein Array von Namen anzugeben. Dann sendet der Dienst die Nachricht an die Namen via einen SendFax-Dienstes, wobei jedes Fax bei dem Empfänger via SendSMSbyName-Dienst vermerkt wird. Anschliessend sendet der Dienst eine Meldung zurück, die angibt, dass alle Faxe und Nachrichten verschickt wurden.

Das Blackboxverhalten des Dienstes an seinen Schnittstellen wird durch ein STD angegeben. Hierbei werden die Eingabeports der Interfaces durch ein vorangestelltes Fragezeichen (?) und die Ausgabeports durch das entsprechende Ausrufungszeichen (!) vermerkt:

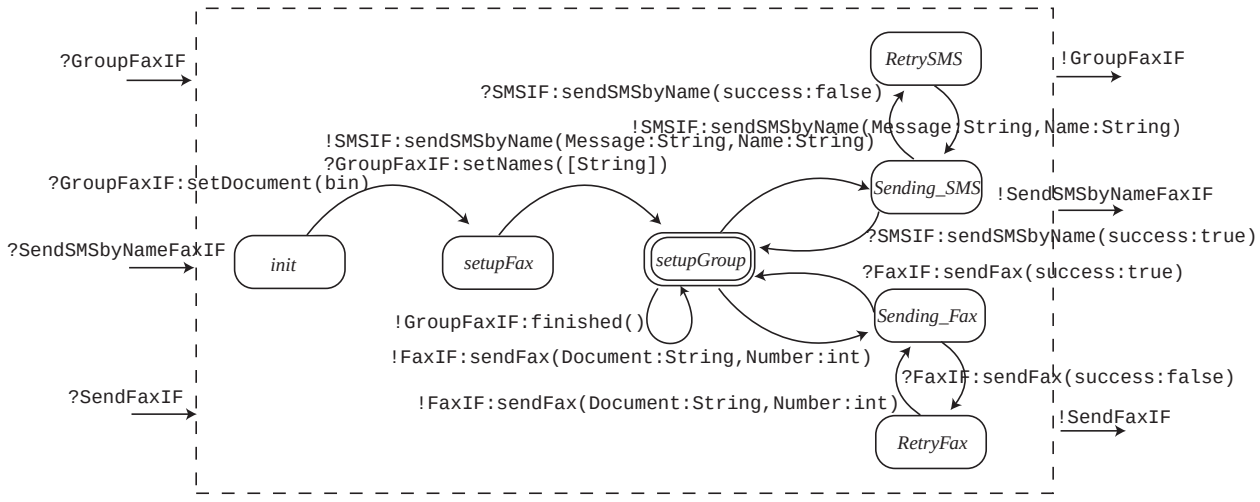


Abbildung 11.3: Das Blackboxverhalten des “GroupFax” Services

PrintSchedule

Der PrintSchedule-Dienst ist schließlich der letzte Dienst, der hier definiert wird. Er bietet die Möglichkeit, für einen Kalender die Tagestermine eines Tages in Form eines Tagesplans auf einem Drucker auszudrucken. Dafür wird sowohl ein Druckdienst als

auch ein Kalenderdienst benötigt. Nach erfolgreichem Druckvorgang wird eine Quittungsmeldung als Erfolgs- bzw. Misserfolgsmeldung zurückgegeben.

Service :*PrintSchedule*
Behavior : `Specification_PrintSchedule`
NeededServices `<Print,Calendar>`
Interface : `Print_Interface,Calendar_Interface, PrintSchedule_Interface`

Die beiden Interfaces zu den benötigten Diensten entsprechen den bereits definierten, zuzüglich der bei einem benötigten Dienst anfallenden Umpolung. Das Interface zum Dienstnutzer hat die folgende Form:

Interface :*PrintSchedule_Interface*
InputMessageTypes :
`<login(name:String,password:String), printSchedule(date:int)>`
OutputMessageTypes :
`<login(success:boolean), printSchedule(printer:string)>`

Nach dem Login und der Angabe des Datums gibt der Dienst den jeweiligen Drucker aus, der für den Druckdienst genutzt wurde.

Das Verhalten des Dienstes wird als gegeben vorausgesetzt.

11.1.2 Service Architectures

Auf Basis der jetzt definierten Dienste kann nun für ein Szenario die invariante Servicearchitecture definiert werden. Wir definieren hierbei die invariante funktionale Abhängigkeit und Erreichbarkeit der Dienste, indem die logischen Zusammenhänge der Dienste spezifiziert werden.

In dem Szenario sollen der Groupfax-Dienst und der PrintSchedule-Dienst jeweils unabhängig voneinander genutzt werden. Hierbei sei noch zu beachten, dass das Szenario in einer Umgebung des Faxgeräts angesiedelt sein soll, das also als Legacy-Komponente eingebunden wird, da es fest installiert ist. Es kann nur durch ein Gateway angesprochen werden, das lediglich über http angesprochen werden kann. Dieser Umstand wird durch die folgende Sandbox (SubnetFax) modelliert, die alle Verbindungen nach aussen, die nicht vom Kommunikationstyp (Calltype) “http” sind, blockiert. Man beachte, dass das Faxgerät selber durchaus andere Kommunikationstypen beherrscht, jedoch der Gateway sie blockiert.

Es existieren also zwei abstrakte Umgebungen, die durch die folgenden beiden Sandboxes spezifiziert werden, wobei die eine die Restriktionen des Fax-Subnetztes modelliert, die andere den freien Raum darstellt.

Sandbox :*SubnetFax*

Parent : `Ether`
Children : `<>`
HostedUnitInstances : `<FaxGeraet>`
BlockedBindingTypes : `<(*,*,{*}\{http},*)>`

Um den unrestriktierten freien Raum adressieren zu können wird die folgende Sandbox spezifiziert:

Sandbox :*Ether*

Parent : `<>`
Children : `<SubnetFax>`
HostedUnitInstances : `<SendSMS, Phonebook, Print Calendar, SendSMSbyName, PrintSchedule`
BlockedBindingTypes : `<>`

Betrachtet man nun die beiden Dienste, deren Dienst-Abhängigkeit, die Legacy-Komponente und das Umfeld, so kann die folgende Service-Architecture spezifiziert werden, die in Abbildung 11.4 grafisch dargestellt ist.

Servicearchitecture :*ServicearchitectureID*

Units : `<GroupFax,SMSbyName,SendSMS,Phonebook, FaxGeraet,PrintSchedule,Calendar,Print>`
Sandboxes : `<SubnetFax,Ether>`
Bindings : `<GroupFax_SMSbyName,GroupFax_FaxGeraet, SMSbyName_SendSMS,SMSbyName_Phonebook, PrintSchedule_Print,PrintSchedule_Calendar >`
Hostings : `<Faxgeraet_SubnetFax>`

Die einzelnen Binding- und Hostingrelationen sind nur aufgezählt, die einzelnen Spezifikationen werden aufgrund des Umfangs hier nicht aufgeführt. Um die Spezifikation zu vereinfachen, werden Units, für die kein explizites Hosting definiert ist, grundsätzlich auf der obersten Sandboxebene, also in diesem Fall der Ether-Sandbox, angesiedelt. Man beachte, dass die einzelnen Bindings der Dienste in der Servicearchitecture sich von der reinen Dienstabhängigkeit insofern unterscheiden, dass die Bindings auch die Art der Kommunikation ausdrücken können, also beispielsweise den Calltype. Dadurch wird die Menge der späteren Konfigurationen bereits eingeschränkt.

Nun gilt es, diese logische Architektur unter Zuhilfenahme einer Umgebung, also einer Menge an Komponenten und Umgebungen, auf die Menge der Konfigurationen abzubilden, die diese logische Architektur mittels dieser Komponenten und Umgebungen realisiert.

11.2 Einsatzumgebung

Eine Einsatzumgebung (engl. Environment) besteht aus einer Menge an spezifizierten Komponenten und Sandboxes. Wie anfangs erwähnt, besteht das hier beschriebene

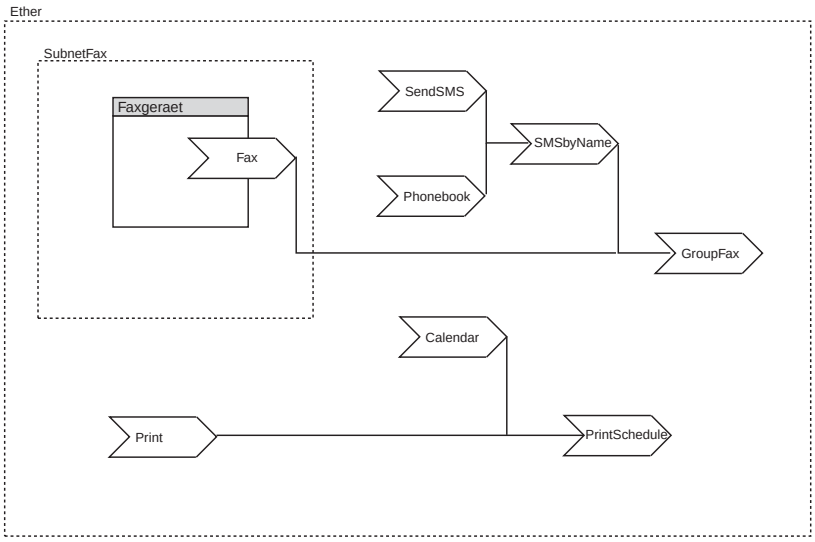


Abbildung 11.4: Die logische Servicearchitecture des Szenarios

Beispiel aus folgenden Komponenten:

- Ein **Mobiltelefon**, das einen integrierten Organizer mit Adressverwaltung besitzt sowie verschiedene Dienste, wie den Short Messaging Service (SMS) beherrscht.
- Ein **Personal Digital Assistant (PDA)** (z.B. Palm, PocketPC), dessen Hauptfunktionen ein bequemes Organizermodul mit integriertem Termin- und Adressverwalter sowie einem Emailclient.
- Ein **Faxgerät**, das sowohl Dokumente im Postscriptformat ausdrucken kann als auch Dokumente im Fax-Format senden oder drucken kann.
- Ein **Drucker**, der Dokumente im Postscriptformat drucken kann.

Außerdem ist das Faxgerät hinter einer Firewall in einem eigenen Subnetz angeschlossen, was den Zugriff restringiert.

Das Environment wird durch das folgende Schema definiert, wobei die Komponenten und Sandboxes anschliessend im einzelnen spezifiziert werden.

Environment :*ID*
Components : `<Mobiltelefon,Faxgeraet,Drucker,PDA>`
Sandboxes : `<Subnet_Fax>`

11.2.1 Komponenten

Im folgenden werden nun die einzelnen Komponenten spezifiziert. Aus Gründen des Umfangs und der Übersichtlichkeit werden nicht alle Komponenten vollständig spezifiziert, sondern nur Beispiele aufgeführt. Man beachte, dass die Komponentenspezifikation die *Struktur*, also den externen und Internen Aufbau, einer Komponente festlegt, nicht jedoch das Verhalten. Dieses wird bei der Spezifikation der Dienste (siehe oben) angegeben und bestimmt somit indirekt das Verhalten der Komponente.

Mobiltelefon Das Mobiltelefon ist eine der komplexeren Komponenten in diesem Szenario. Es besteht aus vier Subkomponenten und bietet vier Dienste nach aussen an, wobei ein Dienst benötigt wird. Die angebotenen Dienste sind im einzelnen ein Phonebook-Dienst, ein SendFax-Dienst, ein SendSMS-Dienst und ein SendSMSbyName-Dienst. Diese Dienste wurden bereits im vorhergehenden Abschnitt spezifiziert. Von aussen benötigt wird ein Phonebook-Dienst. Zwar wird dieser von der Komponente auch angeboten, jedoch kann er auch von einer Fremdkomponente erbracht werden und ist daher extra spezifiziert.

Intern besteht das Mobiltelefon aus vier Komponente: Einer Phonebook-Komponente, die den Phonebook-Dienst bereitstellt, einer FaxManager-Komponente, die den Fax-Dienst anbietet, einer SMSManager-Komponente, die die SMS-Funktionalität bietet sowie schließlich einer SMSbyNameUnit-Komponente, die den SendSMSbyName-Dienst unter Zuhilfenahme eines Phonebook- und eines SendSMS-Dienstes realisiert. Dieser Aufbau wird in der folgenden Abbildung 11.5 grafisch nach den Symbolen, die in Kapitel 6 definiert wurden und auch in der Entwicklungsumgebung aus Kapitel 10 verwendet wurden, dargestellt. Das entsprechende Schema wird hier nur auf

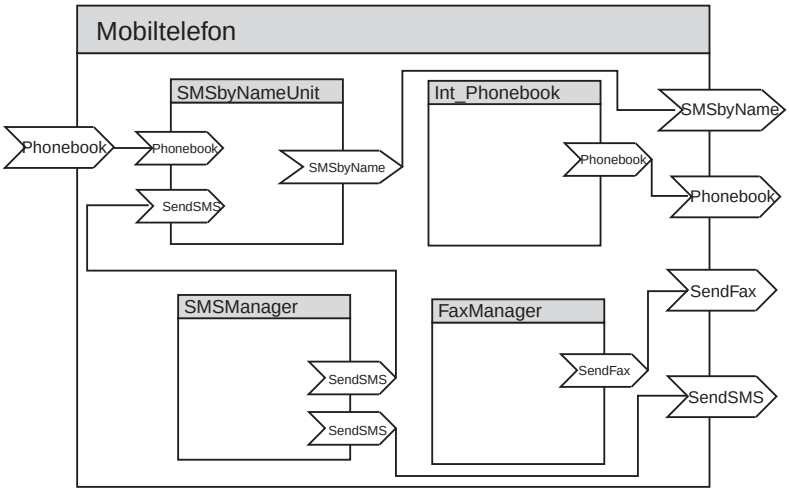


Abbildung 11.5: Die graphische Repräsentation des strukturellen Aufbaus der Mobiltelefonkomponente

der obersten Ebene angegeben, das heißt, die Unterkomponenten sowie die einzelnen Subunit-Bindings werden aufgrund des Umfangs nicht explizit spezifiziert.

Componenttype :*Mobiltelefon*

NeededServices <Phonebook>
ProvidedServices : <SendSMS,Phonebook,SMSbyName,
SendFax,Calendar>
SDependency : (SMSbyName; Phonebook)
Subunits :
<Int_Phonebook,SMSManager,SMSbyNameUnit,
FaxManager,CalendarManager>
Subunitsbindings :
<(Int_Phonebook,Mobiltelefon,Local,Phonebook),
(SMSManager,Mobiltelefon,Local,SendSMS),
(SMSManager,SMSByNameManager,Local,SendSMS),
(SMSbyNameManager,Mobiltelefon,Local,SMSbyName),
(Mobiltelefon,SMSByNameManager,Local,Phonebook)>
(FaxUnit,Mobiltelefon,Local,SendFax)>
(CalendarManager,Mobiltelefon,Local,Calendar)>

Man kann nun am Beispiel des Mobiltelefons erkennen, dass die Komponente autark ist, da der einzig benötigte Dienst, nämlich *Phonebook*, selbst angeboten wird. Des Weiteren sieht man an dem Beispiel, dass die Spezifikation der Komponente selbst reine Strukturinformation ist – Verhalten und funktionale Abhängigkeit sind in den jeweiligen Diensten spezifiziert.

Faxgerät Das Faxgerät bietet nach aussen einen Faxdienst an sowie einen Print-Dienst und einen Groupfax-Dienst. Hierfür benötigt sie von aussen einen SendSMSbyName-Dienst.

Intern besteht die Komponente aus drei Subkomponenten, die für jeweils einen der angebotenen Dienste verantwortlich sind. Den internen Aufbau und die internen Bindungen sind im folgenden Schema definiert.

Componenttype :*Faxgeraet*

NeededServices <SendSMSbyName>
ProvidedServices : <SendFax,Print,GroupFax>
SDependency : (GroupFax; SendSMSbyName)
Subunits :
<GroupManager,FaxManager,PrintManager>
Subunitsbindings :
<(FaxManager,Faxgerät,Local,SendFax),
(PrintManager,Faxgerät,Local,Print),
(GroupManager,Faxgerät,Local,GroupFax),
(Faxgerät,GroupManager,Local,SMSbyName),

Eine Spezifikation der einzelnen Bindungen wird aus Platzgründen hier nicht aufgeführt.

Drucker Der Drucker ist eine einfache Komponente, die lediglich einen Druck-Dienst anbietet, wobei keine Dienste benötigt werden. Der Drucker wird hier zur Vereinfachung als atomare, also keine Subkomponenten besitzende, Komponente spezifiziert.

Componenttype :*Printer*
NeededServices<>
ProvidedServices : <Print>
SDependency : <>
Subunits : <>
Subunitsbindings :
<>

PDA Ein PDA ist ein mobiler Kleinstcomputer, der in Form von Produkten wie Palm oder PocketPC Computern bekannt ist. Ein PDA bietet eine Vielzahl von Funktionalitäten an, jedoch benötigt er auch zwei Dienste. Der PDA ist wie alle anderen Komponenten via ein Netzwerk, in diesem Fall ein drahtloses, mit den anderen Komponenten verbunden.

Der PDA besteht wiederum aus einer Menge von Subkomponenten, die für bestimmte Dienste zuständig sind. Der Aufbau ist dem folgenden Schema zu entnehmen.

Componenttype :*PDA*
NeededServices <SendSMS,Print>
ProvidedServices :<Calendar,PrintSchedule,Phonebook,
SendSMSbyName,GroupFax>
SDependency : (SendSMSbyName;SendSMS),
(PrintSchedule,Print)
Subunits :
<Adressbook,CalendarManager,GroupScheduler>
Subunitsbindings :
<(CalendarManager,PDA,Local,Calendar),
(CalendarManager,PDA,Local,PrintSchedule),
(Adressbook,PDA,Local,SendSMSbyName),
(PDA,Adressbook,Local,SendSMS),
(PDA,CalendarManager,Local,Print),
(PDA,GroupScheduler,Local,SendFax),
(PDA,GroupScheduler,Local,SendSMSbyName),
(GroupScheduler,PDA,Local,GroupFax),

Die einzelnen Bindings sind wieder als Tupel angegeben, ohne dass aus Gründen des Umfangs die einzelnen Schemas definiert werden.

11.2.2 Sandboxes

Im Szenario treten zwei verschiedene Umgebungen auf. Es handelt sich um das Intranet, in dem sich das Szenario abspielt, und um das durch eine Firewall gesicherte

Subnet des Faxgerätes (siehe Abbildung 11.1). Beide Netzwerke werden als Umgebungsprofile, also als Sandboxes, modelliert.

Sandbox :*Intranet*
Parent : `Ether`
Children : `<Subnet_Fax>`
HostedUnitInstances : `<>`
BlockedBindingTypes : `<>`

Der Bezeichner *Ether* steht für die immer existierende oberste Umgebung.

Da die Firewall des Subnetzes keine Aufrufe vom Typ RMI (Java Remote Method Invocation) zulässt, blockiert diese Sandbox alle Bindingtypen, die den Calltype “RMI” besitzen:

Sandbox :*Subnet_Fax*
Parent : `Intranet`
Children : `<>`
HostedUnitInstances : `<FaxGeraet_Nr._4711>`
BlockedBindingTypes : `<(*,*,RMI,*)>`

11.3 Abbildung auf technische Architekturen

Es gilt nun zu bestimmen, welche Realisierungen des Szenarios mit den vorhandenen Komponenten möglich sind, und wie sich die Realisierungen bezüglich ihrer Architektur unterscheiden.

Es wird also die Service-Architecture in Verbindung mit dem definierten Environment auf eine Menge von technischen Konfigurationen abgebildet, deren Architektureigenschaften anschliessend bewertet werden. Die Menge der möglichen Konfigurationen umfasst für die beiden spezifizierten Szenarios zwölf Elemente.

11.3.1 Eigenschaften der logischen Architektur

Diese Dienstarchitektur hat folgende Abhängigkeitsstruktur: Es existieren somit zwei

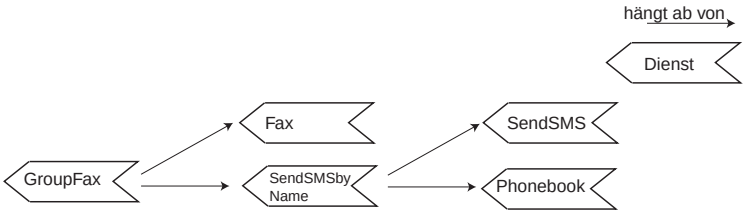


Abbildung 11.6: Abhängigkeitsstruktur des GroupFax Dienstes

Indirektionsstufen mit jeweils zwei Entitäten, d.h die Anzahl der Konfigurationen $\sharp K$ ist somit :

$$\begin{aligned} \sharp K_{GrpfFax} &= (\sharp K_{Fax}) \times (\sharp K_{SendSMSbyName}) \\ &= (\sharp K_{Fax}) \times (\sharp K_{Phonebook} \times \sharp K_{SendSMS}) \\ &= 2 \times 2 \times 1 = 4 \end{aligned}$$

Die Anzahl der Konfigurationen $\sharp K$ für die atomaren Dienste Fax, SendSMS und Phonebook entsprechen der Anzahl an Komponenten um jeweiligen Environment, die diesen Dienst als Provided-Service anbieten.

Gleiches gilt für den *Print Schedule*-Dienst. Er hängt von zwei atomaren Diensten, Calendar und Print, ab.

$$\begin{aligned} \sharp K_{PrintSchdl} &= (\sharp K_{Calendar}) \times (\sharp K_{Print}) \\ &= 1 \times 2 = 2 \end{aligned}$$

Die Anzahl der Konfigurationen des hier behandelten Szenarios ist nun das Produkt der beteiligten unabhaengigen Dienste:

$$\begin{aligned} \sharp K_{Szenario} &= (\sharp K_{PrintSchdl}) \times (\sharp K_{GrpFax}) \\ &= 4 \times 2 = 8 \end{aligned}$$

Man beachte hierbei, dass dies lediglich die Anzahl der Konfiguration bei stationären Komponenten ist, also bei Diensten, die an die Umgebung (Sandbox) fest gebunden sind. Wären die Dienste mobil, so würde sich die Anzahl der Konfigurationen noch entsprechend der möglichen Hosting-Beziehungen erhöhen.

Aufgrund des Umfangs greifen wir lediglich zwei der acht möglichen Konfigurationen heraus.

Configuration :*ConfigurationA*

```
Units : <Faxgeraet, PDA, Drucker, Mobiltelefon>
Sandboxes : <Intranet,SubetFax>
Bindings :
 <(Faxgeraet,PDA,SOAP,Fax),
 (Drucker,PDA,RMI,Print),
 (Mobiltelefon,PDA,RMI,SendSMSbyName),
 (Mobiltelefon,Mobiltelefon,Local,Phonebook)>
Hostings :
 <(Fax,SubnetFax),(PDA,Intranet),
 (Mobiltelefon,Intranet),(Drucker,Intranet)>
```

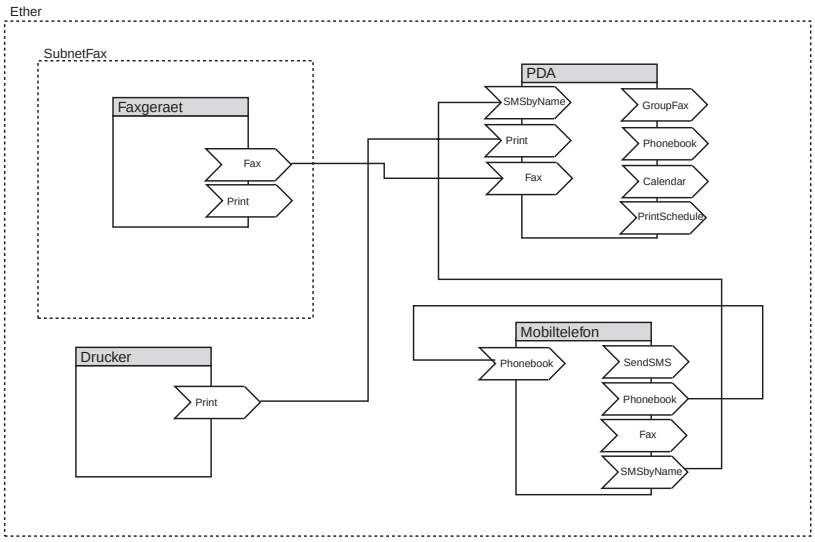


Abbildung 11.7: Konfiguration A der Servicearchitecture im der gegebenen Einsatzumgebung

Configuration :*ConfigurationB*

Units : `<Faxgeraet, PDA, Mobiltelefon>`
Sandboxes : `<Intranet,SubetFax>`
Bindings :

```
 <(Faxgeraet,PDA,SOAP,Print),
 (Faxgeraet,PDA,RMI,Fax),
 (Mobiltelefon,PDA,RMI,SendSMSbyName),
 (PDA,Mobiltelefon,RMI,Phonebook)>
```

Hostings :

```
 <(Fax,SubnetFax),(PDA,Intranet),
 (Mobiltelefon,Intranet)>
```

Betrachten wir zunächst die **Redundanz** verschiedener Dienste bezüglich der Einsatzumgebung: sowohl für den Fax-Dienst als auch für den Print-Dienst gilt, dass in der Konfiguration A zumindest ein überschüssiger angebotener Dienst existiert. Im Falle eines Ausfalls stände also immer noch ein redundanter Dienst zur Verfügung, der durch eine relativ einfache Umkonfiguration (ohne Migration) einbindbar ist. In Konfiguration B hingegen gilt dies nur für den Fax-Dienst, da der Print-Dienst lediglich von der Faxgerät-Komponente angeboten wird.

Die Konfiguration A unterscheidet sich von der Konfiguration B auch bezüglich der **Relevanz** der Faxgerät-Komponente: fällt diese Komponente aus, so sind in Konfiguration B zwei Dienste (Fax und Print) betroffen, in Konfiguration A lediglich der Fax-Dienst.

Die Konfiguration A ist **stabil** bezüglich Fax, Phonebook und Print-Dienst. Konfiguration B hingegen nur bezüglich Fax und Phonebook.

Der Leser möge beachten, dass es sich hier natürlich um ein relativ triviales Anschau-

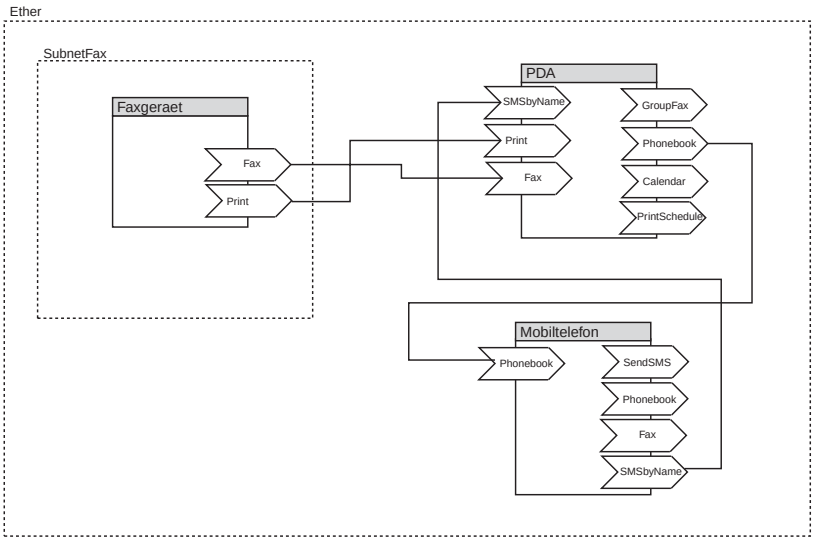


Abbildung 11.8: Konfiguration B der Servicearchitecture im der gegebenen Einsatzumgebung

ungsbeispiel handelt. Bei Betrachtung eines solchen Szenarios unter realistischen Bedingungen wäre eine Grössenordnung von mehreren Dutzenden von Komponenten und Diensten typisch. Dies würde zu Konfigurationen in Bereich von mehreren Hunderten führen, was dann nicht mehr so einsichtig wie in diesem Beispiel wäre.

11.4 Generierung von gekapselten Komponenten

Bis zu diesem Zeitpunkt wurde ausschliesslich mit abstrakten Entitäten des Engineeringmodells gearbeitet, die sich nicht auf eine bestimmte Plattform, wie Java/Jini oder .NET festlegen. Dies ist insbesondere wichtig, um auch heterogene Systeme modellieren zu können.

Da aber eine Abbildung auf gängige Plattformen existiert, soll hier kurz die Abbildung einer Komponente aus dem Beispiel auf die entsprechende Software-Komponente gezeigt werden.

Wie bereits erwähnt, werden Komponenten auf Jini-Klassen abgebildet, die angebotenen Dienste der Komponente resultieren in Interfaces, die die Klasse implementiert. Den benötigten Diensten einer Komponente entsprechen Assoziationen auf die entsprechenden Interfaces. Die Schnittstellen eines Dienstes werden auf die Schnittstellen der Interfaces übertragen, indem die Nachrichtentypen (Messagetypes) den Methoden des Interfaces entsprechen, und die Parametertypen (Parametertypes) den Parametern der Methoden.

Die **Verhaltensspezifikation** eines Dienstes wird zur Überwachung der jeweiligen Methoden des Interfaces eingesetzt. Dies bedeutet, dass beim Aufruf einer Methode erst deren Zulässigkeit gemäß der Verhaltensspezifikation geprüft wird (Methode

`STD_Print(Methodname string)`) und erst im positiven Falle die eigentliche Funktion aufgerufen wird (siehe hierzu auch Abschnitt 10.2.4). Dadurch kann im Falle eines nicht geplanten Aufrufs eines Dienstes der Verursacher ermittelt werden und eine eventuelle Fehlfunktion verhindert werden. Folgendes Codebeispiel zeigt das Skelett der Drucker-Komponente mit der Überwachung des Print-Interfaces. Zur besseren Übersicht wurden die JINI-spezifischen Hilfs-Methoden weggelassen.

```
class Drucker extends DANTE_Component implements Print {
// This code is automatically generated by the
// DANTE Environment (c) TUM 2001

    int PRINT_STATE = 0;

    // PRINT INTERFACE
    void setDocument(stream document) {
        if (STD_Print("setDocument"))
            setDocument_Code(document);
        else
            generateError("setDocument(document)");
    }
    void setCopies(int copies){
        if (STD_Print("setCopies")
            setCopies_Code(copies));
        else
            generateError("setCopies(copies)");
    }
    void setPagesetup(string pagesetup){
        if (STD_Print("setPagesetup")
            setPagesetup_Code(pagesetup));
        else
            generateError("setPagesetup(pagesetup)");
    }
    void start(){
        if (STD_Print("start"))
            start_Code();
        else
            generateError("start");
    }
    void pause(){
        if (STD_Print("pause"))
            pause_Code();
        else
            generateError("pause()");
    }
    void resume(){
        if (STD_Print("resume"))
            resume_Code();
        else
            generateError("resume()");
    }
    void cancel(){
        if (STD_Print("cancel"))
            cancel_Code();
        else
            generateError("cancel()");
    }
```

```
    // CODE TO IMPLEMENT
    void setDocument_Code(stream document) {
    // implement your code here ...   }

    void setCopies_Code(int copies){
    // implement your code here ...   }

    void setPagesetup_Code(string pagesetup){
    // implement your code here ...   }

     void start_Code(){
    // implement your code here ...   }

     void pause_Code(){
    // implement your code here ...   }

    void resume_Code(){
    // implement your code here ...   }

    void cancel_Code(){
    // implement your code here ...   }



    // IMPLEMENTATION STD OF SERVICE 'PRINT'
    boolean STD_Print(methodname string){

    case (methodname = "setDocument" && PRINT_STATE=0)
         PRINT_STATE=1;
         return true;
    case (methodname = "setCopies" && PRINT_STATE=1)
         PRINT_STATE=1;
         return true;
    case (methodname = "setPagesetup" && PRINT_STATE=1)
         PRINT_STATE=1;
         return true;
    case (methodname = "start" && PRINT_STATE=1)
         PRINT_STATE=2;
         return true;
    case (methodname = "pause" && PRINT_STATE=2)
         PRINT_STATE=3;
         return true;
    case (methodname = "resume" && PRINT_STATE=2)
         PRINT_STATE=2;
         return true;
    case (methodname = "cancel" && PRINT_STATE=2)
         PRINT_STATE=1;
         return true;
    else
        return false;
    }
}
```

11.5 Zusammenfassung

Das hier vorgestellte Beispiel verdeutlicht im möglichst einfacher Form die Vorteile des hier präsentierten Ansatzes:

- **Logische und technische Architektur** können getrennt spezifiziert und automatisiert in Beziehung gesetzt werden. Dadurch ist es möglich, die Funktionalität eines Systems unabhängig von dessen Struktur zu entwerfen.
- **Heterogene Netzwerke** und **Orte** können bereits im Entwurf explizit einbezogen werden.
- **Abstrakte Spezifikation** erlaubt, die Modellierung unabhängig von einer Plattform durchzuführen und damit **heterogene Systeme** zu modellieren.
- Die Spezifikation des **Blackboxverhaltens eines Dienstes** schlägt sich bis in den Code durch, indem die Spezifikation zur Überwachung der Dienstschnittstellen eingesetzt wird und so unerlaubte Aufrufe verhindert werden.

Teil V

Resümee

KAPITEL

ZWÖLF

Ergebnisse und Ausblick

In diesem letzten Kapitel werden die wesentlichen Ergebnisse der Arbeit noch einmal zusammengefasst und die Anwendungsmöglichkeiten des hier gewählten Ansatzes diskutiert. Schließlich werden die Fragestellungen und offenen Punkte, die aufgrund des begrenzten Umfangs einer solchen Arbeit bewusst ausgespart wurden sowie mögliche zukünftige Weiterentwicklungen und Ansätze angesprochen.

12.1 Zusammenfassung und Ergebnisse

Spontan interagierende Komponenten und Dienste entwickeln sich rapide zu einem gewichtigen Gebiet der Softwaretechnik. Neueste Entwicklungen, wie .NET [Cor01] oder ONE [Mic01] bekräftigen diese These. Allerdings aufgrund der hohen Dynamik und der starken Verflechtung solcher Systeme mit ihrer Netzwerkumgebung zusätzliche Techniken für den Entwurf und die systematische Entwicklung benötigt. Insbesondere ist es notwendig die autonome Komposition solcher Systeme, die zur Laufzeit stattfindet, zu steuern. Hierfür eignen sich Beschreibungen des Systems, die die benötigten Architekturinformationen ausdrücken und vom System automatisiert ausgewertet werden können. Diese Architekturbeschreibungen sollten begleitend zum Systementwurf mitentwickelt werden, um den Aufwand in Grenzen zu halten.

Dieser Arbeit war die Prämisse vorrausgestellt, Ergebnisse und Konstrukte aus verschiedenen Gebieten der Grundlagen- und Anwendungsforschung zu einem praktikablen Modell zu vereinigen, das es erlaubt, spontane Komponentensysteme in realistischem Umfang systematisch zu entwerfen. Die Ziele lassen sich grob in drei Blöcke unterteilen:

Grundlagen und Abstraktion: Es gilt eine Basisabstraktion, ein sogenanntes Basismodell, zu definieren, das in der Lage ist Verhalten, Struktur und die einzelnen Begriffe präzise zu fassen. Dieses Modell muss in der Lage sein die Eigenheiten der Anwendungsdomäne, namentlich *Dienste*, *Orte* und eine Unterscheidung zwischen *logischer* und *technischer Ebene*, implizit auszudrücken. Die Präzision geht jedoch im allgemeinen auf Kosten der Anschaulichkeit.

Ingenieurstechnische Modellbildung: Ein formales Basismodell abstrahiert die meisten Eigenschaften implizit, um sich auf eine möglichst geringe Anzahl an

Axiomen zu stützen. Dies ist notwendig und sinnvoll, um Vergleiche und Verifikationen effizient zu gestalten. Für die ingenieursmässige Anwendung ist es aber notwendig, möglichst sämtliche Charakteristika einer Anwendungsdomäne explizit im Modell zu isolieren. Charakteristika, wie Orte und logische Ebene sind im Basismodell nur implizit ausdrückbar. Es muss daher ein für die ingenieursmässige Anwendung geeignetes Modell entwickelt werden, das zwar auf dem formalen Basismodell aufbaut, jedoch die Charakteristika der Anwendungsdomäne explizit darstellt.

Praktische Realisierung unter realistischen Bedingungen: Um die Anwendbarkeit der hier vorgestellten Techniken unter Beweis zu stellen, muss eine Abbildung auf realistische Plattformen, wie beispielsweise Java und .NET, erfolgen. Die entwickelten Beschreibungstechniken sollten in einem allgemein anwendbaren und geeigneten Datenformat, beispielsweise eine Beschreibungssprache, realisiert werden, um einen automatisierten Austausch der Architekturbeschreibungen und deren automatisierte Auswertung zur Komposition zu ermöglichen. Eine prototypische Werkzeugumgebung, die es ermöglicht, aus einer auf dem Engineeringmodell aufbauenden Beschreibung Komponenten und deren Architekturbeschreibung zu generieren, rundet die Fallstudie ab.

Als wichtigste Ergebnisse in den einzelnen Gebieten sind hier zu nennen:

Grundlagen und Abstraktion

- Ein auf FOCUS basierendes **formales Basismodell**, das um einen **Dienstbegriff** erweitert wurde und mit dem es möglich ist die notwendigen Systemteile präzise auszudrücken. Insbesondere ist **Verhalten** mittels mathematischer Konstrukte beschreibbar, wobei auch grafische Techniken (STDs, MSCs) zur Spezifikation vorhanden sind.
- Durch Nutzung von **logischen** und **technischen Netzwerken** ist eine getrennte Spezifikation von logischer und technischer Ebene möglich.

Ingenieurstechnische Modellbildung

- Ein auf formalen Grundlagen basierendes **Engineeringmodell**, das es erlaubt, die für spontane Komponentensysteme charakteristischen Eigenschaften, die im Basismodell nur implizit vorhanden sind, explizit darzustellen.
- Eine **explizite Trennung zwischen logischer und technischer Architektur** ist auch hier möglich. Durch diese explizite Trennung ist es möglich die invarianten Strukturen spontaner Komponentensysteme unabhängig von den wechselnden Konfigurationen zu spezifizieren.
- Ein Konzept von **Lokationen und Umgebungsprofilen**, das es erlaubt Wide-Area-Anwendungen mit den damit verbundenen Eigenheiten wie Erreichbarkeit im Entwurf zu berücksichtigen.

- Eine Reihe von **Architektur-Qualitätskriterien**, die speziell für spontane Komponentensystemen ausschlaggebend sind und durch das Engineeringmodell **klar definierbar und quantifizierbar** sind.

Praktische Realisierung unter realistischen Bedingungen

- Eine auf XML basierende **Beschreibungssprache** (SADL), die sämtliche Entitäten des Engineeringmodells und die Qualitätskriterien ausdrücken kann und damit eine **automatisierte Auswertung** bei der spontanen Interaktion für die **autonome Komposition** der Komponenten möglich macht.
- Die **Abbildung** des Modells auf ein **Jini-basiertes Komponentenmodell**, das transaktionssicheres Umkonfigurieren und sichere Schnittstellen garantiert.
- Die **Realisierung des Modells** in Form einer Entwicklungsumgebung, die Spezifikation, Deployment und Codegenerierung ermöglicht.
- Die Illustration der **Vorteile von Verhaltensspezifikation** im Entwurf spontaner Komponentensysteme durch die Generierung von sicheren Schnittstellen.
- Die **Erprobung der vorgestellten Techniken** durch eine prototypische Realisierung, mittels gängiger Plattformen (Jini und .NET).

12.2 Anwendungsgebiete

In denen in Abschnitt 2.3 genannten Anwendungsdomänen, wie beispielsweise Mobile-Computing oder Internetsysteme, gibt es mehrere Gebiete, in denen ein strukturierter Einsatz der hier vorgestellten Techniken Sinn macht. Wir greifen exemplarisch drei Themen heraus, anhand derer wir den Nutzen der hier vorgestellten Techniken kurz skizzieren.

Systematische Entwicklung

Die hier vorgestellten Techniken, wie abstrakte Spezifikation, Deployment, logische und technische Sicht etc. bieten sich an für eine Systematische Entwicklung. Obgleich eine Methodik nicht im Mittelpunkt dieser Arbeit stand, können diese Techniken, in eine logische Anordnung eingegliedert, als wichtige Entwicklungsschritte für spontane Komponentensysteme dienen. Durch die Einbeziehung von Orten und von deren Umgebungsprofilen ist eine explizite Modellierung von Umgebungen und Orten möglich, die für Anwendungsbereiche wie Mobile Computing notwendig ist.

Die hier vorgestellten Techniken sind insbesondere geeignet um hoch-dynamische und spontane Systeme, beispielsweise mobile Dienste, im UMTS-Umfeld, zu modellieren. Hier wurden bereits erste Techniken, nämlich das hier vorgestellte Modell und die Definitionssprache SADL, eingesetzt [DFS01].

Ein weiteres Anwendungsfeld, das hohe Beachtung verdient, ist das der internetbasierten Web-Services, wie sie in den Plattformen .NET und ONE propagiert werden. Die Vorteile der hier vorgestellten Techniken gegenüber der momentan existierenden, wie beispielsweise WSDL (Web Service Description Language) sind leicht zu erkennen und wurden in Kapitel 9.2 dargestellt.

Heterogene Systemstruktur von Wide-Area-Anwendungen

Wide-Area-Netze, z.B. das Internet, sind heterogen. Daher werden verteilte Systeme auf diesen Netzen ebenfalls heterogen sein bezüglich Plattform, Implementierungssprache und anderer Merkmale. Eine Spezifikation solcher Systeme muss also hinreichend abstrakt sein, um alle heterogenen Merkmale einzubeziehen. Desweiteren sollten die heterogenen Netzwerkumgebungen, die durch Firewalls getrennt sind, im Modell abbildbar sein. Das hier vorgestellte Engineeringmodell ist sowohl mächtig genug, um Plattformen wie .NET oder Java zu abstrahieren, als auch in der Lage, Umgebungsprofile explizit in den Entwurf mit einzubeziehen.

Nutzung von Architekturkriterien zur spontanen Auswahl von Diensten

Ein weiterer Vorteil ist die Möglichkeit, nun auch *architekturelle* Kriterien neben den reinen Schnittstelleninformationen bei der spontanen Auswahl von Diensten verwenden zu können. War es bisher nur möglich, reine Schnittstellenbeschreibung als Auswahlkriterium bei einem Lookup bzw. Trading-Service anzugeben, beispielsweise “Komponente, die sowohl Schnittstelle A als auch Schnittstelle B implementiert”, so ist es mit dem hier vorgestellten Modell möglich, auch makroskopische Eigenschaften wie die Stabilität einer Architektur auszudrücken. Eine Anfrage nach einer Komponente, wie beispielsweise “Komponente, die sowohl Schnittstelle A als auch Schnittstelle B implementiert und sich auf die momentane Konfiguration möglichst stabilisierend auswirken soll” ist nun möglich.

12.3 Ausblick

Die Entwicklung spontaner, verteilter Systeme resultiert in Systemen, die von hoher Flexibilität geprägt sind und viele Freiheitsgrade aufweisen, und ist daher zwangsweise aufwendiger als die statischer Systeme. Ziel dieser Arbeit war es, einzelne Techniken zu entwickeln, die beim Entwurf solcher Systeme effizientere und bessere Ergebnisse liefern. Für eine komplette Entwicklungsmethodik müssen natürlich noch weitere Abschnitte aus dem Gebiet der Softwaretechnik aufgegriffen werden und gegebenenfalls für die Bedürfnisse spontaner Systeme ebenso angepasst werden, wie es hier für die Abschnitte Entwurf und Architektur getan wurde.

Aus den Ergebnissen dieser Arbeit ergeben sich eine Vielzahl von Anknüpfungspunkten für weitere Themen, die auf den hier präsentierten Ergebnissen aufbauen oder sie ergänzen. Wir gehen im folgenden kurz darauf ein.

Abbildung von Requirements auf Architekturanforderungen

Um einen vollständigen Entwicklungsprozess zu erhalten, ist es essentiell, die Anforderungen, seien sie funktional oder nicht funktional, in die Struktur des Entwurfs mit einfliessen zu lassen. Erste Ansätze sind hierzu in [DS99] skizziert.

Formale Abbildung der Umkonfiguration

In dem hier vorgestellten Modell stand die Formalisierung von Architektur im Vordergrund, um die funktionale Äquivalenz einer Menge von technischen Konfigurationen einer logischen Architektur zu zeigen. Ebenso kann eine Formalisierung der jeweiligen Umkonfiguration, also des Übergangs zwischen zwei funktional äquivalenten Konfigurationen, von Nutzen sein. Diese Umkonfigurationen, beschrieben in Abschnitt 8.1.3, bestehen aus dem Schalten bzw. Fallen lassen von Kanälen, sowie dem Hinzunehmen bzw. Fallen lassen von Komponenten. Diese Operationen lassen sich relativ einfach durch eine Erweiterung des Basismodells um die Eigenschaften von “dynamischem FOCUS” [GS96, HS96] einbeziehen. Dadurch wäre der Umkonfigurationsvorgang innerhalb des Basismodells beschreibbar.

Reflektionsbasierte Verhaltensäquivalenz

Wie bereits in Abschnitt 10.3.1 beschrieben, stellt der Begriff der Verhaltensäquivalenz einen zentralen Aspekt bei der spontanen Auswahl von Diensten und Komponenten dar. Die Realisierung dieses Begriffes ist momentan auf die implizite Äquivalenz standardisierter Schnittstellen, wie es beispielsweise bei COM oder JINI praktiziert wird, beschränkt. In dieser Arbeit wurde ein etwas weitergehender Ansatz vorgestellt, der einen minimierten Automaten als Verhaltensbeschriebung des Blackboxverhaltens des Dienstes nutzt. Obgleich eine solche Darstellung nicht vollständig ausreichend für eine eindeutigen Äquivalenzbegriff ist, ist die Darstellung zumindest modulo Umbenennung der Aktionen und Zustandsbezeicher eindeutig. Hier wäre ein formalisierter Äquivalenzbeggriff, der es erlaubt Verhaltensbeschriebungen tatsächlich auf Gleichheit zu testen, von hohem Nutzen.

LITERATURVERZEICHNIS

[AaDH00] Ken Arnold and James Gosling amd David Holmes. *The Java (tm) Programming Language, Third Edition*. Addison-Wesley Pub. Co., 2000.

[All97] Robert J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, 1997.

[BB90] Gérard Berry and Gérard Boudol. The chemical abstract machine. In *Proceeding of the 17th ACM Symposium on Principles of Programming Languages*, pages 81–94. ACM Press, 1990.

[BC97] Krishna Bharat and Luca Cardelli. Migratory applications. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 3–6. Springer-Verlag, Heidelberg, April 1997.

[BDD+92] Manfred Broy, Frank Dederich, Claus Dendorfer, Max Fuchs, Thomas Gritzner, and Rainer Weber. The design of distributed systems - an introduction to focus. Technical Report TUM-I9202, Technische Univerität München, 1992.

[BEK+00] Don Box, David Ehnebushe, Gobal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Sahish Thalle, and Dave Winer. Simple object access protocol (soap) 1.1. Technical report, W3C Consortium, May 2000.

[Box98] Don Box. *Essential COM – (The DevelopMentor Series)*. Addison-Wesley Pub. Co., 1998.

[BPSM97a] T. Bray, J. Paoli, and C.M. Sperberg-McQueen. Extensible markup language (xml). http://www.w3.org/TR/PR-xml-971208, 1997.

[BPSM97b] Tim Bray, Jean Paoli, and C.M. Sperberg-McQueen. The extensible markup language (xml). Technical report, W3C Consortium, 1997.

[Bro98a] Manfred Broy. Compositional refinement of interactive systems modelled by reations. In W.-P. de Roever, H. Langmaack, and A. Pnueli,

editors, *Compositionality: The Significant Difference*, number 1536 in LNCS, pages 130–149. Springer Verlag, 1998.

[Bro98b] Manfred Broy. Dynamics and mobility in hardware/software nets - towards a mathematical model. Not yet published, 1998.

[BS00] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems – Focus on Streams, Interfaces, and Refinement*. Monographs in Computer Science. Springer Verlag, 2000.

[Car97] Luca Cardelli. Mobile computations. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 3–6. Springer-Verlag, Heidelberg, April 1997.

[Car98] Luca Cardelli. Mobile ambients. In D. Le Métaye, editor, *Foundations of Software Science and Computational Structures*, number 1378 in LNCS. Springer Verlag, 1998.

[Car99a] L. Cardelli. Foundations for wide-area systems. In *Proceedings of the Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, 1999.

[Car99b] Luca Cardelli. Abstractions for mobile computation. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, number 1603 in LNCS. Springer Verlag, 1999.

[CCMW01a] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. Technical Report NOTE-wsdl-20010315, World Wide Web Consortium, 2001.

[CCMW01b] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1, 2001.

[CG99] Luca Cardelli and Andrew Gordon. Types for mobile ambients. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 79–92. ACM Press, 1999.

[CG00] Luca Cardelli and Andrew Gordon. Anytime, anywhere. modal logics for mobile ambients. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, pages 365–377. ACM Press, 2000.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. Technical report, Microsoft Research LTD., 98.

[CIP00] R. G. G. Cattell, Jim Inscore, and Enterprise Partners. *J2EE(tm) Technology in Practice: Building Business Applications with the Java(tm) 2 Platform, Enterprise Edition*. Addison-Wesley Pub. Co., 2000.

[Con00] The UPnP Consortium. Universal plug and play specification. Homepage: http://www.upnp.org, 2000.

[Con01] The UDDI Consortium. The uddi homepage. http://www.uddi.org, 2001.

[Cor] Microsoft Corp. Millennium homepage. http://research.microsoft.com/sn/Millenium.

[Cor01] Microsoft Corporation. The .net homepage. http://msdn.microsoft.com/NET/, 2001.

[CPV97] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Designing distributed applications with mobile code paradigms. In *Proceedings of ICSE 97, Boston MA*, 1997.

[DFS01] Markus Dillinger, Michael Fahrmair, and Chris Salzmann. An architecture for mobile middleware. Internal Report Siemens AG, 2001.

[DS99] Bernhard Deifel and Chris Salzmann. Requirements and conditions for dynamics in evolutionary software systems. In *Proceedings of the IWPSE'99 Intl. Workshop on Principles of Software Evolution*, 1999.

[DW98] Desmond Francis D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks With Uml : The Catalysis Approach*. Addison-Wesley, 1998.

[Edw99] W. Keith Edwards. *Core Jini*. Prentice Hall, 1999.

[FGH+99] Michael Frank, Martin Gitsels, Roland Haratsch, Chris Salzmann, and Maurice Schoenmakers. Jini prototype – functional and design specification. Technical Report A30148-J148-M60-1-69, Siemens ICN – internal Report, 1999.

[FGM+97] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – http / 1.1. http://www.ietf.org/rfc/rfc2068.txt, 1997.

[Fle94] Albert Fleischmann. *Distributed Systems – Software Design & Imlementation*. Springer Verlag, 1994.

[FMS01] Michael Fahrmair, Katherine Mickan, and Chris Salzmann. A software architecture for mobile middleware. Technical report, Siemes ICM – Internes Papier, 2001.

[Fog99] Karl Franz Fogel. *Open Source Development with CVS: Learn How to Work With Open Source Software*. The Coriolis Group, 1999.

[Fou98] Cédric Fournet. *The Join Calvulus: A Calculus for Distributed Mobile Programming*. PhD thesis, L'École Polytechnique, 1998.

[FPV98] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 1998.

[FS97] Martin Fowler and Kendall Scott. *UML Distilled*. Addison Wesley, New York, 1997.

[FSS00a] M. Fahrmair, C. Salzmann, and M. Schoenmakers. A reflection based tool for observing jini services. In *Reflection and Software Engineering*, number 1826 in LNCS. Springer Verlag, 2000.

[FSS00b] Michael Fahrmair, Chris Salzmann, and Maurice Schoenmakers. Verfahren zur vorauswahl mobiler dienste. Technical report, Deutsches Patentamt, 2000.

[G+94] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[GCL+99] Y. Goland, T. Cai, P. Leach, Y. Gu, and S. Albright. Simple service discovery protocol/1.0. http://www.ietf.org/internet-drafts/draft-ietf/draft-goland-fxpp-00.txt, 1999.

[Gel85] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7 1985.

[GJB96] D. Gupta, P. Jlote, and G. Barua. A formal framework for online software version change. *IEEE Transactions on Software Engineering*, 22(2), February 1996.

[GMW97] David Garlan, Robert T. Monroe, and David Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[Gon01] Li Gong. Project jxta: A technology overview. Technical report, Sun Microsoystems Inc., 2001.

[Got01] Heiko Gottschling. Design and implementation of a repository for spontaneously interacting component systems. Master's thesis, Fakultät für Informatik, 2001.

[GQ96] M. M. Gorlick and A. Quilici. Visual programming-in-the-large versus programming-in-the-small. In *Proceedings of the IEEE Symposium on Visual Languages*. IEEE Computer Society Press, 1996.

[GR91] M. M. Gorlick and R. R. Razouk. Using weaves or software construction and analysis. In *Proceedings of the 13th International Conference on Software Engineering*. IEEE Computer Society Press, 1991.

[Gro98] The Object Management Group. Common object request broker archictecture, 1998.

[GS96] R. Grosu and K. Stoelen. Specification of dynamic networks. In M. Haveraaen and O. Owe, editors, *Proceedings of the 8th Nordic Workshop on Programming Theory, Oslo, Norway*. University of Oslo, 1996.

[GSB97] R. Grosu, K. Stølen, and M. Broy. A denotational model for mobile point-to-point data-flow networks with channel sharing. Technical Report TUM-I 9724, Technische Univerität München, 1997.

[Hel96] Richard Helm. Patterns, architecture and software. In *ACM Sigplan PATTERNS*, pages 2–3, 1996.

[HMS99] Wiebe Hordijk, Sascha Molterer, and Chris Salzmann. On the reuse benefit of business objects. Technical Report TUM-I9934, Munich University of Technology, 1999.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HOE96] Dan Harkey, Robert Orfali, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. WILEY & SONS, 1996.

[HS96] Ursula Hinkel and Katharina Spies. Anleitung zur spezifikation von mobilen, dynamischen focus-netzen. Technical Report TUM-I9639, Munich University of Technology, 1996. German.

[HSSS96] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. Autofocus – a tool for distributed system specification. In *Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'96)*, number 1135 in LNCS. Springer Verlag, 1996.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley , Reading, MA, 1979.

[Inc] Java Soft Inc. The java language specification 1.0. SUN Microsystems Inc.

[Inc92] NeXT Software Inc. *Object Oriented Programming and the Objective-C Language*. NeXT Press, 1992.

[JW96] Daniel Jackson and Jeannette Wing. Lightweight formal methods. *IEEE Computer*, April 1996.

[KdRB91] Gregor Kiczales, Jim des Rivieres, and Danny Bobrow. *The Art of The Metaobject Protocol*. MIT Press, 1991.

[Kic92] Gregor Kiczales. Towards a new model of abstraction in the engineering of software. In *Proceedings of IMSA*, 1992.

[KM90] Jeff Kramer and Jeff Magee. The evolving philosophors problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11), November 1990.

[Kru00] Philippe Kruchten. *The Rational Unified Process – An Introduction*. Addison-Wesley, second edition, 2000.

[LS00] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*. ACM Press, 2000.

[Ltd98] Wireless Application Protocol Forum Ltd. Wml 1.1 dtd definition, 1998.

[Luc96] David C. Luckham. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. In *DIMACS Partial Order Methods Workshop IV*, 1996.

[Mas99] Cecilia Mascolo. Specification, analysis, and prototyping of mobile systems. In *Proceedings of ICSE 99, Los Angeles*, 1999.

[Mat00] Friedemann Mattern. Jini, java-card und rfids; bausteine für das ubiquitous computing. Presentation Slides – ETH Zürich, 2000.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of 5th European Software Engineering Conference (ESEC 95), Sitges, Spain, September 1995*, 1995.

[Med96] Neno Medvidovic. A classification and comparison framework for software architecture describtion languages. Technical Report UCI-ICS-97-02, University of California, Irvine, 1996.

[Mic01] SUN Microsystems. The sun open network environment (sun one). http://www.sun.com/software/sunone/wp-arch/, 2001.

[Mil89] Robin Milner. *Communication and concurrency*. Prentice Hall, 1989.

[Mil91] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1991.

[MJ81] S. Muchnik and N.D. Jones. *Program flow analysis*. Prentice Hall, 1981.

[MK96] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM Press, 1996.

[MN88] Pattie Maes and Daniele Nardi, editors. *Meta–Level Architecture and Reflection*. North Holland, 1988.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus for mobile processes. *Information and Computation*, 100:1–77, 1992.

[MS92] Max Mühlhäuser and Alexander Schill. *Software Engineering für verteilte Anwendungen: Mechanismen und Werkzeuge*. Springer Verlag, 1992.

[Mul94] Sape Mullender. *Distributed Systems*. Addison Wesley, second edition, 1994.

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of ICSE 20: Intl. Conf. on Software Engineering*, 1998.

[Par72] David Lorge Parnas. A technique for software modul specification with examples. *Communications of the ACM*, 15(5), 1972.

[Par94] David Lorge Parnas. Mathematical descriptions and specification of software. In *Proceedings of IFIP World Congress*, volume 1, pages 354–359, 1994.

[Par96] David Lorge Parnas. Mathematical methods: What we need and don't need. *IEEE Computer*, April 1996.

[PHL97] J. Peterson, P. Hudak, and G.S. Ling. Principled dynamic code improvement. Technical Report DCS/RR-1135, Yale University, 97.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. PhD thesis, Munich University of Technology, 1996.

[Sal99] Chris Salzmann. Managing shared business objects. In W. Emmerich, editor, *Proceedings of ICSE 21: Workshop on Engineering Distributed Objects*, 1999.

[Sal00] Chris Salzmann. Design principles for dynamic object systems. In *Proceedings of Intl. Workshop on Distributed Objects Programming Paradigms - ECOOP 2000*, 2000.

[Sch01] Florian Schönherr. Transaktionssicheres management von konfigurationen spontan interagierender komponentensysteme. Master's thesis, Technische Universität München, 2001.

[SG96] Mary Shaw and David Garlan. *Software Architecture – Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real–Time Object Oriented Modeling*. Wiley & Sons, 1994.

[SHM+00] Bill Shannon, Mark Hapner, Vlada Matena, James Davidson, Eduardo Pelegri-Llopart, Larry Cable, and Enterprise Team. *Java (tm) 2 Platform, Enterprise Edition: Platform and Component Specifications (The Java (tm) Series)*. Addison-Wesley Pub. Co., 2000.

[Smi84] Brian Smith. Reflection and semantics in lisp. In *Proceedings of Principles of Programming Languages POPL*, 1984.

[SS95] Bernhard Schätz and Katharina Spies. Formale syntax zur logischen kernsprache der focus-entwicklungsmethodik. Technical Report TUM-I9529, Technische Universität München, 1995.

[SS99] Chris Salzmann and Maurice Schoenmakers. Dynamics and mobility in software architecture. In Jan Bosch, editor, *Proceedings of NOSA 99 – Second Nordic Workshop on Software Architecture*, 1999.

[SS01] Chris Salzmann and Wolfgang Schwerin. Logical and technical abstractions for software architecture. Technical Report Working Paper, Technische Univerität München, 2001.

[Sto99] Ketil Stolen. Specification of dynamic reconfiguration in the context of input/output relations. In *Proceedings of the third International Conference on Formal Methods for Open Object-Based Distributed Systems FMOODS 99*, 1999.

[Sur00] Jacques Surveyer. The microsoft.net strategy: Risky, brilliant, or both? *Dr. Dobbs Journal*, August 2000.

[SW98] Peter Sewell and Pawel T. Wojciechowski. Location-independent communication for mobile agents: a two level architecture. In *Proceedings of WIPL '98*, LNCS. Springer Verlag, 1998.

[Szy97] Clemens A. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Pub Co, 1997.

[Tai98] Stefan Tai. A connector model for object-oriented component integration. In *Proc. ICSE'98 International Workshop on Component-Based Software Software Engineering (CBSE 98)*, 1998.

[Tic85] Walter F. Tichy. RCS: A system for version control. *Software Practice and Experience*, 15(7):637–654, 1985.

[vSHHT98] M. van Steen, F.J. Hauck, P. Homburg, and A.S. Tanenbaum. Locating objects in wide-area systems. *IEEE Communications Magazine*, 1998.

[vSTKS98] Maarten van Steen, Andrew S. Tannenbaum, Ihor Kuz, and Henk J. Sips. A scalable solution for advanced wide-area web services. Technical Report IR-446, Dep. of Computer Science, Vrije Universitait Amsterdam, March 1998.

[Wal] Jim Waldo. Jini architecture overview. `http://java.sun.com`.

[Wat99] Andrew Watson. What's new in corba 3.0, 1999.

[Wir71] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 4, April 1971.

[Wir90] Martin Wirsing. Algebraic specification. In In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Amsterdam, 1990. Elsevier.

[WK98] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language (OCL): Precise Modeling with UML*. Addison-Wesley – The Technology Series, 1998.

[Zel96] Gregory Zelesnik. The unicon language reference manual. Technical report, Carnegie Mellon University, 1996.

ABBILDUNGSVERZEICHNIS

1.1 Der Ablaufzyklus spontaner Systeme, wobei insbesondere die autonome Komposition und deren Kompositionsregeln in dieser Arbeit behandelt werden. 5

1.2 Zwei mögliche Konfigurationen eines spontanen Systems: Beziehungen zwischen Komponenteninstanzen. 6

1.3 Verteilung eines Systems über ein sehr großes Netzwerk (WAN) bringt unterschiedlich Umgebungen (Rechte, Autorisierung, Zeit etc.) in unterschiedlichen Netzsegmenten mit sich. 8

1.4 Klassifikation der Anwendungsarten 11

1.5 Problembeschreibung: Statische Funktionsbeschreibung muss auf eine dynamische Konfiguration abgebildet werden, diese wiederum auf eine Implementierung. 12

1.6 Ablauf und Ergebnisse der Entwicklung spontaner Komponentensysteme 14

2.1 Implizite Lokationen im π-Kalkül 20

2.2 Systemkonfiguration und Konfigurationsmanagement (aus [KM90]) . 24

2.3 Inkrementelles Konfigurationsmanagement (aus [KM90]) 24

2.4 DARWIN-Spezifikation einer Pipeline aus [MK96] 25

2.5 Die Struktur der in Abbildung 2.4 spezifizierten Pipeline 26

2.6 Eine Komponente (U_i) wird durch die Ein- (I) und Ausgangsströme (O) modelliert. 26

2.7 Eine IDL Spezifikation des Stacks aus Abbildung 2.8 32

2.8 Eine OCL Spezifikation des Stacks aus Abbildung 2.7 33

2.9 Entwicklung der Internets bzgl. der Anwendungsarten (aus [Mat00]) . 34

3.1 Schichtenaufbau von statischen Middleware-Plattformen 38

3.2 Grobübersicht der Phasen und einzelnen Workflows des Rational Unified Process 40
3.3 CORBA Kommunikationsstruktur 42
3.4 COM Kommunikationsstruktur 43
3.5 Schichtenaufbau von spontanen Middleware-Plattformen 45
3.6 Dynamisches Verhalten: Umschalten eines Kanals (Ein-/Ausgabe-Ports als schwarze bzw. weisse Kreise) 46
3.7 Mobilität in einem homogenen Netzwerk 46
3.8 Mobilität in einem heterogenen Netzwerk 47
3.9 Mobile Netzwerke 47
3.10 JINI Service Discovery Protokoll 51
3.11 Aufbau eines UPnP enabled Device 52
3.12 Webservices unter .NET 54

4.1 Abbildung von logischer Architektur auf technische Architektur bei statischen und spontanen Systemen 64
4.2 Statische funktionale Abhängigkeiten (A, B und C) werden durch Hinzunahme von Komponenten (W,X,Y und Z) auf drei Konfigurationen abgebildet. 66
4.3 Beispielszenario “Mobile Office” 69
4.4 Die zwei möglichen Konfigurationen des Dienstszenarios “SendSMSbyName” 71

5.1 Das Verhalten einer Komponente wird durch die Relation zwischen Ein- (I) und Ausgabeströme (O) modelliert. 78
5.2 Zerlegen einer Komponente in Dienste 80
5.3 Eine hierarchische Komponenteabstraktion und deren flaches Pendant im Basismodell (rechts) 82
5.4 Schema-Notation einer Spezifikation eines logischen Basismodell-Netzwerkes 84
5.5 Spezifikationsnotation technischer Basismodell-Netzwerke 85
5.6 Spezifikation von Basismodellkomponenten mit ausschließlich technischen Bestandteilen 88
5.7 Spezifikation von Basismodellkomponenten mit ausschließlich logischen Netzwerken 89
5.8 Spezifikation einer Basismodellkomponente mit sowohl logischen als auch technischen Netzwerken 90

6.1 Abstrakte Entitäten des Engineeringmodells werden auf konkrete Entitäten durch das Deployment abgebildet. 94
6.2 Das Blackbox-Verhalten des “Print” Dienste 100
6.3 Struktur der Mobiltelefon-Komponente aus Fallbeispiel 4.3 103
6.4 Verschachtelte Sandboxes 105
6.5 Eine Dienstarchitektur mit Komponenten, Diensten und Sandboxes . . 109
6.6 Ablauf der Entwicklung spontaner Komponentensysteme unter Einbeziehung der hier vorgestellten Techniken 111

7.1 Entitäten des Engineeringmodells und deren Abbildungen im Basismodell 116
7.2 Engineeringmodell-Komponente und eines ihrer flaches Basismodell Pendants (Ein/Ausgabeports jeweils weiß bzw. schwarz) 119
7.3 Prinzipieller Ablauf der Abbildung: Umwandlung einer Engineeringmodell-Spezifikation in eine Basismodell-Netzwerk-Spezifikation, die dann die jeweilige Menge an Netzwerken bestimmt. 121
7.4 Beispielkomponente “Printer” 124
7.5 Mehrfachbelegung eines Ports bei Komponente Y (*p* ist lokaler Portbezeichner, *t* der jeweilige Typ) 129
7.6 Die Abbildung einer Sandbox und deren Hostingrelationen auf Komponenten im Basismodell 131

8.1 Aufbau einer Architekturabstraktion: Abbildung der (statischen) Dienstebene auf eine (dynamische) Konfigurationsebene 138
8.2 Transformationen zwischen Konfigurationen 143
8.3 Die Breite von Dienst *A* (links) ist trotz der geringeren Anzahl der direkt benötigten Dienste größer als die des Dienstes *A'* (rechts). . . . 149
8.4 Flexibilität der Konfigurationen bei Abhängigkeiten auf logischer Dienstebene 149
8.5 Stabilität 151
8.6 Beispielkonfiguration (siehe Text) 152
8.7 Ein das Modell und die damit mögliche Architekturbeschreibung auswertender Trading-Service 154

9.1 Aufbau eines SADL Spezifikationsdokumentes 159
9.2 Aufbau einer SADL-Spezifikation einer Komponente 161
9.3 SADL Struktur einer Dienst-Spezifikation 162

9.4 Aufbau der SADL Spezifikation einer Sandbox 163
9.5 Aufbau einer SADL Dienst-Architecture Spezifikation 166
9.6 Struktur einen SADL Environment-Spezifikation 167
9.7 Der Automat des “Print” Dienste 167

10.1 Zusammenhang zwischen der statischen, logischen Architektur und den einzelnen technischen Architekturausprägungen (Konfigurationen) 178
10.2 Grobübersicht der DANTE-Umgebung 179
10.3 Die Zugriffsschichten des Repositories 180
10.4 Der Client mit graphischer Ausgabe und STD Editor 181
10.5 Aufbau des generierten Komponententyps: Objektskelett (Methodenrümpfe), das von einem Container umgeben ist, der die Aufrufreihenfolge gemäß der Blackbox-Spezifikation überwacht und Transaktionssicherheit regelt. 185

11.1 Anwendungsbeispiel: Spontan interagierende Geräte und ihre Dienste in einem Ad-Hoc Netzwerk 192
11.2 Das Blackboxverhalten des “Print” Services 194
11.3 Das Blackboxverhalten des “GroupFax” Services 199
11.4 Die logische Servicearchitecture des Szenarios 202
11.5 Die graphische Repräsentation des strukturellen Aufbaus der Mobiltelefonkomponente 203
11.6 Abhängigkeitsstruktur des GroupFax Dienstes 206
11.7 Konfiguration A der Servicearchitecture im der gegebenen Einsatzumgebung 208
11.8 Konfiguration B der Servicearchitecture im der gegebenen Einsatzumgebung 209

TABELLENVERZEICHNIS

5.1 Notation für die Bestandteile von Basismodellkomponenten und Netzwerken. 79

6.1 Die Merkmale spontaner Komponentensysteme und die Abbildung auf ihre Pendants im Engineeringmodell (siehe auch Tabelle 7.1 und 9.2). 96

7.1 Die Merkmale spontaner Komponentensysteme und die Abbildung auf ihre Pendants im Engineeringmodell sowie deren Repräsentation im formalen Basismodell. (siehe auch Tabelle 6.1 und 6.1) 133

9.1 Gegenüberstellung von WSDL und SADL 170

9.2 Die Merkmale spontaner Komponentensysteme und die Abbildung auf ihre Pendants im Engineeringmodell, deren Repräsentation im formalen Basismodell und die letztendliche Abbildung auf Java (bzw. Jini) (siehe auch Tabelle 6.1 174

Teil VI

Anhänge

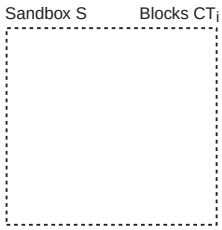
Zeichenerklärung

Basismodell

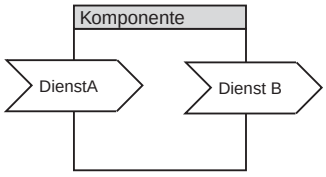
$\|\ \|$	Kardinalität einer Menge
$\mathcal{P}$	Potenzmengenoperator
M^*	Menge der endlichen Sequenzen über M
$x \downarrow i$	Sequenz der ersten i Sequenzen des Stromes $x \in M^\omega$
$x\|_{C'} \in \overline{C'}$	Einschränkung von $x \in \overline{C}$ auf die Kanäle von $C' \subseteq C$
M^ω	Menge der gezeiteten Ströme
!	Schreibrecht auf Kanal n
?	Leserecht auf Kanal n
$C(K),$ für $K \subseteq \mathbf{CH}$	Menge der Komponenten über K
$\overline{c_i} = c_{i1}, \ldots, c_{im}$	Menge von Eingangs-Kanälen (Hilfskonstrukt)
$\overline{c_o} = c_{o1}, \ldots, c_{on}$	Menge von Ausgangs-Kanälen (Hilfskonstrukt)
$\otimes$	Kompositionsoperator für Komponenten
$NW(K),$ für $K \subseteq \mathbf{CH}$	Menge der Komponentennetzwerke über K
$NW(\mathbf{CH})$	Menge aller Komponentennetzwerke
comp.nw	Menge der Komponenten in nw, für $nw \in NW(\mathbf{CH})$
in.nw	Menge der Eingangskanäle in nw, für $nw \in NW(\mathbf{CH})$
out.nw	Menge der Ausgangskanäle in nw, für $nw \in NW(\mathbf{CH})$
behav.nw	Verhalten des Netzwerks nw, für $nw \in NW(\mathbf{CH})$
$\otimes_{net}$	Netzwerk-Kompositions-Operator
$\overline{C} :\ \mathbf{CH} \rightarrow M^\omega$	Bündel von Strömen
$\overrightarrow{\mathbf{CH}}$	Kanalbelegung
$\mathbf{CH}$	Menge der getypten Kanäle
$\mathbf{M}$	Menge der Nachrichten
$\mathbf{T_M}$	Menge der Nachrichtentypen
$\mathbf{T_{CH}}$	Menge der Kanaltypen

$\mathbf{C}_E$	Menge aller Komponenten
$\mathbf{TC}_E$	Menge aller Komponententypen
$\mathbf{S}_E$	Menge aller Dienste
$\mathbf{TS}_E$	Menge aller Diensttypen
$\mathbf{INT}_E$	Menge aller Interfaces
$\mathbf{TINT}_E$	Menge aller Interfacetypen
$\mathbf{B}_E$	Menge aller Bindungen
$\mathbf{TB}_E$	Menge aller Bindungstypen
$\mathbf{H}_E$	Menge aller Hostings (Ansiedelungen)
$\mathbf{SBX}_E$	Menge aller Sandboxen
$\mathbf{CT}_E$	Menge aller Calltypes
$\mathbf{CONF}_E$	Menge aller Konfigurationen
$\mathbf{U}_E$	Menge aller Units
$\mathbf{SA}_E$	Menge aller Dienstarchitekturen (Servicearchitecture)
$\mathbf{MT}_E$	Menge aller Nachrichtentypen (Messagetypes)
$\mathbf{PT}_E$	Menge aller Parametertypen
$\overline{E} = (E_1, \ldots, E_n)$	Vektorisierung von Elementen (z.B. Diensten etc.)
$type(s) \in \mathbf{TS}_E$	Der Typ eines Dienstes s
$SU.C$	Subunits einer Komponente C
$SUB.C$	Subunit-Bindungen einer Komponente C
$D.C$	Abhängigkeitsfunktion einer Komponente C
$PS.C$	Menge der provided Services einer Komponete C
$NS.C$	Menge der needed Services einer Komponete C
$NS.S$	Menge der needed Services eines Dienstes S
$FS.SA$	Menge der freien Services einer Servicearchitecture SA
$BT.SBX$	Menge der blockierten Bindungstypen einer Sandbox *SBX*
$C.ENV$	Menge der Komponenten einer Einsatzumgebung ENV
$COMP.Conf$	Menge aller Komponenten der Konfiguration *Conf*

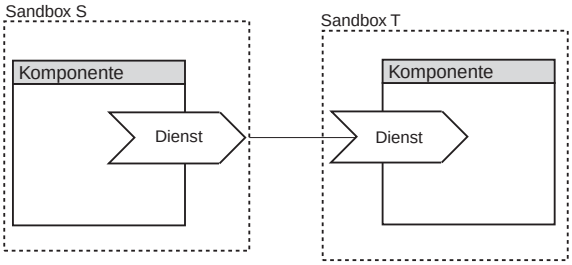
Graphische Notation



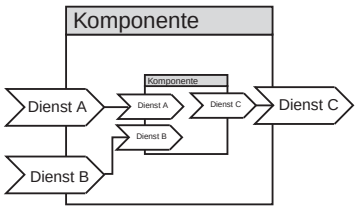
Sandbox S, die den Calltype CT_i blockiert



Komponente mit angebotenen Dienst B und benötigten Dienst A

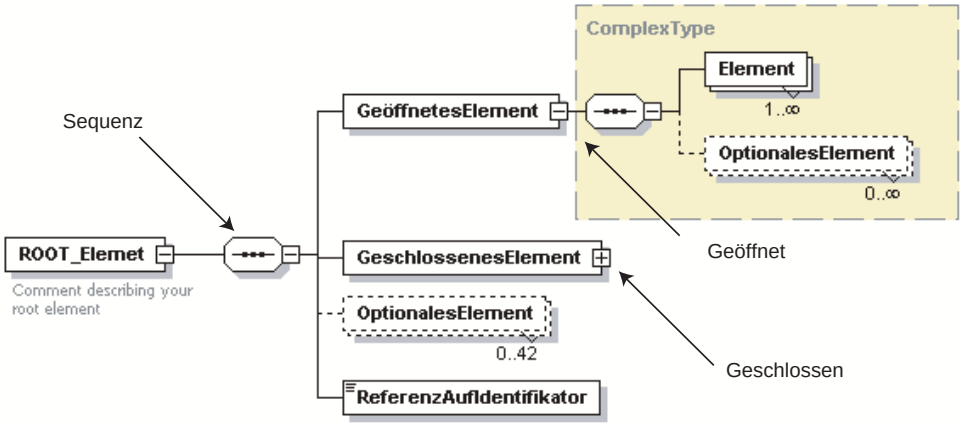


Bindung zwischen zwei Komponentenbezüglich eines Dienstes.



Komponente mit Subkomponente und den entsprechenden Subkomponenten-Bindungen.

Graphische XML Notation



Glossar

Ad-hoc System: ein Ad-hoc-System (auch dynamisches System) ist ein verteiltes →spontanes System, das seine Kommunikationsstrukturen (die Topologie) zur Laufzeit autonom verändert. Des Weiteren ist die Menge der Komponenten nicht fest, sondern es können Komponenten zur Laufzeit das System betreten bzw. Komponenten können das System verlassen. All dies soll vom System autonom gehandhabt werden, ohne den Benutzer in die →Umkonfiguration mit einzubeziehen.

Abstrakte Entitäten: abstrakte Entitäten sind Spezifikationskonstrukte des →Engineeringmodells, die durch die →Deploymentabbildung auf →konkrete Entitäten abgebildet werden. Daher besitzen abstrakte Entitäten kein eindeutiges Pendant in der →Konfiguration, sondern eine Menge an möglichen Pendants. Beispiele für abstrakte Entitäten sind →Dienste oder →Dienstarchitekturen.

Architektur: siehe →Softwarearchitektur.

Architektureigenschaft: eine Architektureigenschaft ist ein Merkmal einer Architektur, die im Engineeringmodell ausgedrückt und gegebenenfalls quantifiziert werden kann. Durch Architektureigenschaften ist es möglich Architekturen zu klassifizieren und gemäß dieser Klassen automatisiert auszuwählen.

Autarkie (Architektureigenschaft): die *Autarkie* einer Komponente gibt an, wie viele der für die angebotenen Dienste benötigten Dienst die Komponente selbst anbietet.

Basismodell: das Basismodell ist ein auf mathematischen Konzepten basiertes Systemmodell (FOCUS), das ein verteiltes System als Relationen zwischen Ein- und Ausgabeströmen modelliert. Das Basismodell stellt den semantischen Kern des Engineeringmodells dar. Engineeringmodellspezifikationen können auf eine mathematische Basismodellspezifikation abgebildet werden.

Bindung: eine Bindung (engl. Binding) ist die einzige Kommunikationsverbindung im Engineeringmodell. Zwei Komponenten oder →freie Dienste sind durch eine gerichtete Bindung bezüglich eines →Dienstes verbunden. Eine Bindung besteht aus einem Kanalpaar und einem →Calltype, der die Kommunikationsform festlegt. Bindungen können von →Sandboxes blockiert werden.

Blackbox-Sicht: eine Blackbox-Sicht drückt, im Gegensatz zur →Glassbox-Sicht, die Signatur aus, die ein Komponentennetzwerk besitzen müssen um eine Spezifikation eines →freien Dienstes des Engineeringmodells zu erfüllen.

Breite (Architektureigenschaft): die Breite eines Dienstes drückt die tatsächliche Menge an benötigten Diensten aus. Da die bloße Betrachtung der direkt benötigten Dienste (needed Services) nicht ausreicht steht die Breite eines Dienstes für die transitive Hülle der benötigten Dienste.

Calltype: ein Calltype ist ein Identifikator der Kommunikationsart einer Bindung. Beispiele für Calltypes in der Praxis könnten RMI, IIOP oder lokale Kommunikationsverbindungen sein.

Deployment Abbildung: die Deployment-Abbildung bildet eine logische Architektur zusammen mit eine Einsatzumgebung, also einer Menge an Komponenten und Sandboxes, ab auf eine Menge von technischen Architekturen (Konfigurationen), die die spezifizierte logische Architektur mittels der Komponenten und Sandboxes der Einsatzumgebung realisieren.

Dienst: ein Dienst ist eine abstrakte Entität des Engineeringmodells, die ein gewisses Blackbox-Verhalten bezüglich einer Schnittstelle (→Interface) definiert, ohne jegliche Struktureigenschaften (Glassbox) festzulegen. Dienste werden in zwei Zusammenhängen eingesetzt: Um Teilfunktionalität von Komponenten zu spezifizieren (benötigte und angebotene Dienste), sowie um in Dienstarchitekturen Unterspezifikationen in Form von →freien Dienste zu ermöglichen.

Dienstarchitektur: die Dienstarchitektur ist die Repräsentation der →logischen Architektur im →Engineeringmodell.

Dynamisches System: siehe →Ad-hoc System.

Dynamisches Verhalten: eines verteilten Systems bezeichnet den Umstand, dass das System sowohl die Menge der Komponenten, als auch die Kommunikationsbeziehungen zwischen diesen, zur Laufzeit verändern kann.

Einsatzumgebung: eine Menge von Umgebungs- und Komponentenspezifikationen (Komponenten und Sandboxes) des Engineeringmodells, die eine →Dienstarchitektur realisieren sollen. Wird zusammen mit einer →Dienstarchitektur durch die →Deployment-Abbildung auf eine Menge von →Konfigurationen abgebildet.

Engineeringmodell: das Engineeringmodell ist ein auf dem formalen Basismodell aufbauendes Systemmodell, das die speziellen Charakteristika spontaner Komponentensysteme (→Dienste, gesonderte →logische und →technische Architektur, →Sandboxes, →Deployment, →spontane Interaktion etc.) explizit behandelt und näher an den praktischen Abstraktionen des Software-Entwurfs angelehnt ist.

Firewall: eine Firewall ist ein →Gateway, der nur bestimmte Nachrichtentypen passieren lässt. Die zu blockierenden Nachritentypen können beispielsweise bezüglich der Herkunftsadresse, des Ports auf dem sie verschickt werden, oder

des Inhaltstyps (sog. *Packagefiltering*) definiert werden. Im Engineeringmodell werden Firewalls durch →Sandboxes modelliert, die die jeweiligen Bindungstypen, also beispielsweise alle Bindungen in ein anderes Netzwerk (Sandbox), blockieren.

Flexibilität (Architektureigenschaft): die *Flexibilität* eines →Dienstes in einer →Dienstarchitektur stellt den Abhängigkeitsbaum der benötigten Dienste dar. Bezüglich einer →Einsatzumgebung kann aus der Flexibilität die Anzahl der Konfigurationen bestimmt werden: je höher die Flexibilität, desto Höher die Anzahl der möglichen Konfigurationen.

Freie Dienste: sind →abstrakte Entitäten des →Engineeringmodells und treten nur in →Dienstarchitekturen auf. Sie stellen angebotene →Dienste dar, die nicht in eine Komponente eingebunden sind. Sie werden durch die Deployment-Abbildung durch Komponenten der →Einsatzumgebung ersetzt.

Gateway: ein Gateway ist ein Übergang zwischen zwei Netzwerken, die evtl. verschiedenen Typs sind oder verschiedene Eigenschaften, wie beispielsweise Sicherheit, haben. Nachrichten, die vom einen in das andere Netz fliessen sollen müssen den Gateway passieren. Gateways, die manche Nachrichten blockieren bezeichnet man als →Firewall.

Glassbox-Sicht: legt den internen Aufbau eines Komponenten-Netzwerkes des Basismodells fest, das es erfüllen muss um einer Komponentenspezifikation des Engineeringmodells zu entsprechen.

Interface: ein Interface im Engineeringmodell ist die Signatur, bezüglich derer das Verhalten eines Dienstes spezifiziert wird. Es besteht aus einem Ein- und Ausgabe-→Port, der wiederum die →Nachrichtentypen des Dienstes festlegt.

JINI: Java Intelligent Network Infrastructure. Auf der Programmiersprache Java basierende populäre Plattform für spontane Systeme. Insbesondere im Bereich Mobile Computing und Ad-Hoc Systeme eingesetzt.

Kanal: ein Kanal des Basismodells ist ein getypter, gerichteter Kommunikationsweg zwischen zwei (Basimodell-)Komponenten. Auf einem Kanal fliessen getypte Nachrichten, die wiederum getypte Parameter besitzen.

Komponente (Basismodell): das Verhalten einer Komponente im Basismodell ist durch die Relation der Ein- und Ausgangskanalhistorien modelliert. Ein Netzwerk solcher über gemeinsame Kanäle kommunizierende Komponenten bezeichnet man als Komponenten-Netzwerk.

Komponente (Engineeringmodell): eine *Komponente* ist eine →konkrete Entität des →Engineeringmodells, deren Spezifikation sowohl Verhaltens- als auch Struktur-Information besitzt. Eine Komponente besteht aus einer Menge an benötigten und einer Menge an angebotenen →Diensten. Die Dienste selbst werden gesondert spezifiziert und definieren implizit das Verhalten der Komponente. Des Weiteren ergibt sich durch sie die Abhängigkeitsfunktion der Komponente, die angibt welche benötigten von welchem angebotenen Diensten induziert

werden. Die Strukturinformation der Komponente wird durch eine Menge von Subkomponenten und deren →Bindungen definiert, was dem internen Aufbau der Komponente entspricht.

Komponenten-Netzwerk: siehe →Komponente (Basismodell).

Konkrete Entitäten: sind Entitäten des Engineeringmodells, die injektiv auf die Systemteile der Implementierung abgebildet werden. Sie besitzen also ein eindeutiges Pendant im letztendlichen System. Beispiele sind Komponenten, Sandboxes und Bindungen.

Konfiguration: eine Konfiguration stellt eine Realisierung einer logischen Architektur ausschliesslich mit →konkreten Entitäten des →Engineeringmodells dar. Die Konfiguration ist die Repräsentationsform einer →technischen Architektur im →Engineeringmodell und wird bezüglich ihrer Struktur injektiv auf die Implementierung abgebildet.

Konfigurationsklasse: eine Konfigurationsklasse ist die Menge der Konfigurationen einer Deployment-Zielmenge, die die dazugehörige logische Architektur mit der selben Menge von Komponenten und Sandboxes realisiert. Die →Umkonfiguration innerhalb einer Konfigurationsklasse ist ausschliesslich durch das Fallen lassen bzw. Aufbauen von Bindungen bzw. Migration von Komponenten möglich. Zwischen Konfigurationsklassen ist das Umkonfigurieren nur durch Hinzufügen bzw. Löschen von Komponenten möglich, was stärkere Auswirkungen auf die →Transaktionssicherheit hat.

Konfigurationsmanagement: unter Konfigurationsmanagement versteht man das Regeln von strukturellen Änderungen in einem (verteilten) System, wobei die Änderungen das System in einen konsistenten Zustand versetzen sollen und der Systemablauf möglichst wenig beeinträchtigt werden soll.

Logische Architektur: die logische Architektur eines verteilten Systems spiegelt die invarianten Abhängigkeiten der einzelnen Funktionalitäten wieder ohne sich auf konkrete Strukturen festzulegen. Bei →spontanen Systemen ist die logische Architektur die invariante →Dienstarchitektur, die durch verschiedene technische →Konfigurationen realisiert wird.

Lokation: siehe →Mobilität

Lookup-Service: siehe →Trading-Service.

Mobilität: die Einführung von Lokationen und eine zeitliche Zuordnung von Komponenten auf diese. Um zwischen Lokationen zu wechseln, können Komponenten migrieren. Hierbei unterscheidet man zwischen der zustandslosen Code-Migration und der zustandserhaltenden Prozessmigration. Die Auswirkungen der Struktur wird in zwei Klassen unterschieden: Bei homogener Mobilität in einer homogenen Umgebung gibt es keine weiteren Auswirkungen. Heterogene Mobilität in einer heterogenen Umgebung kann Auswirkungen auf Funktionalität bewirken, beispielsweise durch unterschiedliche Rechteumgebungen.

Nachrichtentypen: (engl. Message-Types) setzen die Arten von Nachrichten fest, die über ein →Interface fliessen können. Jeder Nachrichtentyp kann darüberhinaus eine Anzahl von Parametertypen besitzen, die bei dessen Spezifikation definiert werden.

.NET: auf Windows basierende populäre Plattform für spontane Systeme, insbesondere für Mobile- und Wide-Area-Systeme, die unter .NET als *Webservices* bezeichnet werden.

Port: ein Port ist Teil eines →Interfaces eines Dienstes. Es wird zwischen Ein- und Ausgang-Port unterschieden, wobei der jeweilige Port festlegt welche Nachrichtentypen fliessen können.

Redundanz (Architektureigenschaft): die *Redundanz* eines Dienstes in einer Dienstarchitektur bezüglich einer Einsatzumgebung (engl. Environment) ist die Anzahl an überschüssigen provided Dienste dieses Diensttyps.

Relevanz (Architektureigenschaft): die *Relevanz* einer Komponente für eine Dienstarchitektur gibt die Anzahl an Diensten aus der Menge ihrer angebotenen Dienste an, die in der Dienstarchitektur als freie Dienste vorkommen.

Sandbox: eine →konkrete Entität des Engineeringmodells, die Umgebungsprofile modelliert. Sandboxes sind *explizit* modellierte und adressierbare Orte, die verschiedene Umgebungsmerkmale, wie beispielsweise Kommunikationsrechte oder Durchsatz, festlegen. Mit Sandboxes können beispielsweise Wirtsrechner (engl. Hosts) in mobile Systemen, oder →Firewalls in Wide-Area-Systemen modelliert werden.

Softwarearchitektur (Architektur): mit Softwarearchitektur wird das Zerlegen eines Gesamtsystems in Teilsysteme und deren Beziehungen bezeichnet. Architekturbeschreibung beruft sich meist auf das sog. *Komponenten & Konnektoren-Modell*, das Teilsysteme als Komponenten (Berechnungseinheiten) und Konnektoren (Verbindungseinheiten), beschreibt.

Spontane Interaktion: Komponenteninstanzen spontaner Systeme sind im Gegensatz zu statischen Systemen nicht fest verdrahtet, sondern müssen sich während der Laufzeit finden (discovery), auf gegenseitige Eignung prüfen (description) und binden (join). Dieser Prozess, der im Gegensatz zu statischen verteilten Systemen, autonom abläuft wird als spontane Interaktion bezeichnet.

Spontanes (Komponenten-)System: ein spontanes Komponentensystem (kurz spontanes System) stellt ein verteiltes System dar, das zusätzlich die folgenden Eigenschaften besitzen kann: *dynamische Struktur*, *Mobilität*, *Umgebungsprofile*, sowie *spontane Interaktion*.

Stabilität (Architektureigenschaft): eine Konfiguration K ist *stabil* bezüglich eines Dienstes S_s einer Komponente C_i, wenn es eine Konfiguration K_j in der selben Konfigurationsklasse gibt, die den Dienst S_s von C_i nicht benötigt.

Technische Architektur: siehe →Konfiguration.

Topologie: eines verteilten Systems ist die Struktur der Kommunikationsbeziehungen zwischen den jeweiligen Systemteilen.

Tradingservice: ein Tradingservice (auch Lookupservice) ist ein zentraler Dienst in der Middleware spontaner Systeme, der die spontanen Interaktion regelt. Hierbei wird aufgrund von Beschreibungen ein Dienst gesucht und gegebenenfalls übergeben. Abhängig von der Mächtigkeit des Modells sind verschiedene Eigenschaften ausdrückbar. Gängige Tradingservices, wie der JINI-Lookupservice oder der .NET UDDI Service sind lediglich in der Lage Schnittstellensignaturen zu beschreiben. Durch das hier Beschriebene Engineeringmodell ist man in der Lage auch Struktur- und Verhaltensbeschreibungen abzuprüfen.

Transaktion: eine Transaktion im Engineeringmodell ist ein Nachrichtenaustausch zwischen exakt zwei Komponenten bezüglich eines Dienstes, wobei die Komponente, die den Dienst benötigt, die Transaktion initiiert. Die Transaktion beeinflusst den internen Zustand der Parteien und besteht aus einer endlichen Sequenz von Nachrichten, die über die Bindung der beiden Komponenten ausgetauscht werden. Eine Transaktion wird als abschließbar in endlicher Zeit angenommen und wird ausschließlich von der Komponente, die sie initiiert hat, also die den Dienst benötigende, abgeschlossen.

Transaktionssicherheit: unter Transaktionssicherheit spontaner Komponentensysteme versteht man die Bewahrung der Transaktionseigenschaften bei bewussten →Umkonfigurationen des spontanen Systems.

Umgebungsprofil: ein Umgebungsprofil ist eine Menge von Kommunikationsrechten, die einer Komponente oder einem Dienst, abhängig von seiner Ansiedlung, die Kommunikationsmöglichkeiten mit anderen Partnern in anderen Umgebungsprofile einschränkt. Durch die Einbeziehung von Komponenten als Nachrichten bestimmt das Umgebungsprofil neben den Kommunikationsrechten auch die Migrationsrechte.

Umkonfiguration: eine Umkonfiguration eines spontanen Systems ist der Wechsel zwischen zwei →Konfigurationen (→technischen Architekturen) einer →Dienstarchitektur (→logische Architektur). Die Umkonfiguration kann innerhalb einer →Konfigurationsklasse durch die Operationen *move* (migration), *bind* (Bindung etablieren) und *drop* (Bindung fallen lassen) ausgedrückt werden. Zwischen zwei →Konfigurationsklassen wird der Übergang zusätzlich durch die Operationen *add* (Komponente hinzufügten) und *kill* (Komponente fallen lassen) erreicht.

Wide-Area-System: ein Wide-Area-System ist ein auf einem →Wide-Area-Netzwerk angesiedeltes verteiltes System. Dadurch wird insbesondere heterogene →Mobilität impliziert. Beispiel für Wide-Area-Systeme sind Webservice-Systeme unter .NET.

Wide-Area-Netzwerk: ein Wide-Area-Netzwerk (WAN) ist ein durch verschiedene Netzwerkzonen, mit jeweils eigenen Umgebungsprofilen (z.B. Durchsatz, lokale Zeit, Rechte etc.), charakterisiertes Netzwerk. Diese Netzwerkzonen sind durch →Gateways jeweils verbunden. Beispiel hierfür ist das Internet.

SADL Definition

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) by Chris Salzmann                                  Feb 2001 -->
<!-- salzmann@in.tum.de (Munich University of Technology)        -->
<!-- http://www.w3.org/2001/XMLSchema" -->

<xsd:schema targetNamespace="http://www4.in.tum.de/despiau/sadl/"
 xmlns="http://www4.in.tum.de/despiau/sadl/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="project">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="definitions" type="definitions"/>
                <xsd:element name="deployment" type="deployment"
                 minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:ID"/>
        </xsd:complexType>
    </xsd:element>
    <xsd:complexType name="definitions">
        <xsd:sequence>
            <xsd:element name="definingCallTypes"
             type="definingCallType"/>
            <xsd:element name="definingServices"
             type="definingServices"/>
            <xsd:element name="definingComponents"
             type="definingComponents"/>
            <xsd:element name="definingServiceArchitectures"
             type="definingServiceArchitecture"/>
            <xsd:element name="definingSandboxes"
             type="definingSandboxes"/>
            <xsd:element name="definingEnvironments"
             type="definingEnvironments"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:simpleType name="callType">
        <xsd:restriction base="xsd:ID"/>
    </xsd:simpleType>
    <xsd:complexType name="deployment">
        <xsd:sequence>
            <xsd:element name="configuration" type="configuration"
             maxOccurs="unbounded"/>
            <xsd:element name="environment" type="xsd:IDREF"/>
```

```
        <xsd:element name="serviceArchitecture"
         type="xsd:IDREF"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="MessageType">
    <xsd:sequence>
        <xsd:element name="parameterType" type="parameterType"
         minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:complexType name="definingServiceArchitecture">
    <xsd:sequence>
        <xsd:element name="serviceArchitecture"
         maxOccurs="unbounded">
            <xsd:complexType>
                <xsd:complexContent>
                    <xsd:extension base="serviceArchitecture"/>
                </xsd:complexContent>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="definingEnvironments">
    <xsd:sequence>
        <xsd:element name="environment" maxOccurs="unbounded">
            <xsd:complexType>
                <xsd:complexContent>
                    <xsd:extension base="environment"/>
                </xsd:complexContent>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="definingCallType">
    <xsd:sequence>
        <xsd:element name="callType" type="callType"
         minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="definingComponents">
    <xsd:sequence>
        <xsd:element name="component" type="component"
         maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="definingServices">
    <xsd:sequence>
        <xsd:element name="service" type="service"
         maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="definingSandboxes">
    <xsd:sequence>
        <xsd:element name="sandbox" type="sandbox"
         maxOccurs="unbounded"/>
    </xsd:sequence>
```

```
</xsd:complexType>
<xsd:complexType name="possibleConfigurations">
    <xsd:sequence>
        <xsd:element name="Configuration" type="configuration"
         minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="componentInstance">
    <xsd:sequence>
        <xsd:element name="Type" type="xsd:IDREF"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:ID" use="required"/>
</xsd:complexType>
<xsd:complexType name="component">
    <xsd:sequence>
        <xsd:element name="providedService" type="xsd:IDREF"
         minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="neededService" type="xsd:IDREF"
         minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="interface" type="xsd:IDREF"
         minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="subComponent" type="componentInstance"
         minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="subComponentBinding" type="binding"
         minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:ID" use="required"/>
</xsd:complexType>
<xsd:complexType name="port">
    <xsd:sequence>
        <xsd:element name="messageType" type="MessageType"
         maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="binding">
    <xsd:attribute name="provider" type="xsd:string" use="optional"/>
    <xsd:attribute name="consumer" type="xsd:string" use="optional"/>
    <xsd:attribute name="service" type="xsd:string" use="optional"/>
    <xsd:attribute name="callType" type="xsd:string" use="optional"/>
</xsd:complexType>
<xsd:complexType name="mapping">
    <xsd:sequence>
        <xsd:sequence maxOccurs="unbounded">
            <xsd:element name="unitInstance" type="xsd:IDREF"/>
            <xsd:element name="componentInstance" type="xsd:IDREF"/>
        </xsd:sequence>
        <xsd:sequence maxOccurs="unbounded">
            <xsd:element name="serviceArchitectureSandbox"
             type="xsd:IDREF"/>
            <xsd:element name="environmentSandbox"
             type="xsd:IDREF"/>
        </xsd:sequence>
    </xsd:sequence>
    <xsd:attribute name="unit" type="xsd:IDREF" use="required"/>
    <xsd:attribute name="host" type="xsd:IDREF" use="required"/>
</xsd:complexType>
<xsd:complexType name="serviceInstance">
```

```
    <xsd:sequence>
        <xsd:element name="Type" type="xsd:IDREF"/>
        <xsd:element name="InterfaceInstance" minOccurs="0"
         maxOccurs="unbounded">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="CallType"
                     type="xsd:IDREF"/>
                    <xsd:element name="Interface"
                     type="xsd:IDREF"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:ID" use="required"/>
</xsd:complexType>
<xsd:complexType name="service">
    <xsd:sequence>
        <xsd:element name="neededService" type="xsd:IDREF"
         minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="interface" type="interface"
         maxOccurs="unbounded"/>
        <xsd:element name="behavior" type="behavior"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:ID" use="required"/>
</xsd:complexType>
<xsd:complexType name="interface">
    <xsd:sequence>
        <xsd:element name="inputMessageTypes" type="MessageType"
         minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="outputMessageTypes" type="MessageType"
         minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="callType" type="callType"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:ID" use="required"/>
</xsd:complexType>
<xsd:complexType name="behavior">
    <xsd:sequence>
        <xsd:element name="url" type="xsd:string"/>
        <xsd:element name="properties" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="sandbox">
    <xsd:sequence>
        <xsd:element name="superSandbox" type="xsd:IDREF"
         minOccurs="0"/>
        <xsd:element name="subSandbox" type="xsd:IDREF"
         minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="blockedBindingType" type="binding"
         minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:ID" use="required"/>
</xsd:complexType>
<xsd:complexType name="environment">
    <xsd:sequence>
        <xsd:element name="componentInstance"
         maxOccurs="unbounded">
```

```
            <xsd:complexType>
                <xsd:complexContent>
                    <xsd:extension base="componentInstance"/>
                </xsd:complexContent>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="sandboxInstance" type="sandbox"
         maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:ID"/>
</xsd:complexType>
<xsd:complexType name="serviceArchitecture">
    <xsd:sequence>
        <xsd:element name="componentInstance" minOccurs="0"
         maxOccurs="unbounded">
            <xsd:complexType>
                <xsd:complexContent>
                    <xsd:extension base="componentInstance"/>
                </xsd:complexContent>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="serviceInstance" type="serviceInstance"
         minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="sandbox" type="sandbox" minOccurs="0"
         maxOccurs="unbounded"/>
        <xsd:element name="binding" type="binding" minOccurs="0"
         maxOccurs="unbounded"/>
        <xsd:element name="hosting" type="hosting" minOccurs="0"
         maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:ID"/>
</xsd:complexType>
<xsd:complexType name="configuration">
    <xsd:sequence>
        <xsd:element name="componentInstance" type="xsd:IDREF"
         maxOccurs="unbounded"/>
        <xsd:element name="sandbox" type="xsd:IDREF"
         maxOccurs="unbounded"/>
        <xsd:element name="binding" type="binding"
         maxOccurs="unbounded"/>
        <xsd:element name="hosting" type="hosting"
         maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:ID"/>
</xsd:complexType>
<xsd:complexType name="parameterType">
    <xsd:simpleContent>
        <xsd:extension base="xsd:string">
            <xsd:attribute name="name" type="xsd:string"
             use="required"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>
<xsd:element name="definingSandboxes"/>
<xsd:complexType name="hosting">
    <xsd:attribute name="host" type="xsd:IDREF"
     use="required"/>
```

```
        <xsd:attribute name="units" type="xsd:IDREFS"
         use="required"/>
    </xsd:complexType>
</xsd:schema>
```

INDEX

M^{ω} – Menge der gezeiteten Ströme, 76
$\overline{C}$ – Bündel von Strömen, 77
$\overrightarrow{\mathbf{CH}}$ – Kanalbelegung, 77
π-Kalkül, 18
CH – Menge der getypten Kanäle , 75
M – Menge der Nachrichten, 75
$\mathbf{T_M}$ – Menge der Nachrichtentypen, 75
$\mathbf{T_{CH}}$ – Menge der Kanaltypen, 75
.NET, 53

ACME, 29
Ambient-Kalkül, 21
Anwendung
    Tradingservice, 153
Architektur, 28
    Architekturstil, 30
    Beschreibungen, 28
    dynamisch änderbare, 30
    Eigenschaften, 147
        Autarkie, 150
        Breite, 148
        Flexibilität, 149
        Redundanz, 148
        Relevanz, 150
        Stabilität, 151
Architekturabstraktion eines spontanen Systems, 138
Atomare Spezifikationen, 83
Aufwand eines Modells, 61

Basismodell
    Komponente, 77
Basismodell , 75
    Dienst, 79
    Komponenten-Netzwerke, 81
    Spezifikation, 83
    Spezifikationen, 82
    Komponentenkomposition, 78
Bindung, 103

COM - Component Object Model, 42
CORBA - Common Object Request Broker Architecture, 31, 41

DANTE, 179
    Aufbau, 179
    Codegenerator, 183
    Deployment, 186
    Funktionalitäten, 186
    Graphischer Client, 181
    Repository, 180
    SADL Client, 182
    Verhaltensäquivalenz, 186
DARWIN, 25
Dot-Net, 53
Dynamik, 73
dynamisches Verhalten, 64

Engineeringmodell, 93
    Dienst
        Blackbox-Sicht, 128
        Verhaltens-Sicht, 128
    Konkrete Entitäten, 96
    Abbildung auf Java Jini, 170
    abstrakte Entitäten, 108
    Bindung, 103, 130
    Deployment Abbildung, 112
    Deployment-Abbildung, 95
    Dienst, 108
    Dienstarchitektur, 109
    Dynamik, 95
    Einsatzumgebung, 106
    Entwurfsablauf, 110

freier Dienst, 108
Hosting, 106
Komponente, 96
Konfiguration, 107
logische Architektur, 95
Mobilität, 95
Sandbox, 104
Sandboxes
geq, 106
Spezifikation, 110
technische Architektur, 95
Unit, 102
Entwicklungsprozess, 39
Entwurfsablauf, 177

Fallbeispiel, 191
Komponente, 202
Drucker, 205
Faxgerät, 204
Mobiltelefon, 203
PDA, 205
Service
Calendar, 191, 195
Environment, 201
GroupFax, 193, 198
Phonebook, 192, 195
Print, 191, 194
PrintSchedule, 193, 199
SendFax, 192, 198
SendSMS, 192, 196
SendSMSbyName, 193, 196
Servicearchitecture, 200
Services, 193
Fallbeispiel - “Mobile Office Szenario”, 68
Feature-Interaction, 79
FOCUS, 26

Hosting, 106

IDL - Interface Definition Language, 31
Interoperabilität, 66

JINI - Java Intelligent Network Infrastructure, 49

Kanalbelegung, 77
Komponente, 29
Basismodell, *siehe* Basismodell
Ruhezustand, 146
Komponenten & Konnektoren, 28
Komponentenkomposition, *siehe* Basismodell
Komponentennetzwerk, 81
Spezifikation von, 83
Komponentennetzwerk
logisch, 89
technisch, 87
technisch und logisch, 89
Kompositionsspezifikationen, 86
Konfigurationsmanagement, 23
Konnektor, 29

logische Architektur, 65
logische Dienstebene, *siehe* logische Dienstebene, 139
Lokation, 29

Mächtigkeit eines Modells, 61
Mobile Computing, 35
mobiles Verhalten, 64
Mobilität, 45, 73
Mobilität
strukturelle, 46
Modell
Basis, *siehe* Basismodell
Modell
Engineeringmodell
see Engineeringmodell, 93
Modellierungsumgebung
sieheDANTE, 179

OCL - Object Constraint Language, 31

Pervasive Computing, 34

Reflektionsmodelle, 27
Restriktion, 106
Ruhezustand
Algorithmus, 147
RUP - Rational Unified Process, 39

SADL
Automaten, 166
Binding, 164

Component, 161
Componenttype, 161
Dienst, 161
Dienstarchitektur, 165
DienstType, 163
Envrionment, 165
Hosting, 164
Sandbox, 163
Service Architecture Definition Language, 155
Units, 160
Vergleich mit WSDL, 168
Verhaltensspezifikation, 166
Spontane Interaktion, 48, 73
STD – State Transition Diagram, 168
Strom, 76

Technische Konfigurationsebene, 141
Tradingservice, 153
Transaktion, 144
Übergänge, 144

Ubiquitous Computing, 34
Umgebungsprofile, 47, 65, 73
UML - Unified Modeling Language, 32
UPnP - Universal Plug and Play, 52

Wide-Area-Computing, 35
WSDL – Web Service Definition Language, 168

XML – Extensible Markup Language, 156