Institut für Informatik
der Technischen Universität München

Fehlertoleranz durch automatisierte Diversität im Management verteilter nebenläufiger Systeme

Jörg Preißinger

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. Uwe Baumgarten

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Peter Paul Spies
2. Univ.-Prof. Dr. Arndt Bode

Die Dissertation wurde am 18. September 2007 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 13. Februar 2008 angenommen.

Kurzfassung

Die Geschichte der Softwareentwicklung zeigt, dass Maßnahmen zur Fehlervermeidung einen großen Beitrag zur Verbesserung der Zuverlässigkeit von Softwaresystemen leisten. Allerdings bleiben trotz dieser Verfahren in komplexen Softwaresystemen in der Regel Fehler zurück, deren Auftreten häufig von seltenen, besonderen Ausführungsbedingungen, wie zeitlichen Ausführungsabläufen, Ressourcen-Engpässen oder Umwelteinflüssen abhängen. Insbesondere in verteilten Systemen müssen deshalb die fehlervermeidenden Verfahren mit fehlertolerierenden Verfahren kombiniert werden, um die Zuverlässigkeit von Systemen weiter zu verbessern. Dazu sind Verfahren notwendig, die zur Toleranz dieser Art unvorhersehbarer, ablaufabhängiger Fehler dienen.

In dieser Arbeit wurde im Rahmen des Projekts MoDiS ein Konzept zur Toleranz dieser Fehler entwickelt. Das Projekt MoDiS verfolgt einen sprachbasierten Gesamtsystemansatz zur Konstruktion verteilter, kooperativer Systeme. In der objektbasierten Sprache INSEL werden auf hohem Abstraktionsniveau verteilte Anwendungen spezifiziert, wobei Details der technischen Realisierung, insbesondere der Verteilung und Umsetzung der Kommunikation vor dem Programmierer verborgen bleiben. Das zentrale Konzept des MoDiS-Projekts ist die Nutzung der zur Übersetzungs- und Laufzeit gewonnenen Anwendungsinformationen im Management des Systems, um eine anwendungsorientierte Ausführung auf der gegebenen Hardwarekonfiguration zu gewährleisten.

Das in dieser Arbeit entwickelte Verfahren zur Toleranz der oben beschriebenen Fehler basiert auf automatisierter Diversität im System-Management. Die Entscheidungen des System-Managements zur Konkretisierung der abstrakten Spezifikation beeinflussen zeitliche Ausführungsabläufe und Ressourcen-Verwaltung, und somit die genannten Ausführungsbedingungen, die häufig Auslöser der ge-

nannten Fehler sind. Das Fehlertoleranzverfahren basiert auf konsistenten, gespeicherten Systemzuständen zur Rückwärtsbehebung, um bei erneuter Ausführung die Konkretisierungs-Entscheidungen zu variieren. Das Management nutzt dazu die Freiheitsgrade der Konkretisierung, um auf Basis gesammelter Informationen geeignete Alternativen im Entscheidungsraum zu finden. Es wird beschrieben, auf welcher Grundlage das Management Entscheidungen treffen kann, um sowohl den Grad der Diversität der Ausführung als auch den Einfluss auf die System-Performanz zu berücksichtigen. Der Unterschied zweier Ausführungsabläufe wird in der Arbeit als Varianz bezeichnet und stellt einen Teil der Entscheidungsgrundlage dar. Es wird die Annahme erläutert, dass die Varianz direkt proportional zur Wahrscheinlichkeit der Toleranz eines ablaufabhängigen Fehlers ist. Der Ansatz wird anhand der Zuweisung von Akteuren zu Stellen im MoDiS-Projekt veranschaulicht.

Zur Rückwärtsbehebung wurde ein optimiertes, kommunikationsinduziertes Checkpoint-Rollback Verfahren entwickelt. Die Performanz kommunikationsinduzierter Checkpoint-Rollback-Verfahren leidet in der Praxis unter zu kurzen Checkpoint-Intervallen auf Grund erzwungener Checkpoints. Um dies zu vermeiden werden im Übersetzer des MoDiS-Projekts die Kommunikationsabhängigkeiten der aktiven Komponenten analysiert und die Platzierung freiwilliger Checkpoints vorbereitet, sodass zu Gunsten einer verbesserten Gesamtperformanz kurze Intervalle durch erzwungene Checkpoints mit hoher Wahrscheinlichkeit vermieden werden. Zur Laufzeit werden die zur Übersetzungszeit getroffenen Entscheidungen überprüft und ggf. revidiert, falls das tatsächliche Kommunikationsverhalten von der Approximation abweicht.

Zur Fehlererkennung werden zwei im Rahmen dieser Arbeit in das MoDiS-Projekt integrierte Mechanismen genutzt. Zum einen wird der globale Systemzustand beobachtet und ein Graph kausaler Abhängigkeiten der Ereignisse konstruiert, wobei die Schwächen der auf Vektoruhren basierenden Verfahren in Bezug auf dynamische Prozesssysteme umgangen werden. Die Theorie zur Konstruktion konsistenter Sichten wurde dazu für Systeme mit gemeinsamem, verteilten Speicher erweitert. Ferner wurde zur Fehlererkennung das objektorientierte Vertragskonzept *Design by Contract* für INSEL-Komponenten adaptiert und in die Sprache integriert. Die Laufzeitüberprüfung der Verträge wird im Übersetzer vorbereitet und ermöglicht Fehlererkennung auf Komponentenebene. Die erarbeiteten Konzepte und Verfahren werden jeweils in Bezug auf ihre Leistungsfähigkeit sowie ihre Anwendungsmöglichkeit für andere Systeme diskutiert.

Danksagung

An erster Stelle gilt mein herzlichster Dank Herrn Prof. Dr. P. P. Spies für die Betreuung meiner Arbeit, die persönliche und fachliche Unterstützung in den Jahren meiner Ausbildung sowie für die zahlreichen Denkanstöße, die zum Gelingen dieser Arbeit entscheidend beigetragen haben. Ebenso herzlich möchte ich mich bei Herrn Prof. Dr. A. Bode für die Begutachtung dieser Arbeit bedanken.

Ich bedanke mich bei allen Kollegen des Lehrstuhls für die fruchtbaren Diskussionen, Anregungen und gute Zusammenarbeit, aber auch für das kollegiale Klima in den vergangenen Jahren. Besonderer Dank gebührt Herrn Dr. C. Rehn für das Korrekturlesen von Teilen dieser Arbeit.

Ferner haben eine Reihe studentischer Hilfskräfte, Praktikanten und Diplomanden zur Entstehung dieser Arbeit und den zu Grunde liegenden Implementierungen beigetragen. Stellvertretend seien hier Mark Pflüger und Alexander Mayer genannt, die über ihre gestellten Aufgaben hinaus noch an zwei Veröffentlichungen mitgewirkt haben.

Besonderer Dank geht zudem an Herrn Dr. T. M. Peschel-Findeisen für seine ausgiebige Betreuung während meines Studiums, die mir praxisnahe Einblicke in die aktive Forschung im Bereich der Informatik gewährte und meinen Entschluss zur Promotion festigte.

Nicht zuletzt danke ich meinen Eltern, Klaus und Eva Preißinger, die mich stets in der Erstellung dieser Dissertation unterstützt haben.

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Aufgabenstellung 4
1.2. Lösungsansatz 4
1.3. Gliederung der Arbeit 5

2. Konstruktion verteilter nebenläufiger Systeme **9**
2.1. Verteilte Systeme 10
2.2. Konzepte zur Konstruktion verteilter nebenläufiger Systeme . . . 12
2.2.1. Komponenten 13
2.2.1.1. Identifikation 17
2.2.1.2. Ordern 18
2.2.1.3. Depots 18
2.2.1.4. Akteure 19
2.2.2. Systemstrukturen 20
2.2.2.1. Definitionsstruktur δ 21
2.2.2.2. Ausführungsstruktur α 21
2.2.2.3. Lokalitätsstruktur λ 23
2.2.2.4. Lebenszeitstruktur ϵ 23
2.2.3. Konkretisierung 25
2.2.4. Systembeschreibung 27
2.2.5. INSEL 29
2.3. MoDiS-Experimentalsystem 29
2.3.1. Übersetzer 30
2.3.2. Binder 31
2.3.3. Laufzeit-Management 32
2.4. Zusammenfassung 33

3. Fehlertoleranz in verteilten Systemen **35**
3.1. Fehler 36
3.1.1. Versagen – failure 36
3.1.2. Fehlerzustand – error 38
3.1.3. Fehlerursache – fault 38
3.1.4. Fehler – allgemein 39
3.2. Fehlererkennung 40
3.2.1. Timing-Tests 41
3.2.2. Replikations-Tests 42
3.2.3. Struktur-Tests 42
3.2.4. Semantik-Tests 42
3.2.5. Diagnose-Tests 43
3.2.6. Sonstige Mechanismen 43
3.3. Fehlertoleranz 44
3.3.1. Fehlerbehebung 45
3.3.1.1. Rückwärtsbehebung 45
3.3.1.2. Vorwärtsbehebung 46
3.3.1.3. Seitwärtsbehebung 47
3.3.2. Toleranz von Hardwarefehlern 47
3.3.3. Toleranz von Softwarefehlern 49
3.4. Zusammenfassung 51

4. Fehlererkennung in MoDiS **53**
4.1. Systembeobachtung 54
4.1.1. Konsistente Sichten in MoDiS 56
4.1.1.1. Erzeugung und Terminierung von Akteuren . . . 59
4.1.1.2. Gemeinsamer verteilter Speicher 60
4.1.2. Realisierung 66
4.1.2.1. Logische Uhren 66
4.1.2.2. Verweise direkter Abhängigkeiten 68
4.1.2.3. Konstruktion der Ereignisordnung 69
4.1.3. Fehlererkennung auf Basis konsistenter Sichten 72
4.1.4. Bewertung 77
4.2. Integration des Vertragskonzepts in INSEL 79
4.2.1. Design by Contract 79
4.2.1.1. Vorteile von Design by Contract 81
4.2.1.2. Realisierungs-Ansätze von Design by Contract . 82
4.2.1.3. Bewertung der Ansätze 84
4.2.2. Verträge in INSEL 85
4.2.2.1. Depots 86
4.2.2.2. Routinen 89

4.2.2.3. BS-Order 89
4.2.2.4. K-Akteure und K-Order 90
4.2.2.5. Auswertung der Verträge 90
4.2.2.6. Implementierung 91
4.2.3. Fehlererkennung auf Basis der Verträge 92
4.2.4. Bewertung 94
4.3. Zusammenfassung 95

5. Speicherung konsistenter Schnitte 97
5.1. Checkpoint-Verfahren 97
5.1.1. Unkoordiniertes Checkpointing 99
5.1.2. Koordiniertes Checkpointing 101
5.1.3. Kommunikationsinduziertes Checkpointing 103
5.1.4. Diskusssion 105
5.2. Kommunikationsinduziertes Checkpointing in MoDiS 110
5.2.1. Speicherung der Rücksetzpunkte 110
5.2.2. Checkpoint-Protokoll in MoDiS 119
5.2.3. Rollback 123
5.2.4. Verwerfen der Checkpoints 126
5.2.5. Anwendungsangepasste Checkpoint-Platzierung 127
5.2.5.1. Greedy-Verfahren 128
5.2.5.2. Vorhersage der Systementwicklung 129
5.2.5.3. Positions-Bewertung ohne Abhängigkeiten 130
5.2.5.4. Einbezug von Kommunikation 131
5.2.5.5. Einbezug von Erzeugung und Terminierung . . . 135
5.2.5.6. Abbruchbedingung 136
5.2.5.7. Bewertungsfunktion 136
5.2.5.8. Schleifen und bedingte Anweisungen 137
5.3. Bewertung 139
5.3.1. Simulator 139
5.3.2. Ergebnisse 141
5.4. Verwandte Arbeiten 144
5.5. Zusammenfassung 145

6. Automatisierte Diversität auf Management-Ebene 147
6.1. Kernidee 147
6.1.1. Probleme der Softwareentwicklung 148
6.1.1.1. Qualitätssicherungs-Maßnahmen 150
6.1.2. Diversität 152
6.1.2.1. Diskussion diversitärer Programmierung 153
6.1.2.2. Sicherheit durch Diversität 155

6.1.3. Diversität auf Management-Ebene 156
6.2. Fehlertoleranz durch Diversität auf Management-Ebene 159
6.2.1. Varianz 161
6.2.2. Kosten 166
6.2.2.1. Stellenzuweisung und Systemperformanz 169
6.2.2.2. Kosten verschiedener Stellenzuweisungen 174
6.2.3. Strategie 181
6.2.4. Generierung von Alternativen 183
6.2.4.1. Suchverfahren 184
6.2.4.2. Algorithmische Generierung 187
6.2.4.3. Informationsgewinnung 189
6.2.4.4. Umsetzung einer Alternative 190
6.2.5. Entscheidungsräume 191
6.2.5.1. Kombination von Entscheidungen 196
6.2.6. Gesamtverfahren im Überblick 200
6.3. Diskussion 203
6.3.1. Leistungsfähigkeit 203
6.3.1.1. Tolerierbare Fehler 203
6.3.1.2. Effizienz 206
6.3.1.3. Verbesserung der Alternativenwahl 207
6.3.2. Fehlerbehebung durch Informationen über Alternativen . 207
6.4. Verwandte Arbeiten 208
6.5. Zusammenfassung 209

7. Zusammenfassung und Ausblick 211
7.1. Zusammenfassung 211
7.1.1. Fehlererkennung 212
7.1.2. Fehlerbehebung 213
7.2. Diskussion 214
7.2.1. Anwendungsorientiertes Management 215
7.3. Ausblick 217

A. INSEL-Beispiel 219

B. Generierte Ereignisgraphen 221

C. INSEL-Beispiel Verklemmung 223

D. Grammatik-Erweiterungen im Übersetzer für Verträge 227

Definitionsverzeichnis 229

Abbildungsverzeichnis **231**

Literaturverzeichnis **233**

Kapitel 1

Einleitung

Inhalt

1.1. Aufgabenstellung 4
1.2. Lösungsansatz 4
1.3. Gliederung der Arbeit 5

Informationsverarbeitende Systeme werden in immer mehr Bereichen des Lebens unverzichtbar, die zu lösenden Anwendungsprobleme werden zugleich vielfältiger und komplexer. Verschiedene Gründe erfordern den Einsatz verteilter Systeme zur Lösung dieser Anwendungsprobleme. Ein Grund sind Abhängigkeiten räumlich verteilter Systemteile, wie beispielsweise verschiedene kooperierende Steuergeräte in Fahrzeugen, oder die Kooperation von Business-Anwendungen verschiedener Firmen. Trotz stetig steigender Leistung der Rechner ist auch die Komplexität zu lösender Probleme ein Grund für den Einsatz verteilter Systeme, da in vielen Fällen nur eine kooperative, parallele Lösungsberechnung die gewünschte Performanz erzielen kann. Beispiele für entsprechend aufwändige Berechnungen sind (nach [KK07], S. 193):

- Simulationen: Dazu zählen beispielsweise Strömungssimulationen, wie unter anderem Wetter- und Klima-Berechnungen. Auch astronomische Simulationen, wie beispielsweise die Simulationen von Körpern, die sich gegenseitig durch Gravitation beeinflussen, sind sehr rechenaufwändig.
- Optimierungsprobleme: Die Berechnungen optimaler Ressourcennutzung können ebenfalls auf Grund der verschiedenen Kriterien, die berücksichtigt

werden müssen, sehr komplex sein. Zu dieser Klasse gehört beispielsweise die Planung einer Fluggesellschaft über den Einsatz ihrer Flugzeuge und Crews (unter Berücksichtigung vorgeschriebener Ruhezeiten, Wartungsintervalle der Flugzeiten, Lizenzen der Piloten usw.).

- Biochemische Berechnungen: Im Bereich der Biochemie sind häufig sehr große Zustandsräume in Berechnungen abzudecken, beispielsweise sind Proteinfaltungen komplex genug, dass heutige Rechenleistungen für einen praktischen Einsatz dieser Technik noch lange nicht ausreichen.

In diesen rechenaufwändigen und dadurch lange laufenden Berechnungen sollen auftretende Soft- oder Hardwarefehler nicht zum Verlust der bereits erbrachten Ergebnisse führen. Ein Verlust der Zwischenergebnisse mit folgendem Neustart der Berechnungen ist in der Regel mit hohen Kosten verbunden. Darüber hinaus existieren Systeme, bei denen ein Ausfall neben Kosten auch katastrophale Folgen nach sich ziehen kann. Beispielsweise bei Steuerungssystemen von Flugzeugen, Autos, Industrieprozessen usw. kann ein Versagen menschlichen Schaden verursachen. Aus diesen Gründen werden insbesondere an verteilte Systeme häufig hohe Anforderungen der Zuverlässigkeit gestellt. Gleichzeitig ist die Software verteilter Systeme allerdings auf Grund der kooperativen, nebenläufigen Problemlösungen höchst komplex.

Zuverlässigkeit verteilter Systeme. Im Bereich der Softwareentwicklung wurde viel Forschungsarbeit zur Verbesserung der Qualität von Software geleistet. Die Methoden und Verfahren zur Fehlervermeidung, wie beispielsweise Testverfahren oder Model-Checking, tragen dazu bei, die Zahl der Softwarefehler zu reduzieren. Allerdings wächst die Komplexität der Software in heutigen Systemen zunehmend, sodass trotz fehlervermeidender Verfahren komplexe Software in der Praxis nie gänzlich fehlerfrei ist. Ferner ist die Hardware, auf der die Software ausgeführt wird, auf Grund physikalischer Beeinflussung, Alterung, sowie Mängeln im Produktionsprozess ebenfalls eine Quelle von Fehlern. Das Problem auftretender Hardwarefehler zur Ausführungszeit von Systemen kann durch Fehlertoleranz weitgehend gelöst werden. Durch Redundanz wird erreicht, dass Fehler einzelner Hardware-Einheiten nicht zum Versagen eines Systems führen, sondern toleriert werden können (vgl. [KK07], Kap. 2).

Um die steigenden Anforderungen an die Zuverlässigkeit von Systemen erfüllen zu können, müssen Konzepte entwickelt werden, die die fehlervermeidenden Verfahren der Softwareentwicklung um fehlertolerierende Verfahren ergänzen, sodass – zusätzlich zu Hardwarefehlern – auch verbleibende Softwarefehler zur Ausführungszeit toleriert werden können. Im Gegensatz zu Hardwarefehlern, die bei

einzelnen Einheiten durch äußerliche Einflüsse entstehen, ist Software ein abstraktes Konzept zur Problemlösung. Software ist demnach entweder fehlerfrei, oder aber jede Instanz, also jede Kopie dieser Software enthält in jeder Ausführung die identischen Softwarefehler. Ob diese Softwarefehler allerdings zu Tage treten und zum Versagen führen, kann wiederum in verschiedenen Ausführungen je nach Umgebung und Eingabe unterschiedlich sein.

Toleranz von Softwarefehlern. Um allgemeine, nicht näher spezifizierte Softwarefehler tolerieren zu können, muss demnach die Redundanz um Diversität erweitert werden. Die Entwicklung verschiedener Versionen funktional identischer Software ist der als *diversitäre Programmierung* bezeichnete Ansatz, dies zu realisieren. Dabei soll von den verschiedenen Implementierungen einer Software die gleiche Problemlösung redundant berechnet werden. Man geht jedoch davon aus, dass die einzelnen Implementierungen unterschiedliche Softwarefehler enthalten, sodass fehlerhafte Zwischenergebnisse erkannt und behoben werden können. Praktische Erfahrungen zeigen allerdings, dass einerseits die Entwicklung mehrerer funktional identischer Softwareversionen zu kostspielig und andererseits verschiedene Programmierer häufig ähnliche Fehler machen, sodass die verschiedenen Implementierungen der Software häufig an den gleichen Stellen versagen. Das Problem der Toleranz nicht näher spezifizierter Softwarefehler ist demnach bis dato nicht zufriedenstellend gelöst.

Die Softwarefehler, die häufig trotz fehlervermeidender Verfahren der Softwareentwicklung unentdeckt bleiben, haben häufig die Eigenschaft, dass sie nur in bestimmten, seltenen Systemabläufen auftreten, wie beispielsweise Race Conditions oder Heisenbugs (vgl. [Gra86]). Ressourcenverwaltung und zeitliche Abläufe einer Ausführung sind häufig ausschlaggebend für das Auftreten dieser Fehler. Eine Möglichkeit zur Toleranz dieser Softwarefehler ist demnach die Änderung der Ausführungsabläufe einer fehlerhaften Softwareversion, sodass ein aufgetretener Fehler nicht erneut auftritt. Dies kann erreicht werden, indem das Management des Systems automatisiert seine Entscheidungen zur Konkretisierung, also Entscheidungen zur Übersetzung, Ausführungssteuerung und Ressourcenverwaltung variiert, sodass unterschiedliche Ausführungsabläufe erzeugt werden. Im Gegensatz zu diversitärer Programmierung liegt die Diversität bei diesem Ansatz nicht im Quellcode der Software, sondern in den Ausführungsabläufen. Das Management sucht nach einer Ausführungsalternative, bei der im Quellcode der Anwendung enthaltene Softwarefehler nicht auftreten.

Verteilte Systeme im Projekt MoDiS. Am Lehrstuhl für Betriebssysteme und Systemarchitektur wurden im Rahmen des Projekts MoDiS Konzepte zur Kon-

struktion verteilter, nebenläufiger Systeme entwickelt, die eine einfache und effiziente Nutzungsmöglichkeit der verteilten Hardwarekonfiguration ermöglichen (vgl. [SEL+96]). Die Systeme werden in einem sprachbasierten, Top-Down-orientierten Gesamtsystemansatz konstruiert. Der Programmierer kann kooperative Problemlösungen auf hohem Abstraktionsniveau in der Sprache INSEL spezifizieren (vgl. [RW96]). Die in INSEL realisierten Konzepte ermöglichen die Konkretisierung dieser Spezifikation unter Einbezug der Anwendungsinformationen, sodass eine – der Anwendung angepasste – verteilte Ressourcenverwaltung realisiert wird. Einerseits wird somit der Programmierer von der komplexen Aufgabe entlastet, die Ressourcen der verteilten Hardwarekonfiguration selbst zu verwalten. Andererseits kann dennoch eine hohe Effizienz der Ausführung erzielt werden, da die Entscheidungen zur Ressourcenverwaltung an die Bedürfnisse der Anwendung angepasst sind, und nicht der häufig schlechten Performanz von uniformen Strategien zur Ressourcenverwaltung unterliegen.

1.1 Aufgabenstellung

In dieser Dissertation sollen im Rahmen des Projekts MoDiS Konzepte und Verfahren zur Fehlertoleranz, insbesondere zur Toleranz von Softwarefehlern in verteilten Systemen erarbeitet werden. Der Schwerpunkt der Arbeit liegt in der Konzeption eines Verfahrens zur Toleranz von Softwarefehlern unter Vermeidung der Kritikpunkte diversitärer Programmierung. Dazu müssen Diversität und Redundanz auf Management-Ebene automatisiert realisiert werden, um Soft- und Hardwarefehler, deren Auftreten nicht deterministisch ist, in bestimmten Ausführungsabläufen durch Variation der Management-Entscheidungen tolerieren zu können.

Ein weiterer Schwerpunkt der Arbeit besteht darin, gemäß der Konzepte des Projekts MoDiS die Verfahren und Mechanismen in das Management des Systems zu integrieren, sodass die Toleranz von Fehlern für den Benutzer bzw. Programmierer transparent und automatisiert durchgeführt wird. Um dieses Ziel zu erreichen sollen Anwendungsinformationen genutzt werden, auf deren Basis die Entscheidungen des Managements an die ausgeführte Anwendung angepasst und somit optimiert werden.

1.2 Lösungsansatz

In dieser Arbeit wurde zur Lösung der Aufgabenstellung ein Ansatz verfolgt, in welchem ein Konzept zur Fehlertoleranz in verteilten Systemen in drei aufeinander aufbauende Teile strukturiert wird und jeweils geeignete Verfahren erarbeitet werden:

- Mechanismen zur Fehlererkennung werden an das Systemmodell des Projekts MoDiS angepasst und in das Management integriert. Dabei ist das Ziel, möglichst viele Klassen von Fehlern, insbesondere Softwarefehler, zu erkennen, um im Management mit fehlerbehebenden Verfahren darauf reagieren zu können.

- Ein Ansatz zur Rückwärtsbehebung wird verfolgt, um eine wiederholte Berechnung von Teilen der Problemlösung zu ermöglichen. Das Verfahren soll automatisiert vom Management des Systems gesteuert werden und im Normalbetrieb die Systemperformanz möglichst wenig beeinflussen. Im Fehlerfall sollen Teile der Problemlösung möglichst flexibel wiederholt ausgeführt werden können, um die Fehlerbehebung durch variierte Management-Entscheidungen zu ermöglichen.

- Das Konzept der automatisierten Diversität im Management wird ausgearbeitet. Im Detail wird erläutert, wie das Management seine Entscheidungen automatisiert variieren und dadurch geeignete Alternativen der Ausführung finden kann, um aufgetretene Fehler zu tolerieren.

1.3 Gliederung der Arbeit

Die weitere Arbeit gliedert sich in sechs auf diese Einleitung folgende Kapitel, wobei Kapitel 2 und 3 Grundlagen zu MoDiS und zum Thema Fehlertoleranz beschreiben, Kapitel 4, 5 und 6 den Hauptteil in Form der drei aufeinander aufbauenden Teile des Verfahrens bilden, und Kapitel 7 eine abschließende Zusammenfassung und kapitelübergreifende Bewertung enthält:

Kapitel 2 gibt einen Überblick über die Konzepte des Projekts MoDiS zur Konstruktion verteilter, nebenläufiger Systeme. Neben den Konzepten wird auch die aktuelle Implementierung dieser Konzepte in Form des MoDiS-Experimentalsystems kurz vorgestellt.

Eine Einführung in den Bereich Fehlertoleranz wird in **Kapitel 3** gegeben. Der Fehlerbegriff wird mit seinen unterschiedlichen Ausprägungen erklärt sowie typische Ansätze zur Fehlererkennung beschrieben. Vorgehensweisen zur Fehlerbehebung werden erläutert und deren Umsetzung und praktischer Einsatz zur Toleranz von Hard- und Softwarefehlern differenziert.

In **Kapitel 4** werden zwei erarbeitete Verfahren zur Fehlererkennung vorgestellt. Die kausalen Zusammenhänge in Systemen mit dynamischer Erzeugung und Auflösung von Prozessen und gemeinsamem, verteilten Speicher werden in Abschnitt 4.1 formal erarbeitet und ein Verfahren zur Erfassung einer kausalen Ordnung der Ereignisse im Systemmodell des Projekts MoDiS erklärt. Die Kenntnis der kausalen Abhängigkeiten ist relevant für die Einhaltung der Konsistenz in den später beschriebenen Verfahren zum Speichern systemglobaler Zustände. Die konstruierte Ereignisordnung wird zur Erkennung von Timing-Fehlern genutzt. Zudem wurde das Konzept der Verträge für die Sprache INSEL und die in MoDiS zur Verfügung stehenden Komponentenarten adaptiert. In Abschnitt 4.2 ist das Vertragskonzept allgemein beschrieben, ferner werden relevante Realisierungsalternativen anhand der Sprache Java beleuchtet. Die Adaption an die Komponentenarten der Sprache INSEL und die Integration in den Übersetzer werden erklärt. Die Fehlererkennung durch Überprüfung spezifizierter Verträge zur Laufzeit wird diskutiert; die Ergebnisse der beiden Verfahren zur Fehlererkennung werden zusammengefasst.

Kapitel 5 beschreibt die Basis zur Rückwärtsbehebung in Form eines an das MoDiS-Systemmodell angepassten, kommunikationsinduzierten Checkpoint-Rollback-Verfahrens. Dieses Verfahren realisiert die Speicherung konsistenter Systemschnitte, zu denen im Fall eines erkannten Fehlers zurück gesetzt werden kann, um die Ausführung von dort fortzusetzen. Das Problem der schlechten Performanz kommunikationsinduzierter Checkpoint-Rollback-Verfahren auf Grund zu kurzer Checkpoint-Intervalle wird adressiert, indem die Platzierung der Checkpoints im Übersetzer vorbereitet wird. Dazu wird das Kommunikationsverhalten der Anwendung analysiert und der resultierende Overhead verschiedener Checkpoint-Platzierungen approximiert und bewertet. So kann der durchschnittliche Overhead des Verfahrens zur Laufzeit verringert werden. Der Performanzgewinn durch die Platzierung im Übersetzer wird mittels Simulation evaluiert und die Ergebnisse werden zusammengefasst.

Das Konzept der Fehlertoleranz auf Basis automatisierter Diversität im Management wird in **Kapitel 6** erarbeitet. Durch die Verfahren der Kapitel 4 und 5 können Hard- und Softwarefehler erkannt, und das System zu einem zuvor gespeicherten Zustand zurück gesetzt werden. Um zu verhindern, dass ein vorhandener Softwarefehler bei erneuter Ausführung identisch auftritt, werden nun die

Managemententscheidungen variiert, da diese häufig das Auftreten von Softwarefehlern, wie beispielsweise Race Conditions, beeinflussen. Es wird vorgestellt, auf welcher Basis Entscheidungen im Management mit dem Ziel der Fehlerbehebung getroffen und entsprechende Alternativen der Konkretisierung generiert werden können. Das Konzept wird anhand der Zuweisung von Akteuren zu Stellen im MoDiS-Management veranschaulicht und diskutiert.

Kapitel 7 fasst die Ergebnisse der Arbeit zusammen und diskutiert diese in Hinblick auf die Konstruktion verteilter Systeme einerseits und auf Fehlertoleranz andererseits. Auf mögliche fortführende Arbeiten zur Weiterentwicklung der Konzepte wird ein Ausblick gegeben.

Kapitel 2

Konstruktion verteilter nebenläufiger Systeme

Inhalt

2.1. Verteilte Systeme 10

2.2. Konzepte zur Konstruktion verteilter nebenläufiger Systeme . 12

2.3. MoDiS-Experimentalsystem 29

2.4. Zusammenfassung 33

Im Rahmen des Projekts MoDiS (Model oriented Distributed Systems) wurden Konzepte zur Konstruktion verteilter, nebenläufiger Systeme entwickelt. Diese Konzepte werden in Form von Komponenten durch die objektbasierte Sprache INSEL (INtegration and SEparation supporting Language) zur Verfügung gestellt, um kooperative Problemlösungen auf hohem Abstraktionsniveau spezifizieren zu können. Die abstrakte Spezifikation wird durch das MoDiS-Management schrittweise zu einem System – bestehend aus Management und Anwendung – transformiert und ausgeführt. In Form des MoDiS-Experimentalsystems existiert eine Implementierung dieser Konzepte, auf deren Basis die hier vorliegende Arbeit entstanden ist. In diesem Kapitel werden die grundlegenden Konzepte des MoDiS-Projekts zur Konstruktion verteilter, nebenläufiger Systeme sowie die entscheidenden Bestandteile des Experimentalsystems vorgestellt und erklärt.

2.1 Verteilte Systeme

Es existieren unterschiedliche Ausprägungen verteilter Systeme sowie verschiedene Ansätze zu deren Konstruktion. Verteilte Systeme unterscheiden sich in Details des Systemmodells, wie beispielsweise der Art der Kommunikation, der zur Verfügung stehenden Mechanismen zur Transparenz der Verteilung und der Abstraktion vom konkreten Rechensystem sowie dem Ansatz zur Entwicklung der Anwendungen. Bevor auf das MoDiS-Projekt eingegangen wird, soll deshalb festgelegt werden, was ein verteiltes System allgemein ist; gleichzeitig werden grundsätzliche Aufgaben und Terminologien eingeführt. Für den Begriff *verteiltes System* finden sich in der Literatur verschiedene Definitionen, wobei eine allgemein gehaltene Charakterisierung von Tanenbaum und van Steen gegeben wird (vgl. [TS03], S. 18):

Definition 2.1 (Verteiltes System)
Ein verteiltes System ist eine Menge voneinander unabhängiger Computer, die dem Benutzer wie ein einzelnes, kohärentes System erscheinen.

Das verteilte System hat demnach die Aufgabe, die autonomen, durch ein Netzwerk verbundenen Rechner mit lokaler Rechen- und Speicherfähigkeit derart zu verwalten, dass sie zu einem einzigen, virtuellen System verschmelzen. Dazu muss von der gegebenen Hardware abstrahiert werden; im Gegensatz zu einem herkömmlichen Betriebssystem müssen zudem die Rechnergrenzen transparent überwunden werden. Die Ressourcen des Systems müssen derart eingesetzt und verwaltet werden, dass die Anwendung des Systems effizient und zuverlässig ausgeführt wird. Dabei ergeben sich einige grundlegende Probleme, die ein verteiltes System – bzw. das Management eines verteilten Systems – lösen muss.

Eine Anwendung eines verteilten Systems besteht typischer Weise aus kooperierenden Prozessen, die gemeinsam die Lösung eines Problems berechnen. Die Kooperation dieser Prozesse muss sowohl für Prozesse eines Rechners, als auch über Stellengrenzen hinweg transparent durch Kommunikationsmechanismen ermöglicht werden. Üblich für verteilte Systeme sind in erster Linie Mechanismen zum Nachrichtenaustausch, d.h. das Management stellt Mechanismen bereit, mit Hilfe derer Prozesse ortstransparent – also ohne die Kenntnis auf welcher Stelle sich ein anderer Prozess befindet – Nachrichten an diesen senden können. Systeme, die ausschließlich den Versand von Nachrichten als Mittel der Kommunikation bereitstellen, werden in dieser Arbeit als *Nachrichtenkommunikationssysteme*[1] bezeichnet.

[1] engl.: message passing systems

Ein weiterer Mechanismus zur Kooperation in verteilten Systemen ist der entfernte Funktionsaufruf[2]. Dabei ruft ein Prozess eine Funktion eines Objekts ohne die Kenntnis über dessen Ort auf. Liegt das Objekt lokal auf der selben Stelle, so wird ein herkömmlicher Funktionsaufruf durchgeführt, liegt das Objekt entfernt auf einem anderen Rechner, so simuliert das Management einen lokalen Funktionsaufruf, indem der aufrufende Prozess blockiert, die Parameter serialisiert und übertragen, und auf dem anderen Rechner wieder deserialisiert werden. Dort wird die Funktion aufgerufen und anschließend das Ergebnis analog zurück gesendet. Nach dem Empfang wird der aufrufende Prozess entblockiert und erhält das Ergebnis der Funktion.

Eine dritte Möglichkeit zur Kooperation besteht im gemeinsamen Zugriff auf einen verteilten Adress- und Speicherraum[3]. Das Management eines Systems sorgt dafür, dass jeder Prozess – unabhängig auf welchem Rechner er ausgeführt wird – transparent auf den gemeinsamen Speicher zugreifen kann, als wäre dieser Speicher lokal vorhanden. Die Verwaltung und Übertragung der Speicher-Objekte unter Einhaltung eines festgelegten Konsistenzmodells obliegt dem Management des Systems.

Die Strategien der Kooperationsmechanismen sind in der Regel relativ stark durch die Anwendung bestimmt; beispielsweise ist durch die Anwendung festgelegt, wann eine Nachricht in einem Nachrichtenkommunikationssystem versandt wird. Gute Strategien zur Verwaltung anderer Ressourcen, wie beispielsweise der Rechenfähigkeit, also der Zuweisung von Prozessen zu Rechnern, und auf jedem Rechner wiederum das Scheduling, sind wesentlich schwieriger zu realisieren. Das Ziel sollte sein, die Kapazitäten optimal zu nutzen, und dadurch die bestmögliche Performanz für die Anwendung zu erzielen. Die Abhängigkeitsstrukturen in kooperierenden Anwendungen sind jedoch so komplex, dass in den meisten verteilten Systemen Standard-Entscheidungen, also anwendungsunabhängige Entscheidungen getroffen werden. Dies gilt generell für die Strategien zur Verwaltung der Ressourcen in den meisten verteilten Systemen. In den weiteren Abschnitten dieses Kapitels wird beschrieben, wie im Projekt MoDiS anwendungsorientierte Strategien zur Verwaltung der Ressourcen getroffen werden.

Eine weitere Aufgabe verteilter Systeme ist die Toleranz von Fehlern. Je mehr Rechner in einem verteilten System vernetzt sind, umso höher ist bereits die Wahrscheinlichkeit von Hardwarefehlern im System. Angenommen ein Rechner hat eine *mean-time-to-failure* (MTTF)[4] von einem Jahr, dann hat ein System be-

[2] engl.: remote procedure call (rpc)
[3] engl.: distributed shared memory (dsm)
[4] Die mean-time-to-failure gibt einen Durchschnittswert der Zeit an, nach der eine Komponente einen Fehler aufweist.

stehend aus 12 derartigen Rechnern eine MTTF von einem Monat, da im Schnitt jeder Rechner einmal im Jahr einen Hardwarefehler aufweist. Verteilte Systeme enthalten auf Grund der Kommunikation über die Vernetzung neue mögliche Fehlerursachen, die in einem einzelnen Rechner nicht auftreten können.

Besonders problematisch sind Softwarefehler, die in nahezu jedem größeren Softwareprojekt enthalten sind. Ein Problem in verteilten Systemen ist allerdings die erhöhte Komplexität der Software, sodass gerade Softwarefehler in Anwendung und System die Zuverlässigkeit von verteilten Systemen beeinträchtigen, und deshalb die Notwendigkeit von Maßnahmen zur Fehlertoleranz von Hard- und Softwarefehlern entsteht. Die Konzeption von Fehlertoleranzverfahren mit besonderem Augenmerk auf Softwarefehler sowie die Integration in das MoDiS-Projekt ist Kern der vorliegenden Arbeit.

Neben der allgemeinen Charakterisierung eines verteilten Systems aus Definition 2.1 sind weitere Forderungen an die zu konstruierenden Systeme sinnvoll, um ihren Nutzen zu erhöhen. Die Rechen- und Speicherfähigkeit des Systems müssen den Nutzern derart zur Verfügung gestellt werden, dass kooperative, parallele Problemlösungen einfach zu entwickeln und effizient zu berechnen sind. Dies erfordert die Spezifikation dieser Problemlösungen frei von technischen Details des Systems auf hohem Abstraktionsniveau. Verteilte Systeme, die diesen Anforderungen genügen, ermöglichen eine effiziente und geeignet nutzbare Informationsverarbeitung. Die Konzepte des MoDiS-Projekts wurden für diese Anforderungen entwickelt.

Folgend werden die Konzepte des Projekts MoDiS, in der für diese Arbeit notwendigen Tiefe, erläutert. Ausführlichere bzw. alternative Beschreibungen dieser Konzepte sind in [SEL+96], [EP99] und [Reh04] zu finden. Definitionen und Nomenklatur sind [Reh04] entnommen bzw. daher abgeleitet.

2.2 Konzepte zur Konstruktion verteilter nebenläufiger Systeme

Im Projekt MoDiS wird ein *Top-Down*-orientierter, *sprachbasierter Gesamtsystemansatz* zur Konstruktion verteilter Systeme verfolgt. Die Top-Down-Orientierung bedeutet, dass die Anforderungen zur Spezifikation einer verteilt ausgeführten Anwendung Ausgangspunkt zur Konstruktion des Systems waren und sind. Ein herkömmliches Vorgehen bei der Konstruktion von Betriebssystemen oder verteilten Systemen wäre entgegengesetzt, also Bottom-Up: die Hardware wird veredelt und Systemdienste zu deren Nutzung bereitgestellt, unabhängig davon,

welche Anforderungen die später auf dem System ausgeführten Anwendungen tatsächlich haben.

Aus den Anforderungen der Anwendungen wurden *Komponenten-Arten* und geeignete *Strukturen* entwickelt, um die Spezifikation von Anwendungen des verteilten Systems auf hohem Abstraktionsniveau zu spezifizieren. Die Sprache INSEL stellt in einer *Ada*-ähnlichen Syntax die Sprachmittel zu Spezifikation der Komponenten zur Verfügung. Das Management des verteilten Systems wird bei der Übersetzung zusammen mit der Anwendung erzeugt, sodass Anwendung und Management zusammen das gesamte verteilte System bilden.

Das *Management* eines Systems bezeichnet einerseits die Summe der strategischen Entscheidungen, die notwendig sind, um die abstrakte Spezifikation zu konkretisieren und auf der Hardwarekonfiguration zur Ausführung zu bringen sowie andererseits die Mechanismen zur Umsetzung dieser Entscheidungen. Das MoDiS-Management besteht aus verschiedenen Management-Anteilen, unter anderem dem Übersetzer der Sprache INSEL, der Laufzeitumgebung, darunter verschiedene Mechanismen zur Realisierung der Kommunikation von INSEL-Komponenten sowie einem klassischen Betriebssystemkern, der die Basisfunktionalität jedes Rechners bereitstellt.

Eine entscheidende Vorgehensweise des Managements im MoDiS-Projekt ist die Orientierung an der Anwendung. Sowohl Informationen, die statisch im Übersetzer gewonnen werden, als auch Laufzeitinformationen, die dynamisch gesammelt werden, nutzt das Management um geeignete Entscheidungen zur Ausführung der Anwendung zu treffen. *Geeignet* bedeutet der Anwendung angepasst. Dies impliziert in der Regel eine Optimierung der Performanz. Dies gilt allerdings nicht generell; die Ziele für eine anwendungsangepasste Ausführung müssen je nach Situation und Anwendung unterschiedlich interpretiert werden. Im Rahmen dieser Arbeit werden beispielsweise Entscheidungen im Management getroffen, die Anwendungsinformationen nutzen, um Fehler zu tolerieren.

2.2.1 Komponenten

MoDiS-Systeme sind aus Komponenten aufgebaut, für die *innen* und *aussen* festgelegt sind. Die Eigenschaften der enthaltenen Komponenten eines Systems sowie deren Abhängigkeiten legen die Eigenschaften des Systems im Ganzen fest. Durch das *Schachtelungsprinzip* wird festgelegt, welche Komponenten *innen* bzw. *aussen* zu einer Komponente sind, wobei alle Komponenten *innen* zur initialen Komponente sind. Eine Komponente ist durch Operationen nutzbar, wobei gemäß dem *Objektprinzip* zwischen äusseren und inneren Operationen unterschieden wird.

Entsprechend können Komponenten von aussen nur äussere Operationen einer Komponente nutzen, wobei eine Komponente selbst und innen liegende Komponenten auch innere Operationen der Komponente nutzen können.

Es werden *aktive* und *passive* Komponenten unterschieden. Aktive Komponenten werden *Akteure* genannt und haben sowohl Rechen- als auch Speicherfähigkeit. Aktive Komponenten sind mit Prozessen oder Threads herkömmlicher Betriebssysteme vergleichbar. Passive Komponenten besitzen ausschließlich Speicherfähigkeit und werden von den Akteuren genutzt.

Zur Kooperation von Akteuren werden zwei Mechanismen bereit gestellt. Einerseits können Akteure über die Veränderung des Zustands einer passiven Komponente kooperieren, was dem Prinzip eines geteilten Speicherbereichs entspricht. Andererseits können Akteure über synchrones, Operationen-orientiertes Rendezvous kommunizieren, dabei werden die beiden Ausführungsfäden zweier Akteure virtuell zusammengeführt um gemeinsam eine Funktion auszuführen, anschließend kann jeder Akteur wieder seine Berechnungen fortsetzen. Die Komponenten, insbesondere die Akteure werden dynamisch erzeugt und aufgelöst, sodass die Spezifikation paralleler Anwendungen auf hohem Abstraktionsniveau flexibel ermöglicht wird.

Die Konzepte von MoDiS zur Konstruktion verteilter Systeme wurden realisiert, indem Komponenten-Arten konzipiert wurden, die den Konzepten entsprechen. Die Eigenschaften der Komponentenarten legen die Konzepte zur Konstruktion, und somit die Möglichkeiten zur Spezifikation der verteilten Systeme fest. Einerseits bieten die Komponentenarten gewissen Eigenschaften, die die Spezifikation einer verteilten Anwendung bzw. eines verteilten Systems ermöglichen, andererseits sind Strukturen und Abhängigkeiten zwischen diesen Komponentenarten festgelegt, die einerseits das Gesamtverhalten des Systems festlegen, und andererseits dem Management Informationen zur Konkretisierung der Komponenten bieten. In Abbildung 2.1 sind alle verfügbaren Komponenten dargestellt.

Zunächst werden die Komponenten in DE- und DA-Komponenten untergliedert. DE-Komponenten bestehen lediglich aus einem Deklarationsteil und zeichnen sich dadurch aus, dass ihre inneren und äusseren Eigenschaften identisch sind. Bei den links in der Abbildung dargestellten DA-Komponenten unterscheiden sich innere und äussere Operationen, sie werden auch als *wesentliche* Komponenten bezeichnet.

DE-Komponenten werden weiter unterschieden in Generatoren und wertorientierte DE-Komponenten. Die zuletzt genannten sind einfach oder strukturierte Daten bzw. Zeiger, und haben demnach lediglich die Aufgabe, Werte zu spei-

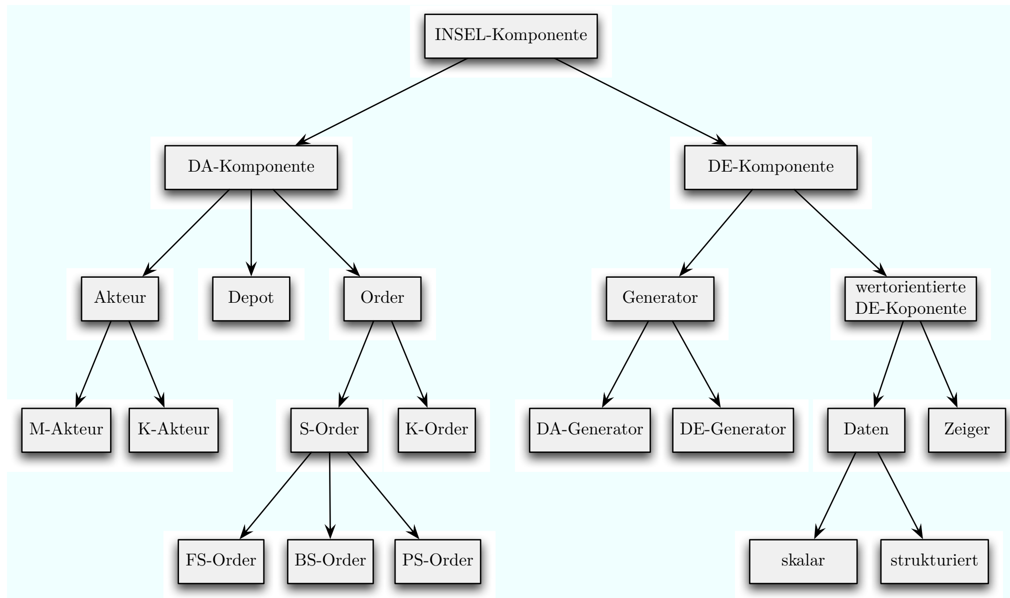


Abbildung 2.1.: Komponentenarten in MoDiS

chern. Diese können, falls die DE-Komponente zu einer Komponente innen liegt, von dieser gelesen oder verändert, bzw. bei Zeigern referenziert und dereferenziert werden.

Generatoren sind mit Klassen in objektorientierten Sprachen zu vergleichen. Sie bilden die Vorlagen für Instanzen aller Komponenten, wobei die Strukturierung der Instanzen mit der Strukturierung der Generatoren übereinstimmt. Für einen Generator ist die äussere Funktion `erzeuge`, durch die eine Instanz einer Komponente des Generators erzeugt wird, implizit definiert. DE-Generatoren entsprechen dabei dem Datentyp der wertorientierten DE-Komponente.

Die Ausführung eines Systems wird durch die DA-Komponenten bestimmt, die neben dem Deklarationsteil auch einen Anweisungsteil beinhalten. DA-Komponenten sind wiederum aus INSEL-Komponenten zusammengesetzt und besitzen mindestens eine äussere Operation, die festgelegt und je nach Komponentenart unterschiedlich benannt ist. Diese implizit definierte äussere Operation wird für die einzelnen DA-Komponenten getrennt erklärt. Zudem besitzt jede DA-Komponente eine innere Operation, die als *kanonische Operation* bezeichnet wird.

Die Ausführung der kanonischen Operation besteht aus der Erarbeitung des Deklarationsteils und der Ausführung des Anweisungsteils. Eine DA-Komponente

befindet sich stets in einem der Zustände V (vorbereitet), A (in Ausführung), R (rechenbereit), W (wartend) oder T (terminiert). DA-Komponenten werden (mit Ausnahme der initialen Komponente) dynamisch von anderen DA-Komponenten erzeugt, indem diese die `erzeuge`-Operation des Generators der Komponente aufrufen. Die neu erzeugte Komponente befindet sich dann im Zustand V; ihre implizit definierte äussere Operation wird ausgeführt und bewirkt folgende Schritte:

- Die Ein- und Ausgabeparameter werden der Komponente zugewiesen.
- Die Ausführung der Komponente wechselt von aussen nach innen, dabei geht der Zustand der Komponente von V über zu A bzw. R.
- Die kanonische Operation der Komponente wird ausgeführt und unterteilt sich wiederum in die in Abbildung 2.2 dargestellten Phasen.
- Nach Ausführung der kanonischen Operation befindet sich die Komponente in Zustand T, die Operationsausführung wechselt von innen nach aussen.

Die eigentliche Berechnung der DA-Komponente wird in der kanonischen Operation durchgeführt. Zuvor wird im Zustand V lediglich die Anfangssynchronisation mit Übergabe der Parameter vollzogen. Nach Abarbeitung der kanonischen Operation ist eine DA-Komponente terminiert, sie berechnet nichts mehr, ihre Ergebnisse stehen jedoch noch zur Verfügung, bis sie von ihrem Erzeuger[5] aufgelöst wird. Depots stellen hier eine Sonderrolle dar, da bei ihnen die kanonische Operation zur Initialisierung dient, und die Depots als Speicherobjekte erst im Zustand T von Akteuren als passive Speicherobjekte genutzt werden können. Die kanonische Operation ist in logische Phasen unterteilt, wie in Abbildung 2.2 dargestellt.

Zunächst wird in der Aufbauphase der Deklarationsteil einer DA-Komponente erarbeitet. Im Deklarationsteil werden lokal definierte Komponenten generiert, wobei häufig durch die Schachtelungsstruktur von den generierten Komponenten weitere Komponenten erzeugt werden. Anschließend folgt die Berechnungsphase, untergliedert in Hauptphase, Synchronisationsphase und Ergebnisphase. In der Hauptphase wird der Anweisungsteil mit Ausnahme der letzten Anweisung `return` (nur bei FS-Order) ausgeführt. In der Synchronisationsphase wird auf die Terminierung, also den Zustandsübergang in den Zustand T, aller von der Komponente erzeugten Akteure gewartet. Handelt es sich bei der Komponente

[5] Die Beziehung zwischen erzeugender und erzeugter Komponente wird in Abschnitt 2.2.2 zu den Systemstrukturen erklärt.

um eine FS-Order, so folgt noch eine `return`-Anweisung in der Ergebnisphase, in der Ausgabe- und Ein-/Ausgabeparameter zugewiesen werden. Nach Abarbeitung der kanonischen Operation geht die Komponente in den Zustand T über. Während die DA-Komponente sich im Zustand T befindet (für FS-Ordern bereits während der Ergebnisphase) können andere Komponenten die berechneten Ergebnisse nutzen; dann kann die Komponente aufgelöst werden. Die Übergabe der Ergebnisse und die Auflösung der Komponente werden als Abschlusssynchronisation bezeichnet.

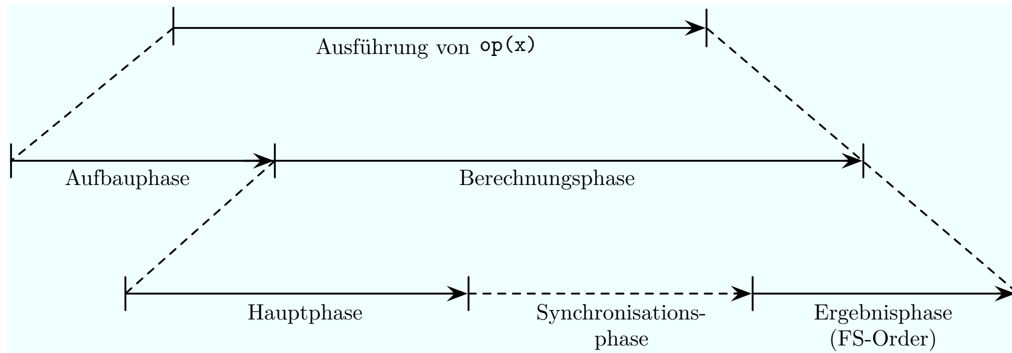


Abbildung 2.2.: Phasen der Ausführung der kanonischen Operation

Im Folgenden wird auf die Unterschiede und speziellen Eigenschaften der einzelnen DA-Komponenten sowie deren Einsatz eingegangen. Neben diesen Komponentenarten und deren generellen Eigenschaften stellt die Sprache INSEL zur Spezifikation des Anweisungsteils zu imperativen Sprachen vergleichbare Sprachkonstrukte bereit. Darunter fallen unter anderem Zuweisungen, arithmetische Operationen, bedingte Anweisungen, Schleifen sowie Ein- und Ausgabefähigkeit. Bevor die einzelnen Komponenten erklärt werden, muss noch auf die Identifikation der Komponenten eingegangen werden.

2.2.1.1 Identifikation

Komponenten müssen eindeutig identifizierbar sein, um von anderen Komponenten benutzt werden zu können. In MoDiS existieren *benannte* Komponenten, die auch als *N-Komponenten* bezeichnet werden. N-Komponenten werden erzeugt, indem Deklarationen ihrer Komponentenklasse erarbeitet werden; dabei wird ihnen ein im Kontext eindeutiger Name zugewiesen.

Wird eine Komponente über die *erzeuge*-Operation ihres Generators instantiiert, so wird sie einem Zeiger zugewiesen und ist über diesen identifizierbar. Diese

Komponenten werden als *anonyme* oder *Z-Komponenten* bezeichnet. Komponenten, die keine äussere Operation anbieten, müssen nicht von anderen Komponenten identifiziert werden können. Sie werden deshalb als *nicht explizit benannte* Komponenten bezeichnet. Dazu gehören M-Akteure und Order-Inkarnationen.

2.2.1.2 Ordern

Ordern sind vergleichbar mit Unterprogrammen in imperativen Programmiersprachen, und dienen in erster Linie der Strukturierung und Wiederverwendung des Codes. Jede Order besitzt nur eine einzige äussere Operation `führe_aus`, die implizit definiert ist. Durch Ausführung dieser Operation geht die Order von Zustand V in A über und führt ihre kanonische Operation aus, die den wesentlichen Teil der Order ausmacht. Nach Ausführung der kanonischen Operation geht die Order in den Zustand T über.

Wie in Abbildung 2.1 dargestellt, werden Ordern weiter in S- und K-Ordern untergliedert. K-Ordern sind spezielle Kommunikations-Ordern, die zur Realisierung des Operationen-orientierten Rendezvous zwischen Akteuren genutzt werden. K-Ordern können nur innerhalb K-Akteuren erzeugt werden. Auf die Funktionalität der K-Ordern und des Operationen-orientierten Rendezvous wird in Abschnitt 2.2.1.4 zu K-Akteuren eingegangen.

S-Ordern werden untergliedert in PS-, FS- und BS-Ordern. Diese sind mit Prozeduren, Funktionen und Blöcken in imperativen Programmiersprachen direkt vergleichbar. BS-Ordern dienen in erster Linie der Synchronisation: werden beispielsweise zwei Akteure in einer BS-Order erzeugt, so ist sichergestellt, dass die BS-Order erst terminiert (also der Code im Anschluss an die BS-Order ausgeführt wird), wenn beide erzeugten Akteure terminiert sind. PS- und FS-Ordern dienen der Strukturierung durch Unterprogramme, wobei nur FS-Ordern einen expliziten Rückgabewert haben, der in der Ergebnisphase der kanonischen Operation zugewiesen wird (vgl. Abbildung 2.2).

2.2.1.3 Depots

Depots sind passive Komponenten mit Speicherfähigkeit. Sie sind mit Objekten aus objektorientierten Sprachen vergleichbar. Die implizit definierte äussere Operation `initialisiere` führt die kanonische Operation des Depots aus. Die kanonische Operation eines Depots ist mit einem Konstruktor vergleichbar. Im Anweisungsteil wird das Depot im Sinne seiner innen liegenden Komponenten initialisiert, und erst nach Ausführung der kanonischen Operation, also im Zu-

stand T, ist das Depot als Speicherobjekt benutzbar. Es können beliebig weitere äussere Operationen, beispielsweise in Form von FS-, PS-Ordern oder auch Akteuren, im Depot definiert sein. Über diese Operationen kann eine zum Depot aussen liegende Komponente auf das Depot zugreifen. Dies ermöglicht eine unsynchronisierte, mittelbare Kommunikation zwischen Akteuren. Zudem können auch Depot-lokale S-Ordern zur Strukturierung definiert werden, die nicht von aussen aufgerufen werden können.

Im Rahmen dieser Arbeit werden in Anlehnung an die Nomenklatur objektorientierter Sprachen die lokalen Ordern auch als *privat*, die äusseren als *exportiert* bezeichnet. Das Schlüsselwort `sync` veranlasst den synchronisierten Zugriff auf Depots, vergleichbar zu *synchronized*-Methoden der Sprache Java.

2.2.1.4 Akteure

Akteure sind aktive Komponenten, sie besitzen sowohl Speicher- als auch Rechenfähigkeit und sind im wesentlichen mit Prozessen oder Threads herkömmlicher Systeme vergleichbar. Akteure werden dynamisch erzeugt, parallel ausgeführt und anschließend nach ihrer Terminierung implizit gemäß den Systemstrukturen, auf die später noch eingegangen wird, aufgelöst. Das Konzept eines Akteurs ermöglicht die Spezifikation paralleler Kontrollflüsse im System auf hohem Abstraktionsniveau. Für jeden Akteur ist die äussere Operation `starte` implizit definiert. Bei ihrem Aufruf wird der Akteur vom Zustand V in den Zustand R überführt und seine kanonische Operation ausgeführt.

Ein Akteur kann in seiner kanonischen Operation sowohl weitere Akteure, als auch andere Komponenten, wie Depots oder Ordern, erzeugen. Die Abarbeitung der passiven Komponenten geschieht sequentiell in die Ausführung des Akteurs eingeordnet, wohingegen ein erzeugter Akteur parallel ausgeführt wird. Für passive Komponenten ist die Ausführung deren kanonischer Operation mit einem Unterprogrammaufruf in einer imperativen Sprache vergleichbar. Die Komponente, deren kanonische Operation vom Ausführungsfaden des Akteurs zu einem Zeitpunkt ausgeführt wird, wird als *Ausführungskomponente* des Akteurs bezeichnet.

M-Akteure. Ist die implizite äussere Operation `starte` die einzige äussere Operation eines Akteurs, so wird er als M-Akteur (mono-operational) bezeichnet. M-Akteure bieten keine Möglichkeit, ihre Ausführung unmittelbar zu beeinflussen. Sie können sich lediglich aktiv (durch einen entsprechenden Aufruf) mit ihrem Pendant, den K-Akteuren, synchronisieren, oder über Datenobjekte, wie beispielsweise Depots, koordinieren. M-Akteure können ausschließlich im Anwei-

sungsteil von DA-Komponenten als nicht explizit benannte Komponenten erzeugt werden.

K-Akteure. K-Akteure bieten im Gegensatz zu M-Akteuren explizit weitere äussere Operationen in Form von K-Order-Generatoren an. Vereinfacht kann man sagen, dass die K-Ordern Funktionen darstellen, deren Aufruf andere Akteure anstoßen können. Dabei handelt es sich um eine Ausführung nach dem Operationen-orientierten Rendezvous, also einer synchronen Funktionsausführung.

Ein M- oder K-Akteur a_m ruft eine K-Order eines K-Akteurs auf und wird dadurch in den Zustand W überführt. Der K-Akteur kann nur durch die Ausführung einer `select`- oder `accept`-Anweisung einen K-Order-Aufruf eines Akteurs annehmen. Die K-Order selbst wird also synchron ausgeführt. Sie wird vom K-Akteur, bzw. vom K-Order-Generator des K-Akteurs erzeugt und sequentiell in die Ausführung des K-Akteurs eingeordnet. Die K-Order ist demnach die Ausführungskomponente des K-Akteurs. Die K-Order wird – analog zu S-Ordern – ausgeführt und nach Abarbeitung ihrer kanonischen Operation wird das Ergebnis dem aufrufenden M- oder K-Akteur zur Verfügung gestellt. Dieser wird in den Zustand R überführt und kann die Ausführung seiner kanonischen Operation fortführen. Der K-Akteur, dessen Ausführungskomponente die K-Order war, kann nach deren Terminierung ebenfalls seine Berechnungen direkt nach der entsprechenden `select`- oder `accept`-Anweisung im Code fortsetzen.

Die K-Order dient also der Synchronisation zweier Akteure, und erlaubt die Kooperation durch Datenaustausch, also die Ein- und Ausgabeparameter der K-Order. Die Kommunikation mittels Operationen-orientiertem Rendezvous stellt damit eine spezielle, eingeschränkte und genau definierte Form der Nachrichtenkommunikation dar. In MoDiS werden insgesamt zwei Mechanismen zur Kommunikation von Akteuren angeboten, einerseits der Zugriff auf gemeinsamen Speicher in Form von Depots, andererseits die synchronisierte Nachrichtenkommunikation durch das Operationen-orientierte Rendezvous.

2.2.2 Systemstrukturen

Mit den beschriebenen Komponentenarten lassen sich parallele, kooperative Anwendungen auf hohem Abstraktionsniveau spezifizieren. Die beschriebenen Eigenschaften der Komponenten zeigen bereits, dass strukturelle Abhängigkeiten zwischen diesen bestehen. Im Gegensatz zu vielen verteilten Systemen sind die strukturellen Abhängigkeiten der MoDiS-Komponenten wohl definiert und festgelegt. Sie werden durch die Nutzung der Komponenten implizit festgelegt und

können vom Management als Informationsquelle zur Konkretisierung des Systems sowie zum Ressourcenmanagement genutzt werden. Einige der Strukturen hängen direkt von der Schachtelung der Komponenten in der Spezifikation ab, da die Schachtelung innen und aussen und damit die Sichtbarkeit bzw. Zugriffsmöglichkeiten bestimmt. Jedes MoDiS-System startet mit einem M-Akteur als *Wurzel-* oder *Hauptkomponente*. Dieser Akteur bzw. sein Generator ist dementsprechend in der Schachtelung der Spezifikation alles andere umschließend. Die einzelnen Strukturen sowie die Regeln zur jeweiligen Einordnung der Komponentenarten werden im folgenden knapp erklärt.

2.2.2.1 Definitionsstruktur δ

Die Definitionsstruktur δ legt für die gesamte Lebenszeit einer Komponente bzw. einer Inkarnation i eines Generators fest, welche Komponenten von i benutzbar sind. Die DA-Komponente *da*, die den Generator von i als lokale N-Komponente enthält, ist dafür ausschlaggebend. Alle Komponenten, die innerhalb von *da* definiert sind, die also für *da innen* sind, sind auch von i nutzbar. Diesen Zusammenhang zwischen i und *da* stellt die zweistellige Relation *unmittelbar-δ-innen* auf der Menge der DA-Komponenten her.

Definition 2.2 (δ-Struktur)
Sei X_t die Menge der DA-Komponenten eines Systems zum Zeitpunkt t und `da`, `i` $\in X_t$. `i` *ist definitorisch abhängig von* `da` – $(\texttt{i}, \texttt{da}) \in \tilde{\delta}_t$ – *gdw. der Generator* `g` *von* `i` *im Deklarationsteil von* `da` *als lokale N-Komponenten definiert ist.*
Die Relation δ-innen*, kurz δ, ist die transitive Hülle der Relation unmittelbar-δ-innen.*

Die Relation δ bildet zu jedem Zeitpunkt des Systems einen Baum mit der Hauptkomponente als Wurzel. Alle Komponenten werden bei ihrer Erzeugung gemäß der Schachtelungsstruktur in δ ein-, und bei ihrer Auflösung wieder ausgeordnet. Die durch δ festgelegte Menge benutzbarer oder *sichtbarer* Komponenten legt demnach die Ausführungsumgebung einer Komponente fest.

2.2.2.2 Ausführungsstruktur α

Die Ausführungsstruktur α beschreibt die Kontrollflüsse des Systems. Dies entspricht drei Teilaspekten, den parallelen Kontrollflüssen, der sequentiellen Einbettung in einen Kontrollfluss sowie den kommunikationsbedingten Abhängigkeiten.

Die Ausführungsstruktur α ist die transitive Hülle der Relation *unmittelbar-α-innen* und setzt sich aus den drei Relationen σ, π und κ zusammen.

Die Relation σ gibt die sequentielle Einordnung wieder, wenn also der Ausführungsfaden eines Akteurs die kanonische Operation einer anderen Komponente ausführt, was bisher als *Ausführungskomponente* bezeichnet wurde:

Definition 2.3 (σ-Struktur)
Sei `i` $\in X_t$ *eine S-Order, K-Order oder ein Depot im Zustand A.* `i` *ist unmittelbar sequentiell abhängig von* `da` $\in X_t$ – $(\mathtt{i}, \mathtt{da}) \in \tilde{\sigma}_t$ – *gdw.* `i` *eine S-Order ist und bei Ausführung der kanonischen Operation von* `da` *erzeugt worden ist bzw.* `i` *eine K-Order des K-Akteurs* `da` *ist, und die kanonische Operation von* `i` *sequentiell eingeordnet in die Ausführung der kanonischen Operation von* `da` *ausgeführt wird. Die Relation σ-*innen*, kurz σ, ist die transitive Hülle der Relation unmittelbar-σ-innen.*

S-Ordern sind während ihrer gesamten Lebenszeit in die σ-Struktur eingeordnet, K-Ordern und Depots dagegen nicht. Bei K-Ordern fallen auf Grund der Rendezvous-Semantik die Erzeugung und der Beginn der Ausführung nicht zusammen. Depots sind nur in die σ-Relation eingeordnet bis sie in den Zustand T übergehen. In diesem Zustand sind sie benutzbar, leisten jedoch keinen Beitrag mehr zur Ausführung.

Die Einordnung paralleler Kontrollflüsse wird durch die Relation *unmittelbar parallel abhängig* erfasst, die die Beziehung zwischen erzeugendem und erzeugtem Akteur erfasst.

Definition 2.4 (π-Struktur)
Sei `m` $\in X_t$ *ein M-Akteur*[6]. `m` *ist unmittelbar parallel abhängig von der Komponente* `da` $\in X_t$ – $(\mathtt{m}, \mathtt{da}) \in \tilde{\pi}_t$ – *gdw.* `m` *von* `da` *erzeugt wurde.* π_t *ist die transitive Hülle von* $\tilde{\pi}_t$.

Die Abhängigkeit zwischen Auftraggeber und Auftragnehmer der Kooperation durch Operationen-orientiertes Rendezvous ist die dritte Komponente der Ausführungsstruktur α und wird durch die Kommunikationsstruktur κ erfasst.

Definition 2.5 (κ-Struktur)
Seien `kord`, `da` $\in X_t$, `kord` *sei eine K-Order.* `kord` *ist unmittelbar kooperationsabhängig von* `da` – $(\mathtt{kord}, \mathtt{da}) \in \tilde{\kappa}_t$ – *gdw. es einen Akteur* `caller` $\in X_t$ *gibt, der Auftraggeber für* `kord` *ist und dessen Ausführungskomponente* `da` *ist.*

[6] Für K-Akteure müssen Sonderfälle beachtet werden. Details hierzu sind in [SEL+96] beschrieben.

Beim Operationen-orientierten Rendezvous werden demnach zweierlei Abhängigkeiten innerhalb der α-Struktur erfasst. Die Ausführung der K-Order im Ausführungsfaden des K-Akteurs wird in der Relation σ in Bezug gesetzt. Die Abhängigkeit von Auftraggeber zu K-Order wird durch κ erfasst. Die Ausführungs-Relation α fasst die drei Relationen σ, π und κ zusammen.

Definition 2.6 (α-Struktur)
Seien $\texttt{da}_1, \texttt{da}_2 \in X_t$*. Die Ausführung von* $\texttt{da}_1$ *ist unmittelbar-α-innen zu* $\texttt{da}_2$ – $(\texttt{da}_1, \texttt{da}_2) \in \tilde{\alpha}_t$ – *gdw.* $(\texttt{da}_1, \texttt{da}_2) \in \tilde{\sigma}_t \vee (\texttt{da}_1, \texttt{da}_2) \in \tilde{\pi}_t \vee (\texttt{da}_1, \texttt{da}_2) \in \tilde{\kappa}_t$.
α_t *ist die transitive Hülle von* $\tilde{\alpha}_t$*.*

Die Ausführungsabhängigkeiten, die in α erfasst sind, bilden eine der grundlegenden Informationsquellen, die für Managemententscheidungen zum angepassten Ressourcenmanagement sowie für das in dieser Arbeit vorgestellte Fehlertoleranzkonzept genutzt werden.

2.2.2.3 Lokalitätsstruktur λ

Ein benanntes Depot oder ein benannter K-Akteur werden erzeugt, indem ihre Deklaration im Rahmen der Ausführung einer kanonischen Operation einer DA-Komponente erarbeitet wird. Diese eindeutige Zuordnung wird in der Lokalitätsstruktur festgehalten.

Definition 2.7 (λ-Struktur)
Seien $\texttt{bdk}, \texttt{da} \in X_t$*. Sei* $\texttt{bdk}$ *ein benanntes Depot im Zustand T oder ein benannter K-Akteur.* $\texttt{bdk}$ *ist* unmittelbar-λ-innen *zu* $\texttt{da}$ – $(\texttt{bdk}, \texttt{da}) \in \tilde{\lambda}_t$ – *gdw.* $\texttt{bdk}$ *im Rahmen der Ausführung der kanonischen Operation von* $\texttt{da}$ *erzeugt wurde und* $\texttt{bdk}$ *lokale N-Komponente von* $\texttt{da}$ *ist.*
λ_t *ist die transitive Hülle von* $\tilde{\lambda}_t$*.*

Mit Hilfe der Lokalitätsstruktur λ kann die Lebenszeitstruktur definiert werden.

2.2.2.4 Lebenszeitstruktur ϵ

In Systemen mit dynamischer Erzeugung und Auflösung von Komponenten muss folgender Zusammenhang sichergestellt werden: existiert eine Komponente da_1, die eine Komponente da_2 benutzen kann, so folgt die Existenz von da_2. Das bedeutet, dass eine Komponente da_2 nicht nach ihrer Terminierung direkt aufgelöst

werden darf, sondern die Lebenszeit und damit die Auflösung durch diejenigen Komponenten festgelegt ist, die da_2 benutzen können. Diese Abhängigkeit wird durch die Lebenszeitstruktur ϵ definiert. Dazu muss unterschieden werden, wann eine Komponente benutzbar ist. Bei S-Ordern, benannten Depots im Zustand A und M-Akteuren gibt die σ- bzw. π-Einordnung an, bei welcher Komponente sie einen sequentiellen bzw. parallelen Beitrag zur Ausführung deren kanonischen Operation leisten. Dieser Komponente müssen sie auch bzgl. der Lebenszeit zugeordnet werden. Benannte Depots im Zustand T und benannte K-Akteure sind nur solange benutzbar, wie die Komponente existiert, bei der sie gemäß der Lokalitätsstruktur λ eingeordnet sind. Die Lebenszeit einer K-Order ist von der Lebenszeit ihres Auftraggebers, bei dem sie in der κ-Relation eingeordnet ist, abhängig.

Anonyme Depots und anonyme K-Akteure werden durch Zeiger identifiziert und benutzt. Diese Zeiger können weitergereicht, dupliziert und aufgelöst werden, somit ist die Komponente, die den Zeigergenerator enthält, für die Lebenszeit entscheidend. Diese Abhängigkeit wird durch die γ-Abbildung festgelegt.

Definition 2.8 (γ-Abbildung)
Sei `anonym` $\in X_t$ *eine anonyme Komponente,* $\texttt{g}_{\texttt{anonym}} \in X_t$ *der Generator für* `anonym`*,* `zg` $\in X_t$ *ein mit* $\texttt{g}_{\texttt{anonym}}$ *qualifizierter Zeigergenerator und* `z` $\in X_t$ *eine Inkarnation bezüglich* `zg`*.* $\gamma_t(\texttt{anonym})$ *ist diejenige DA-Komponente, die den Zeigergenerator* `zg` *als benannte lokale Komponente enthält.*

Mit Hilfe dieser Abbildung und den bisher eingeführten Strukturen kann nun die Lebenszeitstruktur ϵ definiert werden.

Definition 2.9 (ϵ-Struktur)
Seien $\texttt{da}_1, \texttt{da}_2 \in X_t$*.* $\texttt{da}_1$ *ist* unmittelbar-ϵ-innen *zu* $\texttt{da}_2$ – $(\texttt{da}_1, \texttt{da}_2) \in \tilde{\epsilon}_t$ – *gdw. gilt:*

- $(\texttt{da}_1, \texttt{da}_2) \in \tilde{\sigma}_t$*, falls* $\texttt{da}_1$ *S-Order oder benanntes Depot im Zustand A ist.*
- $(\texttt{da}_1, \texttt{da}_2) \in \tilde{\pi}_t$*, falls* $\texttt{da}_1$ *M-Akteur ist.*
- $(\texttt{da}_1, \texttt{da}_2) \in \tilde{\lambda}_t$*, falls* $\texttt{da}_1$ *benanntes Depot im Zustand T oder benannter K-Akteur ist.*
- $(\texttt{da}_1, \texttt{da}_2) \in \tilde{\kappa}_t$*, falls* $\texttt{da}_1$ *K-Order ist.*
- $\texttt{da}_2 = \gamma_t(\texttt{da}_1)$*, falls* $\texttt{da}_1$ *anonymes Depot oder K-Akteur ist.*

ϵ_t *ist die transitive Hülle von* $\tilde{\epsilon}_t$*.*

DA-Komponenten sind von Erzeugung bis Auflösung in die Lebenszeitstruktur eingeordnet. Somit kann sichergestellt werden, dass keine Komponente aufgelöst wird, wenn noch abhängige Komponenten existieren. Die Lebenszeitstruktur stellt eine Baumstruktur über alle Komponenten mit der Hauptkomponente als Wurzel dar.

2.2.3 Konkretisierung

Eingangs wurde schon erwähnt, dass die MoDiS-Konzepte einen Top-Down Ansatz verfolgen. Neben der Bereitstellung der beschriebenen Komponenten mit ihren klar definierten Eigenschaften und Strukturen, wird das Top-Down-Prinzip in der Konkretisierung der Systeme angewandt. Die beschriebenen Komponenten ermöglichen die Spezifikation einer kooperativen Problemlösung auf hohem Abstraktionsniveau, unabhängig von technischen Details des realen verteilten Systems, bzw. der Hardwarekonfiguration. Die schrittweise Verfeinerung des abstrakt spezifizierten Systems durch Hinzunahme von technischen Details bzw. Abbildung auf vorhandene Ressourcen und bereit gestellte Mechanismen wird als *Konkretisierung* bezeichnet.

Bei der Konkretisierung wird das abstrakte System *schrittweise* in ein konkretes System transformiert und auf der vorhandenen Hardwarekonfiguration zur Ausführung gebracht. Alle Entscheidungen, die vom MoDiS-Management zur Konkretisierung getroffen werden müssen, werden als *Konkretisierungs-Entscheidungen* bezeichnet. Konkretisierungs-Entscheidungen umfassen sowohl Entscheidungen zur Übersetzungszeit, bei denen die abstrakt spezifizierten Komponenten in ausführbaren Code transformiert werden, als auch Entscheidungen zur Laufzeit, wie beispielsweise Strategien zum Ressourcen-Management.

Im Gegensatz zu herkömmlichen Ansätzen der Systemkonstruktion werden Informationen, die statisch und dynamisch über die Anwendung gewonnen werden können, zum Treffen anwendungsorientierter Konkretisierungs-Entscheidungen genutzt. Anstatt uniforme Standard-Strategien einzusetzen, die im Mittel gut, für einzelne Anwendungen aber beliebig schlecht sein können, werden die Konkretisierungs-Entscheidungen für die jeweils spezifizierte Anwendung getroffen. Natürlich muss dazu sowohl ein Repertoire an Möglichkeiten zur Transformation bei den Verfeinerungsschritten als auch ein Repertoire an Strategien zur Ressourcenverwaltung zur Verfügung stehen.

Anhand der Anwendungsinformationen kann aus dem verfügbaren Repertoire die geeignete Lösung ausgewählt werden, wobei einerseits statische Informationen, aber auch dynamische Informationen, wie der aktuelle Systemzustand, oder

die zuletzt getroffenen Entscheidungen des Managements, berücksichtigt werden. Die Flexibilität dieses Ansatzes ermöglicht zudem, die Ziele der Entscheidungen dynamisch zu ändern und diese Änderung in die Konkretisierungs-Entscheidungen einfließen zu lassen. Grundsätzlich ist die Performanz des Systems eines der Ziele, die in den Konkretisierungs-Entscheidungen im Vordergrund steht. Im Rahmen dieser Arbeit wird noch gezeigt, wie im Fall eines Fehlers zur Ausführungszeit die Toleranz des Fehlers höher priorisiert werden kann, und somit – für einen begrenzten Zeitraum der Ausführung – das Management mit dem Ziel der Fehlertoleranz andere Konkretisierungs-Entscheidungen trifft, als mit dem Ziel der Performanzoptimierung.

Die flexible Nutzung der statischen und dynamischen Anwendungsinformationen, die zur Übersetzungs- und Laufzeit gewonnen werden, ist nur möglich, wenn das Management und die Anwendung eng miteinander verknüpft sind, was wiederum der Gesamtsystemansatz in MoDiS ermöglicht. Aus der abstrakten Spezifikation der parallelen, kooperativen Problemlösung wird ein System erzeugt, indem Management-Anteile und Anwendungsanteile verschmelzen. Durch diese enge Kopplung werden einerseits bereits statisch zur Übersetzungszeit grundsätzliche Entscheidungen getroffen und das Management diesbezüglich generiert. Andererseits kann so dem Management möglichst elegant eine Fülle an Informationen über die Anwendung zur Verfügung gestellt werden.

Das Management des Systems, von dem bisher nur abstrakt gesprochen wurde, ist unterteilt in logisch zusammengehörende Teile. Grundsätzlich ist das *generative Management*, das die Transformation vor dem Start der Ausführung übernimmt, vom *Laufzeit-Management* zu unterscheiden, das alle Entscheidungen zur Ausführungszeit trifft. Bei genauerer Betrachtung lässt sich das generative Management weiter unterteilen in den Übersetzer und den Binder. Das Laufzeit-Management dagegen in dedizierte Manager, die nur für ausgewählte Komponenten zuständig sind sowie gemeinsame Anteile, die wiederum nach ihrer Aufgabe in stellenlokale und stellenübergreifende Managementanteile zu gliedern sind. Im Rahmen dieser Arbeit ist diese feingranulare Unterscheidung jedoch nicht notwendig, deshalb wird im wesentlichen nur zwischen Laufzeit-Management und generativem Management unterschieden. Muss beim generativen Management zwischen Übersetzer und Binder unterschieden werden, so wird darauf explizit hingewiesen.

2.2.4 Systembeschreibung

Im Rahmen dieser Arbeit sind zur Erklärung abstrakte Sichten auf Ausführungsabläufe verteilter Systeme sowie Modelle zu deren Beschreibung notwendig. In diesem Abschnitt werden einerseits Abkürzungen und verwendete Zeichen zur Beschreibung der Systeme eingeführt, die an die Bedürfnisse der Arbeit angepasst sind und sich daher zum Teil von den Beschreibungen anderer Arbeiten über MoDiS sowie von den Abkürzungen im vorhergehenden Abschnitt unterscheiden[7]. Andererseits werden Modelle und Beschreibungen für Ausführungen verteilter Systeme eingeführt, die in den späteren Kapiteln zur Veranschaulichung genutzt werden.

Die nichtleere, endliche Menge aller Akteure eines Systems in Ausführung wird mit A bezeichnet[8], einzelne Akteure werden unten indiziert unterschieden, beispielsweise $a_m \in A$. Die Menge aller K-Akteure wird mit $A_k \subset A$ bezeichnet.

Die Ausführung von Akteuren lässt sich allgemein durch Ereignisse und Abhängigkeiten zwischen diesen geeignet beschreiben. Die Granularität dieser Ereignisse kann dabei unterschiedlich ausfallen, beispielsweise können einzelne Instruktionen oder aber zusammengehörende Abschnitte als Ereignisse betrachtet werden. Ereignisse, die sowohl in der Wirkung als auch in der Zeit atomar sind, repräsentieren die feinste Granularität. Diese Ereignisse können allerdings zusammengesetzt betrachtet, und somit die Granularität vergröbert werden. Entscheidend ist, dass die Granularität der Ereignisse gemäß der betrachteten Abhängigkeiten gewählt wird. Im Rahmen dieser Arbeit werden Ereignisse in relativ grober Granularität betrachtet, um auf hohem Abstraktionsniveau die Ausführung von Systemen zu analysieren und zu beschreiben. Die konkreten Ereignisse sind an der Implementierung der MoDiS-Konzepte im MoDiS-Experimentalsystem orientiert und werden an den jeweiligen Stellen beschrieben. Die Menge aller Ereignisse der Ausführung eines Akteurs $a_m \in A$ ist E_{a_m}. Die Vereinigungsmenge aller Ereignisse jedes Akteurs im System ergibt die Menge E aller Ereignisse im System:

$$E = \bigcup_{\forall a_m \in A} E_{a_m}.$$

In einzelnen Fällen werden weitere Teilmengen der Ereignisse betrachtet, die dann explizit erklärt werden. Die Ereignisse eines Akteurs $a_m \in A$ sind total geordnet

[7] Die Nomenklatur des bisherigen Abschnitts hat sich zur Wahrung der Konsistenz mit anderen Arbeiten an den gängigen Abkürzungen und Zeichen orientiert.

[8] Zur Übersichtlichkeit wird nicht explizit der Zeitpunkt unterschieden, es geht jedoch stets aus dem Kontext hervor, ob die zu einem Zeitpunkt existierenden Akteure, oder alle Inkarnationen über die gesamte Systemlaufzeit gemeint sind.

und werden entsprechend ihrer Ordnung oben indiziert: $E_{a_m} = \{e^1_{a_m} \dots e^n_{a_m}\}$. Ist die Zugehörigkeit zum Akteur bzw. die Indizierung innerhalb des Akteurs irrelevant, so wird diese z.T. weggelassen. Die totale Ordnung der Ereignisse eines Akteurs ist in seiner sequentiellen Abarbeitung begründet, eine genauere Erklärung folgt in Kapitel 4.1, an dieser Stelle soll lediglich die Nomenklatur und Modellierung eingeführt werden.

Durch Kooperation und Erzeuger-Erzeugter-Beziehungen entstehen Abhängigkeiten zwischen Akteuren. Werden die Ereignisse der Akteure in der Granularität entsprechend dieser Abhängigkeiten betrachtet, so können die Abhängigkeiten den Ereignissen der Akteure zugeordnet werden. In Kapitel 4.1 werden die Abhängigkeiten der Akteure, die für diese Arbeit betrachtet werden müssen, detailliert erklärt. An dieser Stelle reicht es aus, dass zwischen einzelnen Ereignissen Abhängigkeiten existieren, die über die totale Ordnung der Ereignisse eines Akteurs hinaus eine partielle Ordnung aller Ereignisse im System festlegen. Diese partielle Ordnung, bestehend aus totaler Ordnung der Ereignisse jedes Akteurs und zusätzlichen Abhängigkeiten zwischen einzelnen Ereignissen verschiedener Akteure, kann als gerichteter Graph visualisiert werden. In Abbildung 2.3 ist ein solcher Ereignisgraph dargestellt, wobei die Knoten Ereignisse repräsentieren, und die partielle Ordnung durch die Kanten dargestellt wird.

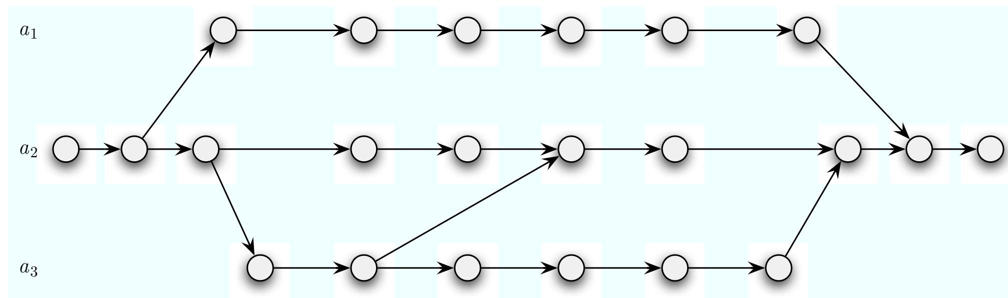


Abbildung 2.3.: Gerichteter Graph der partiellen Ordnung der Ereignisse eines Systems

Im Lauf der Erklärungen der vorliegenden Arbeit werden verschiedene Ereignisgraphen betrachtet, die sich in Granularität und eingehenden Abhängigkeiten unterscheiden. Die entsprechenden Abweichungen von diesem allgemeinen Graphen werden an den entsprechenden Stellen erklärt. Wird die Granularität der betrachteten Ereignisse und Abhängigkeiten derart gewählt, dass für jede Komponente Erzeugung und Auflösung sowie der Übergang von Zustand V zu A und von Zustand A zu T als einzige Ereignisse zusammen mit der ϵ-Struktur als Abhän-

gigkeiten eingehen, so entspricht die partielle Ordnung einer MoDiS-Ausführung einem Ereignisverband[9].

Verbände haben auf Grund ihrer Struktur bei der Analyse und Beschreibung von Systemabläufen den Vorteil, dass sehr gut von Teilen der Systemausführung in Raum oder Zeit abstrahiert werden kann, indem ganze Teile des Verbands auf ihr gemeinsames Infimum und Supremum zusammengeschrumpft werden. Die Ordnung der Ereignisse des in Abbildung 2.3 dargestellten Ereignisgraphs erfüllt die Eigenschaften eines Verbands. In Abschnitt 2.3.1 über die Realisierung der MoDiS-Konzepte im aktuellen Experimentalsystem wird erläutert, wie bereits zur Übersetzungszeit ein Ereignisverband mit bestimmtem Abstraktionsgrad generiert wird, der später für Entscheidungen des Laufzeit-Managements als Informationsquelle herangezogen wird.

2.2.5 INSEL

Die Sprache INSEL stellt die erklärten Komponenten und Operationen in einer *Ada*-ähnlichen Syntax zur Verfügung. Die Sprache INSEL ist objektbasiert, d.h. im Gegensatz zu modularen Programmiersprachen werden die Operationen, die auf einen Datentyp anwendbar sind, zusammen mit diesem Datentyp definiert. Unterstützt eine Sprache zusätzlich Vererbung, so wird sie als *objektorientierte Sprache* bezeichnet, dies ist in INSEL allerdings nicht der Fall. Details zur Sprache INSEL werden in der vorliegenden Arbeit an den jeweiligen Stellen soweit notwendig erklärt. Ein ausführlicher Bericht über die Sprache INSEL und deren Syntax ist in [RW96] zu finden. Ein einfaches INSEL-Programm ist in Anhang A aufgelistet.

2.3 MoDiS-Experimentalsystem

Während der Forschungsarbeiten zu den Konzepten zur Konstruktion verteilter, nebenläufiger Systeme wurden verschiedene experimentelle Implementierungen erarbeitet, um die Erstellung der Systeme mit Hilfe der Konzepte praktisch durchführen, und deren Vor- und Nachteile experimentell prüfen zu können. Seit einigen Jahren existiert eine weitgehend vollständige Implementierung der MoDiS-Konzepte auf Basis des GNU C Compilers und des Linux-Kerns. Diese Implementierung wird folgend als MoDiS-Experimentalsystem, oder kurz Experi-

[9] Eine Halbordnung, in der für jede zwei-elementige Teilmenge das Infimum und das Supremum existieren, heißt *Verband*.

mentalsystem bezeichnet. In den folgenden Abschnitten werden die wesentlichen Bestandteile sowie wichtige Funktionen bzw. Eigenschaften des Experimentalsystems kurz erklärt. Dabei wird zunächst auf die beiden generativen Managementanteile Übersetzer und Binder eingegangen, und anschließend einige interessante Gesichtspunkte des Laufzeit-Managements beschrieben.

2.3.1 Übersetzer

Der Übersetzeranteil des Managements hat im wesentlichen zwei grundsätzliche Aufgaben. Einerseits müssen gemäß der Top-Down-Konkretisierung die abstrakten Spezifikationen schrittweise verfeinert werden, um ausgeführt werden zu können. Der Übersetzer muss also den INSEL-Code einlesen, analysieren, und unter Wahrung der beschriebenen Komponenten-Eigenschaften ausführbaren Code für die zur Verfügung stehende Hardwarekonfiguration generieren. Die zweite Aufgabe des Übersetzers besteht darin, bei der Analyse der Spezifikation Informationen über die Anwendung zu sammeln und diese direkt oder indirekt dem Laufzeit-Management zur Verfügung zu stellen. Direkt bedeutet, dass die Anwendungsinformationen bereits in die Codegenerierung des Laufzeit-Managements einfließen. Eine indirekte Bereitstellung der Informationen über die Anwendung geschieht, indem die Informationen gespeichert werden, die im Laufzeit-Management zur Ausführungszeit genutzt werden können.

Der Übersetzer wird als *gic* (GNU INSEL Compiler, vgl. [Piz97]) bezeichnet und basiert auf dem GNU C Compiler (gcc, vgl. [gcc]). Zur statischen Analyse wird der abstrakte Syntax-Baum[10] mit Hilfe des Attributauswerters MAX (vgl. [APH93] und [PH93]) analysiert und mit Attributen versehen. Dabei werden zusätzlich zu den statisch analysierbaren Eigenschaften auch Attribute für dynamische Eigenschaften vorbereitet, deren Werte abgeschätzt und zur Laufzeit erst mit endgültigen Werten belegt werden. Auf diese Art werden beispielsweise kommunikationsbedingte Abhängigkeiten und approximierte Ausführungszeiten für Akteure erfasst. In [Reh04] ist beschrieben, wie auf diese Weise im Übersetzer potentielle Ereignisspuren erarbeitet werden, aus denen zur Laufzeit *Attributierte Nebenläufigkeitsverbände* (ANV) bzw. *Attributierte Potentielle Nebenläufigkeitsverbände* (APNV) erzeugt werden.

Ein ANV ist ein Ereignisverband, in dem alle nebenläufigen Ereignisse, die also keine Abhängigkeiten zu anderen Akteuren aufweisen, zu jeweils einem Ereignis zusammengefasst werden. Zudem werden die Kanten des Ereignisverbands mit den approximierten Ausführungszeiten gewichtet. In einem APNV sind nicht-

[10] engl.: abstract syntax tree (AST)

deterministische Entwicklungen in Form von Schleifen und bedingten Anweisungen noch nicht mit konkreten Werten belegt und die dadurch entstehenden verschiedenen Ausführungsmöglichkeiten vereinfacht dargestellt. Die notwendigen Abhängigkeiten zur Konstruktion dieser Verbände werden erst in Kapitel 4.1 formal eingeführt und erklärt, die Verbände selbst für Managemententscheidungen in den Kapiteln 4, 5 und 6 genutzt. Dennoch wird bereits hier zum besseren Verständnis ein ANV in Abbildung 2.4 dargestellt.

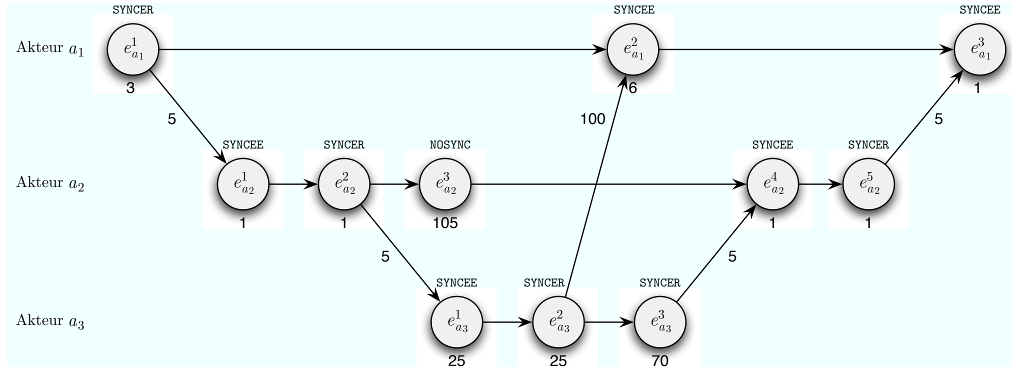


Abbildung 2.4.: Beispiel für einen Attributierten Nebenläufigkeitsverband

Dieser Verband wird zur Laufzeit aus den zur Übersetzungszeit generierten Ereignisspuren der Akteure erzeugt. Der ANV ist in der Abbildung als gerichteter, azyklischer Graph dargestellt. Alle Ereignisse der MoDiS-Konzepte werden den Klassen `SYNCER`, `SYNCEE` und `NOSYNC` zugeordnet. Nebenläufige Ereignisse ohne Synchronisationsabhängigkeiten werden dabei zusammengefasst. Den Knoten zugeordnete Zahlen sind approximierte, relative Berechnungszeiten der Akteure für diese Ereignisse, wie beispielsweise 3 für das Ereignis $e^1_{a_1}$. Analog können Synchronisationskanten mit approximierten Kommunikationszeiten gewichtet werden, wie beispielsweise 5 für die Kante von $e^1_{a_1}$ zu $e^1_{a_2}$.

2.3.2 Binder

Der generative Managementanteil ist in MoDiS mit dem Laufzeit-Management eng verbunden. Dies zeigt sich einerseits an der Bereitstellung von Informationen Seitens des Übersetzers, die im Laufzeit-Management genutzt werden. Der zweite Anteil des generativen Managements, in Form des Binders *FLink* (Flexible incremental and dynamic Link loader), ermöglicht eine Informationsweitergabe in entgegengesetzter Richtung, also vom Laufzeit-Management zum Übersetzer.

Der Binder FLink kann die Programmmodule inkrementell in den Speicher laden und offene Bezüge zur Laufzeit auflösen.

Auf Grund der bekannten Eigenschaften der Komponenten können Erzeugungsaufrufe auch für unvollständige Komponentenklassen generiert werden. Dies wird mittels dreier Bindungsarten realisiert, die sich im Zeitpunkt und der Technik der Adressauflösung unterscheiden. Das inkrementelle Binden erlaubt, Managemententscheidungen, die der Übersetzer bei der Konkretisierung zu treffen hat, bis zur Laufzeit hinauszuzögern und erst mit zusätzlicher Laufzeitinformation Teile des Systems (neu) zu übersetzen und den erzeugten Code zum laufenden System zu binden. Weitere Beschreibungen des Binders FLink sind in [Reh98] und [Jek99] zu finden.

2.3.3 Laufzeit-Management

Das Laufzeit-Management *ISE* des MoDiS-Experimentalsystems übernimmt sowohl die stellenlokalen Aufgaben zur Ausführung der lokalen Komponenten, welche mit den Aufgaben eines herkömmlichen Betriebssystemkerns zusammen mit entsprechenden Laufzeitbibliotheken vergleichbar sind. Daneben zählt die Verwaltung der verteilten Hardwarekonfiguration und die Bereitstellung der Mechanismen zur effizienten Nutzung und Verwaltung dieser Hardwarekonfiguration zu den Aufgaben des Laufzeit-Managements. Der dritte und entscheidende Punkt ist jedoch die Entscheidungsfindung bei der Konkretisierung, dies sind die die anwendungsorientierten Strategien zur Nutzung und Verwaltung der (lokalen und verteilten) Betriebsmittel. Die beiden letzten Punkte werden im wesentlichen durch Manager ausgeführt, die den Akteuren zugeordnet sind und deren Anforderungen erfüllen müssen.

Das Laufzeit-Management basiert auf einem Linux-Kern der Version 2.4.24, der die Basisfunktionalität zur stellenlokalen Ausführung zur Verfügung stellt. Beispielsweise werden Akteure zu leichtgewichtigen Linux-Tasks mit geringfügig geändertem Speicherlayout konkretisiert.

Das Laufzeit-Management stellt für die stellenübergreifenden Funktionen eine Reihe von Basismechanismen zur Verfügung, die zur Konkretisierung genutzt werden können. Dazu zählt ein Kommunikator-Modul, um das der Linux-Kern erweitert wurde, mit Hilfe dessen einerseits die Kommunikation der Manager, also der Laufzeit-Management-Anteile jeder Komponente, realisiert wird. Ferner werden auch die Nachrichten des Operationen-orientierten Rendezvous durch den Kommunikator verschickt.

Ein gemeinsamer, verteilter Speicher wird bereitgestellt, um den transparenten Zugriff auf Datenobjekte im Sichtbarkeitsbereich zu realisieren. Der gemeinsame Speicher erfüllt sequentielle Konsistenz und basiert selbst wiederum auf dem Kommunikator und lokalen Speicher-Managern. Der gemeinsame, verteilte Speicher arbeitet seitenbasiert, d.h. falls auf eine Speicherseite des gemeinsamen Adressraums zugegriffen wird, wird eine Kopie dieser Seite auf die entsprechende Stelle übertragen und dort in den Adressraum des Akteurs eingeblendet. Die Speicher-Manager regeln die Lese- und Schreibzugriffe, sodass sequentielle Konsistenz gewahrt wird.

Neben der Bereitstellung flexibler Kommunikationsmechanismen können die Komponenten, insbesondere die Akteure, vom Laufzeit-Management explizit bestimmten Stellen zugewiesen werden. Über dies hinaus können sie von einer Stelle zu einer anderen migriert werden. Die Platzierung und Migration von Akteuren sind Beispiele für die anwendungsorientierten Konkretisierungs-Entscheidungen, die vom Laufzeit-Management getroffen werden. Dabei werden Informationen über das Kooperationsverhalten der Akteure aus der Vergangenheit sowie im Übersetzer generierte Vorhersagen über deren Zukunft genutzt. Initial werden die Komponenten dann mit dem Ziel maximaler Performanz auf Stellen zugewiesen. In Kapitel 6 wird dieses Verfahren noch näher beschrieben, ausführliche Abhandlungen sind in [Reh04] und [Reh06] zu finden.

2.4 Zusammenfassung

Im Projekt MoDiS wurden Konzepte zur Konstruktion verteilter, nebenläufiger Systeme erarbeitet. Dabei wurde ein sprachbasierter Gesamtsystemansatz zur Top-Down Konkretisierung verfolgt. Die Sprache INSEL stellt die sprachlichen Mittel zur Verfügung, verteilte, kooperative Problemlösungen auf hohem Abstraktionsniveau zu spezifizieren. Dazu werden geeignete Komponentenarten bereitgestellt, die mittelbare Kooperation über gemeinsamen Zugriff auf passive Datenobjekte und unmittelbare Kooperation über Operationen-orientiertes Rendezvous ermöglichen. Die Abhängigkeiten dieser Komponenten sind klar festgelegt und somit das Verhalten des System spezifiziert.

Die eigentliche Problemlösung wird gemäß dem Gesamtsystemansatz zusammen mit dem Management generiert, somit stehen sowohl statische als auch dynamische Informationen über die Anwendung für die Konkretisierungs-Entscheidungen zur Verfügung.

Die MoDiS-Konzepte sind in Form des MoDiS-Experimentalsystems implementiert, um die Machbarkeit zu beweisen und praktische Erfahrungen zu sammeln. Entscheidende Bausteine des Experimentalsystems sind der Übersetzer *gic*, der Binder *FLink* sowie das Laufzeit-Management *ISE*. Der Übersetzer stellt neben der Transformation der Spezifikation in ausführbaren Code auch Anwendungsinformationen für das Laufzeit-Management bereit. Der Binder ermöglicht, Informationen aus der Ausführung in die Übersetzung einfließen zu lassen, da Code dynamisch zur Laufzeit zum bestehenden System gebunden werden kann. Das Laufzeit-Management selbst stellt alle Mechanismen zur stellenlokalen und stellenübergreifenden Ausführung bereit und ermöglicht das Treffen anwendungsorientierter Entscheidungen, wie beispielsweise die Entscheidung der Stellenzuweisung.

Kapitel 3

Fehlertoleranz in verteilten Systemen

Inhalt

3.1. Fehler 36

3.2. Fehlererkennung 40

3.3. Fehlertoleranz 44

3.4. Zusammenfassung 51

In dieser Arbeit sind Konzepte und Verfahren zur Toleranz von Fehlern in verteilten, nebenläufigen Systemen am Beispiel des Projekts MoDiS entstanden. Um die erarbeiteten Verfahren zur Erkennung und Toleranz von Fehlern beschreiben zu können, werden in diesem Kapitel allgemeine Begriffe und Mechanismen der Fehlertoleranz eingeführt. Die Verwendung der verschiedenen Fehlerbegriffe und deren Klassifizierung unterscheidet sich in der Literatur teilweise, deshalb werden diese Begriffe festgelegt. Grundlegende Techniken zur Fehlererkennung und Fehlertoleranz werden in der für die vorliegende Arbeit notwendigen Tiefe beschrieben.

Die Begriffsdefinitionen und Beschreibungen gängiger Verfahren dieses Kapitels orientieren sich an den Arbeiten von Max Breitling (vgl. [Bre01]), sowie an den Büchern von Pankaj Jalote (vgl. [Jal98]) und Israel Koren, C. Mani Krishna (vgl. [KK07]).

3.1 Fehler

Als Fehler werden allgemein falsche Ergebnisse bzw. unerwünschte Vorkommnisse bezeichnet. Tatsächlich muss dabei zwischen verschiedenen Facetten dieses Begriffs unterschieden werden. Angenommen ein Programmierer macht einen Rundungs-Fehler bei der Implementierung einer Steuerungsroutine eines Kraftwerks, sodass fälschlicherweise der Wert einer sicherheitskritischen Variablen für manche Sensoren den Wert 0 anstatt 1 annimmt. Bei Wert 0 schalte sich das Kraftwerk aus Sicherheitsgründen ab. Bei diesem Beispiel sind bereits drei Facetten des Fehlerbegriffs zu Tage getreten, die weitläufig als *Fehler* bezeichnet werden:

- Der Fehler des Programmierers bei der Implementierung der Routine.
- Der falsche Wert in der sicherheitskritischen Variablen.
- Das Abschalten des Kraftwerks, obwohl eigentlich alles in Ordnung war.

Die Beschreibung von Verfahren zur Fehlererkennung, Fehlervermeidung und Fehlertoleranz erfordert offensichtlich eine klare Definition bzw. genauere Unterscheidung des Begriffs *Fehler*. In der englischsprachigen Literatur werden meist die Begriffe *fault*, *error*, und *failure* benutzt, sowie zum Teil zusätzlich *malfunction* und *mistake*. Im deutschen sind die Begriffe *Fehler*, *Fehlerursache*, *Fehlerzustand*[1], *Ausfall* und *Versagen* gebräuchlich, wobei diese nicht eins zu eins den englischen Begriffen entsprechen. In der vorliegenden Arbeit werden die Begriffe analog zu [Bre01] und [LAK92] wie folgt verwendet:

Es wird nur zwischen der *Fehlerursache* (engl. *fault*), dem *Fehlerzustand* (engl. *error*) und dem *Versagen* (engl. *failure*) unterschieden. Die Bedeutung der Begriffe wird analog zu den englischen Begriffen verwendet:

3.1.1 Versagen – failure

Ein Versagen des Systems liegt vor, wenn das System nach aussen, also in einer Black-Box-Sicht nicht das gewünschte Verhalten zeigt. Das gewünschte Verhalten ist zu interpretieren als das in einer abstrakten Spezifikation festgehaltene Soll-Verhalten eines Systems. Bereits hier zeigt sich, dass auch diese Definition problematisch sein kann, da beispielsweise eine Spezifikation durchaus von den

[1] Wird auch als *Fehlzustand* bezeichnet.

eigentlichen Vorstellungen eines Kunden oder Anwenders abweichen und damit fehlerhaft sein kann. Diese Unterscheidung ist allerdings nicht Thema der vorliegenden Arbeit, deshalb wird vom rational sinnvoll erscheinenden intendierten Verhalten, wie beispielsweise der endlichen Terminierung eines Systems ausgegangen.

Die Art des Versagens kann weiter unterschieden werden. Beispielsweise kann das Versagen *wert-* oder *zeitbezogen* sein. Soll ein System einen bestimmten Wert berechnen, so könnte einerseits das Ergebnis rechnerisch falsch sein, andererseits kann es auch rechnerisch richtig, aber nicht innerhalb der spezifizierten Zeit ausgegeben werden. Ferner kann ein System vollständig ausfallen und gar keine Ausgabe produzieren.

Die Art des Versagens wird in Klassen unterschieden, die aus den Schwierigkeiten der Handhabung dieser Fehlerarten entstanden sind. Zum Teil wird diese Einordnung auch für die Fehlerursachen verwendet, die zu den entsprechenden Arten des Versagens führen. Für diese Arbeit werden die Arten des Versagens eingeführt, die Begriffe aber sowohl für das Versagen als auch für die verantwortlichen Fehlerzustände und Fehlerursachen verwendet.

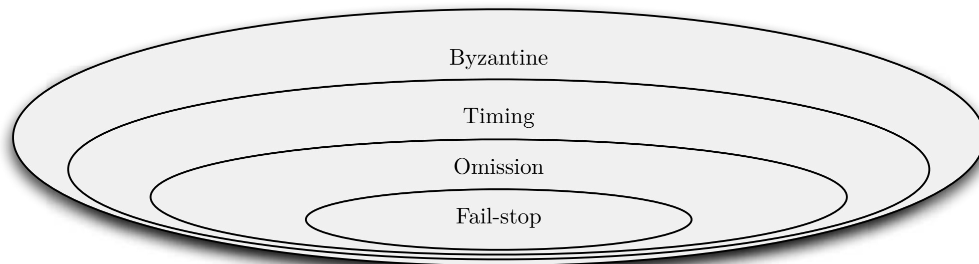


Abbildung 3.1.: Klassifizierung des Versagens

Die Klassen bilden eine Hierarchie, wobei die allgemeinste Form des Versagens als *Byzantine failures* bezeichnet wird. Jede Klasse schließt die jeweils restriktivere bzw. genauer festgelegte mit ein, wie in Abbildung 3.1 dargestellt.

- *Fail-stop failures:* Eine Komponente hält im Fehlerfall an, ohne falsche Ausgaben zu produzieren.

- *Omission failures:* Eine Komponente reagiert nicht mehr auf bestimmte Eingaben, beispielsweise auf ankommende Nachrichten.

- *Timing failures:* Eine Komponente produziert ihr Ergebnis zu früh, zu spät (oder gar nicht). Falls das Ergebnis noch produziert wird, handelt es sich allerdings um das korrekte Ergebnis.
- *Byzantine failures:* Eine Komponente verhält sich beliebig falsch bzw. chaotisch. Diese Klasse schließt jegliches Fehlverhalten mit ein.

Von Fail-stop failures hin zu Bynatine failures nimmt die Schwierigkeit zu, diese Fehler zu erkennen bzw. geeignet darauf zu reagieren. Ein Versagen nach Fail-stop Semantik kann leicht bemerkt werden und ist in der Regel wenig kritisch, da das Versagen des Teilsystems andere Teilsysteme auf genau festgelegte Art beeinflusst. Im eingangs geschilderten Beispiel entspricht das Abschalten des Kraftwerks einer Fail-stop Semantik und ist sicherlich wünschenswerter als wenn das Kraftwerk beispielsweise explodieren könnte, was einem byzantinischen Versagen entspräche.

3.1.2 Fehlerzustand – error

Ein Fehlerzustand ist ein interner Zustand des Systems, der von den gewünschten Zuständen abweicht, aber (noch) nicht nach aussen sichtbar ist. Dies können sowohl falsche Werte eines Nutzdatums als auch falsche Werte in Kontrollstrukturen sein. Der falsche Zustand ist nicht unbedingt an ein Datenobjekt gebunden, so könnte beispielsweise ein Zustand, in dem zwei bestimmte Prozesse gleichzeitig aktiv sind, als falsch betrachtet werden, meist sind jedoch relevante Zustände durch Werte in Nutzdaten oder Kontrollstrukturen repräsentiert.

Ein Fehlerzustand ist die kausale Ursache für ein Versagen, d.h. ein Fehlerzustand kann – muss aber nicht zwingend – zum Systemversagen führen. Ein Versagen kann aber nur auftreten, wenn ein Fehlerzustand vorlag. Eine Variable mit falschem Wert (Fehlerzustand) könnte beispielsweise zu einer falschen Ausgabe (Versagen) führen oder aber einfach mit einem korrekten Wert überschrieben werden, sodass der Fehlerzustand keine Auswirkungen nach aussen zeigt.

3.1.3 Fehlerursache – fault

Die Fehlerursache ist der eigentliche Grund für das Auftreten von Fehlerzuständen und Versagen. Die Fehlerursache kann in verschiedensten Bereichen liegen und ist manchmal nicht eindeutig zuzuordnen. Fehlerursachen können in Annahmen der Systemumgebung, in der Spezifikation, in der Implementierung und in

der Hardware zu finden sein. Wird die Fehlerursache als der Unterschied zwischen einem korrekt arbeitenden System und einem System mit Fehlerzuständen definiert, so ist dies oft nicht eindeutig. Im Eingangs erwähnten Beispiel war ein Rundungsfehler der Fehlzustand, allerdings ist dadurch nicht eindeutig, ob die Programmierung fehlerhaft ist, oder beispielsweise die Informationen über den Wertebereich des ausgelesenen Sensors falsch waren.

Eine Fehlerursache ist die kausale Ursache für einen Fehlerzustand, d.h. wiederum, dass eine Fehlerursache Fehlerzustände hervorrufen kann, aber nicht muss. Jeder Fehlerzustand hat allerdings eine Fehlerursache. Die Fehlerursache ist häufig statisch im System vorhanden, wie beispielsweise ein Programmierfehler. Fehlerzustände treten nur zur Ausführungszeit auf und sind demnach auch nur zur Ausführungszeit erkennbar. Ein Programmierfehler kann beispielsweise in manchen Ausführungen zum Fehlerzustand führen, in manchen auch nicht.

In der Literatur werden Fehlerursachen nach verschiedenen Kriterien unterschieden, wie beispielsweise nach der Lokalisierung, dem Verursacher, dem Zeitpunkt des Entstehens, der Dauer, dem Grund, sowie der Schwere. In der vorliegenden Arbeit sind diese Klassifizierungen nicht notwendig, lediglich die Dauer einer Fehlerursache ist von Interesse: Es werden *permanente* Fehlerursachen von *temporären* unterschieden. Permanente Fehlerursachen verbleiben im System, dazu zählen beispielsweise Softwarefehler, also Fehler in der Programmierung bzw. im System-Design. Temporäre Fehler werden weiter in *transiente* und *intermittierende* Fehlerursachen untergliedert. Eine transiente Fehlerursache besteht für einen begrenzten Zeitraum und verschwindet dann von selbst. Eine kurze elektromagnetische Störung, die den Inhalt einer Speicherzelle verändert, wäre ein Beispiel für eine transiente Fehlerursache. Tritt die gleiche transiente Fehlerursache immer wieder auf, aber nicht durchgehend, so wird sie als intermittierend bezeichnet.

Auch für Fehlerursachen werden zum Teil die Klassifizierungen gemäß Abbildung 3.1 gebraucht, falls diese zu Fehlerzuständen und entsprechend zum Versagen gemäß dieser Klassen führen.

3.1.4 Fehler – allgemein

In dieser Arbeit sollen auftretende Fehler zur Laufzeit erkannt und toleriert werden. Genau sind es also Fehlerzustände, die auf Grund vorhandener Fehlerursachen auftreten können, die aber nicht zum Auftreten eines Versagens führen sollen. Tritt ein Fehlerzustand auf und es wird verhindert, dass dieser zum Versagen führt, so wird dies als *Fehlertoleranz* bezeichnet. Allgemeiner formuliert

ist das Ziel der Fehlertoleranz, dass trotz vorhandener Fehlerursachen im System kein Versagen auftritt.

Damit das Systemmanagement dies aktiv durchführen kann, müssen die Fehlerzustände zunächst erkannt werden; dies erfordert Mechanismen zur *Fehlererkennung*. Offensichtlich sind es Fehlerzustände die *erkannt* werden müssen, um den kausalen Zusammenhang zwischen diesen und dem Versagen zu durchbrechen, was dann der eigentlichen Toleranz des Fehlers entspricht. Die Fehlererkennung ist also eine Voraussetzung für Fehlertoleranz und kann als Teil der Fehlertoleranz bezeichnet werden. In der Literatur wird zwischen Fehlertoleranz und Fehlerbehebung unterschieden, wobei sich die Fehlertoleranz dann aus der Fehlererkennung, der Fehlerlokalisierung und der Fehlerbehebung zusammensetzt. Diese feingranulare Unterscheidung ist in der vorliegenden Arbeit nicht immer notwendig, sodass häufig nur von Fehlererkennung und darauf aufbauender Fehlertoleranz gesprochen wird.

Fehlervermeidende Verfahren, wie beispielsweise Softwaretests, versuchen dagegen *Fehlerursachen* zu entdecken und zu beheben, und somit das Auftreten von Fehlerzuständen auszuschließen. An dieser Stelle zeigt sich bereits, dass sich fehlervermeidende und fehlertolerierende Ansätze ergänzen, da sie an verschiedenen Stellen des Problems ansetzen. Je nach Fehler kann es einfacher sein, die Fehlerursache zu finden und zu beheben, oder aber die hervorgerufenen Fehlerzustände zu erkennen und zu verhindern, dass dadurch ein Versagen hervorgerufen wird. Auf diese Diskussion wird allerdings in Kapitel 6 noch detaillierter eingegangen.

Im Rahmen dieser Arbeit wird der Begriff *Fehler* allgemein verwendet, wenn die genauere Unterscheidung zwischen Fehlerursache, Fehlerzustand und Versagen irrelevant ist. Die Begriffe Fehlertoleranz, Fehlererkennung und Fehlervermeidung werden wie eben erklärt gebraucht.

3.2 Fehlererkennung

Wie bereits erklärt wird als *Fehlererkennung* das Erkennen von Fehlerzuständen zur Laufzeit bezeichnet. Dies setzt voraus, dass eine (mehr oder weniger genaue) Kenntnis über den korrekten Zustand vorliegt, sodass ein Fehlerzustand als Abweichung von diesem korrekten Zustand festgestellt werden kann. Fehlererkennung entspricht demzufolge stets einem Vergleich von *Ist*-Zustand und *Soll*-Zustand. Dies impliziert, dass die Möglichkeiten eines Systems, Fehler zu erkennen, direkt von den vorhandenen Informationen über den Soll-Zustand abhängen.

Fehlerursachen können nicht vom System erkannt werden. Allerdings basieren fehlervermeidende Verfahren, wie beispielsweise Testverfahren teilweise darauf, von erkannten Fehlerzuständen auf die Fehlerursachen zu schließen. Dies wird üblicherweise als *debugging* bezeichnet.

Das Erkennen eines Unterschieds zwischen Ist- und Soll-Zustand wird als Test bezeichnet. Ein *idealer* Test hat folgende Eigenschaften (nach [Jal98], S. 10):

- Er wird nur von der Spezifikation, nicht vom internen Systemdesign abgeleitet, da sonst ein Designfehler auch ein Fehlverhalten des Tests verursachen könnte.
- Ein idealer Test ist *vollständig* und *korrekt*, d.h. der Test erkennt *alle* auftretenden Fehlerzustände, die toleriert werden sollen, und meldet keine *false-positives*, also Fehlerzustände die eigentlich korrekte Zustände sind.
- Ein idealer Test ist zudem *unabhängig* vom System in Bezug auf die Fehleranfälligkeit, d.h. falls ein Fehler im System vorliegt, wie beispielsweise eine ausgefallene Hardwarekomponente, so hat dies keine Implikation auf die Fehlerwahrscheinlichkeit des Tests.

Offensichtlich sind ideale Tests mit diesen Eigenschaften in der Praxis nur selten oder gar nicht realisierbar. Der Aufwand an Kosten und Rechenleistung für Tests steigt unverhältnismäßig an, je mehr man sich an die Forderung nach Vollständigkeit und Korrektheit annähert. Ferner ist die Forderung nach Unabhängigkeit bereits auf Grund möglicher physikalischer Einflüsse nicht vollständig erfüllbar, wenn System und Tests räumlich zusammen liegen.

In der Praxis sind deshalb so genannte *Akzeptanztests* von Bedeutung. Akzeptanztests sind ein Kompromiss aus geringen Kosten, geringen Performanzeinbußen und hoher Wahrscheinlichkeit, Fehler zu erkennen. Je nach System sind unterschiedliche Akzeptanztests sinnvoll, wobei einige typische Arten vorgestellt werden:

3.2.1 Timing-Tests

Sind Ausführungszeiten von Systemteilen oder Abläufen bekannt, so kann die Einhaltung dieser Zeiten überprüft werden. Je großzügiger dabei die Schranken gewählt werden, umso unwahrscheinlicher sind *false-positives*. Bei vernetzten Systemen sind Timing-Tests zum Erkennen von Fehlern der Kommunikation üblich.

In verteilten Systemen werden Timing-Tests auch häufig eingesetzt um den Ausfall einer Stelle zu erkennen. Timing-Tests sind in der Lage, die Klasse der Timing failures zu erkennen, die bereits die Klasse der Omission-Fehler einschließt.

3.2.2 Replikations-Tests

Die Replikation von Daten oder Komponenten ist ein wichtiger grundsätzlicher Ansatz zur Fehlertoleranz. Replikations-Tests vergleichen die Ausgabe von beispielsweise parallel ausgeführten identischen Komponenten, um Fehler in Ausführungen zu erkennen. Es ist ein Vorteil von Replikations-Tests, dass für den Test keine Information über das berechnete Ergebnis notwendig ist, sondern lediglich mehrere Ergebnisse verglichen werden müssen. Andererseits ist die parallele Ausführung identischer Funktionalität in Bezug auf die Ressourcen ein sehr aufwändiger Ansatz zur Fehlertoleranz.

3.2.3 Struktur-Tests

Sind Daten auf eine bestimmte Weise strukturiert, so kann die Einhaltung dieser Struktur überprüft werden. Dies ist möglich, falls die Daten in sich Redundanz enthalten, oder diese Redundanz den Daten hinzugefügt wird. Bei der Kodierung von Daten werden solche Verfahren angewandt, wie beispielsweise Paritäts-Bits oder crc-Codes (cyclic redundancy check). Struktur-Tests sind in erster Linie bei der Datenkodierung üblich, können aber auch eingeschränkt auf höherer Ebene eingesetzt werden. Der Vorteil auf Bit-Ebene ist die eingeschränkte Möglichkeit der Veränderung der Daten durch nur zwei Zustände.

3.2.4 Semantik-Tests

Eine Reihe von Tests der Semantik eines berechneten Ergebnisses, die in der Literatur unterschiedlich bezeichnet werden, werden hier als Semantik-Tests zusammengefasst. Dazu zählen unter anderem die Begriffe *verification of output*, *probabilistic checks* und *range checks*. Die Tests haben gemeinsam, dass Informationen über ein berechnetes Ergebnis (bekannter Eingabe) verwendet werden, um dessen Korrektheit nachzuprüfen. Der ideale Fall liegt vor, wenn ein aufwändig zu berechnendes Ergebnis eine verhältnismäßig einfache Proberechnung hat; in diesen Fällen kann die Korrektheit des Ergebnisses verifiziert werden. Wird beispielsweise die Quadratwurzel von einem Wert berechnet, so kann als Test ohne zu großen Einfluss auf die Performanz des Systems das Ergebnis zur Kontrolle

quadriert werden. Ist die vollständige Berechnung der Probe zu aufwändig, so kann zufallsbasiert ein Teil der Rechnung überprüft werden. Beispielsweise könnte bei der Berechnung einer Matrix-Multiplikation die Korrektheit einer zufällig ausgewählten Zeile mit alternativem Algorithmus geprüft werden.

Eine weitere Möglichkeit von Semantik-Tests ist die Überprüfung, ob das Ergebnis innerhalb eines eingeschränkten Wertebereichs liegt. So sollte beispielsweise das Ergebnis einer Sinus- oder Cosinusfunktion stets im Intervall $[-1; 1]$ liegen. Unter Umständen kann auch die relative Veränderung eines Wertes überprüft werden. Beispielsweise kann eine zu starke Veränderung eines Sensorwertes in kurzem Zeitintervall auf einen Fehler hinweisen. Je enger der Wertebereich festgelegt ist, den eine Funktion als korrektes Ergebnis liefern kann, desto höher ist die Wahrscheinlichkeit, Fehler durch Semantik-Tests zu entdecken.

3.2.5 Diagnose-Tests

Diagnose-Tests werden häufig auch als Selbst-Tests bezeichnet und werden im Gegensatz zu den bisher genannten explizit aufgerufen, um die Funktion einzelner Komponenten eines Systems zu überprüfen. Üblicherweise werden Diagnose-Tests nicht im laufenden Betrieb, sondern beispielsweise beim Neustart von Systemen oder nach Rekonfigurationen durchgeführt.

3.2.6 Sonstige Mechanismen

Neben den beschriebenen generellen Test-Arten existieren weitere Möglichkeiten zur Fehlererkennung, die für spezielle, klar festgelegte Fehler konzipiert werden. Ein Beispiel für diese Klasse wären Verfahren zur Deadlock-Erkennung, die in der Regel auf Zyklenerkennung in Graphen basieren (vgl. [Sta02], Kap. 14.4).

Alle Fehlererkennungsverfahren basieren jedoch auf dem Prinzip, den Unterschied zwischen einem Soll- und einem Ist-Zustand zu erkennen. Wird dieser Unterschied erkannt, so muss geeignet reagiert werden. Der Fehlerzustand kann gemeldet werden, wie beispielsweise durch eine Ausgabe, einen Eintrag in eine Log-Datei oder auch eine e-mail an den Administrator. Ferner kann das Management eines Systems auf den Fehlerzustand reagieren, im einfachsten Fall durch Beendigung des Programms. Soll die Berechnung trotz des Fehlerzustands korrekt fortgesetzt werden, ohne zum Versagen des Systems zu führen, so müssen Verfahren zur Toleranz des Fehlers eingesetzt werden.

3.3 Fehlertoleranz

Ist ein Fehlerzustand erkannt, so ist das Ziel der Fehlertoleranz, die Ausführung des Systems korrekt fortzuführen, also ein Versagen zu vermeiden. Da das Versagen eines Systems abhängig von den gestellten Anforderungen ist, sind auch die Anforderungen an das Fehlertoleranz-Verfahren systemabhängig. Beispielsweise wird in vielen Systemen ein Fehlerzustand auch als toleriert angesehen, wenn das Ergebnis zwar etwas später vorliegt, aber korrekt ist. In Echtzeitsystemen kann dies bereits ein Versagen darstellen. Auch die erwartete Genauigkeit einer Berechnung kann variieren. Wird in einer Wettersimulation durch einen Rundungsfehler ein Temperaturwert um 0,1 Grad Celsius falsch berechnet, so gilt dies sicherlich nicht als Versagen, wohingegen ein um 0,1 Euro falsch berechneter Aktienkurs großen finanziellen Schaden anrichten kann und sicherlich als fehlerhaft gilt. Das Versagen eines Systems hängt vom erwarteten Verhalten ab. Die Anforderungen an ein System schränken demnach bereits die möglichen Verfahren zur Fehlertoleranz ein.

Generell basieren Fehlertoleranz-Verfahren auf *Redundanz*. Redundanz bezeichnet die allgemeine Eigenschaft, mehr von einer Ressource zur Verfügung zu haben als eigentlich notwendig wäre. Diese Beschreibung ist sehr allgemein gehalten, da Redundanz sich in verschiedene Arten untergliedern lässt. In der Literatur sind unterschiedliche Untergliederungen zu finden, wobei folgende allgemein gehaltene Einordnung mit gängigen Werken übereinstimmt:

- *Strukturelle Redundanz* bezeichnet die Vervielfältigung von Komponenten im System.

- *Funktionale Redundanz* bezeichnet das Vorhandensein zusätzlicher Funktionalität (in Soft- oder Hardware), die nicht zum Zweck der eigentlichen Problemlösung dient. Diese Funktionalität ist also nur zum Zweck der Fehlertoleranz, also beispielsweise zur Erkennung, Lokalisierung oder Behebung von Fehlern integriert.

- *Informationsredundanz* bezeichnet das mehrfache Speichern von Informationen im System. Dies beinhaltet sowohl vollständige Kopien von Daten als auch Prüfsummen und redundante Kodierungsverfahren.

- *Zeitliche Redundanz* liegt vor, wenn mehr Zeit zur Ausführung aufgewendet wird, als eigentlich zur Problemlösung notwendig wäre. Funktionale Redundanz impliziert bereits zeitliche Redundanz, da zusätzliche Funktionalität

nur für die Fehlertoleranz ausgeführt wird[2]. Wird nach einem Fehler eine Berechnung erneut ausgeführt, so entspricht dies ebenfalls zeitlicher Redundanz.

- *Statische Redundanz* bezeichnet redundante Einheiten, die auch in Abwesenheit von Fehlern einen Beitrag zur Ausführung des Systems leisten.
- *Dynamische Redundanz* bezeichnet redundante Einheiten, die erst im Fall eines aufgetretenen Fehlers einen Beitrag zur Ausführung des Systems leisten.

Folgend werden generelle Vorgehensweisen erklärt, wie auf Basis von Redundanz Fehler behoben werden können. Anschliessend werden Verfahren zur Toleranz von Hard- und Softwarefehlern erläutert.

3.3.1 Fehlerbehebung

Allgemein ist das Ziel des Fehlertoleranzverfahrens, den kausalen Zusammenhang von Fehlerzustand zu Versagen zu durchbrechen, was auch als Fehlerbehebung bezeichnet wird. Dies wird erreicht, indem ein erkannter Fehlerzustand in einen korrekten Zustand transformiert wird. Drei Ansätze dieser Transformation werden unterschieden: *Rückwärtsbehebung*, *Vorwärtsbehebung* und *Seitwärtsbehebung* (bzw. *Fehlerkompensation*).

3.3.1.1 Rückwärtsbehebung

Rückwärtsbehebung basiert auf zeitlicher Redundanz. Im Fall eines Fehlers wird ein zuvor gespeicherter Zustand wieder hergestellt und die Berechnung ab diesem Zustand fortgesetzt. Dies erfordert das regelmäßige Speichern des Zustands eines Systems. In verteilten, nebenläufigen Systemen ist das Speichern eines Systemzustands eine nichttriviale, aufwändige Aufgabe, da der Zustand *konsistent* sein muss; auf diese Problematik wird in den Kapiteln 4.1 und 5 ausführlich eingegangen.

Wird nach einem erkannten Fehler ein vormals gespeicherter Zustand wiederhergestellt, so wird der gleiche Fehlerzustand erneut auftreten, falls er die Folge einer *permanenten* Fehlerursache ist. Generell eignet sich daher die Rückwärtsbehebung nur zur Toleranz transienter oder intermittierender Fehler. Im Rahmen

[2]Diese Implikation gilt nicht, falls die Ausführungszeit der Anwendung durch die funktionale Redundanz auf Grund zusätzlicher Hardware nicht beeinträchtigt wird.

dieser Arbeit wird ein Konzept vorgestellt, mit dem auch permanente Fehler auf der Basis von Rückwärtsbehebung toleriert werden können (vgl. Kapitel 6).

Es ist ein Vorteil der Rückwärtsbehebung, dass keine genaue Kenntnis des Fehlers notwendig ist. Der Fehler muss nicht lokalisiert werden; Fehlerlokalisierung kann eine schwierige Aufgabe im Rahmen der Fehlertoleranz darstellen. Ein Fehlerzustand kann sich im Zeitraum zwischen seinem Auftreten und seiner Erkennung bereits ausgebreitet haben und Fehler in anderen Systemteilen oder Komponenten verursacht haben. Beispielsweise kann ein falsch berechneter Wert als Zwischenergebnis für weitere Berechnungen verwendet, und zudem über die Vernetzung zu einem anderen Prozess versendet worden sein. Bei der Fehlerlokalisierung muss festgestellt werden, welche Systemteile von dem Fehler beeinflusst sind bzw. beeinflusst sein könnten.

Der größte Vorteil der Rückwärtsbehebung ist, dass keine Informationen über den eigentlich korrekten Zustand benötigt werden, um einen Fehlerzustand zu beheben. Es reicht aus zu erkennen, dass ein Fehlerzustand vorliegt; die Kenntnis des eigentlich erwünschten Zustands ist darüber hinaus nicht erforderlich, da vom letzten gespeicherten Zustand dieselbe Ausführung erneut versucht wird.

Ein grosser Nachteil der Rückwärtsbehebung ist die Beeinträchtigung der Systemperformanz durch das Speichern der Zustände im fehlerfreien Betrieb. *Checkpoint-Rollback-Verfahren* realisieren das Speichern von Systemzuständen in verteilten Systemen. Diese Verfahren werden in Abschnitt 5.1 näher erläutert und Unterschiede diskutiert.

3.3.1.2 Vorwärtsbehebung

Wird ein Fehlerzustand erkannt, so wird durch Vorwärtsbehebung versucht, den Fehlerzustand in einen korrekten Zustand zu ändern. Dazu muss einerseits der Fehler lokalisiert werden: der bereits entstandene Schaden muss vollständig erfasst werden, obwohl der Fehlerzustand sich bereits ausgebreitet haben kann. Alle betroffenen Systemteile müssen dann in einem zweiten Schritt in einen korrekten Zustand versetzt werden.

Um dies zu realisieren werden Informationen benötigt, welche Zustände korrekt sind. In manchen Systemen reicht auch ein Zustand aus, der *nahe* dem eigentlich berechneten liegt, und approximiert werden kann. Beispielsweise könnte bei einem Fehlerzustand eines Sensorwertes ein neuer Wert aus den vergangenen Werten approximiert werden. Dieser Wert entspricht zwar unter Umständen nicht dem tatsächlich vorliegenden, liegt aber wahrscheinlich nahe genug daran,

dass die Weiterverarbeitung (beispielsweise die Steuerung eines Prozesses) die gleiche Reaktion zeigt, wie bei dem eigentlich korrekten Sensorwert, sodass kein Versagen folgt.

Vorwärtsbehebung setzt viel Detail-Wissen über die Anwendung, auftretende Fehlerzustände und deren Fehlerursachen voraus; deshalb kann Vorwärtsbehebung nur in speziellen Systemen eingesetzt werden, und nur in eingeschränkter Form in verteilten, nebenläufigen Systemen. Einzelne, in sich abgeschlossene Systemkomponenten oder Module können durch Vorwärtsbehebung Fehler tolerieren, falls das entsprechende Wissen über die korrekten Zustände vorhanden ist.

3.3.1.3 Seitwärtsbehebung

Durch Informationsredundanz kann ein Fehlerzustand unter Umständen direkt zu dem eigentlich korrekten Wert geändert werden; dies wird als Seitwärtsbehebung bzw. Fehlerkompensation bezeichnet. Im Gegensatz zur Vorwärtsbehebung ist der korrigierte Wert nicht ein angenommener oder approximierter Wert, sondern er entspricht exakt dem berechneten und ist durch die Informationsredundanz ermittelt worden. Fehlerkorrigierende Codes oder Raid-Systeme basieren auf Seitwärtsbehebung. Bis zu einer gewissen Anzahl auftretender Fehlerzustände können die korrekten Daten durch die redundante Information rekonstruiert werden.

Seitwärtsbehebung und Vorwärtsbehebung werden in der Literatur zum Teil nicht unterschieden und allgemein als Vorwärtsbehebung bezeichnet. Beide Ansätze grenzen sich von der Rückwärtsbehebung dadurch ab, dass keine erneute Ausführung der Berechnung stattfindet. Seitwärtsbehebung kann demnach auch als Spezialfall der Vorwärtsbehebung betrachtet werden, bei dem ein fehlerhafter Wert durch das Vorhandensein von Informationsredundanz zum korrekten Wert geändert werden kann.

3.3.2 Toleranz von Hardwarefehlern

Im Gegensatz zu Software können technische Geräte auf Grund physikalischer Beeinflussung und Alterung nie dauerhaft fehlerfrei arbeiten. Diese Schwäche der Hardware in Rechnern und Systemen motivierte ursprünglich den Ansatz zur Fehlertoleranz, der für Hardwarefehler – im Gegensatz zu Softwarefehlern – in der Praxis heute gängig ist. Auch wenn in dieser Arbeit die Toleranz von

Softwarefehlern im Vordergrund steht, wird dennoch ein kurzer Überblick über die Toleranz von Hardwarefehlern gegeben, um die Unterschiede zur Toleranz von Softwarefehlern zu verdeutlichen.

Zunächst ist entscheidend, dass bei Hardware in der Regel davon ausgegangen wird, dass nach der Herstellung die Hardware im Design fehlerfrei ist, die einzelnen Einheiten aber eine begrenzte Lebensdauer haben, also nach einer gewissen Zeit permanente Fehler auftreten bzw. einzelne produzierte Einheiten unter Umständen bereits von Anfang an defekt sind. Zudem können durch physikalische Einflüsse transiente Fehlerursachen auftreten. Es müssen also transiente und permanente Fehler toleriert werden, wobei davon ausgegangen werden darf, dass ein Teil der Einheiten für eine beschränkte Lebensdauer korrekt funktioniert. Diese Annahme unterscheidet die Toleranz von Hardwarefehlern essentiell von der Toleranz von Softwarefehlern, bei denen davon ausgegangen werden muss, dass ein Softwarefehler in allen Instanzen der Software identisch vorhanden ist.

Die gängigste Technik zur Toleranz von transienten Hardwarefehlern ist *Triple Modular Redundancy (TMR)* und entspricht einer Seitwärtsbehebung durch strukturelle, statische Redundanz. Bei diesem Ansatz werden Geräte so konstruiert, dass drei identische Hardware-Einheiten (z.B. Schaltungen) verbaut werden und parallel die gleiche Eingabe verarbeiten. Ein so genannter *Voter* erhält das Ergebnis der drei Einheiten und gibt das Ergebnis, das von mindestens zwei der drei Einheiten identisch ist, weiter. Somit kann trotz einer fehlerhaften Einheit die gesamte Hardware-Einheit fehlerfrei arbeiten. Eine Weiterentwicklung von TMR-Schaltungen sind entsprechende NMR-Schaltungen, bei denen analog N identische Schaltungen parallel die gleiche Eingabe verarbeiten, und somit mehr gleichzeitig fehlerhafte Schaltungen tolerieren können. Ein Vorteil von TMR- bzw. NMR-Schaltungen ist, dass an funktionaler Redundanz lediglich der Voter notwendig ist, keine sonstige Funktionalität zum Erkennen, Lokalisieren oder Beheben von Fehlern.

Der permanente Ausfall von Geräten wird häufig durch strukturelle, dynamische Redundanz toleriert. So genannte *hot standby* oder *spare* Geräte bzw. Einheiten ersetzen eine ausgefallene Einheit. Im Gegensatz zu TMR benötigt dieser Ansatz explizite Fehlererkennung, -lokalisierung und das automatische, transparente Ersetzen ausgefallener Einheiten durch die standby-Einheiten. Ist beispielsweise in einem Raid-System eine zusätzliche Festplatte enthalten, so kann diese eine ausgefallene Platte ersetzen. Die Daten der ausgefallenen Platte können durch die redundanten Informationen der anderen Platten des Raid-Systems rekonstruiert werden.

Generell können Hardwarefehler toleriert werden, indem strukturelle Redundanz in Systeme integriert wird, da davon ausgegangen wird, dass keine Fehler im Design der Hardware enthalten sind, sondern lediglich einzelne produzierte Instanzen fehlerhaft sein können, die Lebenszeit beschränkt ist und transiente Fehler auftreten. Ist genügend strukturelle Redundanz in einem System vorhanden, so ist die Wahrscheinlichkeit, dass alle Instanzen einer enthaltenen Hardware-Einheit gleichzeitig ausfallen, stark verringert. Natürlich ist es keine triviale Aufgabe, mit Hilfe dieser strukturellen Redundanz entsprechend transparent Fehler zu tolerieren, aber die entwickelten Verfahren haben sich in der Praxis bewährt und werden eingesetzt.

3.3.3 Toleranz von Softwarefehlern

Der Ansatz struktureller Redundanz ist für Software nur sehr bedingt anwendbar, da die Fehlerursachen in Software in der Regel Design-Fehler bzw. Programmierfehler sind. Es sind also Fehlerursachen vorhanden, die von Anfang an in jeder Instanz der Software zu Fehlerzuständen führen können, sodass mehrere identische Kopien eines Softwaremoduls bei gleicher Eingabe den gleichen Fehler produzieren. P. Jalote fasst diesen Unterschied gut zusammen: „The root cause of failure of hardware components like nodes and links is frequently some physical failure. Software, in contrast, has no physical properties; it is a totally conceptual entity. Hence, physical faults cannot exist in software. Software faults are always design faults. That is, a fault in the software is due to incorrect design or the presence of bugs, which are usually caused by human errors (in contrast to physical failures, which are caused by natural laws).“ (aus [Jal98], S. 355).

Verfahren zur Toleranz von Softwarefehlern müssen wiederum unterschieden werden in solche, die zur Toleranz bestimmter, vorhersehbarer Fehler entworfen werden, und allgemeinen Verfahren, die unabhängig von der Art des aufgetretenen Fehlers arbeiten. Zur ersten Klasse gehören die Verfahren, die in der Regel mittels *Exceptions*, also Ausnahmebehandlungen angestoßen werden. Dabei bieten die Programmiersprachen entsprechende Konstrukte, um festgelegte Fehlerzustände abzufangen und anschließend eine definierte Behandlungsroutine aufzurufen. Die Fehlerbehebung geschieht also durch die von Hand programmierte Ausnahmebehandlungsroutine. Zum Teil können ganze Fehlerklassen mit identischen Ausnahmebehandlungsroutinen behandelt werden.

Diese Mechanismen erlauben allerdings nur die Toleranz von vorhersehbaren, definierten Fehlerzuständen der Ausführung und können somit schon fast als Teil der Problemlösung angesehen werden. Die Behandlungsroutinen selbst können

natürlich analog zur Problemlösung ebenfalls Softwarefehler enthalten. Die meisten Fehlerursachen der Software lösen Fehlerzustände aus, die nicht vorhersehbar sind, oder zumindest nicht mit Standardroutinen zu behandeln sind. Diese Softwarefehler können nicht durch Ausnahmebehandlungen toleriert werden.

Um *nicht vorhersehbare* Softwarefehler tolerieren zu können, wurde der Ansatz struktureller Redundanz weiterentwickelt zu *Diversität*: Mehrere *unterschiedliche* Implementierungen[3] eines Softwaremoduls, die funktional identisch sind, können zur Toleranz von Designfehlern eingesetzt werden. Diversitäre Programmierung, also die Erstellung von mehreren Versionen der gleichen Software, galt eine Zeit lang als die Lösung für das Problem, zuverlässige Software zu erstellen.

Zwei Ansätze zur Toleranz von Softwarefehlern auf Basis verschiedener Software-Versionen wurden entwickelt: *N-Version Programming* und *Recovery Blocks*. N-Version Programming arbeitet analog zu NMR-Systemen, die verschiedenen Versionen führen parallel die Berechnungen mit der gleichen Eingabe aus, es handelt sich also um Seitwärtsbehebung. Der Ansatz *Recovery Blocks* basiert auf Wiederholung der Ausführung, also Rückwärtsbehebung.

N-Version Programming. Das Konzept des N-Version Programming wurde im Jahr 1977 von A. Avižienis und L. Chen vorgestellt [AC77]. Es müssen $N \geq 2$ funktional gleiche Versionen der Software ausgehend von derselben Spezifikation erstellt werden, die die Eigenschaft besitzen, parallel ausführbar zu sein. Zusätzlich müssen zu festgelegten Zeitpunkten die Ergebnisse vergleichbar sein, sodass von der Ausführungsumgebung eine Entscheidung über das richtige Ergebnis getroffen werden kann. Der Ansatz entspricht im Wesentlichen dem Ansatz der TMR für Hardware mit dem Unterschied, dass die Versionen sich unterscheiden. Entscheidend ist, dass die verschiedenen Versionen eines Moduls parallel ausgeführt werden.

Recovery Blocks. Auch Recovery Blocks basieren auf mehreren funktional identischen Versionen der Software. Im Gegensatz zu N-Version Programming wird jedoch immer nur eine Version tatsächlich ausgeführt. Zusätzlich sind Akzeptanz-Tests notwendig, die entscheiden können, ob das Ergebnis einer Funktion fehlerfrei ist. Ist dies nicht der Fall, so kann ein Zustand vor Funktionsausführung, Recovery Block genannt, geladen werden. Von diesem Recovery Block aus wird die Berechnung von einer der anderen, funktional identischen Software-Versionen ausgeführt.

[3] Die verschiedenen Implementierungen werden auch als *Versionen* bezeichnet, sind allerdings nicht mit aufeinander folgenden Software-Versionen zu verwechseln.

Der Erfolg oder Misserfolg beider diversitärer Ansätze hängt an der Frage, ob ein Fehler, der in einer Version aufgetreten ist, in den anderen Versionen *nicht* auftritt. Es scheint nahe liegend, dass die Wahrscheinlichkeit für die Fehlertoleranz steigt, je größer die Diversität unter den Versionen ist, je mehr sich also die Versionen in ihren Implementierungsdetails unterscheiden. A. Avižienis fordert daher, Diversität über verschiedene Elemente in den Versionen zu fördern, dies reicht von unterschiedlich erfahrenen, räumlich getrennten Programmierteams über unterschiedliche Programmiersprachen und Implementierungswerkzeuge bis hin zu verschiedenen Softwareentwicklungsprozessen [Avi95]. Ferner wird Diversität auf verschiedenen Ebenen von Systemen vorgeschlagen.

Auf die Probleme diversitärer Programmierung wird in Abschnitt 6.1.2.1 ausführlich eingegangen und erläutert, warum sich diversitäre Programmierung in der Praxis nicht als Verfahren zur Toleranz von Softwarefehlern durchsetzen konnte. Dabei wird der Ansatz der Fehlertoleranz durch Diversität auf Management-Ebene, der in dieser Arbeit vorgestellt wird, motiviert.

3.4 Zusammenfassung

In diesem Kapitel wurden Grundlagen zu Fehlertoleranz erklärt und notwendige Begriffe eingeführt. Fehlertoleranz ist eine Maßnahme zur Konstruktion zuverlässiger Systeme, da trotz der Anwesenheit von Fehlerursachen das Versagen des Systems verhindert wird. Dazu müssen Fehlerzustände erkannt, und durch Redundanz das Versagen vermieden werden.

Fehlererkennung basiert generell auf dem Vergleich von Ist- und Soll-Zuständen und wird in Form von Akzeptanz-Tests realisiert. Abhängig von den Informationen über den Soll-Zustand eines Systems können unterschiedliche Kriterien während der Ausführung überprüft werden, wie beispielsweise Timing, Struktur oder Semantik.

Fehlertoleranz-Verfahren beheben einen Fehlerzustand entweder durch Rückwärtsbehebung, also erneute Ausführung ab einem zuvor gespeicherten Zustand, oder durch Vorwärts- bzw. Seitwärtsbehebung, also durch ein Ändern des Fehlerzustands in einen korrekten Zustand.

Die Toleranz von physischen Fehlern der Hardware ist durch strukturelle Redundanz erreichbar, da von der Abwesenheit grundsätzlicher Fehlerursachen in den Komponenten auszugehen ist. Transiente Fehler können durch NMR-Systeme to-

leriert werden, permanente Fehler durch standby-Komponenten, also den transparenten Austausch der defekten mit einer ungenutzten Komponente.

Die Toleranz von Softwarefehlern zeigt sich in der Praxis als wesentlich problematischer, da Softwarefehler von Anfang an in jeder Instanz der Software vorhanden sind. Diversitäre Programmierung ist ein Ansatz, der das Prinzip der NMR-Systeme auf Software zu übertragen versucht und später in dieser Arbeit diskutiert wird. Ansonsten existieren lediglich Verfahren zur Toleranz vorhersehbarer, festgelegter Fehlerzustände, die nicht zur Toleranz beliebiger Softwarefehler einsetzbar sind.

Kapitel 4

Fehlererkennung in MoDiS

Inhalt

4.1. Systembeobachtung 54

4.2. Integration des Vertragskonzepts in INSEL 79

4.3. Zusammenfassung 95

Im Rahmen der vorliegenden Arbeit wurden zwei Verfahren in das MoDiS-Projekt integriert, die zur Fehlererkennung in verteilten, nebenläufigen Systemen eingesetzt werden können. Erstens wurden Mechanismen zur Systembeobachtung auf Basis kausaler Abhängigkeiten für Systeme mit gemeinsamem, verteilten Speicher erarbeitet und im MoDiS-Management umgesetzt. Zweitens wurde nach dem Prinzip von *Design by Contract* die Sprache INSEL um das Konzept der Verträge erweitert. Die Übersetzer-Komponente des MoDiS-Managements wurde erweitert, sodass Code zur Überprüfung der Verträge generiert wird. Beide Ansätze dienen in der Theorie nicht ausschließlich der Fehlererkennung, werden im Rahmen dieser Arbeit aber dazu genutzt. Das in den Kapiteln 5 und 6 beschriebene Verfahren zur Fehlertoleranz durch Rückwärtsbehebung basiert auf den Fehlererkennungsmechanismen dieses Kapitels, wäre jedoch auch mit anderen Mechanismen zur Fehlererkennung kombinierbar. Das Ziel der beiden Verfahren ist die Erkennung einer möglichst großen Klasse an Fehlern zur Laufzeit, um im Management geeignet darauf reagieren zu können.

In Abschnitt 4.1 wird der Mechanismus zur Systembeobachtung verteilter Systeme erläutert. Anschließend werden in Abschnitt 4.2 das Vertragskonzept erklärt

und die Integration in INSEL vorgestellt. In beiden Abschnitten wird auf die jeweilige Eignung der Verfahren zur Fehlererkennung eingegangen.

4.1 Systembeobachtung

In verteilten, nebenläufigen Systemen lassen sich verschiedene Aufgaben auf das generelle Problem zurückführen, eine konsistente Sicht auf ein System in Ausführung konstruieren zu können. Zu diesen Aufgaben zählen unter anderem Systembeobachtung, Fehlererkennung, Debug-Verfahren sowie der Beweis von Systemeigenschaften bzw. Prädikaten. Das nahe liegende Vorgehen zur Konstruktion einer Sicht auf ein verteiltes, nebenläufiges System wäre die Speicherung des Systemzustands jedes beteiligten Akteurs zu einem festgelegten Zeitpunkt. Die Rechner-internen Uhren verteilter Systeme lassen sich allerdings nicht genügend exakt synchronisieren, dass eine globale, exakt synchrone Uhr realisierbar wäre. Es könnte nach dieser Methode also ein Zustand als Systemsicht gespeichert werden, der in der Ausführung auf Grund kausaler Abhängigkeiten nicht auftreten kann. Die kausalen Abhängigkeiten der Ereignisse der Akteure eines Systems (vgl. Abschnitt 2.2.4) sind deshalb die Grundlage zur Konstruktion einer konsistenten Sicht.

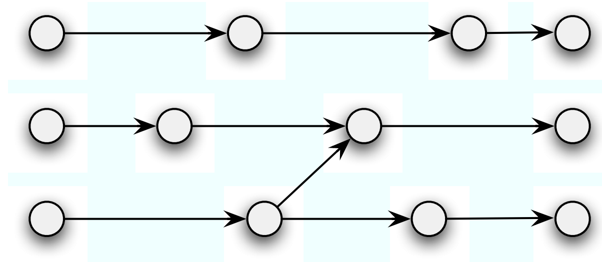


Abbildung 4.1.: Gerichteter Graph der kausalen Abhängigkeiten zwischen Ereignissen

Sind die kausalen Abhängigkeiten der Ereignisse der Akteure eines Systems in Form einer partiellen Ordnung bekannt, so lässt sich daraus ableiten, welches Systemverhalten bzw. welche Systemzustände in einer Ausführung möglich sind. Die partielle Ordnung lässt sich als gerichteter Graph darstellen, wobei die Knoten die Ereignisse und die Kanten die Abhängigkeiten repräsentieren. In Abbildung 4.1 ist ein solcher Graph visualisiert. Ein Ereignis kann nur eintreten, wenn die Vorgänger-Ereignisse im Graphen bereits eingetreten sind. Die Abhängigkeiten und Eigenschaften dieser Sicht auf ein System werden in den folgenden Abschnit-

ten definiert und erklärt. Ferner wird erläutert, mit welchen Verfahren eine solche konsistente, partielle Ereignisordnung konstruiert werden kann.

Seit den Arbeiten von L. Lamport im Jahr 1978 wurden kausale Abhängigkeiten mittels verschiedener logischer Uhren repräsentiert, darauf aufbauend konnten konsistente Sichten konstruiert werden (vgl. [Lam78]). Seitdem ist eine Fülle an Forschungsarbeiten zu diesem Thema entstanden, allerdings wurden dabei unterschiedliche, meist im Vergleich zur Praxis vereinfachte Systemmodelle betrachtet. Eine ausführliche Einführung zu den grundlegenden Ansätzen und Problemen haben Babaoğlu und Marzullo erarbeitet (vgl. [BM93]).

Die in der Literatur vorhandenen Lösungen sind ausschließlich Verfahren zur Erfassung konsistenter Sichten für Systemmodelle mit koexistierenden sequentiellen Prozessen mit separaten Zustandsräumen und fester, konstanter Prozessanzahl. Es existieren keine Arbeiten, die ein dynamisches Prozesssystem sowie einen gemeinsamen, verteilten Speicher berücksichtigen. Die kausalen Abhängigkeiten bei Systemen mit einem gemeinsamen, verteilten Speicher sind komplexer als bei den in der Literatur angenommenen Systemmodellen. Die vorhandenen Ansätze wurden deshalb zusammen mit T. Landes um diese zusätzlichen Abhängigkeiten erweitert und die theoretische Grundlage für eine Realisierung geschaffen (vgl. [PL05]).

Die in der Literatur beschriebenen Lösungen gehen zudem von einer endlichen, konstanten Zahl von Prozessen aus. In realisierten verteilten Systemen ist diese Einschränkung in der Regel nicht zutreffend. Zur Berechnung eines komplexen Problems mittels kooperierender, nebenläufiger Prozesse ist die Dynamik der Lebzeiten dieser Prozesse (oder Akteure) unabdingbar. In MoDiS werden Akteure dynamisch erzeugt und wieder aufgelöst, dies muss im Systemmodell zur Konstruktion der konsistenten Sichten erfasst werden. Die in der Literatur beschriebenen Realisierungen bauen auf logischen Uhren, insbesondere auf Vektoruhren auf, die nur für Systeme mit konstanter Zahl an Prozessen geeignet sind. In der Realisierung der erarbeiteten theoretischen Grundlagen für die Erfassung konsistenter Sichten in MoDiS wurde die Dynamik des Prozesssystems berücksichtigt und ein alternativer Ansatz verfolgt. In den folgenden Abschnitten wird zunächst das theoretische Modell zur Erfassung der kausalen Abhängigkeiten erläutert. In Abschnitt 4.1.2 werden dann die Realisierungsmöglichkeiten verglichen und der in dieser Arbeit verfolgte Ansatz vorgestellt.

4.1.1 Konsistente Sichten in MoDiS

Eine konsistente Sicht auf ein System muss den globalen Zustand eines Systems reflektieren, um für das System aussagekräftig im Sinne der genannten Anwendungsmöglichkeiten zu sein. Eine konsistente Sicht kann über die Ereignisse beschrieben werden, die enthalten bzw. bereits aufgetreten sind. Durch die den Ereignissen zugeordnete Wirkung ist implizit der Zustand des Systems dieser Sicht definiert. Die Ereignisse eines Systems sind partiell geordnet durch die kausalen Abhängigkeiten. Eine Systemsicht ist demnach festgelegt durch eine Partitionierung der Ereignisse in *Vergangenheit* und *Zukunft*. Da die Ereignisse jedes Akteurs total geordnet sind, ist eine solche Partitionierung eindeutig durch das jeweils letzte enthaltene Ereignis jedes Akteurs festgelegt. Das Präfix der geordneten Ereignisse eines Akteurs ist eine Akteur-Ereignisfolge:

Definition 4.1 (Akteur-Ereignisfolge)
Seien $e^1_{a_m}, e^2_{a_m}, \ldots, e^n_{a_m}$ die total geordneten Ereignisse eines Akteurs $a_m \in A$. Eine Akteur-Ereignisfolge *$\eta^j_{a_m}$ ist ein endliches Präfix dieser Ereignisse $e^1_{a_m}, \ldots, e^j_{a_m}$ mit $1 \leq j \leq n$.*

Die Partitionierung der Ereignisse des Systems lässt sich mit Hilfe der Akteur-Ereignisfolgen festlegen, eine solche Partitionierung wird auch als *Schnitt* bezeichnet:

Definition 4.2 (Schnitt)
Ein Schnitt *C ist eine Teilmenge der aufgetretenen Ereignisse eines Systems und enthält von jedem Akteur $a_m \in A$ des Systems eine Akteur-Ereignisfolge $\eta^j_{a_m}$. Seien $\eta^i_{a_1}, \ldots, \eta^j_{a_n}$ die Akteur-Ereignisfolgen der n Akteure eines Schnitts C, dann ist C durch das n-Tupel $(e^i_{a_1}, \ldots, e^j_{a_n})$ der jeweils letzten Ereignisse jeder Akteur-Ereignisfolge $\eta^i_{a_1}, \ldots, \eta^j_{a_n}$ eindeutig festgelegt.*

Zur Vereinfachung wird in dieser Arbeit auch häufig nur das Tupel der letzten Ereignisse jedes Akteurs angegeben, das den Schnitt beschreibt, beispielsweise $C(e^i_{a_1}, \ldots, e^j_{a_n})$. Ein Schnitt ist konsistent, wenn keine kausalen Abhängigkeiten durch ihn verletzt werden, wenn also für alle Ereignisse des Schnitts gilt, dass deren kausale Vorgänger ebenfalls im Schnitt enthalten sind. Im Rahmen dieser Arbeit wird kausale Abhängigkeit wie folgt definiert:

Definition 4.3 (Kausale Abhängigkeit)
Ist ein Ereignis $e^i \in E$ die Ursache für das Eintreten eines Ereignisses $e^j \in E$, so ist e^j von e^i kausal abhängig, *in Zeichen $e^i \rightarrow e^j$.*
Gilt $e^i \rightarrow e^j$ und $e^j \rightarrow e^k$, so gilt auch $e^i \rightarrow e^k$.

Ist ein Ereignis e^j von e^i kausal Abhängig, so wird e^i auch als *Ursache* oder *Vorgänger*, e^j als *Wirkung* oder *Nachfolger* bezeichnet. Der Grund der kausalen Abhängigkeit ist in der Informatik die Möglichkeit des Informationsflusses: Kann von einem Ereignis e^i Information zu einem Ereignis e^j fließen bzw. kann e^j die von e^i zur Verfügung gestellten Informationen nutzen, so ist e^i Ursache von e^j. In dieser Arbeit wird z.T. unterschieden, ob die kausale Abhängigkeit nur auf Grund der Transitivität besteht, dann wird ein Ereignis als *transitiv abhängig* bezeichnet, sonst als *direkt abhängig*. Dies entspricht der in Abbildung 4.1 visualisierten Abhängigkeit. Ein konsistenter Schnitt ist demnach wie folgt definiert:

Definition 4.4 (Konsistenter Schnitt)
Sei $C(e^i_{a_1}, \ldots, e^j_{a_n})$ ein Schnitt. Der Schnitt C ist genau dann konsistent, *wenn folgende Bedingung gilt:*
$\forall e^k, e^l \in E : (e^k \rightarrow e^l \wedge e^l \in C)$ *impliziert*[1] $e^k \in C$.

Jeder nach obiger Definition konsistente Schnitt entspricht einem Systemzustand, der tatsächlich in einem Systemablauf auftreten könnte. Um einen konsistenten Schnitt zu konstruieren, bzw. zur Laufzeit des Systems zu erfassen, müssen die kausalen Abhängigkeiten der Ereignisse des Systems festgelegt sein. Es ist eine konsistente partielle Ordnung der Ereignisse zu finden. Eine partielle Ereignis-Ordnung ist konsistent, falls sie den kausalen Abhängigkeiten entspricht. Eigenschaften des Systems können anhand dieser Systemzustände erkannt bzw. bewiesen werden. Ist beispielsweise ein definierter Schnitt, der einen Fehlzustand repräsentiert, konsistent, so ist das System fehlerhaft. Ist die partielle Ordnung bekannt, die die kausalen Abhängigkeiten der Ereignisse repräsentiert, so ist der gesamte mögliche Raum an Ausführungsabläufen festgelegt und jeder mögliche Zustand kann auf Erreichbarkeit überprüft werden.

Es müssen demnach zunächst die kausalen Abhängigkeiten der Ereignisse von Systemen mit gemeinsamem, verteiltem Speicher und Nachrichtenkommunikation identifiziert werden und anschließend Mechanismen zu deren Erfassung zur Laufzeit erarbeitet werden.

Die n Ereignisse E_{a_m} eines Akteurs $a_m \in A$ sind durch die so genannte *Program Order* total geordnet, in Zeichen $e^i_{a_m} \overset{po}{\Rightarrow} e^{i+1}_{a_m}, 1 \leq i < n$. Die *Program Order* ist die durch sequentielle Ausführung festgelegte total geordnete Reihenfolge der Ereignisse eines Prozesses bzw. Akteurs. Zusätzlich entstehen kausale Abhängigkeiten durch das Versenden von Nachrichten. Der Empfang einer Nachricht m hat offensichtlich als kausale Ursache das Versenden derselben Nachricht. Somit gilt

[1]Auf Grund der Verwechselungsgefahr mit anderen Pfeilsymbolen dieser Arbeit wird auf die Verwendung des Implikations-Pfeils in Definitionen verzichtet.

$e^k \rightarrow e^l$, falls e^k das Sende-Ereignis der Nachricht ist, die in Ereignis e^l empfangen wurde. Lamport hat diese Abhängigkeiten für Nachrichtenkommunikationssysteme in der *Happened-Before* Relation zusammengefasst (nach [Lam78]):

Definition 4.5 (Happened-Before)
Die Happened-Before-Relation $\overset{hb}{\Rightarrow}$ ist die kleinste Relation, die die folgenden Bedingungen erfüllt:
Falls $e^i \overset{po}{\Rightarrow} e^j$, dann $e^i \overset{hb}{\Rightarrow} e^j$.
Falls e^i eine Sende-Ereignis und e^j das Empfangs-Ereignis der selben Nachricht ist, so gilt $e^i \overset{hb}{\Rightarrow} e^j$.
Gilt $e^i \overset{hb}{\Rightarrow} e^j$ und $e^j \overset{hb}{\Rightarrow} e^k$, so gilt auch $e^i \overset{hb}{\Rightarrow} e^k$.

Die Happened-Before-Relation bildet die transitive Hülle über die durch Program Order und Sende-Empfangs-Beziehung bestehenden kausalen Abhängigkeiten. Gilt $e^i \overset{hb}{\not\Rightarrow} e^j$ und $e^j \overset{hb}{\not\Rightarrow} e^i$, so sind die beiden Ereignisse nebenläufig, in Zeichen $e^i \parallel e^j$. Werden die Ereignisse und ihre kausalen Abhängigkeiten als gerichteter Graph visualisiert, dann ist eine Linie, die die Ereignisse jedes Akteurs in zwei disjunkte Mengen Vergangenheit und Zukunft teilt, ein Schnitt. Der Schnitt ist konsistent nach Definition 4.4, falls keine gerichtete Kante von der Zukunft in die Vergangenheit zeigt. In Abbildung 4.2 ist dieser Zusammenhang für ein vereinfachtes Systemmodell ohne Prozesserzeugung und ohne gemeinsamen, verteilten Speicher dargestellt. Es existiert eine Sender-Empfänger-Abhängigkeit zwischen Ereignis $e^2_{a_3}$ und $e^3_{a_2}$. Schnitt C_1 verletzt diese Abhängigkeit, da $e^2_{a_3} \overset{hb}{\Rightarrow} e^3_{a_2}$ und $e^3_{a_2} \in C_1$ gelten, aber $e^2_{a_3} \notin C_1$. C_2 ist dagegen einer der möglichen konsistenten Schnitte, er ist in der Visualisierung dadurch erkennbar, dass keine gerichtete Kante die Linie von der rechten zur linken Ereignismenge durchkreuzt.

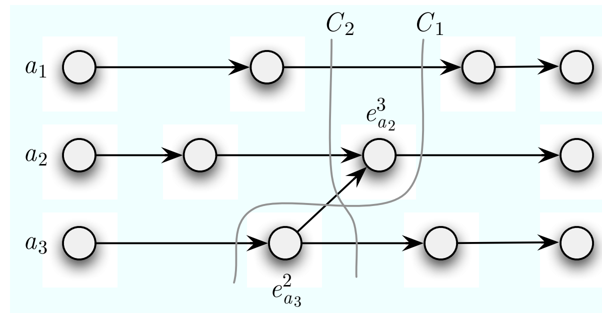


Abbildung 4.2.: Inkonsistenter Schnitt C_1 und konsistenter Schnitt C_2 in Nachrichtenkommunikationssystemen

In MoDiS ist die Nachrichtenkommunikation über das Operationen-orientierte Rendezvous-Konzept realisiert, das eine eingeschränkte Art der allgemeinen Nach-

richtenkommunikation darstellt. Die beteiligten Ereignisse können aber zunächst als Sende- und Empfangs-Ereignisse betrachtet werden. In Abschnitt 4.1.2.3 werden die genauen Ereignisse der Rendezvous-Kommunikation erklärt und ihre Abhängigkeiten beschrieben.

4.1.1.1 Erzeugung und Terminierung von Akteuren

Die beschriebene Happened-Before-Relation muss zunächst um die zusätzlichen kausalen Abhängigkeiten durch ein dynamisches Prozessmodell, wie das der Akteure in MoDiS, erweitert werden. In MoDiS existiert zu Beginn und Ende einer Systemausführung stets nur ein Akteur `SYSTEM`. Während der Ausführung kann jedoch jede Komponente weitere Akteure erzeugen. Der Erzeuger eines Akteurs kann erst terminieren, wenn alle erzeugten Akteure terminiert sind, wie in Abschnitt 2.2.2 zur ϵ-Struktur erklärt. Eine Komponente ist dabei stets in den Ausführungsfaden eines Akteurs eingeordnet, weshalb an dieser Stelle vereinfacht davon gesprochen wird, dass ein Akteur stets von einem Akteur erzeugt wird. Ein Akteur kann einen weiteren Akteur mittels eines `create`-Ereignisses erzeugen. Daraufhin wird vom Management der Akteur erzeugt und sein erstes Ereignis `init` ausgeführt. Der kausale Zusammenhang zwischen `create`- und `init`-Ereignissen ist analog zu einem Sende- und Empfangs-Ereignis in Nachrichtenkommunikationssystemen zu modellieren; `create` ist der kausale Vorgänger des Ereignisses `init`.

Das letzte Ereignis eines jeden Akteurs ist sein Terminierungs-Ereignis `term`, anschließend stellt das Management die Rückgabewerte des Akteurs dem Vater-Akteur zur Verfügung. Der Vater-Akteur kann nicht terminieren, ohne zuvor ein `join`-Ereignis für jeden erzeugten Akteur auszuführen, nach welchem das Management die terminierten Akteure endgültig auflöst. Das `join`-Ereignis ist kausal vom `term`-Ereignis des Kind-Akteurs abhängig, da die Ergebnisse des Kind-Akteurs fertig berechnet sein müssen. Die kausale Abhängigkeit ist demnach `term` $\rightarrow$ `join` und ebenfalls wie Nachrichtenkommunikation modellierbar. Diese Abhängigkeiten lassen sich in die Happened-Before-Relation für das MoDiS-Systemmodell folgendermaßen integrieren:

Definition 4.6 (Erweiterte Happened-Before-Relation)
Die erweiterte Happened-Before-Relation $\overset{ehb}{\Rightarrow}$ *ist die kleinste Relation, die die folgenden Bedingungen erfüllt:*
Falls $e^i \overset{po}{\Rightarrow} e^j$, *dann* $e^i \overset{ehb}{\Rightarrow} e^j$.
Falls e^i *eine Sende-Ereignis und* e^j *das Empfangs-Ereignis der selben Nachricht ist, so gilt* $e^i \overset{ehb}{\Rightarrow} e^j$.
Falls e^i *ein* `create`*-Ereignis und* e^1 *das zugehörige* `init`*-Ereignis des erzeugten*

Akteurs ist, so gilt $e^i \overset{ehb}{\Rightarrow} e^1$.
Falls e^i *ein* `term`*-Ereignis und* e^j *das zugehörige* `join`*-Ereignis des Vater-Akteurs ist, so gilt* $e^i \overset{ehb}{\Rightarrow} e^j$.

Zur späteren Weiterverwendung wurde die Transitivität an dieser Stelle als zusätzliche Relation eingeführt:

Definition 4.7 (Erweiterte transitive Happened-Before-Relation)
Die erweiterte Happened-Before-Relation wird mit dem Symbol $\overset{ehb^*}{\Rightarrow}$ *um die Transitivität ergänzt:*
$\forall e^i, e^j \in E: \ e^i \overset{ehb}{\Rightarrow} e^j$ *impliziert* $e^i \overset{ehb^*}{\Rightarrow} e^j$.
$\forall e^i, e^j, e^k \in E: \ (e^i \overset{ehb^*}{\Rightarrow} e^j \wedge e^j \overset{ehb^*}{\Rightarrow} e^k)$ *impliziert* $e^i \overset{ehb^*}{\Rightarrow} e^k$.

In die partielle Ordnung der Ereignisse, die durch die erweiterte transitive Happened-Before-Relation festgelegt ist, sind nun die kausalen Abhängigkeiten durch Zugriffe auf den gemeinsamen, verteilten Speicher zu integrieren.

4.1.1.2 Gemeinsamer verteilter Speicher

Die kausalen Abhängigkeiten in Systemen mit gemeinsamem, verteilten Speicher zu erfassen ist deutlich komplexer als in Nachrichtenkommunikationssystemen. Eine Nachricht hat je einen Sender und einen Empfänger; der Sender ist über den Inhalt der Nachricht identifizierbar und die Nachricht existiert nach dem Empfangsereignis nicht weiter. Greift dagegen ein Akteur auf eine Adresse des gemeinsamen Speichers zu, so ist nicht festgelegt, wer den Wert an diese Adresse geschrieben hat. Ferner ist nicht festgelegt, ob ein geschriebener Wert von keinem, einem oder mehreren Akteuren gelesen wird.

In MoDiS-Systemen wird vom Management ein gemeinsamer, verteilter Speicher mit sequentieller Konsistenz zur Verfügung gestellt, auf dem Schreibe- und Lese-Operationen durch die Ereignisse `write` und `read` ausführbar sind. Das Konsistenzmodell eines gemeinsamen Speichers legt fest, welche Werte bei einer bestimmten Abfolge von Zugriffen bei korrekter Implementierung als Resultat einer Lese-Operation zurück gegeben werden können. Die partielle Ereignisordnung ist demnach genau dann konsistent, wenn die Speicherzugriffe bei einer Ausführung gemäß der Ereignisordnung sequentielle Konsistenz realisieren.

Sequentielle Konsistenz kann informell wie folgt beschrieben werden (nach [Lam79] und [Ray02]): Eine Ausführung ist genau dann korrekt, wenn die nebenläufigen Operationen in eine totale Ordnung gebracht werden können, sodass

einerseits die *Program Order* jedes Prozesses bewahrt bleibt, und andererseits jede Lese-Operation genau *den* Wert liest, der gemäß der *totalen Ordnung* zuletzt an die Adresse geschrieben wurde.

Die *Program Order* wurde in den erarbeiteten Relationen bereits berücksichtigt, sodass zur Konstruktion einer konsistenten partiellen Ereignisordnung noch zu beachten ist, dass jede totale Ordnung, die der partiellen Ordnung genügt, die genannte Bedingung bzgl. gelesener Werte erfüllt. Die partielle Ordnung muss demnach jedes `read`-Ereignis nach einem passenden `write`-Ereignis ordnen, ohne ein weiteres `write`-Ereignis dazwischen. Passend bedeutet hier, dass der Wert, der durch das `write`-Ereignis geschrieben wird, der gleiche ist, der durch das `read`-Ereignis gelesen wird, und beide Ereignisse auf die selbe Speicheradresse zugreifen. Ereignisse, die verschiedene Speicheradressen betreffen, sind voneinander unabhängig und damit beliebig geordnet.

Jegliche totale oder partielle Ordnung, die diese Abhängigkeit berücksichtigt, ist konsistent. Existieren mehrere `write`-Ereignisse, die den gleichen Wert an die gleiche Adresse schreiben, so kann ein `read`-Ereignis beliebig nach einem dieser `write`-Ereignisse geordnet werden, falls dadurch keine anderen kausalen Abhängigkeiten aus der erweiterten Happened-Before-Relation verletzt werden. Damit ergeben sich insgesamt folgende Bedingungen an eine partielle Ordnung, um konsistent zu sein (vgl. [PL05]), wobei für Abkürzungen gelte:
e_w ist ein beliebiges `write`-Ereignis;
e_r ein beliebiges `read`-Ereignis;
E_W und E_R seien die Mengen aller `write`- und `read`-Ereignisse;
$E_{W(x)}$ und $E_{R(x)}$ seien die Mengen aller `write`- und `read`-Ereignisse auf Adresse x;
$E_{W(x)a}$ und $E_{R(x)a}$ seien die Mengen aller `write`- und `read`-Ereignisse auf Adresse x, die den Wert a gelesen bzw. geschrieben haben.

Theorem 4.1
Jede totale Ereignis-Ordnung $\succ$ ist genau dann konsistent, wenn sie folgende Bedingungen erfüllt:
(1) $\forall e^i, e^j \in E: \; e^i \overset{ehb}{\Rightarrow} e^j$ *impliziert* $e^i \succ e^j$.
(2) $\forall e \in E_{R(x)a}$ *muss eine der folgenden Bedingungen gelten*[2]*:*

(i) $\exists e_w \in E_{W(x)a}: \; (e_w \succ^* e) \wedge$
$(\forall e_x \in E_{W(x)} \setminus \{e_w\}: \; (e_x \succ^* e_w) \vee (e \succ^* e_x))$.
(ii) $\forall e_x \in E_{W(x)}: \; (e \succ^* e_x) \wedge$
$\forall e_r \in E_{R(x)b}:$ *Fall (i) gilt für* $a \neq b$.

[2]Mit $e^i \succ^* e^j$ wird ein transitiver Pfad in $\succ$ von e^i zu e^j bezeichnet

Beweis
Zuerst wird bewiesen, dass jede Bedingung *notwendig* ist, damit eine partielle Ordnung konsistent sein kann. Eine Ordnung ist genau dann konsistent, wenn sie während eines Systemablaufs auftreten könnte. Bedingung *(1)* folgt direkt aus den kausalen Abhängigkeitsbeziehungen nach Lamport, deren Notwendigkeit ist in [CL85] bewiesen. Es bleibt also zu zeigen, dass Bedingung *(2)* notwendig ist. Angenommen, eine Ordnung $\dot{\succ}$ sei konsistent, verletze aber Bedingung *(2)*. Es existieren zwei Möglichkeiten für die Ordnung, Bedingung *(2)* zu verletzen. In der ersten ordnet $\dot{\succ}$ ein `write`-Ereignis $e_x \in E_{W(x)b}$ zwischen ein `write`-Ereignis $e_w \in E_{W(x)a}$ und darauf folgende `read`-Ereignisse $e_r \in E_{R(x)a}$. Dies würde bedeuten, dass ein bereits überschriebener Wert gelesen wird, und stellt ein Widerspruch zur sequentiellen Konsistenz des gemeinsamen, verteilten Speichers dar. Die zweite Möglichkeit Bedingung *(2)* zu verletzen besteht darin, zwei `read`-Ereignisse $e_i \in E_{R(x)a}$ und $e_j \in E_{R(x)b}$, die zwei verschiedene Werte $a \neq b$ lesen, vor dem ersten `write`-Ereignis $e_w \in E_{W(x)}$ dieser Speicheradresse zu ordnen. Da diese Lese-Reihenfolge in einer realen Ausführung nicht vorkommen könnte, ist diese Ordnung nicht konsistent.

Nun wird bewiesen, dass die in Theorem 4.1 aufgestellten Bedingungen *hinreichend* für die Konsistenz jeder Ordnung sind, die die Bedingungen erfüllt. Angenommen eine Ordnung $\dot{\succ}$ genügt den Bedingungen, sei aber inkonsistent. Dies würde bedeuten, dass nach $\dot{\succ}$ ein Ereignis e_i vor Ereignis e_j geordnet wird ohne die Bedingungen zu verletzen und dennoch diese Reihenfolge in einer tatsächlichen Ausführung nicht auftreten könnte. Es existieren vier Möglichkeiten für eine Ordnung inkonsistent zu sein. Die ersten drei Fälle sind Verletzungen der kausalen Abhängigkeiten von *Program Order*, Sender-Empfänger-Abhängigkeit und `create-init`- bzw. `term-join`-Abhängigkeiten. Die Verletzungen dieser Abhängigkeiten würden direkt Bedingung *(1)* verletzen und führen daher zu einem Widerspruch. Die vierte Möglichkeit zur Inkonsistenz von Ordnung $\dot{\succ}$ ist eine Verletzung der sequentiellen Konsistenz des gemeinsamen, verteilten Speichers. Ein `read`-Ereignis $e_i \in E_{R(x)a}$ kann im System nur auftreten, falls entweder vorher der Wert a auf Adresse x durch ein `write`-Ereignis $e_w \in E_{W(x)a}$ geschrieben wurde, oder aber seit Systemstart noch kein `write`-Ereignis $e_w \in E_{W(x)}$ auf diese Adresse stattgefunden hat und a der einzige (initial im Speicher vorhandene) Wert ist, der vor dem ersten `write`-Ereignis $e_w \in E_{W(x)}$ auf diese Adresse gelesen wird. Die – nach Annahme – inkonsistente Ordnung $\dot{\succ}$ müsste demnach entweder Bedingung *(2)(i)* oder Bedingung *(2)(ii)* verletzen, was ebenfalls zu einem Widerspruch führt. □

Definition 4.8 (Relation kausaler Abhängigkeiten)
Eine Relation kausaler Abhängigkeiten $\overset{c}{\Rrightarrow}$ *ist jegliche transitive Ordnungsrelation, die die in Theorem 4.1 geforderten Bedingungen erfüllt.*

Jede partielle oder totale Ereignis-Ordnung, die der Relation kausaler Abhängigkeiten genügt, ist für das vorgestellte Systemmodell konsistent. Ein größeres Problem stellt die algorithmische Konstruktion einer solchen Ereignisordnung dar, die diese Relation erfüllt. Gibbons und Korach zeigten in ihrer Arbeit über die Verifikation von Konsistenz-Modellen, dass die Konstruktion einer totalen Ordnung der Ereignisse eines gemeinsamen, verteilten Speichers, ohne weitere Informationen NP-vollständig ist (vgl. [GK97]). Um eine Ordnung zu konstruieren, die der Relation kausaler Abhängigkeiten genügt, müssen folgende Probleme gelöst werden:

- Eine totale Ordnung mehrerer `write`-Ereignisse $e_w \in E_{W(x)}$ der gleichen Speicheradresse x muss gefunden werden, sodass keine anderen Abhängigkeiten der Relation verletzt werden.
- Jedes `read`-Ereignisses $e_r \in E_{R(x)a}$ muss zu einem passenden `write`-Ereignis $e_w \in E_{W(x)a}$ zugeordnet werden, ohne die anderen Abhängigkeiten der Relation zu verletzen.

Für das in dieser Arbeit vorliegende Problem können zusätzliche, zur Laufzeit gesammelte Informationen die Komplexität der Konstruktion deutlich vereinfachen. Einerseits muss die Ordnung aller schreibenden Zugriffe auf den gemeinsamen Speicher protokolliert werden, sodass auf Grund der Exklusivität schreibender Zugriffe eine totale Ordnung $\succ_{wo}$ über alle `write`-Ereignisse $e_w \in E_{W(x)}$ einer Speicheradresse x entsteht. Ferner wird eine Abbildung

$$f_{rm} : E_R \mapsto E_W \cup \{\perp\}$$

benötigt, die jedem `read`-Ereignis $e_r \in E_{R(x)a}$ genau das `write`-Ereignis $e_w \in E_{W(x)a}$ zuordnet, das zur Ausführungszeit den letzten schreibenden Zugriff auf diese Speicheradresse x ausgeführt hat. Diese Information ist ebenfalls nur durch Protokollierung der Zugriffe zur Laufzeit erhältlich. Dabei ist das zusätzliche Zeichen $\perp$ eingeführt, um den seltenen Fall[3] zu markieren, dass während eines Systemablaufs auf eine Adresse des Speichers, ohne ein vorheriges Schreiben eines der Akteure auf diese Speicheradresse, zugegriffen wurde.

Die Beachtung dieser zur Laufzeit aufgetretenen und protokollierten Reihenfolgen, also der Ordnung $\succ_{wo}$ und einer Ordnung gemäß der Abbildung f_{rm}, schränkt die Anzahl verschiedener, konstruierbarer, partieller Ordnungen ein. Es

[3] Dieser Fall ist in der Regel ein Fehler in der Programmierung, dennoch muss er in die Ordnung integriert werden, um Konsistenz zu bewahren.

gibt also in der Regel mehr konsistente, partielle Ordnungen, falls diese Beschränkungen nicht beachtet werden, allerdings ist sichergestellt, dass auch mit der Beschränkung mindestens eine konsistente, partielle Ordnung zu finden ist, da die Restriktionen bei einer tatsächlichen Ausführung protokolliert wurden. Mit Hilfe von $\succ_{wo}$ und f_{rm} kann die *eingeschränkte Relation kausaler Abhängigkeiten* basierend auf Definition 4.8 definiert werden (vgl. [PL05]):

Definition 4.9 (Eingeschränkte Relation kausaler Abhängigkeiten)
Sei $\succ_{wo}$ die zur Laufzeit protokollierte, totale Ordnung aller `write`-Ereignisse $e_w \in E_{W(x)}$ jeweils einer Speicheradresse. Sei $f_{rm} : E_R \mapsto E_W \cup \{\bot\}$ eine Abbildung, die jedem `read`-Ereignis $e_r \in E_{R(x)a}$ genau das `write`-Ereignis $e_w \in E_{W(x)a}$ zuordnet, das zur Ausführungszeit den letzten schreibenden Zugriff auf diese Speicheradresse x ausgeführt hat. Die eingeschränkte Relation kausaler Abhängigkeiten *$\stackrel{rc}{\Rightarrow}$ ist die kleinste Relation, die folgende Bedingungen erfüllt:*
(1) $\forall e^i, e^j \in E:\ e^i \stackrel{ehb}{\Rightarrow} e^j$ impliziert $e^i \stackrel{rc}{\Rightarrow} e^j$.
(2) $\forall e^i, e^j \in E_{W(x)}:\ e^i \succ_{wo} e^j$ impliziert $e^i \stackrel{rc}{\Rightarrow} e^j$.
(3) $\forall e \in E_{R(x)a}$ muss eine der beiden folgenden Bedingungen gelten:
(i) $\exists e_w \in E_{W(x)a}:\ (f_{rm}(e) = e_w) \wedge (e_w \stackrel{rc}{\Rightarrow} e) \wedge$
$(\forall e_x \in E_{W(x)} \setminus \{e_w\}:\ e_w \succ_{wo} e_x$ impliziert $e \stackrel{rc}{\Rightarrow} e_x)$.
(ii) $(f_{rm}(e) = \bot) \wedge (\forall e_x \in E_{W(x)} \setminus \{e_w\}:\ e \stackrel{rc}{\Rightarrow} e_x)$.
(4) $\forall e^i, e^j, e^k \in E:\ (e^i \stackrel{rc}{\Rightarrow} e^j \wedge e^j \stackrel{rc}{\Rightarrow} e^k)$ impliziert $e^i \stackrel{rc}{\Rightarrow} e^k$.

Für das in dieser Arbeit vorgestellte Systemmodell (vgl. Kapitel 2) bildet die Relation kausaler Abhängigkeiten (vgl. Definition 4.8) die transitive Hülle der geforderten und allgemein definierten kausalen Abhängigkeiten aus Definition 4.3. Die *eingeschränkte Relation kausaler Abhängigkeiten* ist eine stärker restringierende Version, die ebenso die kausalen Abhängigkeiten des Systemmodells beachtet, über dies hinaus aber zusätzliche Freiheitsgrade der partiellen Ordnungen einschränkt, um eine effiziente Konstruktion der konsistenten Ereignisordnungen zu ermöglichen. Da alle kausalen Abhängigkeiten in $\stackrel{rc}{\Rightarrow}$ enthalten sind, ist ein Schnitt C (vgl. Definition 4.2) genau dann konsistent, wenn folgende Bedingung gilt:
$\forall e^i, e^j \in E : (e^i \stackrel{rc}{\Rightarrow} e^j \wedge e^j \in C)$ impliziert $e^i \in C$.

In Abbildung 4.3 ist ein Ereignisgraph dargestellt, in dem lediglich die in der erweiterten Happened-Before-Relation (vgl. Definition 4.6) erfassten Abhängigkeiten (Program Order, Nachrichtenversand, Erzeugung und Terminierung von

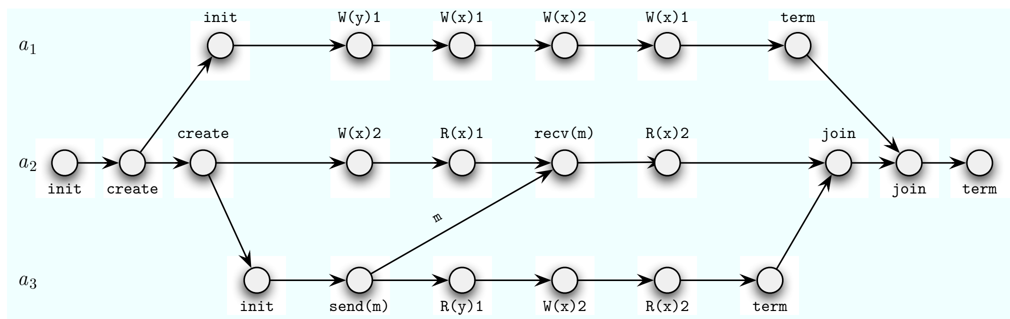


Abbildung 4.3.: Ereignisgraph ohne Abhängigkeiten des Speicherzugriffs

Akteuren) enthalten sind[4]. Die durch Zugriff auf den gemeinsamen Speicher verursachten Abhängigkeiten fehlen. Offensichtlich reicht die Information nicht, um Aussagen über konsistente Schnitte des Systems zu treffen.

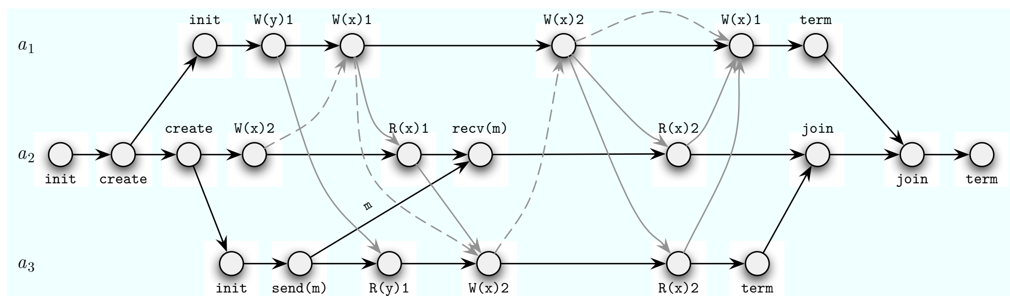


Abbildung 4.4.: Ereignisgraph mit allen kausalen Abhängigkeiten

In Abbildung 4.4 sind in das Beispiel aus Abbildung 4.3 zusätzlich die Abhängigkeiten durch $\succ_{wo}$ grau gestrichelt und die auf f_{rm} basierenden Abhängigkeiten in grauen Kanten eingezeichnet. Alle Kanten zusammen entsprechen der eingeschränkten Relation kausaler Abhängigkeiten $\stackrel{rc}{\Rightarrow}$. Anhand der dargestellten, partiellen Ordnung ist festgelegt, welche Schnitte konsistent und welche zugehörigen globalen Zustände im System folglich erreichbar sind. Auf Grund der Informationen der Ereignisse über deren Zugriff auf den gemeinsamen Speicher kann jedem Schnitt zudem ein globaler Systemzustand zugeordnet werden. Anhand des konstruierten Graphen können Prädikate, wie beispielsweise das nebenläufige Eintreten von Ereignissen, die einen kritischen Abschnitt markieren, überprüft

[4]Zur besseren Übersichtlichkeit wurden in der Abbildung die Ereignisse mit ihrer Funktionalität beschriftet, wobei $W(x)1$ einen Schreibzugriff auf Adresse x mit Wert 1 und analog $R(x)1$ einen entsprechenden Lesezugriff bezeichnet.

werden. Im Folgenden wird erläutert, wie ein entsprechender Graph konstruiert werden kann und welche Informationen dazu zur Laufzeit zu protokollieren sind.

4.1.2 Realisierung

Die Konstruktion des Ereignisgraphen wird zur Laufzeit vom Management des Systems durchgeführt. Die Ereignisse jedes Akteurs müssen protokolliert und mit den notwendigen Informationen ausgestattet werden. Auf Grund der Menge zu protokollierender Ereignisse ist für die Performanz des Systems entscheidend, die zu speichernden Informationen pro Ereignis möglichst gering zu halten. Die Information über die kausalen Abhängigkeiten nach $\stackrel{rc}{\Rightarrow}$ ist zur Konstruktion des Ereignisgraphen (bzw. zum Aufbau der partiellen Ordnung in beliebiger Repräsentation) notwendig. Darüber hinaus können Informationen an die Ereignisse geheftet werden, die nach der Konstruktion bei der auf der Ordnung basierenden Analyse genutzt werden. Zunächst wird der in der Literatur gängige Ansatz logischer Uhren vorgestellt und diskutiert. Anschließend wird eine Alternative erklärt, die auf der Speicherung direkter Abhängigkeiten basiert. Vor- und Nachteile dieses Ansatzes im Vergleich zu logischen Uhren werden diskutiert.

4.1.2.1 Logische Uhren

Logische Uhren basieren auf dem Prinzip, jedes Ereignis mit einem bzw. mehreren Werten zu versehen, die einer virtuellen Zeit entsprechen. Der Uhrwert eines Ereignisses wird festgelegt, indem er im Vergleich zu seinen kausalen Vorgängern vergrößert wird. Bekannte Vertreter dieses Prinzips für Nachrichtenkommunikationssysteme sind *Vektoruhren*, bei denen jedem Ereignis ein Vektor skalarer Werte zugeordnet wird. Jeder Eintrag des Vektors ist einem Prozess des Systems zugeordnet, die maximale Anzahl möglicher Prozesse bestimmt demnach die Größe des Vektors. Der Vektor eines auftretenden Ereignisses wird gebildet, indem die Vektoren direkter, kausaler Vorgänger-Ereignisse elementweise verglichen werden und für jeden Eintrag der größere in den neuen Vektor übertragen wird. Zudem wird der Eintrag des neuen Vektors mit dem Index des eigenen Prozesses inkrementiert. Werden nun die Vektoren zweier beliebiger Ereignisse des Systems elementweise verglichen, so kann die Abhängigkeitsbeziehung bestimmt werden, wobei $v(e^i)$ den Vektor des Ereignisses e^i bezeichne:

- gilt elementweise $v(e^i) \leq v(e^j)$, dann folgt $e^i \rightarrow e^j$,
- gilt elementweise $v(e^j) \leq v(e^i)$, dann folgt $e^j \rightarrow e^i$,

- sonst $e^i \parallel e^j$, mit $e^i, e^j \in E$.

Vektoruhren sind ein geeignetes Mittel, sowohl direkte, als auch transitive kausale Abhängigkeiten effizient zu erkennen. Nach dem elementweisen Vergleich zweier Vektoren ist eindeutig festgelegt, ob die beiden Ereignisse kausal abhängig oder nebenläufig sind. In Abbildung 4.5 sind Vektoruhren für ein Systemmodell mit Nachrichtenkommunikation visualisiert. Bei der Übertragung von Nachrichten werden die Vektoren der Sende-Ereignisse mit übertragen und die Vektoren der Empfangs-Ereignisse angepasst. Jeder Eintrag eines Vektors entspricht somit dem letzten bekannten Ereignis des entsprechenden Prozesses. Beispielsweise ist für das letzte Ereignis von Prozess a_1 mit Vektor (4,4,2) das Ereignis mit Index 4 von Prozess a_2 und Ereignis mit Index 2 von Prozess a_3 das jeweils jüngste bekannte Ereignis. Alle kausalen Vorgänger des Ereignisses haben an jedem Eintrag des Vektors einen gleichen oder kleineren Wert.

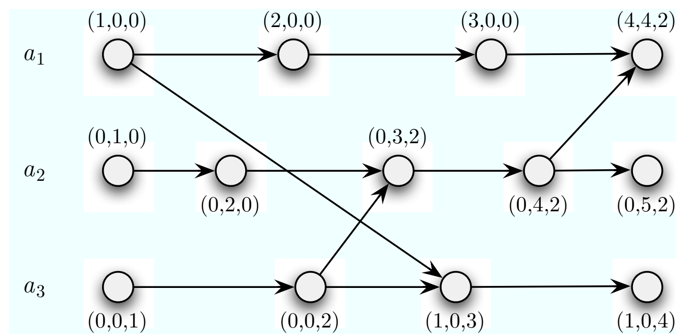


Abbildung 4.5.: Ereignisse mit Vektoren zur Reflexion kausaler Abhängigkeiten

Vektoruhren reflektieren kausale Abhängigkeiten von Ereignissen. Problematisch ist der Zusammenhang zwischen der Größe der Vektoren und der Prozessmenge. In einem System mit vielen Prozessen existieren häufig nicht zwischen allen Prozessen Kommunikationsbeziehungen; ein einzelner Prozess kann auch mit nur wenigen anderen Prozessen kommunizieren. Auf Grund der impliziten Zuordnung des Vektorindex zu Prozessen muss dennoch ein Vektor in der Größe entsprechend der Gesamtzahl aller im System befindlichen Prozesse gespeichert und verschickt werden. Dieses Problem wäre durch eine Matrix zu umgehen, indem eine zusätzliche Spalte zu jedem Vektor-Eintrag den zugeordneten Prozess-Identifikator enthält. Es verbleibt jedoch das Problem, dass die Größe des Vektors (bzw. der Matrix) sich daran orientiert, wie viele Prozesse bis zum aktuellen Zeitpunkt der Ausführung im System existiert haben. Das heißt, dass in einem dynamischen Prozesssystem, in dem Prozesse terminieren und neue Prozesse erzeugt werden, die Einträge bereits terminierter Prozesse nicht entfernt werden

können, da sonst nicht mehr alle Abhängigkeiten reflektiert und daraus falsche kausale Abhängigkeitsbeziehungen geschlossen würden.

Die Größe der Vektoren ist demnach nicht durch die in einem System zu einem Zeitpunkt existierenden Prozesse beschränkt, sondern durch die in einem System über den gesamten Ablauf erzeugten Prozesse. Diese Zahl ist unbegrenzt und kann in der Praxis bei lange laufenden Systemen groß genug werden, dass die Vektoruhren kein geeignetes Mittel mehr zur Repräsentation der kausalen Abhängigkeiten darstellen. Für Systeme mit dynamischer Erzeugung und Auflösung von Prozessen wurde deshalb die im Folgenden vorgestellte Realisierung mit Verweisen direkter Abhängigkeiten gewählt.

4.1.2.2 Verweise direkter Abhängigkeiten

Auf Grund der kausalen Abhängigkeiten der verschiedenen Ereignisarten ist festgelegt, dass jedes Ereignis genau einen oder zwei kausale Vorgänger hat[5]. Werden systemweit eindeutige Identifikatoren an die Ereignisse vergeben, so können bei jedem Ereignis durch maximal zwei dieser Identifikatoren die direkten Abhängigkeiten gespeichert werden. Ist ein Ereignis $e^i_{a_m}$ eines Akteurs $a_m \in A$ von zwei Ereignissen $e^{i-1}_{a_m}$ und $e^j_{a_n}$ von $a_n \in A$ kausal abhängig, so werden die Identifikatoren $e^{i-1}_{a_m}$ und $e^j_{a_n}$ als Abhängigkeitsverweise mit Ereignis $e^i_{a_m}$ gespeichert.

Im Gegensatz zu Vektoruhren sind transitive Abhängigkeiten bei diesem Verfahren nicht implizit enthalten. Beim Vergleich zweier beliebiger Ereignisse kann demnach nicht ähnlich effizient zu Vektoruhren entschieden werden, ob die beiden Ereignisse geordnet oder nebenläufig sind. Dazu ist bei diesem Verfahren eine Pfadsuche im Abhängigkeitsgraph notwendig. Im Rahmen dieser Arbeit dienen die Abhängigkeiten dazu, einen Abhängigkeitsgraph analog zu Abbildung 4.4 zu konstruieren, mit dessen Hilfe Eigenschaften des Systems analysiert und Fehler erkannt werden können. Die Konstruktion eines solchen Abhängigkeitsgraphen ist auf Basis der gespeicherten direkten Abhängigkeiten effizient möglich. Es ist somit abhängig vom Einsatz der konsistenten Sicht, ob der hohe Aufwand der Vektoruhren in Systemen mit dynamischer Prozesserzeugung gerechtfertigt ist.

Ein weiterer Vorteil bzgl. der Performanz des in dieser Arbeit verfolgten Ansatzes besteht darin, dass in einem verteilten System auf Grund von Synchronisationsbedingten Wartezeiten häufig ungenutzte Rechenzeit auf einzelnen Stellen verbleibt, die für Berechnungen und Analysen des Managements genutzt werden

[5] Das Ereignis `init` des Akteurs `SYSTEM` stellt eine Ausnahme dar, da es das erste Ereignis im System ist und kein Vorgänger-Ereignis hat. Dies kann hier jedoch vernachlässigt werden, da lediglich die obere Schranke an Vorgängern für das Verfahren von Bedeutung ist.

können. Dies gilt auch für die Analysen auf Basis des konstruierten Abhängigkeitsgraphen. Die Performanz-Verluste der Anwendungsprozesse zur Speicherung der Abhängigkeiten, die bei Vektoruhren höher sind, fallen demnach mehr ins Gewicht als die Performanz-Verluste des Managements bei der Auswertung der Ereignisgraphen, da diese Auswertung zu gewissen Teilen mit ungenutzten Rechenkapazitäten erledigt werden kann und somit weniger negative Auswirkung auf die Performanz der Anwendung hat.

4.1.2.3 Konstruktion der Ereignisordnung

Die Konstruktion eines Ereignisgraphen gemäß $\overset{rc}{\Rightarrow}$ auf Basis gespeicherter, direkter Abhängigkeiten der Ereignisse erfordert die Protokollierung der Informationen zur Laufzeit. Jedes Ereignis muss mit einem systemweit eindeutigen Identifikator versehen werden. Ferner müssen beim Eintritt eines Ereignisses die Identifikatoren der kausalen Vorgänger zur Verfügung stehen, um die Abhängigkeiten zu speichern. Die Realisierung dieser Mechanismen wird im Folgenden erklärt (vgl. [LP06]), eine detaillierte Beschreibung der Implementierung ist [Haa05] zu entnehmen.

Protokollierung der Informationen. Die Bereitstellung systemweit eindeutiger Identifikatoren geschieht durch eine Kombination aus Akteur-Identifikator und Ereigniszähler, der die Ereignisse eines Akteurs aufsteigend indiziert. Die in $\overset{rc}{\Rightarrow}$ enthaltene *Program Order* ist demnach durch die Indizes der Ereignisse eines Akteurs bereits implizit enthalten. Ein expliziter Verweis auf das Vorgänger-Ereignis desselben Akteurs kann somit in der Implementierung zu Gunsten der Effizienz vermieden werden. Es muss also lediglich der eindeutige Identifikator des aufgetretenen Ereignisses und ggf. zusätzlich ein Identifikator eines kausalen Vorgängers eines anderen Akteurs gespeichert werden.

Die Nachrichtenkommunikation ist in MoDiS auf den Spezialfall des *Operationen-orientierten Rendezvous* beschränkt (vgl. Abschnitt 2.2.1.4). Der Kommunikationsablauf mit den enthaltenen Ereignissen ist in Abbildung 4.6 dargestellt. Ein Akteur ruft eine K-Order mit dem Ereignis `corder_callsend` auf und ist ab diesem Moment blockiert, in der Abbildung durch eine gestrichelte Linie visualisiert. Der K-Akteur nimmt den Aufruf mit dem Ereignis `corder_callrecv` an. Dies entspricht der ersten Abhängigkeit bzgl. Nachrichtenkommunikation nach $\overset{rc}{\Rightarrow}$, der Ereignisidentifikator von `corder_callsend` wird deshalb mit der Nachricht übertragen und bei dem Ereignis `corder_callrecv` als Abhängigkeit gespeichert. Anschließend wird die K-Order ausgeführt. Dies kann eine beliebige

Funktion sein, der Übersichtlichkeit halber wurde die Berechnung als einziges Ereignis zusammengefasst. Liegt das Ergebnis vor, so wird es in einer Nachricht an den Aufrufer der K-Order gesendet, zusammen mit dem Identifikator des Sende-Ereignisses `corder_retsend`, der beim Empfangsereignis `corder_retrecv` wiederum als Abhängigkeit gespeichert wird.

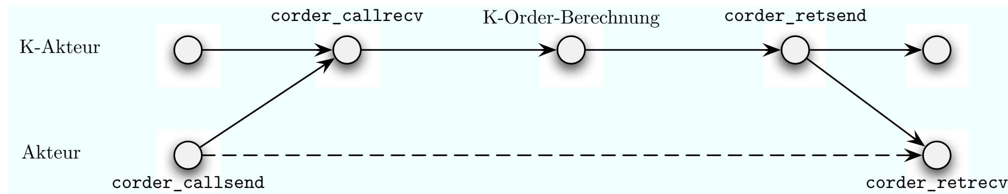


Abbildung 4.6.: Aufruf einer K-Order

Die Erzeugung und Terminierung von Akteuren wird – analog zum Versand der Nachrichten der K-Order – von der Laufzeitumgebung des MoDiS-Managements realisiert. Somit kann unabhängig vom Ort der Akteurerzeugung (auf der selben Stelle wie der Vater oder entfernt) der Identifikator des Ereignisses `create` mit übertragen werden und beim Ereignis `init` des erzeugten Akteurs als Abhängigkeit gespeichert werden. Analog wird beim Ereignis `join` des Vaters die Abhängigkeit zur Terminierung des Kindes, also der Identifikator des Ereignisses `term` gespeichert. Somit sind die Abhängigkeiten der Akteur-Struktur gemäß $\overset{rc}{\Rrightarrow}$ enthalten.

Problematisch ist die Realisierung der Protokollierung der Abhängigkeiten von Zugriffen auf den gemeinsamen, verteilten Speicher. In Abschnitt 2.3.3 wurde erklärt, dass der gemeinsame, verteilte Speicher in MoDiS seitenbasiert realisiert ist, sodass ein Akteur auf lokale Seiten des gemeinsamen Speichers analog zu seinem Prozess-lokalen Speicher effizient zugreifen kann. Ist die Speicherseite nicht lokal vorhanden, so wird ein Seitenfehler ausgelöst und die veränderte Routine zur Seitenfehlerbehandlung[6] des Kerns veranlasst das Management, die fehlende Speicherseite zu holen. Diese Art der Speicher-Realisierung hat zur Folge, dass Zugriffe auf lokal vorhandene Seiten des gemeinsamen Speichers ohne ein Beitragen des Laufzeit-Managements erfolgen. Somit ist die Protokollierung der Ereignisse nicht analog zu Prozess-Erzeugung bzw. Nachrichtenversand zu realisieren, da das Management nicht aufgerufen wird.

Die i386-Architektur bietet keine ausreichende Möglichkeit, die Zugriffe auf ausgewählte Speicherbereiche zu überwachen, um bei Zugriff den entsprechenden Code zur Protokollierung des Zugriffs auszuführen. Es existieren lediglich vier

[6] Die Routine wird auch englisch als *pagefaulthandler* bezeichnet.

Hardware-Debug-Register, mit Hilfe derer Speicherbereiche angegeben werden können, bei deren Zugriff eine kontrollierte Ausnahmebehandlung stattfindet. Dies würde eine zu große Einschränkung des gemeinsamen Adressraums bedeuten und ist somit nicht anwendbar. Eine weitere Möglichkeit wäre die Überwachung der Ausführung im Debug-Modus mit *Watchpoints* analog zu Werkzeugen wie *gdb* (gnu debugger, vgl. [SP02]). Diese Werkzeuge realisieren die Überwachung der Speicherzugriffe durch eine Ausführung im *Single-Step-Mode*, was zu Performanz-Einbußen des Faktors 100 führen würde. Dieser Ansatz ist deshalb ebenso nicht zur Realisierung geeignet.

Ein weiterer Ansatz wäre, das *memory-mapping* der Akteure zu verändern, sodass im Fall eines Zugriffs auf den gemeinsamen Adressraum immer ein Seitenfehler ausgelöst wird, sodass der pagefaulthandler die Protokollierung vornehmen kann. Dazu müsste sichergestellt werden, dass nach jedem erfolgten Speicherzugriff auf den gemeinsamen Adressraum die Seite wieder aus dem Speicher des Akteurs entfernt wird. Bei jedem Speicherzugriff – insbesondere bei mehreren, aufeinander folgenden – würde bei diesem Ansatz der Aufwand hinzukommen, die adressierte Seite in das memory-mapping des Akteurs zu laden und danach wieder zu entfernen. Dieser Aufwand wäre eher zu rechtfertigen, der Ansatz entspricht aber nach wie vor keiner zufrieden stellenden Lösung.

Ein weiteres Problem besteht im Erkennen der Art des Zugriffs. Unabhängig von den genannten Methoden zur Überwachung der Speicherzugriffe muss erkannt werden, ob ein *lesender* oder *schreibender* Zugriff auf den gemeinsamen Speicher stattfindet. Die einzelne Maschinenoperation ist eindeutig als Lese- oder Schreibzugriff identifizierbar. Die Ereignisse sind für ihren Einsatz allerdings auf dem Abstraktionsniveau von INSEL-Operationen, d.h. die Ereignisse, die Lese- und Schreibzugriffe modellieren, müssen von technischen Details wie Caches und Optimierungen durch den Übersetzer abstrahieren.

Diese beiden Probleme der Protokollierung der Ereignisse des Speicherzugriffs lassen sich elegant durch Modifikation des INSEL-Übersetzers *gic* lösen (vgl. Abschnitt 2.3.1). Im Übersetzer ist anhand der INSEL-Sprachkonstrukte einerseits direkt erkennbar, ob es sich um einen schreibenden Zugriff handelt (Zuweisungen) oder einen lesenden. Zudem ist im Übersetzer auf Grund der Deklaration bekannt, ob ein adressiertes Objekt im gemeinsamen oder lokalen Adressraum eines Akteurs liegt. Bei Zugriffen auf den gemeinsamen Speicherbereich wird vom Compiler zusätzlicher Code generiert, sodass der Akteur selbst nach einem Speicherzugriff einen Aufruf in die Laufzeitumgebung zur Protokollierung des Ereignisses ausführt. Der Speicherzugriff und die Protokollierung werden als kritischer Bereich atomar ausgeführt, sodass die protokollierten Ereignisse mit den sequentiellen Zugriffen konsistent bleiben.

Zur Protokollierung der Speicherzugriffs-Ereignisse wird die Speicheradresse als systemweit eindeutiger Identifikator verwendet. Um jedes Speicherzugriffs-Ereignis eindeutig identifizieren zu können, wird zusätzlich für jedes adressierbare Objekt des gemeinsamen Adressraums ein Zähler eingeführt. Der Zähler wird bei jedem Schreibzugriff auf dieses Objekt inkrementiert. Die Abhängigkeiten von Speicherzugriffs-Ereignissen gemäß $\overset{rc}{\Rrightarrow}$ werden gespeichert, indem bei einem Schreibzugriff ein Ereignis `write` des Akteurs mit Speicheradresse und inkrementiertem Zähler gespeichert wird. Die notwendige totale Ordnung der Schreib-Ereignisse für die Konstruktion des Ereignisgraphen ist durch den Zähler festgelegt. Bei lesendem Zugriff wird ein Ereignis `read` mit Adresse und Zähler gespeichert, *ohne* diesen zu inkrementieren. Somit ist die Zuordnung von `read`-Ereignissen zu `write`-Ereignissen gewährleistet.

Konstruktion der Ereignisordnung. Die protokollierten Ereignisse können – abhängig vom Ziel der Analyse – zur Laufzeit oder nach Terminierung des Systems zu einem gerichteten Graphen kombiniert und ggf. graphisch visualisiert werden. Im MoDiS-Projekt wurden die Graphen in *GML* (Graph Modelling Language, vgl. [Him96]), einer Beschreibungssprache für Graphen, erstellt und gespeichert, um für weitere Systemanalysen zur Verfügung zu stehen und mit Standard-Werkzeugen für GML-Graphen bearbeitet werden zu können. Zur Konstruktion des Graphen werden die Ereignisse gemäß der beschriebenen, gespeicherten, direkten Abhängigkeiten mit Kanten verbunden. Bei `write`-Ereignissen wird eine Abhängigkeit zum `write`-Ereignis mit kleinerem Zähler eingetragen, für `read`-Ereignisse wird eine Kante vom `write`-Ereignis mit gleichem Zähler zum `read`-Ereignis, und zudem vom `read`-Ereignis zum `write`-Ereignis mit nächst höherem Zähler eingetragen. Ein Beispiel für ein INSEL-Programm und dem dazugehörigen, während des Systemablaufs generierten Graphen ist in Anhang B zu finden. Zur Fehlererkennung wird die konstruierte Ereignisordnung genutzt, wenn auch die graphische Darstellung als Graph dafür nicht notwendig wäre.

4.1.3 Fehlererkennung auf Basis konsistenter Sichten

Eine der typischen Anwendungen der generierten konsistenten Sichten ist die Analyse des Systems durch Prädikatsbeweise. Auf Basis der generierten Ereignisordnung können Eigenschaften des Systems in Form von Prädikaten formuliert und deren Erreichbarkeit bzgl. der kausalen Abhängigkeiten überprüft werden. Allgemein entspricht dieses Vorgehen einem Model-Checking-Ansatz (vgl. Abschnitt 6.1.1.1), wobei üblicherweise die Modelle statisch aus Spezifikationen, nicht dynamisch zur Laufzeit gewonnen und analysiert werden. Es werden da-

bei keine Fehler zur Laufzeit erkannt, sondern die Erreichbarkeit eines Fehlerzustands anhand der Spezifikation geprüft. Angenommen, in einer Anwendung sei der wechselseitige Ausschluss mehrerer Prozesse implementiert. Anhand des Ereignisgraphen könnte überprüft werden, ob der wechselseitige Ausschluss korrekt spezifiziert ist, indem die Ordnung der Ein- und Austritts- Ereignisse verschiedener Prozesse geprüft wird. Existiert grundsätzlich eine Ausführungsmöglichkeit, sodass mehrere Prozesse gleichzeitig den kritischen Bereich betreten, so könnte dies im Ereignisgraphen erkannt werden.

Im Rahmen dieser Arbeit wurde die Systembeobachtung jedoch nicht zum Zweck der statischen Analyse nach der Terminierung des Systems, sondern zur Fehlererkennung zur Laufzeit integriert. Eine statische Analyse der Eigenschaften wäre also zusätzlich auf Basis der konstruierten Ordnung denkbar, ist jedoch nicht Teil dieser Arbeit. Zur Fehlererkennung dient der Ereignisgraph, da er eine Repräsentation der Ist-Situation der Systemausführung darstellt. Wie in Abschnitt 3.2 beschrieben, sind Mechanismen zur Fehlererkennung generell Vergleiche zwischen einem Ist- und einem Soll-Zustand. Der adäquate Soll-Zustand bzw. die Soll-Sicht auf das System ist durch die INSEL-Spezifikation gegeben. Die Spezifikation wird vom generativen Managementanteil durch die Attributierung (vgl. Abschnitt 2.3.1) aufbereitet und kann so zum Vergleich zwischen Ist- und Soll-Zustand der Ausführung genutzt werden.

Für viele Ansätze der Fehlertoleranz ist nicht nur die Fehlererkennung, sondern insbesondere die Fehlerlokalisierung und möglichst exakte Identifizierung der Fehlerursache relevant. Für den in dieser Arbeit verfolgten Ansatz zur Fehlertoleranz durch Diversität auf Management-Ebene (vgl. Kapitel 6) ist lediglich entscheidend festzustellen, dass ein Fehlerzustand aufgetreten ist, ohne genauer zu wissen welcher Art dieser Fehler ist bzw. Rückschlüsse auf dessen Fehlerursachen zu ziehen.

Aus diesem Grund wird der Ist-Soll-Vergleich auf Basis der konsistenten Systemsicht derart konstruiert, dass eine möglichst große Klasse an Fehlern erkannt werden kann. Auf Basis der Ereignisprotokollierung können Timing failures (vgl. Abschnitt 3.1) erkannt werden. Da im Ereignisgraphen die Wirkung der Berechnung, also die semantische Korrektheit der berechneten Ergebnisse nicht enthalten ist, kann die (vollständige) Klasse der byzantinischen Fehler durch diesen Vergleich nicht erkannt werden.

Welches Verhalten eines Systems ein Timing failure ist, hängt wiederum vom erwarteten Systemverhalten ab. Das Versagen eines Systems ist definiert als ein nach aussen sichtbares Verhalten, das nicht dem gewünschten entspricht. Im Projekt MoDiS ist keine Spezifikation zeitlicher Schranken oder Forderungen, wie

beispielsweise in Echtzeitsystemen, vorgesehen. Insofern entspricht die Klasse der Timing failures für diese Systeme der Klasse der Omission failures. Dennoch wären mit dem erarbeiteten Verfahren allgemeinere Timing-Fehler erkennbar, abhängig von der in der Spezifikation enthaltenen Information über das gewünschte (zeitliche) Verhalten. Für das vorgestellte Systemmodell im MoDiS-Projekt ist demnach lediglich relevant, Timing failures zu erkennen, bei denen eine Komponente für eine bestimmte Zeit *nicht* mehr auf bestimmte Eingaben reagiert.

Vereinfacht dargestellt wird zur Fehlererkennung das Auftreten der erwarteten Ereignisse, die durch den Übersetzer *gic* in Form der APNV zur Verfügung gestellt werden (vgl. Abschnitt 2.3.1), zur Laufzeit mit den tatsächlich aufgetretenen Ereignissen verglichen. Angenommen, die vom Übersetzer approximierten Laufzeiten für Berechnung und Kommunikation wären korrekt, so könnten diese approximierten Zeiten mit den tatsächlichen Zeiten der Ereignisse verglichen und bei Abweichung ein Timing failure gemeldet werden. Dieses Vorgehen wirft allerdings einige Probleme auf:

- Die Ausführungszeiten werden vom Übersetzer nur approximiert, die tatsächliche Ausführung kann davon deutlich abweichen, insbesondere durch Konkretisierungs-Entscheidungen des Managements wie beispielsweise Stellenzuweisung oder Scheduling.

- Die Ausführungszeiten der protokollierten Ereignisse stammen von verschiedenen Rechnern eines verteilten Systems und sind daher auf Grund der relativ ungenauen Uhrsynchronisation nur grob miteinander vergleichbar.

- Die Ereignisse des APNV sind nur eine Teilmenge der protokollierten Ereignisse der konstruierten Sicht.

Trotz dieser Hindernisse können auf Basis der Ist- und Soll- Ereignisgraphen Timing-Tests durchgeführt werden, um Fehler zu erkennen. Das Problem der Vergleichbarkeit der approximierten Ausführungszeiten mit den tatsächlichen Ausführungszeiten wird wie folgt gelöst: Auf Grund der feingranularen Ereignisprotokollierung der konstruierten Ereignisordnung wird erkannt, ob eine Komponente auf Grund der beschriebenen Abhängigkeiten auf andere Ereignisse warten muss, wie beispielsweise durch das Operationen-orientierte Rendezvous, durch DSM-Zugriffe sowie Start- und Abschlusssynchronisation. Werden die Laufzeiten zwischen diesen Ereignissen gesondert berücksichtigt, so verbleibt eine Annäherung an die tatsächliche Berechnungszeit der Komponenten.

Die durch den Übersetzer approximierten Ausführungszeiten müssen gewichtet werden, um mit den tatsächlichen Ausführungszeiten vergleichbar zu sein. Generell muss dabei ein Kompromiss zwischen der Wahrscheinlichkeit von *false positives* und der Verzögerung der Fehlererkennung getroffen werden. Je größer also die Gewichtung g der approximierten Zeiten t_a beim Vergleich mit den tatsächlichen Laufzeiten t_l, desto länger dauert es, bis ein aufgetretener Fehler tatsächlich erkannt wird, aber desto geringer ist auch die Wahrscheinlichkeit eines false positives, also dass die tatsächliche Ausführungszeit zwischen zwei Ereignissen so viel länger dauert, dass ein Fehler gemeldet wird, obwohl eigentlich keiner aufgetreten ist.

Eine mögliche Vorgehensweise für einen geeigneten Kompromiss dieser Gewichtung ist eine dynamische Bestimmung zur Laufzeit. Initial wird eine sehr hohe Gewichtung g gewählt, die dann sukzessive zur Laufzeit an die gemessenen Ausführungszeiten angepasst wird. Dieses Vorgehen berücksichtigt implizit die Ungenauigkeiten der Uhrsynchronisationen und kann bis zu einem unteren Schwellwert s_u und einem oberen Schwellwert s_o angepasst werden. Der Schwellwert gibt den Rahmen für den Kompromiss vor, also die gewünschte Wahl zwischen zeitnaher Erkennung von Fehlern und dem Vermeiden von false positives. Der Initialwert der Gewichtung wird absichtlich hoch gewählt, da die initiale Annäherung in den durch die Schwellen angegebenen Bereich von oben erfolgen muss, um nicht anfangs false positives zu melden. Wird beispielsweise als Initialwert der Gewichtung $g = 10$ und als Schwellwerte $s_u = 2$ und $s_o = 3$ gewählt, so wäre zu erwarten, dass das Gewicht zur Laufzeit auf einen Wert von $g = 3$ absinkt, falls die approximierte Zeit t_a in etwa der Laufzeit t_l entspricht. Generell wird zur Ausführungszeit in festen Intervallen das Gewicht g angepasst, sodass durchschnittlich folgender Zusammenhang gilt: $s_o \cdot t_l \geq g \cdot t_a \geq s_u \cdot t_l$. Somit ist sichergestellt, dass die Gewichtung dem durch die Schranken gewählten Kompromiss entspricht.

Für alle protokollierten Ereignisse, die auch im APNV enthalten sind, kann nun überprüft werden, ob sie die gewichtete, approximierte Ausführungszeit überschreiten, ob also $t_l > g \cdot t_a$ gilt. Ist dies der Fall, so müsste ein Fehler angenommen werden, allerdings müssen Wartezeiten durch Abhängigkeiten berücksichtigt werden. Aus diesem Grund wird überprüft, ob gemäß dem letzten protokollierten Ereignis der Komponente und deren gemäß APNV folgendem Ereignis ein Wartezustand vorliegen kann. Dies kann durch die beschriebenen Abhängigkeiten auftreten. In diesem Fall wird der Fortschritt *der* Komponente überprüft, auf deren Ereignis gewartet werden muss. Wird dies über mehrere Komponenten fortgesetzt, so wird geprüft, ob ein Zyklus zu einer der (potentiell) wartenden Komponenten besteht. Dies würde einen Hinweis auf eine Verklemmung bedeuten, die jedoch gesondert untersucht werden müsste. Da der Schwerpunkt dieser

Arbeit auf der Fehlertoleranz liegt, wird bei Verletzung der obigen Formel von einem Fehler ausgegangen, auch wenn dies false-positives nicht ausschließt.

Mit dem beschriebenen Verfahren lassen sich Timing failures erkennen. Dies schließt einerseits die meisten der typischen Hardware-Fehler in verteilten Systemen ein, wie beispielsweise den Absturz einzelner Stellen oder den Ausfall von Netzwerkverbindungen. Andererseits können sich auch viele Softwarefehler als Timing failures auswirken, wie beispielsweise Verklemmungen. In Anhang C ist ein einfaches INSEL-Beispiel gegeben, bei dem je nach Ausführungsreihenfolge eine Verklemmung auftreten kann. Dabei werden zwei Mutexe als K-Akteur-Instanzen `mutex_1` und `mutex_2` modelliert, die die K-Ordern `lock` und `unlock` anbieten. Zwei M-Akteure `act_1` und `act_2` benötigen jeweils beide Ressourcen, rufen jedoch die Sperrfunktionen in unterschiedlicher Reihenfolge auf:

```
mactor act_1 is
 begin
   mutex_1.lock;
   mutex_2.lock;
     -- ... kritischer Bereich ...
   mutex_2.unlock;
   mutex_1.unlock;
 end act_1;

 mactor act_2 is
 begin
   mutex_2.lock;
   mutex_1.lock;
     -- ... kritischer Bereich ...
   mutex_1.unlock;
   mutex_2.unlock;
 end act_2;
```

Offensichtlich kommt es immer dann zur Verklemmung, wenn nach dem ersten `lock`-Aufruf eines Akteurs der erste `lock`-Aufruf des anderen Akteurs stattfindet, bevor der zweite `lock`-Aufruf stattfinden konnte. Kann einer der beiden Akteure dagegen beide `lock`-Aufrufe vor dem anderen Akteur ausführen, so kann das Beispiel ohne Verklemmung terminieren. Im Fall der Verklemmung würden beide Akteure blockiert in ihrem K-Order-Aufruf warten, die jedoch nicht angenommen werden, solange kein `unlock` erfolgt.

Das Management würde in beiden Fällen nach Überschreitung der Zeitschranken einen Timing-Fehler melden. Es wäre zwar nicht sicher festgestellt, ob es sich tatsächlich um eine Verklemmung handelt, dazu müsste mit entsprechenden Algorithmen der Abhängigkeitsgraph näher untersucht werden. Für den in dieser Arbeit vorgestellten Ansatz ist das Wissen über den genauen Fehlerzustand bzw. dessen Fehlerursache allerdings nicht notwendig. In Abschnitt 6.3 wird beschrieben, wie die aufgetretene Verklemmung dieses Beispiels toleriert werden kann.

Die Mächtigkeit der Fehlererkennung eines Vergleichs zwischen Beobachtung des Systems in Ausführung und Systemspezifikation hängt von der Granularität der beiden Sichten auf das System ab. Je feiner die Sprachkonstrukte in festgelegte, protokollierbare Ereignisse gegliedert werden, umso genauer kann ein Vergleich mit der Ausführung zur Laufzeit durchgeführt werden. Im Rahmen dieser Arbeit wurde für Komponenten-lokale Softwarefehler das in der zweiten Hälfte dieses Kapitels beschriebene Vertragskonzept in die Sprache INSEL integriert. Für diese Arbeit ist demnach eine Fehlererkennung in systemglobaler Sicht ausreichend, in der erkennbar ist, ob die einzelnen Komponenten – auf hoher Abstraktionsebene – der Spezifikation entsprechend in ihren Berechnungen fortschreiten.

Die Systembeobachtung als Erfassung konsistenter Sichten hat darüber hinaus Potential, das Verständnis der komplexen Abhängigkeiten nebenläufiger Systeme zu verbessern. Die durch die Sprachkonstrukte festgelegten Strukturen müssten zur Laufzeit mit der Spezifikation verglichen und deren Einhaltung überprüft werden. Diese Systemanalysen gehen allerdings über den Rahmen der vorliegenden Arbeit hinaus.

4.1.4 Bewertung

Generell ist die Beobachtung von Systemen mit zwei Problemen verbunden: einerseits wirkt sich die Beobachtung eines Systems negativ auf die Systemperformanz aus, da ein Teil der vorhandenen Ressourcen, insbesondere der Rechenzeit, für die Systembeobachtung aufgewendet wird. Andererseits verändert die Beobachtung eines Systems dessen Verhalten und Eigenschaften. Dieses Phänomen wurde nach einem Gedankenexperiment des Quantenphysikers E. Schrödinger auch als *Schrödingers Katze* benannt (vgl. [Sch35]). Bei Systembeobachtungen, die ausschließlich während der Softwareentwicklung durchgeführt werden, um Fehler zu entdecken, stellt dies ein großes Problem dar, da die Systeme sich ohne die Beobachtungen anders verhalten, insbesondere die enthaltenen Fehler unter Umständen erst ohne die Beobachtung auftreten können. Das hier vorgestellte Verfahren soll im Nutzbetrieb des Systems eingesetzt werden, das System existiert also nur

inklusive der Systembeobachtung, deshalb kann das Problem von Schrödingers Katze hier vernachlässigt werden. Die Auswirkungen auf die Systemperformanz sind allerdings nicht zu vernachlässigen, sondern sind nur durch das Ziel der Zuverlässigkeit der konstruierten Systeme zu rechtfertigen.

Die eigentliche Konstruktion des Ereignisgraphen sowie die Fehlererkennung auf dessen Basis muss nicht zwingend von einer genutzten Stelle des Systems ausgeführt werden, zudem kann diesbezüglich, abhängig von den Systemanforderungen, mehr oder weniger Aufwand betrieben werden. Generell wirkt sich die Fehlererkennung allerdings negativ auf die Systemperformanz aus. Der Performanzverlust durch die Ereignisprotokollierung bleibt in jedem Fall relevant und wird hier bzgl. der Auswirkung auf die Performanz des Systems näher betrachtet.

Die Protokollierung eines Ereignisses entspricht wenigen Maschinen-Operationen, da lediglich ein Zähler erhöht und der Identifikator des Ereignisses sowie ggf. der Identifikator des kausalen Vorgängers eines anderen Akteurs gespeichert werden müssen. Bei der Protokollierung von Sende- und Empfangsereignissen ist die Ausführungsdauer des eigentlichen Nachrichtenversands im Vergleich zur Protokollierung der Ereignisse so viel länger, dass der Performanzverlust der Protokollierung vernachlässigbar ist. Bei der Erzeugung und Auflösung von Akteuren gilt dies ebenso. Bei Zugriffen auf den gemeinsamen Adressraum sind allerdings echte Performanzeinbußen durch die Protokollierung zu verzeichnen. Ist eine Speicherseite bereits auf der Stelle des zugreifenden Akteurs realisiert und in dessen memory-mapping enthalten, so kann auf den gemeinsamen Speicher analog zu Akteur-lokalem Speicher zugegriffen werden. Wird ein solcher Zugriff protokolliert, so sind zwischen vier und sechs zusätzliche Speicherzugriffe notwendig. Die Auswirkungen dieser zusätzlichen Speicherzugriffe auf die Gesamtperformanz des Systems hängt stark von der Anwendung ab, insbesondere von der Häufigkeit der Zugriffe auf den verteilten, gemeinsamen Speicher. Grundsätzlich ist auf Grund der Verteilung des Speichers dessen exzessive Benutzung mit hohen Performanzeinbußen verbunden, da Speicherseiten vor dem Zugriff unter Umständen erst von einer anderen Stelle angefordert und über das Netzwerk übertragen werden müssen. Die Performanzeinbußen in diesen Fällen wirken sich wesentlich stärker aus als die Ereignisprotokollierung.

4.2 Integration des Vertragskonzepts in INSEL

Ein interessantes Konzept zur Softwareentwicklung und Fehlererkennung stellte B. Meyer im Jahr 1992 unter dem Namen *Design by Contract* vor (vgl. [Mey92]). Dabei werden Vor- und Nachbedingungen für die Methoden von Objekten sowie Invarianten für Klassen spezifiziert, die einen Vertrag zwischen Aufrufer und Implementierung darstellen. Die Erfüllung der Verträge kann zur Laufzeit geprüft und dadurch Fehler erkannt werden. Neben den Tests zur Laufzeit bietet das Konzept – auf die richtige Weise realisiert – eine Vielzahl weiterer Vorteile zur Softwareentwicklung. Im folgenden wird das Konzept Design by Contract sowie die vergangene Entwicklung in Form der Umsetzungen des Konzepts in verschiedenen Sprachen, insbesondere in Java, diskutiert. Anschließend wird die im Rahmen dieser Arbeit entstandene Integration in die Sprache INSEL erläutert.

4.2.1 Design by Contract

Design by Contract (DbC) ist einerseits eine Methodik zur Modellierung von Software, die den Softwareentwicklungsprozess erleichtern und hinsichtlich Fehlerfreiheit verbessern soll. Andererseits besteht der Kern von DbC aus einer Implementierung von Laufzeitüberprüfungen der spezifizierten Verträge, um Fehler der Software zu erkennen, zu lokalisieren und Maßnahmen zu deren Toleranz anstoßen zu können.

Nach der Methodik von DbC werden beim Softwareentwurf abstrakte Spezifikationen für Klassen und Methoden erstellt, die deren Eigenschaften beschreiben. Diese Spezifikationen werden Verträge[7] genannt und enthalten Vor- und Nachbedingungen für Methoden sowie Invarianten für Klassen. Dabei wird analog zu einem Schichtenmodell von der Implementierung der Funktionalität abstrahiert: es ist nicht von Interesse *wie* eine Schicht einen Dienst erbringt, lediglich die Details zur Nutzung der Schnittstelle sind für die darüber liegende Schicht interessant. Für das Design eines Softwareprojekts ist zunächst nur das Ein- und Ausgabe-Verhalten einer Methode zu spezifizieren, ohne die Details der Implementierung festzulegen.

In den Vorbedingungen werden die Anforderungen spezifiziert, die beim Aufruf der Methode erfüllt sein müssen, damit diese korrekt ausgeführt werden kann. Diese Vorbedingungen enthalten demnach die Anforderungen an die Aufrufpara-

[7] englisch: contracts

meter der Methode. Erfüllen diese die Spezifikation, so ist die Implementierung der Methode verpflichtet, die spezifizierten Nachbedingungen, nämlich die Wirkung der Methode, zu erfüllen. Es besteht also ein Vertrag zwischen Methode und Aufrufer über die Wirkung der Methode, abhängig von der Eingabe. Für Klassen können Invarianten spezifiziert werden. Dies sind Eigenschaften der Instanzen dieser Klassen, die für die Sicht nach aussen, also vor und nach jeder Methodenausführung gelten müssen. Sie beschreiben die Konsistenz der abgeleiteten Objekte und müssen gelten, wenn das Objekt von aussen nutzbar ist. Um dies sicherzustellen müssen die Methoden neben den Vor- und Nachbedingungen zusätzlich die Invarianten des Objekts erfüllen.

Die Verträge dienen als Spezifikation der Software für die Programmierer, die anhand der spezifizierten Vor- und Nachbedingungen die Methoden implementieren. Neben der Spezifikation sollen die Verträge zudem zur Laufzeit überprüft werden, deshalb müssen die Verträge in einer Sprache spezifiziert sein, aus der Code zur Überprüfung generiert werden kann. Es ist daher nahe liegend, die Verträge in der gleichen Sprache zu spezifizieren, in der die Software implementiert wird. Meyer führte DbC bereits als einen Bestandteil der Sprache *Eiffel* ein. Zur Laufzeit können somit die spezifizierten Bedingungen ausgeführt werden und zu `true` oder `false` ausgewertet werden.

Zur Veranschaulichung der Verträge bietet sich die Funktion zur Berechnung der Quadratwurzel an, da es sich um eine partiell definierte Funktion handelt und somit Forderungen an den Eingabeparameter gestellt werden müssen. Der Eingabetyp der Funktion ist `float`, die Funktion ist für negative Zahlen nicht definiert, was in der Vorbedingung der Funktion überprüft wird. Als Nachbedingung kann die korrekte Berechnung geprüft werden, indem der Rückgabewert quadriert wird und mit dem ursprünglichen Eingabewert verglichen wird. Der Vertrag der Funktion lautet dann:

$$\begin{array}{ll} sqrt: & in : float \mapsto out : float \\ \textbf{pre} & in \geq 0 \\ \textbf{post} & out \cdot out - in \leq 0.0001 \end{array}$$

Würde die Nachbedingung überprüfen, ob $out \cdot out = in$ gilt, was der abstrakten Spezifikation entspricht, so würden Rundungsfehler der Berechnung als falsches Ergebnis gewertet. Dies zeigt, dass der theoretische Ansatz, wonach die abstrakte Spezifikation für die Laufzeitüberprüfung genutzt werden kann, zunächst scheitert. In der Tat müssen entweder bei der abstrakten Spezifikation der Verträge bereits technische Beschränkungen wie Rundungsfehler berücksichtigt oder die abstrakte Spezifikation der Verträge für den Einsatz der Laufzeitüberprüfungen noch überarbeitet werden.

Der Eingabe-Parameter *in* befindet sich nur im Vorbereich der Funktion, wird also nicht während der Berechnung verändert. Werden Eingabe-Parameter während einer Funktion geändert, so kann es für die Formulierung der Nachbedingung notwendig sein, dennoch auf den Wert des Parameters oder Objekts, der zum Zeitpunkt der Eingabe gültig war, zugreifen zu können. Wird beispielsweise in eine Listen-Datenstruktur ein Element eingefügt, so kann in einer Nachbedingung abgefragt werden, ob die Größe am Ende der Funktion inkrementiert ist: $size = @size + 1$. DbC muss diese Funktionalität realisieren; in einigen Sprachen wird der Zugriff auf die Werte eines Objekts vor Ausführung der Methode syntaktisch mittels „@“ gekennzeichnet; im Folgenden sei dies als *old-value* bezeichnet.

Werden Vor- und Nachbedingungen zur Laufzeit überprüft, so werden die Ausdrücke zu `true` oder `false` ausgewertet. Da die Auswertung der Vor- und Nachbedingung lediglich eine Überprüfung der Methodenberechnung ist, darf die Auswertung den Zustand des Systems nicht ändern, die Auswertung muss frei von Seiteneffekten sein.

Invarianten müssen für ein Objekt einer Klasse eingehalten werden, können demnach also auf Vor- und Nachbedingungen aller Methoden abgebildet werden, da sich das Objekt sowohl vor dem Aufruf als auch nach der Ausführung einer Methode in definiertem Zustand befinden soll. Unter Umständen müssen während der Ausführung einer Methode allerdings Hilfsfunktionen aufgerufen werden, solange sich ein Objekt in inkonsistentem Zustand befindet. Die Handhabung dieses Problems wird in Abschnitt 4.2.2.1 näher beschrieben.

4.2.1.1 Vorteile von Design by Contract

Der zunächst offensichtliche Vorteil von DbC ist die Fehlererkennung in Form der Überprüfung der Verträge zur Laufzeit. Neben diesem zentralen Zweck bietet das Konzept jedoch einige weitere Vorteile, besonders in der Entwicklungsphase der Software:

- Besseres Verständnis der Objekt-Orientierung;
- Methodischer Ansatz zur Fehlervermeidung;
- Spezifikation zur Unterstützung der Implementierung;
- Effektives Framework zum Debugging und Testen von Software;
- Methode zur Dokumentation von Software;

- Verbesserung der Code-Lesbarkeit und Software-Wartung;
- Mechanismus zur Ausnahmebehandlung.

Ein Teil der genannten Vorteile ist generell gültig, unabhängig davon wie die Vor- und Nachbedingungen einer Methode zu spezifizieren sind bzw. wie die Laufzeit-Überprüfungen realisiert werden. Ein Teil der Vorteile ist allerdings abhängig von der Implementierung, beispielsweise wird die Code-Lesbarkeit und damit die Wartung der Software nur dann verbessert, wenn die Spezifikation der Verträge auch tatsächlich bei dem Code, der eine Methode implementiert, zu finden sind. Im folgenden Abschnitt werden Ansätze zur Integration von DbC seit der Einführung in Eiffel beleuchtet, um einen Überblick über die Entwicklung des Konzepts, insbesondere die aus den unterschiedlichen Implementierungen hervorgehenden Vor- und Nachteile zu erhalten. Aus diesen werden Schlussfolgerungen für die Realisierung im MoDiS-Projekt gezogen und der gewählte Ansatz zur Implementierung begründet.

4.2.1.2 Realisierungs-Ansätze von Design by Contract

Seit der Einführung von DbC in *Eiffel* wurde das Konzept für verschiedenste Programmiersprachen realisiert, wie beispielsweise *C++* [PP99], *Smalltalk* [CCGMPR96] und Python [Plö97]. Für *Java* wurde DbC sogar in verschiedenen Ansätzen mit völlig unterschiedlichen Zielen und Schwerpunkten implementiert: *JMSAssert* [Man00], *iContract* [Kra98], *jContractor* [KHB99], *HandShake* [DH98], *Kopi* [LKP02] und *Jass* [BFMW04].

Die hier genannten Ansätze für die Unterstützung von DbC in Java im Detail zu vergleichen würde den Rahmen dieser Arbeit sprengen. Die prinzipiellen Ansätze der Implementierungen und deren Ziele werden erläutert, um die Motivation für den in INSEL gewählten Ansatz darzulegen. Weitere Vergleiche einiger der genannten Implementierungen sind in [Plö02], [LKP02] und [PM07] nachzulesen. Generell werden drei Ansätze zur Integration unterschieden:

1. *Built-in:* DbC wird vollständig in die Sprache integriert; der Übersetzer der Sprache muss erweitert werden, um die zusätzlichen Sprachkonstrukte verarbeiten zu können. Der Übersetzer überprüft syntaktische Korrektheit der Verträge und generiert den Code zur Laufzeit-Überprüfung der Verträge.
2. *Preprocessing:* Die Verträge werden unabhängig von der Programmiersprache als Kommentare oder in separaten Dateien spezifiziert. Ein Präprozessor transformiert die spezifizierten Verträge in Quellcode der Sprache und

integriert diesen an die entsprechenden Stellen des Quellcodes der Anwendung. Der Vorteil dieses Ansatzes besteht darin, dass Sprache und Übersetzer nicht geändert werden müssen.

3. *Library-based:* Die Laufzeitumgebung der Sprache wird erweitert, sodass zur Laufzeit der Code für die Vertragsüberprüfung auf Basis externer Spezifikationen ausgeführt werden kann. Der Vorteil dieser Vorgehensweise ist, dass so auch Verträge zu Software, die nicht im Quellcode sondern lediglich in bereits übersetztem Objektcode vorliegt, hinzugefügt werden können. Dieser Ansatz wird auch der Metaprogrammierung zugeordnet.

Die meisten Realisierungen für Java verfolgen Ansatz 2 mit Präprozessor. Der Schwerpunkt dieser Ansätze liegt auf der Kompatibilität mit existierendem Java-Compiler und Java-Virtual-Machine (JVM). *JMSAssert* nutzt dabei *JMScript*, um die als Kommentare spezifizierten Verträge mit Hilfe einer zusätzlichen, dynamisch gebundenen Bibliothek auszuwerten. *iContract* nutzt ebenfalls Kommentare zur Spezifikation der Verträge, übersetzt diese aber zu Java-Code, der auf der unveränderten JVM ausführbar ist. Die Implementierung von *Jass* ist dazu vergleichbar, unterscheidet sich von *iContract* lediglich in Syntax und einigen Erweiterungen, die die Spezifikation der Dynamik in Objekten ermöglichen sollen.

JContractor und *HandShake* basieren auf Ansatz 3. Mittels festgelegter Namens-Schemata werden in *JContractor* die Verträge als normale Java-Methoden programmiert, durch Pattern-Matching zur Ladezeit vom geänderten Class-Loader erkannt und entsprechend zum Code gebunden. Abgesehen vom Class-Loader kann eine unveränderte Java-Umgebung genutzt werden. Bei *Handshake* werden die Verträge in extra Dateien spezifiziert, das Vorhandensein des Quellcodes der Anwendung ist nicht notwendig. Zur Ladezeit verändert eine dynamisch hinzugebundene Bibliothek die Objektdateien der Anwendung und integriert den Code zur Vertragsprüfung.

Lediglich die Implementierung von Lackner im Projekt *Kopi* integriert DbC vollständig in die Sprache nach Ansatz 1, wie dies auch in Eiffel realisiert wurde. Für die Spezifikation der Verträge wurde die Sprache um neue Schlüsselwörter erweitert. Der auf einem Standard-Java-Übersetzer basierende *Kopi* kann diese in normalen Java-Byte-Code übersetzen, wobei die Verträge soweit möglich über die in Java integrierten Assertions realisiert werden und somit hohe Performanz erreicht wird.

4.2.1.3 Bewertung der Ansätze

Die Vorteile der Implementierungen nach Ansatz 2 oder 3 liegen bei der Kompatibilität mit existierenden Werkzeugen, wie Java-Übersetzer und JVM. Um DbC in bestehende Projekte zu integrieren, bieten sich diese Ansätze an. Auf lange Sicht gesehen ist entscheidend, welcher Ansatz die meisten Vorteile von DbC unterstützt und mehr dazu beiträgt, zuverlässige Software zu entwickeln. Die Überprüfung der Verträge zur Laufzeit lassen sich mit jedem der Ansätze durchführen, der Kern von DbC ist somit erfüllt. Unterschiedlich sind jedoch die Möglichkeiten, die Information der abstrakten Spezifikationen über dies hinaus zu nutzen. Die Vorteile einer abstrakten Spezifikation, die einerseits zur Implementierung dient, ferner als stets aktuelle Dokumentation genutzt wird und zudem die Code-Lesbarkeit erhöht, ist bei Ansatz 3 nur sehr begrenzt gegeben.

Ansatz 2 erfüllt diese Anforderungen zwar, allerdings sind die Informationen über die Implementierung (Sicht 1) und die Informationen über die Verträge (Sicht 2) nicht zusammen nutzbar. Der Präprozessor übersetzt die Verträge zu Java-Code, der Java-Übersetzer kann demnach nicht mehr zwischen eigentlicher Anwendung und Verträgen unterscheiden. Wird DbC vollständig in die Sprache integriert, wie im *Kopi*-Projekt, so kann der Übersetzer die Informationen der Anwendung und der Verträge nebeneinander nutzen. Das Potential dieser Informationen wird an einigen Beispielen verdeutlicht:

- Die Syntaktische und Semantische Analyse des Übersetzers kann auch für die Verträge verwendet werden. Zusätzlich kann überprüft werden, ob durch die Auswertung einer Vertragsbedingung eine Endlos-Rekursion entsteht. Dies wäre der Fall, wenn in der Bedingung eine Funktion aufgerufen wird, die (indirekt) wieder zu der gleichen Bedingung führt.

- Die *old-values* können effizient realisiert werden, da der Übersetzer mittels Traversieren des abstrakten Syntax-Baums die Sichtbarkeit der Objekte ermitteln kann. Die Kopien der Objekte, in denen der Zustand des Aufrufs gespeichert wird, können im Übersetzer analog zu den originalen Anwendungsobjekten alloziert und übersetzt werden, sodass für die Laufzeitumgebung keine Änderungen entstehen.

- Sind sowohl Informationen aus Sicht 1, als auch aus Sicht 2 vorhanden, so kann der Übersetzer analysieren, ob die Auswertungen der Vertragsbedingungen frei von Seiteneffekten sind. Falls dies nicht der Fall ist, kann eine Fehlermeldung ausgegeben werden.

- Es werden keine Debug-Informationen, wie beispielsweise Zeilennummern verfälscht, wie dies bei einem Präprozessor-Ansatz der Fall ist.
- Die beiden Sichten auf die Anwendung können für weitere Analysen genutzt werden. Die Kombination dieser Informationen erschließt neue Möglichkeiten in den Bereichen Verifikation, Fehlertoleranz und adaptives Management; näheres dazu wird in Abschnitt 4.2.4 erläutert.

Es zeigt sich, dass auf lange Sicht eine vollständige Integration in eine Sprache die meisten Vorteile bietet. Die Ansätze 2 und 3 bieten sich lediglich an, falls eine Änderung des Übersetzers nicht möglich ist. Für das MoDiS-Projekt stellt dies keine Hürde dar, deshalb wurde DbC nach Ansatz 1 vollständig in die Sprache INSEL und den Übersetzer gic integriert.

4.2.2 Verträge in INSEL

Die Sprache INSEL wurde erweitert, um die Verträge nach dem Konzept von DbC in vollem Umfang spezifizieren zu können. In diesem Unterkapitel wird die Spracherweiterung und Implementierung der Integration vorgestellt (vgl. [May06] und [PM07]). Es wurde darauf geachtet, dass die Integration keine Nebeneffekte für die Programmiersprache mit sich bringt. Der Kern der Erweiterung ist die Definition der Vertragsbedingungen:

```
contract begin
  pre Ausdruck ;
  post Ausdruck ;
  inv Ausdruck ;
end;
```

Einem Objekt können beliebig viele Vertragsbedingungen zugeordnet werden, es wird jeweils die Konjunktion der Vorbedingungen, Nachbedingungen bzw. Invarianten ausgewertet:

$$pre = \bigwedge_i pre_i$$

$$post = \bigwedge_i post_i$$

$$inv = \bigwedge_i inv_i$$

Die Bedingungen sind beliebige INSEL-Ausdrücke, die Boole'sche Werte ergeben. Wie eingangs erwähnt darf der Ausdruck einer Bedingung keine Seiteneffekte enthalten und zudem nicht zu einer Endlosrekursion führen. Dies muss generell vom Programmierer beachtet werden; wird jedoch zudem vom Übersetzer erkannt. Die Vertragsbedingungen werden in der Definition des Generators des Objekts beschrieben. Im Gegensatz zu gängigen objektorientierten Sprachen ist INSEL eine Objektbasierte Sprache, in der einerseits keine Vererbung vorgesehen ist, was zu einer Vereinfachung für die Umsetzung von DbC führt. Andererseits gibt es in INSEL nicht einheitliche Klassen, sondern Generatoren verschiedener Komponentenarten, für die einzeln untersucht werden musste, wie sich das Konzept der Verträge integrieren lässt.

4.2.2.1 Depots

Die direkte Analogie zu einem Objekt, dessen Zustand nur über definierte Operationen verändert werden kann, bildet in INSEL die Komponentenart Depot (vgl. Abschnitt 2.2.1.3). Bei der Erzeugung eines Depots wird implizit die äussere Operation `initialisiere` aufgerufen, die die kanonische Operation, bestehend aus Deklarations- und Anweisungsteil, abarbeitet. Dies dient der Initialisierung des Depots, das erst nach der Ausführung der kanonischen Operation genutzt werden kann. Die Abarbeitung der kanonischen Operation ist demnach mit dem Konstruktor einer Klasse vergleichbar, wobei ein Depot in INSEL nur genau eine kanonische Operation hat. Das Depot kann genutzt werden, indem im Deklarationsteil erzeugte benannte Komponenten als *exportiert* markiert werden. Hierzu eignen sich FS-Order, PS-Order oder M-Akteure, diese drei Komponentenarten werden im Folgenden zur Vereinfachung unter dem Begriff *Routine* zusammengefasst. Hat die Zugriffsroutine nur einen Rückgabewert, so kann dies über eine FS-Order umgesetzt werden. Ist ein Tupel das Ergebnis der Berechnung, so muss eine PS-Order oder ein M-Akteur verwendet werden, die Wahl zwischen diesen beiden Komponentenarten ist Implementierungsentscheidung, abhängig von gewünschter Nebenläufigkeit.

Üblicherweise beziehen sich die Vorbedingungen auf die Parameter der exportierten Zugriffsfunktionen. In INSEL hängen die tatsächlichen Parameter, also die Werte, die eine exportierte Routine als Eingabe verwendet, allerdings auch von der Schachtelungsstruktur ab und werden von der Bindungsanalyse ermittelt. Vorbedingungen können also sowohl bezüglich der Parameter der Routine, als auch bzgl. anderer Objekte im Sichtbarkeitsbereich der Komponente formuliert

werden. Das Beispiel der Quadratwurzelberechnung würde in INSEL folgendermaßen umgesetzt:

```
function sqrt(x: in real) return real is
  contract begin
    pre  x >= 0;
    post result*result - x <= 0.001;
  end;
begin
  ...
end;
```

Jede FS-Order enthält zur Spezifikation der Nachbedingungen den Bezeichner `result`, über den auf das Ergebnis zugegriffen werden kann. Am Beispiel einer Implementierung eines Keller-Datentyps lässt sich die Erweiterung der *old-values* erläutern. Die Prozedur `push` kann nur ausgeführt werden, wenn die (implementierungsbedingte) Kellergröße nicht überschritten wird:

```
procedure push(x: in Element) is
  contract begin
    pre  anzahl_elemente < max_anzahl;
    post anzahl_elemente = @anzahl_elemente + 1;
  end;
begin
  ...
end;
```

In der Vorbedingung wird überprüft, ob die Variable `anzahl_elemente` bereits der maximalen Kellergröße `max_anzahl` entspricht. Als Nachbedingung wird erwartet, dass der Keller um ein Element größer geworden ist. Durch das @-Zeichen im Ausdruck `@anzahl_elemente` kann in der Nachbedingung zum Zeitpunkt nach der Ausführung der Prozedur noch auf den Wert der Variable `anzahl_elemente` vor Prozedurausführung zugegriffen werden. Dies wird über lokale Kopien von Objekten bzw. Komponenten im Übersetzer realisiert.

Sollen Bedingungen die Konsistenz eines Objekts sicherstellen und demnach *vor* und *nach* jeder exportierten Routine gelten, so sind diese als Invarianten zu spezifizieren. Am Beispiel des Kellers könnte eine Invariante spezifizieren, dass `anzahl_elemente` nicht negativ sein darf:

```
depot Stack is
  contract begin
    inv anzahl_elemente >= 0;
  end;
begin
  ...
end;
```

Nach der Ausführung einer Routine werden Invarianten überprüft, um sicherzustellen, dass die Routine einen konsistenten Zustand hinterlassen hat. Die Überprüfung der Invarianten vor der Ausführung der Routine begründet sich durch zwei Fälle: Erstens erlaubt INSEL nebenläufigen Zugriff auf Depots, der durch das Synchronisations-Konstrukt `sync` geregelt sein muss. Wurde ein Fehler bei der Synchronisation gemacht, so stellt die Konsistenzüberprüfung in der Vorbedingung sicher, dass keine weitere Routine auf das Depot zugreift, während es durch die Ausführung einer anderen Routine in inkonsistentem Zustand ist. Der zweite Grund für die Überprüfung ist in einem grundsätzlichen Problem begründet, das in der objektorientierten Programmierung auftritt (vgl. [BS02]). Zur Erläuterung muss zunächst die Fernwirkung einer Routine definiert werden:

Definition 4.10 (Fernwirkung)
Seien x und y Instanzen der Depot-Generatoren X und Y, a sei eine Routine von X und b eine Routine von Y. Dann impliziert $x.a \rightsquigarrow y.b$, dass die Ausführung von $x.a$ den Aufruf von $y.b$ verursachen kann.

Sind $x.a$ und $y.b$ exportierte Routinen, so erwarten beide Routinen einen konsistenten Zustand des Objekts und müssen einen konsistenten Zustand hinterlassen. Dies ist auch der Fall, wenn $x.a \rightsquigarrow y.b$ gilt. Es entsteht allerdings ein Problem, falls $x = y$ gilt, also die Ausführung der exportierten Routine $x.a$ den Aufruf der Routine $x.b$ des selben Depots verursacht:

$$x.a \rightsquigarrow x.b$$

Würde die Invariante nicht vor jedem Aufruf einer exportierten Routine geprüft, so könnte $x.b$ aufgerufen werden, während Depot x sich in einem inkonsistenten Zustand befindet; somit wäre die korrekte Ausführung der Routine nicht mehr sichergestellt. Auf der anderen Seite kann es für sauberes Design des Depots durchaus sinnvoll sein, eine Routine zur Verfügung zu stellen, die auch bei inkonsistentem Zustand des Depots aufgerufen werden darf, um Teilberechnungen der Routine $x.a$ auszuführen. Für die Integration von Design by Contract wurde diese Unterscheidung anhand der *Export*-Markierung getroffen:

- Für jede exportierte Routine werden Invarianten vor und nach Ausführung geprüft. Exportierte Routinen können demnach einen konsistenten Zustand voraussetzen.
- Lokale Routinen und Operationen überprüfen keine Invarianten. Die Routinen müssen für jeden Depotzustand ausführbar sein.

Diese Festlegungen schränken die Freiheiten der Programmierung ein, da auch ein konsistenter Zustand vorliegen muss, falls eine private Routine eine exportierte Routine aufruft. Diese Einschränkungen sind nach Erfahrungswerten mehr Vor- als Nachteil, da ein klareres objektorientiertes Design entsteht und durch die geforderte Disziplin des Programmierers Fehler vermieden werden.

4.2.2.2 Routinen

Depots sind das direkte Pendant zu Objekten in objektorientierten Sprachen, da sie Daten kapseln und über definierte Zugriffsroutinen eine Zustandsänderung ermöglichen. Das Konzept DbC lässt sich deshalb sehr direkt auf Depots übertragen. Die INSEL-Komponenten FS-Order, PS-Order und M-Akteur, die unter dem Begriff Routinen zusammengefasst wurden, können nicht nur in Depots geschachtelt, sondern beliebig in INSEL-Programme eingeordnet werden. Für diese Fälle wurde untersucht, inwiefern sich die Methodik von DbC auch für diese Komponenten anwenden lässt. Die Komponenten können lokale Daten enthalten, sowie auf Daten in ihrem Sichtbarkeitsbereich zugreifen. Routinen haben allerdings nur eine Zugriffsfunktion, für Ordern `führe_aus` und für M-Akteure `starte`, die jeweils implizit vom System ausgeführt wird (vgl. Abschnitt 2.2.1). Diese Zugriffsfunktion wird nur ein einziges Mal ausgeführt, abgebildet auf ein Objekt würde dies sozusagen nur für diese Zeit existieren und nur einen Zugriff erleben. Invarianten ergeben aus diesem Grund keinen Sinn für Routinen, Vor- und Nachbedingungen allerdings durchaus, da die Ausführung der einen kanonischen Operation analog zu jeder Funktion gewisse Vorbedingungen haben und natürlich die Korrektheit des Ergebnisses gewisse Nachbedingungen erfüllen kann. Für Routinen, die nicht innerhalb eines Depots deklariert sind, sind deshalb Vor- und Nachbedingungen, allerdings keine Invarianten, realisiert.

4.2.2.3 BS-Order

Die Deklaration von BS-Ordern steht direkt im Anweisungsteil an den Stellen, an denen sie erzeugt werden sollen. BS-Ordern dienen in erster Linie der Syn-

chronisation, da eine BS-Order erst aufgelöst wird, wenn die Abschlusssynchronisation aller parallel ausgeführten, in die BS-Order eingeordneten Komponenten abgeschlossen sind. Beispielsweise kann mit Hilfe einer BS-Order sichergestellt werden, dass erzeugte M-Akteure terminiert sind, bevor mit den Berechnungen fortgefahren wird. BS-Ordern können analog zu den unter dem Begriff Routinen zusammengefassten Komponenten mit Vor- und Nachbedingungen versehen werden. Beispielsweise könnte das Verhältnis verschiedener, innerhalb der BS-Order durch M-Akteure parallel berechneter Ergebnisse überprüft werden. In der Praxis ist allerdings die BS-Order in erster Linie eine strukturierende Komponente, sodass selten eine Notwendigkeit für Vor- und Nachbedingungen besteht.

4.2.2.4 K-Akteure und K-Order

Zunächst ist ein K-Akteur eine aktive Komponente mit kanonischer Operation und deshalb mit einem M-Akteur vergleichbar. Das zu M-Akteuren Beschriebene gilt demnach auch für K-Akteure und deren Vor- und Nachbedingungen, mit der einzigen Ausnahme, dass K-Akteure keine Rückgabewerte haben und sich deshalb die Nachbedingungen nur auf Objekte des Sichtbarkeitsbereichs des K-Akteurs beziehen. Zusätzlich koordinieren sich K-Akteure über die K-Ordern, deren Annahme durch das `select`-statement eingeschränkt sein kann. K-Ordern sind in Vor- und Nachbedingung mit PS-Ordern vergleichbar, allerdings werden sie in die Ausführung des K-Akteurs eingeordnet, der dies explizit annehmen muss. Die Bedingung für diese Annahme, die über das `select`-statement eingeschränkt werden kann, kann in der Vorbedingung einer K-Order geprüft werden.

4.2.2.5 Auswertung der Verträge

Vorbedingungen, Nachbedingungen und Invarianten beziehen sich auf Objekte im Sichtbarkeitsbereich der Komponente, wobei diese weiter zu unterscheiden sind in Parameter, lokale Objekte, nichtlokale Objekte und Rückgabewerte. Der Zeitpunkt der Auswertung der Vertragsbedingungen muss so gewählt werden, dass die verschiedenen Bedingungen trotz möglicher Verschattung auf die richtigen Objekte bezogen werden. Dazu wurde die kanonische Operation (vgl. Abschnitt 2.2.1) erweitert, wie in Abbildung 4.7 dargestellt.

Die Vorbedingungen werden in der V-Phase zu Beginn der kanonischen Operation ausgewertet. Die Parameter, auf die in den Vorbedingungen zugegriffen wird, wurden bereits während der Erzeugung der Komponente erstellt. Die kanonische Operation wird nur weiter ausgeführt, falls die Konjunktion der Vorbedingun-

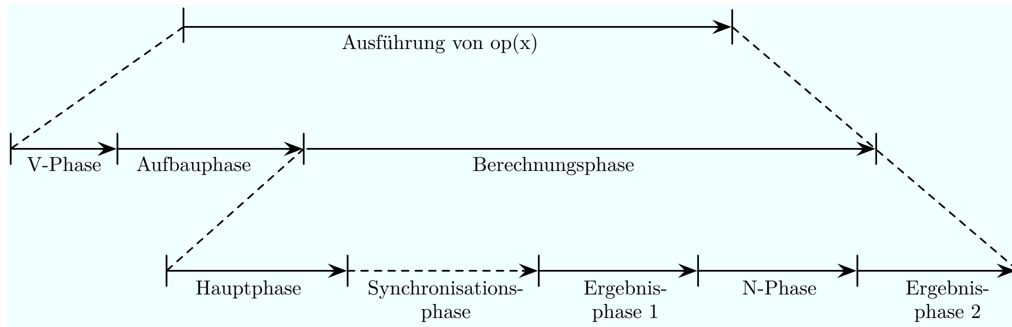


Abbildung 4.7.: Phasen der erweiterten kanonischen Operation

gen (und ggf. Invarianten) zu `true` ausgewertet werden. Wird `false` ermittelt, so wird die kanonische Operation nicht ausgeführt, das Objekt bleibt somit in konsistentem Zustand.

Nach Abarbeitung der nach INSEL-Spezifikation beschriebenen Phasen (vgl. [SEL+96]) wird die Ergebnisphase in zwei Teile aufgeteilt. Dazwischen findet die N-Phase statt, in der die Konjunktion aus Nachbedingungen und Invarianten (falls vorhanden) ausgewertet werden. Die Aufteilung der Ergebnisphase von FS-Ordern auf zwei Phasen dient dazu, dass bereits in Ergebnisphase 1 der Ergebnisausdruck von FS-Ordern erarbeitet werden muss, jedoch vorerst nur der zusätzlichen Variable `result` zugewiesen wird. Dieser Wert kann in die Auswertung der Nachbedingungen eingehen. Bei einer Auswertung zu `true` wird schließlich in Ergebnisphase 2 die `return`-Anweisung ausgeführt und der Ergebniswert der FS-Order dem Aufrufer übergeben.

4.2.2.6 Implementierung

Die beschriebenen Vertragskonzepte wurden in den INSEL-Übersetzer *gic* integriert. Die Erweiterungen der Grammatik sind in Anhang D zu finden. Der Übersetzer generiert Objekt-Code zur Auswertung der Verträge. Der Zugriff auf *old-values* wird realisiert, indem von der entsprechenden Komponente eine Kopie erzeugt wird, deren Bezeichner mit dem Kürzel `_pre_` beginnt. Da Anwendungskomponenten in INSEL nicht mit einem Unterstrich beginnen dürfen, kann dies nicht zu Namenskonflikten führen.

Dies gilt ebenso für zusätzlich angelegte Variablen für den Ergebniswert von FS-Ordern, die intern als `.result` angelegt werden, da Anwendungsvariablen

nicht mit einem Punkt beginnen dürfen. Einige Details zur Implementierung sind in der Diplomarbeit von A. Mayer näher beschrieben (vgl. [May06]).

4.2.3 Fehlererkennung auf Basis der Verträge

DbC stellt eine Methodik bereit, die das Ziel verfolgt, zuverlässige Software zu entwickeln. Neben der Vermeidung von Fehlern durch die bereits genannten Vorteile im Softwareentwicklungsprozess können zur Laufzeit Fehler erkannt werden. Die Übersetzer-Komponente des Managements generiert Code zur Überprüfung der Verträge. Wird ein Vertrag verletzt, so kann das Laufzeit-Management darauf reagieren. Somit kann DbC als Basis für Fehlertoleranzverfahren genutzt werden, indem nach einem erkannten Fehler eine Ausnahmebehandlung eingeleitet wird. In dieser Ausnahmebehandlung kann beliebig durch das Management auf den Fehler reagiert werden. In der vorliegenden Arbeit sollen erkannte Fehler durch Rückwärtsbehebung toleriert werden. Die Mechanismen hierfür werden in den Kapiteln 5 und 6 erläutert. Folgend werden Fehlerarten identifiziert, die mit Hilfe der Laufzeitüberprüfungen der Verträge erkannt werden können.

Grundsätzlich kann bei der Überprüfung der spezifizierten Verträge jeder Fehlerzustand erkannt werden, der als Bedingung formulierbar ist, deren Auswertung im Fehlerfall `false` und im fehlerfreien Betrieb `true` ergibt. Auf Grund des Aufwands der Spezifikation der Verträge und des Performanzverlusts durch deren Überprüfung werden in der Praxis nur nahe liegende Problemfälle spezifiziert und überprüft. Generell sind dies Vergleiche von Werten der Datenobjekte. Die Überprüfung der Nachbedingungen entspricht dabei klassischen *Semantik-Tests* (vgl. Abschnitt 3.2) Je genauer der *fehlerfreie* Wertebereich eines Datenobjekts bei Eingabe oder Ergebnis einer Routine festgelegt werden kann, umso besser die Fehlererkennung. Kann beispielsweise nur überprüft werden, dass das Ergebnis einer Routine mit dem Datentyp `integer` nicht 0 sein darf, so können dennoch viele logische Fehler in der Routine enthalten sein, die nicht erkannt werden. Kann dagegen, wie im Beispiel der Quadratwurzelfunktion, eine Kontrollrechnung gemacht werden, die jeden Wert eindeutig als korrekt oder fehlerhaft erkennt, so können logische Fehler in der Regel erkannt werden.

Die Überprüfung der Zustände von Datenobjekten kann allerdings neben logischen Berechnungsfehlern ebenso Fehler erkennen, die nur als Symptom im falschen Wert enden. Beispielsweise kann mit einer gewissen Wahrscheinlichkeit fehlerhafte Synchronisation erkannt werden, da die Konsistenz eines Objekts zu Beginn einer exportierten Routine gewährleistet sein muss, und diese verletzt

ist, falls das (exklusiv zu benutzende) Objekt gerade von einer anderen Routine geändert wird.

Ebenso können Fehler im Speichermanagement, wie beispielsweise Pufferüberlauf, zum Überschreiben von Objekten führen. Dieser Fehler kann mittels Verträgen an zwei Stellen erkannt werden, erstens in der Routine, die den Inhalt des Puffers füllt und über die Grenzen hinaus schreibt, da die Puffergröße und zu schreibender Inhalt verglichen werden können. Zweitens kann eine Routine, die auf dem Objekt arbeitet, das im Speicher hinter dem Puffer liegt und überschrieben wurde, den inkonsistenten Zustand des Objekts durch fehlerhafte Inhalte feststellen.

Werden falsche Typen an Funktionen übergeben, so kann dies bereits der INSEL-Compiler gic feststellen, da eine Typprüfung integriert ist. In anderen Sprachen könnte die Typprüfung in den Verträgen spezifiziert sein und somit Fehler der Typisierung erkannt werden.

Die Reduktions- und Abbruchbedingungen für rekursiv aufgerufene Funktionen können ebenfalls in Nachbedingungen überprüft werden, sodass Fehler die zu Endlosrekursion führen, erkannt werden können. Wird Beispielsweise eine Operation iterativ auf jedem Eintrag einer Liste ausgeführt, so könnte überprüft werden, dass der Index der Abbruchbedingung sich dieser annähert.

DbC ermöglicht das Erkennen von byzantinischen Fehlern, da auch semantisch falsche Ergebnisse erkannt werden können. Auf der anderen Seite werden offensichtlich nicht alle byzantinischen Fehler erkannt. Die tatsächlich erkennbare Fehlerklasse ist abhängig von den zu überprüfenden Funktionen. So ist beispielsweise eine Probe für eine Quadratwurzelberechnung sehr einfach zu berechnen, eine Probe für eine vollständige Matrixmultiplikation nicht.

Andererseits hängt die tatsächlich erkennbare Fehlerklasse ab von dem Aufwand, der bei der Spezifikation der Verträge geleistet wird, sowie von dem Aufwand, der für deren Berechnung investiert werden soll. Eine präzise Spezifikation der Vorbedingungen, Nachbedingungen und Invarianten ist wünschenswert, um die Wahrscheinlichkeit zur Fehlererkennung zu erhöhen. Dies ist allerdings aufwändig und senkt die Performanz der Ausführung, da zur Laufzeit ein beträchtlicher Anteil der Rechenzeit zur Auswertung der Vertragsbedingungen verbraucht wird.

Die Vertragsprüfungen könnten nur zu Debug-Zwecken zugeschaltet und im normalen Betrieb des Systems darauf verzichtet werden. Im Rahmen dieser Arbeit wird die Fehlererkennung mittels DbC als Basis für ein Fehlertoleranzverfahren

eingesetzt, deshalb müssen die Vertragsauswertungen immer stattfinden. Es ist Aufgabe des Programmierers, einen geeigneten Kompromiss zwischen Aufwand und Nutzen bei der Spezifikation der Verträge zu finden.

4.2.4 Bewertung

DbC erleichtert die Entwicklung hochqualitativer Software. Neben der Fehlererkennung durch die Laufzeitüberprüfung ist die auferlegte Methodik hilfreich für ein klares Design im objektorientierten Sinn. Die Spezifikation von Verträgen fördert automatisch die Reflexion des Programmierers über das Design und die Abhängigkeiten der Module der zu erstellenden Software. So genannte *Workarounds* oder *Dirty Hacks*, also Implementierungen mit unsauberem Design oder unklarer Strukturierung werden so frühzeitig vermieden. Für die vorliegende Arbeit ist DbC in erster Linie als Mechanismus zur Fehlererkennung genutzt, um ein Fehlertoleranzverfahren darauf aufzubauen. Die Überprüfung der Verträge zur Laufzeit verschlechtert die Performanz des Systems, rechtfertigt sich aber durch die Fehlererkennung. DbC bietet allerdings weit mehr Potential zur Systemkonstruktion als ausschließlich die Fehlererkennung. Folgend werden hierzu noch einige Einsatzmöglichkeiten angeschnitten, die nicht Teil der vorliegenden Arbeit sind.

Insbesondere bei einer vollen Integration des Konzepts in die Sprache, also des in dieser Arbeit verfolgten Ansatzes, eröffnen sich eine Reihe von Möglichkeiten, die beiden Sichten auf die Anwendung gemeinsam im Management zu nutzen. Die Verträge bieten eine abstrakte Sicht auf die Zustandsänderungen, die eine Funktion bewirkt. Die Implementierung entspricht der konkreten Sicht auf diese Zustandsänderungen und muss in ihrer Ausprägung der abstrakten Sicht entsprechen. Es sei hier betont, dass die Verträge kein Teil der Implementierung sind bzw. sein dürfen, sondern eine Spezifikation der wesentlichen Eigenschaften, die von Details abstrahiert. Die Vorteile von DbC in der Softwareentwicklung, speziell die bessere Lesbarkeit des Codes basiert auf dem Vorhandensein dieser beiden Sichten unterschiedlicher Abstraktion auf die gleiche Funktionalität. Eine Reihe neuer Forschungsthemen ergibt sich, sobald diese beiden Sichten nebeneinander automatisiert verarbeitet werden können. Ist DbC voll in die Sprache und damit in den Übersetzer integriert, so kann die Analyse im Übersetzer die Informationen der Implementierung und der Verträge *zusammen* nutzen.

Eine Anwendungsmöglichkeit, diese Informationen zu nutzen, besteht in der Automatisierung von Test und Verifikation. Die Verträge geben die Rahmenbedingungen für das Verhalten der Implementierung vor, sodass die Einhaltung

dieser Rahmenbedingungen sicherlich zum Teil verifizierbar, zum Teil durch automatisierte Tests überprüfbar ist.

Neben dem Ziel der Fehlererkennung könnte die Information genutzt werden, um Fehlertoleranzverfahren direkt zu unterstützen. Die Überprüfung der Verträge lässt Fehler erkennen und lokalisieren, die Information könnte aber zudem genutzt werden, um mittels Vorwärtsbehebung die aufgetretenen Fehler zu tolerieren. Es wäre denkbar, auf Basis des Bestandteils der konjugierten Bedingung, der zu `false` ausgewertet wurde, den Zustand eines Datenobjekts gemäß der Vertragsbedingung zu ändern, um den aufgetretenen Fehler zu tolerieren.

Ferner kann die abstrakte Sicht auf die Funktionalität, im speziellen die Invarianten von Depots, als zusätzliche Informationen über die Anwendung im Management des Systems genutzt werden. Es wäre beispielsweise denkbar, dass die Vorbedingungen eines entfernten Aufrufs einer Methode bereits auf der aufrufenden Stelle geprüft werden, um die Nachrichtenlast des Netzwerks im Fall einer Verletzung der Bedingung gering zu halten.

Das bestehende Konzept von DbC erlaubt nur relativ eingeschränkte Spezifikation der Vertrags-Bedingungen, da diese von der objektorientierten Sicht bzw. ursprünglich von der Spezifikation abstrakter Datentypen abgeleitet ist. Für den Einsatz in Komponenten-basierten, verteilten Systemen wäre eine Ausdrucksstärkere Spezifikation denkbar. Beispielsweise könnten temporale Quantoren integriert werden, um Eigenschaften der Dynamik eines Systems besser zu reflektieren.

4.3 Zusammenfassung

In diesem Kapitel wurden zwei in das MoDiS-Projekt integrierte Verfahren zur Fehlererkennung erläutert. Die Erfassung konsistenter Sichten verteilter Systeme auf Basis kausaler Abhängigkeiten wurde an das Systemmodell des MoDiS-Projekts angepasst. Die *Happened-Before*-Relation nach Lamport wurde um das dynamische Prozessmodell sowie um die Existenz des gemeinsamen, verteilten Speichers mit sequentieller Konsistenz erweitert, sodass die kausalen Abhängigkeiten des vorliegenden Systemmodells vollständig erfasst werden.

Es wurde erläutert, welche Informationen zur Laufzeit vom Management protokolliert werden müssen, um effizient eine konsistente, partielle Ereignisordnung zu konstruieren. Die Realisierungsmöglichkeiten zur Protokollierung der Speicherzugriffs-Ereignisse wurden dargelegt und erklärt, dass nur im Übersetzer

diese Protokollierung geeignet vorbereitet werden kann. Ferner wurde erläutert, wie Timing-Fehler auf Basis der konstruierten Sicht durch einen Vergleich mit den Ereignisspuren der Spezifikation erkannt werden können. Die konstruierten Ereignisordnungen könnten darüber hinaus zu Analysen der Systeme in Form von Prädikatsbeweisen genutzt werden.

Im zweiten Teil des Kapitels wurde die Integration des Vertragskonzepts in die Sprache INSEL beschrieben. Es wurde dargelegt, wie die Verträge zum Softwareentwicklungsprozess bereits in frühem Stadium beitragen können und letztendlich zur Laufzeit die Verträge überprüft werden, um Fehler der Implementierung zu erkennen. Die verschiedenen Möglichkeiten der Integration des Konzepts in eine Sprache wurden diskutiert und der Schluss gefolgert, dass eine vollständige Sprach- und Übersetzer-Erweiterung die meisten Vorteile bietet.

Die Übertragung der für Objekte entwickelten Verträge auf INSEL-Komponenten wurde erklärt. Auch für Akteure und Ordern mit nur einer einzigen äusseren Operation ist die Vertragsintegration sinnvoll und kann zur Fehlererkennung genutzt werden. Die Implementierung der Vertragsprüfungen im Übersetzer *gic* wurden beschrieben. Die Überprüfung der Verträge zur Laufzeit ermöglicht das Erkennen verschiedener Fehlerzustände, die wie beschrieben sowohl Soft- als auch Hardware-Fehlerursachen haben können.

Die in diesem Kapitel erarbeiteten Verfahren ermöglichen eine Fehlererkennung im Management zur Laufzeit, indem einerseits Informationen in Form der ANV vom Übersetzer vorbereitet werden und andererseits der Code zur Vertragsprüfung vom Übersetzter generiert wird. In den folgenden Kapiteln wird erklärt, wie das Management geeignet auf die erkannten Fehler reagieren kann, um sie zu tolerieren.

Kapitel 5

Speicherung konsistenter Schnitte

Inhalt

5.1. Checkpoint-Verfahren 97
5.2. Kommunikationsinduziertes Checkpointing in MoDiS 110
5.3. Bewertung 139
5.4. Verwandte Arbeiten 144
5.5. Zusammenfassung 145

In diesem Kapitel wird ein Verfahren erarbeitet, welches das Speichern konsistenter Schnitte sowie das Zurücksetzen des Systemzustands zu diesen Schnitten in MoDiS realisiert. Es wird das Ziel verfolgt, dass das Management automatisiert und effizient konsistente Schnitte der Anwendung speichern und im Fall eines mit den bisher beschriebenen Verfahren erkannten Fehlers zu einem Systemschnitt zurücksetzen kann. Zunächst werden gängige Checkpoint-Rollback-Verfahren verglichen, um den Ansatz des erarbeiteten Verfahrens zu motivieren. Anschließend wird das im Rahmen dieser Arbeit entwickelte Verfahren im Detail erläutert und die Integration in das Management beschrieben und bewertet.

5.1 Checkpoint-Verfahren

Zur Toleranz von Softwarefehlern in verteilten, nebenläufigen Systemen bieten sich Verfahren an, die auf Rückwärtsbehebung basieren, da bei diesen Verfah-

ren keine Informationen zur Überführung von fehlerhaften in korrekte Zustände nötig sind (vgl. Abschnitt 3.3). Im Fall eines erkannten Fehlerzustands wird ein in der Vergangenheit der Systemausführung gespeicherter Zustand wieder hergestellt und die Ausführung von diesem Zustand aus fortgesetzt. Der Zustand muss konsistent sein, damit die Berechnung zu diesem Zustand zurückgesetzt werden kann, ohne die Funktionalität der Anwendung zu ändern. Ein solcher globaler, konsistenter Zustand ist in verteilten Systemen ein *konsistenter Schnitt* (vgl. Definition 4.4).

Eine Methode zur Rückwärtsbehebung sind Checkpoint-Rollback-Verfahren[1] (oder kurz Checkpoint-Verfahren): ein konsistenter Schnitt, auch als *Recovery-Line* bezeichnet, wird aus einzelnen gespeicherten Prozess-Zuständen, Checkpoints genannt, gebildet. Ein Checkpoint-Rollback-Verfahren setzt sich also aus zwei voneinander abhängigen Teilen zusammen:

- Im fehlerfreien Betrieb des Systems müssen Checkpoints der Prozesse gespeichert werden.
- Im Fall eines Fehlers müssen eine Recovery-Line ermittelt und die Zustände der beteiligten Prozesse durch die in der Recovery-Line enthaltenen Checkpoints ersetzt werden.

Es existieren verschiedene Verfahren, nach welchen Regeln die Checkpoints der Prozesse gespeichert werden und wie die Recovery-Line daraus gebildet wird. Die drei grundsätzlichen Ansätze sind *unkoordiniertes Checkpointing*, *koordiniertes Checkpointing* und *kommunikationsinduziertes Checkpointing*. Diese werden in den Abschnitten 5.1.1, 5.1.2 und 5.1.3 näher beschrieben.

Neben reinen Checkpoint-Rollback-Verfahren existieren zur Rückwärtsbehebung auch die aufwändigeren *Log-basierenden Rollback-Verfahren*, die neben Checkpoints alle nicht-deterministischen Ereignisse protokollieren. Dieser Ansatz basiert auf der *piecewise deterministic*-Annahme[2]: in diesen Systemmodellen muss gelten, dass alle nicht-deterministischen Ereignisse eines Prozesses identifiziert und die Informationen zu deren erneuten Ausführung (bzw. Erzeugung) in einem Log gespeichert werden können. Die Berechnungen zwischen den nicht-deterministischen Ereignissen sind deterministisch und nur von den nicht-deterministischen Ereignissen abhängig, sodass mit Hilfe eines Checkpoints (oder Startzustandes) durch erneutes Ausführen (bzw. Erzeugen) der nicht-deterministischen Ereignisse jeder Systemzustand wiederhergestellt werden kann. Log-

[1] Checkpoint-Rollback-Verfahren werden auch als Checkpoint-Recovery-Verfahren bezeichnet.
[2] wörtlich übersetzt: Stückweise deterministisch

basierende Rollback-Verfahren bieten sich für Systeme an, die viel Kommunikation mit der Systemumgebung aufweisen und zudem der piecewise deterministic-Annahme genügen (vgl. [Alv96], [AM98]). Für das Systemmodell des MoDiS-Projekts (vgl. Kapitel 2) ist allerdings ein Checkpoint-Rollback-Verfahren ausreichend.

Die drei gängigen Checkpoint-Verfahren unterscheiden sich in den Vorgehensweisen zur Sicherstellung der Konsistenz der durch die einzelnen Checkpoints gebildeten Schnitte. Zu jedem der drei Ansätze existiert eine Vielzahl unterschiedlicher Implementierungen, die sich einerseits im Systemmodell unterscheiden und andererseits auf Kosten höherer Komplexität die Effizienz verbessern. Im Folgenden werden die drei Ansätze grundsätzlich beleuchtet und verglichen; ausführliche Abhandlungen sind in [Jal98] und [EAWJ02], eine kürzere Beschreibung auch in [KK07] zu finden. In allen Fällen entspricht ein *Checkpoint* dem vollständigen, gespeicherten Zustand eines Prozesses zu einem Zeitpunkt.

5.1.1 Unkoordiniertes Checkpointing

Beim unkoordinierten Checkpointing[3] wird nicht sichergestellt, dass die gespeicherten Checkpoints der einzelnen Prozesse einen konsistenten Schnitt bilden. Jeder Prozess (bzw. dessen Manager) kann unabhängig von anderen Prozessen entscheiden, wann sein aktueller Zustand gespeichert wird. Dabei werden keine bereits gespeicherten Checkpoints überschrieben, sondern zunächst nur neue Checkpoints erzeugt. Im Fehlerfall[4] muss zunächst untersucht werden, welche Checkpoints zusammen einen konsistenten Schnitt bilden, zu dem der Zustand des Systems zurückgesetzt werden kann.

Ein grundsätzlicher Vorteil von unkoordiniertem Checkpointing ist der verhältnismäßig geringe Aufwand, der im Normalbetrieb des Systems zu leisten ist, da für jeden Prozess getrennt entschieden werden kann, wann ein Checkpoint gespeichert werden soll. Beispielsweise kann die aktuelle Größe des zu speichernden Zustands eines Prozesses bei der Wahl des Zeitpunkts berücksichtigt werden.

Im Fehlerfall wird untersucht, welche Checkpoints zusammen einen konsistenten Schnitt ergeben. Die Unabhängigkeit der Checkpoint-Erstellung im Normalbetrieb führt hier zu gravierenden Nachteilen. In Abbildung 5.1 ist ein Systemablauf dargestellt, in dem die Prozesse unabhängig voneinander Checkpoints

[3]Unkoordiniertes Checkpointing wird auch als *asynchrones Checkpointing* bezeichnet.

[4]In diesem Kapitel wird mit *Fehler* oder *Fehlerfall* bezeichnet, dass ein *Fehlerzustand* im System auftritt, erkannt wird, und ein Rollback auf einen konsistenten Schnitt durchgeführt werden soll.

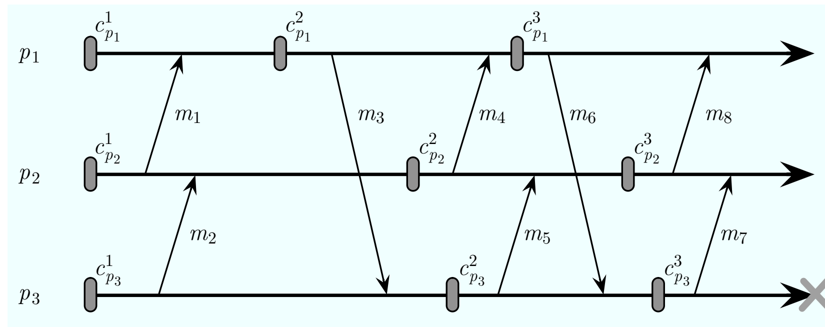


Abbildung 5.1.: Domino-Effekt bei unkoordiniertem Checkpoint-Verfahren

gespeichert haben. In der Abbildung sind die Berechnungsfortschritte dreier Prozesse p_1, p_2 und p_3, sowie ausgetauschte Nachrichten zwischen diesen als Pfeile dargestellt. Gespeicherte Checkpoints der Prozesse sind als dunkle, abgerundete Rechtecke visualisiert. Bei unkoordiniertem Checkpointing können sogenannte *unnütze* Checkpoints auftreten. Das sind Checkpoints, die auf Grund der Kommunikationsabhängigkeiten nie Teil eines konsistenten Schnitts sein können und deren Aufwand zur Speicherung dementsprechend vergebens war, wie beispielsweise $c_{p_1}^3$ in der Abbildung.

Verschlimmert wird die Situation, wenn viele Checkpoints hintereinander auf Grund der Kommunikationsabhängigkeiten keinen konsistenten Schnitt ergeben. In Abbildung 5.1 ist genau dieser Fall dargestellt. Bei einem Fehler von Prozess p_3 wird zunächst dessen jüngster Checkpoint $c_{p_3}^3$ betrachtet. Beim Zurücksetzen zu diesem Checkpoint wäre das Senden der Nachricht m_7 nicht mehr reflektiert, insofern dürfte zur Erhaltung der Konsistenz auch das Empfangen dieser Nachricht nicht im System reflektiert werden. Deshalb müsste auch p_2 zu dem letzten Checkpoint $c_{p_2}^3$ zurückgesetzt werden. Auf Grund der Nachricht m_8 gilt das gleiche auch für Prozess p_1, der dann zum letzten Checkpoint $c_{p_1}^3$ zurückgesetzt werden müsste. Dessen Nachricht m_6 führt allerdings dazu, dass kein konsistenter Schnitt mit $c_{p_3}^3$ möglich ist, somit muss p_3 bis zum Checkpoint $c_{p_3}^2$ zurückgesetzt werden. Dies setzt sich mit weiteren Abhängigkeiten, die jeweils die Konsistenz verletzen, fort bis zum Startzustand der drei Prozesse. Dieser Effekt bei unkoordiniertem Checkpointing wird als *Domino-Effekt* bezeichnet und stellt das größte Problem des Verfahrens dar, da beliebig viel der bereits berechneten Ergebnisse beim Rollback verworfen werden müssen.

Bei unkoordiniertem Checkpointing dürfen zudem nicht beliebig alte Checkpoints gelöscht werden, da nicht sichergestellt ist, dass ein neuer Checkpoint eines Prozesses Teil eines konsistenten Schnitts ist. Da Speicherplatz begrenzt ist, muss

eine *Garbage Collection* realisiert werden, mit Hilfe derer nach sinnvoller Strategie unnütze oder nicht mehr benötigte Checkpoints ermittelt und gelöscht werden.

Es existieren verschiedene Verfahren, im Fehlerfall die Checkpoints zu ermitteln, die einen konsistenten Schnitt bilden. Während des fehlerfreien[5] Betriebs werden die Abhängigkeiten der Prozesse durch Nachrichten protokolliert. Im Fehlerfall initiiert ein koordinierender Prozess einen Rollback, indem er eine Nachricht an alle sendet, um die Abhängigkeiten zu ermitteln. Jeder Prozess antwortet und der Initiator kann mit Hilfe der empfangenen Informationen den konsistenten Schnitt berechnen. Die Berechnung des Schnittes basiert beispielsweise auf *rollback-dependency-Graphen* (vgl. [BL88]).

5.1.2 Koordiniertes Checkpointing

Bei *Koordiniertem Checkpointing*[6] wird das Speichern der einzelnen Checkpoints der Prozesse koordiniert. Verschiedene Protokolle werden eingesetzt um sicherzustellen, dass die jeweils zuletzt gespeicherten Checkpoints jedes Prozesses zusammen einen konsistenten Schnitt bilden. Dadurch kann weder der Domino-Effekt auftreten, noch entstehen unnütze Checkpoints. Ferner muss jeder Prozess nur einen einzigen Checkpoint gespeichert halten, sodass der benötigte Speicherplatz reduziert wird und der Aufwand für eine Garbage Collection entfällt.

Der Hauptnachteil des koordinierten Checkpointing ist die Ineffizienz beim Erstellen der Checkpoints eines neuen konsistenten Schnitts, da die Koordination zusätzlichen Aufwand darstellt und in dem Zeitraum, bis alle Prozesse ihre Checkpoints gespeichert haben, die Anwendungsberechnungen nur eingeschränkt fortgesetzt werden können. Verschiedene Realisierungen des allgemeinen koordinierten Checkpointing existieren, in denen versucht wird, den Performanzverlust zu mindern.

Ein einfacher Ansatz ist die Unterbindung jeglicher Kommunikation der Anwendung während das Checkpointing-Protokoll abläuft (vgl. [TS84]). In einem Protokoll mit zwei Phasen sendet zunächst ein Koordinator eine Nachricht an alle Prozesse und fordert sie auf, einen Checkpoint zu speichern. Jeder Prozess empfängt alle noch wartenden Nachrichten, darf aber keine Anwendungsnachrichten mehr versenden. Dann speichert jeder Prozess einen vorläufigen Checkpoint und sendet ein `commit` an den Koordinator, der wiederum nach Erhalt aller Meldun-

[5] Als fehlerfreier Betrieb wird in diesem Kapitel der Ablauf des Systems bezeichnet, solange kein Fehlerzustand *erkannt* wurde. Es wird nicht gefordert, dass kein (unerkannter) Fehlerzustand vorliegen darf.

[6] Koordiniertes Checkpointing wird auch als *synchrones Checkpointing* bezeichnet.

gen ein `commit` oder `abort` an alle Prozesse schickt. Diese verwerfen bei einem `abort` ihre Checkpoints oder ersetzen in einer atomaren Aktion ihren letzten Checkpoint mit dem vorläufigen Checkpoint. Erst danach können die Prozesse normal ihre Berechnungen fortführen.

Dieses ineffiziente Verfahren wurde durch nicht-blockierende Verfahren verbessert. Hier tritt das Problem auf, dass ein nicht-blockierter Prozess nach dem Speichern des Checkpoints eine Anwendungsnachricht versenden könnte, die ein anderer Prozess vor dem eigentlichen Checkpoint-Auftrag erhält und verarbeitet und dadurch die Konsistenz des durch die Checkpoints gebildeten Schnitts verletzt würde. Es wurden Protokolle entwickelt, bei denen die Checkpoints mit Sequenznummern ausgestattet werden und so genannte Marker-Nachrichten mit diesen Sequenznummern verschickt werden. Hält die Nachrichtenübermittlung eine FIFO-Semantik ein und sendet jeder Prozess immer die Sequenznummer seiner letzten empfangenen Marker-Nachricht mit, so kann ein Prozess bei eingehenden Nachrichten erkennen, ob diese vor oder nach Erhalt des Checkpoint-Aufrufs versendet wurden (vgl. [EJZ92] und [Sil97]).

Ein weiterer koordinierender Ansatz ist Checkpointing mit synchronisierten Uhren. Synchronisierte Uhren ermöglichen die Erstellung eines Checkpoints jedes Prozesses zur (etwa) gleichen Zeit; dies erfordert allerdings die Kenntnis der maximal benötigten Zeiten der Nachrichtenzustellung sowie die Kenntnis der maximal notwendigen Zeit, einen Fehler in einem Prozess zu erkennen. Zunächst wird dabei die Zeitspanne abgewartet, die zur Nachrichtenzustellung und zum Erkennen eines fehlerhaften Prozesses benötigt wird, ohne neue Nachrichten zu versenden. Zusätzlich muss berücksichtigt werden, dass die Synchronisierung von Uhren in verteilten Systemen nie exakt ist, deshalb muss auch der maximale Unterschied der einzelnen Uhrzeiten der Rechner noch als Zeitspanne gewartet werden. Nach diesen Zeitspannen können alle Prozesse einen Checkpoint speichern. Diese Methode scheitert in der Praxis in der Regel an der Kenntnis bzw. Garantie dieser Zeitschranken.

Die Performanzeinbußen der meisten koordinierenden Verfahren liegen auch darin begründet, dass sich *alle* Prozesse des Systems koordinieren müssen, um jeweils einen neuen Checkpoint zu speichern. Wird ausgehend von einem existierenden konsistenten Schnitt von einem Prozess das Speichern eines neuen konsistenten Schnitts angestoßen, so kann die Performanz der Erstellung drastisch verbessert werden, wenn nur ein Teil der Prozesse des Systems einen neuen Checkpoint speichern müssen. Um einen konsistenten Schnitt zu erhalten kann bei allen Prozessen, die nicht direkt oder transitiv vom Initiator abhängig sind, der letzte Checkpoint verwendet werden, da ohne neue Abhängigkeiten der letzte Checkpoint Teil eines konsistenten Schnitts ist. Es werden dazu Protokolle angestrebt,

die lediglich *die* Prozesse zum neuen Erstellen eines Checkpoints zwingen, zu denen eine direkte oder transitive Abhängigkeit vom Initiator besteht.

In einem zwei-Phasen Protokoll nach Koo und Toueg kann die Menge der Prozesse ermittelt werden, die einen neuen Checkpoint erstellen müssen (vgl. [KT86]). Dabei sendet der Initiator jedem Prozess, mit dem er seit der letzten Erstellung eines Checkpoints kommuniziert hat, eine Nachricht. Diese Prozesse schicken iterativ wiederum Nachrichten an alle Prozesse, mit denen Sie kommuniziert haben, bis die gesamte transitive Hülle erreicht wurde, wobei jeder der Prozesse dem Initiator mitgeteilt wird. In der zweiten Phase beauftragt der Initiator alle so erreichten Prozesse, einen Checkpoint zu erstellen. Die Prozesse dürfen zwischen der ersten Nachricht und dem Erstellen des Checkpoints keine Anwendungsnachrichten versenden, um die Konsistenz des Checkpoints nicht zu verletzen.

5.1.3 Kommunikationsinduziertes Checkpointing

Die Idee von kommunikationsinduziertem Checkpointing ist, die Vorteile von koordiniertem und unkoordiniertem Checkpointing zu verbinden. Analog zum unkoordinierten Checkpointing kann jeder Prozess unabhängig entscheiden, einen Checkpoint zu speichern. Diese Checkpoints werden als *freiwillige* Checkpoints bezeichnet. Um allerdings sicherzustellen, dass die gespeicherten Checkpoints der Prozesse einen konsistenten Schnitt bilden, werden mittels *piggybacking*[7] mit Anwendungsnachrichten Checkpointing-Informationen des Senders versandt, sodass der Empfänger entscheiden kann, ob *vor* der Verarbeitung der Anwendungs-Nachricht ein Checkpoint gespeichert werden muss. Diese Checkpoints werden als *erzwungene* Checkpoints bezeichnet. Mit diesem Ansatz wird die Konsistenz des Schnittes sichergestellt, den die gespeicherten Checkpoints bilden. Somit kann weder der Domino-Effekt auftreten, noch können unnütze Checkpoints gespeichert werden. Eine Garbage Collection kann – je nach Implementierung – ebenfalls vermieden oder sehr einfach realisiert werden. Dennoch können die Prozesse ohne koordinierende Maßnahmen entscheiden, freiwillige Checkpoints zu speichern.

Die Performanzeinbußen der Synchronisation der koordinierenden Verfahren werden vermieden. Einerseits müssen keine zusätzlichen Nachrichten im System versendet werden, andererseits kann jeder Prozess nach dem Speichern seines Checkpoints direkt seine Berechnungen wieder fortführen, ohne auf andere Prozesse warten zu müssen. Es existieren zwar nicht-blockierende, koordinierende

[7] Piggybacking wird auch als Huckepack-Verfahren bezeichnet: Systeminformationen werden an Anwendungsnachrichten angehängt und mit diesen zusammen versendet.

Checkpoint-Verfahren, diese schließen allerdings nicht nur die minimale Anzahl an Prozessen für die Erstellung eines konsistenten Schnitts ein. Cao und Singhal haben bewiesen, dass *kein* koordinierender Algorithmus existieren kann, der nicht-blockierend nur die minimal notwendige Anzahl an Prozessen zum Speichern eines Checkpoints veranlasst (vgl. [CS98]).

Beim Ansatz des kommunikationsinduzierten Checkpointing werden wiederum zwei Verfahren unterschieden: *Modell-basierendes*[8] und *Index-basierendes*[9] Checkpointing. Die beiden Verfahren unterscheiden sich in der Art der Entscheidung, ob ein erzwungener Checkpoint beim Empfang einer Nachricht zu speichern ist.

Beim Modell-basierenden Checkpointing wird versucht, die Graph-Strukturen, die zur Inkonsistenz eines Schnitts führen können, verteilt zu erkennen und durch das Speichern eines Checkpoints zu durchbrechen. Die Graph-Strukturen werden als *zigzag-Paths*, *Z-Cycles* (vgl. [NX95]) oder *z-dependencies* (vgl. [CS98]) bezeichnet. Kommunikationsinduziertes Checkpointing ist nicht zentral koordiniert, jeder Prozess muss nach den eigenen Informationen und den mit einer Anwendungsnachricht erhaltenen Informationen lokal entscheiden, ob die Möglichkeit besteht, dass eine solche Graph-Struktur im System entsteht und ggf. einen (erzwungenen) Checkpoint speichern. Unter Umständen werden auf Grund der unvollständigen Information über die Gesamtstruktur des Graphen deshalb mehr erzwungene Checkpoints gespeichert als eigentlich zur Gewährung der Konsistenz notwendig wären, da mehrere Prozesse die Möglichkeit einer solchen Struktur erkennen.

Index-basierende Checkpoint-Verfahren nutzen logische Uhren für Checkpoints, deren Werte ebenfalls per piggybacking an Anwendungsnachrichten angehängt werden. Anhand der Uhrwerte des letzten Checkpoints des Senders einer Nachricht kann der Empfänger entscheiden, ob ein neuer Checkpoint erzeugt werden muss, um die Konsistenz des Schnittes zu gewährleisten. Ein einfaches, Index-basierendes Checkpointing-Protokoll ist in [BCS84] beschrieben: jeder gespeicherte Checkpoint eines Prozesses hat einen skalaren Index, der für jeden neuen Checkpoint inkrementiert und an Anwendungsnachrichten angehängt wird. Empfängt ein Prozess eine Nachricht mit einem höheren Index als sein eigener, so speichert der Prozess einen erzwungenen Checkpoint bevor die Nachricht verarbeitet wird und setzt den eigenen Index gleich dem Empfangenen. Auch bei Index-basierendem Checkpointing kann nicht ausgeschlossen werden, dass mehr

[8] engl.: model based
[9] engl.: index based

Checkpoints erzwungen werden als zur Gewährung der Konsistenz des Schnittes notwendig wären.

Obwohl die beiden kommunikationsinduzierten Verfahren unterschiedliche Ansätze verfolgen, um zu entscheiden, ob ein erzwungener Checkpoint gespeichert werden muss, zeigten Hélary, Mostéfaoui und Raynal, dass die Verfahren fundamental äquivalent sind (vgl. [HMR97]). Beide Ansätze führen demnach in gleichem Maße zu mehr erzwungenen Checkpoints als minimal zur Erhaltung der Konsistenz notwendig wären.

5.1.4 Diskusssion

Die Entwicklung von unkoordinierten zu koordinierten Checkpoint-Verfahren hat ihren Ursprung in der technischen Weiterentwicklung der Rechnernetze. Als die Vernetzung noch verhältnismäßig geringe Bandbreite und hohe Latenz hatte, konnte die Effizienz von unkoordiniertem Checkpointing überwiegen, da das mehrfache Schreiben von Checkpoints weniger Aufwand als die zusätzlichen Koordinationsnachrichten bedeutete. Mit der Weiterentwicklung der Übertragungstechniken änderte sich dieser Gesichtspunkt, sodass trotz der Koordinationsnachrichten bei koordiniertem Checkpointing eine höhere Effizienz erzielt werden konnte. Heutzutage ist lediglich die Abwägung zwischen dem Nachteil des Domino-Effekts bei unkoordiniertem Checkpointing und dem Overhead durch Synchronisation beim koordinierten Checkpointing entscheidend. Da der Domino-Effekt zum vollständigen Verlust der bisher berechneten Ergebnisse führen kann, sind hier die koordinierenden Verfahren — abgesehen von Spezialfällen mit sehr eingeschränkter Kommunikation – klar vorzuziehen.

In der Theorie ist kommunikationsinduziertes Checkpointing sowohl den unkoordinierten, als auch den koordinierten Checkpoint-Verfahren in der Performanz überlegen, da die Vorteile beider Verfahren kombiniert wurden. Zudem sollte kommunikationsinduziertes Checkpointing gut für große Systeme mit vielen Prozessen skalieren, da nicht alle Prozesse synchronisiert werden, sondern nur die tatsächlich auf Grund von Kommunikationsabhängigkeiten betroffenen Prozesse zum Speichern eines Checkpoints veranlasst werden. Praktische Tests und Simulationen zeigen jedoch genau das Gegenteil: kommunikationsinduziertes Checkpointing ist effizient für kleine Systeme mit wenigen Prozessen, mit steigender Anzahl von Prozessen und zunehmender Kommunikation zwischen diesen steigt der Overhead des Verfahrens jedoch rapide an.

Die Einteilung sowie die Vorherbestimmung des benötigten Plattenspeichers für einen Anwendungslauf sind schwer zu planen, da zu nicht vorherbestimm-

ten Zeitpunkten erzwungene Checkpoints auftreten können. Der große Vorteil, dass Prozesse eigenständig Checkpoints zu selbstgewählten Zeitpunkten erstellen können, scheint sich nicht auszuzahlen, da in experimentellen Implementierungen mindestens doppelt so viele erzwungene Checkpoints wie freiwillige Checkpoints auftraten (vgl. [AER+99]).

Um die Ursachen der schlechten Performanz des kommunikationsinduzierten Ansatzes in der Praxis genauer zu untersuchen, muss zunächst der Overhead erklärt werden. Der Gesamt-Overhead eines Checkpointing-Verfahrens entspricht der zusätzlichen Ausführungszeit eines Systems beim Einsatz des Verfahrens. Diese zusätzliche Ausführungszeit wird von verschiedenen Faktoren beeinflusst. Die Größe der Prozesszustände, die gesichert werden müssen, beeinflusst die Wartezeit für das Speichern eines Checkpoints eines Prozesses. Zusätzlicher Kommunikationsaufwand durch das Checkpoint-Verfahren, üblicherweise nur bei koordiniertem Checkpointing, beeinflusst die Ausführungszeit sowie auch synchronisationsbedingte Wartezeiten der Prozesse während blockierenden Protokollen.

Ferner beeinflusst die Häufigkeit der Speicherung von Checkpoints auf zwei Arten den Overhead. Einerseits steigt die Ausführungszeit mit jedem zusätzlichen Checkpoint, andererseits muss im Fehlerfall berücksichtigt werden, wie viel der bereits ausgeführten Berechnungen der Anwendung beim Rollback verloren geht. Je kürzer die Intervalle zwischen den einzelnen Checkpoints, umso höher ist demnach der Overhead durch das reine Speichern von Checkpoints, aber umso niedriger ist der Overhead eines Rollback. Die Abhängigkeit des Gesamt-Overheads von der Länge der Intervalle zwischen den Checkpoints lässt sich mathematisch erfassen. An dieser Stelle ist ein Checkpoint-Intervall als der Zeitraum zu verstehen, der ab einem gespeicherten Checkpoint eines Prozesses bis nach dem erfolgreichen Speichern des darauf folgenden Checkpoints vergeht; in Abschnitt 5.2.1 wird der Begriff Checkpoint-Intervall für das MoDiS-Projekt formal definiert.

Wir nehmen an, dass allgemein das Auftreten von Fehlerzuständen in der Berechnung eines Prozesses einer Poisson-Verteilung entspricht (vgl. [KK07], Kap. 2.4.1). Die Wahrscheinlichkeit, dass innerhalb einer abstrakten Zeiteinheit ein Fehlerzustand auftritt, wird durch die Fehlerrate[10] λ angegeben. Solche Prozesse werden auch als Poisson-Prozesse bezeichnet.

Eine gängige Modellierung der vollständigen Ausführungszeit eines Poisson-Prozesses, in der kein Checkpointing enthalten ist, sondern im Fall eines Fehlers einfach neu gestartet wird, ist in [KK07] zu finden: Sei $T(I)$ die erwartete Aus-

[10] In diesem Abschnitt bezeichnet λ die Fehlerwahrscheinlichkeit oder Ereignisrate der Poisson-Verteilung gemäß der üblichen Nomenklatur aus der Wahrscheinlichkeitstheorie und ist nicht mit der Lokalitätsstruktur λ der MoDiS-Systeme zu verwechseln.

führungszeit inklusive der Verluste durch Neustart im Fehlerfall, wobei I die Ausführungszeit des Prozesses ohne Fehler sei. Dieser Fall tritt mit Wahrscheinlichkeit $\mathrm{e}^{-\lambda I}$ ein. Somit trägt der fehlerfreie Fall zur Ausführungszeit $I\mathrm{e}^{-\lambda I}$ bei. Der Beitrag für den Fall, in dem mindestens ein Fehler während der Ausführung auftritt, ist komplexer zu berechnen. Angenommen, nach x Zeiteinheiten tritt der erste Fehler auf, so ergibt sich die Ausführungszeit zu $x + T(I)$. Die Wahrscheinlichkeit, dass im Intervall $[x, x + \mathrm{d}x]$ der erste Fehler auftritt, ist dann $\lambda\mathrm{e}^{-\lambda x}\mathrm{d}x$, wobei x im Intervall $[0, I]$ liegen kann. Der Beitrag zurAusführungszeit ergibt sich damit zu:

$$\int_{x=0}^{I} (x + T(I))\lambda\mathrm{e}^{-\lambda x}\mathrm{d}x = \frac{1}{\lambda} + T(I) - \mathrm{e}^{-\lambda I}\left\{\frac{1}{\lambda} + I + T(I)\right\}$$

Dieser Beitrag wird zum fehlerfreien Fall addiert:

$$T(I) = I\mathrm{e}^{-\lambda I} + \frac{1}{\lambda} + T(I) - \mathrm{e}^{-\lambda I}\left\{\frac{1}{\lambda} + I + T(I)\right\}$$

Aufgelöst nach der erwarteten Ausführungszeit $T(I)$ ergibt sich daraus

$$T(I) = \frac{\mathrm{e}^{\lambda I} - 1}{\lambda}.$$

Diese Abschätzung der Gesamtlaufzeit eines Poisson-Prozesses, der im Fehlerfall neu gestartet wird, kann als Abschätzung für die Laufzeit eines Poisson-Prozesses innerhalb eines festgelegten Intervalls I zwischen zwei Checkpoints verwendet werden. Anstatt des Neustarts wird zum letzten Checkpoint zurückgesetzt. Lediglich das Speichern des folgenden Checkpoints muss analog in die Abschätzung integriert werden. Die Ausführungszeit für das Sichern eines Checkpoints sei konstant, fehlerfrei und betrage c Zeiteinheiten. Dann ergibt sich die erwartete Ausführungszeit für ein Intervall I zwischen zwei Checkpoints als (vgl. [PP07]):

$$T(I) = \frac{\mathrm{e}^{\lambda I} + \lambda c - 1}{\lambda}$$

Die durchschnittliche Ausführungszeit $T(I)$ enthält sowohl die Kosten des Checkpointing als auch die mit der Fehlerwahrscheinlichkeit gewichteten Kosten im Fall eines Rollbacks. Der Gesamt-Overhead kann angegeben werden als Overhead-Ratio $O(I)$ eines Intervalls I und wird berechnet als das Verhältnis aus Checkpoint-Rollback-Zeit und Ausführungszeit der Anwendung:

$$O(I) = \frac{T(I) - I}{I} = \frac{T(I)}{I} - 1 = \frac{\mathrm{e}^{\lambda I} + \lambda c - 1}{\lambda I} - 1$$

Da die Poisson-Verteilung stationär ist und konstante Intervalle angenommen wurden, ist die Overhead-Ratio $O(I)$ eines Intervalls gleich der Overhead-Ratio des gesamten Systems. Der Einfluss der Intervall-Länge auf den Overhead, einerseits durch viel Aufwand bei häufigem Speichern von Checkpoints und andererseits durch viel Verlust bei Rollback zu weit zurückliegenden Schnitten, lässt sich an der Overhead-Ratio erkennen.

In Abbildung 5.2 ist die Overhead-Ratio kommunikationsinduzierter Checkpoint-Verfahren dargestellt. Der relative Verlauf der Kurve ist entscheidend und für relevante Werte, also $0 < \lambda < 1$ und $c > 0$ wie abgebildet. Ausgehend vom Minimum der Kurve steigt die Overhead-Ratio sowohl für kürzere, als auch für längere Chekpoint-Intervalle an. Je kürzer die Checkpoint-Intervalle, umso schlechter wird das Verhältnis aus Rechenzeit der Anwendung und Rechenzeit für den Checkpointing-Vorgang bis hin zu dem Extremfall, dass die gesamte Rechenzeit nur für Checkpointing aufgewendet wird: die Ratio geht gegen unendlich. Bei vom Minimum ausgehend größeren Checkpoint-Intervallen überwiegt der Verlust der Berechnungen im Fall eines Rollbacks, daher steigt auch hier die Overhead-Ratio an. Das Minimum der Kurve liegt bei dem Checkpoint-Intervall mit günstigstem Verhältnis, fortan als Idealintervall I_{ideal} bezeichnet. Die entsprechend erreichte minimale Overhead-Ratio wird analog als Minimal-Overhead O_{min} bezeichnet.

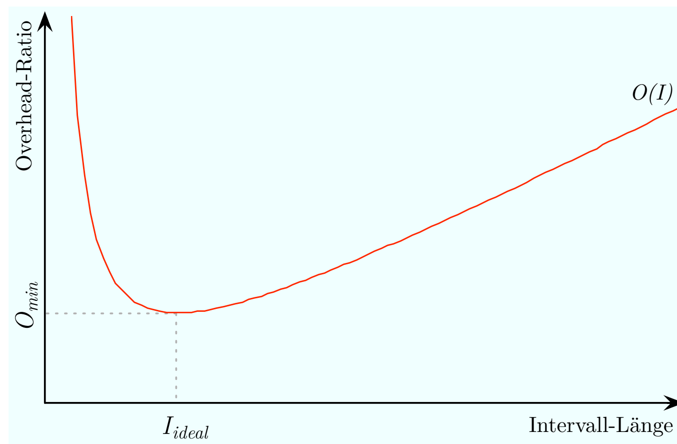


Abbildung 5.2.: Overhead-Ratio in Abhängigkeit zum Checkpoint-Intervall

Der generelle Verlauf der Kurve ist für jedes der Checkpoint-Verfahren, also auch für koordiniertes und unkoordiniertes Checkpointing analog, lediglich die Steigungen und die absolute Position des Minimums unterscheiden sich. Der Overhead von jedem der Verfahren ist davon abhängig, ob das tatsächlich auftretende Checkpoint-Intervall nahe dem Idealintervall liegt.

Bei unkoordiniertem Checkpointing kann zwar eine untere Schranke für das Intervall eingehalten werden, das Intervall kann jedoch beliebig groß werden, da beim Rollback durch den Domino-Effekt beliebig weit (bis maximal zum Startzustand des Systems) zurückgesetzt werden muss. Beim koordinierten Checkpointing kann zwar das Idealintervall gut angenähert werden, durch die hohen Synchronisationskosten liegt in diesem Fall die erzielbare minimale Overhead-Ratio O_{min} generell höher als bei unkoordiniertem oder kommunikationsinduziertem Checkpointing. Bei kommunikationsinduziertem Checkpointing kann eine obere Schranke eingehalten werden, indem jeder Prozess beispielsweise jeweils nach Ablauf des Idealintervalls einen freiwilligen Checkpoint speichert. Durch Kommunikationsabhängigkeiten können allerdings zusätzlich erzwungene Checkpoints gespeichert werden, die das tatsächlich resultierende Checkpoint-Intervall beliebig verkleinern und somit die Overhead-Ratio beliebig ansteigt. Dieser Anstieg des Overheads bei vielen erzwungenen Checkpoints und den daraus resultierenden kurzen Checkpoint-Intervallen ist der Grund für die schlechte Performanz des Verfahrens in der Praxis.

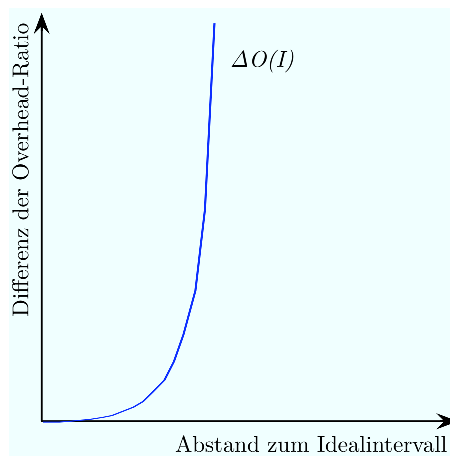


Abbildung 5.3.: Differenz der Steigung der Teilfunktionen ausgehend vom Idealintervall

Der Verlauf der Overhead-Ratio in Abbildung 5.2 zeigt bereits, dass eine Verkleinerung der Checkpoint-Intervalle ausgehend vom Idealintervall den Overhead deutlich stärker erhöht als eine Vergrößerung. Dieser Zusammenhang kommt deutlich hervor, wenn man die Differenz der beiden Teilkurven in Abhängigkeit des Abstands zum Idealintervall betrachtet. In Abbildung 5.3 ist diese Differenz dargestellt. Die Kurve zeigt, wie viel höher die Performanzeinbußen bei verkürzten Checkpoint-Intervallen im Gegensatz zu verlängerten Intervallen, ausgehend vom Idealintervall, sind. Da diese Differenz der Overhead-Ratio – unabhängig von

den tatsächlichen Werten von λ und c – exponentiell wächst, kann die Performanz des kommunikationsinduzierten Ansatzes in der Praxis entscheidend verbessert werden, wenn kurze Checkpoint-Intervalle durch *erzwungene* Checkpoints vermieden werden. Im Rahmen dieser Arbeit wurde ein Verfahren entwickelt, diese Fälle zu vermeiden, indem Kommunikationsabhängigkeiten der Anwendung im Übersetzer analysiert werden und anhand dieser die Platzierung *freiwilliger* Checkpoints derart vorbereitet wird, dass kurze Checkpoint-Intervalle erzwungener Checkpoints vermieden werden. Das Laufzeit-Management überprüft dann die Vorbereitung des Übersetzers und führt das eigentliche Speichern der Checkpoints aus. Durch die Anpassung der Checkpoint-Platzierung an die Anwendung und deren Kommunikationsverhalten wird die Performanz des kommunikationsinduzierten Checkpoint-Verfahrens verbessert.

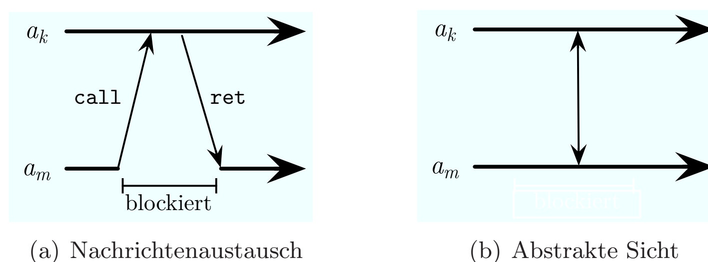
5.2 Kommunikationsinduziertes Checkpointing in MoDiS

In diesem Abschnitt werden zunächst die Details des im Rahmen dieser Arbeit für das MoDiS-Projekt entstandenen, kommunikationsinduzierten Checkpoint-Verfahrens beschrieben. Dabei wird erklärt, wie das allgemeine Verfahren nach Briatico (vgl. [BCS84]) an die Abhängigkeiten in MoDiS-Systemen angepasst wird. Darauf folgend wird das Protokoll erklärt, mit dem das Management die Checkpointinformationen verwaltet und Entscheidungen über erzwungene Checkpoints trifft. Anschließend wird die Vorbereitung der Platzierung freiwilliger Checkpoints im Übersetzer erarbeitet, die den hohen Overhead kurzer Checkpoint-Intervalle umgeht.

5.2.1 Speicherung der Rücksetzpunkte

Die Kommunikation in MoDiS basiert auf dem Operationen-orientierten Rendezvous, realisiert durch K-Akteure, die über K-Ordern Schnittstellen zur Kommunikation anbieten (vgl. Abschnitte 2.2.1.4 und 4.1.2.3). Die Kommunikation zwischen M-Akteuren und K-Akteuren über das Operationen-orientierte Rendezvous ist eine spezielle, eingeschränkte Form des Nachrichtenaustauschs. Im Gegensatz zu allgemeinen Nachrichtenkommunikationssystemen können nicht in beliebiger Weise Nachrichten verschickt werden. Das Nachrichtenschema ist in Abbildung 5.4(a) dargestellt[11]: Ein M- oder K-Akteur $a_m \in A$ sendet zunächst den Aufruf

[11] In diesem Kapitel werden zur besseren Übersicht die einzelnen INSEL-Ereignisse beim Berechnungsfortschritt in den Graphen nur dargestellt, falls diese zum Verständnis notwendig



(a) Nachrichtenaustausch (b) Abstrakte Sicht

Abbildung 5.4.: Nachrichtenaustausch und abstrakte Sicht des Operationen-orientierten Rendezvous

an einen K-Akteur $a_k \in A_k$. Während der Verarbeitung dieser Nachricht und der aufgerufenen K-Order ist der aufrufende Akteur a_m blockiert, sein Zustand verändert sich in dieser Zeit nicht. Nach Abarbeitung der K-Order sendet a_k eine Antwortnachricht zurück an a_m, dieser wird entblockiert und kann die Nachricht verarbeiten.

Die speziellen Eigenschaften dieses festgelegten Schemas des Nachrichtenaustauschs ermöglichen eine Optimierung des allgemeinen Prinzips von kommunikationsinduziertem Checkpointing für die Anwendung in MoDiS. Zunächst ist von Vorteil, dass bei jedem Nachrichtenaustausch ein Informationsfluss in beide Richtungen statt findet, also von a_m zu a_k und zurück. Ferner kann für das Verfahren genutzt werden, dass sich der Zustand des Aufrufers a_m zwischen seinem Aufruf und der erhaltenen Antwort auf Grund der Blockierung nicht ändert. Für das kommunikationsinduzierte Checkpoint-Verfahren ist relevant, welche Möglichkeiten der Informationsweitergabe per piggybacking über zuletzt gespeicherte lokale Checkpoints bestehen, bevor die Nachrichten selbst verarbeitet werden.

Abstrakt betrachtet kann die Kommunikation im Operationen-orientierten Rendezvous als bidirektionale Informationsweitergabe verstanden werden, wie in Abbildung 5.4(b) visualisiert. In beiden Richtungen können weitergeleitete Informationen über Checkpoints genutzt werden, noch bevor die Anwendungsnachrichten, also K-Order-Aufruf `call` und K-Order-Antwort `ret` verarbeitet werden. Beim Aufruf der K-Order ist dies offensichtlich: Die Information des Aufrufers a_m wird mitgesendet, der K-Akteur erhält die Nachricht, das Management überprüft die Checkpoint-Informationen, gegebenenfalls wird ein Checkpoint gespeichert, erst dann wird die Nachricht verarbeitet, also die K-Order ausgeführt.

sind. Ansonsten geben die etwas dicker visualisierten Pfeile den relativen zeitlichen Berechnungsfortschritt der Akteure wieder.

In der Gegenrichtung wird die Blockierung des aufrufenden Akteurs a_m nach dem Absenden seines Aufrufs `call` genutzt. Dieser Akteur muss noch vor dem Absenden des Aufrufs das Speichern eines lokalen Checkpoints vorbereiten und die Änderungen in seinem Zustand für das Senden des Aufrufs geeignet protokollieren. Empfängt dann der Aufrufer a_m letztendlich die Antwortnachricht `ret` von a_k, so muss das Management zunächst prüfen, ob a_m zur Einhaltung der Konsistenz bereits vor seinem Aufruf einen erzwungenen Checkpoint hätte speichern müssen. Ist dies der Fall, so wird der vorher vorbereitete Checkpoint gespeichert und anschließend die Nachricht verarbeitet.

Da der Zustand des Aufrufers von dem Zeitpunkt der Checkpoint-Vorbereitung bis zum Empfang der Antwortnachricht auf Grund der Blockierung, abgesehen vom Senden des Aufrufs, nicht verändert wurde, kann dieses Protokollieren der Änderungen effizient durch eine *copy-on-write* Technik im Speichermanagement realisiert werden. Die wenigen Speicherseiten des Aufrufers, die beim Absenden des Aufrufs geändert werden, kopiert das Speichermanagement beim Schreibzugriff, somit kann ein Checkpoint des Zustands von a_m, wie er *vor* dem Aufruf gewesen ist, zum Zeitpunkt *nach* Erhalt der Antwortnachricht gespeichert werden. Die Performanzeinbußen dieser Technik sind gering, da lediglich im Arbeitsspeicher einige wenige Seiten kopiert werden, auf den langsamen Hintergrundspeicher muss nur geschrieben werden, wenn tatsächlich ein Checkpoint erzwungen wird.

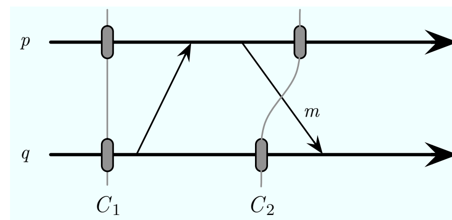


Abbildung 5.5.: Nachricht m wird Teil eines konsistenten Schnitts

Durch dieses Verfahren, bei dem basierend auf dem Operationen-orientierten Rendezvous eine Informationsweitergabe immer vom Aufrufer zum K-Akteur und zurück stattfindet, lässt sich das kommunikationsinduzierte Checkpointing-Verfahren realisieren, ohne Nachrichten zu speichern. In Abbildung 5.5 ist ein Fall dargestellt, in dem ein herkömmliches Nachrichtenkommunikationssystem ein kommunikationsinduziertes Checkpoint-Verfahren ausführt. Gespeicherte Checkpoints der beiden Prozesse sind als graue, abgerundete Rechtecke eingezeichnet. Die Nachricht m von Prozess p zu Prozess q ist Teil des konsistenten Schnitts C_2 und muss deshalb mit gespeichert und im Fall eines Rollbacks reproduziert werden. Nachrichten, die Teil eines konsistenten Schnitts sind, werden als *verwais-*

te Nachrichten[12] bezeichnet. Der Sender der Nachricht konnte nicht informiert werden, dass der Empfänger schon einen Checkpoint gesetzt hatte, wodurch die Nachricht zum Teil eines konsistenten Schnitts wurde, da deren Sende-Ereignis im Schnitt reflektiert, das Empfangs-Ereignis jedoch nicht reflektiert wird.

Beim Operationen-orientierten Rendezvous kann bei jeder Kommunikation auch Information vom Empfänger (K-Akteur) zum Aufrufer (M-Akteur) fließen. Der beschriebene Fall, indem Nachrichten reproduziert werden müssen, kann so vermieden werden, da der Aufrufer (nachträglich) einen Checkpoint vor dem Sendeereignis setzen kann und somit immer sowohl Sende- als auch Empfangsereignis entweder beide nach einem Checkpoint oder beide vor dem nächsten Checkpoint sind. Die Nachrichten selbst sind somit nie Teil eines konsistenten Schnitts. Die Einschränkung, dass innerhalb der Ausführung von K-Ordern deshalb keine Checkpoints gespeichert werden dürfen, stellt in der Praxis kein Problem dar. Auf Grund der blockierenden Semantik des Rendezvous ist es grundsätzlich sinnvoll, die Berechnungen von K-Ordern kurz zu halten und aufwendige Berechnungen nebenläufig durch einen erzeugten Akteur zu spezifizieren.

Eine weitere Anpassung des allgemeinen kommunikationsinduzierten Checkpointing an das Systemmodell im MoDiS-Projekt ist durch die dynamische Akteurerzeugung und -terminierung erforderlich. Wie bereits bei der Analyse der kausalen Abhängigkeiten zur Systembeobachtung erläutert, lassen sich die Abhängigkeiten durch die Erzeugung und Terminierung von Akteuren ähnlich zu Nachrichtenkommunikation modellieren (vgl. 4.1.1.1). Dies gilt auch für das Checkpoint-Verfahren, wobei beim Rollback im Fehlerfall analog zur Reflexion der Akteur-Zustände auch die Existenz der Akteure berücksichtigt werden muss, um Konsistenz zu gewährleisten.

Ein Schnitt, der das Empfangsereignis einer Nachricht m enthält, ist nur konsistent, falls auch das Sende-Ereignis der Nachricht m im Schnitt enthalten ist. Analog ist ein Schnitt, in dem ein Akteur a_m existiert bzw. in dem das erste Ereignis `init` des Akteurs enthalten ist, nur dann konsistent, wenn auch das `create`-Ereignis seiner Erzeugung im Schnitt enthalten ist. Analoges gilt für die Ereignisse `term` und `join` bei der Terminierung von Akteuren. Zur Gewährung der Konsistenz muss deshalb vom Management im Fall eines Rollbacks unter Umständen ein Akteur ganz aus dem System entfernt oder aber ein bereits aufgelöster Akteur wieder erzeugt werden.

Das an die erklärten Kommunikationsmuster in MoDiS angepasste Protokoll für kommunikationsinduziertes Checkpointing wird im Folgenden vorgestellt. Hierfür wird zunächst davon ausgegangen, dass die Übersetzer-Komponente des Ma-

[12] engl.: orphan Messages

nagements bereits nach einer Metrik, die in Abschnitt 5.2.5 erklärt wird, potentielle freiwillige Checkpoints vorbereitet. Im Ausführungscode jeder aktiven Komponente sind an diesen Stellen Aufrufe an die Laufzeitumgebung vorhanden, sodass das Management nach folgendem Protokoll zur Laufzeit freiwillige Checkpoints speichern kann. Je nach tatsächlichem Ablauf des Systems entscheidet das Management unter Umständen auch, einen freiwilligen Checkpoint auszulassen. Zusätzlich sorgt das Protokoll dafür, dass Checkpoints erzwungen werden, wenn dies für die Einhaltung der konsistenten Schnitte notwendig ist.

Im Fall eines Fehlers einer Komponente müssen Teile des Systems einen Rollback durchführen. Diese Teile sind durch die transitiven Kommunikations-Abhängigkeiten der fehlerhaften Komponente seit dem letzten Checkpoint dieser Komponente festgelegt. Es bietet sich daher an, die Abhängigkeitsinformationen bereits zusammen mit der Checkpoint-Information per piggybacking mit Anwendungsnachrichten mitzusenden, um im Fall eines Rollbacks diese Informationen direkt zur Verfügung zu haben.

Jeder Checkpoint des Systems wird eindeutig identifiziert durch die Zugehörigkeit zu einem Akteur sowie den Index der totalen Ordnung aller Checkpoints des Akteurs, $c^i_{a_m}$ sei demnach der i-te Checkpoint des Akteurs $a_m \in A$. Für die folgenden Ausführungen sei zudem der Initialzustand des Systems bezeichnet als $c^0_{a_m}$, wobei a_m die Hauptkomponente `SYSTEM` sei. Auf Grund der dynamischen Prozesserzeugung im MoDiS-Projekt haben Akteure kausale Abhängigkeiten über ihren eigenen Startzustand hinaus zu ihrem erzeugenden Akteur.

Die Checkpoint-Intervalle sind relevant für die Erfassung der Abhängigkeiten, deshalb ist das erste Checkpoint-Intervall eines Akteurs nicht durch dessen Startzustand, sondern durch den letzten Checkpoint seines Erzeugers begrenzt (mit Ausnahme des Akteurs `SYSTEM`). Analog gilt für die Terminierung von Akteuren, dass eine Abhängigkeit vom letzten Checkpoint des terminierenden Akteurs zum nächst folgenden Checkpoint des Erzeugers besteht.

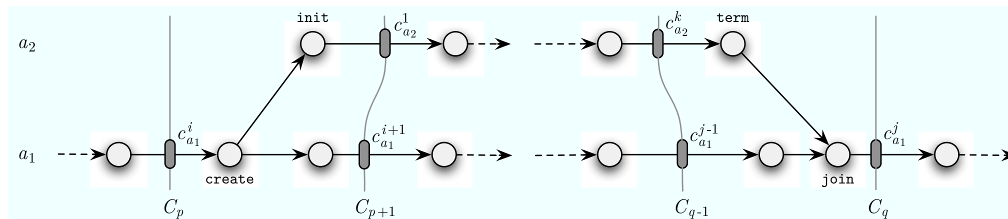


Abbildung 5.6.: Abhängigkeiten durch dynamisches Prozesssystem

In Abbildung 5.6 sind die beiden Situationen anhand von Ausschnitten eines MoDiS-Ereignisgraphs visualisiert. Akteur a_1 erzeugt Akteur a_2, wobei der Checkpoint $c^i_{a_1}$ der letzte Checkpoint vor dem `create`-Ereignis ist. Ausser der dargestellten Erzeugung und Terminierung seien keine weiteren Abhängigkeiten vorhanden, insofern bildet $c^i_{a_1}$ einen konsistenten Schnitt, bezeichnet als C_p. Der nächst folgende konsistente Schnitt ist durch die jeweils ersten Checkpoints beider Prozesse nach der Erzeugung festgelegt, C_{p+1} beinhaltet also die Checkpoints $c^{i+1}_{a_1}$ und $c^1_{a_2}$.

Die Erzeuger-Abhängigkeit ist relevant, falls ein Fehler auftritt, während beide Prozesse aktiv sind, aber zu Schnitt C_p zurückgesetzt wird. In diesem Fall ist die kausale Ursache des Akteurs a_2, nämlich das `create`-Ereignis nicht im Schnitt enthalten, demnach muss beim Rollback der Akteur a_2 entfernt werden. Analog müsste bei einem Fehler, der erst nach dem `join`-Ereignis auftritt, falls ein Rollback zu einem der Schnitte $C_{p+1} \dots C_{q-1}$ durchgeführt wird, der Akteur a_2 vom Management erzeugt und in den Zustand des im Schnitt enthaltenen Checkpoints, beispielsweise $c^k_{a_2}$ bei C_{q-1}, gesetzt werden.

Definition 5.1 (Checkpoint-Intervall)
Seien $c^i_{a_m}$ und $c^{i+1}_{a_m}$ zwei aufeinander folgende Checkpoints des Akteurs $a_m \in A$. Das Checkpoint-Intervall *$I(c^{i+1}_{a_m})$ ist der Zeitraum zwischen den beiden Checkpoints, exklusive der Zeit zum Speichern von $c^i_{a_m}$, inklusive der Zeit zum Speichern des Checkpoints $c^{i+1}_{a_m}$.*
Wird ein Akteur $a_n \in A$ von Akteur a_m erzeugt und sei der letzte gespeicherte Checkpoint von a_m vor der Akteurerzeugung $c^i_{a_m}$, so ist das erste Checkpoint-Intervall $I(c^1_{a_n})$ von a_n begrenzt durch $c^i_{a_m}$ und $c^1_{a_n}$.
Terminiert ein Akteur $a_n \in A$, sei a_m sein Erzeuger und sei der folgende gespeicherte Checkpoint von a_m nach der Auflösung $c^j_{a_m}$, der letzte Checkpoint von a_n vor der Terminierung $c^i_{a_n}$, so ist das letzte Checkpoint-Intervall von a_n begrenzt durch $c^i_{a_n}$ und $c^j_{a_m}$ und wird gesondert als $I(c^i_{a_n}, c^j_{a_m})$ bezeichnet.
Sei a_m die Hauptkomponente `SYSTEM`, dann ist das erste Checkpoint-Intervall $I(c^1_{a_m})$ der Hauptkomponente `SYSTEM` durch den Initialzustand $c^0_{a_m}$ und den ersten Checkpoint $c^1_{a_m}$ begrenzt.

Ein Checkpoint-Intervall ist durch zwei aufeinander folgende Checkpoints begrenzt. Bei der Akteurerzeugung und -terminierung ist das erste und letzte Checkpoint-Intervall durch den einschließenden Checkpoint des Erzeugers begrenzt. In Abbildung 5.6 ist beispielsweise der Checkpoint $c^i_{a_1}$ sowohl der Anfang des Checkpoint-Intervalls $I(c^{i+1}_{a_1})$ als auch der Anfang des Checkpoint-Intervalls $I(c^1_{a_2})$. Analog begrenzen $c^{j-1}_{a_1}$ und $c^j_{a_1}$ das Checkpoint-Intervall $I(c^j_{a_1})$ sowie $c^k_{a_2}$ und $c^j_{a_1}$ das Checkpoint-Intervall $I(c^k_{a_2}, c^j_{a_1})$, das zur Eindeutigkeit mit beiden Gren-

zen bezeichnet wird. Zur Vereinfachung sei im Folgenden alles über Checkpoint-Intervalle Ausgeführte ebenfalls für diese durch zwei Checkpointidentifikatoren bezeichneten Intervalle gültig, ausser diese werden explizit unterschieden.

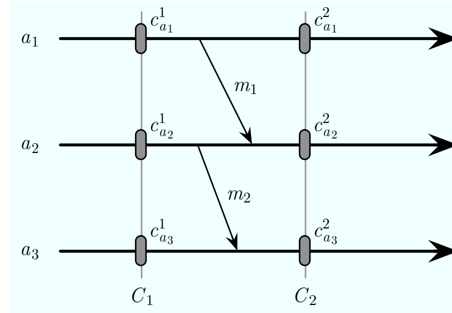


Abbildung 5.7.: Konsistente Zuordnung von Checkpoints

Für die Gewährleistung der Konsistenz ist entscheidend, welche kausalen Abhängigkeiten während eines Checkpoint-Intervalls durch Kommunikation auftreten. Im Fall eines Rollbacks sind nicht nur transitive kausale Abhängigkeiten (vgl. Definition 4.3), sondern darüber hinaus Pfad-Strukturen, die in Modell-basierenden Ansätzen als zigzag-Paths bezeichnet werden, relevant. In Abbildung 5.7 sind drei Akteure mit jeweils zwei Checkpoints und zwei Nachrichten dargestellt, um das Problem zu erläutern.

Es besteht keine kausale Abhängigkeit zwischen dem ersten Checkpoint von Akteur a_1 und dem zweiten Checkpoint von Akteur a_3, also $c^2_{a_3} \nrightarrow c^1_{a_1} \wedge c^1_{a_1} \nrightarrow c^2_{a_3}$. Dennoch müsste im Fall eines Fehlers von a_1 zwischen Versand von m_1 und der Speicherung von $c^2_{a_1}$ und dem folgenden Rollback des Akteurs auf Checkpoint $c^1_{a_1}$ auch Akteur a_2 auf $c^1_{a_2}$ zurückgesetzt werden. Wegen des Rollbacks von a_2 müsste auch a_3 auf Checkpoint $c^1_{a_3}$ zurückgesetzt werden. Es besteht demnach eine indirekte Abhängigkeit zwischen a_1 und a_3, die die Konsistenz betrifft. Würde jedes Checkpoint-Intervall als zeit-atomare Einheit betrachtet werden, in der also keine Reihenfolge zwischen eingehenden und ausgehenden Nachrichten unterschieden werden kann, so wäre die kausale Abhängigkeit durch Nachrichten zwischen diesen Intervallen gleich der hier vorliegenden Abhängigkeit.

Es ist demnach für die hier relevante Abhängigkeit ausreichend, wenn zwei beliebige Ereignisse e_{a_m} und e_{a_n} zweier Checkpoint-Intervalle zweier unterschiedlicher Akteure $a_m, a_n \in A, a_m \neq a_n$ gemäß der erweiterten Happened-Before-Relation (vgl. Definition 4.6) geordnet sind: $e_{a_m} \overset{ehb}{\Rightarrow} e_{a_n} \vee e_{a_n} \overset{ehb}{\Rightarrow} e_{a_m}$. Besteht diese Abhängigkeit zwischen zwei Intervallen, so werden die Anfangs-Checkpoints der Intervalle als einander *konsistent zugeordnet* bezeichnet. Wie beschrieben können

im MoDiS-Systemmodell die Nachrichten bidirektional betrachtet werden, somit ist auch die Relation symmetrisch:

Definition 5.2 (Konsistente Zuordnung)
Seien zwei Ereignisse $e^k_{a_m}$ *und* $e^l_{a_n}$ *zweier Akteure* $a_m, a_n \in A, a_m \neq a_n$ *in den Checkpoint-Intervallen* $I(c^i_{a_m})$ *und* $I(c^j_{a_n})$ *bzgl. der erweiterten Happened-Before-Relation geordnet. Dann sind die Checkpoints* $c^{i-1}_{a_m}$ *und* $c^{j-1}_{a_n}$ *jeweils einander* konsistent zugeordnete *Checkpoints:* $c^{i-1}_{a_m} \stackrel{k}{\longleftrightarrow} c^{j-1}_{a_n}$.
Gilt $c^i_{a_m} \stackrel{k}{\longleftrightarrow} c^j_{a_n}$ *und* $c^j_{a_n} \stackrel{k}{\longleftrightarrow} c^k_{a_o}$*, so gilt auch* $c^i_{a_m} \stackrel{k}{\longleftrightarrow} c^k_{a_o}$*, mit* $a_m \neq a_n \neq a_o$.

Im Fall eines Rollbacks müssen Akteure berücksichtigt werden, die in der beschriebenen Art eine Abhängigkeit zum fehlerhaften Akteur aufweisen. Die Menge der konsistent zugeordneten Checkpoints wird deshalb im später erläuterten Protokoll iterativ aufgebaut. Im Fall eines Rollbacks werden zu den bereits erfassten und gespeicherten konsistent zugeordneten Checkpoints die transitiv konsistent zugeordneten Checkpoints hinzugefügt. Diese Menge an Checkpoints legt fest, welche Prozesse zu welchen Checkpoints zurückgesetzt werden müssen. Die Vereinigungsmenge aller einem Checkpoint konsistent zugeordneten Checkpoints wird als *Abhängigkeitsmenge* bezeichnet:

Definition 5.3 (Abhängigkeitsmenge)
Die Abhängigkeitsmenge $AM(c^i_{a_m})$ *eines Checkpoints* $c^i_{a_m}$ *eines Akteurs* $a_m \in A$ *ist die Vereinigungsmenge aller Checkpoints, die dem Checkpoint* $c^i_{a_m}$ *konsistent zugeordnet sind:*
$c^j_{a_n} \stackrel{k}{\longleftrightarrow} c^i_{a_m}$ *impliziert* $c^j_{a_n} \in AM(c^i_{a_m})$ *mit* $a_n \in A \setminus \{a_m\}$.

Kommunikationsinduziertes Checkpointing stellt sicher, dass der jüngste gespeicherte Checkpoint eines Akteurs Teil eines konsistenten Schnitts ist und deshalb im Falle eines Fehlers zu genau diesem Checkpoint ein Rollback durchgeführt werden kann. Auf Grund des Einsatzes dieses Checkpoint-Verfahrens als Basis für das in Kapitel 6 erläuterte Fehlertoleranzverfahren ist es notwendig, auch zu älteren Checkpoints, die einen konsistenten Schnitt bilden, ein Rollback durchführen zu können. Dies bedeutet, dass nicht nur die Abhängigkeitsmenge des zuletzt gespeicherten Checkpoints für das Rollback von Bedeutung ist, sondern die Vereinigungsmenge aller Abhängigkeitsmengen der Checkpoints, angefangen bei dem Checkpoint, zu dem zurückgesetzt wird, bis hin zum zuletzt gespeicherten. Die Vereinigung dieser Abhängigkeitsmengen wird als *Rollback-Menge* bezeichnet:

Definition 5.4 (Rollback-Menge)
Sei $c^i_{a_m}$ *ein Checkpoint eines Akteurs* $a_m \in A$*. Die* $c^i_{a_m}$ *zugeordnete* Rollback-Menge $R(c^i_{a_m})$ *ist die Vereinigungsmenge aus der Abhängigkeitsmenge des Check-*

points selbst und den Abhängigkeitsmengen aller Checkpoints des selben Akteurs mit höherem Index:
$R(c^i_{a_m}) = \bigcup_{\forall j \geq i} AM(c^j_{a_m})$.

Die Rollback-Menge entspricht der transitiven Hülle der abhängigen Checkpoints. Die Rollback-Menge enthält von jedem Prozess, der auf Grund von Abhängigkeiten ebenfalls zurückgesetzt werden muss, mindestens einen Checkpoint. Bei einem tatsächlichen Rollback muss jeder Prozess zu dem Checkpoint der Rollback-Menge mit jeweils kleinstem Index zurückgesetzt werden. Existieren in der Rollback-Menge $R(c^i_{a_m})$ eines Akteurs $a_m \in A$ mehrere Checkpoints $c^k_{a_n}, c^{k+1}_{a_n}, \ldots, c^{k+l}_{a_n}$ eines Akteurs a_n, so ist nur der Checkpoint $c^k_{a_n}$ mit dem kleinsten Index zu dem konsistenten Schnitt, der durch $c^i_{a_m}$ festgelegt ist. Wird die Rollback-Menge $R(c^i_{a_m})$ auf diese Art verkleinert, so verbleiben genau die Checkpoints, die zusammen mit $c^i_{a_m}$ einen konsistenten Schnitt ergeben, zu dem das Rollback dann durchgeführt werden kann.

Auf Grund der dynamischen Erzeugung und Auflösung von Akteuren muss zudem berücksichtigt werden, ob ein Akteur, dessen Checkpoints in $R(c^i_{a_m})$ enthalten sind, Teil des konsistenten Schnitts ist, ob also der Systemzustand die Existenz des Akteurs reflektiert. Wurde ein Akteur a_o von einem Akteur a_n in dessen Checkpoint-Intervall $I(c^j_{a_n})$ erzeugt und gilt $c^k_{a_n} \in R(c^i_{a_m})$ mit $k \leq j$, so ist die Existenz von a_o im Schnitt nicht enthalten, der Akteur muss beim Rollback aus dem System entfernt werden. Analoges gilt für den Fall, dass ein Checkpoint eines Akteurs in der Rollback-Menge enthalten ist, der zum Zeitpunkt des Fehlers nicht mehr existent war. Gilt $c^l_{a_o} \in R(c^i_{a_m})$, obwohl zum Zeitpunkt des Fehlers von Akteur a_m der Akteur a_o bereits terminiert und aufgelöst war, so muss bei einem Rollback von a_m zu Checkpoint $c^i_{a_m}$ der Akteur a_o mit seinem Zustand $c^l_{a_o}$ wieder gestartet werden.

Das Checkpoint-Verfahren für das MoDiS-Projekt unterstützt keine Anwendungen, die den gemeinsamen, verteilten Speicher des Systems nutzen. Der gemeinsame, verteilte Speicher stellt eine effiziente Kommunikationsform dar, die die Spezifikation kooperativer Problemlösungen erleichtert. Der gemeinsame, verteilte Speicher macht jedoch auf Grund seiner Abhängigkeiten (vgl. Abschnitt 4.1.1.2) den Einsatz eines effizienten Checkpoint-Verfahrens praktisch unmöglich. Vor jedem Schreib- und Lese-zugriff müsste analog zu Nachrichten überprüft werden, ob auf Grund der Abhängigkeiten gemäß der Ordnung $\succ_{wo}$ und gemäß der Abbildung f_{rm} (vgl. Abschnitt 4.1.1.2) ein erzwungener Checkpoint zu speichern ist. Ein kommunikationsinduziertes Verfahren, das die Abhängigkeiten des gemeinsamen, verteilten Speichers berücksichtigt, würde daher durch

den hohen Overhead der erzwungenen Checkpoints die Performanz des Systems zu sehr beeinträchtigen.

Die Kommunikation über gemeinsamen, verteilten Speicher kann semantisch äquivalent durch den Einsatz von K-Ordern und K-Akteuren realisiert werden, da jedes Objekt des gemeinsamen Speichers als lokales Objekt eines K-Akteurs und jede Zugriffs-Methode des Objekts als K-Order des K-Akteurs realisiert werden kann. Aus diesem Grund scheint es sinnvoll, für das hier vorgestellte Checkpoint-Verfahren die Forderung an die Anwendung zu stellen, den gemeinsamen, verteilten Speicher nicht zu nutzen.

5.2.2 Checkpoint-Protokoll in MoDiS

Jeder Akteur speichert folgende Informationen in geeigneten Datenstrukturen:

- Systemweit eindeutiger Akteur-Identifikator a_m
- Identifikator des letzten gespeicherten Checkpoints $c_{a_m}^{index}$
- Aktuelle Abhängigkeitsmenge: $AM(c_{a_m}^{index})$

Die Zustände der Datenstrukturen werden bei folgenden Ereignissen geändert: Akteurerzeugung, Akteurjoin, K-Order-Aufruf, K-Order-Antwort, freiwilliger Checkpoint und erzwungener Checkpoint (wird bei K-Order-Aufruf bzw. K-Order-Antwort behandelt). Im Folgenden werden die einzelnen Operationen mit den Auswirkungen auf das Verfahren beschrieben. Teile des Verfahrens, wie beispielsweise das Speichern der Abhängigkeitsmengen, dienen nicht der Konsistenz des Schnitts, sondern der Effizienz beim Rollback.

Akteurerzeugung. Wird ein Akteur a_n von einem Akteur a_m während dessen Checkpoint-Intervall $I(c_{a_m}^{i+1})$ erzeugt, so sind folgende Änderungen zu speichern: Der *index* von a_n wird mit dem Wert 1 initialisiert, die Abhängigkeitsmenge $AM(c_{a_m}^{i})$ seines Erzeugers übernommen. Solange der neu erzeugte Akteur a_n keinen eigenen ersten Checkpoint erstellt hat, ist der letzte von seinem Erzeuger gespeicherte Checkpoint $c_{a_m}^{i}$ die jüngste Rollback-Möglichkeit. Daher sind auch die abhängigen Akteure, die bei einem Rollback während dieses Intervalls mit zurückgesetzt werden müssten, die gleichen wie die des erzeugenden Akteurs a_m. Zusätzlich wird beim erzeugten Akteur a_n ein Verweis auf den letzten Checkpoint des Erzeugers a_m gespeichert: a_n.`creator` := $c_{a_m}^{i}$; beim Erzeuger wird

dem bereits gespeicherten Checkpoint ein Verweis auf den neu erzeugten Akteur vermerkt: $c^i_{a_m}.\texttt{createe} := c^i_{a_m}.\texttt{createe} \cup \{a_n\}$.

Akteurjoin. Im Protokoll ist festgelegt, dass ein Akteur $a_m \in A$ vor seiner Terminierung einen letzten Checkpoint setzt. Dieser Checkpoint mit Index $c^i_{a_m}$ ist die einzige Abhängigkeit, die von dem einfangenden Vater-Akteur a_n übernommen werden muss, da der terminierte Akteur a_m nach seiner Terminierung keine zusätzlichen Abhängigkeiten erhalten kann. Der Checkpoint $c^i_{a_m}$ ist durch die Join-Abhängigkeit dem letzten Checkpoint des Vaters konsistent zugeordnet. Die Abhängigkeitsmenge des Vater-Akteurs wird daher um diesen Checkpoint erweitert: $AM(c^j_{a_n}) := AM(c^j_{a_n}) \cup \{c^i_{a_m}\}$. Auch nach Terminierung und Join eines Akteurs müssen die von ihm gesetzten Checkpoints mit ihren Verwaltungsinformationen weiter auf persistentem Speicher gehalten werden, da auch weiterhin Rollbacks zu konsistenten Schnitten möglich sind, die einen dieser Checkpoints enthalten.

Freiwilliger Checkpoint. Wird von einem Akteur der vom Übersetzer generierte Funktionsaufruf ins Laufzeit-Management zum Speichern eines freiwilligen Checkpoints aufgerufen, so überprüft das Laufzeit-Management die Länge des aktuellen Checkpoint-Intervalls. Ist dieses Checkpoint-Intervall kleiner als das im System eingestellte Idealintervall, so wird kein Checkpoint gespeichert, die Funktion springt zurück zur Ausführung des Akteurs. Ist dagegen das Intervall gleich oder größer als das Idealintervall, so wird ein freiwilliger Checkpoint $c^i_{a_m}$ gespeichert. Neben dem Zustand des Akteurs wird die aktuelle, nun vollständige Abhängigkeitsmenge $AM(c^{i-1}_{a_m})$ des letzten Checkpoints gespeichert. Anschließend wird der lokale Checkpointindex inkrementiert und die neue Abhängigkeitsmenge leer initialisiert: $AM(c^i_{a_m}) = \{\}$.

K-Order-Aufruf. Ruft ein Akteur $a_m \in A$ eine K-Order eines K-Akteurs a_k auf, so wird bereits vor der Methode zum Aufruf der Akteur a_m in copy-on-write-Modus gesetzt, sodass zum Zeitpunkt des Empfangs der Antwortnachricht das Speichern eines Checkpoints des Zustands vor dem Aufruf möglich ist. Der Aufruf wird generiert und folgende Informationen zusammen mit dem Aufruf an den K-Akteur a_k übertragen: Der Checkpointidentifikator des zuletzt gesicherten Checkpoints $c^i_{a_m}$ sowie die aktuelle Abhängigkeitsmenge $AM(c^i_{a_m})$.

Trifft die Nachricht beim aufgerufenen K-Akteur a_k ein, so wird zunächst überprüft, ob in seiner aktuellen Abhängigkeitsmenge $AM(c^j_{a_k})$ ein Checkpointidentifkator des Aufrufers a_m vermerkt ist. Es werden zwei Fälle unterschieden:

1. Ist *kein* Checkpointidentifikator des Aufrufers a_m oder aber der mitgesendete Checkpointidentifikator $c^i_{a_m}$ in $AM(c^j_{a_k})$ enthalten, so wird die Vereinigungsmenge aus der eigenen Abhängigkeitsmenge, der mitgesendeten Abhängigkeitsmenge und dem mitgesendeten Checkpointidentifikator gebildet:
   $AM(c^j_{a_k}) := AM(c^j_{a_k}) \cup AM(c^i_{a_m}) \cup \{c^i_{a_m}\}$.
   Es muss kein Checkpoint gesetzt werden, der K-Order-Aufruf kann bearbeitet werden.

2. Ist ein *älterer* Checkpointidentifikator $c^h_{a_m}, h < i$ des aufrufenden Akteurs a_m als der mitgesendete Checkpointidentifikator $c^i_{a_m}$ in der Abhängigkeitsmenge des aufgerufenen K-Akteurs a_k vorhanden, so muss ein erzwungener Checkpoint von a_k gespeichert werden. Der Zustand von a_k wird als Checkpoint $c^{j+1}_{a_k}$ zusammen mit der Abhängigkeitsmenge $AM(c^j_{a_k})$ gespeichert. Anschließend wird der lokale Checkpointzähler inkrementiert und als neue Abhängigkeitsmenge die Vereinigungsmenge aus der mitgesendeten Abhängigkeitsmenge und dem mitgesendeten Checkpointidentifikator gebildet:
   $AM(c^{j+1}_{a_k}) := AM(c^i_{a_m}) \cup \{c^i_{a_m}\}$.
   Im Anschluss kann der K-Order-Aufruf bearbeitet werden.

K-Order-Antwort. Nach Abarbeitung einer K-Order durch K-Akteur a_k wird das Ergebnis zum Versenden an den Aufrufer a_m vorbereitet. Die Nachricht wird wiederum erweitert um den Checkpointidentifikator $c^j_{a_k}$ des zuletzt gesicherten Checkpoints sowie die Abhängigkeitsmenge $AM(c^j_{a_k})$ des aktuellen Intervalls und an den Aufrufer a_m übertragen.

Bevor der Aufrufer a_m aus seiner Blockierung gelöst wird, um das K-Order-Ergebnis zu verarbeiten, werden analog zum K-Order-Aufruf die mit der Antwort übertragenen Informationen mit den lokalen Informationen der Abhängigkeitsmenge $AM(c^i_{a_m})$ verglichen und die äquivalenten zwei Fälle unterschieden:

1. Ist kein Checkpointidentifikator des K-Akteurs a_k oder aber der mitgesendete Checkpointidentifikator $c^j_{a_k}$ in $AM(c^i_{a_m})$ enthalten, so wird die Vereinigungsmenge aus der eigenen Abhängigkeitsmenge, der mitgesendeten Abhängigkeitsmenge und dem mitgesendeten Checkpointidentifikator gebildet:
   $AM(c^i_{a_m}) := AM(c^i_{a_m}) \cup AM(c^j_{a_k}) \cup \{c^j_{a_k}\}$.
   Es muss kein Checkpoint gespeichert werden, der copy-on-write-Modus kann verlassen werden. Die in diesem Modus kopierten Seiten werden verworfen und der Akteur entblockiert, um die Verarbeitung des K-Order-Ergebnisses zu beginnen.

2. Ist ein älterer Checkpointidentifikator $c^h_{a_k}, h < j$ des K-Akteurs a_k, dessen Index h kleiner als der des mitgesendete Checkpointidentifikators $c^j_{a_k}$ ist, in der Abhängigkeitsmenge $AM(c^i_{a_m})$ des Akteurs a_m vorhanden, so muss ein erzwungener Checkpoint von a_m gespeichert werden. Der erzwungene Checkpoint muss von dem Zustand des Akteurs a_m zum Zeitpunkt vor dem K-Order-Aufruf gespeichert werden, damit die Checkpoints einen konsistenten Schnitt bilden. Dies ist möglich, da Akteur a_m seit dem Aufruf blockiert ist und die für den Aufruf veränderten Speicherseiten durch copy-on-write kopiert wurden. Der Zustand von a_m wird als Checkpoint $c^{i+1}_{a_m}$ gespeichert und zu dem Zustand die Abhängigkeitsmenge $AM(c^i_{a_m})$ mit gespeichert. Anschließend wird der lokale Checkpointzähler inkrementiert. Als neue Abhängigkeitsmenge wird die Vereinigungsmenge aus der mitgesendeten Abhängigkeitsmenge und dem mitgesendeten Checkpointidentifikator gebildet:
   $AM(c^{i+1}_{a_m}) := AM(c^j_{a_k}) \cup \{c^j_{a_k}\}$.
   Im Anschluss kann der Akteur entblockiert werden und mit der Verarbeitung des K-Order-Ergebnisses beginnen.

Das vorgestellte Protokoll stellt sicher, dass die Checkpoints einen konsistenten Schnitt ergeben. Würden Abhängigkeiten Akteurerzeugung, Terminierung, oder K-Order-Nachrichten die Konsistenz verletzen, so wird ein erzwungener Checkpoint gespeichert.

Die Abhängigkeiten zwischen Akteuren, die sich durch deren Kommunikationsbeziehungen oder Vater-Kind-Beziehungen ergeben, sind Grundlage für die Bestimmung des konsistenten Schnitts bei einem Rollback. Um diesen konsistenten Schnitt effizient bestimmen zu können, werden bei Ausführung des in diesem Abschnitt beschriebenen Checkpointprotokolls die Abhängigkeitsmengen gebildet und gespeichert. Nur die in der Rollback-Menge durch enthaltene Checkpointidentifikatoren identifizierten Akteure sind von einem Rollback betroffen. Die Abhängigkeitsmengen verschiedener Checkpoint-Intervalle werden im Fehlerfall benötigt, um die einem Checkpoint zugeordnete Rollback-Menge zu bestimmen, weshalb die Abhängigkeitsmengen bei jeder Änderung bei den oben beschriebenen Operationen gespeichert werden. Im Fehlerfall werden die Mengen des jeweils aktuellen Checkpoint-Intervalls zusammen mit Mengen schon abgeschlossener Intervalle genutzt, um den konsistenten Schnitt zu berechnen.

Der folgende Abschnitt beschreibt, wie mit Hilfe der gespeicherten Abhängigkeitsmengen im Fehlerfall die Rollback-Menge ermittelt werden kann. Das Verfahren ermittelt zugleich, welche Checkpoints dieser Gesamtmenge an zurückzusetzenden Akteuren einen konsistenten Schnitt ergeben.

5.2.3 Rollback

Erkennt das Laufzeit-Management einen Fehlerzustand (vgl. Kapitel 4) eines Akteurs, so muss dieser Akteur und dessen abhängige Akteure auf einen gültigen Zustand, also auf gespeicherte Checkpoints, die zusammen einen konsistenten Schnitt bilden, zurückgesetzt werden.

Es gibt in der Regel eine Vielzahl konsistenter Schnitte, zu denen zurückgesetzt werden kann. An dieser Stelle wird angenommen, dass zum jüngsten konsistenten Schnitt, der also den geringsten Verlust an bereits ausgeführter Berechnung bedeutet, zurückgesetzt werden soll. Andere Strategien zur Auswahl eines geeigneten Schnitts für ein Rollback werden in Kapitel 6 erläutert. Der Rollback wird synchron durchgeführt. Sobald der Fehler eines Akteurs erkannt wurde, werden alle Akteure des Systems angehalten.

Bevor der konsistente Schnitt berechnet werden kann, müssen noch aktuell im Umlauf befindliche Nachrichten berücksichtigt werden. Der Kommunikator des Managements wird daraufhin überprüft, ob zu diesem Zeitpunkt Anwendungsnachrichten in Umlauf sind. Analog zum Nachrichtenempfang der normalen Ausführung werden wie beschrieben die Abhängigkeitsmengen so erweitert, als wären die Nachrichten bereits empfangen worden. Nach Abarbeitung des Rollbacks werden diese Änderungen wieder entfernt. Sind somit diese Abhängigkeiten ebenfalls berücksichtigt, so kann die Berechnung des konsistenten Schnitts erfolgen.

An dieser Stelle sei angenommen, dass der Checkpoint mit Checkpointidentifikator $c^i_{a_m}$ eines fehlerbehafteten Akteurs $a_m \in A$ als letzter Checkpoint gespeichert wurde. Alle Akteure, die seit diesem Checkpoint eine direkte oder transitive Abhängigkeit zu a_m aufgebaut haben, müssen ebenfalls zurückgesetzt werden. Der Systemzustand, der diese Abhängigkeiten nicht enthalten hat, ist der $c^i_{a_m}$ zum Fehlerzeitpunkt zugeordnete konsistente Schnitt. Dieser lässt sich wie folgt iterativ aus den gespeicherten Abhängigkeitsmengen bestimmen:

Die Abhängigkeitsmengen aller auf $c^i_{a_m}$ folgenden Checkpoint-Intervalle werden vereinigt. In diesem Fall ist der Checkpoint $c^i_{a_m}$ der zuletzt gesicherte Checkpoint, somit ist nur die aktuelle Abhängigkeitsmenge $AM(c^i_{a_m})$ zu berücksichtigen und bildet die initiale Rollback-Menge $R(c^i_{a_m})$.

Für jeden Checkpointidentifikator $c^j_{a_n}$ in dieser Rollback-Menge $R(c^i_{a_m})$ werden nun iterativ wiederum die Abhängigkeitsmengen vom identifizierten Checkpoint bis zum aktuellen Zeitpunkt $AM(c^j_{a_n}), \ldots, AM(c^{j+k}_{a_n})$ vereinigt und zur Rollback-Menge $R(c^i_{a_m})$ hinzugefügt. In diesen Fällen kann es durchaus sein, dass der entsprechende Akteur a_n nach dem indizierten Checkpoint $c^j_{a_n}$ noch weitere Check-

points $c_{a_n}^{j+1}, \dots, c_{a_n}^{j+k}$ gespeichert hat. Dieses Verfahren wird rekursiv wiederholt, bis die Rollback-Menge sich nicht weiter vergrößert und die gesamte transitive Hülle der Abhängigkeiten erfasst ist.

Aus dieser Rollback-Menge lässt sich nun in einem zweiten Schritt mit Hilfe der Abhängigkeiten und der aufsteigenden Checkpoint-Indizierung der konsistente Schnitt berechnen, der durch $c_{a_m}^{i}$ indiziert wird. Zur Steigerung der Effizienz ließen sich beide Schritte kombinieren und bereits beim erweitern der Menge jeweils nur die Checkpointidentifikatoren mit kleinstem Index zur Menge hinzunehmen. Unabhängig davon müssen auf Basis der Vater-Kind-Beziehungen der Akteure wie beschrieben einerseits zusätzliche Abhängigkeiten berücksichtigt werden und andererseits Akteure zum Entfernen markiert werden, falls diese zum Zeitpunkt des Schnittes noch nicht existiert haben. Dies geschieht mit Hilfe der gespeicherten `createe`-Verweise.

In Algorithmus 5.1 ist der Ablauf zur Übersichtlichkeit ohne mögliche Optimierungen dargestellt. Ausgehend vom Checkpointidentifikator $c_{a_m}^{i}$ wird der konsistente Schnitt und zugleich die Menge zu entfernender Akteure berechnet.

Algorithmus 5.1 (konsistenter Schnitt)
Gegeben:

- *Der Checkpoint des fehlerhaften Akteurs* $c_{a_m}^{i}$;
- *Alle laut Protokoll gespeicherten Abhängigkeitsmengen und Informationen;*
- *Eine Hilfsmenge* `Erledigt` *zum Speichern schon abgearbeiteter Checkpointidentifikatoren;*

Gesucht:

- *Die Menge* `Schnitt`*, die die Checkpointidentifikatoren des durch* $c_{a_m}^{i}$ *festgelegten konsistenten Schnitts enthält.*
- *Die Menge* `Entfernen`*, die alle Akteuridentifikatoren enthält, die bei einem Rollback zu* `Schnitt` *aus dem System entfernt werden müssen, da sie zum Zeitpunkt der Checkpoints noch nicht erzeugt waren.*

Algorithmus:

Hilfsfunktion* `genAbh`*:

- `genAbh`($c^i_{a_m}$: *Checkpointidentifikator):*
  - *Wurde der Checkpoint schon bearbeitet?*
    `if` ($c^i_{a_m} \in$ `Erledigt`) `then return`;
  - *Abhängigkeitsmenge erweitern:*
    `Erledigt` $:=$ `Erledigt` $\cup \{c^i_{a_m}\}$;
    `Schnitt` $:=$ `Schnitt` $\cup AM(c^i_{a_m})$;
  - *Rekursiver Aufruf für transitive Abhängigkeit:*
    $\forall c^j_{a_n} \in AM(c^i_{a_m})$: `genAbh`($c^j_{a_n}$);
  - *Falls jüngere Checkpoints existieren, müssen auch deren Abhängigkeiten (rekursiv) erfasst werden:*
    `if` ($\exists c^{i+1}_{a_m}$) `then genAbh`($c^{i+1}_{a_m}$);
  - *Falls ein anderer Akteur erzeugt wurde, so ist dieser im Schnitt später nicht enthalten, seine Abhängigkeiten sind jedoch ebenfalls zu berücksichtigen:*
    $\forall c^1_{a_n} \in c^i_{a_m}$`.createe` :
    * `Entfernen` $:=$ `Entfernen` $\cup \{a_n\}$;
    * `genAbh`($c^1_{a_n}$);
  - *Und die Funktion verlassen:*
    `return`;

Initialisierungsphase:

- *Die Mengen werden mit den Anfangsabhängigkeiten initialisiert:*
  `Schnitt` $:= AM(c^i_{a_m}) \cup \{c^i_{a_m}\}$;
  `Erledigt` $:= \{c^i_{a_m}\}$;
  `Entfernen` $:= \{\}$;

Berechnung der Mengen:

- *Die transitive Hülle der Rollbackmenge wird rekursiv berechnet:*
  $\forall c^j_{a_n} \in AM(c^i_{a_m})$: `genAbh`($c^j_{a_n}$);

- *Falls mehrere Checkpointidentifikatoren eines Akteurs in der Menge sind, so werden alle bis auf den ältesten entfernt:*
  $\forall c^i_{a_m} \in \texttt{Schnitt}:$
  $\texttt{if}\ (\exists c^{i-k}_{a_m} \in \texttt{Schnitt})\ \texttt{then}\ \texttt{Schnitt} := \texttt{Schnitt} \setminus \{c^i_{a_m}\};$
- *Falls Checkpointidentifikatoren von zu entfernenden Akteuren in der Menge* `Schnitt` *enthalten sind, werden sie entfernt:*
  $\forall a_n \in \texttt{Entfernen}:$
  $\texttt{if}\ (\exists c^i_{a_n} \in \texttt{Schnitt})\ \texttt{then}\ \texttt{Schnitt} := \texttt{Schnitt} \setminus \{c^i_{a_n}\};$

Ergebnis:
Die Menge `Schnitt` *enthält die Checkpoints, zu denen zurückgesetzt werden muss. Die Akteure der Menge* `Entfernen` *existierten in dem konsistenten Schnitt noch nicht, sie sind deshalb zu entfernen.*

Die Akteure, deren Checkpoints in dem berechneten konsistenten Schnitt enthalten sind, werden auf die gespeicherten Zustände zurückgesetzt. Erst wenn alle Zustände der enthaltenen Akteure zurückgesetzt sind, können die Akteure des gesamten Systems wieder in den Zustand rechenbereit gesetzt werden und das System weiterlaufen. Somit können Inkonsistenzen zwischen abhängigen Akteuren sowie durch neue Abhängigkeiten während der Recovery-Line-Bestimmung vermieden werden. Entscheidend für die Performanz des Verfahrens ist in erster Linie der Aufwand während des fehlerfreien Betriebs, nicht der Aufwand für einen Rollback zu einem konsistenten Schnitt. In Systemen mit sehr vielen Akteuren, in denen die Häufigkeit eines fehlerhaften Akteurs somit zunimmt, müsste für einen effizienteren Rollback-Mechanismus ein komplexeres Protokoll eingesetzt werden, das den Berechnungsfortschritt nicht betroffener Prozesse während des Rollbacks ermöglicht, also auch im Fehlerfall blockierungsfrei arbeitet.

5.2.4 Verwerfen der Checkpoints

In kommunikationsinduzierten Checkpoint-Verfahren können Checkpoints wieder verworfen werden, falls immer zum jüngsten konsistenten Schnitt zurückgesetzt werden soll. Welche Checkpoints im jüngsten konsistenten Schnitt enthalten sind, hängt davon ab, in welchem Akteur ein Fehler aufgetreten ist. Für jeden Akteur $a_m \in A$ legt sein jüngster Checkpoint $c^i_{a_m}$ den konsistenten Schnitt fest, zu dem ein Rollback stattfindet, falls in der Ausführung von a_m ein Fehler auftritt. Die Checkpoints anderer Akteure, die zusammen mit $c^i_{a_m}$ einen konsistenten Schnitt ergeben, sind auf Basis der Abhängigkeitsmengen bestimmbar, wie im vorhergehenden Abschnitt 5.2.3 beschrieben.

Eine Garbage Collection dürfte jeden Checkpoint verwerfen, für den folgendes gilt: Der Checkpoint ist nicht Teil eines konsistenten Schnitts, der durch den jüngsten Checkpoint eines beliebigen Akteurs festgelegt wird. Um so eine Garbage Collection effizient zu realisieren, wäre eine Erweiterung des Protokolls notwendig, sodass die Garbage Collection die Informationen über gespeicherte Abhängigkeitsmengen vergleicht und daraus ermittelt, für welche Checkpoints diese Bedingung erfüllt ist.

In der vorliegenden Arbeit ist keine Garbage Collection erarbeitet, die dieses Verhalten realisiert. Der Ansatz zur Fehlertoleranz, der auf dem vorgestellten Checkpoint-Rollback-Verfahren basiert, soll explizit auch Rollbacks zu älteren konsistenten Schnitten durchführen können. Dies wird in Abschnitt 6.2.5.1 erklärt. Aus diesem Grund wird zunächst davon ausgegangen, dass alle Checkpoints weiter gespeichert bleiben, um beliebig Rollbacks zu älteren konsistenten Schnitten durchführen zu können.

5.2.5 Anwendungsangepasste Checkpoint-Platzierung

Das bisher beschriebene Verfahren für kommunikationsinduziertes Checkpointing in MoDiS ist den in Abschnitt 5.1.4 beschriebenen Performanzproblemen durch zu kurze Checkpoint-Intervalle auf Grund erzwungener Checkpoints unterlegen. Die Intervalle werden bei kommunikationsinduziertem Checkpointing durch entstehende Abhängigkeiten, also durch das Kommunikationsverhalten der Anwendung beeinflusst. In den folgenden Abschnitten wird erläutert, wie dies zur Übersetzungszeit analysiert werden kann, um die Platzierung freiwilliger Checkpoints vorzubereiten, sodass ungünstige Intervalle erzwungener Checkpoints vermieden werden.

Das Ziel ist dabei, den zusätzlichen Overhead kurzer Intervalle gegen den Overhead längerer Intervalle abzuwägen. Die Folge ist, dass in Berechnungsphasen mit viel Kommunikation die Intervalle der freiwilligen Checkpoints größer gewählt werden, sodass kurze Intervalle durch erzwungene Checkpoints vermieden werden können. In den folgenden Abschnitten wird dieses Vorgehen und die konkrete Bewertungs-Metrik in Teilprobleme zerlegt und iterativ erklärt. Die einzelnen Abschnitte sind wie folgt gegliedert:

- Das generelle Vorgehen zur Bewertung einzelner Positionen für freiwillige Checkpoints in Form eines Greedy-Verfahrens wird erklärt.
- Es folgt die statische Vorhersage des Akteur-Verhaltens im Übersetzer, auf der die Bewertung basiert.

- Die Bewertung für den Fall, dass keine zusätzlichen Abhängigkeiten bestehen, wird erläutert.
- Dann werden zusätzlich mögliche erzwungene Checkpoints durch Kommunikationsbeziehungen in der Bewertung berücksichtigt.
- Die Positionierung von Checkpoints bei Akteurerzeugung und -terminierung wird als nächstes integriert.
- Die Abbruchbedingung des Greedy-Verfahrens, die sich direkt aus der Bewertungsfunktion ableiten lässt, wird erläutert.
- Mit Hilfe der erklärten Bestandteile wird die vollständige Bewertungsfunktion zusammengesetzt.
- Abschließend wird auf die Beachtung von nicht vorhersagbarem Verhalten durch bedingte Anweisungen und Schleifen eingegangen.

5.2.5.1 Greedy-Verfahren

Statisch zur Übersetzungszeit vorhandene Informationen werden genutzt, um potentielle freiwillige Checkpoints vorzubereiten. Die Befehlsfolgen der Akteure dienen dabei als Abschätzungsgrundlage der Intervalle, die Funktionsaufrufe zum Speichern eines freiwilligen Checkpoints können also vom Übersetzer direkt in den Code der Akteure eingebaut werden. Diese Aufrufe werden als *potentielle Checkpoints* bezeichnet, da erst zur Laufzeit entschieden wird, ob tatsächlich ein Checkpoint gespeichert wird. Die Grundlage dieser Entscheidung ist, ob in der vergangenen Ausführung eines Akteurs der potentielle Checkpoint tatsächlich ein sinnvolles Intervall ergibt oder aber der Aufruf übersprungen wird.

Die Vorbereitung dieser Checkpoints wird in einem Greedy-Verfahren realisiert. Für jeden Akteur wird seine Ereignisspur analysiert und auf Basis der statisch vorhandenen Informationen abgeschätzt, an welchen Stellen im Code ein freiwilliger Checkpoint den jeweils geringsten Gesamt-Overhead erzeugt. Ist eine erste Position mit lokalem Optimum gefunden, so wird der Code für den Checkpoint-Aufruf integriert und von dieser Stelle aus die nächste Position für einen potentiellen Checkpoint gesucht. Das Greedy-Verfahren arbeitet derart einen Akteur nach dem anderen ab, bis in jedem Akteur alle potentiellen Checkpoints vorbereitet sind. Die Aufrufe für freiwillige Checkpoints sind damit statisch festgelegt.

Die Grundlage zur Entscheidung über geeignete Platzierungen dieser potentiellen Checkpoints ist eine Minimierung des Overheads. In Abschnitt 5.1.4 wurde erläutert, dass der Gesamt-Overhead des Verfahrens von den tatsächlichen

Checkpoint-Intervallen abhängt, wie in Abbildung 5.2 dargestellt. In den folgenden Abschnitten wird Schritt für Schritt erarbeitet, welche Positionen im Code bzw. in der Ereignisfolge eines Akteurs bewertet werden müssen und wie sich die Bewertungsfunktion des Gesamt-Overheads zusammensetzt. Im einzelnen müssen neben den Checkpoint-Intervallen auch die K-Order-Aufrufe sowie die Abhängigkeiten durch Akteurerzeugung und -terminierung berücksichtigt werden.

5.2.5.2 Vorhersage der Systementwicklung

Die Vorbereitung der Checkpoint-Platzierung basiert auf einer Vorhersage der Systemausführung. Zur Übersetzungszeit ist die Systemausführung nicht vollständig, aber in Teilen vorhersagbar. Für die hier vorgestellte Metrik wird eine Vorhersage über potentielle Kommunikationspartner, geschätzten Berechnungsaufwand zwischen Kommunikationsereignissen sowie potentielle Komponentenerzeugung genutzt. Die vorhersagbaren Informationen stellen eine relevante Verbesserung der Performanz durch statische Checkpoint-Platzierung im Vergleich zu herkömmlichen, zur Laufzeit mittels Protokoll gesteuerten, kommunikationsinduzierten Verfahren dar.

Die Vorhersage der Systementwicklung basiert auf der Generierung von *Attributierten Potentiellen Nebenläufigkeitsverbänden (APNV)*, wie in Abschnitt 2.3.1 und ausführlich in [Reh04] beschrieben. Trotz des nicht vorhersagbaren, nichtdeterministischen Verhaltens in Schleifen und bedingten Anweisungen lassen sich zur Übersetzungszeit folgende Informationen prognostizieren und im APNV darstellen:

1. Für jeden Akteur des Systems ist festgelegt:
   - Mit welchen K-Akteuren er durch einen K-Order-Aufruf kommuniziert;
   - Die Reihenfolge dieser `corder_callsend` Aufrufe;
   - Die Reihenfolge von Akteur-Erzeugungs-Ereignissen (`create`);
   - Die Reihenfolge von Akteur-Auflösungs-Ereignissen (`join`);
   - Die relative Berechnungszeit zwischen diesen Ereignissen;
2. Für jeden K-Akteur des Systems ist festgelegt:
   - Welche Akteure mit ihm per K-Order-Aufurf kommunizieren können, die Reihenfolge jedoch nicht;
   - Die relative Berechnungszeit zwischen K-Order-Annahmen;

Insgesamt steht demnach zur Übersetzungszeit genug Information zur Verfügung, die Abhängigkeitsbeziehungen und den Rechenaufwand als Grundlage zur Checkpoint-Platzierung zu verwenden. Bedingte Anweisungen und Schleifen führen dazu, dass einige der genannten Punkte nur beschränkt vorherbestimmt werden können. Zur Laufzeit werden deshalb die Entscheidungen überprüft und je nach tatsächlichem Systemverhalten angepasst.

5.2.5.3 Positions-Bewertung ohne Abhängigkeiten

Aus der Overhead-Berechnung geht hervor, dass bei einem Akteur ohne Abhängigkeiten die Abstände zwischen den potentiellen Checkpoints jeweils das Idealintervall betragen sollten. Ein Akteur, dessen kanonische Funktion keine Kommunikations- oder Akteur-Erzeugungs-Ereignisse enthält, kann daher jeweils nach dem prognostizierten Berechnungsaufwand des Idealintervalls I_{ideal} den Funktionsaufruf für einen potentiellen Checkpoint erhalten, wie in Abbildung 5.8 dargestellt. Die dunkel eingezeichneten abgerundeten Rechtecke entsprechen bereits gesetzten potentiellen Checkpoints (pc), nicht eingefärbt ist die nächste interessante Position p_1, die bewertet wird.

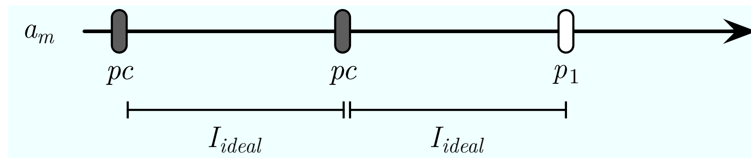


Abbildung 5.8.: Potentielle Checkpoints eines Akteurs ohne Abhängigkeiten

Sind keine Abhängigkeiten vorhanden, so wird von dem Greedy-Verfahren jeweils die Position nach dem Idealintervall gewählt und der potentielle Checkpoint gesetzt, d.h. der Aufruf an die Laufzeitumgebung in den Code des Akteurs integriert. Der erste Bestandteil der Bewertungsfunktion für Positionen ist demnach die Overhead-Ratio $O(I)$ (vgl. 5.1.4).

Das Idealintervall lässt sich wie in Abschnitt 5.1.4 erklärt für prognostizierte Fehlerwahrscheinlichkeit und geschätzte Kosten für lokale Checkpoints berechnen und wird auf Basis von INSEL-Instruktionen approximiert. Diese Abschätzung kann beliebig aufwändiger gestaltet werden, indem praxisrelevante Beispiel-Systeme mit verschiedenen Werten des Intervalls simuliert und der Overhead verglichen wird. Es ist jedoch für den Erfolg des Verfahrens nicht entscheidend, das Idealintervall exakt zu kennen, es ist ausreichend, in der Nähe des Overhead-Minimums zu liegen.

5.2.5.4 Einbezug von Kommunikation

Erzwungene Checkpoints können das resultierende Intervall beliebig verkürzen. Kommunikationsereignisse müssen demnach in die Platzierungsentscheidung potentieller Checkpoints miteinfließen. Es wird bei der Platzierung der potentiellen Checkpoints beachtet, ob durch den freiwilligen Checkpoint eventuell ein Checkpoint bei einem anderen Akteur erzwungen werden kann. Ist dies der Fall, so wird abgeschätzt, wie hoch der zusätzliche Overhead auf Grund des Intervalls dieses erzwungenen Checkpoints werden kann und mit einer alternativen Platzierung des freiwilligen Checkpoints verglichen. Es muss also zunächst untersucht werden, unter welchen Bedingungen ein Checkpoint bei einem Akteur erzwungen werden kann.

Bei jedem K-Order-Aufruf eines Akteurs $a_m \in A$ an einen K-Akteur $a_k \in A_k$ kann potentiell ein Checkpoint bei einem der beiden Akteure erzwungen werden. Tatsächlich passiert dies zur Laufzeit, falls zum Zeitpunkt der Kommunikation folgende notwendige und hinreichende Bedingung erfüllt ist (vgl. Abschnitt 5.2.2):

In der aktuellen Abhängigkeitsmenge eines von zwei miteinander kommunizierenden Akteure $a_m, a_n \in A$ ist ein älterer Checkpointidentifikator als der jüngste Identifikator des anderen Akteurs enthalten:
$\{c^x_{a_n}\} \in AM(c^j_{a_m})$ mit $c^{x+i}_{a_n}$ jüngster Checkpoint von a_n.
Dies impliziert auf Grund des angegebenen Protokolls, dass zugleich in der aktuellen Abhängigkeitsmenge des anderen Akteurs kein Checkpointidentifikator des Kommunikationspartners enthalten ist:
$\{c^y_{a_m}\} \notin AM(c^{x+i}_{a_n})$.
Diese Bedingung für die Möglichkeit eines Akteurs, einen Checkpoint zu erzwingen, wird als *Zwang-Bedingung* bezeichnet.

Zunächst werden die Fälle betrachtet, in denen kein Checkpoint erzwungen werden kann. Dabei bleiben zwei mögliche Fälle zu unterscheiden: entweder enthält die aktuelle Abhängigkeitsmenge keinen Checkpointidentifikator des Kommunikationspartners oder sie enthält den jüngsten Checkpointidentifikator. Der erste dieser beiden Fälle kann nur entstehen, wenn seit dem jeweils letzten Checkpoint beider Akteure keine direkte oder transitive Abhängigkeit zwischen den Akteuren entstanden ist. Unter diesen Umständen sind die beiden letzten Checkpoints der Akteure Teil eines konsistenten Schnitts, es muss kein Checkpoint erzwungen werden. Der zweite Fall kann nur entstehen, falls seit dem Speichern des jüngsten Checkpoints des Kommunikationspartners bereits direkt oder transitiv Informationen von ihm empfangen wurden.

Die beschriebene Konstellation, in der ein Checkpoint bei einem anderen Akteur erzwungen werden kann, wenn also die Zwang-Bedingung erfüllt ist, entsteht nach folgendem Schema: Zwei Akteure kommunizieren direkt oder transitiv und tauschen nach dem beschriebenen Verfahren (vgl. Abschnitt 5.2.2) Checkpointidentifikatoren aus, um Abhängigkeiten zu repräsentieren. Speichert nun einer der beiden Akteure einen oder mehrere weitere Checkpoints, der andere jedoch nicht, so ist die Zwang-Bedingung erfüllt: die Abhängigkeitsmenge des Akteurs, der keinen weiteren Checkpoint gespeichert hat, enthält einen älteren Checkpointidentifikator des anderen Akteurs.

Ausgehend von dieser Bedingung wird nun erklärt, mit welcher Vorgehensweise im Übersetzer auf Basis der statischen Informationen eine geeignete Platzierung gefunden werden kann, die den Overhead durch erzwungene Checkpoints berücksichtigt. Das Verfahren überprüft, in welchen Abschnitten der Ereignisfolgen gemäß der Zwang-Bedingung Checkpoints erzwungen werden können und wie hoch der entsprechende zusätzliche Overhead beim Kommunikationspartner maximal sein kann. Dieser zusätzliche Overhead wird ebenfalls in der Bewertungsfunktion erfasst.

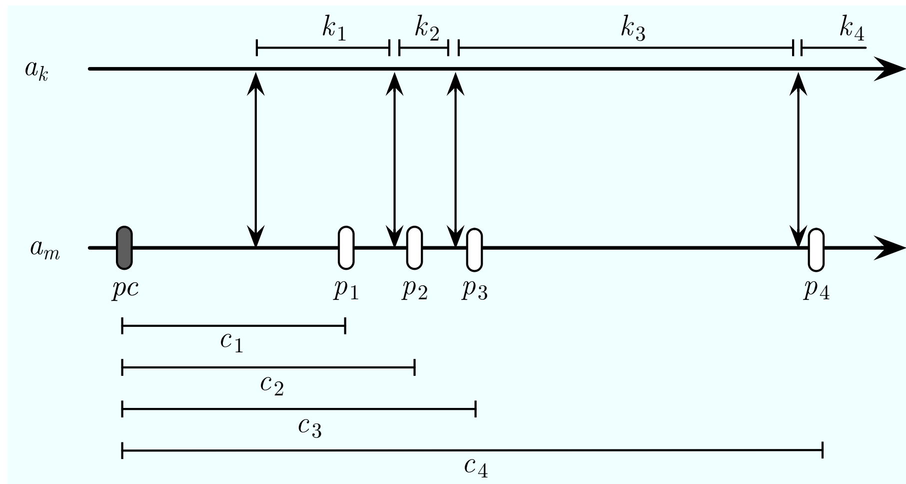


Abbildung 5.9.: Positionen für Potentielle Checkpoints eines Akteurs mit K-Order-Aufrufen

Nach dem Greedy-Prinzip wird ausgehend von einem bereits gesetzten potentiellen Checkpoint *pc* die folgende Entwicklung des Akteurs selbst sowie die der direkten Kommunikationspartner betrachtet. In Abbildung 5.9 ist die prognostizierte Ausführung eines Akteurs a_m und eines K-Akteurs a_k dargestellt. Wie bereits erläutert, kann die Kommunikation über Operationen-orientiertes Rendez-

vous abstrakt als Pfeil in beide Richtungen betrachtet werden. Der Berechnungs-Fortschritt der Akteure ist nach rechts dargestellt, wobei die Instruktionen auf eine abstrakte Zeiteinheit abgebildet werden. Das dunkel dargestellte abgerundete Rechteck pc sei als letzter potentieller Checkpoint für Akteur a_m ausgewählt worden; nun betrachtet das Verfahren die weitere Entwicklung, um eine lokale Entscheidung für den nächsten pc zu treffen.

Die erste interessante Position dafür ist nach dem Idealintervall I_{ideal} und ist in der Abbildung mit p_1 bezeichnet. Das zugehörige Intervall (in diesem Fall das Idealintervall) ist mit c_1 beschriftet. Im Beispiel sind weitere Positionen p_2, p_3 und p_4 jeweils direkt nach Kommunikationskanten eingezeichnet. Die erläuterte Zwang-Bedingung, also eine Konstellation, in der ein Checkpoint bei einem anderen Akteur erzwungen werden kann, tritt in folgenden Fällen auf: Hat Akteur a_m seit seiner letzten Kommunikation mit Akteur a_k einen Checkpoint gesetzt, so kann bei Akteur a_k potentiell ein Checkpoint erzwungen werden, sobald a_m und a_k erneut miteinander kommunizieren.

In Abbildung 5.9 erzeugen also die betrachteten Positionen p_1, p_2, p_3 und p_4 so eine Situation, da sie von zwei Kommunikationskanten eingeschlossen werden. Das Intervall, das sich bei einem K-Akteur durch zwei aufeinander folgende K-Order-Aufrufe des selben Akteurs ergibt und eine Position einschließt, wie beispielsweise Intervall k_1 für Position p_1, wird folgend als *umschließendes Kommunikationsintervall* bezeichnet. Existiert nur eine einzelne Kommunikationsbeziehung, so existiert kein umschließendes Kommunikationsintervall.

Wird ein potentieller Checkpoint an einer der Positionen p_1, p_2, p_3 oder p_4 erzeugt, so könnte dieser Checkpoint beim Systemablauf einen Checkpoint bei Akteur a_k erzwingen. Ob dies tatsächlich der Fall ist, hängt von den freiwilligen Checkpoints von a_k sowie von den durch andere Akteure bei a_k erzwungenen Checkpoints ab. Die tatsächliche Reihenfolge der Kommunikation und damit die eventuell erzwungenen Checkpoints lassen sich jedoch nicht statisch vorherbestimmen. Für die Metrik ist also die Information, dass eventuell ein Checkpoint erzwungen wird, ausschlaggebend für die Positionierung der freiwilligen Checkpoints.

Interessant ist nun, welcher Overhead durch einen erzwungenen Checkpoint bei Akteur a_k durch einen freiwilligen Checkpoint von Akteur a_m bei den jeweiligen Positionen p_1, p_2, p_3 oder p_4 verursacht würde. Dieser Schaden wird ausgedrückt durch zusätzlichen Overhead, der sich durch ein ungünstiges Checkpoint-Intervall bei a_k ergibt (vgl. Abschnitt 5.1.4). Der zusätzliche Overhead lässt sich abschätzen, indem das kleinstmögliche Checkpoint-Intervall, das durch den erzwungenen Checkpoint bei a_k entstehen kann, als Basis verwendet wird. Der Overhead ist

also eine worst-case-Abschätzung des erzeugten Overheads bei a_k durch einen erzwungenen Checkpoint.

Das kleinstmögliche Checkpoint-Intervall, das durch einen freiwilligen Checkpoint von a_m an Position p_1 bei a_k erzwungen werden kann, ist genau das umschließende Kommunikationsintervall k_1: Falls der K-Akteur a_k seinen letzten Checkpoint vor der zuvor stattfindenden Kommunikationskante gespeichert hat, also vor dem Intervall k_1, so ist das Intervall zum erzwungenen Checkpoint mindestens k_1. Speichert dagegen a_k einen Checkpoint während des Intervalls k_1, so kann a_m durch einen Checkpoint an Position p_1 keinen Checkpoint erzwingen; die Zwang-Bedingung ist nicht erfüllt.

Der Overhead $O(k_1)$ dieses minimalen Intervalls k_1 wird berechnet (vgl. Abschnitt 5.1.4). Das Intervall k_1 stellt die untere Schranke für ein durch einen Checkpoint an Position p_1 erzwungenes Checkpoint-Intervall dar, der errechnete Overhead $O(k_1)$ ist demnach die obere Schranke für den Overhead des erzwungenen Checkpoints. Der *zusätzliche* Gesamt-Overhead durch diesen erzwungenen Checkpoint – im Gegensatz zu einem freiwilligen Checkpoint bei a_k – ist demnach die Differenz $O(k_1) - O(I_{ideal})$, da der minimale Overhead für das Speichern eines Checkpoints $O(I_{ideal})$ beträgt. Dieser Checkpoint mit einschließendem Kommunikationsintervall k_1 kann von Akteur a_m nicht erzwungen werden, falls die auf Position p_1 folgende Kommunikationskante übersprungen und erst bei p_2 wieder ein pc vorbereitet wird. Allerdings wird dadurch das Checkpoint-Intervall bei a_m größer, was ebenfalls zusätzlichen Overhead erzeugt. Dieser zusätzliche Overhead lässt sich durch das Intervall c_2 zwischen dem letzten gesetzten pc und der betrachteten Position p_2 als $O(c_2)$ berechnen.

Der Gesamt-Overhead einer Position ergibt sich demnach aus dem Overhead durch erzwungene Checkpoints, die durch das einschließende Kommunikationsintervall abgeschätzt werden, und dem Overhead des Intervalls des freiwilligen Checkpoints selbst. Als zweiter additiver Bestandteil der Bewertungsfunktion ergibt sich also die Differenz $O(k_x) - O(I_{ideal})$, falls ein einschließendes Kommunikationsintervall k_x für die zu bewertende Position existiert. Dieser zusätzliche Bestandteil der Bewertung spiegelt den zusätzlichen Overhead wider, der bei dieser Platzierung eines Checkpoints durch einen erzwungenen Checkpoint bei einem anderen Akteur erzeugt würde. Für jede sinnvolle Position, also für jedes einschließende Kommunikationsintervall, wird so der Gesamt-Overhead bestimmt und dann die Position mit minimalem Overhead gewählt. Dort wird der nächste pc gesetzt und iterativ das Verfahren wieder angewandt.

5.2.5.5 Einbezug von Erzeugung und Terminierung

Neben kurzen Checkpoint-Intervallen nimmt auch die Anzahl vorhandener Checkpoints im System Einfluss auf den Overhead, falls dadurch die Checkpoint-Intervalle unverändert bleiben. Bei der Erzeugung eines Akteurs tritt genau dieser Fall auf. Angenommen, Akteur a_m erzeugt einen Akteur a_n, so stellt sich die Frage, ob ein freiwilliger Checkpoint besser vor oder nach der Akteurerzeugung zu speichern ist. Zunächst ist das Checkpoint-Intervall ausschlaggebend für den Overhead, allerdings müssen bei einer Akteurerzeugung zusätzliche Faktoren berücksichtigt werden: Wird ein Checkpoint kurz vor der Erzeugung erstellt, so ist ein Rücksetzpunkt im System geschaffen, der im Fehlerfall für beide Akteure genutzt werden kann, obwohl nur der Checkpoint eines Akteurs gespeichert wurde. Wird dagegen erst nach Akteurerzeugung ein Checkpoint erzeugt, so müssen die Zustände beider Akteure, also zwei Checkpoints gespeichert werden.

In der Metrik wird also die Position vor einem Akteurerzeugungs-Ereignis der Position danach vorgezogen. In der Positions-Bewertungsfunktion wird deshalb ein Bonus in Höhe eines gesparten Checkpoints, also des Overheads beim Idealintervall $O(I_{ideal})$ für eine Position gegeben, falls das nächst folgende relevante Ereignis eine Akteurerzeugung `create` ist. Ist ein einschließendes Kommunikationsintervall so kurz, dass trotz des Bonus der Malus durch erzwungene Checkpoints überwiegt, so ist für die Optimierung des Gesamt-Overhead die Lösung mit Checkpoints bei beiden Akteuren nach der Erzeugung effizienter.

Anders ist die Situation beim Beenden eines Akteurs. Da die Terminierung eines Akteurs zeitlich beliebig vor dem `join`-Ereignis seines Erzeugers stattfinden kann, ist eine Speicherung des Zustands noch vor der Terminierung immer zu bevorzugen, da bei einem Rollback dann kein Akteur wieder erzeugt und gestartet werden muss, lediglich der terminale Zustand im Speicher muss wieder hergestellt werden. Da allerdings nach einer Terminierung kein K-Order-Aufruf mehr stattfinden kann, kann auch eine Position vor dem Ereignis `term` nie ein einschließendes Kommunikationsintervall besitzen, es wird also in der Bewertungsfunktion kein zusätzlicher Overhead aufsummiert. Deshalb reicht ein Bonus analog zur Akteurerzeugung in Höhe von $O(I_{ideal})$, um sicherzustellen, dass vor dem `term`-Ereignis ein Checkpoint erstellt wird. Der terminierte Zustand jedes Akteurs wird also als Checkpoint gespeichert. Bei der Entscheidung zur Laufzeit wird berücksichtigt, dass diese Checkpoints nicht übersprungen werden dürfen.

5.2.5.6 Abbruchbedingung

Es werden in einem Schritt natürlich nicht alle Positionen bis zur Terminierung des Akteurs betrachtet, sondern nur solange gesucht, bis ein lokales Minimum des abgeschätzten Overheads gefunden wurde. Die Abbruchbedingung für die iterative Bewertung möglicher Positionen ergibt sich direkt aus der Berechnung: sobald die Zunahme des Overheads durch das vergrößerte Intervall c_x des freiwilligen Checkpoints den zusätzlichen Overhead des erzwungenen Checkpoints der bisherigen besten Position übersteigt, kann abgebrochen werden und die beste Position wird zum potentiellen Checkpoint. Es muss lediglich überprüft werden, ob auf die aktuelle Position ein `create`- oder `term`-Ereignis folgt, da wie im letzten Abschnitt beschrieben in diesen Fällen ein Bonus für die Position einfließt.

5.2.5.7 Bewertungsfunktion

Wird dieses Verfahren nun auf die Kommunikation mit mehreren Akteuren erweitert, so ergibt sich aus den erklärten Bestandteilen folgende *Positions-Bewertungsfunktion* f_{PB}, wobei $A_k \subset A$ die Menge der K-Akteure im System sei und analog zur Abbildung c_x das Intervall zwischen dem letzten pc und der betrachteten Position p_x sei, sowie k_x für den jeweiligen K-Akteur das p_x umschließende Kommunikationsintervall sei, oder 0, falls mit diesem K-Akteur kein Kommunikationsintervall besteht (vgl. [PP07]):

$$f_{PB}(p_x) = O(c_x) + \sum_{\forall a_k \in A_k} M_{forced}(a_k) - B_{create}$$

$$M_{forced}(a_k) = \begin{cases} O(k_x) - O(I_{ideal}) & \text{falls } 0 < k_x < I_{ideal} \\ 0 & \text{sonst} \end{cases}$$

$$B_{create} = \begin{cases} O(I_{ideal}) & \text{falls } \texttt{create} \text{ oder } \texttt{term} \text{ folgt} \\ 0 & \text{sonst} \end{cases}$$

Die Bewertung einer Position p_x setzt sich wie erklärt aus drei Teilen zusammen. Zunächst fließt der Overhead $O(c_x)$ des entstehenden Checkpoint-Intervalls durch den freiwilligen Checkpoint selbst ein. Der zweite Bestandteil ist ein Malus $M_{forced}(a_k)$ durch zusätzlichen Overhead erzwungener Checkpoints. Es kann nur ein Checkpoint erzwungen werden, falls ein einschließendes Kommunikationsintervall besteht und es gibt nur zusätzlichen Overhead durch den erzwungenen Checkpoint, falls das einschließende Kommunikationsintervall kleiner als

das Idealintervall ist, demnach tritt der Fall mit der Bedingung $0 < k_x < I_{ideal}$ ein. Da auch der K-Akteur für einen Checkpoint mindestens den Overhead des Idealintervalls aufwenden muss, ist der zusätzliche Overhead die Differenz aus $O(k_x) - O(I_{ideal})$. Für jeden K-Akteur, mit dem der Akteur der zu bewertenden Position kommuniziert, wird dieser Malus aufsummiert: $\sum_{\forall a_k \in A_k} M_{forced}(a_k)$.

Die erarbeitete Positions-Bewertungsfunktion ist die Basis für die iterative Bewertung der sinnvollen, auf einen festgelegten potentiellen Checkpoint *pc* folgenden Positionen, um den nächsten *pc* vorzubereiten. Es müssen dabei nur bestimmte Positionen betrachtet und bewertet werden, da auf Grund der Berechnung nur folgende Positionen mögliche Kandidaten für Overhead-Minima darstellen: Zunächst die Position, die zum letzten *pc* das Idealintervall I_{ideal} realisiert, da die Overheadfunktion $O(I)$ hier das einzige Minimum hat. Nach dieser Position jede Position direkt nach einer Kommunikationskante, da hier unter Umständen ein geringerer Malus auf Grund des ersparten Overheads eines nicht erzwungenen Checkpoints mit kurzem Intervall einfließt.

Der Bonus B_{create} für das Einsparen eines Checkpoints, der vor einem `create` addiert wird, wird in dem gesamten einschließenden Kommunikationsintervall gegeben, das heißt er bevorzugt ebenfalls entweder eine Position mit Idealintervall oder eine Position direkt nach einer Kommunikation. Zuletzt ist noch die Position direkt vor einem `term`-Ereignis interessant, hier wird der Bonus ebenfalls gegeben, allerdings nur als letzte Position vor der Terminierung, da die relevanten Rückgabedaten des Akteurs berechnet sind und deshalb im Fall eines Rollbacks hier die erneute Berechnung gespart werden kann.

Insgesamt überspringt die Positionierung auf Basis der Bewertungsfunktion f_{PB} genau die Abschnitte eines Akteurs, in denen er mehrfach mit einem K-Akteur kommuniziert, bei dem er Checkpoints mit sehr kurzen Checkpoint-Intervallen erzwingen könnte und akzeptiert dafür etwas längere Checkpoint-Intervalle bei dem Akteur selbst. Zudem werden Positionen vor einer Akteur-Erzeugung bevorzugt und sichergestellt, dass der terminale Zustand eines Akteurs als Checkpoint gespeichert wird. Dies liegt daran, dass gemäß dem MoDiS-Systemmodell die Auflösung eines Akteurs nicht direkt auf seine Terminierung folgt.

5.2.5.8 Schleifen und bedingte Anweisungen

Zur Übersetzungszeit werden somit für jeden Akteur des Systems iterativ günstige Positionen für potentielle Checkpoints ermittelt. An den gewählten Positionen wird in den Anwendungscode des Akteurs ein Aufruf an das Laufzeit-

Management des MoDiS-Projekts generiert. Allerdings ist im Übersetzer zunächst nicht bekannt, welchen Pfad ein Akteur bei der Ausführung einer bedingten Anweisung nimmt und wie oft Schleifen durchlaufen werden. Zur Laufzeit wird für jeden dieser Aufrufe anfangs geprüft, wie groß das tatsächliche Checkpoint-Intervall vom letzten Checkpoint zum aktuellen Aufruf wäre. Ist dies größer als eine festgelegte Schranke, so wird wie geplant der Checkpoint gespeichert. Ist das Intervall kleiner als die Schranke, so überspringt das Management diese Position, die Funktion kehrt aus dem Laufzeit-Management zurück, ohne einen Checkpoint zu speichern. Die Schranke muss demnach entsprechend der Overheadfunktion $O(I)$ so gewählt sein, dass der Overhead des aktuell zu bewertenden Intervalls größer ist, als der Overhead eines Intervalls, das sich aus der Summe des aktuellen Intervalls und dem Idealintervall ergibt.

Prinzipiell muss für alle möglichen Fälle dafür gesorgt werden, dass nach sinnvollen Intervallen ein Checkpoint gespeichert wird. Bei bedingten Anweisungen werden deswegen beide Alternativen betrachtet und so behandelt, als ob sie sicher gewählt würden, sodass bei beiden (allen) Alternativen ein potentieller Checkpoint integriert wird, falls die Pfade lange genug sind. Nach der bedingten Anweisung ist im Greedy-Verfahren zur Übersetzungszeit nicht bekannt, welcher der Pfade gewählt wird, wie weit also tatsächlich der letzte *pc* zurückliegt, es wird daher vom weniger weit zurück liegenden die nächste Position bestimmt. Generell hat diese Entscheidung aber wenig Auswirkung auf den Overhead, da lediglich zu kurze Intervalle stark negative Overhead-Auswirkungen zeigen und diese bei freiwilligen Checkpoints vom Laufzeit-Management verhindert werden.

Für Schleifen wird dieser Mechanismus ebenfalls eingesetzt: Ist zur Übersetzungszeit nicht bekannt, ob eine Schleife so oft durchlaufen wird, dass – auf Grund des Idealintervalls – ein Checkpoint nach einigen Schleifendurchläufen gespeichert werden müsste, so wird immer ein *pc* innerhalb des Schleifenkörpers gesetzt. Zur Laufzeit werden dann nicht bei jedem Schleifendurchlauf Checkpoints gespeichert, sondern wie beschrieben die meisten Aufrufe übersprungen und nur nach einem sinnvollen tatsächlichen Intervall ein Checkpoint gespeichert. Der Greedy-Algorithmus muss demnach bei der Übersetzungszeit für jede Schleife, deren Durchläufe nicht statisch vorhersagbar sind, sicherstellen, dass innerhalb der Schleife ein *pc* gesetzt wird. Nach Verlassen der Schleife wird das Idealintervall für den nächsten *pc* entsprechend vom *pc* innerhalb der Schleife, also vom letzten Schleifendurchlauf angenommen. Der maximale Fehler ist hier, dass das tatsächlich resultierende Intervall zwei mal so groß ist wie das Idealintervall und somit akzeptabel.

Die Funktion im Laufzeit-Management stellt sicher, dass der Zustand eines einzelnen Akteurs als Checkpoint gespeichert wird. Gemäß der Prinzipien des

MoDiS-Projekts soll das Checkpoint-Verfahren transparent für den Anwendungsprogrammierer sein, deshalb wurde ein Kernel-Level Checkpointing realisiert (vgl. [KK07], Kap. 6.2). Die Funktionalität zum Speichern und Wiederherstellen des Zustands eines Prozesses werden als Kern-Funktionen angeboten und vom Laufzeit-Management aufgerufen. Details zur Implementierung des Kern-Moduls sind in [Pfl07] näher beschrieben.

5.3 Bewertung

Die Verbesserung des Overheads des kommunikationsinduzierten Checkpoint-Rollback-Verfahrens durch die Vorbereitung der Positionierung freiwilliger Checkpoints im Übersetzer wurde durch Simulation des Verfahrens mit und ohne Vorbereitung evaluiert. Das simulierte Verfahren ohne zuvor stattfindende Positionierung der Checkpoints basiert auf dem kommunikationsinduzierten Checkpoint-Verfahren nach Briatico (vgl. [BCS84]), erweitert um die beschriebene Anpassung an das dynamische Prozesssystem in MoDiS; das Verfahren wird in diesem Abschnitt zur Unterscheidung dennoch als *Briatico-CIC* bezeichnet. In dem Verfahren wird das Idealintervall berechnet und freiwillige Checkpoints in einem Prozess genau dann gesetzt, wenn seit dem letzten (freiwilligen oder erzwungenen) Checkpoint das Idealintervall an Berechnungsschritten abgelaufen ist.

Im Vergleich dazu wurden bei dem Verfahren mit der beschriebenen Analyse des Kommunikationsverhaltens und der Bewertung der Positionierung der Checkpoints gemäß der Funktion f_{PB} die Positionen für freiwillige Checkpoints vor dem Start der Simulation bestimmt. Während der Simulationsläufe wird dann bei den vorbestimmten Positionen das tatsächlich resultierende Intervall ermittelt und, wie in Abschnitt 5.2.5 beschrieben, entweder ein freiwilliger Checkpoint gesetzt oder die Position übersprungen. Dieses Verfahren wird in der Simulation als *MoDiS-CIC* bezeichnet.

5.3.1 Simulator

Um möglichst praxisnahe Ergebnisse zu erzielen, wurden Beispiele von MoDiS-Anwendungen in Form der protokollierten konsistenten Sichten (vgl. Abschnitt 4.1.2.3 und Anhang B) als Eingabe für die Simulation genutzt. Die Kanten der Graphen wurden mit abstrakten Berechnungseinheiten gewichtet, die die Ausführungszeiten repräsentieren. Die Ausführung dieser Anwendungen wird simuliert, indem die entstandenen Graphen durchlaufen werden. An Prozesskanten

wird in jedem abstrakten Berechnungsschritt gemäß eingestellter Wahrscheinlichkeit überprüft, ob ein Fehler stattgefunden hat. Im Fehlerfall wird die aktuellste Recovery-Line ermittelt und die Simulation von dort fortgesetzt. Ist der letzte Knoten des Eingabegraphen abgearbeitet, so wird ermittelt welche abstrakte Ausführungszeit der Systemablauf in einem System, in dem jeder Akteur auf einer eigenen Stelle ausgeführt wird, benötigt hätte.

Entscheidend für die Vergleichbarkeit der Ergebnisse ist, dass die Zufallswerte, anhand derer mit der eingestellten Fehlerrate λ verglichen wird, ob ein Fehler aufgetreten ist, für beide Verfahren identisch ermittelt werden. Deshalb wird für jeden einzelnen Simulationslauf, in welchem die Ausführung des eingegebenen Graphen für beide Verfahren einmal simuliert wird, ein `random`-Objekt mit identischem `seed` generiert, sodass in den beiden Verfahren Briatico-CIC und MoDiS-CIC die gleichen Zufallswerte erzeugt werden. Mit diesen wird die Fehlerwahrscheinlichkeit verglichen und bei Unterschreitung ein Fehler gemeldet. Ferner kann die einzelne Verteilung der Zufallswerte eines der Verfahren begünstigen; um dennoch aussagekräftige Ergebnisse zu erhalten werden für je eine Kombination der Eingabewerte, also der Fehlerwahrscheinlichkeit λ und den Checkpointkosten c, eine große Zahl dieser Simulationsläufe mit jeweils unterschiedlichen, ebenfalls zufällig generierten Werten für den `seed` des `random`-Objekts durchgeführt. Anschließend werden die arithmetischen Mittelwerte der abstrakten Ausführungszeiten und gespeicherten Checkpoints berechnet.

Die Ereignisgraphen, die als Eingabe für den Simulator verwendet wurden, lassen sich in zwei Klassen einteilen:

1. *Geringe Kooperation:* In diesen Beispielen findet keine oder nur wenig Kommunikation über das Operationen-orientierte Rendezvous statt. Die Kantengewichte der Prozesskanten zwischen den Kommunikationsaufrufen sind im Verhältnis zu den Checkpointkosten c hoch gewichtet.

2. *Intensive Kooperation:* In diesen Beispielen findet verhältnismäßig viel Kommunikation über Operationen-orientiertes Rendezvous statt. Die Prozesskanten zwischen Kommunikations-Ereignissen sind im Verhältnis zu den Checkpointkosten c mit wenig Rechenaufwand gewichtet.

Um bei fester Gewichtung eines Eingabegraphen aussagekräftige Ergebnisse zu erhalten, wurden Simulationen mit verschiedenen Kombinationen aus Fehlerwahrscheinlichkeit λ und Checkpointkosten c durchgeführt. Diese beiden Parameter legen das Idealintervall fest, von dem wiederum die Positionierung der freiwilligen Checkpoints beider Verfahren abhängt. Somit wird ausgeschlossen, dass Beispielgraphen für bestimmte Werte des Idealintervalls eines der beiden

Verfahren bevorzugen, und dadurch das Ergebnis der Simulationen zu stark vom einzelnen Beispiel abhängt.

5.3.2 Ergebnisse

In Abbildung 5.10 sind die durchschnittlichen Laufzeiten eines Beispiels der Klasse mit intensiver Kooperation dargestellt. Für 50 verschiedene Kombinationen aus Fehlerwahrscheinlichkeit λ und Checkpointkosten c sind die durchschnittlichen Laufzeiten der Verfahren Briatico-CIC und MoDiS-CIC dargestellt. Jeder eingetragene Werte entspricht dem Durchschnittswert von 1000 Simulationsläufen pro Verfahren für eine Kombination aus λ und c. Von links nach rechts wird nach jeweils 10 Werten die Fehlerrate λ reduziert; innerhalb der 10 Werte wird der konstante Wert von λ mit den 10 unterschiedlichen Checkpointkosten c kombiniert. Für beide Verfahren gilt, dass die Laufzeit abnimmt, je geringer die Fehlerrate ist, da weniger Rollbacks durchgeführt werden müssen. Zudem ist erkennbar, dass für einen gegebenen Wert λ abhängig von c die Laufzeit stark schwankt. Dies liegt daran, dass das Idealintervall von der Kombination dieser Werte abhängt und das Idealintervall wiederum für die Kommunikations-Strukturen des Beispiels günstig oder ungünstig ausfallen kann, also zum erzwingen vieler Checkpoints führen kann oder nicht. Deutlich erkennbar ist, dass sich für bestimmte Kombinationen aus λ und c die Laufzeit des Verfahrens Briatico-CIC vervielfacht, wohingegen die Laufzeit des Verfahrens MoDiS-CIC zwar schwankt, allerdings bei Weitem nicht die hohen Laufzeit von Briatico-CIC annimmt.

Dieses Verhalten lässt sich durch die Anzahl gespeicherter Checkpoints erklären. In Abbildung 5.11 sind die durchschnittlichen Anzahlen gespeicherter Checkpoints der beiden Verfahren dieser Simulationen dargestellt. Offensichtlich korrelieren die Laufzeitwerte mit den Mengen gespeicherter Checkpoints. Dies bestätigt die eingangs in diesem Kapitel erklärte Kritik an kommunikationsinduzierten Checkpoint-Verfahren: In Systemen mit viel Kommunikation steigt die Zahl an erzwungenen Checkpoints extrem an und führt zu exponentiellem Anstieg des Overheads durch sehr kurze Checkpoint-Intervalle. Offensichtlich bleibt die Anzahl erzwungener Checkpoints für das Verfahren MoDiS-CIC durch die Analyse des Kommunikationsverhaltens gering, sodass die ungünstigen Checkpoint-Intervalle vermieden werden. Die Laufzeiten schwanken deshalb nur moderat.

Beispiele der Klasse *geringe Kooperation* zeigen ein gänzlich anderes Verhalten. In Abbildung 5.12 sind analog die durchschnittlichen Laufzeitwerte eines Ereignisgraphen dargestellt. Es ist zu beachten, dass die Laufzeiten dieses Diagramms auf Grund der unterschiedlichen Skala nicht mit dem Beispiel der Klasse intensi-

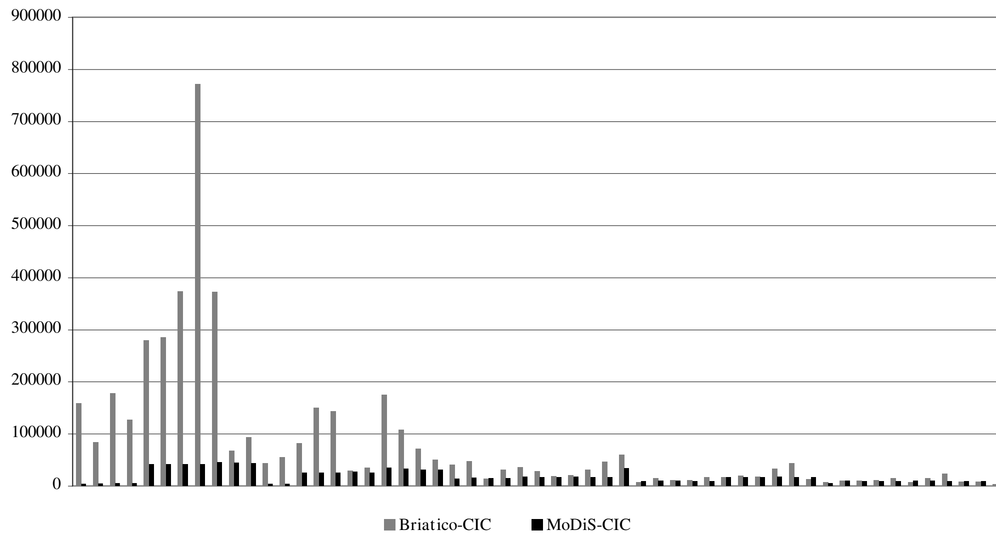


Abbildung 5.10.: Durchschnittliche Laufzeit eines Beispiels mit intensiver Kooperation

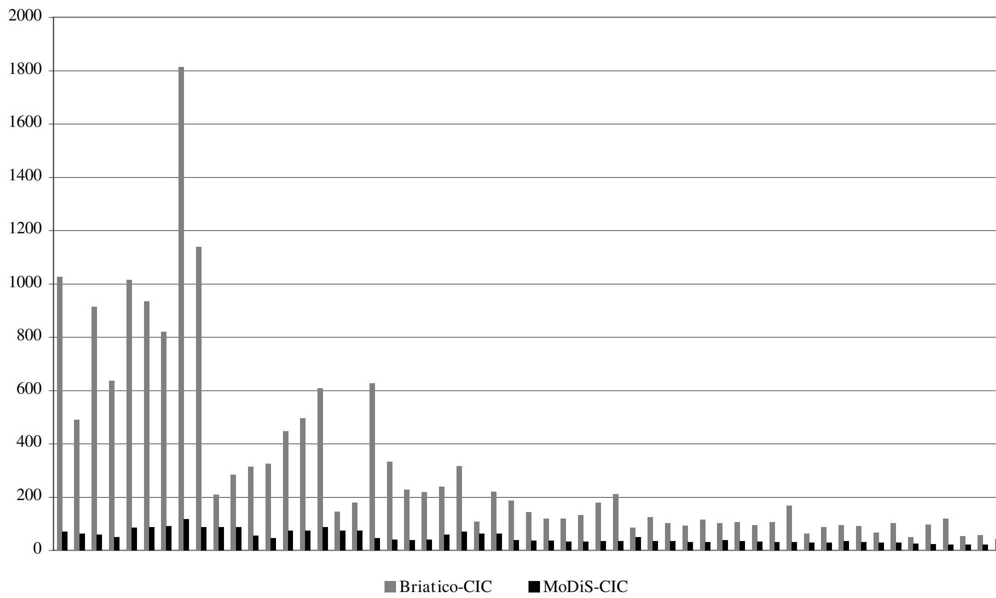


Abbildung 5.11.: Durchschnittliche Anzahl gespeicherter Checkpoints des Beispiels mit intensiver Kooperation

ve Kooperation vergleichbar sind. Beide Verfahren schwanken in ihren Laufzeiten je nach Kombination von λ und c. Allerdings treten die Extremwerte nicht auf, da auf Grund der seltenen Kommunikation nur wenige Checkpoints erzwungen werden. Das Diagramm der durchschnittlich gespeicherten Checkpoints dieser Simulation ist in Abbildung 5.12 dargestellt. Offensichtlich ist für diese Klasse von Beispielen das Verfahren MoDiS-CIC etwas schlechter als Briatico-CIC, da durch die Positionierung freiwilliger Checkpoints vor Systemablauf teilweise längere Intervalle gewählt werden, obwohl zur Laufzeit häufig an diesen Stellen gar keine Checkpoints erzwungen werden; diese Eigenschaft liegt an der lokalen Sicht des Algorithmus zur Positionierung auf einen begrenzten Systemausschnitt. Der Unterschied der beiden Verfahren ist in der Klasse *geringe Kooperation* jedoch wesentlich geringer, als in der Klasse mit intensiver Kooperation.

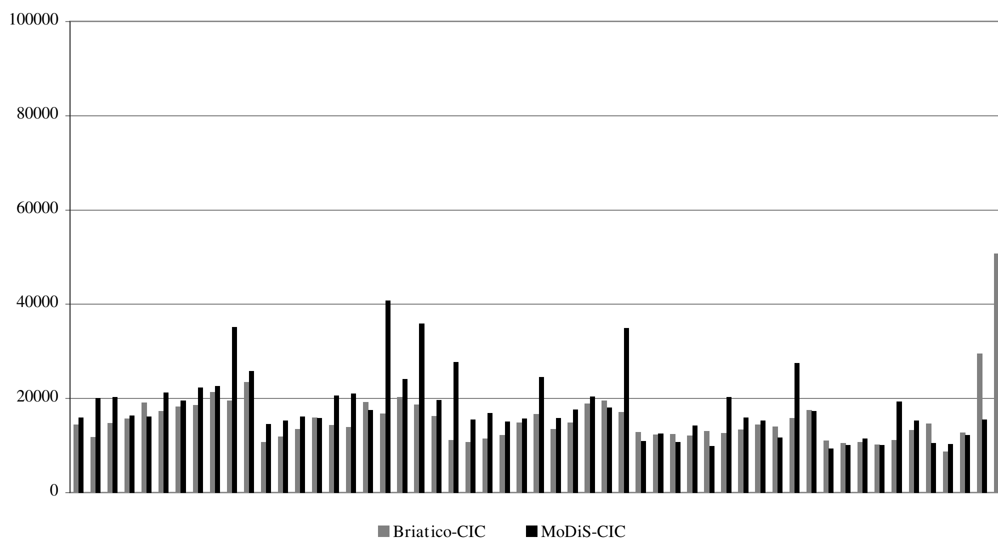


Abbildung 5.12.: Durchschnittliche Laufzeit eines Beispiels mit geringer Kooperation

Insgesamt bestätigt die Simulation, dass das in diesem Kapitel vorgestellte kommunikationsinduzierte Checkpoint-Verfahren auf Grund der Positionierung der freiwilligen Checkpoints im Übersetzer, den zu hohen Overhead in Systemen oder Systemteilen mit intensiver Kommunikation umgeht. Somit kann der Gesamt-Overhead des kommunikationsinduzierten Checkpoint-Verfahrens verbessert werden, denn der eingesparte Overhead für Systeme mit intensiver Kommunikation übersteigt bei weitem den zusätzlichen Overhead in Systemen mit geringer Kommunikation.

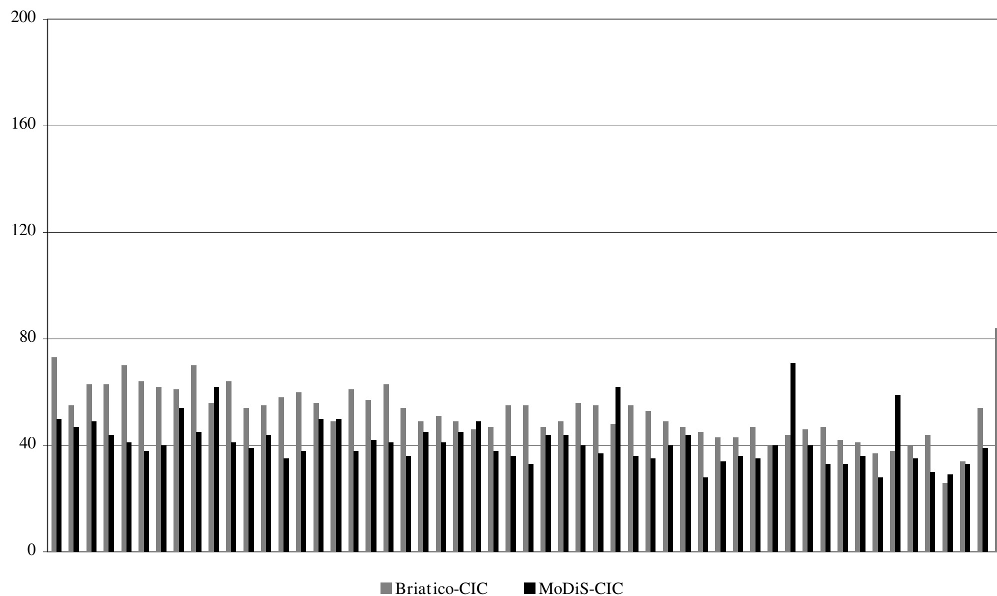


Abbildung 5.13.: Durchschnittliche Anzahl gespeicherter Checkpoints des Beispiels mit geringer Kooperation

Für eine absolute Abschätzung des Overheads der Verfahren müssten beide Verfahren tatsächlich im Management des Experimentalsystems integriert werden. Beispielsweise haben tatsächliche Berechnungs- und Kommunikationszeiten, Stellenzuweisung, sowie Scheduling usw. zusätzlichen Einfluss auf die Ausführung des Systems und damit auf die Auswirkungen gespeicherter Checkpoints auf die Systemperformanz. Diese Komplexität wurde in der Simulation nicht berücksichtigt, da von einem Prozessor pro Akteur und äquivalenten Kosten für Kommunikation, Speichern von Checkpoints und Rechenlast ausgegangen wurde. Dennoch zeigt die Simulation, dass der relative Unterschied der Verfahren in der Klasse mit intensiver Kooperation hoch genug ist, dass insgesamt der Overhead des Checkpoint-Verfahrens durch die Positionierung im Übersetzer entscheidend verbessert werden kann.

5.4 Verwandte Arbeiten

Am Anfang dieses Kapitels wurden bereits alternative Ansätze zu kommunikationsinduziertem Checkpointing vorgestellt. Dem Autor ist kein weiteres Verfahren bekannt, das einen vergleichbaren Ansatz zur Platzierung freiwilliger Checkpoints

mit Hilfe von Übersetzer-Informationen über die Kommunikationsbeziehungen verfolgt.

Einige Forschungsarbeiten versuchen allerdings, andere Informationen im Übersetzer zur Optimierung von Checkpoint-Verfahren zu nutzen. Es wird versucht, den Aufwand des Speicherns eines einzelnen Checkpoints zu verringern, indem die zu speichernden Daten reduziert werden. Somit soll die Gesamtperformanz erhöht werden, da das Speichern eines einzelnen Checkpoints eines Prozesses weniger Overhead verursacht. Grundsätzlich sind diese Verfahren mit dem hier vorgestellten Ansatz kombinierbar. Die Verfahren verfolgen im wesentlichen zwei Techniken, um den zu speichernden Zustand eines Prozesses gering zu halten. Erstens werden die Datenzugriffe der Prozesse im Verlauf untersucht, sodass Daten, auf die nicht mehr zugegriffen wird oder die unveränderlich sind, nicht mitgespeichert werden. Zweitens werden inkrementelle Checkpoints erzeugt, d.h. nur jeweils die Veränderungen des Gesamtzustands zum letzten Checkpoint werden gesichert (vgl. [EAWJ02] und [KK07]).

5.5 Zusammenfassung

In diesem Kapitel wurde ein kommunikationsinduziertes Checkpoint-Verfahren für MoDiS beschrieben. Das allgemeine Verfahren nach Briatico wurde an das Systemmodell des MoDiS-Projekts mit dynamischer Prozesserzeugung und Operationen-orientiertem Rendezvous angepasst und das zugehörige Protokoll erklärt. Die Performanz von kommunikationsinduziertem Checkpointing hängt von den sich ergebenden Checkpoint-Intervallen ab. Sind diese auf Grund von erzwungenen Checkpoints sehr kurz, so kann der Overhead des Verfahrens beliebig hoch werden.

Durch die Analyse des Kommunikationsverhaltens der Anwendung im Übersetzer können die freiwilligen Checkpoints so positioniert werden, dass weniger Checkpoints erzwungen und speziell Checkpoints mit sehr kurzen Intervallen vermieden werden. Eine Bewertungsfunktion vergleicht den Overhead sinnvoller Positionen, indem einerseits der Overhead des entstehenden Checkpoint-Intervalls und andererseits der Overhead durch möglicherweise erzwungene Checkpoints in Betracht gezogen wird. In einem Greedy-Verfahren wird jeweils die Position mit günstigstem Overhead gewählt und der Code zur Erstellung eines Checkpoints in den Akteur-Code eingefügt.

Zur Laufzeit werden die tatsächlich entstehenden Intervalle überprüft und gegebenenfalls übersprungen. Der Gesamt-Overhead von kommunikationsinduziertem

Checkpointing kann so entscheidend verringert werden. Die Simulation der beiden Verfahren zeigt, dass für Anwendungen mit geringer Kooperation der Gesamt-Overhead durch die Positionierung etwas schlechter ausfällt. Bei Anwendungen mit intensiver Kooperation vervielfacht sich allerdings der Overhead des Verfahrens ohne Positionierung, wohingegen mit Positionierung nur ein moderater Anstieg des Gesamt-Overheads zu verzeichnen ist.

Das beschriebene Checkpoint-Verfahren stellt demnach ein effizientes Verfahren zur Rückwärtsbehebung in MoDiS dar. Erkennt das Management mit Hilfe der Verfahren aus Kapitel 4 einen Fehlerzustand, so kann automatisiert das System in einen zuvor gespeicherten, konsistenten Zustand zurück gesetzt und die Ausführung ab diesem fortgesetzt werden. Transiente Fehler können somit bereits toleriert werden. Permanente Fehler, insbesondere Softwarefehler würden dann aber mit hoher Wahrscheinlichkeit erneut auftreten. Im folgenden Kapitel wird das Konzept der automatisierten Diversität erarbeitet, das generell auf Rückwärtsbehebung – wie beispielsweise dem hier vorgestellten Verfahren – basiert, aber durch Variation der Management-Entscheidungen eine geänderte Ausführung ermöglicht.

Kapitel 6

Automatisierte Diversität auf Management-Ebene

Inhalt

6.1. Kernidee 147
6.2. Fehlertoleranz durch Diversität auf Management-Ebene . . . 159
6.3. Diskussion 203
6.4. Verwandte Arbeiten 208
6.5. Zusammenfassung 209

In diesem Kapitel wird ein Konzept zur Fehlertoleranz erarbeitet, das auf automatisierter Diversität im Management basiert. Zunächst wird der Ansatz motiviert und die Kernidee vorgestellt. Anschließend wird iterativ erarbeitet, auf welcher Basis die Konkretisierungsentscheidungen variiert werden können, und die Entscheidungsfindung anhand des MoDiS-Projekts veranschaulicht.

6.1 Kernidee

Folgend werden einige Ansätze und Probleme der Softwareentwicklung in Bezug auf die Zuverlässigkeit und Qualität der entwickelten Systeme aufgegriffen, die zu dem Ansatz der Diversität auf Management-Ebene führen und die zentralen Aspekte beschreiben.

6.1.1 Probleme der Softwareentwicklung

Die Entwicklung fehlerfreier Software stellt seit den Anfängen der Informatik eine Herausforderung dar. Verschiedenste Methoden, Werkzeuge, formale Verfahren, Konstruktionsprinzipien und Modelle werden eingesetzt, damit die entwickelte Software den gestellten Qualitätsanforderungen genügt. Diese Methoden fangen bei Schnittstellenbeschreibungen und formalen Spezifikationen, wie beispielsweise dem in Abschnitt 4.2 vorgestellte Vertragsmodell an und reichen bis zu Werkzeugen für Model-Checking, Test-Verfahren, Verifikation, dem Einsatz von statischen Analyse-Werkzeugen sowie Verfahren zur Erkennung und Toleranz von Fehlern[1] zur Laufzeit. Die Verfahren und Methoden sollen dazu beitragen, zuverlässige Software zu entwickeln.

Die Praxis zeigt, dass trotz der verschiedenen Ansätze und Methoden Software, abgesehen von sehr kleinen, einfachen Programmen, in der Regel fehlerbehaftet bleibt. Speziell gelingt es nicht, alle Fehlerursachen aus der Software zu entfernen bzw. Software so zu entwickeln, dass keine Fehler enthalten sind. Einerseits sind die Gründe hierfür in der steigenden Komplexität der Software zu finden. Dies gilt insbesondere für vernetzte, nebenläufige Systeme, da in diesen die Abhängigkeiten der Software zum Teil nicht mehr adäquat zu erfassen sind. Andererseits sind häufig wirtschaftliche Gesichtspunkte in der Praxis ein Hindernis, da die genannten Ansätze und Methoden aufwändig in Zeit und Kosten sind und somit nur bis zu einem gewissen Grad der erreichten Software-Qualität rentabel bleiben. Die grundsätzliche Ursache für das Vorhandensein von Fehlern in Software liegt an der menschlichen Schwäche, Fehler bei der Programmierung zu machen, und kann nicht prinzipiell ausgeschlossen werden.

Qualitätssicherungs-Maßnahmen der Softwareentwicklung ermöglichen in erster Linie, so genannte *Bohrbugs* aufzudecken und zu beheben. Als Bohrbugs werden Softwarefehler bezeichnet, die bei gleicher Eingabe deterministisch bei jedem Programm- oder Programmteil-Ablauf auftreten. Das Auftreten des Fehlerzustands wird also immer oder zumindest mit sehr hoher Wahrscheinlichkeit von der entsprechenden Fehlerursache hervorgerufen.

Zusätzlich können in nebenläufiger Software Fehlerursachen existieren, die lediglich in ganz bestimmten, unter Umständen seltenen Programmabläufen oder Systemzuständen zum Fehlerzustand führen bzw. ein solcher zu beobachten ist. Klassische Beispiele solcher Fehler sind *Race Conditions*, also Fehler, deren Auf-

[1] In diesem Kapitel wird nur an notwendigen Stellen die Unterscheidung zwischen Fehlerursache und Fehlerzustand gemacht (vgl. Abschnitt 3.1), ansonsten vereinfachend von Fehler gesprochen.

treten vom zeitlichen Ablauf der Ausführung nebenläufiger Prozesse abhängig ist. Ein weiteres Beispiel sind Fehler, die abhängig von der Umgebung eines Systems auftreten, beispielsweise von der Anzahl empfangener Nachrichten oder ähnlichen nicht deterministischen Ereignissen. Ferner zählen Fehler zu dieser Kategorie, die von transienten Ausfällen oder Beeinflussungen der Hardware verursacht werden. Ein Teil dieser Fehler wird in der Literatur als *Heisenbugs* bezeichnet. Heisenbugs sind Fehlerursachen, deren Fehlerzustände nicht mehr auftreten, sobald man sie betrachtet bzw. versucht zu beobachten oder zu reproduzieren (vgl. [Gra86]). Heisenbugs sind deshalb besonders schwer zu entdecken und zu entfernen. Ein Großteil der genannten Fehler fällt in die Klasse *konkretisierungsabhängiger Fehler*:

Definition 6.1 (Konkretisierungsabhängige Fehler)
Als konkretisierungsabhängige Fehler *werden Fehlerursachen bezeichnet, bei denen das Auftreten des Fehlerzustandes von der Summe der Entscheidungen abhängig ist, die bei der Konkretisierung der Systemspezifikation vom Management getroffen werden.*

Wie in Kapitel 2 beschrieben, besteht die Konkretisierung der Spezifikation durch das Management aus allen Entscheidungen, die zu treffen sind, um das System auf der gegebenen Hardwarekonfiguration geeignet auszuführen. Dies beinhaltet die Übersetzung des Quellcodes, die Verwaltung der Betriebsmittel und die Steuerung der Ausführung. Informell können konkretisierungsabhängige Fehler als diejenigen Fehlerursachen bezeichnet werden, deren Fehlerzustände bei gleicher Eingabe bei manchen Systemabläufen auftreten, bei anderen nicht. Dies ist wiederum im Nichtdeterminismus der Gesamtausführung eines nebenläufigen Systems begründet, d.h. selbst bei deterministischen Management-Entscheidungen sind Details des Ausführungsablaufs, wie beispielsweise zeitliche Abläufe, nicht fest bestimmt.

Sind Softwarefehler *nicht* konkretisierungsabhängig, so führt für eine bestimmte Eingabe entweder *jede* oder *keine* Kombination aus Managemententscheidungen zum Fehlerzustand. Offensichtlich ist die Existenz nicht konkretisierungsabhängiger Fehler somit leicht zu erkennen und durch übliche Qualitätssicherungs-Maßnahmen der Softwareentwicklung zu vermeiden. Im folgenden Abschnitt werden einige Verfahren zur Qualitätssicherung in der Softwareentwicklung in Hinblick auf ihre Leistung, Fehlerursachen in der Software zu vermeiden, kurz beleuchtet.

6.1.1.1 Qualitätssicherungs-Maßnahmen

Testverfahren. Testverfahren zählen zu den in der Praxis etabliertesten Methoden der Qualitätssicherung. Einerseits werden einzelne Funktionen oder Module, also Teile der gesamten Software auf ihre Funktionalität geprüft, andererseits wird das Gesamtsystem mit Standardeingaben auf die korrekte Verarbeitung getestet. Im Detail lassen sich Testverfahren nach zwei Dimensionen kategorisieren: die Granularität des Testobjekts und die Methode zur Erzeugung des Testfalls, also der Eingabe (vgl. [Wag06]). Heutzutage nimmt die Testphase in der Softwareentwicklung einen beträchtlichen Teil der Zeit und Kosten der Gesamtentwicklung in Anspruch. Nach [Gra86] können mit Testverfahren in erster Linie die als Bohrbugs bezeichneten Fehler entdeckt werden. Da das Auftreten von konkretisierungsabhängigen Fehlern bzw. deren Fehlerzuständen nicht deterministisch ist, bleiben diese mit einer gewissen Wahrscheinlichkeit von den Tests unentdeckt.

Laut einer fünfjährigen Praxisstudie waren beispielsweise auch nach exzessiven Tests noch verschiedenste konkretisierungsabhängige Fehler im *MVS Operating System* von IBM zu finden (vgl. [SC91]). Die begrenzte *Beobachtbarkeit* (vgl. Abschnitt 4.1) und *Kontrollierbarkeit* verteilter, vernetzter Systeme erschwert für diese Softwareklasse zusätzlich das Auffinden von Fehlern durch Testverfahren (vgl. [Hol02]). Grundsätzlich ist der Nutzen von Testverfahren für die Konstruktion fehlerfreier Software eingeschränkt, wie E. W. Dijkstra treffend formulierte: „Testen kann bestenfalls die Anwesenheit von Fehlern, aber niemals ihre Abwesenheit in Programmen zeigen“. Grundsätzlich werden durch Testverfahren Fehlerzustände aufgedeckt, der entscheidende Rückschluss auf die Fehlerursachen bleibt dem Programmierer überlassen und ist keine triviale Aufgabe. In manchen Fällen kann aus diesen Gründen trotz entdeckter Fehlerzustände die wahre Fehlerursache nicht gefunden und damit nicht behoben werden.

Model-Checking. Model-Checking-Verfahren bieten eine Möglichkeit, automatisiert zu überprüfen, ob bestimmte Systemzustände auftreten können. Auch beim Model-Checking werden eigentlich Fehlerzustände gesucht, der Rückschluss zu deren Fehlerursache obliegt wiederum dem Programmierer. Um einen Fehlerzustand mit diesen Verfahren aufzudecken, ist zunächst eine Spezifikation des fehlerhaften Zustands als Eingabe notwendig. Analog zur Programmierung der Software selbst ist hier wiederum der Mensch als Fehlerquelle eines der Hindernisse, die Software von allen Fehlern zu befreien. Einerseits ist es quasi unmöglich, jeden möglichen Fehlerzustand vorherzusehen, um die Möglichkeit seines Auftretens prüfen zu können, andererseits können bei der Spezifikation eines vorhergesehenen Fehlerzustands wiederum Fehler gemacht werden. Durch die Quantifizierung

der Systemzustände können allerdings mittels Model-Checking-Verfahren auch konkretisierungsabhängige Fehler, wie beispielsweise Race Conditions identifiziert werden, da überprüfbar ist, ob ein bestimmter Zustand im System generell auftreten kann oder nicht. Model-Checking-Verfahren haben somit das Potential, auch Fehlerzustände, die auf Grund der Abhängigkeiten und Interaktionen nebenläufiger Prozesse nur sehr selten tatsächlich auftreten, zu entdecken, da die Erreichbarkeit geprüft wird. Der praktische Einsatz ist einerseits durch die Spezifikation dieser Zustände durch den Mensch begrenzt, andererseits ist ein grundsätzliches Problem von Model-Checking-Verfahren deren hoher Rechenaufwand durch zu große Zustandsräume komplexer Software.

Statische Analyse. Statische Analyse-Werkzeuge prüfen die Software bzgl. der Existenz bestimmter Code-Fragmente, die häufig Fehlerursachen enthalten oder als fehlerkritisch eingestuft werden. Der praktische Einsatz ist jedoch begrenzt, da einerseits nur bestimmte Fehlerarten aufgedeckt werden können und andererseits meist sehr viele „false positives“, also potentielle Fehler, die eigentlich keine sind, identifiziert werden. Der Aufwand der Inspektion aller gefundenen möglichen Fehler durch den Programmierer ist in der Regel im Vergleich zum Nutzen sehr hoch. Zwar werden insgesamt auch viele Fehler von diesen Analyse-Werkzeugen entdeckt, meist sind dies jedoch Fehler, die als harmlos oder leicht zu finden einzustufen sind. Fehler mit schwerwiegenden Auswirkungen, wie z.B. ein Systemabsturz, bestehen häufig aus komplexen Abhängigkeiten und können selten von diesen Werkzeugen entdeckt werden. Die Wahrscheinlichkeit, konkretisierungsabhängige Fehler mit diesen Werkzeugen zu entdecken, bleibt vergleichsweise gering. Insgesamt ist die Kombination mit Testverfahren vielversprechend für die Identifikation von Bohrbugs, die Testverfahren für sich sind im Kosten-Nutzen-Verhältnis jedoch günstiger (vgl. [Wag06]).

Verifikation. Verifikation ist der mathematische Beweis der Korrektheit von Software. Die Korrektheit bedeutet allerdings lediglich die Übereinstimmung mit einer gegebenen Spezifikation. Die Verifikation stellt das einzige Mittel dar, das – zumindest in der Theorie – die Abwesenheit jeglicher Fehlerursachen einer Software sicherstellen kann. In der Praxis scheitert der Ansatz der Verifikation allerdings meist an drei Problemen: erstens kann der Beweis selbst fehlerhaft sein, was zu einer falschen Annahme der Fehlerfreiheit führen kann. Zweitens wird nicht bewiesen, dass die Implementierung fehlerfrei ist, sondern lediglich, dass sie der Spezifikation entspricht. Ist die Spezifikation fehlerhaft, so wird wiederum fälschlicherweise angenommen, die Software sei fehlerfrei. Drittens ist der Korrektheitsbeweis auf Grund der hohen Komplexität nur für sehr kleine Programme

in akzeptabler Zeit berechenbar und kann somit für größere Softwareprojekte in der Praxis nicht eingesetzt werden: „Constructing such formal proofs is currently the subject of much active research; however,the state of the art at the present time is rather primitive, and formal program proving is applicable only to small pieces of software. As a result, it is a reasonable assumption that any large piece of software that is currently in use contains defects“ (in [KK07], S. 147). Manche Autoren stellen die Beweisbarkeit interaktiver Programme sogar generell in Frage (vgl. [Weg97]).

Qualitätssicherung generell. Ein allgemeines Problem der Qualitätssicherung des Softwareentwicklungsprozesses ist die Tatsache, dass bei der Behebung eines erkannten Fehlers sehr häufig neue Fehler in die Software integriert werden, die auf Grund des zeitlichen Ablaufs so einen Teil der Qualitätssicherungs-Maßnahmen umgehen konnten und unerkannt bleiben (vgl. [PP05], [SC91]).

Insgesamt zeigt sich, dass Qualitätssicherungs-Maßnahmen die Anzahl verbleibender Fehler im praktischen Einsatz zwar stark reduzieren können, jedoch in komplexer Software, wie in verteilten, nebenläufigen Systemen mit heutigen Methoden immer Fehler zurückbleiben werden. Die Natur dieser Fehler entspricht – nicht ausschließlich, aber sehr häufig – den in dieser Arbeit als konkretisierungsabhängig bezeichneten, da diese auf Grund ihres seltenen, nichtdeterministischen Auftretens besonders schwer mit den eingesetzten Verfahren und Werkzeugen zu identifizieren sind.

Ein weiterer Ansatz zur Verbesserung der Zuverlässigkeit von Software und Hardware ist die Fehlertoleranz. Ziel der Fehlertoleranz ist zu gewährleisten, dass ein System trotz eines aufgetretenen Fehlerzustands korrekt weiterarbeiten kann, ohne ein Versagen auszulösen. Fehlerursachen sollen also nicht vermieden, sondern deren Fehlerzustände toleriert werden. In Kapitel 3 wurde erklärt, dass die Toleranz von Fehlern nur durch Redundanz zu erreichen ist. Ferner wurde erläutert, dass die Toleranz nicht genauer spezifizierter Softwarefehler, wie unter anderem den hier beschriebenen konkretisierungsabhängigen Fehlern, mittels Diversität realisierbar ist. Auf die Diversität wird im Folgenden nochmals eingegangen, um den Ansatz des vorgestellten Konzepts zu motivieren.

6.1.2 Diversität

Der englische Mathematiker Charles Babbage schlug bereits im 19. Jahrhundert vor, Redundanz und Diversität zur Qualitätssicherung zu nutzen. Als er davon sprach, man müsse mehrere Computer benutzen, um die selbe Berechnung durch-

zuführen, und die Ergebnisse vergleichen, waren mit Computern noch Menschen gemeint: „The most certain and effectual check upon errors which arise in the process of computation, is to cause the same computations to be made by seperate and independent computers; and this check is rendered still more decisive if they make their computations by different methods“ (in [Lar61]).

Diversitäre Programmierung verfolgt exakt die Idee von Charles Babbage: Mehrere, verschiedene, funktional identische Versionen der Software sollen erstellt werden, die redundant das Ergebnis berechnen. *N-Version Programming* nutzt dabei strukturelle Redundanz, also parallele Ausführung der Versionen, *Recovery Blocks* dagegen zeitliche Redundanz, also erneute Berechnung mit einer anderen Version der Software (vgl. Abschnitt 3.3.3).

6.1.2.1 Diskussion diversitärer Programmierung

Der Nutzen diversitärer Programmierung basiert auf der Annahme, dass die verschiedenen Versionen unterschiedliche, unabhängige Fehler enthalten. Diese Annahme wurde gestützt durch die Vorgehensweise, unabhängige und räumlich getrennte Programmierteams mit der Aufgabe der Implementierung zu betrauen und Informationsaustausch zwischen diesen Teams zu unterbinden.

Tatsächlich wurde aber in Studien festgestellt, dass gehäuft die selben Fehler oder Fehler identischen Ursprungs gemacht werden, selbst wenn die Programmierer unterschiedlich ausgebildet sind und verschiedene Programmiersprachen benutzen. Es scheint typische logische Fehler und typische Leichtsinnsfehler zu geben, die von verschiedensten Programmierern gemacht werden. Koren und Krishna fassen einige der Gründe für die Abhängigkeit der Fehler zusammen (nach [KK07], Kap. 5.3.2):

- Fehler, die bereits in der Spezifikation der zu entwickelnden Software enthalten sind, propagieren sich in alle Versionen.
- Besonders hohe Komplexität bzw. Schwierigkeit des Problems für bestimmte Eingaben können für viele Programmierer eine höhere Fehlerquote bedeuten.
- Standard-Algorithmen bzw. Bibliotheken, die das Problem nicht für den gesamten Eingabe-Wertebereich lösen, können von mehreren Programmierern verwendet werden.

- Kulturelle Faktoren und Erziehung können typische, identische Denkmuster und Problemlösungsstrategien und damit auch identische Fehlerquellen bei Programmierern hervorrufen.

- Die Verwendung gleicher Soft- und Hardwareplattformen können ebenfalls dort die selben Fehler hervorrufen, wenn auch in diesem Fall Fehler der Plattform, nicht der Anwendung.

In einer Studie wurde darüber hinaus festgestellt, dass die meisten abhängigen Fehler sich nicht durch Änderungen der Werkzeuge oder des Softwareentwicklungsprozesses verhindern lassen: „In most cases, the failures resulted from fundamental flaws in the algorithms that the programmers designed. Thus we do not expect that changing development tools or methods, or any other simple technique, would reduce significantly the incidence of correlated failures in N-Version software.“ ([BKL90] S. 245).

Die Studie basiert auf einer Softwarespezifikation aus der Praxis, die an zwei Universitäten an insgesamt 27 Programmierer unterschiedlichster Ausbildung, darunter Doktoranden und Studenten, zur Implementierung gegeben wurde. Die 27 Versionen der Software wurden anschließend auf Fehler untersucht und verglichen. Es zeigte sich, dass etwa die Hälfte aller gefundenen Fehler in zwei oder mehr Versionen auftraten (vgl. [KL86]). Zudem war festzustellen, dass einige dieser Fehler anwendungsbezogene Denkfehler waren, also nicht einmal abhängig vom gewählten Algorithmus. In dem konkreten Beispiel der Studie handelte es sich um übersehene Fälle beim Vergleich von Winkeln oder den Sinus- und Cosinus-Funktionen dieser Winkel. Auch von der NASA geleitete Studien an amerikanischen Universitäten führten zu vergleichbaren Ergebnissen (vgl. [KK07], S. 169).

Die Abhängigkeit der Fehler in den verschiedenen Software-Versionen reduziert den theoretischen Gewinn an Zuverlässigkeit der diversitären Programmierung drastisch. Zudem ist diversitäre Programmierung in starker Kritik wegen zu hoher Kosten. Die Kosten der Entwicklung einer Software steigen für jede weitere Version um nahezu 100 Prozent, da lediglich die Teile der Softwareentwicklung, die zur Spezifikation führen, für die Implementierung einer weiteren Version wiederverwendet werden können. In der Praxis scheint aus diesen Gründen die diversitäre Programmierung unrentabel, da das Kosten-Nutzen-Verhältnis ungünstig ist (vgl. [Fel98]).

6.1.2.2 Sicherheit durch Diversität

Bevor die Schlussfolgerungen aus den beschriebenen Möglichkeiten der Qualitätssicherung gezogen werden, um den Ansatz dieser Arbeit zu motivieren, werden im Folgenden noch Forschungsarbeiten zur Diversität im Bereich der Sicherheit beleuchtet. Ein erfolgreicher Angriff auf ein Softwaresystem ist als das gezielte Herbeiführen eines Softwarefehlers und das Ausnutzen dieses Fehlers zu betrachten. Ist ein System fehlerfrei, so ist keine Sicherheitslücke vorhanden, die für Angriffe genutzt werden kann.

Das Thema Sicherheit birgt eine weitere Motivation, Diversität einzusetzen. Die Uniformität heutiger Softwaresysteme vereinfacht Angriffe auf Softwaresysteme. Ist einmal eine Sicherheitslücke für eine Kombination aus Betriebssystem und Anwendungssoftware entdeckt und eine Attacke in Form eines Virus, Wurms, Trojaners o.ä. implementiert, so können damit in der Regel enorm viele Rechner weltweit infiltriert werden, da viele die identische Softwarekonfiguration und damit die identischen Sicherheitslücken aufweisen.

Charles Bain beschreibt in einer Studie über vergangene Wurm und Virusattacken des Internet-Zeitalters den Vorteil von Systemen, die vergleichsweise nicht uniform durch verschiedene Softwareversionen waren und schließt, dass Diversität bereits in relativ schwacher Form den entscheidenden Vorteil bewirken kann, dass ein Angriff erfolglos bleibt (vgl. [BFFW01]). E.G. Barrantes argumentiert ebenfalls, dass die Hauptschwachstelle der Sicherheit die Uniformität unserer Systeme ist (vgl. [BAFS05]). Automatisierte Diversität ist ein Ansatz, diese Uniformität mit akzeptablen Kosten zu durchbrechen. Auch S. Forrest fordert randomisierte Diversität zur Erhöhung der Sicherheit als praxistaugliches Mittel: „Such randomization could potentially increase the robustness of software systems with minimal impact on convenience, usability, and efficiency“ ([FSA97], S. 67).

Automatisierte Diversität von Software gezielt als Schutzmechanismus zur Erhöhung der Sicherheit einzusetzen, ist seit einigen Jahren ein viel verfolgter Ansatz. Xu et al. versuchen durch dynamisches, randomisiertes Speicherlayout im Projekt *Transparent Runtime Randomization (TRR)* jegliche Angriffe zu verhindern, die auf „Unauthorized Control Information Tampering (UCIT)“ basieren. Zu dieser Klasse von Angriffen gehören *buffer overflow*, *format string*, *integer overflow* oder *double-freeing* (vgl. [XKI03]). Die zentrale Idee ist, zufallsbasiert das Speicherlayout bestehend aus Halde, Keller, Bibliotheken und Kontrollstrukturen zu verändern, sodass die Speicheradressen dieser Inhalte, die für die genannten Angriffsarten notwendig sind, bei jedem Programmstart anders sind. Können keine Annahmen über die Adressen mehr gemacht werden, so sind die Angriffe

zum Scheitern verurteilt. Die Arbeit wurde an bekannten Angriffsszenarien aus der Praxis getestet und es zeigte sich, dass TRR die Angriffe scheitern lässt.

E. Barrantes, D. Ackley und S. Forrest versuchen, durch eine randomisierte Verschlüsselung der Operationscodes des Instruktionssatzes des Prozessors die Funktionalität von eingeschleustem Code zu verändern (vgl. [BAFS05]). Bei Programmstart wird der gesamte Objektcode mit einem zufällig gewählten Schlüssel verschlüsselt, zur Ausführung muss jeder Befehl entsprechend entschlüsselt werden. Es ist nahezu unmöglich, auch nur wenige korrekt ausführende Operationen ohne Kenntnis über diesen Schlüssel in das Programm zu integrieren. Tests mit bekannten Angriffen aus der Praxis zeigten den Erfolg des Konzepts.

6.1.3 Diversität auf Management-Ebene

Aus den beschriebenen Erfolgen der fehlervermeidenden Verfahren der Softwareentwicklung und den Kritikpunkten diversitärer Programmierung lassen sich folgende Schlüsse zusammenfassen:

- Es verbleiben in der Praxis auch bei exzessivem Einsatz von Qualitätssicherungs-Verfahren Fehler in komplexer Software, insbesondere in nebenläufigen, verteilten Systemen.
- Bei den verbleibenden Fehlern handelt es sich häufig um konkretisierungsabhängige Fehler.
- Das Prinzip der Diversität ist an sich ein viel versprechender Ansatz, diversitäre Programmierung hat in der Praxis allerdings ein schlechteres Kosten-Nutzen-Verhältnis als fehlervermeidende Qualitätssicherungs-Maßnahmen, insbesondere Testverfahren.

Fehlervermeidende Verfahren sind offensichtlich notwendig und kostengünstig, um einen gewissen Grad der Softwarequalität zu erreichen. Um die Qualität der Software in Bezug auf Zuverlässigkeit weiter zu steigern, ist eine Kombination der fehlervermeidenden Verfahren mit fehlertolerierenden Verfahren notwendig: „Consequently, after doing everything possible to reduce the error rate of individual programs, we have to turn to fault-tolerance techniques to mitigate the impact of software defects (bugs)“ (in [KK07], S. 148).

Die aufgezählten Schlüsse motivieren den Ansatz, dieses Ziel zu erreichen, indem Diversität zur Toleranz konkretisierungsabhängiger Fehler auf Management-Ebene automatisiert eingesetzt wird. Konkretisierungsabhängige Fehler zeichnen

sich dadurch aus, dass mit hoher Wahrscheinlichkeit eine andere Kombination aus Management-Entscheidungen nicht zum Auftreten des Fehlers führt. Es ist demnach möglich, einen konkretisierungsabhängigen Softwarefehler zu tolerieren, ohne den Quellcode des Programms zu ändern. Automatisierte Diversität auf Management-Ebene hat zur Aufgabe, eine Kombination aus Management-Entscheidungen zu finden, die einen aufgetretenen Fehler tolerieren. Beobachtungen in diese Richtung wurden auch von Y. Deswarte in seiner Arbeit über die Ebenen von Diversität gemacht: „The same software run on the same hardware but with a different execution context may behave differently with respect to accidental design faults, and this kind of diversity can have a surprising high efficiency.“ ([DKL98], S. 175).

Die hohen Kosten der diversitären Programmierung sind durch die Kosten der zusätzlichen Programmierung begründet. Wird die Diversität automatisiert, so verbessert sich das Kosten-Nutzen-Verhältnis offensichtlich grundlegend. Bezieht man Systemperformanz und Rechenleistung in die Kostenrechnung mit ein, so kann geschlossen werden, dass die Kosten der Integration von Diversität zur Übersetzungszeit geringer sind als bei der Implementierung, und zur Laufzeit nochmals geringer als zur Übersetzungszeit (vgl. [FSA97]). Die Ansätze und Erfolge automatisierter Diversität im Bereich der Sicherheit motivieren zudem die Prinzipien auf das Ziel der Fehlertoleranz zu übertragen, da ein Angriff auf ein System dem gezielten herbeiführen und ausnutzen eines Softwarefehlers entspricht.

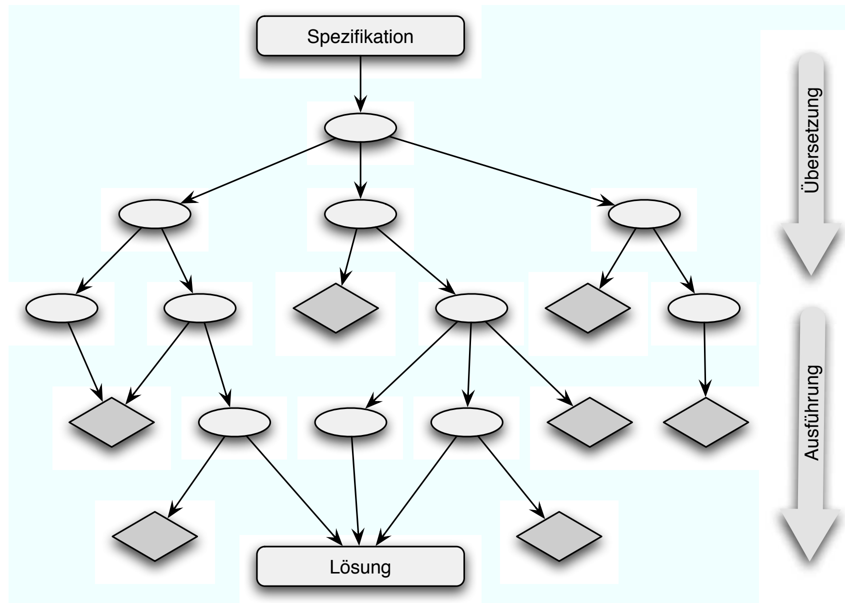


Abbildung 6.1.: Konkretisierungspfade des Managements

Die Kernidee des Ansatzes ist in Abbildung 6.1 veranschaulicht. Ausgehend von der Spezifikation des Systems, einem INSEL-Programm im Quellcode, trifft das Management verschiedene Entscheidungen um die Spezifikation zu konkretisieren und auf der gegebenen Hardwarekonfiguration zur Ausführung zu bringen. Das Ziel ist das korrekte Beenden des Systems mit vorliegender Lösung, in der Abbildung unten dargestellt. Die Entscheidungen, die das Management zu treffen hat, sind zu unterscheiden in Übersetzungszeit und Laufzeit. Diese sind meist, allerdings auf Grund des Binders *FLink* nicht zwingend, zeitlich aufeinander folgend (vgl. Abschnitt 2.3.2).

Enthält die Spezifikation konkretisierungsabhängige Fehlerursachen, so könnte der Pfad möglicher Konkretisierungen vereinfacht wie in Abbildung 6.1 aussehen. Abhängig von den Entscheidungen (elliptisch visualisiert), die das Management zur Übersetzungs- bzw. Ausführungszeit trifft, tritt ein Fehlerzustand auf (als Raute visualisiert) oder die Lösung, also das korrekt terminierte System, wird erreicht. Wäre eine Spezifikation fehlerfrei, so würde jeder mögliche Pfad zur Lösung führen. Ist dagegen ein Fehler in der Spezifikation, der nicht konkretisierungsabhängig ist, so würde jeder Pfad in einen Fehlerzustand führen. Die Idee der automatisierten Diversität auf Management-Ebene besteht darin, erreichte Fehlerzustände zu umgehen, indem durch das Treffen geeigneter Entscheidungen ein Pfad zur Lösung gefunden wird. Verbleiben in der Spezifikation nur konkretisierungsabhängige Fehler, so ist sichergestellt, dass ein solcher Pfad existiert.

Die Schwierigkeit des Ansatzes besteht darin, die richtige Kombination aus Management-Entscheidungen zu finden, da der Entscheidungsraum sowohl zur Übersetzungszeit als auch zur Laufzeit enorm groß ist. Es werden einerseits Methoden benötigt, die die Möglichkeit bieten, Fehler zu erkennen und die Software mit revidierten Management-Entscheidungen erneut auszuführen. Andererseits werden Modelle und Konzepte benötigt, die eine sinnvolle neue Entscheidungskombination finden, und diese algorithmisch im Management durchsetzen können.

Die in Kapitel 4 vorgestellten Mechanismen bieten eine mögliche Basis zur Fehlererkennung für das Toleranzverfahren. Natürlich wären verschiedenste Methoden zur Fehlererkennung dafür geeignet, entscheidend ist lediglich die Fähigkeit, einen fehlerhaften Zustand eines oder mehrerer Prozesse erkennen zu können. Die Fehlerursache muss dabei nicht erkannt werden, es reicht die Fehlersymptome zu protokollieren, um das wiederholte Auftreten des gleichen Fehlers erkennen zu können. In der vorgestellten Arbeit werden Timing-Tests auf Basis der Systembeobachtung (vgl. Abschnitt 4.1) sowie Semantik-Tests in Form der Laufzeitüberprüfung der Verträge (vgl. Abschnitt 4.2) zur Fehlererkennung verwendet.

Um beim Auftreten eines Fehlers die Ausführung des Systems mit einer geänderten, also diversen Kombination von Managemententscheidungen durchführen zu können, bietet sich Rückwärtsbehebung auf Basis eines Checkpoint-Rollback-Verfahrens an. Das in Kapitel 5 vorgestellte kommunikationsinduzierte Checkpoint-Verfahren erlaubt die erneute Ausführung des Systems von verschieden weit in der Vergangenheit liegenden konsistenten Schnitten und ist daher als Basis für das Toleranzverfahren geeignet.

Im folgenden Abschnitt wird ein Konzept vorgestellt, auf Basis dessen für beliebige Systeme die verbleibenden Anforderungen gelöst werden können. Das Konzept beschreibt, auf welcher Basis das Management seine Entscheidungen variieren kann, um ein erneutes Auftreten von Fehlern zu verhindern. Einerseits müssen dazu mögliche Alternativen der Konkretisierung im Management bewertet werden können, und andererseits eine solche Alternative eines Entscheidungsraums generiert und umgesetzt werden. Das Ziel ist ein Verfahren zum automatisierten Finden sinnvoller Kombinationen möglicher Management-Entscheidungen. Das allgemeine Konzept wird anhand eines Beispiels im MoDiS-Projekt veranschaulicht.

6.2 Fehlertoleranz durch Diversität auf Management-Ebene

Die in den vorangehenden Kapiteln beschriebenen Verfahren ermöglichen dem Management, Fehlersituationen zu erkennen und von gesicherten Zuständen die getroffenen Entscheidungen zu revidieren. Welche Konkretisierungs-Entscheidungen das Management eines Systems treffen kann bzw. welche Alternativen in der Übersetzung und Ausführung zur Verfügung stehen, ist Sprach- und Systemabhängig. INSEL legt auf abstraktem Niveau die Eigenschaften und Abhängigkeiten von aktiven und passiven Komponenten fest, die Details zur Konkretisierung bleiben dem Management überlassen, das automatisiert anwendungsorientierte, sinnvolle Entscheidungen für die Realisierung trifft. Sinnvoll bedeutet für das Management in der Regel eine Performanzoptimierung. Tritt ein Fehler auf, so muss das Management zur Toleranz dieses Fehlers sinnvolle Entscheidungen treffen können. Welche Entscheidungen in dieser Situation sinnvoll sind, hängt wiederum vom betrachteten System und der betrachteten Sprache ab.

Einerseits müssen die *Kosten* der Auswirkungen der Entscheidungen berücksichtigt werden, da das allgemeine Ziel der Performanz des Systems nicht außer Acht gelassen werden kann. Auf der anderen Seite muss die Wahrscheinlichkeit,

den aufgetretenen Fehler zu *tolerieren*, also einen Pfad zur Lösung zu finden, mit in Betracht gezogen werden. Diese Wahrscheinlichkeit hängt einerseits von der Wahl der Entscheidungsräume ab, beispielsweise welche der möglichen Entscheidungen revidiert werden. Andererseits können Entscheidungsräume viele Alternativen zur Auswahl stellen, die Wahl einer dieser Alternativen wirkt sich ebenso auf die Wahrscheinlichkeit aus, dass der Fehler erneut auftritt.

Zunächst wird im folgenden erklärt, wie für *einen* Entscheidungsraum automatisierte Entscheidungen zu treffen sind und welche Informationen bei der Systemkonstruktion zu erarbeiten sind. Daraufhin werden Entscheidungsräume eingeführt und erklärt, wie die Entscheidungen für verschiedene Entscheidungsräume zu kombinieren sind. Das erarbeitete Konzept wird am Beispiel des MoDiS-Managements belegt, ist aber allgemein für Systeme mit automatisiertem Management einsetzbar.

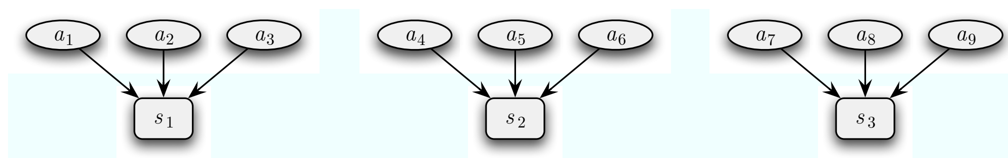


Abbildung 6.2.: Verteilung von Akteuren auf Stellen, Alternative W_1

Als Beispiel für einen Entscheidungsraum dient die Platzierungsentscheidung. Die in MoDiS dynamisch erzeugten Akteure werden vom Management auf die vorhandenen Stellen der Hardwarekonfiguration platziert. Das Management trifft seine Entscheidung nach einer Strategie zur Performanzoptimierung, bei der die zu erwartenden Kommunikationsbeziehungen analysiert und anhand derer kommunikationsintensive Partner zusammen auf Stellen gebunden werden (vgl. [Reh03, Reh04, Reh06]). Die Akteurplatzierung dient im Folgenden als Entscheidungsraum, anhand dessen die Entscheidungsfindung für einen Entscheidungsraum beschrieben wird.

Die möglichen Alternativen dieses Entscheidungsraums sind alle Verteilungen der aktiven Akteure auf vorhandene Stellen. Auf Grund der Möglichkeit der Migration von Akteuren in MoDiS (vgl. [FH04, Reh04]) ist diese Entscheidung sowohl für neu erzeugte als auch für bereits aktive Akteure zu treffen. Die Ausgangssituation stellt sich wie folgt dar: es wurde ein Fehler im System erkannt, der bei einer bekannten Zuweisung von Akteuren auf Stellen aufgetreten ist. Als Beispiel seien neun Akteure auf drei Stellen verteilt, wie in Abbildung 6.2 dargestellt. Nach einem Rollback zum letzten Checkpoint muss nun das Management entscheiden, wie bei erneuter Ausführung die Zuweisung realisiert werden soll.

6.2.1 Varianz

Konkretisierungsabhängige Fehler zeichnen sich dadurch aus, dass bestimmte Systemabläufe, herbeigeführt durch eine Kombination aus Management-Entscheidungen, zu dem fehlerhaften Zustand führen. Der Ansatz der Diversität verfolgt das Ziel, durch alternative Entscheidungen die Systemabläufe, die zu einem Fehler führten, zu umgehen. Es ist nicht bekannt, mit welcher Wahrscheinlichkeit derselbe Fehler in einem alternativen Systemablauf auftritt. Es liegt allerdings nahe, dass sich die Wahrscheinlichkeit für das Auftreten des selben Fehlers reziprok proportional zum Unterschied der Ausführungsabläufe verhält. Je größer also der Unterschied im Ablauf einer erneuten Ausführung des Systems, umso geringer die Wahrscheinlichkeit, dass derselbe konkretisierungsabhängige Fehler der letzten Ausführung nochmals auftritt.

Der Unterschied zweier Ausführungen muss zu diesem Zweck in der *Zeit* und der *Ressourcen-Verwaltung* betrachtet werden. Die Zeit betrifft dabei die Verzahnung der Ausführung der nebenläufigen Prozesse. Bei der Ausführung der abstrakten Komponenten auf der Hardwarekonfiguration können auf Grund der technischen Realisierung Abhängigkeiten und Beschränkungen auftreten, die einerseits zu Fehlern führen können und deshalb hier beachtet werden müssen, von denen andererseits aber in der Sichtweise der INSEL-Komponenten und INSEL-Ereignisse abstrahiert wird. Es handelt sich also nicht um festgelegte, erfasste Abhängigkeiten der Komponentenarten (vgl. Abschnitt 2.2.2), sondern um Abhängigkeiten durch die Konkretisierung auf reale Hardware mit ihren Eigenschaften und Beschränkungen.

Um den Unterschied in der Zeit erfassen zu können, wird er im Folgenden modelliert. Die Modellierung dient jedoch ausschließlich der Begriffserklärung und nicht der tatsächlichen Realisierung. Um zeitlichen Unterschied modellieren zu können, müssen Ausführungsabläufe auf einer feingranularen Ebene betrachtet werden. Die Einheiten dieser feingranularen Ebene werden hier als *Instruktionen* bezeichnet, ohne die Ebene näher festzulegen.

Die Abhängigkeiten, die durch Softwarefehler und technische Realisierungsdetails entstehen können, sind nicht vollständig zu erfassen, deshalb muss die tatsächliche zeitliche Abfolge, wie sie von einer systemweit synchronisierten Uhr aufgezeichnet würde, als Basis des zeitlichen Unterschieds dienen. Auch die tatsächliche zeitliche Abfolge der Instruktionen ist nicht erfassbar, sie dient jedoch zur Modellierung des zeitlichen Unterschieds, der dann letztendlich angenähert werden kann. Jegliche Ordnung durch tatsächliche Abhängigkeiten und Wechselwirkungen sind implizit in der zeitlichen Abfolge enthalten.

Zur Vereinfachung der Modellierung sei die zeitliche Dauer jeder Instruktion gleich, sodass zwei Instruktionen, die auf verschiedenen Stellen ausgeführt werden, entweder exakt gleichzeitig oder nicht überschneidend ausgeführt werden. Eine mögliche Ausführung unter diesen Annahmen ist in Abbildung 6.3 dargestellt. Alle Instruktionen I sind in ein Zeitraster der angenommenen synchronen Uhr eingeordnet, wobei eine Zeile die Instruktionen enthält, die auf einem Prozessor ausgeführt wurden. Die k Instruktionen eines Akteurs $a_m \in A$ sind aufsteigend nummeriert und werden mit $i^1_{a_m}, \ldots, i^k_{a_m}$ bezeichnet. Instruktionen sind übereinander, falls sie parallel auf verschiedenen Stellen ausgeführt wurden. Da es in einer Ausführung auf einer Stelle auf Grund von Synchronisationsbeziehungen auch zu Wartezeiten kommen kann, können auch Einträge im Raster leer sein, da zu diesen Zeitpunkten keine Berechnungen ausgeführt werden.

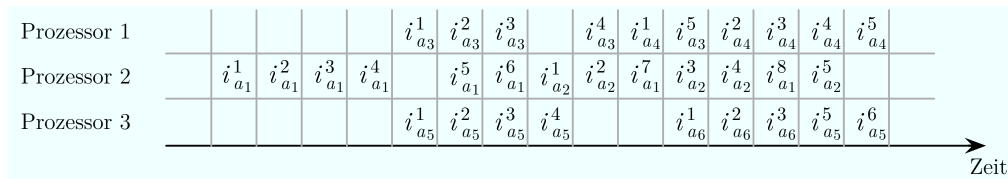


Abbildung 6.3.: Darstellung der Instruktionen im Zeitraster

Der *Unterschied* zweier solcher zeitlicher Ordnungen der Instruktionen kann folgendermaßen modelliert werden: Der Unterschied ist die *minimal* notwendige Anzahl Permutationen, um aus einem Zeitraster das andere zu konstruieren. Eine Permutation sei an dieser Stelle das Vertauschen des Inhalts zweier horizontal oder vertikal im Zeitraster benachbarter Felder (Instruktionen oder freie Felder), beispielsweise das Vertauschen von $i^5_{a_1}$ mit $i^2_{a_5}$. Mit dem Hilfsmittel der Permutationen wird eigentlich der Unterschied des Abstands zweier Instruktionen in den beiden Ausführung bewertet. Dies spiegelt den Ausgangspunkt wider, dass die Wahrscheinlichkeit für Softwarefehler, deren Auftreten von zeitlichen Abläufen abhängt, direkt proportional zu dem Unterschied der Abläufe ist.

Die Ressourcen-Verwaltung bezieht sich auf alle zur Ausführung genutzten Software- und Hardware-Betriebsmittel. Dabei müssen sowohl Unterschiede der Belegung, also Anforderung und Freigabe betrachtet werden, als auch Unterschiede in der Art der Verwendung. Die Menge Speicher, die einem Akteur zur Ausführung zur Verfügung gestellt wird, wäre ein Beispiel für die Belegung, das Speicherlayout, das darin umgesetzt wird, ein Beispiel für die Art der Verwendung. Die Unterschiede der Ressourcen-Verwaltung zweier Ausführungen sind auf Grund der Menge und Verschiedenheit der Ressourcen nicht allgemein modellierbar.

Für den hier vorgestellten Ansatz ist jedoch weder die Berechnung des tatsächlichen Unterschieds in der Zeit, noch die Modellierung des Unterschieds in der Verwaltung der Ressourcen notwendig. Die Auswirkungen verschiedener Alternativen eines Entscheidungsraums müssen in Bezug auf ihre relativen Unterschiede in Zeit und Ressourcen-Verwaltung lediglich abgeschätzt werden. Der *approximierte* Unterschied der Ausführung des Systems in Abhängigkeit zweier verglichener Alternativen einer Entscheidung wird als *Varianz* bezeichnet. Die Varianz ist eine Funktion, die zwei zu vergleichende Alternativen auf einen reellen Zahlenwert abbildet, der den Unterschied der den Alternativen zugehörigen Ausführungen in Zeit und Ressourcen-Verwaltung widerspiegelt:

Definition 6.2 (Varianz)
Sei W die Menge aller wählbaren Alternativen einer Entscheidung, dann ist die Varianz *eine Abbildung $V : W \times W \mapsto \mathbb{R}$, die den Unterschied der Ausführung zweier Alternativen in Zeit und Ressourcen-Verwaltung approximiert.*

Die Varianz ist eine der Entscheidungsgrundlagen für die automatisierte Auswahl einer Alternative, sie repräsentiert die relative, angenommene Wahrscheinlichkeit, dass in einer Alternative ein aufgetretener Fehler nicht nochmals auftritt. Die Varianz ist abhängig vom Entscheidungsraum und ist für jeden Entscheidungsraum zu spezifizieren. Dabei müssen die unterschiedlichen Auswirkungen verschiedener Entscheidungsalternativen auf die Ausführung als Zahlenwerte wiedergegeben werden. Die Qualität der Varianzfunktion ist entscheidend für die Qualität des Fehlertoleranzverfahrens, das darauf aufbaut. Je nach Komplexität des Entscheidungsraums kann das Finden einer guten Varianzfunktion mehr oder weniger aufwändig sein. Die Varianzfunktion ist bei der Systemkonstruktion für jeden Entscheidungsraum einmal festzulegen. Entscheidend ist, dass die Varianz keine Gütekriterien der verschiedenen Alternativen widerspiegeln sollte, sondern lediglich den resultierenden Unterschied in Zeit und Ressourcen-Verwaltung im Ausführungsablauf.

Die Zuweisung von Akteuren auf Stellen ist ein geeignetes Beispiel zur Veranschaulichung der Varianzfunktion. Der Systemablauf unterscheidet sich in der zeitlichen Abfolge, in der Akteure ihre Berechnungen fortführen. Je nach Zuweisung der Akteure auf Stellen können sie schnell (auf einer Stelle) oder langsam (über das Netzwerk) miteinander kommunizieren. Abhängig von den Synchronisationsbeziehungen kann viel oder wenig echt parallel auf verschiedenen Stellen gerechnet werden. Die Stellenzuweisung beeinflusst demnach stark den zeitlichen Ablauf des Systems. Um eine Varianzfunktion festzulegen ist nun folgende Frage ausgehend von einer bestehenden Alternative, also einer Zuweisung aller Akteure

auf Stellen, zu beantworten: Welche Änderung dieser Zuweisung erzeugt mehr Unterschied im Ausführungsablauf?

Für einen Akteur $a \in A$ besteht kein Unterschied in der absoluten Zuweisung, also ob er auf Stelle s_1, s_2 oder s_3 ausgeführt wird, da zur Vereinfachung eine homogene Hardwarekonfiguration angenommen wird. Entscheidend für den Ausführungsablauf ist, welche Akteure mit a zusammen auf der selben Stelle realisiert sind bzw. welche entfernt sind. Bei intensiver Kooperation mit synchroner Kommunikation bedeutet eine Zuweisung auf die gleiche Stelle einen Performanzvorteil, da die Kommunikation wesentlich effizienter durchgeführt werden kann. Für wenig kommunizierende Akteure ist auf Grund der hohen potentiellen Parallelität eine Zuweisung auf verschiedene Stellen effizient.

Für jedes beliebige Paar von Akteuren ist somit der Unterschied in der Ausführung davon abhängig, ob sie der gleichen Stelle oder zwei verschiedenen Stellen zugewiesen werden. Je nach Kommunikationsverhalten wirkt sich diese Entscheidung positiv oder negativ auf die Performanz des Systems aus. Für die Varianz ist jedoch lediglich entscheidend, dass der Systemablauf davon beeinflusst wird, ob zwei Akteure auf einer Stelle oder auf zwei verschiedenen Stellen realisiert sind. Eine Varianzfunktion müsste demnach für jedes beliebige Paar von Akteuren einen höheren Wert erzeugen, falls deren Zuweisung in den beiden verglichenen Alternativen unterschiedlich ist. Unterschiedlich bedeutet hier, dass sie in einer der beiden Alternativen auf der gleichen Stelle, in der anderen auf zwei verschiedenen Stellen realisiert sind. Verschiedene Alternativen dieser Entscheidung sind Zuweisungen aller Akteure $a \in A$ auf die vorhandenen Stellen $s \in S$. In diesem Fall ist die Menge W der Alternativen also eine Menge von Abbildungen $W_i \in W$ der Signatur

$$W_i : A \mapsto S.$$

Der erklärte Unterschied zweier Alternativen $W_i, W_j \in W$ für zwei Akteure $a_m, a_n \in A$, also zweier Abbildungen der Akteure auf Stellen kann mit einer Hilfsfunktion auf die Wertemenge $\{1, 0\}$ abgebildet werden:

$$U : W \times W \times A \times A \mapsto \{0, 1\}$$

$$U(W_i, W_j, a_m, a_n) = \begin{cases} 1 & \text{falls } (W_i(a_m) = W_i(a_n) \wedge W_j(a_m) \neq W_j(a_n)) \vee \\ & (W_i(a_m) \neq W_i(a_n) \wedge W_j(a_m) = W_j(a_n)) \\ 0 & \text{sonst.} \end{cases}$$

Die beschriebenen Überlegungen lassen sich mit Hilfe der eingeführten Hilfsfunktion zu einer Varianzfunktion für die Stellenzuweisung $V_S : W \times W \mapsto I\!R$ zusammenfassen:

$$V_S(W_i, W_j) = \sum_{\forall a_m, a_n \in A} U(W_i, W_j, a_m, a_n)$$

Die Varianz V_S erfüllt die genannten Vorüberlegungen: Je mehr Akteure von einer Alternative zur anderen *unterschiedlich*, also einmal performant auf dem selben Rechner, und einmal langsam via Netzwerk kommunizieren müssen, umso höher der Wert der Varianz.

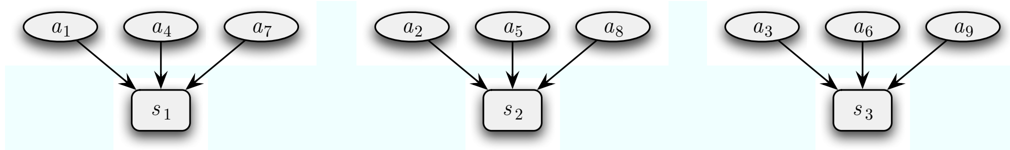


Abbildung 6.4.: Verteilung der Akteure auf Stellen, Alternative W_2

Ausgehend von der Stellenzuweisung in Abbildung 6.2 kann nun der Wert der Varianzfunktion zu der Alternative in Abbildung 6.4 berechnet werden. In Abbildung 6.4 sind die Akteure $a_1, \ldots, a_9$ ebenfalls gleichmäßig auf die Stellen $s_1, \ldots, s_3$ verteilt, allerdings sind die Kombinationen der Akteure so permutiert, dass jeder Akteur mit zwei im Vergleich zur ersten Alternative anderen Akteuren kombiniert auf einer Stelle ist. Die Varianzfunktion $V_S(W_1, W_2)$ ergibt für den Vergleich der beiden visualisierten Alternativen W_1 und W_2 den Wert 18. Werden im Vergleich zu W_1 beispielsweise nur 2 Akteure vertauscht, so hat V_S nur den Wert 8. Dies spiegelt die Tatsache wieder, dass sich bei W_2 die Ausführung stärker ändert, da bei Alternative W_2 jegliche Kommunikation, die bei Alternative W_1 lokal auf einem Rechner stattfinden konnte, über das Netzwerk übertragen werden muss.

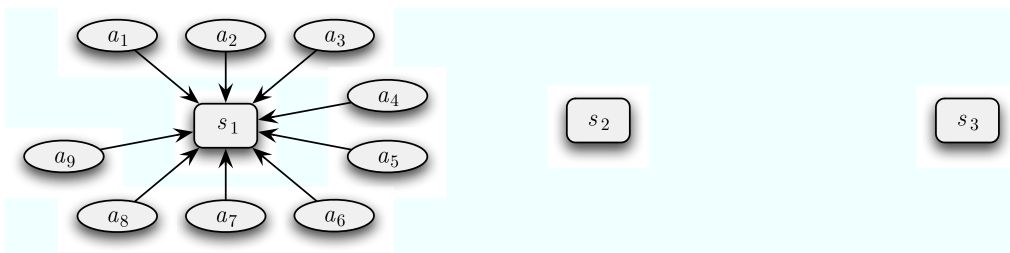


Abbildung 6.5.: Verteilung der Akteure auf Stellen, Alternative W_3

Im Vergleich mit Alternative W_1 realisiert Alternative W_3 in Abbildung 6.5 die maximale Varianz. Die Varianzfunktion $V_S(W_1, W_3)$ ergibt einen Wert von 27. Bei Alternative W_3 sind alle Akteure auf eine Stelle gebunden, sodass keinerlei Kommunikation über das Netzwerk mehr stattfindet. Der Unterschied in der Ausführung im Vergleich zu W_1 wird hier noch höher bewertet als bei W_2, da jeder Akteur zwar mit den gleichen beiden Akteuren performant kommunizieren kann wie zuvor, jedoch mit den sechs anderen Akteuren des Systems zusätzlich.

Die Varianzfunktion für den Entscheidungsraum der Stellenzuweisung zeigt, dass es keineswegs trivial ist, eine geeignete Varianzfunktion zu erstellen. Einerseits sind die Überlegungen, wie sehr die verschiedenen Alternativen die Ausführung in Zeit und Ressourcen-Verwaltung beeinflussen, zum größten Teil intuitiv und beruhen auf Beobachtungen, Annahmen und Erfahrung des Programmierers, nicht auf einem mathematischen Beweis. Andererseits muss ein Kompromiss zwischen Komplexität der Varianzfunktion und ihrer Güte getroffen werden.

Für einen Entscheidungsraum können verschiedene Varianzfunktionen gefunden werden. Das hier gezeigte Beispiel soll einen Eindruck vermitteln, wie eine einfach zu berechnende Funktion die Anforderung erfüllt, den Unterschied in der Ausführung zweier Alternativen abzubilden. Die Varianz ist der erste Bestandteil der Bewertungsfunktion von Alternativen, die das Management zur Variation von Konkretisierungsentscheidungen nutzt.

6.2.2 Kosten

Die Varianz gibt einen Hinweis auf die erwartete Wahrscheinlichkeit, dass ein aufgetretener Fehler bei einer anderen Alternative nicht erneut auftritt. Um eine geeignete Strategie bei der automatisierten Entscheidungsfindung verwenden zu können, müssen zudem die Kosten verschiedener Alternativen berücksichtigt werden. Die Kosten der Ausführung werden in der Regel an der Performanz gemessen, also an der gesamten Ausführungszeit des Systems. Je nach Entscheidungsraum kann der Zusammenhang zwischen Alternativenwahl und Auswirkungen auf die Systemperformanz direkt und offensichtlich oder indirekt und komplex sein. Sind keine Auswirkungen der Alternativen auf die Systemperformanz abzuschätzen, so können andere Kriterien, wie beispielsweise der Ressourcenverbrauch als Kostenbewertung herangezogen werden. Analog zur Varianz müssen auch die Kosten nicht absolut bewertet werden, sondern lediglich verschiedene Alternativen relativ zueinander verglichen werden. Für die Entscheidungsfindung ist eine Kostenfunktion anzugeben, die den erwarteten, relativen Unterschied in den Auswirkungen

auf die Systemperformanz bzw. den Ressourcenverbrauch der Ausführung zweier Alternativen in Form einer reellen Zahl wiedergibt.

Definition 6.3 (Kostenfunktion)
Sei W die Menge aller wählbaren Alternativen einer Entscheidung, dann ist die Kostenfunktion *eine Abbildung $K : W \times W \mapsto \mathbb{R}$, die den relativen Unterschied der Ausführung zweier Alternativen in Systemperformanz und Ressourcenverbrauch angibt.*

Die Kostenfunktion ist analog zur Varianz für jeden Entscheidungsraum von Hand zu erstellen. Im Gegensatz zu den schwer abzuschätzenden Auswirkungen verschiedener Alternativen in den Unterschieden der Ausführungsabläufe kann in der Regel die Kostenfunktion genauer abgeschätzt werden. Für manche Entscheidungsräume sind die Auswirkungen auf die System-Performanz bekannt oder praxisnah abschätzbar. Die Kosten einer Entscheidung können sich aus verschiedenen Teilen zusammensetzen, wodurch eine Kostenfunktion komplex werden kann. Beispielsweise müssen häufig Kosten für das Ändern einer früheren, bereits bestehenden Entscheidung zusätzlich berücksichtigt werden. Am Beispiel der Stellenzuweisung sind diese Probleme veranschaulicht.

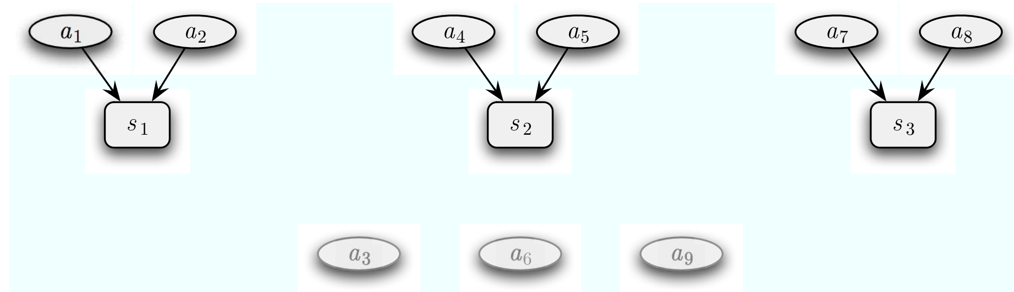


Abbildung 6.6.: Verteilung der Akteure des gespeicherten Zustands zum Zeitpunkt der Entscheidung

In die Kostenfunktion der Stellenzuweisung K_S müssen die zu erwartenden Performanzunterschiede der Ausführung einfließen. In diesem Fall bedeutet das den Unterschied der Ausführungszeit zweier alternativer Verteilungen der Akteure auf Stellen des Systems ab einem konsistenten Schnitt. Zusätzlich müssen allerdings die Kosten berücksichtigt werden, die ausgehend von der im konsistenten Schnitt ausgeführten Alternative durch das Herbeiführen der neuen Alternative entstehen.

Bei der Stellenzuweisung bedeutet das, dass alle Akteure, die im gespeicherten Systemzustand des Schnitts bereits erzeugt und damit einer Stelle zugewiesen sind, entsprechend der neuen Alternative migriert werden müssen. Die Kosten dieser Akteur-Migrationen müssen in der Kostenfunktion berücksichtigt werden. Angenommen bei Ausführung von Alternative W_1 aus Abbildung 6.2 sei ein Fehler aufgetreten. Der konsistente Schnitt, von dem aus eine neue Entscheidung für die Stellenzuweisung zu treffen ist, speichere den in Abbildung 6.6 visualisierten Zustand. Die hellgrau angedeuteten Akteure a_3, a_6 und a_9 seien in dem Zustand noch nicht erzeugt und damit noch keiner Stelle zugewiesen. Die restlichen Akteure $a_1, a_2, a_4, a_5, a_7, a_8$ seien in dem Zustand bereits wie abgebildet auf Stellen verteilt.

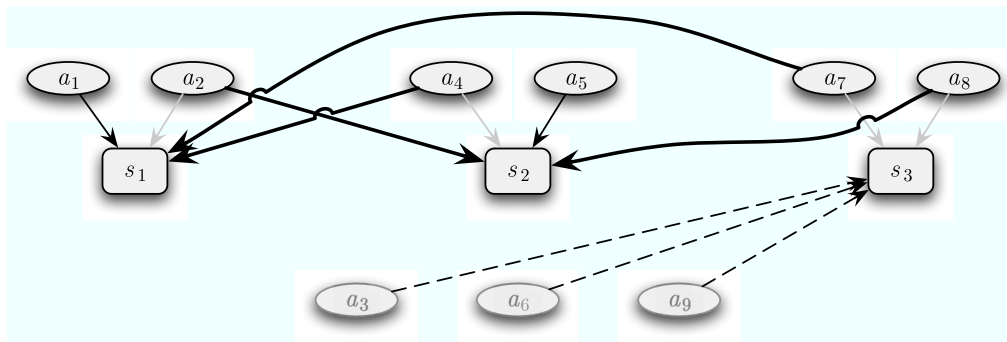


Abbildung 6.7.: Umsetzung von Alternative W_2, ausgehend vom gespeicherten Zustand

Angenommen von diesem Zustand aus wird die Entscheidung getroffen, Alternative W_2 aus Abbildung 6.4 zu realisieren. In diesem Fall müssten die alten Bindungen der Akteure a_2, a_4, a_7 und a_8 aufgelöst werden und die Akteure durch Migration an andere Stellen gebunden werden. In Abbildung 6.7 ist dieser Fall visualisiert, die alten aufzulösenden Bindungen der vier Akteure sind hellgrau dargestellt, ihre neuen Zuweisungen durch dickere schwarze Pfeile markiert. Die im gespeicherten Zustand noch nicht existierenden Akteure a_3, a_6 und a_9 müssen lediglich zum Zeitpunkt ihrer Erzeugung den entsprechenden Stellen zugewiesen werden, in der Abbildung mit gestrichelten Pfeilen dargestellt. In diesem Szenario ist demnach in der Kostenfunktion dem Aufwand für vier Akteur-Migrationen Rechnung zu tragen. Diese Kosten müssen zusammen mit den geänderten Ausführungskosten, die auf Grund der potentiellen Parallelität der neuen Stellenzuweisung existieren, berücksichtigt werden.

Folgend wird eine Abschätzung dieser Kosten nach der Arbeit von C. Rehn vorgestellt [Reh03, Reh04, Reh06], die zur Platzierung eines neu erzeugten Ak-

teurs in eine gegebene Konfiguration dient. Anschließend wird auf Basis dieser Überlegungen eine Kostenfunktion für das hier vorliegende Problem erarbeitet.

6.2.2.1 Stellenzuweisung und Systemperformanz

Die Performanz einer Systemausführung, also die Laufzeit des verteilten Systems, ist von der Zuweisung von Akteuren auf Stellen abhängig, da diese Einfluss auf die parallele Ausführung der Anwendung hat. Sind die Berechnungen der verschiedenen Akteure unabhängig, insbesondere nicht synchronisiert, so kann bei beliebiger Stellenzuweisung stets auf jeder Stelle die Berechnung voranschreiten. In den betrachteten, kooperativen Systemen bestehen in der Regel Abhängigkeiten zwischen Akteuren, die auf Grund der notwendigen Synchronisation zu Wartezeiten führen können.

Ist die Kooperation zwischen zwei Akteuren a_m und a_n intensiv, so kann der Fall auftreten, dass die Laufzeit dieser beiden Akteure kürzer ist, falls diese zusammen einer Stelle zugewiesen werden, als wenn jeder einer einzelnen Stelle zugewiesen wird. Dieser Umstand kommt dadurch zu Stande, dass einerseits bei Kommunikation über ein Netzwerk die Wartezeiten für den Informationsaustausch wesentlich höher sind, und andererseits auf Grund häufiger Synchronisation unter Umständen die beiden Akteure nur wenig echt parallel rechnen, sondern auf Zwischenergebnisse des jeweils anderen warten. Je nach Kombination von Akteuren auf Stellen kann während der Wartezeit eines Akteurs ein anderer Akteur die Rechenfähigkeit der Stelle nutzen, sodass kein Verlust für die Gesamtperformanz entstehen muss. Die tatsächlichen Zusammenhänge und Auswirkungen der Stellenzuweisungen auf die Performanz sind komplex und hängen von der einzelnen Anwendung und den Synchronisationsbeziehungen deren Akteure ab.

Die Systemperformanz ist messbar als die Gesamtlaufzeit des Systems. Um sich die komplexen Auswirkungen der Stellenzuweisung (bzw. Akteurplatzierung) durch Synchronisationsabhängigkeiten auf die Systemperformanz zu veranschaulichen, dient ein Beispielsystem mit drei Akteuren a_1, a_2 und a_3, die auf zwei Stellen zur Ausführung gebracht werden sollen. Zur Beschreibung des Systems ausgehend vom Ereignisverband (vgl. Kapitel 2) werden alle nebenläufigen Ereignisse zusammengefasst und Knoten- und Kantengewichte eingeführt, sodass ein *Attributierter Nebenläufigkeitsverband (ANV)* erzeugt wird (vgl. Kapitel 2.3.1). Alle Ereignisse der MoDiS-Konzepte wurden den Klassen `SYNCER`, `SYNCEE` und `NOSYNC` zugeordnet, nebenläufige Ereignisse wurden zusammengefasst, wie in Abbildung 6.8 dargestellt.

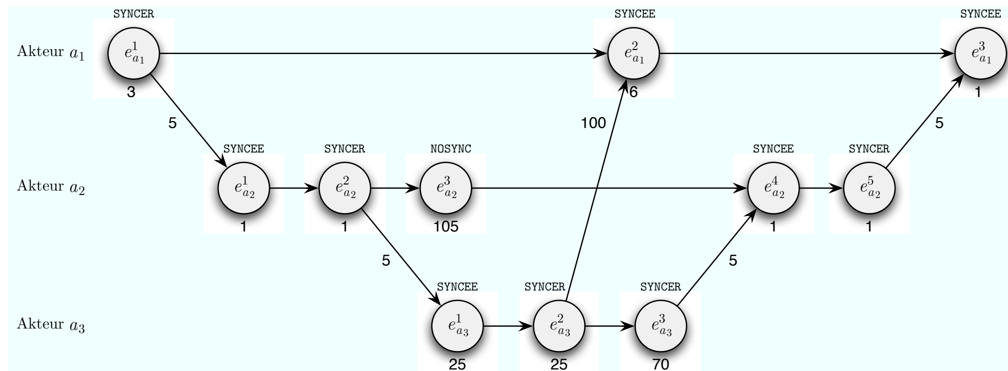


Abbildung 6.8.: ANV eines Beispielsystems

Den Knoten zugeordnete Zahlen sind approximierte, relative Berechnungszeiten der Akteure für diese Ereignisse, wie beispielsweise 3 für das Ereignis $e^1_{a_1}$. Den Synchronisationskanten zugeordnete Zahlen stellen die Zeit dar, die nach dem `SYNCER`-Ereignis vergehen muss, bis das `SYNCEE`-Ereignis eintreten kann, falls die Kommunikation über das Netzwerk stattfindet. Beispielsweise kann die Berechnung von Ereignis $e^2_{a_1}$ erst 100 Zeiteinheiten nach Ereignis $e^2_{a_3}$ beginnen. Die unterschiedliche Behandlung dieser Synchronisationskanten entspricht dem Unterschied der schnellen Kommunikation auf einer Stelle im Vergleich zum langsamen Netzwerk. Je nachdem, ob während dieser Wartezeiten andere Akteure auf der Stelle rechnen können, wirkt sich die Wartezeit negativ auf die Systemlaufzeit aus oder nicht. Dies kann anhand verschiedener Stellenzuweisungen dieser drei Akteure auf zwei Stellen mittels Gantt-Diagrammen veranschaulicht werden.

In Abbildung 6.9(a) ist das Gantt-Diagramm[2] für den Systemablauf dargestellt, falls alle drei Akteure einer Stelle zugewiesen werden. Es ergeben sich natürlich keine durch Netzwerkübertragung bedingten Wartezeiten, die Kantengewichte des Beispielgraphen haben alle den Wert 0, allerdings kann auch nichts parallel berechnet werden, sodass eine Gesamtlaufzeit von 239 Zeiteinheiten abzulesen ist, was der Summe aller Knotengewichte entspricht.

Im Vergleich dazu sind in Abbildungen 6.9(b), 6.10(a) und 6.10(b) die drei weiteren möglichen Zuweisungen der drei Akteure auf zwei Stellen dargestellt. Das Beispiel ist so konstruiert, dass die Auswirkungen verschiedener Platzierun-

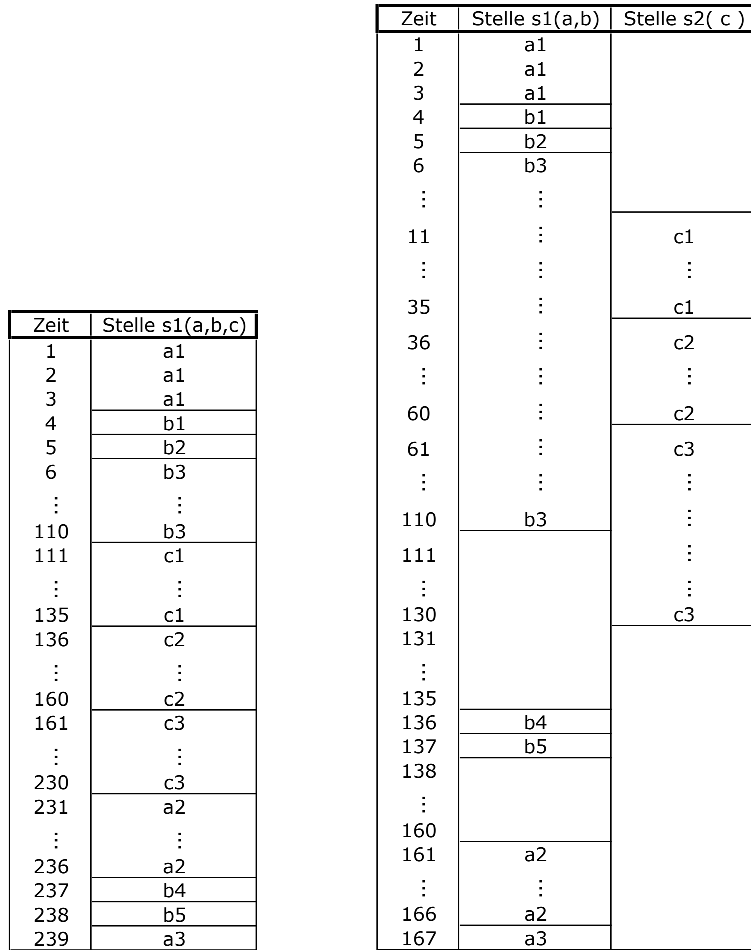
[2]Für bessere Unterscheidbarkeit zwischen Akteur- und Ereigniswechsel sind in den Gantt-Diagrammen der Abbildungen 6.9(a), 6.9(b), 6.10(a) und 6.10(b) entgegen der sonstigen Bezeichner die Akteure a_1, a_2 und a_3 mit den Buchstaben a, b und c bezeichnet. Die Ereignisse $e^1_{a_1} \dots e^3_{a_1}$ werden analog als $a1 \dots a3$, $e^1_{a_2} \dots e^5_{a_2}$ als $b1 \dots b5$ und $e^1_{a_3} \dots e^3_{a_3}$ als $c1 \dots c3$ bezeichnet.

gen auf die Gesamtlaufzeit gut zu erkennen sind. Betrachtet man den Fall der Kombination der Akteure a_1 und a_3 auf Stelle s_1 und a_2 auf Stelle s_2, so wird die kürzeste Systemlaufzeit mit 148 Zeiteinheiten erreicht, da die beiden Rechenintensiven Akteure a_2 und a_3 große Teile ihrer Berechnungen parallel ausführen können und zudem die über ein Netzwerk aufwändige Kommunikation zwischen a_3 und a_1 effizient erfolgen kann. Im Vergleich dazu kann bei einer Kombination von a_3 und a_2 auf eine Stelle wenig parallel gerechnet werden und das System terminiert auf Grund der langsamen Kommunikation erst nach 243 Zeiteinheiten, wie in Abbildung 6.10(a) dargestellt. Bei dieser Stellenzuweisung wäre die Systemlaufzeit länger als bei einer Zuweisung aller Akteure auf eine einzige Stelle. Es ist demnach für eine geeignete Abschätzung des Einflusses der Stellenzuweisung auf die Systemperformanz unabdingbar, die Wartezeiten durch langsame Kommunikation sowie die effektive Parallelität zu berücksichtigen.

Die Platzierungsentscheidung im Management zum Zeitpunkt einer Akteurerzeugung muss den neuen Akteur mit dem Ziel der Performanzoptimierung einer bestimmten Stelle zuweisen. Die Problemstellung ist eine ähnliche wie die in dieser Arbeit für die Kostenfunktion dargestellte, unterscheidet sich jedoch in zwei wichtigen Punkten:

Einerseits muss die Platzierungsentscheidung für einen neu erzeugten Akteur im normalen Systemablauf unter Umständen sehr häufig getroffen werden, sodass ein sehr effizientes Verfahren zur Entscheidungsfindung notwendig ist. Die Kostenfunktion für die Alternativenwahl des Fehlertoleranzverfahrens kann zwar unter Umständen auch häufiger aufgerufen werden, um viele Alternativen zu vergleichen, dies findet jedoch nur im seltenen Fall statt, dass bereits ein Fehler im System aufgetreten ist. In diesem Fall ist eine aufwändigere Entscheidungsfindung eher akzeptabel als im Normalbetrieb des Systems.

Andererseits kann ein induktives Verfahren angewendet werden, um die verschiedenen Möglichkeiten der Platzierung zu bewerten. Für eine Entscheidung sind für das Verfahren alle Akteure fest an Stellen gebunden, lediglich die Auswirkungen des Bindens des neu erzeugten Akteurs muss evaluiert werden. Dies ermöglicht eine Reduktion des Problems auf ein Abschätzen des Performanzgewinns bzw. Performanzverlusts durch Rechen- und Synchronisationszeit des neu erzeugten Akteurs in Bezug zu allen anderen Akteuren für jede Stelle, auf die er platziert werden könnte. Die zukünftige Entwicklung des Systems wird dabei auf Grund der Komplexität nur begrenzt betrachtet: Akteure, die in der Zukunft noch erzeugt werden, fließen nicht in die Platzierungs-Entscheidung des aktuell zu erzeugenden Akteurs mit ein. Es handelt sich also um ein *Greedy*-Verfahren, das keine global optimale, sondern eine für den betrachteten Zeitpunkt optimale Lösung erarbeitet. Die Performanz völlig neuer Permutationen der Stellenzuwei-



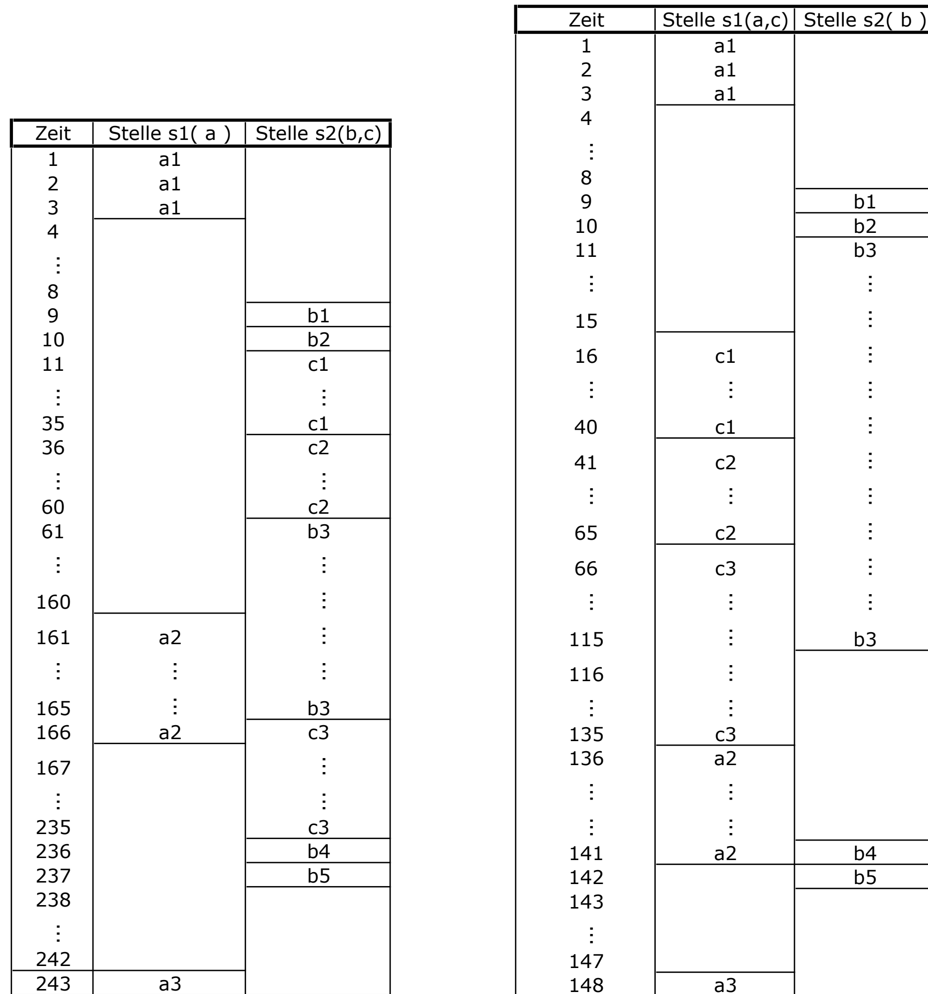
Zeit	Stelle s1(a,b,c)
1	a1
2	a1
3	a1
4	b1
5	b2
6	b3
⋮	⋮
110	b3
111	c1
⋮	⋮
135	c1
136	c2
⋮	⋮
160	c2
161	c3
⋮	⋮
230	c3
231	a2
⋮	⋮
236	a2
237	b4
238	b5
239	a3

(a) Kombination von a_1, a_2, a_3 auf einer Stelle

Zeit	Stelle s1(a,b)	Stelle s2(c)
1	a1	
2	a1	
3	a1	
4	b1	
5	b2	
6	b3	
⋮	⋮	
11	⋮	c1
⋮	⋮	⋮
35	⋮	c1
36	⋮	c2
⋮	⋮	⋮
60	⋮	c2
61	⋮	c3
⋮	⋮	⋮
110	b3	⋮
111		⋮
⋮		⋮
130		c3
131		
⋮		
135		
136	b4	
137	b5	
138		
⋮		
160		
161	a2	
⋮	⋮	
166	a2	
167	a3	

(b) Kombination von a_1, a_2 auf Stelle s_1, a_3 auf s_2

Abbildung 6.9.: Gantt-Diagramme von zwei möglichen Zuweisungen der Akteure auf Stellen[2]



Zeit	Stelle s1(a)	Stelle s2(b,c)
1	a1	
2	a1	
3	a1	
4		
⋮		
8		
9		b1
10		b2
11		c1
⋮		⋮
35		c1
36		c2
⋮		⋮
60		c2
61		b3
⋮		⋮
160		⋮
161	a2	⋮
⋮	⋮	⋮
165	⋮	b3
166	a2	c3
167		⋮
⋮		⋮
235		c3
236		b4
237		b5
238		
⋮		
242		
243	a3	

(a) Kombination von a_2, a_3 auf Stelle s_2, a_1 auf s_1

Zeit	Stelle s1(a,c)	Stelle s2(b)
1	a1	
2	a1	
3	a1	
4		
⋮		
8		
9		b1
10		b2
11		b3
⋮		⋮
15		⋮
16	c1	⋮
⋮	⋮	⋮
40	c1	⋮
41	c2	⋮
⋮	⋮	⋮
65	c2	⋮
66	c3	⋮
⋮	⋮	⋮
115	⋮	b3
116	⋮	
⋮	⋮	
135	c3	
136	a2	
⋮	⋮	
⋮	⋮	
141	a2	b4
142		b5
143		
⋮		
147		
148	a3	

(b) Kombination von a_1, a_3 auf Stelle s_1, a_2 auf s_2

Abbildung 6.10.: Gantt-Diagramme von zwei möglichen Zuweisungen der Akteure auf Stellen[1]

sung, wie sie in der vorliegenden Arbeit zu bewerten sind, können nicht mit einem induktiven Verfahren abgeschätzt werden. Dennoch kann ein Teil der Ansätze übernommen werden, deshalb wird das Verfahren im Folgenden knapp erläutert.

Die *effektive Parallelität* der Berechnungen des neu erzeugten Akteurs wird im Management abgeschätzt:

Definition 6.4 (Effektive Parallelität)
Wird auf einer Stelle s eine Menge A_s von Akteuren ausgeführt, so ist die effektive Parallelität (eP) *eines Akteurs $a_m \notin A_s$ durch die Summe über die Zeitabschnitte bestimmt, in denen sowohl Ereignisse des Akteurs a_m als auch Ereignisse eines Akteurs $a_n \in A_s$ parallel ausgeführt werden könnten, wenn a_m auf einer freien (fiktiven) Stelle s' platziert würde.*

Die effektive Parallelität kann weiter nach der Genauigkeit des Verfahrens unterschieden werden, je nach Güte der Approximationen der Ausführungszeiten für den verteilten oder lokalen Fall. Für die Wahl eines dieser Verfahren ist ein Kompromiss zwischen Genauigkeit der Approximation und dem zu leistenden Aufwand zu treffen. Neben dem Performanzverlust einer Alternative durch weniger effektive Parallelität, durch eP_{W_i} abgekürzt, muss der Performanzgewinn auf Grund gesparter Wartezeiten bei Kommunikation im entfernten Fall berücksichtigt werden, die als Synchronisationskosten SK_{W_i} in die Rechnung eingehen. Der Quotient dieser beiden Größen dient als Basis für die Entscheidung des Managements, auf welche Stelle ein neu erzeugter Akteur initial zu platzieren ist. Die Performanz jeder Alternative W_i, also jeder möglichen Platzierung des neuen Akteurs auf einer Stelle wird mit dem Quotient abgeschätzt:

$$B_{en}(W_i) = \frac{\text{Gewinn bei Platzierung auf Stelle } s_i}{\text{Verlust bei Platzierung auf Stelle } s_i} = \frac{SK_{W_i}}{eP_{W_i}}$$

Unter Berücksichtigung von Sonderfällen mit $SK_{W_i} = 0$ oder $eP_{W_i} = 0$ kann die Alternative mit dem größten Wert der Funktion B_{en} gewählt werden. Die Approximationen der effektiven Parallelität und der Kommunikationskosten basieren auf der Vorhersage von Kommunikationsverhalten und Rechenlast durch zur Übersetzungszeit gesammelte Informationen und werden als ANV modelliert.

6.2.2.2 Kosten verschiedener Stellenzuweisungen

Auf Basis der ANV können die Systemlaufzeiten verschiedener Stellenzuweisungen prognostiziert werden, wie in den Gantt-Diagrammen dargestellt. Im Fall eines Fehlers wird die Entscheidung der Stellenzuweisung ausgehend von einem

gesicherten, konsistenten Schnitt getroffen, von dem aus das System die Berechnungen fortsetzt. Der ANV des Systems muss demnach für diese Situation, also den konsistenten Schnitt angepasst werden, sodass lediglich die Entwicklung des Systems ab dem gesicherten Zustand in die Entscheidung einfließen. Die bis zum gesicherten Zustand bereits abgeschlossenen Berechnungen müssen ausgeblendet werden. Visualisiert bedeutet dies, dass die Kanten- und Knotengewichte des den ANV repräsentierenden azyklischen, gerichteten Graphen bei allen bereits abgeschlossenen Ereignissen auf Null gesetzt werden. Zur Vereinfachung können diese vergangenen Berechnungen dann zu einem Ereignis zusammengefasst werden, von dem aus Kanten zu den Ereignissen folgen, die in jedem Akteur direkt nach dem gespeicherten Zustand folgen.

Angenommen, das in Abbildung 6.8 dargestellte System wird wie in Gantt-Diagramm 6.9(b) dargestellt, also mit der Stellenzuweisung $s_1(a_1, a_2)$, $s2(a_3)$ ausgeführt und es tritt ein Fehler auf. Ferner angenommen, es existieren Checkpoints, die den konsistenten Zustand speichern, in dem die Ereignisse $e^1_{a_1}$, $e^1_{a_2}$ und $e^2_{a_2}$ bereits vollständig enthalten sind, sowie das Ereignis $e^1_{a_3}$ bereits 1 seiner 25 Berechnungseinheiten, und $e^3_{a_2}$ bereits 6 seiner 105 Berechnungseinheiten abgearbeitet haben. Dieser Zustand entspricht dem in Gantt-Diagramm 6.9(b) in der Zeile mit Zeiteinheit 11 dargestellten Ausführungsfortschritt. Der ANV, der zur Entscheidung betrachtet werden muss, entspricht demnach dem Graph in Abbildung 6.11.

Ausgehend von einem neu eingeführten Startereignis e_x kann jeder Akteur seine Berechnungen ab dem gespeicherten Zustand fortführen. Die Berechnungszeiten der Ereignisse $e^1_{a_3}$ und $e^3_{a_2}$, deren Berechnungen zum Checkpoint-Zeitpunkt bereits begonnen hatten, wurden dementsprechend angepasst. Hier sei angemerkt, dass im ANV ein Ereignis eigentlich einer Zusammenfassung nebenläufiger Ereignisse entspricht und es somit keine atomare Einheit darstellt. Die Ausführungszeit für das Wiederherstellen des gespeicherten Zustands könnte in Ereignis e_x modelliert werden, wird hier jedoch zur Einfachheit mit 0 gewichtet, da diese für den Vergleich der Stellenzuweisungen keine Rolle spielt. In der Darstellung ist das Ereignis e_x keinem der Akteure zugeordnet, da das Ereignis für alle bereits aktiven Akteure den gespeicherten Zustand modelliert, an dem die Berechnungen fortsetzen.

Die Migrationskosten können in diese Betrachtung integriert werden, da die Migration einer Datenübertragung über das Netzwerk entspricht, also modellierbar als zusätzliches Kantengewicht der Kante zum nächsten auszuführenden Ereignis. Dieses Vorgehen begründet sich in folgender Überlegung: Angenommen ein Akteur $a \in A$ muss migriert werden, um die gewünschte Stellenzuweisung zu erreichen. Können während der Übertragung des Akteurs über das Netzwerk,

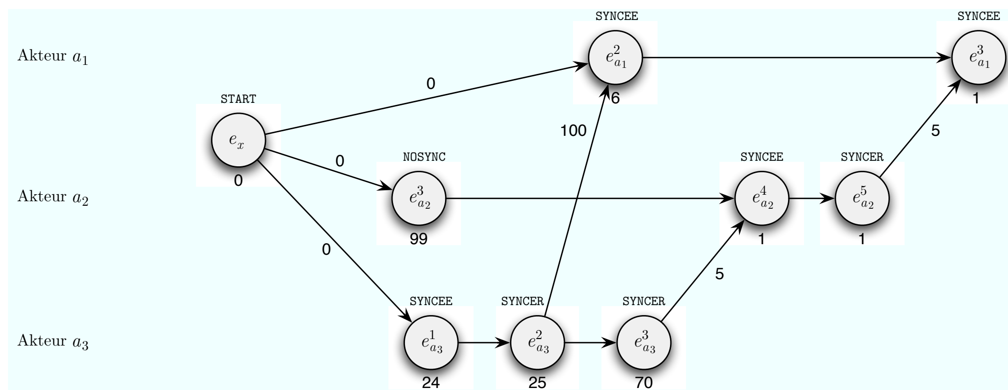


Abbildung 6.11.: Geänderter ANV des Beispielsystems ab einem gesicherten Zustand

also seines Zustandsraums und notwendiger Verwaltungsdaten, auf allen vorhandenen Stellen andere Akteure ihre Berechnungen fortführen, ohne auf Ergebnisse von a warten zu müssen, so bleibt der Performanzverlust für das Gesamtsystem vernachlässigbar. Lediglich zusätzliches Warten von Akteuren im Fall einer Migration führt im Gesamtsystem zu einer Verlängerung der Ausführungszeit und muss damit in den Kosten berücksichtigt werden. Dieser Sachverhalt wird durch die Integration der Migration in den ANV implizit berücksichtigt.

Angenommen von der initialen Zuweisung $s_1(a_1, a_2)$ und $s2(a_3)$, die im konsistenten Schnitt gespeichert ist, soll nach dem erneuten Start die Zuweisung $s_1(a_1, a_3)$ und $s2(a_2)$ realisiert werden, so müssen anfangs die Akteure a_2 und a_3 migriert werden. Es sind also in den zugrunde liegenden ANV die initialen Kanten von e_x zu den Ereignissen $e^1_{a_3}$ und $e^3_{a_2}$ mit den approximierten Kosten der Migration zu versehen, da die Berechnungen dieser beiden Akteure erst nach der Übertragungszeit beginnen können. Diese Kosten werden mit 100 veranschlagt und der sich ergebende Graph ist in Abbildung 6.12 dargestellt.

Dieser mit Migrationskosten versehene, auf den gespeicherten Zustand angepasste ANV dient als Basis für die Kostenfunktion, um die jeweilige Alternative zu bewerten. Die Konstruktion des ANV kann mit der erklärten Erweiterung der Modellierung des im Schnitt gespeicherten Zustands als Startereignis erfolgen. Je nach Stellenzuweisung müssen dann die Kanten mit den Migrationskosten versehen werden. Die Bewertung einer solchen Alternative erfolgt über die approximierte Restlaufzeit des Systems. Analog zu Systemabläufen, die in den Gantt-Diagrammen visualisiert sind, können die Laufzeiten mit implizit integrierten Wartezeiten über den Graph dargestellt werden, indem iterativ für jedes Ereignis

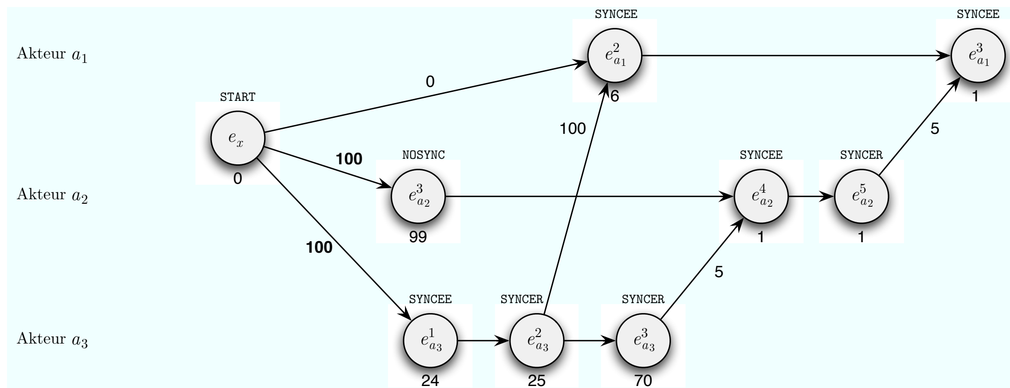


Abbildung 6.12.: ANV des Beispielsystems ab gesichertem Zustand mit Migrationskosten

vor und nach Abarbeitung berechnet wird, wie weit die Zeit fortgeschritten ist und wie viel in dieser Zeit auf der Stelle tatsächlich gerechnet wurde. Iterativ bedeutet hier, dass die Start-Werte eines Ereignisses erst berechnet werden können, wenn die End-Werte aller Vorgänger-Ereignisse vorhanden sind.

Vereinfacht ausgedrückt berechnet der Algorithmus die Gesamtlaufzeit des Systems unter Berücksichtigung von Stellenzuweisung und Wartezeiten. Diese Informationen sind im ANV enthalten, es muss also lediglich im Graphen iterativ für jeden Knoten geprüft werden, zu welchem Zeitpunkt seine Berechnung beginnen kann und wie die gesamte Systemzeit voranschreitet. Dazu werden für die Abarbeitung jedes Knotens zwei Variablen aktualisiert, erstens die Summe ausgeführter Rechenzeit (beliebiger Akteure) auf der zugewiesenen Stelle und zweitens die vergangene Gesamtlaufzeit des Systems nach Abarbeitung des Knotens. Die wesentlichen Schritte des Algorithmus werden im Folgenden beschrieben. Dabei werden für jeden Knoten iterativ die beiden Werte berechnet, wobei die Gesamtlaufzeit auch synchronisationsbedingte Wartezeiten enthält.

Algorithmus 6.1 (Approximation der Systemlaufzeit)
Gegeben:

- *Eine Zuweisung der Akteure auf Stellen;*
- *Der zu untersuchende gewichtete Graph des ANV mit:*
- *Gewichteten Knoten K, die mit den Ereignisnamen der Akteure bezeichnet sind, dabei sei $g(v_i)$ das Gewicht des Knoten $v_i \in K$;*

- *Gewichteten Kanten E, die durch ein Knotenpaar identifiziert werden, dabei sei $g(v_i, v_j)$ das Gewicht der Kante von Knoten v_i zu Knoten v_j;*

- *Einem besonders gekennzeichneten Startereignis e_x;*

- *Einer Funktion $pred(v_i)$, die den Vorgängerknoten von v_i des selben Prozesses zurückgibt;*

- *Einer Funktion $sync(v_i)$, die den Knoten des `SYNCER`-Ereignisses zurück liefert, falls v_i ein `SYNCEE`-Ereignis ist;*

Gesucht: Approximierte Gesamtlaufzeit des Systems.

Algorithmus:

Initialisierungsphase:
Für jeden Knoten $v_i \in K$ müssen die Werte mit 0 initialisiert werden:

- $v_i.t := 0;$ *– – Vergangene Gesamtzeit nach Knoten v_i*

- $v_i.b := 0;$ *– – Summe an ausgeführter Rechenzeit auf zugewiesener Stelle nach Knoten v_i*

Iterative Berechnung der Ausführungszeiten:
Für jeden Knoten $v_i \in K$, dessen Vorgänger bereits berechnet wurden, sind für folgende disjunkte Fälle *die angegebenen Berechnungsvorschriften auszuführen, zum besseren Verständnis wird jeweils ein Beispiel-Knoten aus Abbildung 6.12 angegeben, falls ein solcher Knoten existiert:*

1. *Falls eine Kante von e_x zu v_i existiert, muss unterschieden werden, ob v_i noch eine weitere Eingangskante hat:*

   a) *Hat v_i keine weitere Eingangskante, so wird v_i nach Fall 3 behandelt mit $sync(v_i) := e_x$, dies ist beispielsweise bei $e^3_{a_2}$ der Fall.*

   b) *Hat v_i eine weitere Eingangskante, so wird $pred(v_i) := e_x$ gesetzt und gemäß unten stehender Fälle behandelt, dies ist beispielsweise bei $e^2_{a_1}$ der Fall.*

2. *Falls v_i der Knoten des ersten Ereignisses eines Akteurs ist, der der gleichen Stelle als sein Vater $sync(v_i)$ zugewiesen ist, so sind dessen summierte Berechnungszeiten und die bereits verstrichene Zeit zu addieren:*
   $v_i.t := g(v_i) + sync(v_i).t;$
   $v_i.b := g(v) + sync(v_i).b;$

3. *Falls v_i der Knoten des ersten Ereignisses eines Akteurs ist, der einer anderen Stelle als sein Vater $sync(v_i)$ zugewiesen ist, so startet die Berechnung des Ereignisses erst nach verstrichener Zeit des Vaters und der Übertragungszeit, die Ausführungsdauer auf der neuen Stelle bleibt unberührt:*
   $v_i.t := g(v_i) + sync(v_i).t + g(sync(v_i), v_i);$
   $v_i.b := g(v_i);$

4. *Falls v_i nur die Eingangskante des Vorgängerereignisses des gleichen Akteurs hat, so schreiten Ausführungszeit und Rechenlast additiv fort, beispielsweise bei e_{a2}^5:*
   $v_i.t := g(v_i) + pred(v_i).t;$
   $v_i.b := g(v_i) + pred(v_i).b;$

5. *Falls v_i zwei Vorgänger $pred(v_i)$ und $sync(v_i)$ hat, und der Akteur mit dem Ereignis $sync(v_i)$ der selben Stelle wie v_i zugewiesen ist, so müssen zwei Unterfälle unterschieden werden, beispielsweise bei e_{a2}^4:*

   a) *Falls die Berechnungen beider Vorgänger in der abgelaufenen Zeit stattgefunden haben können, also falls gilt*
      $|pred(v_i).t - sync(v_i).t| \geq min(pred(v_i).b, sync(v_i).b),$
      dann wird die Laufzeit nur zum größeren Laufzeitwert hinzuaddiert:
      $v_i.t := g(v_i) + max(pred(v_i).t, sync(v_i).t);$
      $v_i.b := g(v_i) + pred(v_i).b + sync(v_i).b;$

   b) *Andernfalls muss die zusätzliche Berechnungszeit zur Laufzeit addiert werden:*
      $v_i.t := g(v_i) + max(pred(v_i).t, sync(v_i).t)$
      $+ (min(pred(v_i).b, sync(v_i).b) - |pred(v_i).t - sync(v_i).t|);$
      $v_i.b := g(v_i) + pred(v_i).b + sync(v_i).b;$

6. *Falls v_i zwei Vorgänger $pred(v_i)$ und $sync(v_i)$ hat, und der Akteur mit dem Ereignis $sync(v_i)$ einer anderen Stelle als v_i zugewiesen ist, so muss die spätere Gesamtzeit addiert werden, beispielsweise bei $e^3_{a_1}$:*
$v_i.t := g(v_i) + max(pred(v_i).t, (sync(v_i).t + g(sync(v_i), v_i)));$
$v_i.b := g(v_i) + pred(v_i).b;$

Ergebnis: *Der Algorithmus endet mit der Berechnung der Werte für das letzte Ereignis des Akteurs* `SYSTEM`*. Der Wert $v.t$ des Knotens v des letzten Ereignisses entspricht der approximierten Systemlaufzeit.*

Der beschriebene Algorithmus liefert die Systemlaufzeit einer Stellenzuweisung ab einem konsistenten Schnitt, analog zu den Gantt-Diagrammen. Dabei werden Synchronisations- und Migrations-bedingte Wartezeiten berücksichtigt. Der Algorithmus erfüllt also die Anforderungen für die Kostenfunktion. Die Kostenfunktion

$$K_S : W \times W \mapsto \mathbb{R}$$

kann also definiert werden als die Differenz der berechneten Gesamtlaufzeiten zweier Alternativen. Sei $runtime(W_i)$ die mit Algorithmus 6.1 berechnete Gesamtlaufzeit der Alternative $W_i \in W$, dann ist die Kostenfunktion

$$K_S(W_i, W_j) = runtime(W_j) - runtime(W_i),\ W_i, W_j \in W.$$

Negative Werte der Kostenfunktion würden in diesem Fall bedeuten, dass für die betrachtete Alternative eine effizientere Ausführung zu erwarten ist als für die vor dem Fehlerfall verwendete.

Aus Gründen der Übersichtlichkeit wird kein ANV angegeben, der dem Stellenzuweisungsbeispiel W_1 aus Abbildung 6.2 mit den betrachteten Alternativen W_2 aus Abbildung 6.4 und W_3 aus Abbildung 6.5 mit drei Stellen und neun Akteuren entspricht. Es seien im Folgenden aber die plausiblen Werte angenommen, dass die Kostenfunktion für Alternative W_2 einen Wert von $K_S(W_1, W_2) = 80$ und für Alternative W_3 einen Wert von $K_S(W_1, W_3) = 450$ habe. Dies spiegelt die ebenfalls hohe Parallelität von W_2 im Gegensatz zu W_3 wider.

Die hier vorgestellte Kostenfunktion ist vergleichsweise aufwändig zu berechnen, da für jede betrachtete Alternative der Algorithmus ausgeführt werden muss, dessen Laufzeit linear zur Anzahl Ereignisse im ANV ist. Natürlich kann auch eine einfachere, schneller zu berechnende Kostenfunktion gewählt werden, es ist jedoch damit zu rechnen, dass die daraus resultierenden Entscheidungen dementsprechend schlechter ausfallen können.

6.2.3 Strategie

Mit gegebener Varianz V und Kostenfunktion K muss die Auswahl einer Alternative eines Entscheidungsraums durch das Management getroffen werden. Es gibt unterschiedliche Strategien, die zur Auswahl einer geeigneten Alternative führen können, die jedoch alle auf Varianz und Kosten basieren:

- *Nutzen-optimiert*: Es wird die Alternative mit höchster Varianz gesucht. Diese Alternative hat nach Definition der Varianz die höchste angenommene Wahrscheinlichkeit, den aufgetretenen Fehler zu tolerieren. Haben mehrere Alternativen das gleiche Maximum an Varianz, so kann daraus die Alternative mit geringsten Kosten oder eine zufällige Alternative gewählt werden.
- *Kosten-optimiert*: Es wird die Alternative mit geringsten Kosten gewählt. Ein Sonderfall ist die Alternative, deren Varianz 0 ist, wenn also die gleiche Systemausführung, bei der ein Fehler aufgetreten ist, nochmals versucht wird. Diese Alternative ist häufig, allerdings nicht zwingend diejenige mit den geringsten Kosten, da einerseits der im Schnitt enthaltene Zustand nicht geändert werden muss und andererseits das Management initial Kosten-optimierte Entscheidungen triffft. Haben mehrere Alternativen das gleiche Kosten-Minimum, so kann analog zufällig oder nach bester Varianz gewählt werden.
- *Bestes Verhältnis*: Es wird die Alternative mit maximalem Verhältnis aus Varianz und Kosten bevorzugt.

Jede Strategie vergleicht eine Menge möglicher Alternativen mittels einer Bewertungsfunktion $B : W \times W \mapsto I\!R$, die Alternative mit der höchsten Bewertung wird ausgewählt. Die Strategien hängen dabei von der Art der Kombination von Varianzfunktion und Kostenfunktion zu der Bewertungsfunktion ab. Die beschriebenen drei Strategien wären – von Sonderfällen abgesehen – in folgenden Bewertungsfunktionen realisiert, mit $W_i, W_j \in W$:

- Nutzen-optimiert:
$$B(W_i, W_j) = V(W_i, W_j)$$
- Kosten-optimiert:
$$B(W_i, W_j) = \frac{1}{K(W_i, W_j)}$$

- Bestes Verhältnis:

$$B(W_i, W_j) = \frac{V(W_i, W_j)}{K(W_i, W_j)}$$

Kann die Kostenfunktion bzw. die Varianzfunktion den Wert 0 annehmen, so müssen diese Sonderfälle getrennt berücksichtigt und behandelt werden. Besonders interessant ist der Fall, dass beide Funktionen den Wert 0 ergeben, weil es sich um dieselbe Alternative handelt, deren Ausführung also zum Fehlerfall geführt hat. Die im Verhältnis zur gesamten Systemausführung kurzen Abschnitte zwischen einem Checkpoint und dem aufgetretenen Fehler, die erneut ausgeführt werden müssen, sind in der Regel auf Grund der nebenläufigen Ausführung dennoch nicht deterministisch. Deshalb kann der gleiche Fehler auch bei wiederholter Ausführung der identischen Alternative unter Umständen toleriert werden. Auch wenn die Wahrscheinlichkeit hierfür geringer angenommen wird als bei einer veränderten Systemkonfiguration, so kann dies abhängig von der Strategie dennoch eine gute Wahl sein. Tritt jedoch der selbe Fehler erneut auf, so ist davon auszugehen, dass weitere Ausführungen dieser Alternative keinen Erfolg versprechen. Unabhängig von der Strategie einer ersten Alternative muss bei wiederholt auftretendem Fehler darauf geachtet werden, dass nicht immer wieder die gleiche Alternative ausgewählt wird. Deshalb sind dynamisch veränderliche Strategien einzusetzen.

Für dynamisch veränderliche Strategien bietet sich an, mit einem Extrem der oben genannten Strategien, beispielsweise einer Kosten-optimierten Strategie zu beginnen und bei jeder Iteration die Bewertungsfunktion mehr in Richtung Nutzen-optimiert zu verändern. Dazu können die Alternativen mit Hilfe einer Bewertungsfunktion gruppiert werden, um dann mit der zweiten Bewertungsfunktion aus einer Gruppe eine Alternative zu wählen. Tritt dann der selbe Fehler erneut auf, so kann die Gruppe mit den nächst schlechteren Kostenwerten gewählt werden, um die Alternative mit höchster Varianz daraus zu wählen.

Die Wahl einer geeigneten Strategie hängt in erster Linie von den Systemanforderungen ab und kann deshalb nicht pauschal beantwortet werden. In Systemen mit harten Laufzeitanforderungen, wie beispielsweise Echtzeitsystemen, geht die Kostenfunktion sicherlich stärker in die Strategie ein, unter Umständen auch mit einem Schwellenwert, der nicht überschritten werden darf. In Systemen ohne harte Zeitschranken kann die maximale Wahrscheinlichkeit, den Fehler zu tolerieren, das höchste Ziel sein und deshalb bestes Verhältnis oder sogar Nutzen-optimiert als Bewertungsfunktion gewählt werden. Analog zur Spezifikation von Varianz- und Kostenfunktion ist die Kombination zu einer dynamisch veränderlichen Bewertungsfunktion, die eine den Systemanforderungen angepasste Strategie reali-

siert, eine nicht triviale Aufgabe des Programmierers bei der Konstruktion des Systems.

Für das Beispiel der Stellenzuweisung aus Abbildung 6.2 kann mit den Werten der Varianz V_S und Kostenfunktion K_S geschlossen werden, dass eine rein Kosten-Optimierende Strategie als erstes die gleiche Alternative W_1 erneut wählen würde. Die Nutzen-Optimierende Strategie, die ausschließlich den höchsten Varianz-Wert betrachtet, würde Alternative W_3 aus Abbildung 6.5 bevorzugen. Eine Strategie, die das Verhältnis aus Varianz und Kosten maximiert, würde Alternative W_2 aus Abbildung 6.4 oder eine dazu ähnliche Alternative, je nach tatsächlichen Werten für die Kostenfunktion, wählen.

Für den Entscheidungsraum Stellenzuweisung in MoDiS bietet sich an, als dynamische Strategie die drei genannten Strategien zu kombinieren, indem als erster Versuch Kosten-optimiert, beim zweiten Versuch die Alternative mit bestem Verhältnis, beim dritten Versuch die Varianz-optimierte Alternative ausgeführt wird. Tritt nach drei Ausführungen der Fehler erneut auf, so wird ein anderer Schnitt gewählt, zu dem zurück gesetzt wird. Dies wird in Abschnitt 6.2.5.1 noch erläutert.

6.2.4 Generierung von Alternativen

Die dynamisch veränderliche Bewertungsfunktion erlaubt im Fall einer Rückwärtsbehebung eine strategische Auswahl einer neuen Alternative eines Entscheidungsraums für die erneute Ausführung durch das Management. Für viele Entscheidungsräume, wie beispielsweise die Stellenzuweisung, stehen allerdings sehr viele mögliche Alternativen zur Auswahl. Konkret

$$\sum_{k=1}^{n} Stirling(n,k)$$

Möglichkeiten, die n Akteure auf die (für die Fragestellung nicht unterscheidbaren) k Stellen zu verteilen, wobei $Stirling(n,k)$ die Stirling-Zahlen zweiter Art sind. Diese lassen sich rekursiv berechnen mit

$$Stirling(n,n) = 1, \text{für } n > 0$$

$$Stirling(n,0) = 0, \text{für } n > 0$$

$$Stirling(n,k) = 0, \text{für } k > n$$

$$Stirling(n,k) = k \cdot Stirling(n-1,k) + Stirling(n-1,k-1).$$

Sind beispielsweise die neun Akteure aus Beispiel W_1 neu auf drei Stellen zu verteilen, so müssten bereits 3280 Alternativen mit der Ausgangsalternative verglichen werden (insgesamt existieren 3281 Alternativen). Selbst bei sehr effizient zu berechnender Bewertungsfunktion B_S wäre eine Bewertung aller möglichen Alternativen zu aufwändig. Zum erfolgreichen Einsatz des Verfahrens muss zudem nicht die optimale Lösung gefunden werden, also nicht unbedingt die Alternative mit dem globalen Maximum der Bewertungsfunktion. Da die Bewertungsfunktion nur auf Abschätzungen von Kosten und Nutzen der Alternative beruht, ist es effizienter, weniger Rechenzeit für die Suche nach der optimalen Alternative zu verschwenden und schneller mit den Berechnungen einer guten Alternative zu beginnen, also einer Alternative mit vergleichsweise guter Bewertungsfunktion. Es wird also das Ziel verfolgt, im Management möglichst effizient eine Alternative zu generieren, die einer guten Bewertung entspricht.

Um eine solche Alternative zu erhalten sind zwei Vorgehensweisen möglich. Einerseits handelt es sich generell um ein Suchproblem, dazu können Standard-Suchverfahren aus dem Bereich der künstlichen Intelligenz verwendet werden, wie im folgenden Abschnitt näher erläutert wird. Anschließend wird die zweite Möglichkeit beschrieben, eine speziell für den Entscheidungsraum entworfene algorithmische Lösung einzusetzen.

6.2.4.1 Suchverfahren

Bei Problemen, in denen nicht das globale Optimum notwendig ist, sondern eine gute Lösung ausreicht, die Bewertung der Lösungen aber rechenaufwändig ist, können Suchverfahren mit iterativer Verbesserung eingesetzt werden. Zwei Vertreter dieser Verfahren sind *Hill-Climbing* (vgl. [HW04]) und *Simulated Annealing* (vgl. [KGV83]). In beiden Verfahren müssen Operatoren angegeben werden, die aus einer Alternative mehrere andere Alternativen, als Nachbarschaft bezeichnet, generieren. Dabei sollte keine Redundanz auftreten und die gesamte Menge an Alternativen erreichbar sein. In beiden Verfahren wird ausgehend von einer Alternative die Nachbarschaft generiert und jede der Nachbar-Alternativen mit der Bewertungsfunktion bewertet.

Beim Hill-Climbing wird die Alternative mit höchster Bewertung als nächste Zwischenlösung verwendet und iterativ wieder die Nachbarschaft generiert. Hat kein Nachbar mehr eine bessere Bewertung, so terminiert die Suche. Dieser Algorithmus findet immer ein lokales Optimum, bleibt allerdings in lokalen Optima auch stecken, egal ob diese global betrachtet einer guten oder schlechten Lösung entsprechen.

Simulated Annealing kann lokale Optima überwinden, da mit einer gewissen Wahrscheinlichkeit auch mit einem Nachbarn mit schlechterer Bewertung weitergemacht wird. Die Wahrscheinlichkeit hängt ab von der Differenz zur letzten Bewertung und wird zudem dynamisch verändert. Je länger der Suchalgorithmus bereits läuft, umso geringer wird die Wahrscheinlichkeit, dass mit einer schlechteren Alternative fortgefahren wird. Simulated Annealing entwickelt sich also über die Laufzeit immer mehr zu Hill-Climbing.

Bei dem hier vorliegenden Beispiel der Stellenzuweisung bietet sich auch ein *genetischer Suchalgorithmus* (vgl. [Gol89]) an. Die Idee genetischer Algorithmen ist, analog zur evolutorischen Entwicklung in der Natur, durch Mutation und Kreuzung immer neue Alternativen zu schaffen. Aus einer Population werden die besser bewerteten Alternativen mit höherer Wahrscheinlichkeit zum Überleben ausgewählt, die schlechter bewerteten mit geringerer Wahrscheinlichkeit ebenso. Durch zufällig gesteuerte Kreuzung und Mutation der Population entsteht die Population der folgenden Generation. Hoch bewertete Alternativen einer Population kommen neben Kreuzung oder Mutation zufallsgesteuert auch unverändert in die folgende Population.

Das Prinzip des genetischen Algorithmus ist in Abbildung 6.13 veranschaulicht, wobei die Kanten mit den Bezeichnern Mu für Mutation und Kr für Kreuzung stehen. Je nach Kanten werden die Eigenschaften der Alternativen, hier als geometrische Form und Farbe dargestellt, kombiniert und verändert. Eine Kante ohne Bezeichner bedeutet eine unveränderte Übernahme einer Alternative in die nächste Generation. Die Laufzeit ist bei diesem Verfahren durch die Größe der Populationen und die Anzahl Generationen festgelegt. Der Einsatz eines genetischen Suchverfahrens eignet sich für das Problem der Stellenzuweisung besonders, da sich die Mutations- und Kreuzungs- Operatoren für kombinatorische Probleme sehr einfach festlegen lassen.

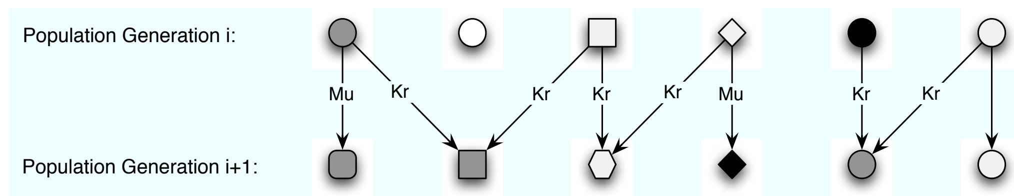


Abbildung 6.13.: Prinzip der Populationsgenerierung eines genetischen Suchalgorithmus

Der Algorithmus kann mit der Ausgangsalternative (die den Fehlerfall erzeugt hat) und einer festen Anzahl zufällig generierter Verteilungen der Akteure auf

Stellen als erste Population starten. Die Alternativen werden mit der Bewertungsfunktion B_S bewertet und anschließend gemäß dem genetischen Algorithmus behandelt. Als Mutation wird der Vorgang definiert, bei einer Alternative zufällig einen Akteur auszuwählen und einer anderen, ebenfalls zufällig gewählten Stelle zuzuweisen.

Bei der Kreuzung ist entscheidend, dass Teile der Eigenschaften einer Alternative mit den Eigenschaften einer anderen kombiniert werden. Die Eigenschaften hängen davon ab, mit welchen anderen Akteuren ein Akteur zusammen auf eine Stelle gebunden ist, nicht die absolute Zuweisung, also ob diese Stelle s_1 oder s_2 ist. Die Kreuzung lässt sich realisieren, indem die Akteure zufallsgesteuert in zwei disjunkte Mengen partitioniert werden, bei neun Akteuren beispielsweise $M_1 = \{a_1, a_2, a_3\}$ und $M_2 = \{a_4 \dots a_9\}$. Das Verhältnis der Akteure aus Menge M_1 wird nun aus einer Alternative $W_i \in W$ übernommen, das Verhältnis der Akteure aus M_2 aus der anderen Alternative $W_j \in W$. Waren also in Alternative W_i Akteure a_1 und a_2 auf einer Stelle kombiniert, a_3 dagegen auf einer anderen Stelle, so ist dies in der neu erzeugten Kreuzungs-Alternative ebenso. Waren in Alternative W_j die Akteure a_4, a_5 und a_6 zusammen auf einer Stelle, a_7 einzeln und a_8 mit a_9 auf eine Stelle gebunden, so ist diese Verteilung auch in der neuen Kreuzungs-Alternative identisch. Lediglich die Kombination der beiden Mengen M_1 und M_2 in der neuen Alternative ist zufallsgesteuert und könnte demnach sechs verschiedene neue Zuweisungen erzeugen (die Stellen werden nicht unterschieden):

$$
\begin{aligned}
&s_1(a_1, a_2, a_4, a_5, a_6),\ s_2(a_3, a_7),\ s_3(a_8, a_9);\\
&s_1(a_1, a_2, a_4, a_5, a_6),\ s_2(a_7),\ s_3(a_3, a_8, a_9);\\
&s_1(a_3, a_4, a_5, a_6),\ s_2(a_7),\ s_3(a_1, a_2, a_8, a_9);\\
&s_1(a_3, a_4, a_5, a_6),\ s_2(a_1, a_2, a_7),\ s_3(a_8, a_9);\\
&s_1(a_4, a_5, a_6),\ s_2(a_3, a_7),\ s_3(a_1, a_2, a_8, a_9);\\
&s_1(a_4, a_5, a_6),\ s_2(a_1, a_2, a_7),\ s_3(a_3, a_8, a_9).
\end{aligned}
$$

Ungeachtet der Kreuzung kann durch Mutation nach maximal $|A| - 1$ Generationen aus jeder Anfangsalternative jede beliebige Stellenzuweisung generiert werden, $|A| - 1$ kann somit als obere Schranke für die Anzahl an Generationen des Verfahrens genutzt werden. Bei einer hohen Anzahl an Akteuren ist jedoch eine kleinere Schranke sinnvoller, da nur eine gute, nicht die optimale Lösung gesucht wird.

6.2.4.2 Algorithmische Generierung

Abhängig vom Entscheidungsraum und der Bewertungsfunktion ist es möglich, einen Algorithmus anzugeben, der aus der bestehenden Alternative direkt eine neue mit guter Bewertung generiert. Kann ein solcher Algorithmus gefunden werden, so ist die Generierung in der Regel wesentlich effizienter als ein Suchverfahren. Allerdings kann es für viele Entscheidungsräume sehr schwierig sein, einen allgemeinen Algorithmus anzugeben, der unter Berücksichtigung der dynamischen Strategie eine gute Alternative generiert. Die Logik des Algorithmus muss sich an der Bewertungsfunktion orientieren. Am Beispiel der Stellenzuweisung kann diese Problematik anhand der beiden Strategien *Varianz-optimiert* und *bestes Verhältnis* demonstriert werden.

Optimiert die Strategie die Varianz der Alternative, so geht ausschließlich die Varianzfunktion in die Bewertungsfunktion ein: $B(W_i, W_j) = V(W_i, W_j)$, mit $W_i, W_j \in W$. Im Beispiel der Stellenzuweisung ist die Varianzfunktion V_S relativ leicht auf einen Algorithmus abzubilden. Die Varianzfunktion V_S bewertet aufgelöste und neu geschlossene Kombinationen von Akteurpaaren. Werden iterativ alle Stellen betrachtet und jeweils alle Akteure, die in der alten Alternative dieser Stelle zusammen zugewiesen waren, nun gleichmäßig auf alle Stellen verteilt, so ist eine gute Varianz sichergestellt. Der Algorithmus läuft wie folgt ab:

Algorithmus 6.2 (Stellenzuweisung)
Gegeben:

- *Die Menge der Akteure* $A = \{a_1, \ldots, a_n\}$;
- *Die Menge der Stellen* $S = \{s_1, \ldots, s_m\}$;
- *Eine gegebene Zuweisung der Akteure auf Stellen als Mengen* $\forall s_x \in S : s_x.akt_{alt} = \{a_y, \ldots, a_z\}, 1 \leq x, y \leq n$;
- *Eine Funktion* $next(s)$, *die die jeweils nächste Stelle der geordneten Menge* S *zurückgibt, mit* $next(s_m) := s_1$.

Gesucht: Neue Stellenzuweisung mit hoher Varianz.

Algorithmus:

Initialisierungsphase:
Für jede Stelle muss die neue Zuweisung leer initialisiert werden:

- $\forall s_x \in S : s_x.akt_{neu} := \{\}$

Neue Stellenzuweisung:

- *In einer Schleife werden alle Stellen der Reihe nach abgearbeitet:*
  $\forall s_x \in S :$
  - *Alle Akteure der betrachteten Stelle werden ab dieser Stelle der Reihe nach verteilt:*
    $s_{to} := s_x;$
    $\forall a_y \in s_x.akt_{alt} :$
    * *Füge aktuellen Akteur zu Stelle:*
      $s_{to}.akt_{neu} := s_{to}.akt_{neu} \cup \{a_y\};$
    * *Gehe um eine Stelle weiter:*
      $s_{to} := next(s_{to});$

Ergebnis:
Die neue Alternative der Stellenzuweisung liegt vor, jeder Stelle sind nun in $s_x.akt_{neu}$ *eine Menge Akteure zugewiesen.*

Für eine neue Verteilung von n Akteuren auf m Stellen liefert der genannte Algorithmus ein Ergebnis mit akzeptabler Varianz, also häufig nicht das Optimum, aber eine gute Lösung, solange $n >> m$ gilt. Sind mehr Stellen als Akteure vorhanden, so muss dieser Sonderfall abgefangen und getrennt in ähnlicher Art behandelt werden. Die Laufzeit des Algorithmus ist mit $O(n)$ begrenzt und demnach wesentlich effizienter als ein Suchverfahren, da alleine die Bewertungsfunktion bereits eine höhere Laufzeit hat und diese ja beim Suchverfahren häufig aufgerufen werden muss. Bei dem vorliegenden Algorithmus wird allerdings ausschließlich nach der Varianz optimiert. Die Kosten können dabei – trotz relativ gleichmäßiger Verteilung der Akteure auf Stellen – auf Grund Synchronisationsbedingter Wartezeiten beliebig steigen.

Um einen Algorithmus anzugeben, der die Strategien *Kosten-optimiert* oder *bestes Verhältnis* realisiert, müssten die Grundlagen der Kostenbewertung mit in den Algorithmus einfließen. Denkbar wäre hier ein Greedy-Verfahren, das den Graph des ANV ähnlich zur Kostenfunktion durchläuft, und abhängig von Kommunikationskanten und Rechenzeiten die Stellenzuweisung ändert. Ein Greedy-Verfahren für diesen Fall kann allerdings beliebig schlecht sein und die Berechnung des Optimums wäre zu aufwändig. Für diesen Entscheidungsraum bietet sich daher eine Lösung mit Suchalgorithmen an.

Für andere Entscheidungsräume mit einfacherer Bewertungsfunktion kann die algorithmische Lösung auf Grund der Effizienz allerdings deutlich attraktiver sein als ein Suchverfahren. Diese Entscheidungen sind für jeden Entscheidungsraum separat zum Zeitpunkt des Systementwurfs zu treffen.

6.2.4.3 Informationsgewinnung

Die Entscheidungsfindung des Managements basiert auf den Informationen der vergangenen Systemausführungen, die zu Fehlerfällen geführt haben, sowie Anwendungsinformationen, die für die Bewertung der Alternativen genutzt werden. Der Einsatz des Verfahrens setzt entsprechende Mechanismen zur Ereignisprotokollierung und Speicherung voraus. Die in Abschnitt 4.1 beschriebenen Mechanismen zur Beobachtung verteilter Systeme können hierzu entsprechend erweitert werden, sodass die notwendigen Informationen gespeichert werden. Die Auswahl der zu speichernden Informationen gehen direkt aus dem Entscheidungsverfahren hervor.

Unabhängig vom Entscheidungsraum müssen alle verfügbaren Informationen über den aufgetretenen Fehlerzustand gespeichert werden, sodass möglichst sicher erkannt werden kann, ob ein bereits aufgetretener Fehler erneut auftritt. Ferner sind Informationen über zuletzt getroffene Entscheidungen relevant, sodass dynamische Strategien, wie in Abschnitt 6.2.3 erklärt, eingesetzt werden können.

Zusätzlich müssen Informationen für den jeweiligen Entscheidungsraum festgehalten werden. Im Fall der Stellenzuweisung sind einerseits für die Berechnung der Bewertungsfunktion die zugrunde liegenden Ereignisspuren während der Übersetzungszeit zu erzeugen, um den ANV des Systems zu konstruieren. Diese Informationen sind Approximationen, da zur Übersetzungszeit nicht-deterministische Ereignisse, wie beispielsweise die Anzahl Schleifendurchläufe, nicht bekannt sind. Die Informationen der Systembeobachtung werden genutzt, um den Teil des ANV bis zum aufgetretenen Fehlerfall an die tatsächlich aufgetrete-

ne Ausführung anzupassen. Dies ermöglicht eine realitätsnähere Bewertung und somit eine Entscheidung mit besserem Ergebnis. Die Stellenzuweisungen, wie sie bis zum Fehlerfall vom Management gewählt wurden, sind ebenfalls Grundlage einer neuen Alternativenwahl. Diese Information wird bereits in der Systembeobachtung erfasst und steht zur Verfügung.

Die Informationen müssen einerseits dem Management zur Verfügung gestellt werden, solange die Entscheidungsfindung dadurch unterstützt wird. Andererseits müssen auf Grund der Menge zu speichernder Daten veraltete Informationen gelöscht werden. Die Rasterung der Ausführung durch Checkpoints bietet hier einen geeigneten Rahmen. Bereits zur Übersetzungszeit werden potentielle Checkpoints in die Ereignisfolge der Akteure integriert (vgl. Abschnitt 5.2.5). Tritt ein Fehler auf, so wird versucht, die Ausführung ab dem letzten Checkpoint erneut zu starten und über eine alternative Ausführung den Fehler zu umgehen. Dies kann mehrere Versuche benötigen; währenddessen sind die Informationen über jeden Versuch von Interesse, um eine geeignete dynamische Strategie zu realisieren und keine Alternativen wiederholt auszuführen. Läuft die Ausführung des Akteurs bis zur erfolgreichen Speicherung seines nächsten Checkpoints, so ist davon auszugehen, dass der Fehler toleriert wurde.

Die Informationen können abgesehen von der zuletzt gefundenen, erfolgreichen Alternative verworfen werden. Die erfolgreiche Alternative wird zusammen mit dem Zustand des Akteurs im Checkpoint gesichert. Muss zu späterer Ausführungszeit doch noch zu einem früheren Checkpoint zurückgesetzt werden, so kann die Information über erfolgreiche Entscheidungen dennoch genutzt werden. Mit dieser Vorgehensweise wird statistisch lediglich die Systembeobachtung der Vergangenheit sowie die Information über erfolgreiche Entscheidungen gespeichert. Diese Informationen können zudem für das Lernen des Managements genutzt werden, das in Abschnitt 6.3.1.2 noch angesprochen wird.

6.2.4.4 Umsetzung einer Alternative

Das Management kann mit den bereitgestellten Informationen wie beschrieben eine Entscheidung für eine Alternative der Ausführung treffen. Um diese Alternative umzusetzen, sind Mechanismen in der Laufzeitumgebung bereitzustellen, die die Ausführung der gewählten Alternative durchsetzen. Diese Mechanismen sind bei der Systemkonstruktion für jeden Entscheidungsraum zu erstellen und geeignet zu kombinieren. Im Fall der Stellenzuweisung sind dies zwei Mechanismen im Laufzeit-Management des MoDiS-Projekts: Einerseits ist die gezielte Platzierung eines neu zu erzeugenden Akteurs auf eine Stelle ein realisierter Me-

chanismus. Andererseits kann im MoDiS-Laufzeitmanagement ein bereits aktiver Akteur transparent auf eine andere Stelle migriert werden.

6.2.5 Entscheidungsräume

Das vorgestellte Verfahren wurde für den Einsatz mit nur einem Entscheidungsraum beschrieben und anhand des Entscheidungsraums der Stellenzuweisung veranschaulicht. Folgend werden Kriterien zur Identifikation sinnvoller Entscheidungsräume erläutert sowie auf die Kombination verschiedener Entscheidungsräume eingegangen.

Die Freiheitsgrade des Managements bei der Konkretisierung einer Spezifikation zur Übersetzungs- und Ausführungszeit bestimmen die möglichen Entscheidungsräume einer Sprache bzw. der in der Sprache spezifizierten Systeme. Muss das Management zur Konkretisierung aus verschiedenen Alternativen eine zur Konkretisierung auswählen, so wird dies für die hier vorliegende Arbeit als *Entscheidung* bezeichnet. Die Vereinigungsmenge aller Alternativen logisch zusammengehöriger Entscheidungen bildet einen *Entscheidungsraum*. Abhängig vom aktuellen Systemzustand steht dem Management für eine Entscheidung dann eine gewisse Teilmenge des Entscheidungsraums als mögliche Alternativen zur Verfügung. Die Einteilung in Entscheidungsräume ist bis zu einem gewissen Grad willkürlich und muss für das Verfahren sinnvoll getroffen werden. Beispielsweise ist die Granularität des Entscheidungsraums sinnvoll zu wählen. So könnte anstatt der Zuweisung eines gesamten Akteurs zu einer Stelle auch die Zuweisung einzelner Instruktionen seiner Ausführung auf Stellen getroffen werden, was offensichtlich nicht zweckdienlich ist. Im MoDiS-Management bestehen vergleichsweise hohe Freiheitsgrade der Konkretisierung, die für das Fehlertoleranzverfahren als Entscheidungsräume genutzt werden könnten.

Bevor mögliche Entscheidungsräume auf Grund der Freiheitsgrade des Managements identifiziert werden, ist in erster Linie von Interesse, welche Entscheidungen der Konkretisierung für das vorgestellte Fehlertoleranzverfahren geeignet sind. Es muss analysiert werden, welche Arten konkretisierungsabhängiger Fehler häufig auftreten und welche Entscheidungen des Managements auf deren Auftreten Einfluss nehmen können.

Es existieren wenig statistische und vergleichbare Daten, welche Arten von Fehlern nach den Qualitätssicherungs-Maßnahmen der Softwareentwicklung in welcher Häufigkeit auftreten (vgl. [Wag06]). Einerseits unterscheiden sich die eingesetzten Techniken zur Qualitätssicherung (vgl. Abschnitt 6.1.1.1), ferner hängt das Ergebnis sehr vom Aufwand ab, beispielsweise von der Anzahl geprüfter Test-

fälle. Ein Hauptproblem ist, dass in der Forschung hauptsächlich über die mit Hilfe der Verfahren entdeckten Fehler berichtet wird. Für die hier vorliegende Arbeit wären Daten über nicht entdeckte Fehler von Interesse. Dazu müssten diese mittels anderer Verfahren (oder beim Einsatz der Software in der Praxis) gefunden und identifiziert werden. Eine weitere Hürde steckt darin, dass das Interesse an der Veröffentlichung nicht gefundener Fehler von Seiten der Qualitätssicherungs-Forschung geringer ist als die Veröffentlichung der Fehler, die mit neuen Verfahren und Werkzeugen gefunden werden können. Dennoch wird hier eine qualitative, nicht vollständige Auflistung typischer Fehlerarten gegeben, deren quantitatives Auftreten auf Grund der verschiedenen Studien nicht untereinander vergleichbar ist (nach [SC91], [BPS00], [WJKT05], [Gra86], [KAYE04], [VMM91], [OH96]):

- Fehlerhafter Speicherzugriff. Diese Klasse untergliedert sich in verschiedene typische Ursachen:
  - Speicherallokation: Eine Speicherregion wird beispielsweise freigegeben, obwohl nochmals darauf zugegriffen wird. Wird der Speicherbereich von einem anderen Modul realloziert, so tritt kein Speicherzugriffsfehler auf, sondern Nutzdaten des Programms werden überschrieben. Bei einem solchen Fehler können verschiedenste Symptome auftreten.
  - Pufferüberlauf: Es wird über die Grenzen eines Puffers hinaus geschrieben oder gelesen. Ist der auf den Puffer folgende Speicherbereich gültig, so wird ebenfalls kein Zugriffsfehler registriert und Nutzdaten überschrieben.
  - Zeigermanagement: Die Adresse eines Objekts wurde verfälscht, nicht initialisiert oder zeigt auf `NULL`.
- Fehler im zeitlichen Ablauf:
  - Fehlerhafte oder fehlende Synchronisation. Dies kann sowohl zu Verklemmungen oder dem Verhungern von Prozessen führen als auch die eigentlich vom Programmierer gewollte Semantik verändern, weil beispielsweise Teilergebnisse einer kritischen, atomar auszuführenden Phase überschrieben werden.
  - Falsche Annahmen des Programmierers über zeitliche Abläufe.
  - Nachrichtenreihenfolgen stimmen nicht mit den Annahmen des Programmierers überein.
- Datentypen passen nicht zueinander.

- Ressourcen reichen nicht für die Anforderungen. Dieser Punkt wäre auch unter die Fehler des zeitlichen Ablaufs einzuordnen. Er wird hier aber extra behandelt, da neben einer veränderten Vergabe der Ressourcen auch die erhöhte Ressourcenbereitstellung zur Lösung führen kann.

- Nutzdaten haben falschen Wert: Dies kann nicht nur an semantischen Fehlern der Programmlogik, sondern auch an fehlender Initialisierung oder fehlerhaftem Speichermanagement liegen.

- Abbruchbedingungen sind fehlerhaft: Rekursionen oder Schleifen laufen endlos.

Die angegebene Liste an Fehlerquellen konkretisierungsabhängiger Fehler soll lediglich einen Überblick geben, welche Fehlerarten auftreten können. Ein entscheidender Vorteil des vorgestellten Verfahrens ist, dass es nicht notwendig ist, Fehler zum Zweck der Fehlertoleranz genau zu identifizieren. Aus diesem Grund reicht es aus, die grundsätzlichen Bereiche typischer Fehler für die Wahl von Entscheidungsräumen in Betracht zu ziehen. Seltene oder unbekannte Fehler können mit einer gewissen Wahrscheinlichkeit dennoch toleriert werden, solange das Management erkennen kann, dass ein Fehlerzustand vorliegt.

Ausgehend von den gezeigten Fehlern ergeben sich aus den Konkretisierungs-Entscheidungen des Managements heraus folgende besonders interessante Bereiche, in denen Entscheidungsräume für die Fehlertoleranz durch Diversität identifiziert werden können, wobei an dieser Stelle nicht zwischen Aufgaben zur Übersetzungs- bzw. Ausführungszeit unterschieden wird:

Nebenläufigkeit. Ist für Komponenten nicht festgelegt, ob sie nebenläufig zu ihrem Erzeuger ausgeführt werden, so kann das Management selbst entscheiden, ob diese sequentiell im Ausführungsfaden ihres Erzeugers oder aber als eigene Aktivität nebenläufig ausgeführt werden können. Dieser Entscheidungsraum betrifft sowohl die zeitlichen Abläufe des Systems als auch den Ressourcenverbrauch. Auf Grund des sequentiellen Denkens des Menschen sind nebenläufige Ausführungen komplexer zu programmieren und damit eine häufige Fehlerquelle. Zunächst bringt die Nebenläufigkeit allerdings potentiell eine Parallelität der Ausführung und damit einen Performanzgewinn mit sich. In diesem Fall müsste die Kostenfunktion diesen Verlust erfassen, die Varianzfunktion müsste den Grad der sequentiellen Ausführung messen.

Speicherlayout. Die Arbeiten von Xu et al. zum randomisierten Speicherlayout von Prozessen mit dem Ziel der Softwaresicherheit begründen sich durch die Probleme im Speichermanagement (vgl. Abschnitt 6.1.2.2 und [XKI03]). Wie beschrieben sind die vermiedenen Attacken nur durch eigentliche Fehler im Speichermanagement möglich, die gezielt genutzt werden. Aus den Studien der Fehlerfälle geht hervor, dass das ungewollte Überschreiben von Nutzdaten im Speicher durch fehlerhafte Allokation, Freigabe oder Pufferüberläufe eines der großen Probleme ist. Nach Sullivan wird dabei in 30 Prozent der Fälle die direkt folgende Datenstruktur überschrieben, wobei die Größe des fälschlich überschriebenen Speichers in 48 Prozent der Fälle weniger als 100 Byte beträgt ([SC91]). Auch wenn diese Zahlen nicht repräsentativ sind, so liegt doch der Schluss nahe, dass ein geändertes Speicherlayout mit Pufferzonen zwischen allozierten Objekten einen Teil der Fehler vermeiden könnte.

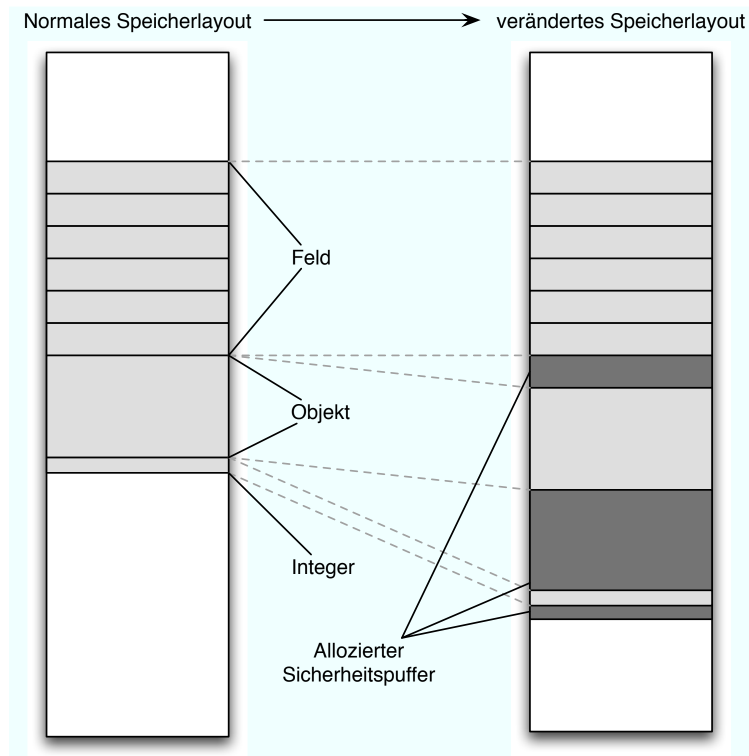


Abbildung 6.14.: Alternatives Speicherlayout mit Sicherheitspuffer

In Abbildung 6.14 ist der Übergang von normalem Speicherlayout zu einem möglichen Layout mit allozierten Sicherheitspuffern dargestellt. Nach jedem allozierten Bereich (hellgrau dargestellt) wird ein Sicherheitspuffer alloziert (dunkel dargestellt), dessen Größe dem Datentyp des allozierten Bereichs entspricht, also

bei einem Feld der Größe des Feldtyps, bei einfachen Datentypen oder Objekten jeweils der Größe des allozierten Bereichs. Das visualisierte Schema lässt sich ebenso für den Keller anwenden, wie hier für die Halde angedeutet. Es existieren bereits Compiler, wie beispielsweise der *gcc* (GNU C Compiler[3], [gcc]), die aus Sicherheitsgründen für den Keller vergleichbare Sicherheitspuffer einfügen. Würde im geänderten Speicher aus Abbildung 6.14 die Feldgrenze um eins überschritten, weil eine Schleifenvariable fehlerhaft programmiert ist, so würde nicht mehr das darauf folgende Objekt überschrieben, sondern der ungenutzte dunkel visualisierte Bereich überschrieben. Ob das Programm auf diese Art korrekt terminieren kann, ist so nicht zu beantworten, dies hängt wiederum davon ab, inwiefern die geschriebenen Daten weiter genutzt werden.

Für den Entscheidungsraum Speicherlayout könnte die Varianz als Ausmaß der Speicherpuffer interpretiert werden, in die Kosten müssten die verschwendete Speichermenge und der Aufwand der Reallokationen eingehen. Wird das Prinzip analog dem Ansatz von Xu et al. randomisiert angewendet, so könnte zudem die Sicherheit der Software erhöht werden.

Ressourcen-Management. Alle Hard- und Software Betriebsmittel, die den Akteuren für ihre Berechnungen zur Verfügung gestellt werden, können in ihrer Verwaltung geändert werden. Unter diese Gruppe fällt natürlich auch der Entscheidungsraum Stellenzuweisung, an dem das Prinzip der Diversität auf Management-Ebene erläutert wurde. Andere Betriebsmittel sind auch die CPU, deren Verwendung durch die Scheduling-Strategie festgelegt ist. Wird das Scheduling dahingehend geändert, dass die Akteure größere Zeitquanten eines Round-Robin-Verfahrens erhalten, so sinkt die Wahrscheinlichkeit, dass Fehler wie Race Conditions durch fehlerhafte Synchronisation tatsächlich auftreten.

Analog zum genannten Kriterium der Nebenläufigkeit ist auch beim Scheduling davon auszugehen, dass eine stärkere Sequentialisierung der Ausführung die Wahrscheinlichkeit für Fehler des zeitlichen Ablaufs reduzieren. Auch die Knappheit von Betriebsmitteln kann zu Fehlerfällen, insbesondere zu Deadlocks führen. Demnach kann eine veränderte Verwaltungsstrategie dieser Betriebsmittel das Auftreten dieser Situationen unter Umständen verhindern. Strategien für eingeschränkte Betriebsmittel-Vergabe können beispielsweise aus dem Bereich der Verfahren für Deadlock-Verhinderung und Deadlock-Vermeidung angepasst werden (vgl. [Tan01]).

Die eben schon erwähnte Stellenzuweisung ist zur Gruppe des Ressourcen-Managements zugehörig, bietet sich dennoch besonders an, da die Stellenzuweisung

[3]Mittlerweile steht das Akronym gcc für GNU Compiler Collection.

starken Einfluss auf die Parallelität und den Aufwand für Kommunikation und somit auf die zeitlichen Abläufe der Akteure nimmt. Lediglich das Speicherlayout wird durch den gemeinsamen verteilten Speicher zwar betroffen, aber nur sehr begrenzt beeinflusst. Insofern würde sich für die Fehlertoleranz durch Diversität auf Management-Ebene für die betrachteten Systeme eine Kombination der beiden Entscheidungsräume *Stellenzuweisung* und *Speicherlayout* anbieten.

Grundsätzlich ist die Auswahl geeigneter Entscheidungsräume abhängig vom konstruierten System. Einerseits müssen die Freiheitsgrade des Managements und deren Auswirkungen auf Ausführungsablauf, Ressourcen-Nutzung und Speicherlayout analysiert werden. Andererseits muss analysiert werden, welche Fehlerarten unter Verwendung der Anwendungs-Programmiersprache häufig oder typisch sind, da diese oft Sprachabhängig sind. Aus der Kombination dieser Punkte kann ein oder mehrere Entscheidungsräume identifiziert werden, deren Varianz durch Alternativenwahl eine hohe Wahrscheinlichkeit zur Toleranz der typischen konkretisierungsabhängigen Fehler verspricht.

6.2.5.1 Kombination von Entscheidungen

In diesem Abschnitt wird beschrieben, wie verschiedene Entscheidungen kombiniert werden können. Dies betrifft zwei Fälle, einerseits den angesprochenen Fall, dass mehrere Entscheidungsräume für das Verfahren zur Verfügung stehen, und andererseits die Auswahl des konsistenten Schnitts, zu dem zurückgesetzt wird, um eine andere Alternative auszuführen.

Die Kombination von Entscheidungen mehrerer Entscheidungsräume kann mit Hilfe von Bewertungsfunktionen nur sinnvoll vom Management getroffen werden, wenn die Entscheidungen unabhängig in Bezug auf ihre Bewertung, also Varianz und Kosten sind. Die Varianz ist die Größe, die die Änderung im Ausführungsablauf angibt und ist proportional zur Wahrscheinlichkeit, einen aufgetretenen Fehler zu tolerieren. Sind zwei Entscheidungsräume voneinander unabhängig, so steigt die Wahrscheinlichkeit, einen Fehler zu tolerieren, direkt proportional zur Summe der Varianz-Werte beider Entscheidungsräume. Ebenso summieren sich die Kosten zweier Entscheidungen, falls die Entscheidungsräume voneinander unabhängig sind.

Um im Management automatisierte Entscheidungen treffen zu können, müssen sowohl Kosten- als auch Varianzwerte beider Entscheidungsräume in irgendeiner Art und Weise vergleichbar sein. Da die Funktionen aber lediglich relative, keine absoluten Werte für Varianz und Kosten liefern und ferner auf unterschiedlichen Approximationen basieren, kann die Entscheidung im Management nur

als Kombination zweier bzw. mehrerer Einzelentscheidungen verstanden werden. Die Strategie dagegen gibt vor, ob und wie die Entscheidungen zu kombinieren sind. Analog zu den bisherigen Strategien *Nutzen-optimiert*, *Kosten-optimiert* und *bestes Verhältnis* können nun dynamische Strategien festgelegt werden, die für jeden Versuch eine neue Kombination der bisherigen Strategien auf die Entscheidungsräume anwenden. Wie die Strategien mit den Entscheidungsräumen zu kombinieren sind, ist abhängig vom Entscheidungsraum und dem System.

Beispielsweise könnte bei einer Kosten-optimierten Strategie für ein System mit den Entscheidungsräumen *Stellenzuweisung* und *Speicherlayout* der Programmierer entscheiden, dass ein geändertes Speicherlayout die Systemperformanz geringer einschränkt, und deshalb im ersten Versuch immer nur das Speicherlayout nach Kosten-optimierter Strategie geändert wird. Sollte der gleiche Fehler erneut auftreten, kann nur eine Alternative für die Stellenzuweisung getroffen werden. Würde für das gleiche System eine Nutzen-optimierte Strategie entworfen, so müsste bereits im ersten Schritt eine Alternative für Speicherlayout *und* Stellenzuweisung gewählt werden. Die Einzelentscheidungen der verschiedenen Entscheidungsräume können mit den bisherigen dynamischen Strategien gewählt werden.

Eine dynamische Strategie ändert bei wiederholt auftretendem Fehler die Wahl der auszuführenden Alternative. Es kann allerdings der Fall eintreten, dass alle Alternativen, die ab dem jüngsten gespeicherten konsistenten Schnitt ausgeführt werden, zum gleichen Fehler führen. Dies ist der Fall, falls eine Entscheidung, die im Management schon zu einem früheren Zeitpunkt getroffen wurde und im gespeicherten Zustand enthalten ist, deterministisch zum Fehlerfall führt. Aus diesem Grund ist es notwendig, dass das Verfahren nicht ausschließlich vom jüngsten konsistenten Schnitt neue Alternativen ausführt, sondern nach einigen Versuchen einen älteren gesicherten Zustand wählt, um mehr an der Ausführung ändern zu können.

Ein Rollback zu einem älteren Zustand bedeutet zusätzliche Kosten, da ein größerer Teil des Systems erneut ausgeführt werden muss. Diese Kosten bestehen aus zwei Teilen, einerseits der Summe der Ausführungskosten der im jüngeren konsistenten Schnitt enthaltenen Akteure zwischen den beiden Schnitten, andererseits der Ausführungszeit eventuell zusätzlich enthaltener Akteure im älteren konsistenten Schnitt bis zum Zeitpunkt des Fehlerfalls. Die Entscheidung ist in Abbildung 6.15 visualisiert: Angenommen, das System stellt einen Fehler fest bevor Ereignis $e_{a_1}^3$ abgeschlossen ist, so könnte entweder zum konsistenten Schnitt C_2 oder zum älteren konsistenten Schnitt C_1 zurückgesetzt werden, um eine alternative Ausführungsentscheidung zu treffen.

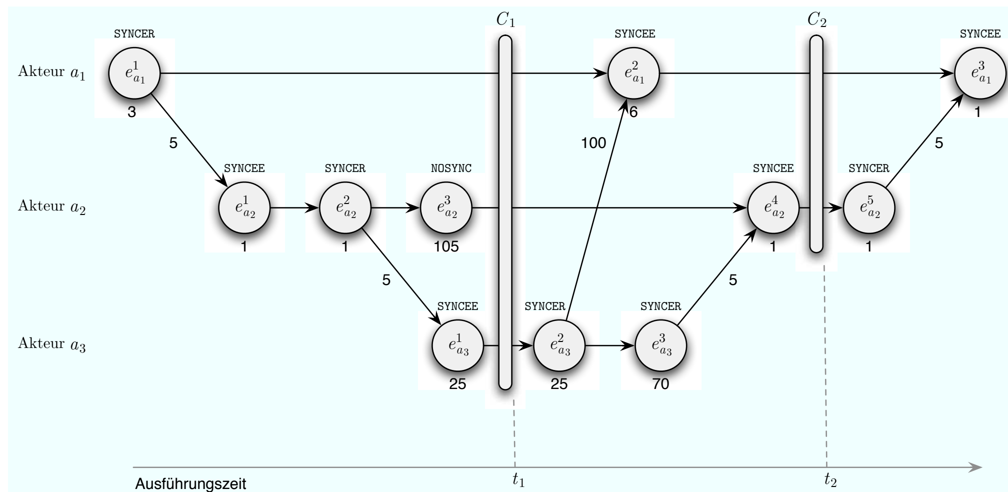


Abbildung 6.15.: Wahl eines konsistenten Schnitts zur Rückwärtsbehebung

Es wäre möglich, Varianz und Kostenfunktion so zu erweitern, dass die Wahl des Checkpoints auf Basis der gleichen Approximationen entschieden wird, dass also die alternativen Ausführungen von C_1 aus mit den Alternativen von C_2 vergleichbar sind und unter Berücksichtigung der Strategie die am Besten bewertete Alternative (inklusive konsistentem Schnitt als Ausgangspunkt) gewählt wird. Die Komplexität einer solchen Berechnung wäre allerdings zu hoch und die Modellierung einer vergleichbaren Varianz aufwändig. Deshalb ist es sinnvoller, zunächst anhand einer Standardabschätzung den konsistenten Schnitt zu wählen, zu dem der Systemzustand zurück gesetzt wird. Anschließend kann mit der Bewertungsfunktion und der dynamischen Strategie eine Alternative gefunden werden. Die Abschätzung zur Wahl des konsistenten Schnitts basiert auf folgenden Schlussfolgerungen des bereits Erläuterten, die von einzelnen Ausnahmen abgesehen gelten:

1. Je älter der konsistente Schnitt, umso höher die mögliche Varianz.
2. Je älter der konsistente Schnitt, umso höher die Kosten.
3. Der Zuwachs der Kosten von einem jüngeren konsistenten Schnitt zum nächst älteren kann durch die Ausführungszeit t zwischen den beiden konsistenten Schnitten approximiert werden.

Die Ausführungszeit, die bei einer Ausführung ab C_1 bis zur erfolgreichen Speicherung von C_2 vergehen würde, also die in der Abbildung als Differenz $t_2 - t_1$ zu

errechnende logische Zeit, kann zur Abschätzung der zusätzlichen Kosten verwendet werden. Je nachdem, wie lange die Ausführung vom konsistenten Schnitt C_2 bis zum Fehler dauert, ist es im Verhältnis rentabler öfter oder weniger oft eine Alternative von C_2 aus zu versuchen, bevor auf C_1 gewechselt wird. Ist t_f der Zeitpunkt des Fehlerfalls, so könnte eine sehr einfache Entscheidungsgrundlage wie folgt definiert werden: Rücksetzen auf

- C_2, solange gilt $anz_{C_2} \cdot (t_f - t_2) \leq (t_f - t_1)$, mit $anz_{C_2} :=$ Anzahl vergangener Rollbacks zu C_2.
- C_1, sonst.

Dies kann induktiv für beliebige aufeinander folgende konsistente Schnitte angewendet werden. Neben der Einfachheit des Verfahrens ist es darin begründet, dass die kürzere Laufzeit für einen Versuch der erneuten Ausführung sich nur bis zu einer gewissen Schranke rentiert, anschliessend überwiegt die höhere Varianz und damit höhere Wahrscheinlichkeit der Fehlertoleranz des teureren Schnitts. Der Ansatz setzt voraus, dass je weiter ein konsistenter Schnitt in der Vergangenheit liegt, umso größer ist sein Abstand zum folgenden konsistenten Schnitt, da die Checkpoints altern und gelöscht werden.

Dies impliziert die Strategie der Verwaltung von Checkpoints, also wann Checkpoints bzw. ganze konsistente Schnitte verworfen werden (vgl. Abschnitt 5.2.4). Für das hier vorgestellte Verfahren ist es sinnvoll, die Checkpoints derart zu verwerfen, dass die Abschnitte zwischen zwei konsistenten Schnitten exponentiell wachsen. Dies begründet sich wiederum genau darin, dass die Strategie zur Toleranz sich dynamisch ändert, also zunächst die Kosten gering hält (junger konsistenter Schnitt), dann aber bei wiederholt auftretendem Fehler die Varianz wichtiger bewertet wird (weit zurück liegender Schnitt). Für die Wahl des konsistenten Schnitts können allerdings weit aufwändigere Verfahren angewandt werden; dies ist auf Grund des zu leistenden Aufwands jedoch in der Regel nicht rentabel, da ohne detaillierte Informationen über den Fehler die Erfolgswahrscheinlichkeit nicht maßgeblich beeinflusst wird.

Eine Garbage Collection, in der Checkpoints derart verworfen werden, sodass die Abstände zwischen den verbleibenden konsistenten Schnitten exponentiell wachsen, ist schwer zu realisieren. Eine grobe Annäherung an diese Forderung ist jedoch relativ einfach realisierbar. Wird die Hauptkomponente als Maßstab verwendet und die konsistenten Schnitte gewählt, die durch die Checkpoints der Hauptkomponente festgelegt sind, so können die Abstände der Checkpoints der Hauptkomponente als Annäherung für die Abstände der konsistenten Schnitte verwendet werden. Der Garbage Collector muss demnach nur in bestimmten

Zeitabständen prüfen, ob einer der Checkpoints der Hauptkomponente entfernt werden kann, sodass trotzdem die sich ergebenden Intervalle der Checkpoints nicht stärker als exponentiell wachsen. Nahe liegender Weise könnte diese Überprüfung nach dem Erstellen eines neuen Checkpoints der Hauptkomponente stattfinden. Wird dann ein Checkpoint der Hauptkomponente gelöscht, so können auch andere Checkpoints verworfen werden, die zum konsistenten Schnitt dieses Checkpoints gehörten.

Bei einer tatsächlichen Realisierung dieser Garbage Collection müssten noch einige implementierungsabhängige Details beachtet werden, auf die hier nicht näher eingegangen wird. Entscheidend für das Fehlertoleranzverfahren ist lediglich, dass auch ein Rollback zu älteren konsistenten Schnitten durchgeführt werden kann. Die Garbage Collection dient dazu, benötigten Speicher für Checkpoints zu reduzieren, verbraucht aber selbst Rechenzeit. Diese Abwägung kann durch komplexere bzw. einfachere Strategien je nach Systemanforderungen verlagert werden.

6.2.6 Gesamtverfahren im Überblick

In diesem Abschnitt wird das Erklärte nochmals zusammengefasst und das gesamte Konzept im Überblick dargestellt. Für die Fehlertoleranz durch automatisierte Diversität auf Management-Ebene muss bei der Systemkonstruktion zunächst ein oder mehrere Entscheidungsräume identifiziert werden, die für das System abhängig von der Anwendungs-Programmiersprache geeignet sind. Geeignet bedeutet, dass das Auftreten der nach Erfahrungswerten Sprach- und Systemtypischen, verbleibenden konkretisierungsabhängigen Fehler durch die Entscheidungen dieser Entscheidungsräume beeinflusst wird:

- Identifikation des Entscheidungsraums

Für jeden Entscheidungsraum müssen Varianz und Kostenfunktion definiert werden, dabei spiegelt die Varianz den Unterschied im Ausführungsablauf wieder:

- Spezifikation der Varianz
- Spezifikation der Kostenfunktion

Je nach Systemanforderungen muss eine dynamisch veränderliche Strategie festgelegt werden, die Varianz und Kostenfunktion zu einer Bewertungsfunktion kombiniert:

- Realisierung von dynamischer Strategie als Bewertungsfunktion

Um Alternativen tatsächlich zu generieren, kann entweder ein gängiges Suchverfahren oder eine algorithmische Lösung programmiert werden, die generierten Alternativen müssen der Strategie entsprechen:

- Festlegung von Suchverfahren oder Algorithmus zur Konstruktion von Alternativen

Das Management muss um die Verfahren zur Generierung und Bewertung von Alternativen erweitert werden. Ferner müssen Mechanismen zur Informationssammlung und zur Umsetzung gewählter Alternativen in das Management integriert werden. Der Einsatz von Fehlererkennungsverfahren und einem Checkpoint-Rollback-Verfahren, wie in der vorangehenden Kapiteln beschrieben, sind Voraussetzung. Die Strategie zur Entscheidung muss letztlich noch erweitert werden, sodass dynamisch nicht immer der gleiche konsistente Schnitt als Ausgangspunkt verwendet wird, sondern iterativ nach Fehlversuchen ein älterer konsistenter Schnitt bevorzugt wird.

Ein beispielhafter Ablauf des Systems mit Fehlertoleranz durch automatisierte Diversität auf Management-Ebene könnte dann wie in Abbildung 6.16 dargestellt stattfinden. In der linken Hälfte ist die Ausführung dreier Akteure a_1, a_2 und a_3 als gerichteter Graph visualisiert, wobei die Kreise zusammengefassten Ereignissen der Akteure des ANV entsprechen. Rechts sind ausgewählte Management-Entscheidungen erläutert. Zunächst läuft das System fehlerfrei, das Management trifft Entscheidungen zur Platzierung der Akteure (Entscheidungen 1 und 3) sowie zur Erstellung von zwei konsistenten Schnitten C_1 und C_2 als Rollback-Punkte. Die Stellenzuweisungen sind rechts visualisiert, die durch Checkpoints der Akteure gebildeten konsistenten Schnitte als Balken im gerichteten Graph eingezeichnet.

Zum Zeitpunkt t_3 stellt das Management einen Fehlzustand von Akteur a_2 fest und muss die beschriebenen Schritte des Verfahrens zur Tolerierung einleiten. Zunächst wird der konsistente Schnitt C_2 als Rücksetzpunkt gewählt. Dann wird eine neue Stellenzuweisung als Alternative gewählt, bei der lediglich Akteur a_2 migriert werden muss. Zum Zeitpunkt t_4 beginnen Akteure a_1 und a_2 ihre Berechnungen wieder ab dem gespeicherten Zustand ihrer Checkpoints. Akteur a_3 wurde nicht beeinflusst, da der Akteur nicht im konsistenten Schnitt C_2 enthalten ist. Zum Zeitpunkt t_5 tritt wieder ein Fehlzustand bei Akteur a_2 auf.

Das Management erkennt, dass die Symptome analog zu dem Fehlzustand von Zeitpunkt t_3 sind und entscheidet vom gleichen Checkpoint eine Alternative mit

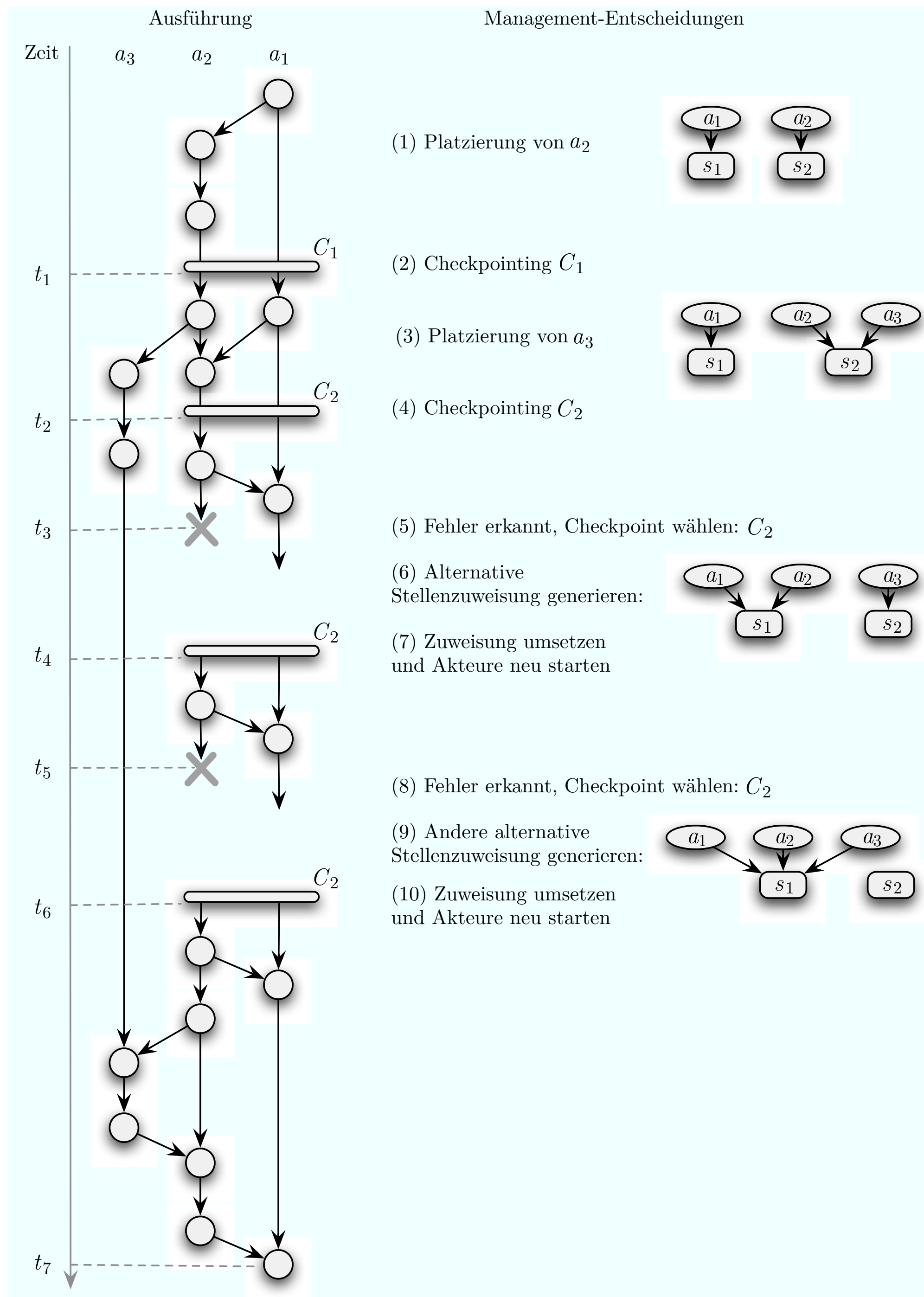


Abbildung 6.16.: Möglicher Ablauf eines Systems mit Fehlertoleranz durch automatisierte Diversität

höherer Varianz zu starten; es wird in Management-Entscheidung (9) die Alternative generiert, in der alle Akteure auf Stelle s_1 ausgeführt werden. Zum Durchsetzen dieser Entscheidung werden Akteur a_1 und a_2 migriert, dies ist in der Abbildung nicht visualisiert. Zum Zeitpunkt t_6 starten Akteure a_1 und a_2 wieder von ihrem gespeicherten Zuständen, diesmal kann das System korrekt seine Berechnungen ausführen und zum Zeitpunkt t_7 terminieren.

6.3 Diskussion

Folgend werden verschiedene Gesichtspunkte des vorgestellten Konzepts, wie Leistungsfähigkeit, weiterführende Möglichkeiten zur Verbesserung der Entscheidungen und die Zusammenhänge mit verwandten Arbeiten diskutiert.

6.3.1 Leistungsfähigkeit

Die Leistungsfähigkeit eines Fehlertoleranzverfahrens kann anhand der Menge tolerierbarer Fehler beurteilt werden. Bei dem vorgestellten Verfahren können jedoch nicht bestimmte Fehlerarten grundsätzlich toleriert werden oder nicht, sondern es besteht eine Wahrscheinlichkeit, mit der Fehler toleriert werden. Zunächst wird untersucht, welche Fehlerarten mit automatisierter Diversität auf Management-Ebene tolerierbar sind. Im Anschluss werden einige Überlegungen zur Effizienz angestellt.

6.3.1.1 Tolerierbare Fehler

Wie in Abschnitt 6.1.1 erläutert, ist die Kernidee der Fehlertoleranz durch automatisierte Diversität auf Management-Ebene, jene Softwarefehler zu tolerieren, die durch Qualitätssicherungs-Maßnahmen häufig nicht zu entdecken sind. Dazu zählen Fehlerursachen, deren Fehlerzustände nur auftreten, wenn bestimmte Systemzustände oder Bedingungen vorliegen, wie beispielsweise ausgehende Ressourcen oder Race Conditions (vgl. [Gra86]). Tatsächlich sind allerdings alle Fehler durch das Verfahren tolerierbar, die *konkretisierungsabhängig* sind (vgl. Definition 6.1).

Darunter können neben Softwarefehlern auch beispielsweise permanente oder transiente Hardwarefehler fallen, falls das Management die Entscheidungsmöglichkeit hat, das System ohne die ausgefallene Hardware auszuführen. Dies trifft

sowohl auf eine ausgefallene Stelle zu, falls das System auf mehr als einer Stelle ausgeführt wird, als auch auf einen defekten Arbeitsspeicher, falls das System so übersetzt werden kann, dass der defekte Speicherbereich nicht genutzt wird. Grundsätzlich können mit dem Konzept Soft- und Hardwarefehler in Systemen toleriert werden, falls für die fehlerhaften Systeme folgende Eigenschaft gilt:

Es existiert mindestens eine Kombinationsmöglichkeit aller zur Konkretisierung notwendigen Management-Entscheidungen, mit der das System terminiert und in diesem Zustand das korrekte Ergebnis vorliegt.

Welche Fehler konkret von einer Implementierung des Konzepts toleriert werden, hängt von den genannten Entscheidungen ab, die zur Konstruktionszeit des Systems zu treffen sind. Insbesondere sind dies die Wahl der Entscheidungsräume und die Strategie zur Auswahl von Alternativen. Folgend werden Beispiele beschrieben, an denen die weitreichenden Auswirkungen der Stellenzuweisung als Entscheidungsraum durch ihren zeitlichen Einfluss deutlich werden.

Offensichtlich sind Fehler in der Synchronisation prädestiniert für zeitlichen Einfluss. Im INSEL-Beispiel aus Anhang C, anhand dessen in Abschnitt 4.1.3 die Fehlererkennung durch Timing-Tests erklärt wurde, tritt abhängig vom zeitlichen Ablauf der K-Order-Aufrufe der Akteure eine Verklemmung auf. Wird der erste `lock`-Aufruf eines der M-Akteure vollständig und der zweite Aufruf noch zumindest bis zur Annahme durch den K-Akteur ausgeführt, bevor der erste `lock`-Aufruf des anderen Akteurs den K-Akteur erreicht[4], so tritt keine Verklemmung auf.

Die Stellenzuweisung beeinflusst dieses zeitliche Verhalten maßgeblich: Werden die M-Akteure `System`, `act_1` und `act_2` auf einer Stelle s_1 realisiert, die K-Akteure `mutex_1` und `mutex_2` dagegen auf Stelle s_2, so benötigen beide M-Akteure für ihre Aufrufe einen Nachrichtenversand über das Netzwerk, so ist das Auftreten der Verklemmung relativ wahrscheinlich: Der Nachrichtentransport zum K-Akteur und zurück dauert ein Vielfaches der Zeit, die jeder Akteur vom Scheduling zur Verfügung gestellt bekommt. Während also einer der beiden M-Akteure blockiert ist, weil sein erster K-Order-Aufruf abgeschickt wurde, wird mit hoher Wahrscheinlichkeit auch der zweite Akteur seinen ersten K-Order-Aufruf abschicken und die Verklemmung ist nicht mehr abzuwenden.

Werden dagegen die M-Akteure `System` und `act_1` auf Stelle s_1, und `act_2` zusammen mit den K-Akteuren `mutex_1` und `mutex_2` auf Stelle s_2 realisiert, so ist ein Ablauf ohne Verklemmung deutlich wahrscheinlicher. Angenommen

[4]Erreichen bedeutet, dass der Aufruf in Form einer Nachricht vom Kommunikator an den K-Akteur übergeben wird.

die beiden Akteure `act_1` und `act_2` starten die Ausführung ihrer kanonischen Operation gleichzeitig, dann kommt es nicht zu einer Verklemmung, falls die drei Task-Wechsel auf Stelle s_2 weniger Zeit brauchen als die Übertragung des K-Order-Aufrufs über das Netzwerk. Auch wenn der tatsächliche Ablauf von vielen weiteren Faktoren, wie beispielsweise Scheduling, sonstiger Prozessor- und Netzauslastung sowie der Prozesserzeugung abhängt, so ist dennoch die Wahrscheinlichkeit für eine Verklemmung geringer als bei der ersten Stellenzuweisung.

Neben klassischen Race Conditions, wie dem gezeigten Fehler der Synchronisation, können allerdings auch andere Fehlerzustände durch zeitliche Änderungen des Ausführungsablaufs umgangen werden. Koren und Krishna haben ein Beispiel skizziert, bei dem ein Prozess drei Gleitkommazahlen $a+b+c$ addiert (vgl. [KK07] , S.156 f.). Auf Grund der Assoziativität der Addition ist theoretisch das Ergebnis unabhängig von der Reihenfolge, in der die Zahlen aufsummiert werden. Die technischen Beschränkungen der Hardware können bei Gleitkomma-Berechnungen allerdings durchaus einen Unterschied hervorrufen: Angenommen a sei $2.2e+20$, $b=5$ und $c=-2.2e+20$, dann kann abhängig von der Präzision (falls die Mantisse nur bis zu 20 Dezimalziffern aufnehmen kann, entspricht 66 Bit) die Addition $a+b=a$ ergeben, sodass das Gesamtergebnis von $a+b+c$ in dieser Reihenfolge gleich 0 und damit falsch ist. Wird dagegen zunächst $a+c$ berechnet und anschliessend erst b addiert, so ist das Ergebnis korrekt. Natürlich ist hier die rein mathematische Korrektheit gemeint, es gibt durchaus Anwendungsfälle, in denen dieser Fall als unwichtiger Rundungsfehler und nicht als Versagen zählt. Durch die technisch beschränkte Genauigkeit von Gleitkommazahlen ist diesem Problem auch nicht generell beizukommen. Dennoch zeigt sich an diesem Beispiel, dass durch die Rundungsfehler die Reihenfolge der Berechnung relevant sein kann. Genau diese Reihenfolge kann durch den vorgestellten Ansatz beeinflusst werden.

Angenommen ein K-Akteur realisiert die beschriebenen Additionen, wobei die Parameter a, b und c jeweils nebenläufig berechnete Zwischenergebnisse von M-Akteuren sind, die dem K-Akteur per K-Order übergeben werden, so würde die tatsächliche Reihenfolge der K-Order-Aufrufe von der Stellenzuweisung beeinflusst. Es könnte demnach durchaus sein, dass bei gewissen Stellenzuweisungen die eingehende Reihenfolge a, b, c, bei anderen Stellenzuweisungen a, c, b ist.

Mit Hilfe des Konzepts der Verträge (vgl. Abschnitt 4.2) könnte beispielsweise in einer Vorbedingung überprüft werden, ob einer der beiden Fälle vorliegt, in denen der relative Rundungsfehler bei Gleitkomma-Berechnungen sehr hoch werden kann. Diese Fälle liegen erstens vor, wenn die Exponenten sehr weit auseinander liegen, wenn also beispielsweise die Differenz der Exponenten größer oder gleich der Mantisse ist, wie in obigem Beispiel. Zweitens treten hohe rela-

tive Rundungsfehler auf, wenn Gleitkommazahlen subtrahiert werden, die nahe beieinander liegen, wenn die Zahlen also bis zu einer bestimmten Stelle identisch sind. Tritt einer dieser Fälle ein, so könnte dies als Fehler betrachtet und entsprechend ein Rollback veranlasst werden. Ob dieser Fehler allerdings toleriert werden kann, hängt davon ab, ob analog zu dem skizzierten Beispiel eine Ausführungsreihenfolge existiert, bei der die Rundungsfehler unter der Schranke bleiben.

Dieses Beispiel ist zwar – wie von Koren und Krishna angemerkt – etwas künstlich konstruiert, es steht jedoch als Repräsentant für eine Vielzahl möglicher Abhängigkeiten, die in einer abstrakten Spezifikation nicht enthalten sind und sich durch die technische Realisierung ergeben. Häufig entscheidet in diesen Fällen die zeitliche Verzahnung der nebenläufigen Berechnungen über das Auftreten von Fehlerzuständen. In diesen Fällen kann eine geänderte Stellenzuweisung unter Umständen den Fehler tolerieren. Ähnliche Beispiele lassen sich auch für den Überlauf primitiver Typen, wie beispielsweise `Integer` konstruieren. Werden zum Beispiel viele positive und negative Werte aufsummiert, so kann die Reihenfolge der Ausführung dafür ausschlaggebend sein, ob ein Überlauf stattfindet oder nicht.

6.3.1.2 Effizienz

Der vorgestellte Ansatz basiert darauf, dass ein nicht genauer identifizierter Fehler mit einer gewissen Wahrscheinlichkeit durch eine Veränderung der Ausführungsabläufe bzw. der Konkretisierungsdetails nicht wieder auftritt. Das Verfahren kann also beliebig viele Versuche benötigen, einen einzigen Fehler zu tolerieren, und ist demnach für Performanz-kritische Systeme oder Echtzeitsysteme nur bedingt einsetzbar. Ist die Hauptanforderung des Systems jedoch nicht die Performanz, sondern die Zuverlässigkeit, so bietet das vorgestellte Verfahren die Möglichkeit, verschiedenste Fehler zu tolerieren, auch wenn dies unter Umständen eine lange Laufzeit benötigen kann, da der gesamte Raum an Kombinationen von Management-Entscheidungen enorm groß ist. Tatsächlich ist allerdings nach ausgiebigen Qualitätssicherungs-Maßnahmen davon auszugehen, dass die konkretisierungsabhängigen Fehler nur bei sehr wenigen, besonderen Systemzuständen auftreten, da sie sonst in der Testphase aufgetreten wären. Insofern muss das System-Management häufig nur eine oder sehr wenige Alternativen der Ausführung durchprobieren, um den Fehler zu tolerieren. Demnach kann die Performanz im Worst-Case beliebig schlecht sein, im Durchschnitt sollte sie allerdings akzeptabel sein.

Die Effizienz von Implementierungen des vorgestellten Konzepts hängt in erster Linie davon ab, welche Entscheidungen getroffen werden. Je mehr Möglichkeiten das Management hat, umso schwieriger wird es, eine automatisierte Entscheidung zu treffen, die einerseits den Fehler toleriert und andererseits möglichst wenig Kosten mit sich bringt. Neben den vorgestellten dynamischen Strategien und Möglichkeiten, Entscheidungen verschiedener Entscheidungsräume zu kombinieren sowie die Wahl des konsistenten Schnitts für die Rückwärtsbehebung iterativ zu ändern, könnten die gemachten Erfahrungen des Managements genutzt werden, um zukünftig bessere Entscheidungen zu treffen.

6.3.1.3 Verbesserung der Alternativenwahl

Im vorgestellten Verfahren werden lediglich bei wiederholtem Auftreten des gleichen Fehlers (genauer der gleichen Fehlersymptome) iterativ andere Entscheidungen getroffen und sozusagen das Wissen über vergangene Versuche bei der Entscheidungsfindung berücksichtigt. Das Management könnte allerdings insgesamt aus den Erfahrungen lernen und für vergleichbare Fehler somit ad hoc eine bessere Entscheidung treffen, falls Fehler mit ähnlichen Fehlersymptomen in der Vergangenheit bereits toleriert wurden. Es müssten demnach die erfolgreichen und nicht erfolgreichen Versuche der Toleranz eines Fehlers genutzt werden, um die Entscheidungsfindung des Managements für vergleichbare Fehlersymptome zu optimieren.

Ferner wurden bei dem vorgestellten Konzept gezielt keine Informationen über die Art des erkannten Fehlers und dessen Ausbreitung genutzt, abgesehen vom zugehörigen Akteur. Diese Vorgehensweise vereinfacht einerseits die Fehlererkennung und benötigt keine Fehlerlokalisierung, andererseits könnte durch den Einbezug von Fehlerinformationen die Wahl von Alternativen der Konkretisierungsentscheidungen verbessert werden. Einerseits könnten Informationen über die Fehlerausbreitung bei der Wahl eines konsistenten Schnitts, zu dem das System zurück gesetzt werden soll, berücksichtigt werden. Ferner könnten Informationen über die Fehlersymptome bei der Wahl eines Entscheidungsraums genutzt werden, um gezielter *diejenigen* Entscheidungen zu variieren, die mit hoher Wahrscheinlichkeit Einfluss auf das Auftreten des Fehlers haben.

6.3.2 Fehlerbehebung durch Informationen über Alternativen

Das entwickelte Konzept dient in erster Linie der Toleranz von Softwarefehlern. Grundsätzlich ist der Softwareentwickler zwar daran interessiert, dass die Software korrekt ausgeführt wird, allerdings ist auf Grund der Performanz tolerierender Verfahren die Fehlerbehebung auf lange Sicht gesehen wichtiger als die Toleranz. Deshalb können die Informationen über die Symptome aufgetretener Fehler zusammen mit den Informationen über Management-Entscheidungen, die zu deren Toleranz geführt haben, zur Lokalisierung und Behebung der Fehler genutzt werden. Die relevanten Informationen müssen für den Einsatz des Toleranz-Verfahrens bereits gesammelt und protokolliert werden (vgl. Abschnitt 6.2.4.3) und können zudem für die Qualitätsverbesserung der Software persistent gespeichert und später analysiert werden.

6.4 Verwandte Arbeiten

Einleitend wurden in diesem Kapitel bereits diversitäre Programmierung als Ansatz zur Fehlertoleranz beleuchtet und die Kritikpunkte hervorgehoben. Das in dieser Arbeit vorgestellte Konzept unterscheidet sich von diversitärer Programmierung grundlegend, da erstens nicht verschiedene Versionen einer Software erstellt werden, sondern lediglich die Freiheiten der Konkretisierung genutzt werden, um unterschiedliche Ausführungsabläufe zu erhalten, und zweitens die Diversität automatisch generiert, nicht von Hand programmiert wird.

Ein Ansatz zur automatisierten Generierung diversitärer Software-Versionen, ohne die eingangs erläuterten Kritikpunkte diversitärer Programmierung, ist von R. Feldt auf Basis genetischer Programmierung untersucht worden (vgl. [Fel98]). Das Prinzip genetischer Programmierung funktioniert analog zu genetischer Suche (vgl. Abschnitt 6.2.4.1), wobei durch Mutation und Kreuzung von Programmen neue Programme geschaffen werden. Genetische Programmierung kann erfolgreich für kleine Programme eingesetzt werden, scheitert jedoch bei komplexer Software, sodass dieser Ansatz für die betrachteten Softwaresysteme nicht in Frage kommt.

In Abschnitt 6.1.2.2 wurde erläutert, dass für den Bereich der Sicherheit einige aktuelle Arbeiten und Ansätze existieren, die automatisierte Diversität nutzen. Der Grundgedanke dieser Ansätze ist die Angreifbarkeit uniformer Systeme, da das Wissen über Details des Speicherlayouts oder Befehlssätze für viele Attacken

die Grundlage ist. Im Unterschied zum hier vorgestellten Ansatz ist nicht das Ziel, hintereinander verschiedene Ausführungsmöglichkeiten der gleichen Softwareteile eines Programms zu generieren, sondern verschiedene Instanzen des selben Programms auf verschiedenen Systemen zu erzeugen. Dennoch lassen sich viele Ideen des hier vorgestellten Ansatzes und der Verfahren für automatisierte Diversität im Sicherheitsbereich übertragen, sodass beide Ansätze zusätzliche Anstöße erhalten.

Wang et al. verfolgen für Message-Passing Systeme einen Ansatz, der zu dem hier vorgestellten Modell passt. Unter dem Namen *Progressive Retry* versuchen sie ebenfalls eine Rückwärtsbehebung mit Diversität, wobei in ihrem Ansatz ausschliesslich die Reihenfolge von Nachrichten variiert wird (vgl. [WHFK97]). Es werden während der normalen Ausführung Checkpoints gespeichert und ausgetauschte Nachrichten der Anwendung über ein *Logging*-Verfahren gesichert. Wird ein Fehler erkannt, so wird der betroffene Prozess zum letzten Checkpoint zurückgesetzt und die von ihm empfangenen Nachrichten in veränderter Reihenfolge reproduziert. Tritt der Fehler erneut auf, so wird das Verfahren vom vorletzten Checkpoint des Prozesses wiederholt. Die praktische Umsetzung von Wang et al. zeigt, dass mit diesem Verfahren bei Systemen mit intensiver Kommunikation Heisenbugs und zum Teil selten auftretende Bohrbugs toleriert werden können. Diese Ergebnisse unterstützen das hier vorgestellte Konzept zur Fehlertoleranz.

6.5 Zusammenfassung

Zuverlässige verteilte Systeme können konstruiert werden, indem fehlervermeidende und fehlertolerierende Verfahren kombiniert werden. Gerade konkretisierungsabhängige Fehler sind durch vermeidende Verfahren nur schwer und in komplexen Systemen nicht gänzlich eliminierbar. Das Auftreten dieser Fehler ist von Konkretisierungs-Entscheidungen abhängig; deshalb bietet sich ein Toleranzverfahren an, das Diversität in die Konkretisierungs-Entscheidungen integriert.

In diesem Kapitel wurde ein Konzept zur Toleranz von konkretisierungsabhängigen Fehlern mittels automatisierter Diversität auf Management-Ebene erarbeitet. Das Konzept baut auf einem Fehlererkennungsverfahren und einem Checkpoint-Rollback-Verfahren auf, dazu dienen die in Kapitel 4 und 5 vorgestellten Arbeiten. Ist ein Fehler erkannt, so wird mittels Rückwärtsbehebung der betroffene Systemteil auf einen gespeicherten, konsistenten Zustand gesetzt.

Um konkretisierungsabhängige Fehler zu tolerieren, wird der Systemteil nicht identisch ausgeführt, sondern die Ausführungsabläufe im Rahmen der Konkreti-

sierungsentscheidungen des Managements verändert. Diese geänderte Ausführung führt mit einer gewissen Wahrscheinlichkeit dazu, dass die von bestimmten Systemzuständen oder zeitlichen Abläufen abhängigen Fehler nicht erneut auftreten. Es wurde ein Modell zur Bewertung verschiedener Alternativen eines Entscheidungsraums anhand ihrer Kosten und Varianz erarbeitet, und Suchverfahren zur Generierung solcher Alternativen auf Basis der Bewertung vorgestellt.

Das Konzept wurde allgemein für verteilte, nebenläufige Systeme mit beliebigen Entscheidungsräumen beschrieben und am Beispiel des MoDiS-Projekts mit dem Entscheidungsraum der Stellenzuweisung erläutert. Die Stellenzuweisung ermöglicht sowohl eine Beeinflussung der Ressourcen-Auslastung als auch eine Änderung der zeitlichen Abläufe nebenläufiger Prozesse und hat damit ein hohes Potential, die erklärten konkretisierungsabhängigen Fehler zu tolerieren. Die Stärken und Schwächen des Ansatzes wurden diskutiert und ein Ausblick auf verwandte Arbeiten sowie weiterführende Forschung gegeben.

Kapitel 7

Zusammenfassung und Ausblick

Inhalt

7.1. Zusammenfassung 211
7.2. Diskussion 214
7.3. Ausblick 217

In diesem Kapitel werden wesentliche Ergebnisse der Arbeit nochmals zusammengefasst und diskutiert. Insbesondere kapitelübergreifende Schlussfolgerungen bzgl. der Konstruktion verteilter, nebenläufiger Systeme sowie bzgl. Fehlertoleranz in nebenläufigen Systemen werden aus den Ergebnissen und Erfahrungen dieser Arbeit gezogen. Darauf folgt eine Auseinandersetzung mit unbeantworteten Fragestellungen, die in weiterführenden Arbeiten auf Basis der Ergebnisse der vorliegenden Dissertation behandelt werden sollten.

7.1 Zusammenfassung

Die Aufgabe der vorliegenden Arbeit bestand in der Konzeption eines Fehlertoleranzverfahrens für verteilte, nebenläufige Systeme am Beispiel des Projekts MoDiS. Es wurden Mechanismen zur Fehlererkennung und Rückwärtsbehebung an das MoDiS-Systemmodell angepasst und in das Management integriert. Ferner wurde ein Konzept zur Toleranz von Hard- und insbesondere Softwarefehlern erarbeitet, das auf Diversität basiert, diese aber im Management automatisiert umsetzt.

7.1.1 Fehlererkennung

Zur Fehlererkennung wurde zunächst die Erfassung konsistenter Sichten auf Basis kausaler Abhängigkeiten erarbeitet. In Abschnitt 4.1 ist die Erweiterung von Lamports Happened-Before-Relation um die Abhängigkeiten eines dynamischen Prozessmodells und eines gemeinsamen, verteilten Speichers beschrieben. Geeignete Realisierungsmöglichkeiten zur Erfassung der partiellen Ereignisordnung zur Laufzeit, die nicht den Schwächen von Vektoruhren unterliegen, wurden erarbeitet und in das Management integriert. Die zur Laufzeit erfassten Ereignisordnungen werden mit den zur Übersetzungszeit generierten Ereignis-Verbänden verglichen, um Timing-Fehler zu erkennen.

Um eine große Klasse an Fehlern zur Laufzeit erkennen zu können, müssen verschiedene Verfahren zur Fehlererkennung kombiniert werden. Semantische Softwarefehler sind schon im Quellcode der Anwendung enthalten, deshalb muss zum Erkennen derartiger Fehler eine vom Quellcode unterschiedliche Spezifikation der Anwendung vorhanden sein, auf deren Basis die Ist-Soll-Vergleiche zur Fehlererkennung durchgeführt werden können. Ein Beispiel dafür ist das Konzept der Verträge; die spezifizierten Vor-, Nachbedingungen und Invarianten stellen eine abstrakte Spezifikation der Funktionalität der Anwendung dar, deren Erfüllung zur Laufzeit überprüft werden kann. Über die Fehlererkennung hinaus bietet das Konzept der Verträge weitere Vorteile bei der Entwicklung von Software hoher Qualität, wie beispielsweise die verbesserte Lesbarkeit des Codes oder die Unterstützung des objektorientierten Designs.

Die Anpassung des Konzepts der Verträge an die Komponentenarten der MoDiS-Konzepte, sowie die Erweiterung der Sprache INSEL ist in Abschnitt 4.2 beschrieben. Neben Depots, die am ehesten dem Konzept eines Objekts entsprechen, können die Verträge auch geeignet für Ordern und Akteure genutzt werden. Es wurde veranschaulicht, dass nur eine vollständige Integration des Vertragskonzepts in Sprache und Übersetzer maximalen Nutzen für die Softwareentwicklung und die spätere Fehlererkennung zur Laufzeit bringt. Die automatisierte Prüfung der Verträge wird durch den Übersetzer vorbereitet und zur Laufzeit durchgeführt. Die Mächtigkeit der Vertragsprüfungen zur Fehlererkennung sind abhängig von der Spezifikation der Verträge: Je genauer die Korrektheit eines Ergebnisses durch eine einfach zu berechnende Probe überprüfbar ist, desto mehr Fehler können erkannt werden. Es wurde erläutert, dass sich verschiedenste Fehlerursachen als Verletzung von Vorbedingungen, Nachbedingungen oder Invarianten äussern können und somit eine große Klasse an Soft- und Hardwarefehlern mit diesem Verfahren erkennbar ist.

7.1.2 Fehlerbehebung

Die beiden im Rahmen dieser Arbeit integrierten Mechanismen zur Fehlererkennung sind als Basis für die Fehlerbehebung nutzbar. Um Fehler tolerieren zu können, wurde ein kommunikationsinduziertes Checkpoint-Rollback-Verfahren an die Kommunikation mit Operationen-orientiertem Rendezvous angepasst und algorithmisch umgesetzt, wie in Kapitel 5 beschrieben. Es wurde veranschaulicht, warum die in der Theorie performanten kommunikationsinduzierten Verfahren in der Praxis schlecht skalieren und zu hohen Overhead auf Grund erzwungener Checkpoints erzeugen.

In Abschnitt 5.2.5 wurde erklärt, wie durch eine Analyse des Kommunikationsverhaltens der Anwendung im Übersetzer der Overhead für verschiedene Positionierungs-Alternativen der freiwilligen Checkpoints der Akteure abgeschätzt und verglichen wird. Mit einem Greedy-Verfahren werden lokal günstige Positionierungen gewählt und der Akteur-Code um die Checkpoint-Aufrufe erweitert. Zur Laufzeit werden die Entscheidungen überprüft und ggf. freiwillige Checkpoints ausgelassen, somit können ungünstige Positionierungen auf Grund nicht statisch vorhersagbarer Ausführungsabläufe, wie im Fall von Schleifen oder bedingten Anweisungen, vermieden werden. Insgesamt werden freiwillige Checkpoints derart vorbereitet, dass in Abschnitten mit wenig Kommunikation das Overhead-minimale Checkpoint-Intervall gewählt wird; in Abschnitten mit Abhängigkeiten werden insgesamt längere Checkpoint-Intervalle bevorzugt, um erzwungene Checkpoints mit ungünstigen Intervallen zu vermeiden. Das Verfahren wurde durch Simulation mit dem kommunikationsinduzierten Checkpoint-Verfahren nach Briatico verglichen und es zeigte sich, dass der exponentielle Anstieg des Overheads bei Anwendungen mit intensiver Kommunikation vermieden werden kann (vgl. Abschnitt 5.3).

Auf den erarbeiteten Verfahren zur Fehlererkennung und Rückwärtsbehebung basierend wurde in Kapitel 6 ein Konzept motiviert und vorgestellt, Soft- und Hardwarefehler zu tolerieren, indem Ausführungsabläufe durch Variation von Management-Entscheidungen geändert werden. Es wurde erläutert, dass das Auftreten der Fehlerzustände vieler Softwarefehler von Ressourcenverwaltung und zeitlichen Abläufen der Ausführung und damit von der Summe der Management-Entscheidungen zur Konkretisierung abhängt. Tritt ein Fehler in einem Ausführungsablauf auf, so kann durch Rückwärtsbehebung die erneute Ausführung mit geänderten Management-Entscheidungen angestoßen werden. Auf Basis von Varianz und Kosten wird automatisiert durch das Management eine neue Alternative für die Konkretisierung generiert. Verschiedene Möglichkeiten zur Generierung dieser Alternativen, wie beispielsweise genetische Suchverfahren, wurden

vorgestellt und diskutiert. Es wurde am Beispiel der Zuweisung von Akteuren zu Stellen gezeigt, wie sich verschiedene Alternativen dieses Entscheidungsraums auf die zeitlichen Abläufe einer Ausführung auswirken können und dadurch das Auftreten von Fehlerzuständen umgangen werden kann.

Dieses Konzept ermöglicht die Toleranz allgemeiner Softwarefehler, falls zwei Bedingungen erfüllt sind: Erstens muss vom Management erkannt werden, dass ein (beliebiger) Fehlerzustand eingetreten ist. Zweitens muss es sich um einen konkretisierungsabhängigen Fehler handeln, d.h. es muss eine Kombination aus Konkretisierungs-Entscheidungen existieren, bei deren Ausführung der Fehlerzustand nicht auftritt. Im beschriebenen Konzept wurde am Beispiel von Race Conditions die Mächtigkeit dieses Ansatzes erläutert.

7.2 Diskussion

Die Zuverlässigkeit verteilter, nebenläufiger Systeme lässt sich verbessern, indem fehlervermeidende mit fehlertolerierenden Verfahren kombiniert werden. Fehlervermeidende Techniken der Softwareentwicklung sind dazu geeignet, eine Vielzahl an Softwarefehlern aufzudecken und zu entfernen, wie in Abschnitt 6.1.1 erläutert. Dennoch ist es im Fall komplexer, kooperativer Problemlösungen praktisch unmöglich, vollständige Fehlerfreiheit zu erreichen. Verbleibende Softwarefehler und umweltbedingte Hardwarefehler müssen daher zur Laufzeit toleriert werden.

Der Grund für Fehler in der Software liegt in der menschlichen Natur, Fehler zu machen. Ansätze zur Fehlertoleranz oder Fehlervermeidung, in die der Mensch involviert ist, indem er beispielsweise bei Model-Checking-Verfahren Prädikate zur Prüfung formulieren muss oder bei Exception-Mechanismen Routinen zur Behebung von Fehlern programmieren und mögliche Fehlerquellen vorherbestimmen muss, unterliegen generell der gleichen Fehlerquelle Mensch. Ferner ist menschlicher Einsatz stets mit hohen Kosten verbunden und deshalb meist nur bis zum Erreichen eines gewissen Grades an Zuverlässigkeit rentabel. Aus diesen Gründen sind automatisierte Verfahren im Management notwendig, um die Zuverlässigkeit der Systeme weiter zu steigern. Auch die Rentabilität spricht für Konzepte im Management, damit diese sich langfristig in der Praxis etablieren können.

Das in dieser Arbeit entwickelte Konzept zur Toleranz von Softwarefehlern auf Basis automatisierter Diversität im Management erfüllt die genannten Anforderungen, da eine große Klasse von Fehlern automatisiert toleriert werden kann, wie in Abschnitt 6.3.1 erläutert. Der Programmieraufwand ist dabei nur zur Konstruktion des Managements, nicht zur Anwendungsprogrammierung zu

leisten, dennoch orientieren sich die Entscheidungen zur Alternativenwahl an der Problemlösung. Viele Softwarefehler sind *konkretisierungsabhängige Fehler* (vgl. Abschnitt 6.1.1 und 6.2.5), es existiert also eine Kombination von Management-Entscheidungen, sodass die Fehlerzustände bei der Ausführung nicht auftreten. Das Konzept erfüllt demnach die Forderung, eine große Klasse nicht näher spezifizierter Softwarefehler tolerieren zu können, falls das Management in verschiedenen Entscheidungsräumen seine Konkretisierungs-Entscheidungen variieren kann. Andererseits unterliegt das erarbeitete Konzept nicht den Kritikpunkten diversitärer Programmierung, da einerseits kein zusätzlicher Aufwand bei der Anwendungsprogrammierung geleistet werden muss und andererseits nicht der Mensch direkt für die Diversität verantwortlich ist. Die Menge tolerierbarer Fehler hängt direkt davon ab, welche Fehler einerseits von den integrierten Fehlererkennungsmechanismen erkannt und welche Konkretisierungs-Entscheidungen andererseits variiert werden.

Das entwickelte Konzept dient der Toleranz von Fehlern in der Anwendung. Die Mechanismen zur Umsetzung der getroffenen Entscheidungen im Management können andererseits ebenfalls Fehler enthalten. Die Zuverlässigkeit dieses im Vergleich zu den auszuführenden Anwendungen gleich bleibenden Teils des Systems muss mit fehlervermeidenden Verfahren erreicht werden, um das beschriebene Konzept für die Toleranz von Fehlern in der Anwendung realisieren zu können.

7.2.1 Anwendungsorientiertes Management

Die Anwendungsorientierung beim Treffen von Entscheidungen im Management zeigte sich in den bisherigen Arbeiten des MoDiS-Projekts als entscheidend zur effizienten Ausführungssteuerung und Ressourcenverwaltung in verteilten, nebenläufigen Systemen. Auch in der vorliegenden Arbeit hat sich dieses Prinzip als gewinnbringend herausgestellt.

Beispielsweise hat die Analyse der vielen entwickelten Checkpoint-Rollback-Verfahren und deren Performanz gezeigt, dass die Verfahren anwendungsorientiert vom Management gesteuert werden müssen, um die Systemperformanz nicht zu sehr einzuschränken (vgl. Abschnitt 5.1.4). Dabei ist der Aufwand im fehlerfreien Betrieb vom Aufwand im Fall eines Rollbacks zu unterscheiden. Das in dieser Arbeit entwickelte Checkpoint-Verfahren zeigt, dass die Performanz kommunikationsinduzierter Verfahren durch die anwendungsorientierte Checkpoint-Positionierung entscheidend verbessert werden kann. Ein weiterer Vorteil des Ansatzes ist, dass die Berechnung zur Analyse und Entscheidungsfindung im Ma-

nagement nicht zur Laufzeit, sondern zur Übersetzungszeit aufgewendet werden, und daher durch diese meist aufwändigen Berechnungen keine negativen Auswirkungen auf die Performanz des Systems zur Laufzeit bestehen. Lediglich die Umsetzung der gewählten Entscheidungen, also das eigentliche Checkpointing, beeinträchtigt die Performanz des Systems.

Ein Beitrag der vorliegenden Arbeit ist die Erweiterung der von Lamport erarbeiteten Happened-Before-Relation. Die Abhängigkeiten durch Erzeugung und Auflösung von Prozessen eines dynamischen Prozesssystems sowie die Zugriffe auf gemeinsamen, verteilten Speicher können erfasst und zur Analyse der Systeme, sowie als Informationsquelle für das Management verwendet werden. Gerade die kausalen Abhängigkeiten der Akteure sind in zweierlei Hinsicht relevant für eine Vielzahl von Management-Entscheidungen. Erstens wird die Performanz der Ausführung entscheidend durch die Abhängigkeiten beeinflusst. Ob sich also eine Strategie zur Ressourcenverwaltung positiv oder negativ auf die Performanz auswirkt, hängt von der Anwendung und insbesondere von den Abhängigkeiten zwischen den einzelnen nebenläufigen Akteuren ab. Zweitens ist die Frage nach der Konsistenz eines Zustands bzw. einer Ausführung, die bei verschiedenen Problemstellungen, wie beispielsweise der Speicherung konsistenter Schnitte oder auch der Beobachtung verteilter Systeme immer wieder auftritt, durch diese Abhängigkeiten geprägt.

Sowohl bei der Erfassung der Abhängigkeiten einer verteilten Berechnung als auch bei deren Beachtung zur Wahrung der Konsistenz, zeigt sich der gemeinsame, verteilte Speicher als besonders problematisch. Die Erfassung und Konstruktion der konsistenten, partiellen Ereignisordnung einer Systemausführung ist durch den gemeinsamen, verteilten Speicher bereits deutlich komplexer und aufwändiger. Beim Einsatz eines Checkpoint-Verfahrens muss in Hinblick auf die Performanz gänzlich auf die Verwendung des gemeinsamen, verteilten Speichers verzichtet werden.

Bei dem erarbeiteten Konzept zur Toleranz von Fehlern spielen die Abhängigkeiten der verteilten Berechnungen während der Ausführung ebenfalls die entscheidende Rolle. Neben den in der Problemlösung durch Kooperation festgelegten Kommunikations-Abhängigkeiten ergeben sich zudem einerseits Abhängigkeiten durch die technische Realisierung (begrenzte Ressourcen, begrenzte Rechengenauigkeit, usw.) und andererseits ungewollte Abhängigkeiten durch Softwarefehler (beispielsweise fehlende Synchronisation). Es zeigt sich, dass diese zusätzlichen Abhängigkeiten die Auslöser für das Auftreten der Fehlerzustände konkretisierungsabhängiger Fehler sein können und wiederum die Management-Entscheidungen durch Beeinflussung dieser Abhängigkeiten auf das Auftreten der Fehlerzustände Einfluss nimmt.

Die Abhängigkeiten verteilter Problemlösungen sind demnach entscheidende Kriterien für geeignete automatisierte Entscheidungen im Management der Systeme. Für die Konstruktion verteilter, nebenläufiger Systeme lässt sich daraus folgern, dass die statisch erfassbaren Abhängigkeiten im Übersetzer aufbereitet und zusammen mit dynamischen, durch die Konkretisierung bedingten Abhängigkeiten als Informationsquelle für die Strategien zur Ressourcenverwaltung genutzt werden müssen, um geeignete Management-Entscheidungen treffen zu können. Auf Basis dieser Entscheidungen wird eine einfache und effiziente Nutzung der verteilten Hardware ermöglicht.

7.3 Ausblick

Mit dieser Arbeit sind ein zukunftsweisendes Konzept zur Toleranz von Softwarefehlern einerseits, sowie andererseits Verfahren zur Fehlererkennung und Rückwärtsbehebung für das Projekt MoDiS entstanden. Für den praktischen Einsatz dieser Verfahren und Konzepte verbleiben auf diese Arbeit aufbauende Aufgaben als Herausforderungen für die Zukunft, die nachfolgend formuliert werden.

Die spezifizierten Vor- und Nachbedingungen des Vertragskonzepts haben eine Reihe von Vorteilen für den Prozess der Softwareentwicklung und werden zur Fehlererkennung genutzt. Über dies hinaus sollte diese abstrakte Beschreibung der Funktionalität im Übersetzer als Informationsquelle genutzt werden, um einerseits Verifikationsverfahren und andererseits das automatisierte Management zu unterstützen. Das Potential zweier Sichten verschiedener Abstraktionsebenen auf eine Anwendung in Form der Implementierung und der Verträge zur automatisierten Analyse und Aufbereitung im Übersetzer wird bislang nicht geeignet genutzt.

Das vorgestellte Konzept der Fehlertoleranz durch automatisierte Diversität im Management wurde am Beispiel der Stellenzuweisung veranschaulicht. Langfristig muss das Konzept anhand verschiedener Entscheidungsräume experimentell getestet und bewertet werden, um Erfahrungen bzgl. der Mächtigkeit zur Toleranz von Softwarefehlern zu sammeln. Ein vielversprechender Entscheidungsraum wäre neben der Stellenzuweisung beispielsweise ein geändertes Speicherlayout, da einerseits fehlerhafter Speicherzugriff eine häufig auftretende Art von Softwarefehlern darstellt und andererseits Änderungen am Speicherlayout auch Vorteile für die Sicherheit mit sich bringen würden, indem beispielsweise typische Overflow-Attacken erschwert werden (vgl. Abschnitt 6.2.5). Nach experimentellen Erfahrungen muss die Fähigkeit zur Fehlertoleranz durch automatisierte Diversi-

tät anhand von Systemen aus der Praxis, die bekannte Softwarefehler enthalten, geprüft werden.

Die Fehlererkennung ist in der Regel deutlich einfacher als die Lokalisierung und Identifikation von Fehlern. Deshalb wurde in der vorliegenden Arbeit ein auf Wahrscheinlichkeit basierendes Konzept erarbeitet, das Fehler toleriert ohne genauere Informationen über deren Ausbreitung bzw. genauere Kenntnis der Fehlertypen zu nutzen. Der Vorteil dieses Ansatzes besteht darin, dass Fehlerlokalisierung und Identifikation nicht notwendig sind, um Fehler zu tolerieren. Dies wirkt sich allerdings zugleich nachteilig auf die Performanz aus, da unter Umständen viele Kombinationen von Management-Entscheidungen ausprobiert werden müssen, um einen Fehler zu tolerieren. Sind Informationen über Art und Ausbreitung eines aufgetretenen Fehlers vorhanden, so müsste das Konzept derart erweitert werden, dass auf Basis dieser Informationen die Management-Entscheidungen gezielt variiert werden, um effizient Fehler zu tolerieren.

Anhang A

INSEL-Beispiel

Folgend ist ein einfaches INSEL-Programm bestehend aus der Hauptkomponente `system` und einem K-Akteur `ca` aufgelistet. Der K-Akteur wird im Deklarationsteil der Hauptkomponente erzeugt und akzeptiert dann den Aufruf der K-Order `ko`. Diese wird vom Akteur `system` in dessen Anweisungsteil aufgerufen.

```
mactor system is

  cactor ca is
    j : integer;

    entry ko (i : integer) is
    begin
      i := 0x23;
    end;

  begin
    j := 0x666;
    select
       accept ko;
    end select;
  end ca;

k : integer;
c : ca;
```

```
begin
  J := 0x815;
  c.ko(4711);
end system;
```

Anhang B

Generierte Ereignisgraphen

In Abbildung B.1 ist ein Ereignisgraph dargestellt, wie er während der Ausführung des Beispielsprogramms aus Anhang A generiert wurde. Die protokollierten Informationen sind an den einzelnen Ereignis-Knoten vermerkt. Die etwas dicker dargestellten Kanten entsprechen dem Kontrollfluss eines Akteurs, also der *Program Order*. Die restlichen Abhängigkeiten gehen aus der Ereignisbeschriftung hervor.

1 ACTOR_INIT (H131.159.12.49:P0x4)
2 ACTOR_CREATE (H131.159.12.49:P0x5)
3 DSM_WRITE (O0x1001f238, 1, i:2069)
1 ACTOR_INIT (H131.159.12.49:P0x5)
2 DSM_WRITE (O0x1001f22c, 1, i:1638)
4 CORDER_CALLSEND (H131.159.12.49:O0x4, 4)
3 CORDER_CALLRECV (H131.159.12.49:O0x4, 4)
4 CORDER_EXEC (H131.159.12.49:P0x4, 4)
5 DSM_WRITE (O0x1001f238, 2, i:35)
6 CORDER_RETSEND (H131.159.12.49:P0x4, 4)
5 CORDER_RETRECV (H131.159.12.49:O0x4, 4)
6 DSM_READ (O0x1001f238, 2, i:35)
7 DSM_WRITE (O0x1001f230, 1, i:35)
7 ACTOR_TERM (H131.159.12.49:P0x5)
8 ACTOR_JOIN (H131.159.12.49:P0x5)
Powered by yFiles
9 ACTOR_TERM (H131.159.12.49:P0x4)

Abbildung B.1.: Ereignisgraph eines einfachen Systemablaufs

Anhang C

INSEL-Beispiel Verklemmung

Folgendes INSEL-Beispiel enthält eine Verklemmung. Die K-Akteur-Instanzen `mutex_1` und `mutex_2` realisieren zwei Mutexe für Ressourcen, die durch die K-Ordern `lock` und `unlock` gesperrt und freigegeben werden können. Die beiden M-Akteure `act_1` und `act_2` benötigen beide Ressourcen, fordern sie aber in unterschiedlicher Reihenfolge an. Wird nun zuerst der jeweils erste `lock`-Aufruf *beider* Akteure ausgeführt, so entsteht eine Verklemmung. Kann einer der Akteure *beide* `lock`-Aufrufe ausführen, bevor der andere den ersten `lock`-Aufruf macht, so kann das System korrekt terminieren.

```
mactor system is

  cactor mutex is
    ende : integer;

    entry lock is
    begin
      WRITESTRING(" locked!\n");
    end;

    entry unlock is
    begin
      WRITESTRING(" unlocked!\n");
    end;
```

```
    entry term is
    begin
      ende := 1;
      WRITESTRING(" mutex terminating!\n");
    end;

  begin
    ende := 0;
    while (ende = 0) loop
      select
         accept lock;
         accept term;
      end select;
      select
        accept unlock;
      end select;
    end loop;
  end mutex;

  mactor act_1 is
  begin
    mutex_1.lock;
    mutex_2.lock;
      -- ... kritischer Bereich ...
    mutex_2.unlock;
    mutex_1.unlock;
  end act_1;

  mactor act_2 is
  begin
    mutex_2.lock;
    mutex_1.lock;
      -- ... kritischer Bereich ...
    mutex_1.unlock;
    mutex_2.unlock;
  end act_2;

mutex_1 : mutex;
mutex_2 : mutex;

begin
```

```
  block actors_work is
  begin
    act_1;
    act_2;
  end actors_work;
  mutex_1.term;
  mutex_2.term;
end system;
```

Anhang D

Grammatik-Erweiterungen im Übersetzer für Verträge

Folgend werden die Erweiterungen der Grammatik für die Integration der Verträge in die Sprache INSEL und den Compiler gic aufgeführt (nach [May06]). Die Parser-Grammatik wurde für den `CONTRACT`-Block erweitert:

```
contract_part
  :  /* empty */
  |  CONTRACT BEGIN_T contract_cond_list END ';'
  ;
contract_cond_list
  :  /* empty */
  |  contract_cond_list contract_cond ';'
  |  contract_cond_list error ';'
  ;
contract_cond
  :  PRE expression | POST expression | INV expression ;
```

Dabei ist `expression` das Nicht-Terminal für einen Ausdruck in INSEL. In der abstrakten Syntax entsprechen dem `CONTRACT`-Block folgende Produktionen:

```
ContractCondList * ContractCond
ContractCond     ( Exp Statement ContractCondType File LineNo )
```

```
ContractCondType = PreCond | PostCond | InvCond
PreCond          ( )
PostCond         ( )
InvCond          ( )
```

Der Vertrag folgt dem Deklarationsteil der Komponente:

```
declaration_and_implementation_part
  :  IS declaration_part contract_part
     BEGIN_T statement_part END opt_identifier
  |  IS EXTERN STRING_LITERAL
  ;
```

Je nach Komponente befindet sich der Knoten des Typs `ContractCondList` im Knoten des Typs `DeclAndImplPart` (bei FS-Order, PS-Order und M-Akteuren) oder im Knoten des Typs `ObjectGen` (bei Depots und K-Akteuren):

```
RoutineGen      ( DefId RoutineType ParamList TypePart DaBody )
ObjectGen       ( DefId ObjectType ParamList DeclList
                  OptUsedId ContractCondList )
DaBody          = String | DeclAndImplPart
DeclAndImplPart ( DeclList StatementList OptUsedId
                  ContractCondList )
```

Der Parser muss für die Namensgebung für den Zugriff auf *old_values* über das @-Zeichen erweitert werden:

```
name
  :  IDENTIFIER
  |  name '.' IDENTIFIER
  |  name DEREF
  |  name '[' expression_list ']'
  |  '@' IDENTIFIER
  ;
```

Definitionsverzeichnis

2.1. Verteiltes System . . . 10

2.2. δ-Struktur . . . 21

2.3. σ-Struktur . . . 22

2.4. π-Struktur . . . 22

2.5. κ-Struktur . . . 22

2.6. α-Struktur . . . 23

2.7. λ-Struktur . . . 23

2.8. γ-Abbildung . . . 24

2.9. ϵ-Struktur . . . 24

4.1. Akteur-Ereignisfolge . . . 56

4.2. Schnitt . . . 56

4.3. Kausale Abhängigkeit . . . 56

4.4. Konsistenter Schnitt . . . 57

4.5. Happened-Before . . . 58

4.6. Erweiterte Happened-Before-Relation . . . 59

4.7. Erweiterte transitive Happened-Before-Relation . . . 60

4.8. Relation kausaler Abhängigkeiten 62
4.9. Eingeschränkte Relation kausaler Abhängigkeiten 64
4.10. Fernwirkung 88
5.1. Checkpoint-Intervall 115
5.2. Konsistente Zuordnung 116
5.3. Abhängigkeitsmenge 117
5.4. Rollback-Menge 117
6.1. Konkretisierungsabhängige Fehler 149
6.2. Varianz 163
6.3. Kostenfunktion 166
6.4. Effektive Parallelität 174

Abbildungsverzeichnis

2.1. Komponentenarten in MoDiS 15
2.2. Phasen der Ausführung der kanonischen Operation 17
2.3. Gerichteter Graph der partiellen Ordnung (...) 28
2.4. Beispiel für einen Attributierten Nebenläufigkeitsverband 31

3.1. Klassifizierung des Versagens 37

4.1. Gerichteter Graph der kausalen Abhängigkeiten (...) 54
4.2. Inkonsistenter Schnitt C_1 und konsistenter Schnitt C_2 (...) 58
4.3. Ereignisgraph ohne Abhängigkeiten des Speicherzugriffs 65
4.4. Ereignisgraph mit allen kausalen Abhängigkeiten 65
4.5. Ereignisse mit Vektoren zur Reflexion kausaler Abhängigkeiten . 67
4.6. Aufruf einer K-Order 70
4.7. Phasen der erweiterten kanonischen Operation 91

5.1. Domino-Effekt bei unkoordiniertem Checkpoint-Verfahren 100
5.2. Overhead-Ratio in Abhängigkeit zum Checkpoint-Intervall 108
5.3. Differenz der Steigung der Teilfunktionen (...) 109
5.4. Nachrichtenaustausch und abstrakte Sicht (...) 111
5.5. Nachricht m wird Teil eines konsistenten Schnitts 112
5.6. Abhängigkeiten durch dynamisches Prozesssystem 114
5.7. Konsistente Zuordnung von Checkpoints 116
5.8. Potentielle Checkpoints eines Akteurs ohne Abhängigkeiten . . . 130
5.9. Positionen für Potentielle Checkpoints eines Akteurs (...) 132
5.10. Durchschnittliche Laufzeit eines Beispiels (...) 142
5.11. Durchschnittliche Anzahl gespeicherter Checkpoints (...) 142
5.12. Durchschnittliche Laufzeit eines Beispiels (...) 143

5.13. Durchschnittliche Anzahl gespeicherter Checkpoints (...) 144

6.1. Konkretisierungspfade des Managements 157
6.2. Verteilung von Akteuren auf Stellen, Alternative W_1 160
6.3. Darstellung der Instruktionen im Zeitraster 162
6.4. Verteilung der Akteure auf Stellen, Alternative W_2 165
6.5. Verteilung der Akteure auf Stellen, Alternative W_3 165
6.6. Verteilung der Akteure des gespeicherten Zustands (...) 167
6.7. Umsetzung von Alternative W_2 (...) 168
6.8. ANV eines Beispielsystems 169
6.9. Gantt-Diagramme von zwei möglichen Zuweisungen (...) 172
6.10. Gantt-Diagramme von zwei möglichen Zuweisungen (...) 173
6.11. Geänderter ANV des Beispielsystems (...) 176
6.12. ANV des Beispielsystems ab gesichertem Zustand (...) 177
6.13. Prinzip der Populationsgenerierung (...) 185
6.14. Alternatives Speicherlayout mit Sicherheitspuffer 194
6.15. Wahl eines konsistenten Schnitts zur Rückwärtsbehebung 198
6.16. Möglicher Ablauf eines Systems mit Fehlertoleranz (...) 202

B.1. Ereignisgraph eines einfachen Systemablaufs 222

Literaturverzeichnis

[AC77] Avizienis, A. ; Chen, L.: *On the Implementation of N-version Programming for Software Fault Tolerance during Execution.* roc. the First IEEE-CS International Computer Software and Applications Conference (COMPSAC 77), Nov 1977. – Chicago [zitiert auf S. 50]

[AER+99] Alvisi, Lorenzo ; Elnozahy, E. ; Rao, Sriram ; Husain, Syed Amir ; De Mel, Asanka: An Analysis of Communication Induced Checkpointing. In: *Symposium on Fault-Tolerant Computing*, 1999, S. 242–249 [zitiert auf S. 106]

[Alv96] Alvisi, Lorenzo: *Understanding the message logging paradigm for masking process crashes.* Ithaca, NY, USA, Diss., 1996 [zitiert auf S. 99]

[AM98] Alvisi, Lorenzo ; Marzullo, Keith: Message Logging: Pessimistic, Optimistic, Causal, and Optimal. In: *IEEE Trans. Softw. Eng.* 24 (1998), Nr. 2, S. 149–159. – ISSN 0098–5589 [zitiert auf S. 99]

[APH93] A. Poetzsch-Heffter, T. E.: *The MAX System - A Tutorial Introduction.* 1993. – Technical Report TUM-I9307 [zitiert auf S. 30]

[Avi95] Avizienis, A.: The Methodology of N-Version Programming. In: Lyu, M. R. (Hrsg.): *Software Fault Tolerance.* Wiley, 1995, Kapitel 2, S. 23–46 [zitiert auf S. 51]

[BAFS05] Barrantes, Elena G. ; Ackley, David H. ; Forrest, Stephanie ; Stefanovi, Darko: Randomized instruction set emulation. In: *ACM Trans. Inf. Syst. Secur.* 8 (2005), Nr. 1, S. 3–40. – ISSN 1094–9224 [zitiert auf S. 155, 156]

[BCS84] Briatico, D. ; Ciuffoletti, A. ; Simoncini, L.: A distributed domino-effect free recovery algorithm. In: *Proceedings of the Fourth International Symposium on Reliability in Distributed Software and Databases* (1984), S. 207–215 [zitiert auf S. 104, 110, 139]

[BFFW01] Bain, Charles ; Faatz, Donald ; Fayad, Amgad ; Williams, Douglas: *Diversity as a Defense Strategy in Information Systems*. MITRE Report, 2001 [zitiert auf S. 155]

[BFMW04] Bartetzko, D. ; Fischer, C. ; Moller, M. ; Wehrheim, H.: Jass - Java with Assertions. In: *Electronic Notes in Theoretical Computer Science* 55 (2004), January, S. 1–15(15) [zitiert auf S. 82]

[BKL90] Brilliant, S.S. ; Knight, J.C. ; Leveson, N.G.: Analysis of faults in an version software experiment. In: *Software Engineering, IEEE Transactions on* 16 (1990), Feb., Nr. 2, S. 238–247 [zitiert auf S. 154]

[BL88] Bhargava, B. ; Lian, S.R.: Independent checkpointing and concurrent rollback for recovery-An optimistic approach. In: *Proc. IEEE Symp. Reliable Distributed Syst* (1988), S. 3–12 [zitiert auf S. 101]

[BM93] Babaoğlu, Özalp ; Marzullo, Keith: Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms / University of Bologna. University of Bologna, 1993. – Forschungsbericht [zitiert auf S. 55]

[BPS00] Bush, William R. ; Pincus, Jonathan D. ; Sielaff, David J.: A static analyzer for finding dynamic programming errors. In: *Softw. Pract. Exper.* 30 (2000), Nr. 7, S. 775–802. – ISSN 0038–0644 [zitiert auf S. 192]

[Bre01] Breitling, Max: *Formale Fehlermodellierung für verteilte reaktive Systeme*, Technische Universität München, Diss., 2001 [zitiert auf S. 35, 36]

[BS02] Broy, M. ; Siedersleben, J.: Objektorientierte Programmierung und Softwareentwicklung. In: *Informatik-Spektrum* 25 (2002), Nr. 1, S. 3–11 [zitiert auf S. 88]

[CCGMPR96] Carrillo-Castellon, M. ; Garcia-Molina, J. ; Pimentel, E. ; Repiso, I.: Design by contract in Smalltalk. In: *Journal of Object-Oriented Programming* (1996), December, Nr. 9(7), S. 23–28 [zitiert auf S. 82]

[CL85] CHANDY, K. M. ; LAMPORT, Leslie: Distributed snapshots: determining global states of distributed systems. In: *ACM Trans. Comput. Syst.* 3 (1985), Nr. 1, S. 63–75 [zitiert auf S. 62]

[CS98] CAO, Guohong ; SINGHAL, Mukesh: On Coordinated Checkpointing in Distributed Systems. In: *IEEE Trans. Parallel Distrib. Syst.* 9 (1998), Nr. 12, S. 1213–1225 [zitiert auf S. 104]

[DH98] DUNCAN, Andrew ; HOELZLE, Urs: Adding Contracts to Java with Handshake. 1998 (TRCS98-32). – Forschungsbericht [zitiert auf S. 82]

[DKL98] DESWARTE, Y. ; KANOUN, K. ; LAPRIE, J.-C.: Diversity against accidental and deliberate faults. In: *Computer Security, Dependability and Assurance: From Needs to Solutions, 1998. Proceedings*, 1998, S. 171–181 [zitiert auf S. 157]

[EAWJ02] ELNOZAHY, E. ; ALVISI, Lorenzo ; WANG, Yi-Min ; JOHNSON, David B.: A survey of rollback-recovery protocols in message-passing systems. In: *ACM Comput. Surv.* 34 (2002), Nr. 3, S. 375–408. – ISSN 0360–0300 [zitiert auf S. 99, 145]

[EJZ92] ELNOZAHY, E.N. ; JOHNSON, D.B. ; ZWAENEPOEL, W.: The performance of consistent checkpointing. In: *Proceedings of the 11th Symposium on Reliable Distributed Systems*, 1992, S. 39–47 [zitiert auf S. 102]

[EP99] ECKERT, C. ; PIZKA, M.: Improving Resource Management in Distributed Systems using Language-level Structuring Concepts. In: *The Journal of Supercomputing, Kluwer Academic Publishers, ISSN 0920-8542* (1999), Nr. 13, S. 35–55 [zitiert auf S. 12]

[Fel98] FELDT, R.: Generating diverse software versions with genetic programming: an experimental study. In: *IEEE Proceedings - Software* 145 (1998), Nr. 6, S. 228–236 [zitiert auf S. 154, 208]

[FH04] FRITZSCH, P. ; HAAS, S.: Migration von Aktivitaeten in vernetzten Linux-Systemen / Technische Universitaet Muenchen. 2004. – Forschungsbericht. – Systementwicklungsprojekt [zitiert auf S. 160]

[FSA97] FORREST, S. ; SOMAYAJI, A. ; ACKLEY, D.H.: Building diverse computer systems. In: *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, 1997, S. 67–72 [zitiert auf S. 155, 157]

[gcc] *GNU Compiler Collection.* : *GNU Compiler Collection.* – http://gcc.gnu.org/ [zitiert auf S. 30, 194]

[GK97] GIBBONS, Phillip B. ; KORACH, Ephraim: Testing Shared Memories. In: *SIAM J. Comput.* 26 (1997), Nr. 4, S. 1208–1244. – ISSN 0097–5397 [zitiert auf S. 63]

[Gol89] GOLDBERG, David E.: *Genetic Algorithms in Search, Optimization and Machine Learning.* Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1989. – ISBN 0201157675 [zitiert auf S. 185]

[Gra86] GRAY, Jim: Why Do Computers Stop and What Can Be Done About It? In: *Symposium on Reliability in Distributed Software and Database Systems*, 1986, S. 3–12 [zitiert auf S. 3, 149, 150, 192, 203]

[Haa05] HAAS, Sebastian: *Erfassung konsistenter Sichten von verteilten, nebenläufigen Systemen*, Technische Universität München, Diplomarbeit, November 2005 [zitiert auf S. 69]

[Him96] HIMSOLT, M.: *GML: Graph Modelling Language.* http://www.infosun.fmi.uni-passau.de/Graphlet/download/misc/GML.ps.gz, 1996. – Universität Passau [zitiert auf S. 72]

[HMR97] HÉLARY, Jean-Michel ; MOSTÉFAOUI, Achour ; RAYNAL, Michel: Virtual Precedence in Asynchronous Systems: Cencept and Applications. In: *WDAG '97: Proceedings of the 11th International Workshop on Distributed Algorithms.* London, UK : Springer-Verlag, 1997. – ISBN 3–540–63575–0, S. 170–184 [zitiert auf S. 105]

[Hol02] HOLZMANN, Gerard J.: The logic of bugs. In: *SIGSOFT Softw. Eng. Notes* 27 (2002), Nr. 6, S. 81–87. – ISSN 0163–5948 [zitiert auf S. 150]

[HW04] HENRY, Winston P. ; WINSTON, Patrick H.: *Artificial Intelligence.* Redwood City, CA, USA : Addison Wesley Longman Publishing Co., Inc., 2004. – ISBN 0201855046 [zitiert auf S. 184]

[Jal98] JALOTE, Pankaj: *Fault tolerance in distributed systems.* Prentice-Hall, 1998. – ISBN 0–13–301367–7 [zitiert auf S. 35, 41, 49, 99]

[Jek99] JEKIMOV, Michail: *Weiterentwicklung des inkrementellen und verteilten Binders FLink.* Systementwicklungsprojekt, Technische Universität München, 1999 [zitiert auf S. 32]

[KAYE04] Kremenek, Ted ; Ashcraft, Ken ; Yang, Junfeng ; Engler, Dawson: Correlation exploitation in error ranking. In: *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*. New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–855–5, S. 83–93 [zitiert auf S. 192]

[KGV83] Kirkpatrick, S. ; Gelatt, Jr. C. D. C. D. ; Vecchi, M. P.: Optimization by Simulated Annealing. In: *Science* 220 (1983), Nr. 4598, S. 671–680 [zitiert auf S. 184]

[KHB99] Karaorman, Murat ; Holzle, Urs ; Bruno, John: jContractor: A Reflective Java Library to Support Design by Contract. Santa Barbara, CA, USA : University of California at Santa Barbara, 1999. – Forschungsbericht [zitiert auf S. 82]

[KK07] Koren, Israel ; Krishna, C.Mani ; Simpson, Dawnmarie (Hrsg.): *Fault-Tolerant Systems*. San Franciso, CA : Morgan-Kaufman Publishers, 2007 [zitiert auf S. 1, 2, 35, 99, 106, 139, 145, 152, 153, 154, 156, 205]

[KL86] Knight, J. C. ; Leveson, N. G.: An experimental evaluation of the assumption of independence in multiversion programming. In: *IEEE Trans. Softw. Eng.* 12 (1986), Nr. 1, S. 96–109. – ISSN 0098–5589 [zitiert auf S. 154]

[Kra98] Kramer, R.: iContract – the Java Design by Contract Tool. In: *Technology of Object-Oriented Languages, TOOLS 26*, IEEE Press, August 1998. – ISBN 0–8186–8482–8, S. 295–307 [zitiert auf S. 82]

[KT86] Koo, Richard ; Toueg, Sam: Checkpointing and Rollback-Recovery for Distributed Systems. In: *Proceedings of ACM Fall Joint Computer Conference*, IEEE Computer Society Press, 1986, S. 1150–1158 [zitiert auf S. 103]

[LAK92] Laprie, J.C. C. (Hrsg.) ; Avizienis, A. (Hrsg.) ; Kopetz, H. (Hrsg.): *Dependability: Basic Concepts and Terminology*. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 1992. – ISBN 0387822968 [zitiert auf S. 36]

[Lam78] Lamport, Leslie: Time, clocks, and the ordering of events in a distributed system. In: *Communications of the ACM* 21 (1978), Nr. 7, S. 558–565 [zitiert auf S. 55, 58]

[Lam79] Lamport, L.: How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. In: *IEEE Trans. Computers* 28 (1979), Nr. 9, S. 690–691 [zitiert auf S. 60]

[Lar61] Lardner, D.: *Charles Babbage and His Calculating Engines*. 1961. – Reprint of the original Version "Babbage's Calculating Engines", in Edinburgh Review, July 1834 [zitiert auf S. 153]

[LKP02] Lackner, M. ; Krall, A. ; Puntigam, F.: *Supporting design by contract in Java*. 2002 [zitiert auf S. 82]

[LP06] Landes, Tobias ; Preissinger, Jörg: Realizing Consistent Event Ordering in Distributed Shared Memory Systems. In: Arabnia, Hamid R. (Hrsg.): *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*. Las Vegas, NV : CSREA Press, Juni 2006, S. 10–16 [zitiert auf S. 69]

[Man00] Man Machine Systems (Hrsg.): *Design by Contract for Java Using JMSAssert*. http://www.mmsindia.com/DBCForJava.html: Man Machine Systems, 2000 [zitiert auf S. 82]

[May06] Mayer, Alexander: *Integration von Design by Contract in das sprachbasierte, verteilte System MoDiS*, Technische Universität München, Diplomarbeit, August 2006 [zitiert auf S. 85, 92, 227]

[Mey92] Meyer, Bertrand: Design by Contract. In: Meyer, B. (Hrsg.) ; Mandrioli, D. (Hrsg.): *Advances in Object-Oriented Software Engineering*, Prentice–Hall, 1992 [zitiert auf S. 79]

[NX95] Netzer, Robert H. B. ; Xu, Jian: Necessary and Sufficient Conditions for Consistent Global Snapshots. In: *IEEE Trans. on Parallel Distrib. Syst.* 6 (1995), Nr. 2, S. 165–169. – ISSN 1045–9219 [zitiert auf S. 104]

[OH96] Offutt, A. J. ; Hayes, J. H.: A semantic model of program faults. In: *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA : ACM Press, 1996. – ISBN 0–89791–787–1, S. 195–200 [zitiert auf S. 192]

[Pfl07] Pflüger, Mark: *Erstellung und Wiederherstellung konsistenter Systemzustände in verteilten parallelen Rechensystemen*. Sysstementwicklungsprojekt Technische Universität München, 2007 [zitiert auf S. 139]

[PH93] Poetzsch-Heffter, A.: Programming Language Specification and Prototyping Using the MAX System. In: Bruynooghe, M. (Hrsg.) ; Penjam, J. (Hrsg.): *Programming Language Implementation and Logic Programming*, Springer-Verlag, 1993 (LNCS 714), S. 137–150 [zitiert auf S. 30]

[Piz97] Pizka, Markus: Design and Implementation of the GNU INSEL Compiler (GIC). 1997 (TUM–I9713, SFB–Bericht 342/09/97 A). – Forschungsbericht [zitiert auf S. 30]

[PL05] Preissinger, Jörg ; Landes, Tobias: Fundamentals for Consistent Event Ordering in Distributed Shared Memory Systems. In: Arabnia, Hamid R. (Hrsg.): *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'05*. Las Vegas, NV : CSREA Press, Juni 2005, S. 890–896 [zitiert auf S. 55, 61, 64]

[Plö97] Plösch, Reinhold: Design by Contract for Python. In: *Fourth Asia-Pacific Software Engineering and International Computer Science Conference*. Washington, DC : IEEE Computer Society, 1997. – ISBN 0–8186–8271–X, S. 213 [zitiert auf S. 82]

[Plö02] Plösch, R.: Evaluation of Assertion Support for the Java Programming Language. In: *Journal OF Object Technology* 1 (2002), Nr. 3, S. 5–17. – special issue: TOOLS USA [zitiert auf S. 82]

[PM07] Preissinger, Jörg ; Mayer, Alexander: Integrating Design by Contract in INSEL. In: *Proceedings of the International MultiConference of Engineers and Computer Scientists, IMECS'2007*. Hong Kong, März 2007, S. 1037–1043 [zitiert auf S. 82, 85]

[PP99] Plösch, Reinhold ; Pichler, Josef: Contracts: From Analysis to C++ Implementation. In: *TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems*. Washington, DC, USA : IEEE Computer Society, 1999. – ISBN 0–7695–0278–4, S. 248 [zitiert auf S. 82]

[PP05] Purushothaman, R. ; Perry, D.E.: Toward understanding the rhetoric of small source code changes. In: *Software Engineering, IEEE Transactions on* 31 (2005), June, Nr. 6, S. 511–526 [zitiert auf S. 152]

[PP07] Preissinger, Jörg ; Pflüger, Mark: Compiler Supported Interval Optimisation for Communication Induced Checkpointing.

In: Arabnia, Hamid R. (Hrsg.): *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'07* Bd. II. Las Vegas, NV : CSREA Press, Juni 2007, S. 550–556 [zitiert auf S. 107, 136]

[Ray02] Raynal, Michel: Sequential consistency as lazy linearizability. In: *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA : ACM Press, 2002. – ISBN 1–58113–529–7, S. 151–152 [zitiert auf S. 60]

[Reh98] Rehn, Christian: *Inkrementelles und dynamisches Binden in einer verteilten Umgebung*. Institut für Informatik, Diplomarbeit, Februar 1998 [zitiert auf S. 32]

[Reh03] Rehn, Christian: Dynamic Global Scheduling in Cooperative Distributed Systems. In: Arabnia, Hamid R. (Hrsg.): *PDPTA*. Las Vegas, NV, Juni 2003. – ISBN 1–892512–43–2, S. 1427–1433 [zitiert auf S. 160, 168]

[Reh04] Rehn, Christian: *Überwindung der Stellengrenzen in kooperativen verteilten Systemen*, Technische Universität München, Diss., März 2004 [zitiert auf S. 12, 30, 33, 129, 160, 168]

[Reh06] Rehn, Christian: Dynamic mapping of cooperating tasks to nodes in a distributed system. In: *Future Generation Comput. Syst.* 22 (2006), Nr. 1, S. 35–45 [zitiert auf S. 33, 160, 168]

[RP06] Rechenberg, P. (Hrsg.) ; Pomberger, G. (Hrsg.): *Informatik Handbuch*. 4. München, Germany : Hanser, München, Germany, 2006 [zitiert auf S. -]

[RW96] Radermacher, R. ; Weimer, F.: INSEL Syntax–Bericht / TU München. 1996. – Forschungsbericht. – SFB–Bericht 342/08/96 A TUM–I9617 [zitiert auf S. 4, 29]

[SC91] Sullivan, M. ; Chillarege, R.: Software defects and their impact on system availability-a study of field failures in operating systems. In: *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, 1991, S. 2–9 [zitiert auf S. 150, 152, 192, 194]

[Sch35] Schrödinger, E.: Die gegenwärtige Situation in der Quantenmechanik. In: *Naturwissenschaften* 23 (1935), Nr. 50, S. 844–849 [zitiert auf S. 77]

[SEL+96] SPIES, P.P. ; ECKERT, C. ; LANGE, M. ; MAREK, D. ; RADERMACHER, R. ; WEIMER, F. ; WINDISCH, H.-M.: Sprachkonzepte zur Konstruktion verteilter Systeme. 1996 (TUM-I9618, SFB 342/09/96 A). – Forschungsbericht [zitiert auf S. 3, 12, 22, 91]

[Sil97] SILVA, L.M.: *Checkpointing Mechanisms for Scientific Parallel Applications*, Univ. of Coimbra, Portugal, Diss., January 1997 [zitiert auf S. 102]

[SP02] STALLMAN, R.M. ; PESCH, R.H.: *Debugging with GDB: The GNU Source-level Debugger*. Free Software Foundation, 2002 [zitiert auf S. 71]

[Sta02] STALLINGS, William ; KRUMM, Heiko (Hrsg.) ; KEMPER, Peter (Hrsg.): *Betriebssysteme*. 4. Prentice Hall, 2002 [zitiert auf S. 43]

[Tan01] TANENBAUM, Andrew S.: *Modern Operating Systems*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 2001. – ISBN 0130313580 [zitiert auf S. 195]

[TS84] TAMIR, Y. ; SEQUIN, C.H.: Error recovery in multicomputers using global checkpoints, 1984, S. 32–41 [zitiert auf S. 101]

[TS03] TANENBAUM, Andrew S. ; STEEN, Maarten van ; MUHR, Judith (Hrsg.): *Verteilte Systeme*. München : Pearson Studium, 2003 (Pearson StudiumInformatik, Verteilte Systeme). – ISBN 3827370574 [zitiert auf S. 10]

[VMM91] VOAS, Jeffrey ; MORREL, Larry ; MILLER, Keith: Predicting Where Faults Can Hide from Testing. In: *IEEE Softw.* 8 (1991), Nr. 2, S. 41–48. – ISSN 0740–7459 [zitiert auf S. 192]

[Wag06] WAGNER, Stefan: A literature survey of the quality economics of defect-detection techniques. In: *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering*. New York, NY, USA : ACM Press, 2006. – ISBN 1–59593–218–6, S. 194–203 [zitiert auf S. 150, 151, 191]

[Weg97] WEGNER, Peter: Why interaction is more powerful than algorithms. In: *Commun. ACM* 40 (1997), Nr. 5, S. 80–91. – ISSN 0001–0782 [zitiert auf S. 152]

[WHFK97] WANG, Yi-Min ; HUANG, Yennun ; FUCHS, W. K. ; KINTALA, Chandra: Progressive Retry for Software Failure Recovery

in Message-Passing Applications. In: *IEEE Trans. Comput.* 46 (1997), Nr. 10, S. 1137–1141. – ISSN 0018–9340 [zitiert auf S. 208]

[WJKT05] Wagner, S. ; Jurjens, J. ; Koller, C. ; Trischberger, P.: Comparing Bug Finding Tools with Reviews and Tests. In: *Proc. 17th International Conference on Testing of Communicating Systems (TestCom'05)* Bd. 3502 of LNCS, Springer, 2005, S. 40–55 [zitiert auf S. 192]

[XKI03] Xu, Jun ; Kalbarczyk, Z. ; Iyer, R.K.: Transparent runtime randomization for security. In: *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*, 2003, S. 260–269 [zitiert auf S. 155, 193]