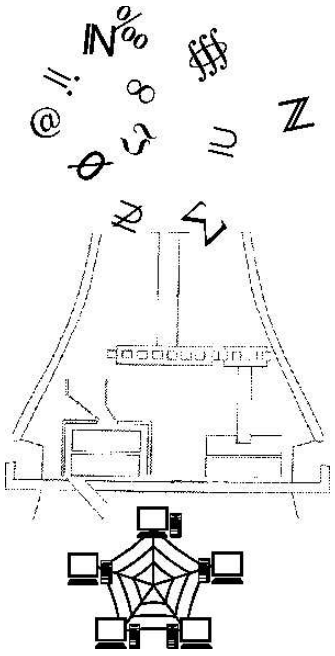
TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Dissertation

# Ein generativer Ansatz zur Konstruktion des Managements verteilter kooperierender Systeme aus abstrakten Spezifikationen

Thomas M. Peschel-Findeisen



# Institut für Informatik
# der Technischen Universität München


# Ein generativer Ansatz zur Konstruktion des Managements verteilter kooperierender Systeme aus abstrakten Spezifikationen


**Thomas M. Peschel-Findeisen**



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. Dr. h.c. Jürgen Eickel

Die Dissertation wurde am 26.06.2003 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 13.02.2004 angenommen.

*Das ist ein weites Feld*

Theodor Fontane – Effi Briest

# Zusammenfassung

Der Entwurf und die Erstellung von Betriebssystemen sind komplexe Aufgabenstellungen, da die Abhängigkeiten zwischen den auszuführenden Systemen und dem Betriebssystem und andererseits den Teilen des Betriebssystems sehr eng und im Allgemeinen nur implizit gegeben sind. Diese Problemstellung wird bei der Betrachtung verteilter Systeme weiter verschärft. Als problematisch erweist sich dabei insbesondere das mangelnde Wissen des Betriebssystems über die Anwendungen. Daher wird das Betriebssystem zum Management erweitert, welches speziell die Anforderungen einer Anwendung erfüllt.

Diese Überlegung führt zunächst zur engeren Koppelung zwischen dem auszuführenden System und dessen Management. Mit der Einbringung von Wissen über die Anwendungen in das Management ergibt sich die Möglichkeit, die Anforderungen des Systems an das Management – auch in der Zukunft – besser abschätzen zu können. Das Management wird somit automatisch für eine Anwendung angepasst. Der Anwendungsbereich von verteilten Systemen umfasst sehr unterschiedliche Aufgabenstellungen. Dies manifestiert sich z. B. in der Existenz unterschiedlicher Programmiersprachen für verteilte Systeme. Für jede Klasse von Aufgabenstellungen wird daher ein anderes Management benötigt.

Um die Wiederverwendbarkeit und Anpassungsfähigkeit von Management-Systemen zu erhöhen, ist es notwendig, das Management auf hohem Abstraktionsniveau zu spezifizieren. In dieser Spezifikation muss einerseits die Sprache, die zur Erstellung der Anwendungen verwendet wird, anderseits die Abbildung der Sprachkonstrukte auf die Hardware und somit eine Beschreibung der Ressourcen selbst enthalten sein. Für die Abbildung der Konstrukte der Programmiersprache auf die vorhandenen Ressourcen müssen die Eigenschaften dieser Komponenten eines Systems erfasst und geeignet modelliert werden. Von zentraler Bedeutung sind dabei die Strukturen über den Komponenten. Sowohl die Eigenschaften als auch die Strukturen sind in der Zeit variabel.

Diese Spezifikationen werden durch einen generativen Prozess in ein ausführbares Management transformiert. Durch diesen generativen Vorgang wird die Umsetzung einer Spezifikation als Fehlerquelle bei der Erstellung des Managements ausgeschlossen.

# Danksagung

An der Entstehung einer Dissertation sind direkt oder auch indirekt eine Reihe von Menschen beteiligt, ohne deren Unterstützung eine solche Arbeit nicht möglich wäre.

Zunächst gilt mein Dank Herrn Prof. Dr. P. P. Spies, an dessen Lehrstuhl für Systemarchitektur und Betriebssysteme ich während der Entstehung dieser Arbeit beschäftigt war und dessen persönliche und fachliche Unterstützung zum Gelingen dieser Arbeit wesentlich beigetragen hat.

Den Kollegen des Lehrstuhls, allen voran Betram Hütter, danke ich für zahlreiche fachliche Diskussionen und Anregungen zu dieser Arbeit, aber auch für das kollegiale Klima, in dem diese Arbeit entstanden ist.

Desweiteren haben eine Reihe studentischer Hilfskräfte, Praktikanten und Diplomanden zur Entstehung dieser Dissertation beigetragen. Beispielhaft für all die Anderen sei hier Jörg Preißinger namentlich genannt, der mich und die Entstehung dieser Arbeit als studentische Hilfskraft fast vier Jahre begleitet hat.

Meinen Eltern danke ich für ihre Unterstützung, insbesondere während meiner Studienzeit, in der sie mir finanzielle Unabhängigkeit ermöglicht haben. Ohne diese Grundlage wäre auch diese Arbeit nicht entstanden.

Mein größter Dank allerdings gebührt meiner Frau, Eva Peschel, die mich in den vergangenen Jahren, auch in schwierigen Phasen, unterstützt, angetrieben und aufgemuntert hat. Ohne sie wäre dies alles nicht möglich gewesen. Ein besonderer Dank gilt ihr darüberhinaus für das Korrekturlesen dieser Arbeit.

München/Garching, im Juni 2003

*Thomas Peschel-Findeisen*

# Inhaltsverzeichnis

# 1 Einleitung

Die Verbreitung verteilter Systeme hat in den letzten Jahren ständig zugenommen. Für diese Entwicklung lassen sich eine Reihe von Ursachen festzustellen. Einerseits steigt die Zahl der Rechner an sich, da die Hardware immer leistungsfähiger, aber auch kostengünstiger wird. Selbst in Bereichen, in denen man zunächst keinen Einsatz von Rechnern erwarten würde, begegnen uns heute Computer. Dies führt zur Notwendigkeit des Datenaustausches zwischen diesen Rechnern.

Die Verbreitung von Rechensystemen ist jedoch auch für deren zunehmender Bedeutung als Kommunikationsmedium ursächlich. Die Verwendung von Diensten wie E-Mail, Newsgroups und dem WWW sind heute in der westlichen Gesellschaft so selbstverständlich wie das Telefon oder noch vor 30 Jahren das Schreiben von Briefen. Aber nicht nur zur privaten und geschäftlichen Kommunikation werden diese Dienste heute eingesetzt, sondern auch zum Informationsaustausch in Büros oder anderen Gebieten der engen Zusammenarbeit. Vielfach ist in einem modernen Arbeitsumfeld der Arbeitsablauf beispielsweise auf eine gemeinsame Datenhaltung und Datennutzung ausgerichtet, so dass Dienste wie verteilte Dateisysteme nicht mehr wegzudenken sind. Diese Rechnerkonfigurationen und Dienste werden inzwischen durch mobile Geräte, wie zum Beispiel Laptops, Organizer oder Mobiltelefone, und spezielle Dienste für diese ergänzt. Diese Geräte werden dann ad hoc in ein bestehendes verteiltes System eingebunden, partizipieren an diesem und werden wieder aus dem System entfernt.

Auf der anderen Seite nimmt die Komplexität der Aufgabenstellungen fortlaufend zu. Mathematische Probleme, Simulationen oder die Verarbeitung großer Datenmengen bei physikalischen Experimenten sowie in der Medizin sind nur unter Einsatz großer, leistungsfähiger Rechner möglich. Aufgrund der Kosten dieser Systeme, werden auch in diesen Bereichen vielmals verteilte Systeme eingesetzt, welche kostengünstiger und flexibler sind als Großrechenanlagen, insbesondere da oftmals eine Anbindung an ein bestehendes System der oben beschriebenen Art wünschenswert oder zwingend erforderlich ist.

Gemäß dieser unterschiedlichsten Anforderungen – die Liste ließe sich noch beliebig fortsetzen – existieren auch unterschiedliche Definitionen für verteilte Systeme. Zwei Beispiele für derartige Definitionen sind:

**Definition 1.1 (Verteiltes System (nach A. Tanenbaum))**
*Ein verteiltes System ist eine Ansammlung unabhängiger Rechner, die den Benutzern als ein Einzelrechner erscheinen.*

**Definition 1.2 (verteiltes System (nach L. Lamport))**
*Ein verteiltes System ist ein System, mit dem ich nicht arbeiten kann, weil irgendein Rechner abgestürzt ist, von dem ich nicht einmal weiß, dass es ihn überhaupt gibt.*

Die erste Definition von A. Tanenbaum beschreibt die Sichtweise der Benutzer auf ein verteiltes System. Diese Definition entspricht den oben beschriebenen Anforderungen und Aufgabenstellungen an verteilte Systeme. In der zweiten Definition nach L. Lamport wird in erster Linie deutlich, dass verteilte Systeme, so wie sie heutzutage existieren, Probleme aufwerfen und keineswegs als stabile funktionierende System angesehen werden können.

Der Schritt von Einzelrechnern hin zu verteilten Systemen verschärft die Problemstellungen, die auch bei ersteren existieren. Trotz dieses Faktums haben verteilte Systeme sowohl gegenüber Großrechnern, als auch gegenüber unabhängigen Einzelrechnern unbestreitbare Vorteile, wie Wirtschaftlichkeit, Geschwindigkeit, Skalierbarkeit, Flexibilität, gemeinsame Nutzung von Ressourcen.

Ein zentraler Punkt für verteilte Systeme ist die Unterstützung des Systems durch das Betriebssystem. Die klassische Aufgabenstellung von Betriebssystemen ist einerseits die Veredelung der Hardware zu Komponenten, welche für Anwendungen nutzbar sind. Andererseits besteht die Aufgabe von Betriebssystemen in der Steuerung, Planung und der Kontrolle der Berechnungen, die das Rechensystem ausführt. Nach DIN ist ein Betriebssystem folgendermaßen definiert:

**Definition 1.3 (Betriebssystem (DIN 44300))**
*Die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Grundlage der möglichen Betriebsarten des digitalen Rechensystems bilden und insbesondere die Abwicklung von Programmen steuern und überwachen.*

Für verteilte Systeme existiert eine Reihe von Möglichkeiten zur Realisierung des Betriebssystems, bzw. zur Aufteilung der Aufgabenstellung zwischen Anwendungen und Betriebssystem.

1. **Verteilte Betriebsumgebung über lokalem Betriebssystem**
   Jeder Rechner in einem verteilten System verfügt über sein eigenes privates Betriebssystem, die Aspekte der Verteilung werden vollständig durch die Anwendungen bzw. eine Zwischenschicht erbracht.

2. **Netzwerkbetriebssystem**
   Das individuelle Betriebssystem eines jeden Rechners unterstützt das verteilte System, beispielsweise durch Funktionalitäten zur Kommunikation. Die Mechanismen zur Verteilung sind aber vollständig in den Anwendungen enthalten.

3. **Verteiltes Betriebssystem**
   Ein verteiltes Betriebssystem realisiert sowohl die Kommunikation, als auch die Verteilungsmechanismen vollständig. Die Verteilung ist für den Benutzer des Systems transparent.

So unterschiedlich die Anforderungen und Nutzungsprofile von verteilten Anwendungen sind, so unterschiedlich sind auch die Lösungen der jeweiligen Aufgabenstellungen. Die Anforderungen an ein verteiltes System in einem großen Büro unterscheiden sich vollständig von den Anforderungen an ein verteiltes System zur Berechnung von Crash-Test-Simulationen. Daher ist für ein verteiltes System keine einheitliche Lösung und somit auch kein ideales Betriebssystem für alle Arten von verteilten Systemen möglich. Daher sind für verschiedene Aufgabenstellungen jeweils individuelle Betriebsysteme notwendig.

## 1.1 Entwurf komplexer Systeme

Der Entwurf von Betriebsystemen gilt als eine der komplexesten Aufgaben in der Informatik. Dies ist nicht allein in der Größe der Betriebsysteme begründet. Zwar sind dem Codeumfang nach Betriebssysteme mit die größten Systeme, die auf Computern ausgeführt werden – beispielsweise bestehen aktuelle UNIX Implementierungen aus mehr als 1 Million Codezeilen (vgl. [Tan01]), Windows 2000 aus 29 Million Zeilen – was jedoch nicht allein die hohe Komplexität der Betriebssysteme bedingt.

Die Komplexität von Betriebsystemen liegt insbesondere in der engen Verzahnung der einzelnen Komponenten begründet. In [Tan01] findet sich folgender Vergleich:

> *Betriebssysteme sind nicht die komplexesten Systeme. Flugzeugträger beispielsweise sind weitaus komplexer, aber sie lassen sich besser in*

> *Subsysteme unterteilen. Die Konstrukteure der Toiletten auf einem Flugzeugträger müssen sich keine Gedanken über das Radar-System machen. Die Subsysteme interagieren kaum.*

Dieser Vergleich macht das Hauptproblem des Entwurfs von Betriebsystemen deutlich. In Betriebssystemen sind die Abhängigkeiten zwischen den einzelnen Teilsystemen sehr eng. Daher wirken sich Veränderungen an einer Stelle an den verschiedensten anderen Stellen aus.

Es finden sich in der Literatur nur wenige Ideen für Lösungen bezüglich der Komplexität bei der Entwicklung von Betriebsystemen. Drei Artikel zu diesem Themenkomplex sind [Lam83], [SRC84] und [Cor91]. Diese Artikel diskutieren insbesondere den Entwurf von Betriebsystemen und die Fehler in derartigen Entwürfen. Als Hauptursache für Entwurfs- und Implementierungsfehler wird auch in diesen jeweils die Komplexität der Systeme aufgezeigt.

Die Lösungsansätze, die in diesen Artikeln und auch in [Tan01] angerissen werden, sind einerseits auf die Organisation des Entwicklungsprozesses und andererseits auf die Reihenfolge des Entwurfs und der Implementierung der Systemteile gerichtet. Diese Hinweise sind sicherlich für den Entwurf großer, komplexer Systeme, z. B. Betriebssysteme, nützlich und hilfreich. Allerdings stellen sie keine automatisierte Hilfestellung bereit. Der Erfolg einer Verwendung dieser Methoden ist stark von der Einhaltung dieser Richtlinien abhängig. Insbesondere ist die Anwendung dieser Vorschläge im Hinblick auf die spätere Erweiterung von Betriebsystemen schwierig oder nicht möglich, da bestehende Systemteile erhalten bleiben sollen oder müssen.

Weitere Ursachen für die Komplexität der Betriebssysteme liegen in der Nebenläufigkeit der Systeme. Obwohl der Mensch selbst ein hochgradig nebenläufiges System darstellt, hat er doch Schwierigkeiten mit dieser korrekt umzugehen.

Die Entwickler eines klassischen Betriebsystems wissen im Allgemeinen nicht, wozu ihr System später eingesetzt werden wird, bzw. welche Anwendungen darauf ausgeführt werden. Die Unterstützung der vielfachen potenziellen Anforderungen erschwert den Entwurf der Systeme, insbesondere da das Betriebssystem nicht einmal zur späteren Ausführungszeit der Anwendungen ein umfassendes Wissen über diese besitzt.

Der letzte Punkt, der als Ursache für die Komplexität von Betriebsystemen auszumachen ist, liegt in der Portabilität der Systeme. Vielfach ist es wünschenswert, ein System auf unterschiedlicher Hardware ausführen zu können. Da die Restriktionen und Notwendigkeiten der Hardware aber sehr unterschiedlich sind, ist diese Aufgabe nicht leicht und erfordert oftmals die Anwendung von „Tricks“.

## 1.2 Verteilte Systeme

Die Komplexität des Entwurfs von Betriebssystemen für einzelne Rechner wird durch den Aspekt der Verteilung noch verstärkt. Bei der Konstruktion verteilter Systeme bzw. eines Betriebssystems für verteilte Systeme treten neue Anforderungen auf, wie zum Beispiel:

1. **Transparenz**
   Von einem verteilten System wird Transparenz in verschiedener Hinsicht erwartet, d. h. die Aspekte der Verteiltheit sind für den Benutzer des Systems nicht oder nur teilweise sichtbar. Zentrale Aspekte der Transparenz sind:

   - Zugriffstransparenz
     Lokale und entfernte Zugriffe werden identisch aufgerufen.
   - Ortstransparenz
     Der physische Ort von Ressourcen spielt für den Zugriff keine Rolle. Für die Identifikation werden abstrakte Namen verwendet.
   - Migrationstransparenz
     Werden Ressourcen im System verlagert, so verändert sich weder der Zugriff noch der Name der Ressourcen.
   - Replikationstransparenz
     Ressourcen können im System ohne Wissen des Benutzers repliziert werden.
   - Fehlertransparenz
     Ein Fehler einer Teilkomponente des verteilten Systems führt nicht zum Ausfall des gesamten Systems. Die Daten des verteilten Systems bleiben konsistent.

   Beim Entwurf eines verteilten Systems sollten diese Aspekte der Transparenz berücksichtigt werden. Das bedingt, dass auf einer niedrigen Abstraktionsebene des Systems natürlich keine Transparenz vorhanden ist.

2. **Skalierbarkeit**
   Unter Skalierbarkeit wird im Allgemeinen verstanden, ob und inwieweit ein System unabhängig von der Anzahl der Knoten ist. Dabei spielt insbesondere die dynamische Erweiterbarkeit eine große Rolle. Die Skalierung sollte dabei transparent sein, d. h. ohne Veränderung der Software möglich sein.

3. **Offenheit**
   Verteilte Systeme umfassen meist heterogene Komponenten, sowohl im Bereich der Hard- als auch der Software. Die Zusammenarbeit dieser unterschiedlichen Komponenten bedingt offene Schnittstellen.

4. **Flexibilität**
   Systeme müssen flexibel um neue Komponenten oder Dienste erweitert werden können. Diese Erweiterbarkeit ist eine zentrale Voraussetzung, um auch zukünftige Entwicklungen in bestehende Systeme einbinden zu können. Insbesondere für den Entwurf von Betriebssystemen ist dieser Punkt wesentlich.

Neben diesen gerade beschriebenen Anforderungen an verteilte Systeme sind auch weitere Aspekte von Interesse, wie z. B. Zuverlässigkeit und Effizienz. Während die Erfüllung dieser Erfordernisse bereits bei verteilten Anwendungen problematisch ist, zeigen sich bei Betriebssystemen, welche als Basis für die Ausführung verteilter Systeme dienen, aufgrund der starken Abhängigkeiten zwischen den einzelnen Komponenten noch größere Schwierigkeiten.

Für verteilte Systeme existieren eine Reihe von Programmiermodellen. Die wesentlichen sollen hier kurz genannt werden.

1. **Systeme mit Nachrichtenaustausch**
   Systeme mit Nachrichtenaustausch basieren auf den Primitiven *Senden* und *Empfangen*. Mittels dieser Primitiven können verteilte Systeme aufgebaut werden, wobei bei diesen Systemen im Allgemeinen keinerlei Transparenz gegeben ist. Allerdings unterstützen verschiedene Bibliotheken zum Nachrichtenaustausch die Verwendung heterogener Umgebungen. Zwei bekannte Vertreter dieses Programmiermodells sind PVM (siehe [GBD+94]) und MPI (siehe [SOHL+96]).

2. **Entfernter Prozeduraufruf**
   Eines der ersten Client-Server-Programmiermodelle war der entfernte Prozedur-Aufruf (RPC – Remote Procedure Call). Dieses Modell erweitert das Konzept des lokalen Prozeduraufrufs auf den verteilten Fall. Dadurch wird für den Aufrufer Transparenz erreicht, da ein entfernter Prozeduraufruf sich nicht (oder nur unwesentlich) vom lokalen Fall unterscheidet. Eine erste Realisierung dieses Konzeptes wurde 1984 vorgestellt (vgl. [BN84]).

3. **Objektorientierte Systeme**
   Mit dem Vormarsch objektorientierter Ansätze in Entwurf und Programmierung wurden auch objektorientierte Ansätze zur verteilten Programmierung entwickelt. Der bekannteste Ansatz in diesem Bereich ist CORBA (siehe [OMG01b]). Dabei werden die Objekte als Einheiten der Verteilung betrachtet und die Methoden der Objekte analog zum Konzept der RPCs aufgerufen.

4. **Spezialsysteme**
   Neben diesen allgemeinen Programmiermodellen existieren auch noch eine Reihe weiterer Modelle für spezielle Anwendungen, wie zum Beispiel verteilte Transaktionssysteme (siehe [RDD99]) im Bereich der Datenbanken.

Auch verteilte gemeinsame Speicher sind ein Modell, welches für die Realisierung verteilter Systeme (DSM – Distributed Shared Memory) herangezogen werden kann. Für dieses Konzept existiert eine Reihe von Ansätzen, welche sich in der Granularität der Verteilungseinheiten unterscheiden. Auf der einen Seite stehen die seitenbasierten DSM-Systeme, wie z. B. TreadMarks (siehe [KDCZ94], auf der anderen Seite objektbasierte Ansätze, wie Orca (siehe [BKT92] und [BBH+98]).

Bereits dieser kurze Überblick verdeutlicht die Vielzahl der unterschiedlichen Konzepte für die Programmierung verteilter Systeme. Jedem dieser Konzepte sind Vor- und Nachteile inhärent, welche die einzelnen Modelle für unterschiedliche Anwendungsfälle mehr oder weniger geeignet erscheinen lassen. Daher ist für die Realisierung eines verteilten Systems auch unter diesem Blickwinkel eine uniforme Realisierung nicht möglich. Selbst mit einem System, welches alle bekannten Modelle unterstützt, können nicht alle Anwendungsfälle geeignet abgedeckt werden, da für Spezialanwendungen stets auch neuartige spezialisierte Verteilungsmodelle entwickelt werden.

Diese Erkenntnis der Unmöglichkeit einer uniformen Lösung für verteilte Systeme führt zu der Überlegung, die Systeme und insbesondere deren Laufzeitumgebung aus abstrakten Modellen automatisiert zu erzeugen.

## 1.3 Generierung von Systemen

Ziel eines generativen Ansatzes ist die automatische Erzeugung von lauffähigen Komponenten oder Systemen aus einer abstrakten Beschreibung. Dazu ist es notwendig, die möglichen Bestandteile einer Spezifikation zu beschreiben, sowie deren Abbildung auf lauffähige (Teil-)Systeme. Ein *Generator* ist daher eine Abbildung bzw. ein Werkzeug zur Realisierung einer Abbildung aus der abstrakten Beschreibung (Problemstellung) in eine ausführbare Lösung. Dies ist in Abb. 1.1 auf der nächsten Seite dargestellt. Der Generator muss dabei mehrere Aspekte realisieren, wie die Verhinderung illegaler Kombinationen, die Berücksichtigung von Voreinstellungen, Konstruktionsregeln sowie die Durchführung von Optimierungen.

Dieser generative Vorgang ist dann realisierbar, wenn die Beschreibung der Problemstellung hinreichend vollständig ist. Eine unvollständige Spezifikation kann auch nur zu einer unvollständigen Realisierung führen. Der Generator kann allerdings eine Reihe von impliziten Annahmen enthalten, welche gewisse Unvollständigkeiten ausgleichen können. Diese Voreinstellungen müssen allerdings dem Anwender des Generators bewusst sein bzw. deren Verwendung sollte vom Generator dokumentiert werden. Diese Technik findet sich beispielsweise auch in klassi-

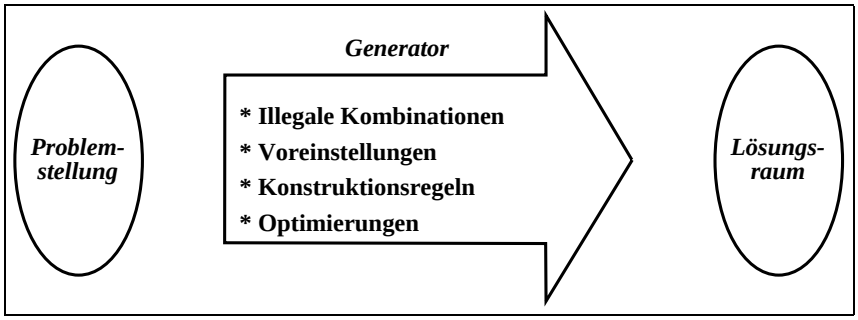



Abbildung 1.1: Das generative Modell.

schen Übersetzern, welche bei fehlenden Angaben im Quellcode eines Programms implizite Annahmen machen und deren Anwendung als Warnung ausgeben.

Ein zentraler Gesichtspunkt bei der Realisierung von Generatoren ist die korrekte Abbildung der Problembeschreibung in den Lösungsraum. Eine Eigenschaft, welche in der Beschreibung der Problemstellung enthalten ist, muss auch im Lösungsraum entsprechend gültig bleiben. Diese Korrektheit des Generators muss nachgewiesen werden. Im Allgemeinen wäre für diesen Nachweis eine formale Verifikation des Generators erforderlich. Dies ist allerdings im Allgemeinen eine komplexe Aufgabe, welche mit den heute bekannten Verifikationstechniken zwar möglich wäre, aber einen erheblichen Aufwand bedeuten würde. So beschränkt man den Nachweis der Korrektheit der Abbildung aus der Problembeschreibung in den Lösungsraum auf informelle, aber schlüssige Begründungen.

Dabei ist zu beachten, dass der Nachweis der korrekten Transformation nicht nur auf einzelne Teile der Problembeschreibung beschränkt werden darf, sondern die Korrektheit der vollständigen Abbildung gewährleistet werden muss.

Die Vorteile dieses generativen Ansatzes zur Realisierung von Systemen liegt in der Konzentration auf die Beschreibung der Problemstellung auf einem hohen Abstraktionsniveau. Mit der Verwendung eines Generators zur Erzeugung einer Implementierung geht eine Verlagerung des Schwerpunktes einer Systementwicklung von den Aspekten bzw. Restriktionen der Implementierung hin zu den Konzepten und Strukturen eines Systems einher. Dieser Schritt ist vergleichbar mit dem Schritt von der Programmierung in Maschinensprache hin zur Verwendung von Hochsprachen wie C oder Pascal.

Generatoren sind durchaus mit Übersetzern für Programmiersprachen vergleichbar. Auch in der Entwicklung von Programmiersprachen wurde das Abstraktionsniveau immer weiter von der realen Maschine entfernt. Während imperative Sprachen sich stark an der Architektur nach von Neumann orientieren, liegt das Abstraktionsniveau bei funktionalen, logischen oder objektorientierten Sprachen

weitaus höher, ist aber, zumindest an verschiedenen Stellen, immer noch an Restriktionen bezüglich der Realisierung gebunden. Ein klassisches Beispiel für diese Tatsache ist, dass in Java (vgl. [GJG96]) ein Programm nur unter Verwendung von Ausnahmen in der Objektorientiertheit der Sprache gestartet werden kann (siehe Abb. 1.2).

```
class main {
  static void main (String args[]) {
  System.out.println("Hello world");
  }
}
```

Abbildung 1.2: `Hello World` in der Sprache Java.

Das Abstraktionsniveau einer Problembeschreibung als Eingabe für einen Generator liegt oberhalb des Niveaus von Programmiersprachen. Maschinenabhängige Aspekte der Realisierung dürfen in der Problembeschreibung keine Rolle spielen. Daher werden diese Problembeschreibungen als Spezifikationen bezeichnet.

## 1.4 Ziele und Aufbau dieser Arbeit

In dieser Arbeit soll nun ein Ansatz erarbeitet werden, der es ermöglicht, das Management verteilter Systeme abstrakt zu beschreiben und aus dieser abstrakten Beschreibung eine Laufzeitumgebung für verteilte Systeme zu generieren. Eine abstrakte Spezifikation ermöglicht es, die Strukturen und Abhängigkeiten eines Systems formal zu erfassen und somit sicht- und handhabbar zu machen. Damit wird auch die geforderte Flexibilität erreicht, da die Integration neuer Aspekte in ein bestehendes Management auf einer abstrakten Ebene leichter zu vollziehen ist, als in einem handgeschriebenen Management.

Ebenso können für einzelne Spezialfälle angepasste Managementstrukturen erzeugt werden, wodurch die Notwendigkeit zum Einsatz einer uniformen Basis für die Ausführung verteilter Systeme entfällt.

Ein wesentlicher Gesichtspunkt des Managements verteilter Systeme ist dabei, dass das Management, um effiziente und korrekte Entscheidungen fällen zu können, die ausgeführten Systeme kennen muss. Dies führt zu einer Spezifikation, in der das „Wissen“ über Systeme eine entscheidende Rolle spielt. Auf der anderen Seite muss das Management aber auch die Realisierungsplattform kennen, auf welcher das System ausgeführt wird. Dies erfordert eine Integration von Anwendungsaspekten, wie sie üblicherweise durch Programmiersprachen beschrie-

ben werden, Aspekten der Realisierungsbasis durch die Spezifikationen, sowie der Transformation zwischen diesen. Damit einerseits ein generatives Verfahren auf der Grundlage der Spezifikation möglich wird und andererseits die Anwendbarkeit des Ansatzes gewährleistet bleibt, ist eine uniforme Spezifikation dieser Teilaspekte notwendig.

Die zentralen Leitlinien dieser Arbeit sind daher die uniforme Spezifikation des Managements verteilter Systeme unter Berücksichtigung des generativen Aspektes dieser abstrakten Beschreibungen. Die Spezifikationen sollen dabei einerseits die Mächtigkeit besitzen, alle notwendigen Aspekte der Systeme bzw. deren Managements ausdrücken zu können. Andererseits ist es notwendig, die Spezifikationen übersichtlich und strukturiert durchzuführen, damit sowohl die abstrakten Beschreibungen als auch die generierten Managementsysteme beherrschbar bleiben. Die Beherrschbarkeit der Systeme ist somit die zweite große Leitlinie dieser Arbeit. Die dritte und letzte Leitlinie stellt die Strukturierung dar. Zum Einen ist die Struktur der Systeme einer der Angelpunkte der Spezifikation, d. h. die Strukturen der Systeme werden durch die Spezifikation erfasst. Zum anderen muss auch die Spezifikation selbst strukturiert werden. Die einzelnen Teilaspekte in einer Spezifikation werden einzeln beschrieben und dann durch das generative Verfahren zusammengeführt. E. W. Dijkstra hat dieses Vorgehen als *principle of seperation of concerns* bezeichnet (vgl. [Dij76]).

Im weiteren ist diese Arbeit wie folgt aufgebaut:

**Kapitel 2** beschreibt die Grundlagen für das Management verteilter Systeme. Dabei werden auch einige Techniken für die Realisierung von anwendungsangepassten Laufzeitumgebungen analysiert.

**Kapitel 3** führt zunächst einige Spezifikationstechniken ein und diskutiert deren Anwendbarkeit für die Spezifikation des Managements verteilter Systeme. Darauf aufbauend werden die Grundlagen für eine Spezifikationssprache für das Management verteilter Systeme unter dem Aspekt der automatischen Generierung erarbeitet.

**Kapitel 4** definiert grundlegende Begriffe der Spezifikationen und beschreibt die Spezifikation von Komponenten, aus welchen die Systeme aufgebaut sind. Die Komponenten werden dabei unterschiedlichen Abstraktionsebenen zugeordnet, um eine einheitliche Spezifikation aller Teile eines Systems zu ermöglichen.

**Kapitel 5** beschäftigt sich mit den dynamischen Aspekten von Systemen. Da Systeme sich mit der Zeit verändern, ist es notwendig, diesen zeitlichen Aspekt in die Spezifikationen einzubeziehen.

**Kapitel 6** verbindet die Aspekte der beiden vorhergehenden Kapitel zu einer vollständigen Spezifikation. Mit diesen Konzepten kann dann das Management eines verteilten Systems vollständig beschrieben werden.

**Kapitel 7** diskutiert weiterführende Möglichkeiten, die Spezifikationen zu analysieren, um auf dieser Grundlage einerseits die Erfüllung gewünschter Eigenschaften sicher zu stellen, andererseits die Konfliktfreiheit der Spezifikationen zu überprüfen.

**Kapitel 8** beschreibt, wie aus der Spezifikation eines verteilten Managements eine ausführbare Infrastruktur für verteilte Systeme generiert werden kann. Mit den beschriebenen Techniken sind die Grundlagen für ein generiertes Management auf der Basis abstrakter Spezifikation gelegt.

**Kapitel 9** stellt PROMETHEUS, die prototypische Implementierung eines Generators für das Management verteilter Systeme auf der Basis der eingeführten Techniken vor. Mit PROMETHEUS wird die praktische Anwendbarkeit des Ansatzes demonstriert.

**Kapitel 10** zeigt die Verwendung der Spezifikationstechnik einerseits und die Verwendung der prototypischen Implementierung PROMETHEUS andererseits anhand verschiedener Beispiele verteilter Systeme auf.

**Kapitel 11** untersucht die Performanz des Ansatzes und beschreibt Möglichkeiten zur Optimierung der generierten Managementsysteme auf der Basis weiterer Analysen der Spezifikation.

**Kapitel 12** fasst die Erkenntnisse der Arbeit zusammen und diskutiert weiterführende Forschungsaufgaben im Bereich der Generierung von Managementsystemen für verteilte Umgebungen.

# 2 Grundlagen von Managementsystemen

Dieses Kapitel beschreibt zunächst einige grundlegende Aspekte des Entwurfs von Managementsystemen. Diese müssen von einer Methodik zur Spezifikation des Managements von Systemen berücksichtigt werden. Dazu werden einerseits verschiedene Ansätze zur Realisierung von nicht funktionalen Aspekten in Systemen untersucht. Andererseits werden verschiedene Betriebssystemrealisierungen diskutiert, welche verschiedene Ansätze für ein flexibles, anwendungsangepasstes Management von Systemen realisieren. Aus diesen Aspekten können die generellen Anforderungen an ein Management für verteilte Systeme abgeleitet werden. Dies ist die Grundlage für den Entwurf einer entsprechenden Spezifikationssprache.

Zunächst muss allerdings der bereits mehrfach gebrauchte, aber in dieser Arbeit bisher nicht definierte Begriff des Managements präzisiert werden. In dieser Arbeit wird die folgende Definition des Begriffs Management verwendet:

**Definition 2.1 (Management)**
*Das Management eines (verteilten) Systems stellt eine Ausführungsumgebung bereit, welche alle Anforderungen des Systems auf allen Abstraktionsebenen befriedigt.*

Das Management eines Systems kann zunächst als Betriebssystem aufgefasst werden. Die Aufgabe des Betriebssystems (vgl. Definition 1.3 auf Seite 2) ist die Bereitstellung von Ressourcen und Koordination der Zugriffe auf diese. Damit befriedigt das Betriebssystem die Anforderungen der Prozesse oder Systeme im Hinblick auf die Nutzung von Ressourcen. Die Anforderungen von Systemen können aber auch weitergehend sein und zum Beispiel die Qualität von Ressourcen betreffen. Ein anderer zentraler Aspekt liegt in der Abstraktionsebene der Systeme. Ein Betriebssystem kennt im Allgemeinen nur Prozesse als aktive Ein-

heiten, sowie Ressourcen als passive Einheiten. Diese Abstraktion für Systeme ist eine sehr niedrige Ebene. Systeme werden üblicherweise auf der Ebene von Programmiersprachen oder einer noch höheren Ebene beschrieben. Das Management eines Systems kennt die Systeme auch auf dieser Ebene der Abstraktion. Daher kennt das Management die Strukturen eines Systems auch auf einer hohen Abstraktionsebene und kann somit die zur Realisierung des Systems notwendigen Schritte selbständig durchführen. Dies ermöglicht einen flexibleren Umgang mit den Anforderungen der Systeme und auch den zur Verfügung stehenden Ressourcen.

Dieser Ansatz wird auch als sprachbasierter Ansatz bezeichnet, da dem Management die Strukturen der Anwendung und weitere Informationen über die auszuführenden Systeme zur Verfügung stehen. Dabei ist allerdings zu beachten, dass sprachbasiert in diesem Fall nicht die Beschränkung auf eine Programmiersprache impliziert wird, sondern durchaus eine Reihe von Sprachen unterstützt werden können.

## 2.1 Entwurfsprinzipien für Betriebsysteme

Für den Entwurf von Betriebsystemen wurden eine Reihe von Prinzipien vorgeschlagen und entwickelt. In diesem Abschnitt sollen einige dieser Prinzipien kurz beschrieben werden.

### Schichtung

Ein in vielen Bereichen der Informatik angewandtes Entwurfsprinzip ist die *Schichtung*. Eines der bekanntesten Beispiele für Schichtung ist das ISO/OSI Referenzmodell für Netzwerke. Das Prinzip eines geschichteten Systemaufbaus besteht darin, dass die Funktionalität einer Schicht auf der Basis der Funktionalität der darunterliegenden Schicht realisiert wird und Funktionen für die darüberliegende Schicht angeboten werden. Dadurch wird es möglich, die einzelnen Schichten unabhängig von einander zu realisieren, sofern die Schnittstellen zwischen den Schichten festgelegt sind und durch unterschiedliche Realisierungen nicht verändert werden. Dieses Vorgehen ermöglicht somit auch die Austauschbarkeit einzelner Schichten, ohne die darüberliegenden Schichten anpassen zu müssen.

Im Bereich der Betriebsysteme war das THE System (vgl. Abb. 2.1 auf der nächsten Seite) von Dijkstra das erste geschichtete System (siehe [Dij68]). Aber auch moderne Systeme, wie Windows 2000 (siehe [SR00]) oder UNIX-Systeme (siehe [MBKS96] sind geschichtete Systeme.

Ein anderes geschichtetes System stellt das Projekt *Flux* (vgl. [FBB+97] und [FHL+96]) dar, das einerseits Schichten definiert, andererseits aber alle Schich-

| Schicht | Funktionalität |
|---|---|
| 5 | Benutzer |
| 4 | Anwendungen |
| 3 | Ein-/Ausgabe |
| 2 | Kommunikation |
| 1 | Speicherverwaltung |
| 0 | Prozessorverwaltung und Multiprogrammierung |

Abbildung 2.1: Die Schichten des THE Systems.

ten mit einer identischen Schnittstelle versieht. Somit können die Schichten des Systems beliebig kombiniert und neue Schichten eingeführt werden.

### Mikrokerne und erweiterbare Kerne

Eine zunehmend bedeutsamer werdende Anforderung an Betriebsysteme ist die (dynamische) Erweiterbarkeit des Betriebsystems, um einerseits auf Veränderungen der Hardware reagieren zu können und andererseits spezielle Anforderungen der Anwendungen erfüllen zu können. Um diese Anforderungen zu erfüllen wurden zwei diametral entgegengesetzte Ansätze entwickelt.

Auf der einen Seite stehen die Mikrokerne mit Mach als ihrem klassischen Vertreter (vgl. [ZK93]) oder $\mu$-Kerne (vgl. [Lie96] und [AH99]), bei denen die Betriebsystemkerne auf ein Minimum reduziert werden und somit im Extremfall nur noch der reinen Abstraktion der Hardware dienen. Die eigentliche Betriebssystemfunktionalität wird in den Benutzerbereich des Systems als sog. Server ausgelagert. Da die Server als eigenständige (Benutzer-)Programme ausgeführt werden, ist ein Austausch der Server oder die Hinzunahme eines weiteren Server leicht möglich. Die Kommunikation zwischen den Servern bzw. den Anwendungen und den Servern erfolgt über Nachrichten. Daher eignet sich dieses Konzept auch für die Realisierung verteilter Betriebsysteme, bei welchen die Nachrichtenkommunikation über Rechnergrenzen hinweg erweitert wird.

Auf der anderen Seite stehen die als erweiterbar bezeichneten Systeme. Bei diesen Systemen ist das Ziel, Module dynamisch in den Kern einbringen zu können. Dabei müssen Schutzmechanismen realisiert werden, die das Einbringen von fehlerhaftem oder bösartigem Code in den Betriebsystemkern verhindern. Beispiele für derartige Ansätze sind *Vino* (vgl. [SSS95] und [SS95]) und *Paramecium* (vgl. [DHT95] und[HDS+95]).

### Trennung von Mechanismen und Politik

Eine der zentralen Erkenntnisse aus dem Entwurf von Betriebsystemen ist die Trennung der Mechanismen von der Politik eines Systems. Damit ist ein Wechsel der Politik des Systems möglich, ohne dass die Mechanismen verändert werden müssen. Werden beispielsweise Dispatcher und Scheduler in einem Betriebsystem sauber getrennt, so kann die Strategie für das Scheduling der Prozesse verändert werden, der notwendige Mechanismus zum Austausch des laufenden Prozesses kann dabei unverändert bleiben.

Die Trennung von Mechanismus und Politik führt zu einem geschichteten System. Eine Politik nutzt die Dienste eines Mechanismus, dieser selbst nutzt für die Erfüllung seiner Aufgabe wiederum Mechanismen in einer bestimmten Art und Weise und ist somit die Realisierung einer Politik auf einer niedrigeren Schicht. Die strikte Trennung der Mechanismen von der Politik eines Systems ermöglicht aber auch den Austausch der Mechanismen ohne Beeinflussung der Politik eines Systems.

Wesentlich ist dabei, dass die Mechanismen unabhängig voneinander kombinierbar sind. Diese Fähigkeit wird als Orthogonalität bezeichnet. Damit ist eine Flexibilität erreichbar, die den Einsatz der Mechanismen gemäß unterschiedlicher Politiken ermöglicht.

### Objektorientierte Systeme

Das Prinzip der Objektorientierung (vgl. [Mey88]), welches sich bei der Entwicklung von Anwendungen im vergangenen Jahrzehnt weitgehend durchgesetzt hat, dringt inzwischen auch in den Bereich der Betriebsysteme vor. Ein objektorientiertes System baut auf Zuständen, Aktivitäten und Kommunikation auf. Das zentrale Paradigma ist dabei die Kapselung der Datenstrukturen, die den Zustand repräsentieren und Operationen, welche von den Aktivitäten ausgeführt werden. Im Bereich der Betriebssysteme können drei unterschiedliche Varianten identifiziert werden, welche sich auf die Objektorientierung beziehen.

1. **Objektmanagementsysteme**
   Objektmanagementsysteme definieren objektorientierte Schnittstellen. Der Kern eines solchen Systems kann allerdings durchaus gemäß einem anderen Paradigma, wie zum Beispiel prozedurorientiert, gestaltet sein.

2. **Objektorientierte Kerne**
   Ein Betriebsystem, dessen Kern objektorientiert entworfen und realisiert wurde, wird als objektorientierter Kern bezeichnet. Die Schnittstelle des Kerns muss dabei allerdings nicht objektorientiert sein.

3. **Objektorientierte Betriebsysteme**
   Werden beide der vorherigen Ansätze kombiniert, besitzt ein System also

einen objektorientierten Kern und eine ebensolche Schnittstelle, so spricht man von einem objektorientierten Betriebsystem.

Der objektorientierte Ansatz stößt allerdings auch an seine Grenzen. Es ist in vielen Bereichen nicht möglich, eine Struktur in Objekte zu zergliedern, so dass alle funktionalen Abhängigkeiten sauber gekapselt sind. Diese Erkenntnis hat in jüngster Zeit zur Entwicklung weiterführender Ansätze, wie der Aspektorientierung (vgl. [CE00], [LV01] und [KLM+97]) geführt. Dieser Ansatz ergänzt die Objektorientierung, indem Funktionalitäten, welche nicht in der Klassenhierarchie repräsentiert werden können, in Aspekten gekapselt werden. Zur Umsetzung der Aspektorientierung existieren mehrere Ansätze.

Aspekte können durch eine eigene Sprache beschrieben und dann mit dem Code des objektorientierten Programms verschmolzen werden. Dieser Ansatz wird als linguistischer Ansatz bezeichnet (vgl. z. B. AspectJ [KHH+01]).

Ein anderer Ansatz sind aspektorientierte Frameworks, die Entwicklungsumgebungen zur Modellierung von Aspekten bereitstellen.

Den dritten interessanten Ansatz stellen reflektive Ansätze dar, bei welchen die Objekthierarchie durch geeignete Datenstrukturen wiedergespiegelt, also reflektiert wird. Diese sog. Meta-Objekte können zur Modellierung und Anwendung der Aspekte genutzt werden (vgl. [Sul01]). Diese Technik lässt sich auch im Bereich der Betriebssysteme anwenden (vgl. [CKF+01]).

Insbesondere der letztgenannte Ansatz wurde im Umfeld der Betriebssystem bereits seit längerer Zeit erforscht. Ziel dieser Forschungsarbeiten ist oder war die Adaptierbarkeit von Systemen zu erhöhen. Unter Adaptierbarkeit versteht man dabei (nach [Son93]) die Fähigkeit zur Veränderung der Infrastrukturen der Systemeinheiten. Der entsprechende Vorgang der Veränderung oder Anpassung wird als Adaptierung bezeichnet. Beispiele für adaptive Systeme, welche Reflektion zur Erreichung dieser Zielvorgabe nutzen sind Muse bzw. Apertos (vgl. [Yok92] und [Yok93]) oder das Java-Betriebsystem JX-OS (vgl. [GFWK02]).

## 2.2 Flexibles Management durch Wissen

Für jede Entscheidung ist Wissen notwendig. Fehlt dieses Wissen oder ist das Wissen nicht korrekt oder aktuell, so führt dies unter Umständen zu gravierenden Fehlentscheidungen. Wissen kann dabei als die Kombination von Information und einer geeigneten Interpretation aufgefasst werden. Erst durch eine geeignete Interpretation, die von Erfahrungen und Intuition gespeist wird, kann aus den reinen Fakten ein anwendbares Wissen entstehen. Im Zusammenhang mit dem Management komplexer Systeme ist Wissen in zweierlei Hinsicht relevant. Einer-

seits muss das Management Wissen über das System an sich besitzen. Dazu zählen die Abhängigkeiten der Teile des Systems untereinander, aber auch Informationen über die bisherige und die zukünftige Entwicklung des Systems. Andererseits ist aber auch Wissen über die Umgebung, d. h. die Basis, des Systems notwendig, da jedes System für seine Ausführung eine Realisierungsbasis benötigt.

Dieses Wissen kann durch (zufällige) Beobachtung, systematische Erfahrung (Experiment) oder deduzierende Erkenntnis gewonnen werden. Daher wächst das Wissen über ein System mit der Zeit.

Sowohl die Umgebung eines Systems als auch das System selbst sind ständigen Veränderungen unterworfen, an die sich das System wieder anpassen muss. Diese Anpassungen müssen vom Management durchgeführt werden. Dazu ist zunächst ein Erkennen der Veränderung und daran anschließend eine geeignete Reaktion notwendig.

Ein Management, welchem kein oder nur geringes Wissen über das System und die Umgebung zur Verfügung steht, ist nicht in der Lage, auf veränderte Situationen oder veränderte Anforderungen geeignet zu reagieren.

Eines der einfachsten und im Bereich der technischen Realisierung von anpassungsfähigen Systemen seit langer Zeit eingesetztes Verfahren sind Regelkreise. Dabei wird ein System beobachtet, die Beobachtung durch einen Regler bewertet und wieder in das System eingespeist (siehe Abb. 2.2).

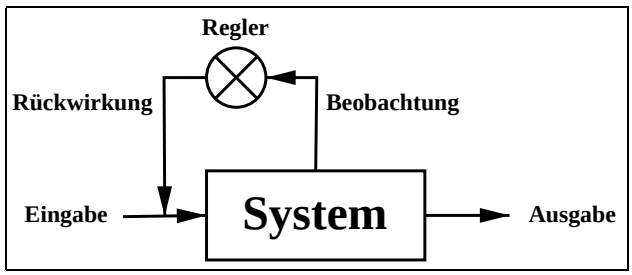



Abbildung 2.2: Der Regelkreis.

Diese Technik wurde im Synthetix-Projekt (vgl. [CPS+95]) untersucht und eingesetzt. Da Synthetix speziell für Multimedia-Anwendungen entwickelt worden ist, wurde die Technik beispielsweise für das Scheduling eingesetzt um Echtzeitfähigkeit, geringe Latenz und eine feine Granularität zu erreichen.

Im Allgemeinen bezeichnet man die dynamische Erfassung von Wissen über ein System als Monitoring (vgl. [BKLW00]). Dabei ist die Granularität der Erfassung von zentraler Bedeutung. Werden die Informationen über ein System mit einer zu feinen Granularität erfasst, so ist die Datenmenge nicht mehr handhabbar. Wird andererseits die Granularität zu grob gewählt, so sind die erfassten Informationen

nicht aussagekräftig. Die Abwägung zwischen diesen beiden Extremen ist komplex und muss für jedes System und jedes Teilsystem neu beantwortet werden.

Andere Techniken, die mit dem Ziel Selbststabilisierung von Systemen eingesetzt werden, sind jene oben bereits erwähnten Systeme mit Reflektion, wie beispielsweise Apertos. Die Ausführungsumgebung eines Objektes wird dabei durch sog. Meta-Objekte gebildet. Diese Meta-Objekte selbst erhalten ihre Ausführungsumgebung wiederum durch (Meta-)Meta-Objekte. Damit bildet sich eine Hierarchie von Objekten und Meta-Objekten. Die Meta-Objekte eines Objektes bilden dessen Meta-Raum. Dieser wird im System durch einen Reflektor repräsentiert. Die Objekte können über den Reflektor mit dem Meta-Raum kommunizieren. Den Abschluss gegenüber der Hardware bildet dabei ein stellenlokaler Mikrokern.

Ein wesentlicher Aspekt bei der Erfassung von Wissen über ein System ist die Erfassung der Strukturen in Systemen (vgl. [PFH02], [EP99a]). Die Systemstrukturen repräsentieren die Abhängigkeiten der Systembestandteile in mehrfacher Hinsicht. Einerseits werden funktionale Abhängigkeiten wie zum Beispiel Aufruf-hierarchien durch Strukturen erfasst, andererseits Nutzungsabhängigkeiten, wie zum Beispiel bei komplexen Datenstrukturen. Diese Strukturen können statischer Natur sein, wie zum Beispiel die Klassenhierarchie in objektorientierten Programmen, auf der anderen Seite aber existieren auch hochdynamische Strukturen. Die Ersteren können bereits vor der Ausführung eines Systems analysiert werden, die Letzteren erst zur Laufzeit, wobei allerdings statische Abschätzungen möglich sind. Diese Abschätzungen lassen sich durch das Wissen, welches beispielsweise bei einer vorherigen Ausführung gewonnen wurde, verbessern. Durch die Nutzung dieser Strukturen lässt sich ein anpassungsfähiges Management von Systemen konstruieren. Ein Beispiel für die Nutzung von Systemstrukturen für das Management von Systemen ist das Projekt MoDiS (vgl. z. B. [Piz99]) und dessen Programmiersprache Insel ([RW96]). Die dort verwendeten Systemstrukturen werden in Kapitel 10.2 auf Seite 191 ausführlich diskutiert.

## 2.3 Abstraktionsebenen von Systemen

Systeme existieren auf unterschiedlichen Abstraktionsebenen. Die unterste Ebene wird dabei durch die zugrundeliegende Hardware gebildet, die oberste Ebene durch eine abstrakte Beschreibung des Systems. Zwischen diesen beiden Extremen können eine Reihe von Ebenen liegen, wie zum Beispiel die programmiersprachliche Beschreibung des Systems in einer Hochsprache. Damit ergeben sich wiederum Schichten, die aufeinander folgen, wobei die Semantik der Systembeschreibung auf jeder Ebene gleich sein muss. Diese Form der Schichtung unterscheidet sich daher von der in Abschnitt 2.1 auf Seite 14 beschriebenen Schichtung eines Systems, bei welcher die Funktionalität mit den Schichten zunimmt.

Für die Konstruktion eines Systems ergibt sich somit ein Top-Down-Ansatz, bei dem ein System auf einer hohen Abstraktionsebene beschrieben wird und dann Schrittweise und Semantik erhaltend auf die vorhandene Realisierungsbasis abgebildet wird (siehe Abb. 2.3). Dieser Vorgang wird als *Konkretisierung* bezeichnet.

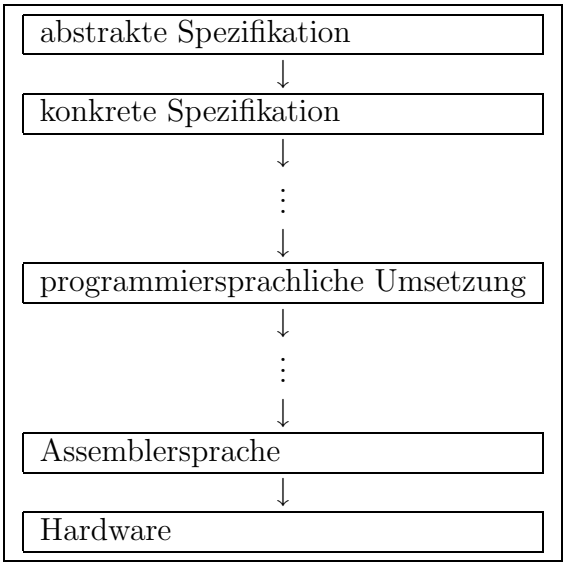



Abbildung 2.3: Abstraktionsebenen und Konkretisierung.

Die Konkretisierung eines Systems ist Aufgabe des Managements. Damit diese Aufgabe gelöst werden kann, ist es notwendig, einerseits das System auf der hohen Abstraktionsebene und andererseits das Ziel, also die Realisierungsbasis zu kennen. In komplexen, verteilten Systemen sind diese beiden Aspekte einer Dynamik unterworfen. Auf der Ebene der Realisierungsbasis können sich Veränderungen bezüglich der vorhandenen Hardware ergeben, indem zum Beispiel Knoten entfernt oder dynamisch hinzugenommen werden. Diese Möglichkeit wird insbesondere durch die Verbreitung mobiler Geräte immer zentraler.

Auf den höheren Abstraktionsebenen können sich ebenfalls Veränderungen ergeben. In einem klassischen Betriebssystem mit mehreren Prozessen beispielsweise können dynamisch neue Prozesse erzeugt und laufende Prozesse beendet werden.

Daher kann das Management die Abbildung von der hohen auf die niedrige Abstraktionsebene nicht oder zumindest nur teilweise statisch durchführen. Das Management muss mit dynamischen Veränderungen des Systems geeignet umgehen können.

### 2.3.1 Sprachbasierte Systeme

Für die Beschreibung eines Systems auf den unterschiedlichen Abstraktionsebenen stehen werden jeweils adäquate Sprachen eingesetzt. Auf einer Abstraktionsebene können dabei jeweils mehrere Sprachen definiert werden.

Für die Beschreibung Problemlösungsverfahrens durch den Anwender des Systems wird üblicherweise eine bzw. eine Menge von Sprachen festgelegt, welche als Programmiersprachen bezeichnet werden. Durch die Wahl einer Programmiersprache werden die Ausdrucksmöglichkeiten festgelegt, welche dem Anwender zur Beschreibung der Problemlösung zur Verfügung stehen.

Mit der Festlegung einer Sprache zur Beschreibung der Problemlösung werden auch die Möglichkeiten zur Strukturierung der Anwendungen definiert. Unterschiedliche Sprachen einer Abstraktionsstufe können dabei durchaus unterschiedliche Strukturierungsmöglichkeiten anbieten. Vielfach unterscheiden sich diese Sprachen beispielsweise in einer unterschiedlichen Gewichtung zwischen Daten und Abläufen. Während objektorientierte Sprachen Strukturierung der Daten in den Vordergrund stellen, sind bei imperativen Sprachen die Abläufe der Problemlösung im Mittelpunkt der Beschreibung.

Für das Management ist es wesentlich die Strukturieungsmöglichkeiten der Programmiersprache zu kennen, damit diese korrekt umgesetzt, d. h. auf die niedrigeren Abstraktionsebenen transformiert werden können. Diese Aufgabe entspricht der klassischen Aufgabenstellung einer Übersetzers (vgl. [WM97] oder [ASU88a]).

Da das Management aber, im Gegensatz zu einem klassischen Übersetzer, die Problemlösungsbeschreibung nicht isoliert von der zur Verfügung stehenden Ausführungsumgebung betrachtet, kann das Management die Transformation der programmiersprachlichen Repräsentation der Problemlösung an die konkret zur Verfügung stehenden Ressourcen durchführen. Eine Veränderung der Ressourcen kann somit zu einer veränderten Transformation führen.

Durch das Management wird auf diese Weise ein Informationsfluss zwischen den Ressourcen und der Problemlösungsbeschreibung etabliert, wodurch eine effizientere und an die Ausführungsumgebung angepasste Transformation ermöglicht wird.

Der Vorgang der Transformation durch das Management ist somit kein rein statische Verfahren, wie dies bei klassischen Übersetzern der Fall ist, sondern enthält dynamische Anteile. Neben der effizienteren Transformation der Problemlösungbeschreibung, eröffnet dies den Programmiersprachen auch weitere Flexibilität, da auch eine dynamische Veränderung der Lösungsbeschreibung durch eine Programmiersprache selbst mittels dieses Verfahren handhabbar wird.

### 2.3.2 Das Management als Ressourcen-Verwalter

Die zur Verfügung stehende Realisierungsbasis bzw. die niedrigeren Abstraktionsebenen für ein System sind die Ressourcen eines Systems auf einer Ebene. Der Begriff Ressource oder Betriebsmittel[1] kann daher folgendermaßen definiert werden:

**Definition 2.2 (Betriebsmittel)**
*Ein Betriebsmittel ist eine Hard- oder Softwarekomponente, welche zur Ausführung eines Systems benötigt wird.*

Für die Konkretisierung eines Systems müssen diese Betriebsmittel durch das Management verwaltet und den Teilsystemen zugeordnet werden. Die besondere Schwierigkeit dieser Aufgabe liegt in der im Allgemeinen begrenzten Anzahl an Betriebsmitteln. Betrachtet man die Zuordnung eines Betriebsmittels zu einem Teilsystem bzw. einer Komponente wiederum als Struktur, so erhält man eine dynamische Abbildung von Komponenten auf Betriebsmittel. Diese Zuordnung der Betriebsmittel wird einerseits durch die funktionalen Anforderungen der Komponenten, d. h. jener Anforderungen, deren Erfüllung einen Fortschritt in den Berechnungen des Systems erbringt, bestimmt. Neben diesen funktionalen Anforderungen existieren aber auch nicht-funktionale Anforderungen, wie zum Beispiel Qualitätsanforderungen. Beispiele hierfür sind Echtzeitsysteme, die sehr restriktive Anforderungen bezüglich der Zeit haben. Auch diese Anforderungen müssen auf einer hohen Abstraktionsebene beschrieben werden und entsprechend bei der Transformation auf eine niedrigere Ebene beachtet werden. Einige Ansätze in dieser Richtung, aus dem Umfeld der objektorientierten verteilten Systeme, finden sich beispielsweise in [BG97]. Das Ziel dabei ist, die Qualität von Diensten zu spezifizieren und für die Anwendungen zu garantieren. Dieser Ansatz wird in der Literatur als *Quality of Service* (QoS) (vgl. [BG98]) bezeichnet.

Bei der Abbildung nicht-funktionaler Anforderungen auf niedrigere Abstraktionsebenen tritt das Phänomen auf, dass diese auf einer niedrigeren Abstraktionsebene zu funktionalen Anforderungen werden. Insbesondere auf der niedrigsten Abstraktionsebene ist nicht zu unterscheiden, ob eine ausgeführte Operation der Erfüllung einer Funktionalität der oberen Ebenen oder einer nicht-funktionalen Anforderung dient.

[1] Da der Begriff „Ressource“ im Kontext dieser Arbeit speziell definiert wird (siehe Def. 4.12 auf Seite 70), wird hier der Begriff Betriebsmittel verwendet.

Im Gegensatz zu einem klassischen Betriebssystem, steht dem Management gemäß dem in Abschnitt 2.3.1 auf Seite 21 definierten Fähigkeiten zur Beschreibung von Problemlösungsverfahren Wissen über die ausgeführten Anwendungen zur Verfügung. Aus diesen Informationen kann das Management Vorhersagen über die zukünftige Entwicklung des Systems ableiten und somit die Nutzung der Ressourcen nicht nur auf die augenblickliche Situation hin optimieren. Damit besteht ein Informationsfluss von der Problemlösungsbeschreibung zu den Betriebsmitteln.

### 2.3.3 Managemententscheidungen

Mit dem vorher Gesagten ergibt sich die Aufgabenstellung des Managements als semantikerhaltende Transformation von einer hohen Abstraktionsebene auf niedrigere Abstraktionsebenen unter der Restriktion sowohl funktionaler, als auch nicht-funktionaler Anforderungen. Für die Erfüllung dieser Aufgabenstellung steht dem Management Wissen über das System einerseits und über die Realisierungsbasis andererseits zur Verfügung.

Das Management trifft auf dieser Basis Entscheidungen zur Realisierung des Systems und setzt diese entsprechend auf der niedrigeren Abstraktionsebene durch. Wie in [Gro98] dargestellt, kann dieser Vorgang als Transformation auf Graphen beschrieben werden. Die Regeln für Entscheidungen sind dann Graphersetzungsregeln, die den Übergang von einem Schnappschuss des Systems zum nächsten beschreiben.

Dabei entstehen aber auch Konflikte zwischen den Teilsystemen bzw. deren Anforderungen. Diese Konflikte müssen durch das Management aufgelöst werden. Für die Konfliktauflösung sind Regeln auf der Basis des Wissens des Managements bezüglich des Systems notwendig. Diese Regeln bestimmen das Verhalten des Managements und unterscheiden sich je nach Ausrichtung des Managements, so dass ein einheitliches Management für alle Systeme nicht möglich ist.

Durch die aufgrund dieser Regeln gefällten Entscheidungen wird das System vorrangetrieben. Das Management des Systems wird somit zum Motor, der den Fortschritt des Systems bewirkt. Die Dynamik des Systems wird also durch das Management selbst erzeugt.

Somit ergibt sich in dieser Sichtweise die Aufgabe des Managements als das Fällen von Entscheidungen bzgl. des „Wie“ der Realisierung eines Systems, sowie dessen Durchsetzung. Für diese Aufgabenstellung steht dem Management eine Sammlung von Mechanismen zur Verfügung, die als Fähigkeiten der niedrigeren Abstraktionsebene gesehen werden können.

Als problematisch zeigt sich dabei die Beschreibung der Regeln auf einer formalen Basis. So wird in [Gro98] zwar der Mechanismus beschrieben, mit dem ein solches Graphersetzungssystem als Management eines verteilten Systems realisiert werden kann, die notwendigen Graphersetzungsregeln werden aber nicht beschrieben.

## 2.4 Zusammenfassung

Das Management eines Systems kann als eine Kombination aus Betriebssystem, Laufzeitumgebung und Übersetzer oder Transformator für eine Systembeschreibung angesehen werden. Die Aufgabe des Managements ist die Verwaltung der Betriebsmittel, die Erfüllung sowohl funktionaler als auch nicht-funktionaler Anforderungen des Systems sowie die semantikerhaltende Abbildung eines Systems über mehrere Abstraktionsebenen. Diese Aufgabenstellung ist aufgrund der vielfachen Abhängigkeiten von hoher Komplexität und erfordert somit einen hohen Grad an Wissen über das System und die Realisierungsbasis. Dieses Wissen kann zum Teil statisch gewonnen oder abgeschätzt werden, muss aber auch durch Monitoring zur Laufzeit gewonnen werden.

Im Gegensatz zu klassischen Realisierungen von Systemen durch unterschiedliche, isolierte Werkzeuge wie Übersetzern, Bindern, Betriebssystemen usw. etabliert das Management einen bidirektionalen Informationsfluss zwischen allen Ebenen der Systemrepräsentation. Somit wird eine Gesamtsicht auf das System möglich, welche eine breite Basis für die Entscheidungen des Managements darstellt.

Für den Entwurf und die Realisierung des Managements existieren verschiedene Ansätze und Prinzipien, wie Schichtung und die Trennung von Mechanismen und Politik. Aber auch objektorientierte Ansätze und die Fragestellung der Adaptierbarkeit von Systemen und deren Management sind für den Entwurf eines Managements von Bedeutung.

Für die abstrakte Beschreibung des Managements mittels einer Spezifikation ist einerseits die Anwendung dieser Prinzipien in der Spezifikation selbst von Bedeutung, damit diese handhabbar bleibt. Andererseits müssen, da mittels der Spezifikationssprache unterschiedliche Managementsysteme beschreibbar sein sollen, die unterschiedlichen Konzepte ausgedrückt werden können.

Ein zentraler Aspekt dabei ist die Dynamik der Systeme, die durch das Management verwaltet werden muss. Einerseits sind die Systeme auf der abstrakten Ebene dynamisch und verändern sich mit dem Fortschritt der Berechnungen. Andererseits ist aber auch die Realisierungsbasis des Systems veränderlich. Beide Aspekte der Dynamik müssen in einer Spezifikation erfasst werden können, insbesondere da mit den Veränderungen des Systems und der Basis auch das

Wissen des Managements veränderlich ist. Das Management reagiert aber nicht nur auf die Dynamik des Systems, sondern ist selbst Motor der Veränderungen des Systems.

Für die Spezifikation sowie die Generierung des Managements ist es somit notwendig die unterschiedlichen Anteile des Managements beschreibbar zu machen. Die Spezifikation des Managements enthält daher Anteile die in klassischen Systemen dem Übersetzer, dem Binder, dem Lader, dem Betriebssystem usw. zugeordnet werden.

# Teil I

# Spezifikation von Managementsystemen


## Zusammenfassung

Im folgenden Teil der Arbeit werden zunächst verschiedene bekannte Techniken zur Systemspezifikation betrachtet. Dabei werden insbesondere die Vor- und Nachteile dieser Spezifikationstechniken zur automatischen Generierung des Managements diskutiert. Anschließend wird eine eigenschaftsorientierte, komponentenbasierte Spezifikationstechnik entwickelt, welche die Voraussetzungen der automatischen Generierung des Managements erfüllt.

# 3 Spezifikationssprachen und ihre Anwendung

Eine Spezifikation ist allgemein eine vollständig formale, von Implementierungen unabhängige Beschreibung des Verhaltens eines zu erstellenden Systems (vgl. [ECS93]). Die Ziele einer Spezifikation sind einerseits ein klares Verständnis des Problems bzw. der Aufgabenstellung. Andererseits beschreibt eine Spezifikation auch Eigenschaften, wie Zuverlässigkeit, Robustheit oder zeitliches Verhalten.

Für die Formulierung von Spezifikationen existieren eine Reihe von Ansätzen. Die meisten Spezifikationen werden in natürlicher Sprache formuliert. Dabei treten allerdings Mehrdeutigkeiten, Unklarheiten und andere Ungenauigkeiten auf. Diese Form der Spezifikation ist somit für weiterführende Zwecke ungeeignet.

Auf der anderen Seite stehen formale Methoden zur Spezifikation, die sich grob in folgende Kategorien untergliedern lassen:

a) Petri-Netze

b) diverse Kalküle der Logik

c) algebraische Spezifikationen

d) algebraische Kalküle und Prozessalgebren

e) sprachliche Ansätze

Zwischen der Spezifikation durch natürliche Sprache und den formalen Methoden sind Spezifikationstechniken einzuordnen, wie z. B. UML, die einerseits nicht die formale Exaktheit der Spezifikation durch formale Methoden erreichen, andererseits aber eine präzisere Spezifikation als mittels natürlicher Sprache erlauben. Diese Methoden zeichnen sich im Allgemeinen dadurch aus, dass sie leichter verständlich sind als formale Methoden, da zumeist eine graphische „Sprache“ zur Beschreibung verwendet wird. Diese Methoden werden heute im Bereich des Software Engineering eingesetzt.

In diesem Kapitel soll zunächst ein Überblick über die existierenden Spezifikationssprachen gegeben werden, wobei deren Vor- und Nachteile im Hinblick auf

einen generativen Ansatz und auf die Anwendbarkeit im Bezug auf die Spezifikation des Managements diskutiert werden. Dieser Überblick beginnt mit der Sprache UML, als Vertreter eines eher informellen Ansatzes. Anschließend werden einige ausgewählte Beispiele für formale Ansätze näher betrachtet. Es existieren neben diesen hier beschriebenen Ansätzen noch eine Reihe weiterer Spezifikationstechniken beispielsweise die sog. ausführbaren Spezifikationen (siehe [GR00], [Exe01]), welche Spezifikationen auf abstrakte Maschinen transformieren, um die Spezifikation testen zu können. Diese Art der Ausführbarkeit bedeutet aber nicht, dass aus den Spezifikationen reale Systeme erzeugt werden. Eine eigenständige Implementierung der Spezifikation ist bei diesen Ansätzen nachwievor erforderlich.

In Anschluss an die Diskussion einiger bekannter Spezifikationstechniken, beschreibt der Abschnitt 3.3 Techniken zur abstrakten Beschreibung von Wissen und dessen Gewinnung. Dabei werden insbesondere einige Techniken beschrieben, die im Umfeld des Übersetzerbaus angesiedelt sind, sowie deren Anwendbarkeit für die Spezifikation des Managements.

Aus diesen Betrachtungen können dann die Anforderungen an den Entwurf einer Spezifikationssprache für die Beschreibung des Managements von Systemen abgeleitet werden.

## 3.1 Informelle Ansätze

Ende der 80er, Anfang der 90er Jahre des vergangenen Jahrhunderts enstanden eine Reihe von objektorientierten Entwurfs- und Analysemethoden auf. Die Unified Modeling Language (UML) (siehe [OMG01a], [BIR97]) ist der Nachfolger eines Teils dieser Methoden. Die Elemente der Sprache UML sind Diagramme, es handelt sich bei UML also um eine graphische Sprache. Die Elemente der Sprache gliedern sich wie folgt:

1. **Anwendungsfalldiagramm**
   Ein Anwendungsfalldiagramm besteht aus einer Menge von Anwendungsfällen und stellt die Beziehungen zwischen Akteuren, d. h. Einheiten außerhalb des Systems, und Anwendungsfällen dar. Es zeigt das äußerlich erkennbare Systemverhalten aus der Sicht eines Anwenders. Ein Anwendungsfalldiagramm beschreibt die Zusammenhänge zwischen verschiedenen Anwendungsfällen untereinander und zwischen Anwendungsfällen und den beteiligten Akteuren.

2. **Klassendiagramm**
   Eine Klasse ist eine Menge von Objekten, in der die Eigenschaften (Attribute), Operationen und die Semantik der Objekte definiert werden. Alle

Objekte einer Klasse entsprechen dieser Festlegung. Objekte sind die agierenden Grundelemente einer Anwendung.

Das Verhalten der Objekte wird durch die Möglichkeit eines Objektes, Nachrichten zu empfangen und zu verstehen, beschrieben. Dazu benötigt das Objekt bestimmte Operationen. Zusätzlich zu Eigenschaften und Fähigkeiten kann eine Klasse auch Definitionen von Zusicherungen, Merkmalen und Stereotypen enthalten.

In einigen Programmiersprachen, z.B. in Smalltalk können auch an Klassen Nachrichten gesendet werden und sie können Klassenattribute besitzen. Die Klassen für die Klassenobjekte werden als Metaklassen bezeichnet.

Eine abstrakte Klasse bildet die Grundlage für weitere Unterklassen. Sie wurde bewusst unvollständig gehalten. Eine abstrakte Klasse stellt häufig einen Allgemeinbegriff dar, einen Oberbegriff, von dem konkrete Begriffe abgeleitet werden.

3. **Aktivitätsdiagramm**
   In einem Aktivitätsdiagramm werden die Objekte eines Programms mittels der Aktivitäten, die sie während des Programmablaufes ausführen, beschrieben. Eine Aktivität ist ein einzelner Schritt innerhalb eines Programmablaufes, d. h. ein spezieller Zustand eines Modellelementes das eine interne Aktion sowie eine oder mehrere von ihm ausgehende Transitionen enthält. Gehen mehrere Transitionen von der Aktivität aus, so müssen diese mittels Bedingungen von einander zu unterscheiden sein. Somit gilt ein Aktivitätsdiagramm als Sonderform eines Zustandsdiagrammes, dessen Zustände in der Mehrzahl als Aktivitäten definiert sind.

   In einem Programmablauf durchläuft ein Modellelement eine Vielzahl von Aktivitäten, d.h. Zuständen, die eine interne Aktion und mindestens eine daraus resultierende Transition enthalten. Die ausgehende Transition impliziert den Abschluss der Aktion und den Übergang des Modellelementes in einen neuen Zustand bzw. eine neue Aktivität. Diese Aktivitäten können in ein Zustandsdiagramm integriert werden oder besser aber in einem eigenen Aktivitätsdiagramm visualisiert werden. Ein Aktivitätsdiagramm ähnelt in gewisser Weise einem prozeduralem Flussdiagramm, jedoch sind alle Aktivitäten eindeutig Objekten zugeordnet, d.h. sie sind entweder einer Klasse, einer Operation oder einem Anwendungsfall eindeutig untergeordnet.

4. **Kollaborationsdiagramm**
   Die verschiedenen Modellelemente eines Programms agieren innerhalb des Programmablaufes miteinander. Um diese Interaktionen für einen bestimmten begrenzten Kontext, unter besonderer Beachtung der Beziehungen unter den einzelnen Objekten und ihrer Topographie, darzustellen, wird das Kollaborationsdiagramm verwendet.

Das Kollaborationsdiagramm visualisiert die einzelnen Objekte und ihre Zusammenarbeit untereinander. Dabei steht im Vergleich zum Sequenzdiagramm der zeitliche Ablauf dieser Interaktionen im Hintergrund, vielmehr werden die für den Programmablauf und das Verständnis des selbigen wichtigen kommunikativen Aspekte zwischen den einzelnen Objekten ereignisbezogen dargestellt. Der zeitliche Verlauf der Interaktionen wird lediglich durch eine Nummerierung der Nachrichten symbolisiert. Die einzelnen Objekte können Nachrichten austauschen, ein Objekt kann sich jedoch auch stets selbst Nachrichten zusenden, ohne dass eine Assoziation vorhanden sein müsste. Das Kollaborationsdiagramm kann für die Darstellung von Entwurfs-Sachverhalten benutzt werden und – in etwas detaillierter Form – von Realisierungssachverhalten. Es beinhaltet stets kontextbezogene begrenzte Projektionen des Gesamtmodells.

5. **Sequenzdiagramm**
   Das Sequenzdiagramm beschreibt die zeitliche Abfolge von Interaktionen zwischen einer Menge von Objekten innerhalb eines zeitlich begrenzten Kontextes.

   Mittels des Sequenzdiagrammes beschreibt man die Interaktionen zwischen den Modellelementen ähnlich wie bei einem Kollaborationsdiagramm, jedoch steht beim Sequenzdiagramm der zeitliche Verlauf des Nachrichtenaustausches im Vordergrund. Die Zeitlinie verläuft senkrecht von oben nach unten, die Objekte werden durch senkrechte Lebenslinien beschrieben und die gesendeten Nachrichten waagerecht entsprechend ihres zeitlichen Auftretens eingetragen.

6. **Zustandsdiagramm**
   Ein Objekt kann in seinem Leben verschiedenartige Zustände annehmen. Mit Hilfe des Zustandsdiagrammes visualisiert man diese, sowie Funktionen, die zu Zustandsänderungen des Objektes führen.

   Ein Zustandsdiagramm beschreibt eine hypothetische Maschine, die sich zu jedem Zeitpunkt in einem Zustand aus einer Menge endlichen Menge an Zustände befindet. Sie besteht aus:

   - einem Anfangszustand
   - einer endlichen Menge von Zuständen
   - einer endlichen Menge von Ereignissen
   - einer endlichen Anzahl von Transitionen, die den Übergang des Objektes von einem zum nächsten Zustand beschreiben
   - einem oder mehreren Endzuständen

7. **Komponentendiagramm**
   Damit bei späterer Implementierung der Softwarelösung Compiler- und Laufzeitabhängigkeiten klar sind, werden die Zusammenhänge der einzelnen Komponenten der späteren Softwarelösung in einem Komponentendiagramm dargestellt.

   Eine Komponente stellt ein physisches Stück Programmcode dar, welches entweder ein Stück Quellcode, Binärcode oder ein ausführbares Teilprogramm der gesamten Softwarelösung sein kann.

8. **Einsatzdiagramm**
   Ein Einsatzdiagramm beschreibt, welche Komponenten (Objekte) auf welchen Knoten ablaufen, d.h. wie diese konfiguriert sind und welche Abhängigkeiten bestehen.

UML versteht sich selbst als Modellierungssprache, nicht als Methode. Durch die graphische Notation wird eine leichte Verständlichkeit der Beschreibung erreicht. Allerdings ist die semantische Bedeutung der Beschreibung an verschiedenen Stellen der Spezifikation nicht vollständig, so dass UML als formale Spezifikation nicht geeignet ist.

Trotzdem ist es möglich aus UML-Diagrammen Code zu generieren. Dies liegt in der teilweise sehr niedrigen Abstraktionsebene der Beschreibungen von UML. Klassendiagramme lassen sich beispielsweise 1:1 in Rahmen für Klassen einer objektorientierten Programmiersprache übersetzen.

Für eine formale Spezifikation ist UML allerdings ungeeignet. Insbesondere für die Beschreibung des Managements von Systemen eignet sich UML nicht. Vielmehr ist UML ein Hilfmittel, um Realisierungen in einem Software-Engineering-Prozess zu beschreiben und eine einheitliche Sprache dafür anzubieten, welche durch Werkzeugunterstützung den Entwicklungsprozess strukturiert und vereinfacht.

## 3.2 Formale Ansätze zur Spezifikation

Die formalen Ansätze zur Spezifikation wurden vor allem in den 70er und 80er Jahren des vorigen Jahrhunderts entwickelt. Die wichtigsten Ansätze, die im folgenden diskutiert werden, sind die Petri-Netze, CCS als Vertreter der Prozessalgebren, sowie CSP als sprachlicher Ansatz.

### 3.2.1 Petri-Netze

Von C. A. Petri wurde bereits 1962 (vgl. [Pet62]) ein Modell zur Beschreibung und Analyse von Abläufen mit nebenläufigen, nicht-deterministischen Vorgängen vorgeschlagen. Ein Petri-Netz ist ein gerichteter Graph mit zweierlei Sorten von Knoten, den Stellen und den Transitionen. Eine Stelle entspricht einer Zwischenablage für Daten, eine Transition beschreibt die Verarbeitung von Daten. Die Kanten des Graphen dürfen nur von Knoten der einen Sorte zu Knoten der anderen Sorte führen. Die Stellen, von welchen eine Kante zu einer Transition $t$ führt, heißen Eingabestellen von $t$, die Stellen, zu denen eine Kante von $t$ führt heißen Ausgabestellen von $t$. Dieser Graph spiegelt feste Ablaufstrukturen wieder. Die Dynamik in Systemen wird beschrieben, indem Stellen mit sog. Marken, die Objekte von einem Datentyp sind, belegt werden und diese durch Transitionen weitergeleitet werden. Sind die Marken von einem boolschen Datentyp, so spricht man von Bedingungs-Ereignis-Netzen, sind die Datentypen von der Sorte der natürlichen Zahlen, so werden die Netze auch als Stellen-Transitionsnetze bezeichnet. Für die Transitionen eines Petri-Netzes gelten die folgenden Regeln:

a) Eine Transition $t$ kann schalten, wenn jede Eingabestelle von $t$ mindestens eine Marke enthält.

b) Mit dem Schalten einer Transition wird von jeder Eingabestelle eine Marke entfernt und auf jeder Ausgabestelle eine Marke hinzugefügt.

Petri-Netze werden zur Modellierung von Ablaufstrukturen, bei denen verschiedene nebenläufige Prozesse synchronisiert werden müssen, oder die Prozesse um Betriebsmittel konkurrieren, verwendet. Für Petri-Netze existieren automatisierte Beweisverfahren (vgl.[SSE02a] und [SSE02b]) für eine Reihe von Fragestellungen, wie zum Beispiel:

1. **Terminierung**
   Die Fragestellung ist in diesem Fall, ob ausgehend von einer Startbelegung der Stellen mit Marken die Transitionen stets nur endlich oft schalten können.

2. **Lebendigkeit**
   Hierbei geht man von einer Startbelegung der Stellen mit Marken aus und untersucht die Fragestellung, ob eine vorgegebene Transition nochmals schalten kann. In diesem Fall wird die Transition als lebendig bezeichnet, andernfalls als tot.

3. **Deadlockfreiheit**
   Wenn ein Petri-Netz verklemmt, so ist eine Situation aufgetreten, in welcher keine Transition mehr schalten kann. Diese Situation wird als Deadlock bezeichnet.

4. **Erreichbarkeit**
   Ausgehend von zwei Belegungen der Stellen mit Marken wird die Fragestellung untersucht, ob ausgehend von der ersten eine Schaltfolge existiert, so dass die zweite Belegung erreicht wird.

Diese Fragestellungen können durch temporale Logik (vgl. [Eme90]) über Petri-Netze formuliert werden. Damit existiert ein formales, mathematisches Instrumentarium zur Untersuchung von nebenläufigen Vorgängen.

Petri-Netze bieten somit ein Instrumentarium zur formalen Verifikation von Systemen. Allerdings ist der Abstraktionsgrad der Petri-Netze so hoch, dass eine Generierung von Systemen aus Petri-Netzen nicht möglich ist. Somit sind Petri-Netze als ein mathematisches Instrumentarium zur Untersuchung von Systemen interessant, aber nicht als Mittel der Spezifikation zur Generierung von Systemen.

Allerdings ist es möglich, bestimmte Teilaspekte einer Spezifikation automatisiert in Petri-Netze zu überführen und somit Eigenschaften der Spezifikation mit den Instrumentarien der Petri-Netze zu verifizieren. Ein derartiger Ansatz wurde beispielsweise in [Sch98] für die Spezifikationssprache SDL (siehe Abschnitte 3.2.6 auf Seite 42), welche im Umfeld der Telekommunikation entwickelt und eingesetzt wird, durchgeführt.

### 3.2.2 Ereignisgraphen

Ereignisgraphen sind ein Modell zur Simulation der zeitlichen Abhängigkeiten in Systemen (siehe z. B. [Sch83], [Bus00]). Dies wird als diskrete Ereignismodellierung bezeichnet. Die zentralen Elemente des Modells der diskreten Ereignissimulationen sind einerseits Zustandsvaribalen, andererseits Ereignisse. Das Modell emuliert das zu untersuchende System durch die Erzeugung von Zustandsspuren. Aussagen bezüglich der Performanz von Systemen werden durch statistische Untersuchungen dieser Zustandsspuren ermöglicht. Diskrete Ereignismodelle haben Zustandsspuren, die abschnittsweise konstant sind. Ein Ereignis ist dann ein Zeitpunkt, zu dem sich mindestens eine Zustandsvariable verändert. Ereignisse selbst sind dabei zeitlos, d. h. es verstreicht keine Zeit, wenn ein Ereignis auftritt.

Das zeitliche Auftreten von Ereignissen wird durch eine Liste der zukünftigen Ereignisse gesteuert. Diese Liste ist eine ToDo-Liste der geplanten Ereignisse. Immer wenn ein Ereignis auftreten soll, so wird eine Ereignisbenachrichtigung in diese Ereignisliste eingefügt. Jedes Ereignis trägt zwei Arten von Information:

a) Welches Ereignis stattfinden soll.

b) Die (simulierte) Zeit, zu der das Ereignis auftreten soll.

Die Liste der zukünftigen Ereignisse verwaltet die Ereignisse entsprechend dieser Zeiten. Ereignisse, die gleichzeitig auftreten sollen, müssen nach weiteren Regeln in eine Reihenfolge gebracht werden.

Die Simulation wird von einem Zeitgeber vorangetrieben. Dieser terminiert, wenn die Ereignisliste leer ist, d. h. die Simulation beendet ist, ansonsten erhöht dieser die simulierte Zeit bis zum ersten Ereignis in dieser Liste. Sobald ein Ereignis auftritt, werden alle Zustandsänderungen ausgeführt, die mit dem Ereignis verbunden sind. Dann werden alle weiteren Ereignisse erzeugt und schließlich wird die Ereignisbenachrichtigung aus der Liste entfernt. Die zu erzeugenden Ereignisse sind durch das Ereignis bestimmt. Für das Modell spielt die Reihenfolge dieser drei Schritte eine Rolle. Insbesondere würde eine Vermischung dieser drei Schritte zu Verwirrungen führen.

Ereignisgraphen sind nun eine Möglichkeit, die Logik in der Liste der zukünftigen Ereignisse zu beschreiben. Ein Ereignisgraph besteht aus Knoten und Kanten, wobei jeder Knoten einem Ereignis oder einem Zustandsübergang entspricht und jede Kante der Erzeugung weiterer Ereignisse. Darüberhinaus kann jede Kante mit einem boolschen Ausdruck versehen werden, sowie der Angabe einer Zeitverzögerung. Ein Ereignis $A$ erzeugt ein Ereignis $B$ mit einer Verzögerung $t$, falls die Bedingung $(C)$ wahr ist (siehe Abb. 3.1).

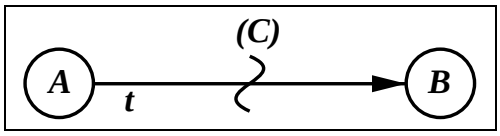



Abbildung 3.1: Ein einfacher Ereignisgraph.

Tritt das Ereignis $B$ ohne Zeitverzögerung ein, so wird $t$ weggelassen, ebenso, falls $B$ unbedingt auftritt, die Bedingung $C$. Ein etwas komplexeres Beispiel, eine Warteschlange ist in Abb. 3.2 auf der nächsten Seite dargestellt. Die Zustandsveränderungen durch die Ereignisse sind am entsprechenden Knoten mit geschweiften Klammern dargestellt ($++$ steht für das Inkrementieren, $--$ für Dekrementieren von Variablen).

Obwohl Ereignisgrahen bzw. diskrete Ereignis Simulationen keine Spezifikationen im eigentlichen Sinn darstellen, sind diese Modelle für die Darstellung von zeitlichen Abläufen gut geeignet. Die im folgenden dargestellten Spezifikationstechniken verwenden durchaus ähnliche Modelle für die Beschreibung zeitlicher Abläufe. Für die Generierung von Systemen sind insbesondere die in diesen Arbeiten gemachten Erfahrungen bzgl. der Umsetzung des Zusammenhangs zwischen zeitlichen Abläufen und den Veränderungen des Zustands interessant.

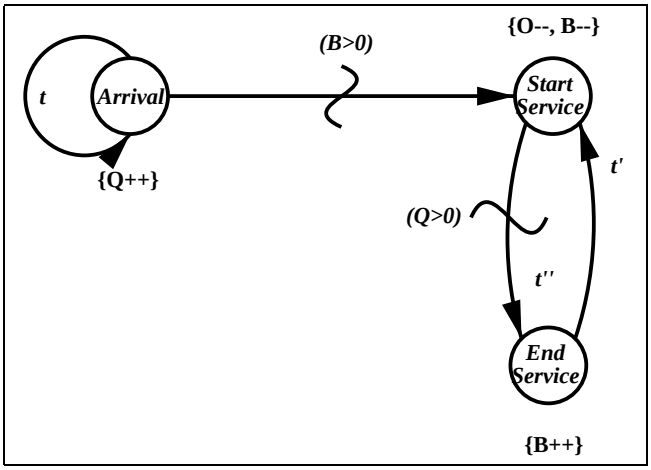



Abbildung 3.2: Ein Ereignisgraph für eine Warteschlange.

### 3.2.3 Communicating Sequential Processes – CSP

CSP (siehe [Hoa85] und [BHR84]) war einer der ersten Ansätze und ist immer noch einer der einfachsten und elegantesten Formalismen. Das zentrale Element zur Beschreibung von Verhalten und Kommunikation ist in CSP ein Ereignis. Ereignisse können entweder einfach sein oder aber mit Daten verknüpft werden.

Ein Prozess in CSP ist eine Einheit, die Ereignisse verarbeiten kann. Die Menge der Ereignisse, die ein Prozess verarbeiten kann, wird als dessen Alphabet bezeichnet. Der einfachste Prozess ist `STOP`, welcher keine Ereignisse verarbeiten kann. Weitere Prozesse werden aus Ereignissen und anderen Prozessen durch Operatoren aufgebaut. Der erste dieser Operatoren ist der Präfix-Operator $\rightarrow$. Ein Prozess, der das Ereignis $e$ verarbeitet, und sich dann wie der Prozess $P$ verhält, wird als $e \rightarrow P$ geschrieben. Durch den Präfix-Operator können auch rekursive Prozesse beschrieben werden, z.B. verarbeitet der Prozess $P = e \rightarrow P$ eine beliebige Anzahl von Ereignissen $e$.

Ein weiterer Operator ist ist die bedingte Auswahl von alternativen Wegen in Prozessen. Dafür stehen zwei Operatoren zur Verfügung. Wird die Entscheidung extern getroffen, so wird die Auswahl zwischen zwei Prozessen als $P \Box Q$ geschrieben. Der zweite Auswahloperator, die interne Auswahl, bei der die Entscheidung nicht-deterministisch durch den Prozess selbst getroffen wird, wird als $P \sqcap Q$ geschrieben.

Aus den Prozessen können Systeme konstruiert werden, indem die Prozesse parallel mittels dem Operator $\|$ kombiniert werden. Durch diesen Operator werden Prozesse zusammengefasst, die über die Ereignisse, welche sie verarbeiten, miteinander kommunizieren. Somit sind die Zusammenhänge zwischen den Prozessen

in CSP durch Ereignisse beschreibbar. Durch Ereignisparameter können Informationen zwischen den Prozessen ausgetauscht werden. Diese Ereignisse werden dann auch als Kanäle bezeichnet.

Mit der Sprache `occam` (siehe [SGS95]) steht eine Programmiersprache zur Verfügung, welche die Konzepte von CSP realisiert. In `occam` werden, analog zu CSP, Prozesse und deren Kommunikation definiert. Die Sprache `occam` besitzt eine so genannte statische Parallelität, d. h. zu Beginn der Abarbeitung eines `occam`-Programms muss feststehen, wieviele Prozesse parallel auftreten können. Die Prozesse und Kanäle werden in `occam` konkreten Prozessoren und Verbindungleitungen zugeordnet. Daher müssen `occam`-Programme Angaben über die verwendete Rechnerarchitektur enthalten. Dies wird als Konfiguration bezeichnet.

Mit CSP können die Abläufe von Prozessen durch Ereignisse, sowie die Zusammenhänge zwischen Prozessen beschrieben werden. Diese Art der Spezifikation ist für die Beschreibung von Abläufen in nebenläufigen Systemen gut geeignet. Da mit der Programmiersprache `occam` eine Realisierung von CSP existiert, können diese Spezifikationen auch leicht in einem generativen Prozess realisiert werden.

Allerdings können mittels CSP keine neuen Prozesse erzeugt werden, deren Ablauf bisher noch nicht bekannt ist. Dies ist aber eine für Managementsysteme wesentliche Anforderung, da zum Zeitpunkt der Spezifikation des Managements noch keinerlei Aussage über die auszuführenden Prozesse vorliegt. Des Weiteren ist eine der Hauptaufgaben eines Managementsystems die Verwaltung und Zuteilung von Ressourcen zu den Einheiten, die durch das Management verwaltet werden. Dabei ist zum Zeitpunkt der Spezifikation die konkrete Ausführungsumgebung im Allgemeinen nicht bekannt. Die in `occam` erforderliche Konfiguration der Systeme ist somit aber nicht möglich.

### 3.2.4 Communicating Concurrent Systems – CCS

Robin Milner entwickelte mit der Prozessalgebra CCS (siehe [Mil80], [Mil89]) ein Kalkül für kommunizierende nebenläufige Systeme. Dieser Kalkül wurde im Laufe der Zeit erweitert und dient als Basis für eine Reihe weiterer verwandter Kalküle, wie z. B. dem $\pi$-Kalkül (siehe [Mil93]) oder dem Actor-Kalkül (siehe [Agh86], [GZ99]).

Das zentrale Element einer Spezifikation in CCS ist ein Agent. Für jeden Agenten kann eine Menge von Ports definiert werden, wobei zwischen Eingabeports und Ausgabeports unterschieden wird. Jeder Agent befindet sich einem Zustand, der von den Aktionen, d. h. den Ein- und Ausgaben der Ports abhängig ist. In CCS werden Agenten und Zustände einander gleichgesetzt, so dass beide, ein Zustand und ein Agent, jeweils als ein Agent in einem Zustand aufgefasst werden.

Die Aktionen können mit Parametern versehen werden, um den Informationsaustausch zwischen Agenten zu modellieren. Ebenso können Agenten parametrisiert sein. Ein Agent $C$, welcher ein einzelnes Datenelement speichern kann, würde beispielsweise in CCS folgendermaßen definiert:

$$C = in(x).C'(x)$$
$$C'(x) = \overline{out}(x).C$$

Alternativ kann $C$ auch äquivalent als

$$C = in(x).\overline{out}(x).C$$

definiert werden. Diese Ausdrücke werden als Agentenausdrücke bezeichnet. Dabei werden Eingabeports bzw. deren Namen in Kleinbuchstaben notiert, während Ausgabeports ebenfalls durch Kleinbuchstaben, aber mit einem Überstrich beschrieben werden. Die Kommunikation zwischen Agenten findet über Ports mit dem gleichen Namen, aber unterschiedlicher Semantik statt, d. h. ein Agent definiert eine Eingabeport $x$ und ein weiterer Agent den Ausgabeport $\overline{x}$. Der Port $\overline{x}$ wird dann auch als Komplement von $x$ bezeichnet.

Agenten können durch eine Reihe von Operatoren miteinander verknüpft werden. Der Operator + steht für die alternative Kombination zweier Agentenausdrücke, die parallele Kombination wird durch | beschrieben. Weitere Operatoren sind die Restriktion von Port(-namen), welche es ermöglicht, die Bindung von Ports zu beschränken und die Umbenennung von Ports, um eine allgemeinere Beschreibung von Agenten zu ermöglichen.

Eine der zentralen Fragestellungen, die mittels CCS untersucht werden kann, ist die Äquivalenz von Agenten und somit von Systemen. Dies ermöglicht Aussagen im Bezug auf die korrekte Parallelisierung von Systemen, indem eine parallele Spezifikation mit einer sequentiellen Variante verglichen wird. Dabei können verschiedene Arten der Äquivalenz unterschieden werden, die starke Äquivalenz und die schwache Äquivalenz. Letztere beschränkt die Äquivalenz auf das nach außen sichtbare Verhalten und abstrahiert somit von internen Abläufen.

CCS kann dazu verwendet werden, die Semantik von Programmiersprachen für parallele Programme zu beschreiben. Programme können in CCS überführt werden, um mittels des Kalküls entsprechende Beweise zu führen. Zur Unterstützung der Beweisführung existiert eine sog. Workbench (vgl. [CJS88]), mit der CCS-Spezifikationen simuliert werden können.

Die Hauptfragestellung, die mit CCS untersucht wird, liegt in der Äquivalenz von Systemen, insbesondere in der sog. Beobachtungsäquivalenz. Dabei wird von internen Vorgängen innerhalb der Systeme abstrahiert und nur das beobachtete Verhalten untersucht. Für die Spezifikation von Managementsystemen ist dagegen insbesondere die Beschreibung der internen Vorgänge sowie Abhängigkeiten von Interesse. CCS ist nicht dafür vorgesehen, Systeme formal vollständig zu

beschreiben und auf dieser Basis durch Generierung reale Systeme zu erzeugen, sondern es ist ein Kalkül, mit dem bestimmte Eigenschaften nachgewiesen werden können.

### 3.2.5 FOCUS – AUTOFOCUS

Am Lehrstuhl IV der Fakultät für Informatik der Technischen Universität wurde die Methodik FOCUS (siehe [BDD+92], [BBSS97]) entwickelt, welche eine schrittweise Verfeinerung von Systemen von einer abstrakten Spezifikation bis zu einem ausführbaren System ermöglicht.

Im Sinne von FOCUS ist ein System eine von der Umgebung abgegrenzte Einheit, welche durch eine definierte Schnittstelle und ein definiertes Verhalten charakterisiert wird. Eine Systemschnittstelle besteht aus einer Menge von Kanalnamen, wobei ein Kanal eine unidirektionale Kommunikationsverbindung darstellt, dem ein Typ zugeordnet ist, als eine Menge von Nachrichten, welche auf diesem Kanal gesendet bzw. empfangen werden können.

Ein System selbst ist wiederum aus einer Menge von Komponenten zusammengesetzt, die als eigenständige Systeme mit Schnittstelle und Verhalten modelliert werden. Die Kommunikation zwischen den Komponenten geschieht ausschließlich durch den Nachrichtenaustausch über Kanäle.

Zur Modellierung der Kommunikation wird in FOCUS das mathematische Modell der Ströme verwendet. Ein Strom über einer beliebigen, nichtleeren Menge $M$ ist eine endliche oder unendliche Sequenz von Elementen aus $M$. Das Verhalten eines Systems wird dann zunächst in einer Black-Box-Sicht als eine Relation zwischen den Eingabeströmen einer Komponente und deren Ausgabeströmen modelliert. Dabei werden Eingabegeschichten mit Ausgabegeschichten in Verbindung gesetzt. In der Black-Box-Spezifikation wird das Verhalten des Systems durch ein Prädikat beschrieben, in welchem die Kanäle als freie Variablen vorkommen.

Durch Transitionssysteme wird beschrieben, wie das Verhalten eines Systems schrittweise erzeugt wird. Während Black-Box-Spezifikationen die Eigenschaften eines Systems beschreiben, können die Transitionssysteme als abstrakte Implementierungen aufgefasst werden. Ein Zustandstransitionssystem $\mathcal{S}$ wird durch ein Tupel $\mathcal{S} = \langle I, O, A, \Upsilon, T \rangle$ beschrieben, wobei $I$ die Eingabekanäle, $O$ die Ausgabekanäle, $A$ zusätzliche lokale Variable, $\Upsilon$ die Startzustände und $T$ die Transitionen sind. Dabei werden die Transitionen $T$ im Allgemeinen nicht direkt, sondern durch Prädikate beschrieben.

Ein Zustand des Transitionssystems besteht aus einer Belegung der Variablen $I \cup O \cup A$. Ein Transitionssystem ist in seiner Funktionsweise simpel. Ausgehend von einem Zustand geht es, ausgelöst durch Eingaben, in einen Nachfolgezu-

stand über, wobei Ausgaben produziert und lokale Variable neu belegt werden können. Transitionssysteme lassen sich auf einfache Art und Weise als gerichtete Graphen darstellen, wobei die Zustände als Knoten und die Transitionen als Kanten dargestellt werden. Diese Kanten werden mit Beschriftungen der Form $\{Pre\}Inputs \triangleright Outputs\{Post\}$ beschrieben, wobei $Inputs$ Eingabemuster und $Outputs$ Ausgabemuster sind. Die Schaltbereitschaft einer Transition wird durch das Prädikat $Pre$ spezifiziert, durch $Post$ wird eine Nachbedingung angegeben, die typischerweise den Datenvariablen neue Werte zuweist.

Zur Konstruktion komplexer Systeme können Teilsysteme zusammengesetzt werden. Die Komposition von Systemen in FOCUS wird dadurch vereinfacht, dass die (Teil-)Systeme nur durch ihre Ein-/Ausgabekanäle mit ihrer Umwelt kommunizieren und keinerlei Seiteneffekte erzeugen. Damit zwei (Teil-)Systeme komponiert werden können, müssen diese kompatibel sein, d. h. keine gemeinsamen Ausgabekanäle besitzen. Dabei werden sowohl die Black-Box-Spezifikationen als auch die Transitionssysteme zu einem gemeinsamen System kombiniert.

Ein in FOCUS interessanter Ansatz ist die schrittweise Verfeinerung, mit der die Entwicklung von komplexen Systemen erleichtert werden soll. Verfeinerungen sind Relationen zwischen zwei Varianten eines Systems, bei der die verfeinerte Variante im Wesentlichen das gleiche System wie die abstrakte Variante beschreibt, aber um zusätzliche Eigenschaften und Konkretisierungen ergänzt wurde. Dazu existieren zwei Verfeinerungsrelationen, die Verhaltens- und die Schnittstellenverfeinerung.

Mit dem Werkzeug AUTOFOCUS (siehe [HSSS96]) steht ein CASE-Tool zur Verfügung, welches die Entwicklung von Systemen mit der Methodik FOCUS unterstützt und teilweise automatisiert. Das Werkzeug AUTOFOCUS bietet daneben ein Projekt und Versionsmanagement sowie graphische Editoren zur Erstellung der Spezifikationen.

Mit der Arbeit [Spi98a] liegt auch ein Ansatz zur Spezifikation eines Betriebssystems mit der Methodik FOCUS vor. In dieser Arbeit werden die klassischen Aufgaben eines Betriebssystems, wie Prozessorverwaltung, Speicherverwaltung, Prozesskooperation und Prozessverwaltung formal beschrieben. Nicht betrachtet werden in dieser Arbeit allerdings die Gewinnung von Wissen über das System und dessen Strukturen, welche für die Generierung eines anwendungsangepassten Managements grundlegend sind. Hierin wird nochmals der grundlegende Unterschied zwischen einem Management und einem klassischen Betriebssystem deutlich.

Mit FOCUS existiert ein mächtiges Instrumentarium zur formalen Beschreibung von Systemen und deren Verhalten. Durch die Unterstützung mittels CASE-Tools ist eine teilautomatische Generierung von Systemen möglich. Aber auch die Methodik FOCUS ist nicht für Spezifikation von Managementsystemen entwickelt

worden, sondern für die Beschreibung von Anwendungssystemen. Zwar kann über die Ein-/Ausgabekanäle die Interaktion mit der Umwelt beschrieben werden, eine engere Kopplung zwischen Anwendung und Management ist allerdings nicht beschreibbar, so fehlen zum Beispiel Beschreibungsmittel für die Spezifikation der Abbildung von Anwendungen auf die real vorhandenen Ressourcen, sowie die Fähigkeit, Wissen über Systeme und deren Strukturen dynamisch zu erfassen.

### 3.2.6 Specification and Description Language – SDL

SDL ([EHS97], [ITU99]) basiert auf dem erweiterten Zustandsautomatenmodell, ähnlich dem Modell der Methodik FOCUS. Sie wurde ursprünglich von der CCITT, dem Vorgänger der ITU-T ( International Telecommunication Union – Telecommunication Standardization Bureau) für die Spezifikation von Vermittlungssoftware entwickelt.

Ein erweiterter endlicher Zustandsautomat (Extended Finite State Machine) $M$ ist ein 5-Tupel $M = (I, O, S, T, F)$, mit

- $I$ (Input) die Menge der Protokolleingänge darstellt,
- $O$ (Output) die Mengen der Protokollausgänge darstellt,
- $S$ (States) die Zustände des Automaten darstellt,
- $T$ (Transition) die Zustandsübergänge darstellt, die durch die Abbildung

  $$T : I \times S \rightarrow S$$

  definiert sind und
- $F$ (Output Function) die Ausgangsfunktionen darstellt, die durch die Abbildung

  $$F : I \times S \rightarrow O$$

  definiert sind.

SDL basiert auf vier Grundkonzepten:

- Ein System wird hierarchisch durch folgende Elemente beschrieben: System, Blöcke, Prozesse und Prozeduren.
- Das Systemverhalten wird durch einen erweiterten endlichen Zustandsautomaten beschrieben.
- Daten werden unter Verwendung von abstrakten Datenstrukturen und variablen Definitionen angegeben.

- Kommunikation wird asynchron über Kanäle, die als unendliche Warteschlangen modelliert werden, abgewickelt.

Die Systemebene bildet den Rahmen der Beschreibung. Sie besteht aus einem oder mehreren Blöcken, die jeweils ein Subsystem beschreiben. Blöcke sind mit Kanälen miteinander verbunden. Ein Block besteht aus mehreren Prozessen, die miteinander kommunizieren. Ein Prozess ist ein erweiterter endlicher Zustandsautomat. Eine Prozedur ist wiederum ein erweiterter Zustandsautomat innerhalb eines Prozesses.

Spezifikationen können in SDL sowohl graphisch, als auch textuell beschrieben werden. Beide Arten der Spezifikation sind äquivalent, so dass eine graphische Spezifikation, welche für den Benutzer leichter verständlich ist, in eine textuelle Repräsentation ohne Informationsverlust überführt werden kann. Die textuelle Repräsentation ermöglicht den Einsatz von Werkzeugen zur Generierung von Code und zur Verifikation.

SDL wurde für die Beschreibung von reaktiven, diskreten, parallelen Systemen entworfen. Für andersartige Systeme ist SDL nur schlecht oder gar nicht geeignet. Bei SDL steht die Reaktion auf Eingaben in Form von Signalen im Vordergrund. SDL ist nicht geeignet um Systeme zu beschreiben, die selbstständig Entscheidungen fällen und diese durchsetzen, wie dies für Managementsysteme erforderlich ist.

## 3.3 Statische Beschreibungen durch attributierte Graphen

Bei der Übersetzung von Programmiersprachen, also der Transformation einer Hochsprache in Maschinensprache, spielen zwei Techniken, sowie ihr Zusammenhang eine zentrale Rolle.

Einerseits werden (Programmier-)Sprachen durch Grammatiken (meist $LR(k)$-Grammatiken) beschrieben. Damit ergibt sich für ein Programm eine Baumstruktur, welche die abstrakte Syntax des Programms wiederspiegelt. Diese Struktur ist die Grundlage für alle Operationen des Übersetzungsvorgangs.

Andererseits wird beim Vorgang der Übersetzung Wissen über das Programm gewonnen, welches zum Einen für die Überprüfung von Kontextbedingungen benötigt wird, welche in einer kontextfreien Grammatik nicht ausdrückbar sind. Dieser Vorgang wird als semantische Analyse bezeichnet. Dieses Wissen wird unter Anderem bei der Generierung des Maschinencodes verwendet, beispielsweise um Optimierungen durchführen zu können.

Die Gewinnung und Nutzung von Wissen auf der Grundlagen der Baumstruktur eines Programms wird durch sog. attributierte Grammatiken beschrieben (siehe [Paa95] für einen Überblick). In einer attributierten Grammatik werden den Symbolen einer kontextfreien Grammatik (siehe [HMU01]) Attribute zugeordnet, welche statische, semantische Informationen tragen. Zusätzlich werden die funktionalen Abhängigkeiten zwischen den Werten von Attributvorkommen beschrieben. Der Vorgang der Transformation des Quellprogramms in die Zielsprache sowie die semantische Analyse ergeben sich somit als Berechnung dieser Attribute gemäß ihren Berechnungsfunktionen.

Formal kann eine attributierte Grammatik folgendermaßen definiert werden (nach [WM97]):

**Definition 3.1 (attributierte Grammatik)**
*Sei* $G = (V_N, V_T, P, S)$ *eine kontextfreie Grammatik. Sei die p-te Produktion in* $P$*:* $p : X_0 \to X_1 \dots X_{n_p}$, $X_i \in V_N \cup V_T$, $1 \leq i \leq n_p$, $X_0 \in V_N$. *Eine* attributierte Grammatik *über* $G$ *besteht aus:*

- *zwei disjunkten Mengen Inh und Syn von* ererbten *(inherited) bzw.* abgeleiteten *(synthesized) Attributen.*
- *einer Zuordnung von zwei Mengen* $Inh(X) \subseteq Inh$ *und* $Syn(X) \subseteq Syn$ *zu jedem Symbol aus* $V_N \cup V_T$*; die Menge der Attribute von* $X$ *wird mit* $Attr(X) = Inh(X) \cup Syn(X)$ *bezeichnet. Ist* $a \in Attr(X)$*, so hat* $a$ *ein* Vorkommen *in Produktion* $p$ *beim Vorkommen von* $X_i$*, geschrieben als* $a_i$*.* $V(p)$ *sei die Menge aller Attributvorkommen in Produktion* $p$*.*
- *der Festlegung eines* Wertebereichs $D_a$ *für jedes Attribut* $a$
- *der Angabe einer* semantischen Regel

  $$a_i = f_{p,a,i}(b^1_{j_1}, \dots b^k_{j_k},\ 0 \leq j_l \leq n_p,\ 1 \leq l \leq k$$

  *für jedes Attribut* $a \in Inh(X_i)$ *für* $1 \leq i \leq n_p$ *und jedes* $a \in Syn(X_0)$ *in jeder Produktion* $p$*, wobei* $b^l_{j_l} \in Attr(X_{j_l})$, $0 \leq j_l \leq n_p$, $1 \leq l$*.* $f_{p,a,i}$ *ist also eine Funktion von* $D_{b^1} \times \dots \times D_{b^k}$ *in* $D_a$*.*

Auf der Basis von attributierten Grammatiken können Auswerter für die semantische Analyse von Programmiersprachen generiert werden. Bei der Auswertung der Attribute bzw. bei der Generierung von Auswertern können dabei unterschiedliche Wege beschritten werden. Einerseits besteht die Möglichkeit, für gewisse

Klassen von attributierten Grammatiken im Vorhinein eine Reihenfolge für die Auswertung der Attribute festzulegen. Andererseits können attributierte Grammatiken auch ausgewertet werden, indem mit der Auswertung an der Wurzel begonnen wird und, ein Eingangsparamter einer Berechungsfunktion ausgewertet wird, falls dieser noch nicht bekannt ist. Damit werden die Abhängigkeiten zwischen den Attributen erst zur Auswertungszeit bestimmt. Der erstere Ansatz hat den Vorteil, dass Zyklen in den Abhängigkeiten bereits zur Generierungszeit erkannt werden können. Allerdings sind die Algorithmen dafür komplex und nur für eine Teilklasse der attributierten Grammatiken effizient anwendbar (vgl. [Kas80]).

Es existieren eine Menge derartiger Attributauswerter bzw. Generatoren für Attributauswerter, wie zum Beispiel die Werkzeugsammlung *Cocktail* (siehe [Gro00a] und [Gro00b]). Eine andere Sammlung von Werkzeugen zur Generierung von Übersetzern ist *Eli* (siehe [GLH+92] und [Slo95]), welches von mehreren Universitäten entwickelt wurde.

Ein weiterer interessanter Ansatz in diesem Bereich ist das am Lehrstuhl II für Informatik der Technischen Universität München von Arndt Poetzsch-Heffter entwickelte Tool *MAX* (Münchner Attributierungssystem für Unix) (siehe [PH93]). Ziel der Entwicklung von MAX war die Vereinheitlichung von syntaktischen und semantischen Methoden und die Kombination mit anderen formalen Konzepten, mit dem Ziel den Spezifikationskomfort zu verbessern und in natürlicher Weise weitere Anwendungsbereiche zu erschließen. Dazu wird die statische Semantik von Programmiersprachen basierend auf einer Kombination aus Prädikatenlogik und Attributgrammatiken beschrieben.

Alle diese Werkzeuge erwarten allerdings einen vollständigen Syntaxbaum, welcher attributiert werden soll. Mit dem Werkzeug *Synthesizer Generator* von Thomas Reps (vgl. [RT89]) wurde ein Ansatz entwickelt, der eine inkrementelle Auswertung von Attributen erlaubt. Damit ist es möglich, mit sich verändernden Strukturen in den Syntaxbäumen umzugehen.

Eine Erweiterung dieses Ansatzes auf verteilte Umgebungen wird zum Beispiel in [MK94] und [Kai93] beschrieben. Dieser Ansatz ermöglicht eine inkrementelle, parallele und verteilte Auswertung von Attributen. Die zentralen Problemstellungen dabei sind zum Einen die Konsistenz der Attribute und zum Anderen die Effizienz der Attributauswertung insbesondere durch Reduzierung der Netzwerkzugriffe. Die Konsistenz der Attribute wird bei diesem Ansatz durch ein dynamisches Sperren von Zugriffen auf bestimmte Baumteile während der inkrementellen Attributauswertung erreicht. Die Autoren bezeichnen diese Sperren als Firewall. Die Reduktion der Netzwerkzugriffe wird durch eine explizite Festlegung der potenziellen Verteilung und eine Einschränkung der Attributfunktionen an diesen Stellen erreicht. Dieses Vorgehen ermöglicht zwar eine effiziente Auswertung, erzwingt aber auch eine statische Festlegung der Verteilung.

Bei der Auswertung der Attribute ist das Vorkommen von Zyklen problematisch, da in diesem Fall keine Terminierung der Attributberechnung erfolgen kann. Daher werden in den meisten Fällen die attributierten Grammatiken auf die Untermenge der zyklenfreien attributierten Grammatiken eingeschränkt. Da die Überprüfung, ob eine Grammatik zyklenfrei ist, im Allgemeinen eine nicht triviale Aufgabe darstellt, werden in der Praxis Algorithmen verwendet, welche zwar zyklische Attributdefinitionen erkennen, aber auch Grammatiken zurückweisen, die nicht zyklisch sind. Eine Reihe von Forschungsarbeiten beschäftigt sich daher mit der Frage, wie zyklische Grammatiken bearbeitet werden können (siehe z. B. [Wal88] und [Hud91]).

Diese Techniken aus dem Bereich des Übersetzerbaus sind für die Spezifikation und Generierung des Managements verteilter Systeme aus zweierlei Gründen von Interesse. Da die Gewinnung von Wissen über eine Anwendung einer der zentralen Punkte für ein effizientes Management ist, muss die Anwendung geeignet analysiert werden. Bei einem sprachbasierten Ansatz, wie er hier verfolgt wird, liegt die Anwendung im Quellcode vor, so dass die Techniken des Übersetzerbaus für die Gewinnung von Wissen vor der Ausführung des Systems einsetzbar sind. Dieses Wissen kann durch Attribute repräsentiert und gemäß den Berechnungsfunktionen der attributierten Grammatik erarbeitet werden. Dabei bleibt die Frage zu klären, auf welche Weise dieser statische Teil des Managements mit dem dynamischen Management zur Laufzeit des Systems interagiert.

Eine andere Sichtweise ergibt sich, wenn man ein System in Ausführung betrachtet. Innerhalb eines Systems gibt es eine Reihe von Strukturen und Abhängigkeiten. Erstellt man zu einem Zeitpunkt im Ablauf des Systems einen Schnappschuss, so kann der damit sichtbare Graph ebenfalls mit Attributen versehen werden. Ein Ansatz zur Konstruktion derartiger Graphen wurde in [Gro98] erarbeitet. Dieser Graph verändert sich im Verlauf der Abarbeitung eines Systems. Diese Veränderungen bedingen Veränderungen in den Attributen. Durch eine inkrementelle Auswertung der Attribute zur Laufzeit können diese den jeweiligen Wissensstand des Managements repräsentieren.

## 3.4 Grundlagen des Entwurfs einer Spezifikationssprache

In diesem Abschnitt soll ein grober Rahmen für eine Spezifikationssprache für das Management von verteilten Systemen skizziert werden. Dabei wird einerseits auf die in Kapitel 2 erarbeiteten Anforderungen an Managementsysteme zurückgegriffen, andererseits fließen die Erfahrungen mit den in den Abschnitten 3.1 bis 3.3 beschriebenen Spezifikationstechniken in diese Skizze ein.

Die generellen Leitsätze für den Entwurf einer Spezifikationssprache sind die bereits in Abschnitt 1.4 auf Seite 9 genannten Punkte:

a) uniforme Spezifikation

b) Beherrschbarkeit

c) Strukturierung

Für die Spezifikation des Managements müssen verschiedene Aspekte in der Spezifikationssprache ausdrückbar sein. Einerseits muss das Wissen des Systems repräsentiert werden. Dazu zählen insbesondere auch die Strukturen des Systems. Diese beinhalten einerseits die Struktur innerhalb einer Abstraktionsebene, anderseits die Struktur der Transformation über die Abstraktionsebenen.

Um diese Strukturen herauszuarbeiten und anderseits die Spezifikation handhabbar zu machen, ist eine Zerlegung des Systems in Komponenten notwendig. Diese Zerlegung kann, da die Spezifikation des Managements vor dem System besteht, nur die potenziellen Komponenten eines Systems beschreiben. Für die Analyse des Systems auf dieser Basis muss auch das Wissen des Systems diesen Komponenten zugeordnet werden. Das Wissen des Systems wird, wie in Abschnitt 2.2 auf Seite 17 verdeutlicht, sowohl aus der statischen Beschreibung, als auch dynamisch durch Monitoring gewonnen. Die Spezifikation muss daher beide Aspekte abdecken. Diese Überlegungen führen zu einem dreiteiligen Ansatz für die Konstruktion eines Systems, wie in Abb. 3.3 dargestellt.

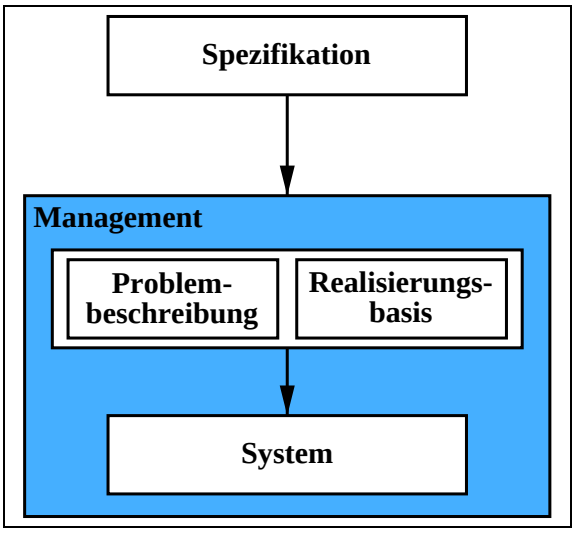



Abbildung 3.3: Dreiteilige Systemkonstruktion.

Aus der Spezifikation wird das Management erzeugt, welches einerseits die Erstellung einer Problembeschreibung oder eines Programms ermöglicht, indem

die Rahmenbedingungen dafür vorgegeben werden, d. h. die Programmiersprache spezifiziert wird. Aus dieser Problembeschreibung wird das System in Ausführung abgeleitet. Diese Ableitung wird als semantikerhaltende Transformation über die Abstraktionsebenen durch das Management geleistet. Damit dieser Prozess durch das Management geleistet werden kann, muss neben der Beschreibung des Problems bzw. dessen Lösung auch die Realisierungsbasis abstrakt beschrieben werden. Auch für diese ist eine Strukturierung und damit eine Zerlegung in Komponenten notwendig. In der Spezifikation werden dazu, analog zur Problembeschreibung, die potenziellen Ressourcen spezifiziert.

Durch die Beschreibung der Strukturen in der Spezifikation können somit auch die Regeln für die Transformationen über die Abstraktionsebenen beschrieben werden. Die Strukturen als Teil des Wissens über das System müssen durch das Management entsprechend verwaltet werden. Dieses Wissen kann den einzelnen Komponenten zugeordnet werden, womit eine verteilte Speicherung der Informationen erreicht wird. Die Beschreibung des Wissens wird in Anlehnung an die Techniken des Übersetzerbaus durch semantische Funktionen beschrieben und muss bei Veränderungen inkrementell erneuert werden.

Für die Modellierung der Dynamik eines Systems werden in den meisten Spezifikationssprachen Ereignisse bzw. Ereignisfolgen eingesetzt. Im Zusammenhang mit der Spezifikation des Managements und dessen Generierung sind dabei zwei Gesichtspunkte wesentlich. Zum Einen kann die Sicht auf die Ereignisse nicht auf die nach außen beobachtbaren Ereignisse eingeschränkt werden, da sonst nur das Verhalten gegenüber der Umwelt beschreibbar wäre und nicht die internen Abläufe, welche in diesem Zusammenhang aber wesentlich sind. Zum Anderen wird in den meisten Modellen eine Ereignisfolge mit einem Ausführungsfaden gleichgesetzt. Wird aber wie oben die Struktur des Systems als Grundlage der Spezifikation gewählt, so ist diese Sichtweise nicht zwingend. Eine Ereignisfolge für ein Teilsystem kann von diversen Ausführungsfäden erzeugt werden.

Ein zentraler Aspekt der Spezifikation ist der Zusammenhang zwischen den Ereignissen bzw. den Ereignisfolgen und dem Wissen des Managements. Die inkrementelle Veränderung des Wissens muss auf der Basis der Ereignisse beschreibbar sein. Die semantischen Funktionen, welche das Wissen bzw. dessen Erarbeitung des Systems spezifizieren, sind daher von den Ereignissen abhängig.

Neben der Strukturierung des Systems ist zur Erreichung einer handhabbaren Spezifikation auch die Strukturierung der abstrakten Beschreibungen erforderlich. Gemeinsame Teilaspekte der Spezifikation der Systemteile sollten isolierbar und wiederverwendbar sein. Diese Überlegung führt zu einer graphartigen Struktur der Spezifikation, ähnlich der Klassenhierarchie eines objektorientierten Entwurfs. Der Ansatz der Objektorientierung ist zwar heutzutage weitverbreitet, aber nach wie vor nicht vollständig erforscht und somit in der wissenschaftlichen Diskussion (siehe [BS02] und [JH02]).

Die Erweiterung der Objektorientierung zur aspektorientierten Programmierung ist ein Zeichen für die Weiterentwicklung des Ansatzes. Dabei werden Teilaspekte, welche sich nicht sauber in die Klassenhierarchie integrieren lassen, in Aspekte gekapselt und unterstützt durch Werkzeuge mit den Klassen verwoben. Die zentrale Erkenntnis aus diesen Arbeiten ist die Notwendigkeit, einzelne Aspekte eines Systems frei kombinieren zu können. Eine einfache Vererbungshierarchie ist dafür nicht ausreichend.

## 3.5 Zusammenfassung

Durch die Spezifikation von Systemen werden im Wesentlichen drei Zielsetzungen verfolgt:

a) Klares Verständnis des Problems und seiner Lösung

b) Verifikation des Systems

c) Erfassung nicht-funktionaler Aspekte

Durch die Zielsetzung der automatischen Generierung von Systemen werden diese Ziele ergänzt um die Forderung nach Vollständigkeit. Die Spezifikation muss alle Aspekte des Systems abdecken, damit eine Realisierung automatisch erzeugt werden kann.

In diesem Bereich existieren eine Reihe von Ansätzen, die einerseits verschiedene Blickwinkel auf ein System ermöglichen, andererseits aber auch eine Reihe von Ähnlichkeiten aufweisen. So werden zum Beispiel in fast allen Spezifikationssprachen Ereignisse und Zustandsübergänge beschrieben.

Allerdings existiert keine Spezifikationssprache, welche speziell für die Beschreibung des Managements von Systemen und dessen Generierung entworfen wurde. Die Anforderungen an eine derartige Spezifikationssprache sind gegenüber der Spezifikation von Anwendungen komplexer, da die Spezifikation nicht nur die Aspekte der Ausführung eines Systems beinhaltet, sondern auch die Beschreibung von Sprachkonzepten und potenziellen Ressourcen, sowie der Transformation zwischen diesen.

Dies führt zur Einbeziehung von Techniken des Übersetzerbaus in die Spezifikation. Diese müssen allerdings so erweitert werden, dass ein Umgang mit dynamischen Veränderungen möglich ist.

# 4 Komponentenbasierte Systeme

Ein effizientes Management dynamischer, verteilter Systeme erfordert zunächst eine Definition des Begriffs „System“ und eine Strukturierung innerhalb der Systeme in Einheiten, die durch das Management verwaltet werden. Die Identifikation der Subjekte des Managements sind die Grundlage für den generativen Ansatz zur Konstruktion von Managementsystemen.

In diesem Kapitel werden daher zunächst der Systembegriff präzisiert und Komponenten als Subjekte des Managements eingeführt. In den anschließenden zwei Abschnitten werden die Grundlagen für die Spezifikation von Komponenten beschrieben. Die Zuordnung von Komponenten zu Abstraktionsebenen ermöglicht eine strukturierte Sichtweise auf Systeme und somit eine übersichtliche Vorgehensweise bei der Spezifikation von Systemen. Diese Zuordnung wird im vierten Abschnitt dieses Kapitels definiert. Der letzte Abschnitt thematisiert die Verwendung der beschriebenen Konzepte für die Erarbeitung von Problemlösungsverfahren, welche die Semantik eines Systems, d. h. die auszuführenden Berechnungen beschreiben.

## 4.1 Systeme

Im Allgemeinen versteht man unter einem System ein geordnetes Ganzes oder eine gegliederte Vereinigung der Teile (vgl. [Ber72]). In der Informatik wird der Systembegriff als Zusammenfassung mehrerer Komponenten zu einer Einheit verwendet (vgl. [ECS93]). Damit ist für ein System *innen* und *außen*, seine *Umwelt*, definiert. Ein System verarbeitet bzw. löst im Allgemeinen eine wohldefinierte Menge von Problemstellungen. Die zu lösenden Aufgaben, und somit die Systeme selbst, sind meist komplex und vernetzt. Die Komplexität von Systemen ergibt sich aus der großen Zahl an Elementen und der Heterogenität der Elemente, sowie wechselnden Umwelteinflüssen und sich verändernden Anforderungen.

Systeme lassen sich durch die folgenden Eigenschaften charakterisieren:

1. **Sichtbares Verhalten**
   Bei der Betrachtung eines Systems als Black-Box ist ausschließlich das Ver-

halten des Systems von Interesse, welches außerhalb des Systems, also in seiner Umwelt beobachtbar ist.

2. **Innere Struktur**
   Geht man von der Black-Box-Sicht auf ein System zu einer Glass-Box-Sicht über, so wird die innere Struktur des Systems sichtbar. Die innere Struktur eines Systems beschreibt die Vernetzung der Komponenten des Systems untereinander und die Wechselwirkungen zwischen den Komponenten.

3. **Eigenschaften**
   Die Eigenschaften eines Systems beschreiben abstrakt die Information, die über dieses verfügbar ist. Eigenschaften sind quasi Hüllen für die Zustände des Systems. Durch die formale Beschreibung der Eigenschaften eines Systems kann das System beschrieben werden. Die Eigenschaften eines Systems können mit unterschiedlichem Detaillierungsgrad beschrieben werden.

Ein System, dessen Eigenschaften sich mit der Zeit, verändern wird als *dynamisches System* bezeichnet. Ein dynamisches System besitzt *Zustände*, die ein System zu *einem* Zeitpunkt beschreiben. Ein Zustand eines Systems ist eine Belegung der Eigenschaften des Systems. Insbesondere ist auch die innere Struktur von dynamischen Systemen ein Zustand, da diese Struktur in der Zeit veränderlich ist. Einerseits entstehen im Ablauf eines Systems neue Komponenten und bestehende Komponenten werden aufgelöst, anderseits sind die Abhängigkeiten der Komponenten untereinander einer Veränderung unterworfen.

Die obige Definition eines Systems verwendet den Begriff der *Komponente*, der im Folgenden geklärt werden soll. Der Begriff Komponente ist in vielen Bereichen der Informatik, insbesondere im Umfeld des Software Engineering mit einer speziellen Bedeutung belegt. Der UML-Standard [OMG01a] zum Beispiel definiert eine Komponente als verteilbare physikalische Einheit inklusive Quellcode, Objektcode und ausführbarem Code. Andere Definitionen stützen sich auf den Begriff der Wiederverwertbarkeit.

Danach ist ein Stück Software dann eine Komponente, wenn es wiederverwertbar ist (vgl. [CE00]). All diese Definitionen sind im hier betrachteten Zusammenhang nicht allgemein genug. Daher wird eine Komponente in dieser Arbeit, in Umkehrung der Definition des Systems, als ein identifizierbarer Teil einer größeren Einheit definiert. Ein Komponente stellt eine spezielle Funktionalität bereit. Analog zu Systemen kann auch für Komponenten *innen* und *außen* definiert werden (vgl. [SEL+96]):

**Definition 4.1 (Komponente)**
*Eine Komponente ist eine Einheit für die **innen** und **außen** definiert sind und die ihrerseits wiederum aus Komponenten zusammengesetzt sein kann.*

Komponenten können durch ihre Eigenschaften beschrieben werden. Die Eigenschaften einer Komponente charakterisieren die Komponente. Die Belegung der Eigenschaften zu einem Zeitpunkt einer Komponente wird als Zustand der Komponente bezeichnet. Dieser Zustand einer Komponente beinhaltet auch die Verbindung zwischen Komponenten.

Die Eigenschaften eines Systems werden beim Übergang von der Black-Box-Sicht zur Glass-Box-Sicht auf ein System auf die Eigenschaften der Komponenten des Systems abgebildet. Mit der oben beschriebenen Veränderung der Komponentenmenge mit der Zeit in einem dynamischen System verändern sich somit auch die Eigenschaften des Systems. Damit wird auch der Zustand eines Systems aus den Zuständen seiner Komponenten gebildet.

Die innere Struktur eines Systems ergibt sich aus den Beziehungen der Komponenten untereinander. Diese Strukturen können als Relationen der Komponenten modelliert werden. Die Relationen zwischen den Komponenten sind Eigenschaften der Komponenten und deren Belegung Teil des Zustands der Komponenten und somit des Systems.

Die Eigenschaften einer Komponente und somit die Zustände können in einen funktionalen Anteil und in einen Managementanteil unterschieden werden. Der funktionale Anteil beschreibt die Rechen- bzw. die Speicherfähigkeit, die eine Komponente auszeichnet. Der Managementanteil am Zustand einer Komponente beschreibt, wie die Rechen- bzw. Speicherfähigkeit der Komponente erlangt werden. Mit anderen Worten, der funktionale Zustand repräsentiert das „Was“ einer Komponente, der Managmentzustand das „Wie“. Die Trennung der Eigenschaften und somit der Zustände in einen funktionalen Anteil und in einen Managementanteil ermöglicht es, die Funktionalität von der Realisierung einer Komponente zu entkoppeln.

**Beispiel 4.1**
Beispielsweise kann eine Komponente, die eine unabhängige Berechnung ausführt in einem verteilten System auf unterschiedliche Weise realisiert werden. Entweder wird die Berechnung als eigene Aktivität auf einem bisher ungenutzten Knoten ausgeführt oder aber als Aktivität auf einem bereits belegten Knoten, wobei die-

se Alternative nur dann Vorteile im Hinblick auf die Optimierung der Laufzeit bringt, falls der Knoten entweder über mehrere Recheneinheiten, d. h. Prozessoren verfügt, oder aber die anderen Aktivitäten auf diesem Knoten angehalten wurden, etwa aufgrund von blockierenden Ein-/Ausgabe-Operationen. Sollte dies nicht der Fall sein, so kann die Berechnung auch sequentiell ausgeführt werden. Ebenso muss berücksichtigt werden, dass die Ausführung durch eine eigenständige Aktivität einen gewissen Overhead erzeugt, der in einem sinnvollen Verhältnis zum Gewinn stehen muss.

Die Veränderung, die dynamische Systeme charakterisiert und die für deren Management von Bedeutung ist, beruht also auf der Veränderung der Komponenten:

1. **Zustandsänderung der Komponenten**
   Der Zustand von Komponenten verändert sich. Hierbei ist der Managementzustand der Komponenten für die Verwaltung des Systems von besonderem Interesse.

2. **Veränderung der Menge der Komponenten**
   Im Ablauf eines Systems entstehen neue Komponenten und bestehende Komponenten werden aufgelöst. Diese Vorgänge müssen vom Management des Systems überwacht bzw. initiiert und durchgeführt werden.

3. **Veränderung der Strukturen**
   Die Belegung der Relationen über die Komponenten verändert sich. Beziehungen zwischen Komponenten können gelöst werden und neue Bindungen können entstehen.

Da die Ordnung der Komponenten durch die Systemstruktur als ein Teil des Zustandes der Komponenten betrachtet werden kann, lässt sich die letztere Art der Veränderung auf Erstere abbilden.

## 4.2 Spezifikation von Komponenten

Für die abstrakte Beschreibung des Managements ist es notwendig, die Komponenten eines Systems sowie deren Eigenschaften und Zustände statisch zu beschreiben. Die statische Beschreibung eines Systems wird aus den statischen Beschreibungen der potenziellen Komponenten gebildet, d. h. der Komponenten, die im laufenden System auftreten können. Je nach dem Zustand des System in der Laufzeit können aber einzelne Komponenten nicht vorhanden sein. Andererseits kann ein System aber keine Komponenten enthalten, für die keine statische Beschreibung existiert. Eine Spezifikation ist in diesem Sinne vollständig.

### 4.2.1 Metatypen

Die statische Beschreibung einer Komponente enthält die Definition der Managementeigenschaften der Komponente ($\mathcal{E}$), sowie deren potenzielle Zustände, im Folgenden als Domäne der Eigenschaften ($dom(E), E \in \mathcal{E}$) bezeichnet. Für eine Teilmenge der Eigenschaften $\mathcal{E}' \subseteq \mathcal{E}$ bezeichnet $dom(\mathcal{E}')$ das Kreuzprodukt $dom(E_1) \times \ldots dom(E_n)$, $1 \leq n \leq |\mathcal{E}'|$. Die Menge aller Domänen der Eigenschaften in $\mathcal{E}' \subseteq \mathcal{E}$ wird mit $\widehat{dom}(\mathcal{E}')$ bezeichnet.

Diese statische Beschreibung der Komponenten wird als *Metatyp*[1] bezeichnet. Jede Komponente $C$ in einem System in Ausführung ist von genau einem Metatypen $M$ abgeleitet, in Zeichen $M \mapsto C$.

Die Metatypen eines Systems beschreiben statisch die Eigenschaften und durch die zugeordneten Domänen die potenziellen Zustände der Komponenten. Die Menge aller Metatypen wird als $\mathcal{M}$ bezeichnet.

Die Eigenschaften der Metatypen implizieren eine Struktur auf der Metatypen. Ein Metatyp $M$ ist ein Untertyp von $M'$ genau dann, wenn die Menge der Eigenschaften von $M$ eine Teilmenge der Eigenschaften von $M'$ ist:

**Definition 4.2 (Untertyp)**
*Sei $\mathcal{E}_M$ die Menge der Eigenschaften eines Metatypen $M$ und $\mathcal{E}_{M'}$ die Menge der Eigenschaften eines Metatypen $M'$. $M$ heißt Untertyp von $M'$, in Zeichen $M \leftarrow M'$, wenn $\mathcal{E}_M \supseteq \mathcal{E}_{M'}$.*

Für die Spezifikation der Metatypen kann diese Beziehung in umgekehrter Richtung zur Konstruktion einer Vererbungshierarchie verwendet werden. Metatypen mit gemeinsamen Eigenschaften werden dann von einem gemeinsamen abstrakten Metatypen abgeleitet, welcher eben diese gemeinsamen Eigenschaften definiert.

Jeder Metatyp $M$ kann daher eine Menge von Obertypen $\mathcal{O}_M$ besitzen, mit der Eigenschaft, dass $M \leftarrow O$, $\forall O \in \mathcal{O}_M$. Diese Hierarchie der Metatypen bildet einen gerichteten, azyklischen Graphen, bei welchem die Menge der Eigenschaften entlang der Kanten zunimmt.

Die Metatypen eines Systems, für welche gilt, dass sie mindestens einen Untertypen besitzen, werden als abstrakte Metatypen bezeichnet.

[1] Eine formale Definition des Begriffs Metatyp findet sich in Definition 6.4 auf Seite 123.

**Definition 4.3 (Abstrakte und nutzbare Metatypen)**
*Ein Metatyp $A \in \mathcal{M}$ heißt abstrakt, genau dann, wenn gilt $\exists M \in \mathcal{M} : M \leftarrow A$. Andernfalls heißt $A$ konkreter oder nutzbarer Metatyp.*

*Die Menge aller nutzbaren Metatypen wird als $\mathcal{U} \subseteq \mathcal{M}$ bezeichnet.*

Die nutzbaren Metatypen können als vollständig in dem Sinn betrachtet werden, dass es keinen Metatypen gibt, der die Menge der Eigenschaften des Metatypen vergrößert. Die Komponenten eines Systems werden ausschließlich von den nutzbaren Metatypen abgeleitet. Diese Festlegung stellt sicher, dass die Eigenschaften einer Komponente hinreichend genau durch den Metatypen beschrieben werden.

**Beispiel 4.2**
Gegeben sei ein System, welches unter anderem die Komponenten der Arten *Objekt*, *Akteur* und *Server* enthält. Ein Objekt stellt Speicherfähigkeit und Zugriffsfunktionen auf den Speicher zur Verfügung. Ein Akteur ist eine Komponente, die Rechenfähigkeit besitzt und somit aktiv die Berechnung des Systems vorantreibt. Ein Server sei eine Komponente, die einerseits Daten verwaltet, andererseits aber auch aktiv ist. Ein Server kombiniert somit die Eigenschaften der Rechenfähigkeit und der Speicherfähigkeit. Die entsprechende Hierarchie der Metatypen ist in Abb. 4.1 dargestellt.

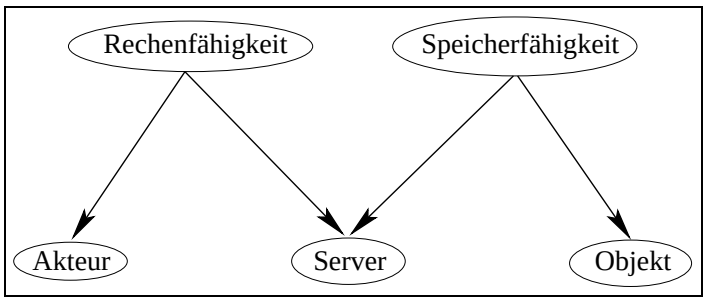



Abbildung 4.1: Hierarchie von Metatypen.

Die Menge der nutzbaren Metatypen in diesem Beispiel besteht aus den Metatypen *Akteur*, *Server* und *Objekt*. Die Komponenten eines Systems, das unter Verwendung dieser Spezifikation entsteht, ist aus Komponenten dieser Typen aufgebaut. Die abstrakten Metatypen *Rechenfähigkeit* und *Speicherfähigkeit* kapseln die Eigenschaften, die mehreren nutzbaren Metatypen gemeinsam sind.

Da Komponenten gemäß obiger Festlegung nur aus den nutzbaren Metatypen abgeleitet werden können, gilt folgendes Lemma:

**Lemma 4.1 (Nutzbare Metatypen)**
*Sei $C$ eine Komponente, dann gilt:*

$$M \mapsto C \Rightarrow M \in \mathcal{U}$$

Der Zusammenhang zwischen Metatypen und Komponenten im Allgemeinen wird durch die Relation $\rightsquigarrow$ beschrieben:

**Definition 4.4 (Metatypen und Komponenten)**
*Sei $\rightsquigarrow$ eine Relation zwischen Metatypen und Komponenten. Sei $C$ eine Komponente und $M$ ein Metatyp. Dann gilt:*

$$M \rightsquigarrow C \Rightarrow M \mapsto C \vee \exists M' : M' \leftarrow M \wedge M' \rightsquigarrow C$$

Durch die Relation $\rightsquigarrow$ wird jedem Metatypen $M$ die Menge aller Komponenten zugeordnet, die von diesem Metatypen oder einem seiner Untertypen abgeleitet wurde.

### 4.2.2 Komponentenzustände und Komponentenklassen

Ein System entsteht aus den Metatypen, in dem Instanzen der Metatypen gebildet werden, welche als Komponenten bezeichnet werden. Alle Komponenten, die von einem Metatypen abgeleitet wurden, zeichnen sich durch identische Mengen von Eigenschaften aus, wobei jedoch die Belegung der Eigenschaften von Komponente zu Komponente unterschiedlich sein kann. Komponenten befinden sich dann in unterschiedlichen Zuständen.

**Definition 4.5 (Zustand)**
*Sei $C$ eine Komponente des Metatypen $M$. Dann ist der Zustand der Komponente $C$ zum Zeitpunkt $t$ die Belegung der Eigenschaften $\mathcal{E}_M$. Der Zustand einer einzelnen Eigenschaft wird durch die Abbildung $\tilde{Z} : \mathcal{C} \times \mathcal{E}_M \times T \rightarrow \widehat{dom}(\mathcal{E}_M)$ repräsentiert.*

*$Z : \mathcal{C} \times \mathcal{E}_M{}^* \times T \rightarrow \widehat{dom}(\mathcal{E})^*$ ist dann die Erweiterung von $\tilde{Z}$ auf Tupel von Eigenschaften.*

Eigenschaften können zu verschiedenen Zeitpunkten und evtl. auch erneut belegt werden. Spätestens mit der Entstehung der Komponente müssen alle Eigenschaften erstmalig belegt sein. Diese Belegung ist der *Startzustand* der Komponente. Eigenschaften von Komponenten können aber bereits vor der Entstehung der Komponente belegt sein, d. h. der Zustand dieser Eigenschaften ist bereits bekannt, obwohl die Komponente noch nicht existiert. Eine simple Möglichkeit für diese Tatsache ist die konstante Belegung von Eigenschaften. Im Beispiel 4.2 auf Seite 56 kann für die Metatypen Akteur und Server die Eigenschaft *Rechenfähigkeit* konstant mit *wahr* belegt werden, für die Metatypen Server und Objekt die Eigenschaft *Speicherfähigkeit*.

Auf der Grundlage der Zustände von Komponenten kann eine Äquivalenz auf Komponenten beschrieben werden. Zwei Komponenten, welche von einem Metatypen abgeleitet wurden, sind äquivalent, wenn ihre Zustände identisch sind. In vielen Fällen ist dieser absolute Äquivalenzbegriff allerdings zu stark. Daher können Zustände als äquivalent bzgl. einer Teilmenge der Eigenschaften aufgefasst werden, falls die Belegung dieser Eigenschaften identisch ist. Diese Zusammenhänge sind in der folgenden Definition zusammengefasst:

**Definition 4.6 (Äquivalente Komponenten)**
*Seien $C_1$ und $C_2$ zwei Komponenten, welche aus einem Metatypen $M$ abgeleitet wurden. $C_1$ ist äquivalent zu $C_2$ zum Zeitpunkt $t$, in Zeichen $C_1 \equiv^t C_2$, wenn $Z(C_1, \mathcal{E}_M, t) = Z(C_2, \mathcal{E}_M, t)$.*

*Sei $\mathcal{K} \subseteq \mathcal{E}_M$. Die Komponenten $C_1$ und $C_2$ sind $\mathcal{K}$-äquivalent oder äquivalent bzgl. $\mathcal{K}$, zum Zeitpunkt $t$, in Zeichen $\equiv^t_{\mathcal{K}}$, genau dann wenn $Z(C_1, \mathcal{K}, t) = Z(C_2, \mathcal{K}, t)$.*

Auf der Grundlage der Definition äquivalenter Komponenten kann eine Klassifizierung von Komponenten durchgeführt werden. Die Klassifizierung orientiert sich dabei an den Eigenschaften, welche für die Komponenten der Klassen zu jedem Zeitpunkt identisch belegt sind:

**Definition 4.7 (Klasseneigenschaften)**
*Sei $\mathcal{K}_e$ eine Teilmenge der Eigenschaften eines Metatyps $\mathcal{M}$, also $\mathcal{K}_e \subseteq \mathcal{E}_M$ und $\mathcal{K}_1, \ldots, \mathcal{K}_n$ eine Zerlegung der Komponenten des Metatyps $M$ in disjunkte Teilmengen, also $\mathcal{K}_i \cap \mathcal{K}_j = \emptyset$ mit $1 \leq i, j \leq n$ und $i \neq j$. Die Menge $\mathcal{K}_e$ heißt Klasseneigenschaften von $M$, wenn zu jedem Zeitpunkt $t$ gilt:*

$$\forall K_j \ : \ 1 \leq j \leq n \ : \ \forall C, C' \in \mathcal{K}_j \ : \ C \equiv^t_{\mathcal{K}_e} C'.$$

*Die disjunkten Teilmengen $\mathcal{K}_j$, $1 \leq j \leq n$ werden dann als Klassen bezeichnet. Die Menge aller Klassen eines Metatypen $M$ wird als $\mathcal{K}_M = \{\mathcal{K}_i | 1 \leq i \leq n\}$ bezeichnet.*

Klasseneigenschaften sind also zu jedem Zeitpunkt für alle Komponenten der Klasse identisch belegt. Die Einteilung der Komponenten in Klassen ist willkürlich und von der Menge der betrachteten Komponenten abhängig. Allerdings ermöglicht diese Definition bei einer Top-Down-Konstruktion die Definition von Komponentenklassen ausgehend von der Spezifikation der Metatypen. Für jeden Metatypen kann eine Teilmenge der Eigenschaften als Klasseneigenschaften festgelegt werden, für die dann obige Festlegung durchgesetzt werden muss. Dies ist Aufgabe des Managements. Die Klasseneigenschaften erhalten damit die Bedeutung eines Informationsflusses zwischen den Komponenten einer Klasse. Dabei ist insbesondere von zentraler Bedeutung, dass die Klasseneigenschaften auch einen Informationsfluss von der Vergangenheit in die Gegenwart darstellen. Eine Komponente kann den Zustand bzgl. der Klasseneigenschaften während ihrer Lebenszeit beeinflussen. Dieser Zustand bleibt auch nach der Auflösung der Komponente erhalten. Eine neu entstehende Komponente wird daher in ihrem Startzustand diese Belegung vorfinden.

Die Metatypen erhalten damit eine duale Bedeutung. Einerseits beschreiben Metatypen die Komponenten eines Systems. Mit der Spezifikation der Metatypen wird aber auch die Klasseneinteilung der Komponenten abstrakt spezifiziert.

Damit kann der Konstrukstruktionsprozess für ein System als dreistufiges Verfahren dargestellt werden. Ausgehend von den Metatypen werden Komponentenklassen gebildet, die wiederum als Basis für die Konstruktion von Komponenten dienen. Ein ähnliches Konzept findet sich auch in objektorientierten Sprachen.

Dort existiert die Möglichkeit, Methoden und Attribute auf Klassenebene zu definieren (vgl. z. B. [Str97]). Allerdings existiert in klassischen objektorientierten Sprachen nur ein Metatyp, nämlich der Metatyp Klasse.

**Beispiel 4.3**
Im Beispiel 4.2 auf Seite 56 ist in Abb. 4.1 eine Hierarchie von Metatypen dargestellt. Ausgehend von diesen Metatypen können nun eine Reihe von Komponentenklassen abgeleitet werden. In Abb. 4.2 sind beispielhaft die Klassen *System* als Ableitung des Metatypen *Akteur*, *WWW-Server* und *Datei-Server* abgeleitet vom Metatypen *Server*, sowie *Datei* als Klasse des Metatypen *Objekt* dargestellt. Die Zustände der Klasseneigenschaften aller von diesen Klassen abgeleiteten Komponenten sind stets identisch. Für die Komponenten der Klasse WWW-Server kann beispielsweise eine Klasseneigenschaft die durchschnittliche Last aller WWW-Server Komponenten repräsentieren. Das Management kann diese Information verwenden, um bei Überschreitung eines Grenzwertes eine neue Komponente dieser Klasse zu erstellen, womit die durchschnittliche Last aller WWW-Server durch Verteilung der Anfragen sinkt.

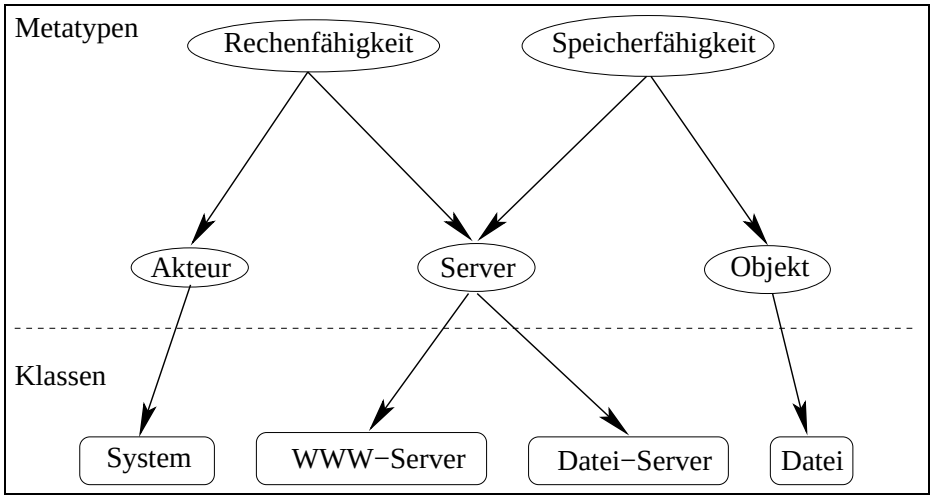



Abbildung 4.2: Metatypen und abgeleitete Klassen.

### 4.2.3 Sorten zur Repräsentation von Daten

Die potenziellen Belegungen der Eigenschaften einer Komponente werden durch die Domänen der Eigenschaften definiert. Die Domänen der Eigenschaften sind die Datensorten des Managements. Die Datensorten sind die Mengen der potenziellen Werte, die Eigenschaften annehmen können. Die Datensorten einer Spezifikation, $\tilde{\mathcal{D}}$, können wie folgt rekursiv definiert werden:

1. **einfache Datensorten**
   Die einfachen Datensorten werden als vorgegeben angenommen. Ihre Realisierung muss durch die Basis des Managements sichergestellt werden. Diese Basis wird im Allgemeinen von der Hardware des Systems gebildet, so dass die einfachen Datensorten den Sorten der ausführenden Maschinen entsprechen. Beispiele für einfache Datensorten sind Integer, Gleitpunktzahlen und die Sorte der boolschen Werte.

2. **Zeit**
   Sei $T$ eine Datensorte zur Repräsentation der Zeit. Dann ist $T \in \tilde{\mathcal{D}}$

3. **Metatypen**
   Die Metatypen bilden eine Menge von Datensorten. Durch die Metatypen einer Spezifikation stehen einerseits Datensorten für die Repräsentation von Komponenten, aber auch Datensorten für die Repräsentation von Komponentenklassen zur Verfügung. Hier wird die Dualität von Metatypen deutlich. Eine Klasse von Komponenten kann somit als Einheit in der Spezifikation repräsentiert werden.

4. **Tupel**
   Aus den Datensorten einer Spezifikation kann durch Tupelbildung eine neue Datensorte gebildet werden. Seien also $d_1, d_2, \ldots, d_n \in \tilde{\mathcal{D}}$ Datensorten, dann ist auch $\langle d_1 \times d_2 \times \ldots \times d_n \rangle$ eine Datensorte.

5. **Abstrakte Datentypen**
   Ein abstrakter Datentyp ist ein Tupel $A = \langle T, F \rangle$, wobei $T$ eine Menge von Sorten und $F$ eine Menge von Funktionen mit einer Abbildung $f_{sig} : F \rightarrow (T^* \cup \tilde{\mathcal{D}}^* \times T \cup \tilde{\mathcal{D}})$ ist. Die Funktion $f_{sig}$ bildet jede Funktion auf ihre Signatur ab.

   Ein *Modell* für einen abstrakten Datentypen $A$ ist eine Zuordnung von Datentypen zur Menge der Sorten, also $M_A : T \rightarrow \tilde{\mathcal{D}}$. Jedes Tupel aus abstraktem Datentyp $A$ und Modell $M_A$ ist eine Datensorte.

Mengen sind ein abstrakter Datentyp, dessen Existenz implizit vorrausgesetzt wird. Der abstrakte Datentyp für Mengen sei $set = \langle T, F \rangle$, mit $T = \{t\}$ und den üblichen Operationen auf Mengen:

$$F = \{ \begin{array}{l} \in: t \times set(t) \to I\!B, \\ \cup : set(t) \times set(t) \to set(t), \\ \cap : set(t) \times set(t) \to set(t), \\ insert : t \times set(t) \to set(t), \\ remove : t \times set(t) \to set(t), \\ isempty : set(t) \to I\!B \} \end{array}$$

Die vollständigen Datensorten einer Spezifikation ergeben sich aus den oben definierten Datensorten $\tilde{\mathcal{D}}$, indem die Wertebereiche der Datensorten jeweils um ein undefiniertes Element $\perp$ erweitert werden. Mittels der Erweiterung um das Element $\perp$ werden partielle Funktionen vermieden. Diese erweiterten Datensorten werden als $\mathcal{D}$ bezeichnet. Formal ergibt sich damit für die vollständigen Datensorten folgende Definition:

**Definition 4.8 (vollständige Datensorten)**
*Die Menge der vollständigen Datensorten $\mathcal{D}$ ist definiert als:*

*a) Sei $\tilde{S}$ eine einfache Datensorte mit*

$$S = \tilde{S} \cup \{\perp\} \in \mathcal{D}$$

*b) Sei $T$ eine Datensorte zur Repräsentation der Zeit, dann ist*

$$T \cup \{\perp\} \in \mathcal{D}$$

*c) Sei $M$ ein Metatyp, dann ist*

$$\{C | M \rightsquigarrow C\} \cup \{\perp\} \in \mathcal{D}$$

*d) Seien $K_1, \ldots, K_n$ die Klassen der Komponenten von $M$ gemäß der Definition 4.7 auf Seite 59. Dann ist*

$$\{K_i | 1 \leq i \leq n\} \cup \{\perp\} \in \mathcal{D}$$

*e) Seien $D_1, \ldots, D_n \in \mathcal{D}$, dann ist das Kreuzprodukt der $D_i$, $1 \leq i \leq n$ als Tupelsorte*

$$V = \langle D_1 \times D_2 \times \ldots \times D_n \rangle \in \mathcal{D}$$

*f) Sei $A = \langle T, F \rangle$ ein abstrakter Datentyp und $M_A : S \to \mathcal{D}$ ein Modell für $A$, dann ist*

$$\langle A, M_A \rangle \cup \{\perp\} \in \mathcal{D}$$

Im folgenden wird der Begriff Datensorte, wenn nicht anders angegeben für eine vollständige Datensorte verwendet.

Mittels dieser Datensorten lassen sich die Domänen der Eigenschaften näher spezifizieren. Eine Domäne $dom(E)$ einer Eigenschaft $E$ ist ein Element der Menge der Datensorten, also $dom(E) \in \mathcal{D}$.

## 4.3 Systemstrukturen

In Abschnitt 4.1 auf Seite 51 wurde bereits kurz auf die Strukturen von Systemen eingegangen. Die Struktur eines Systems ergibt sich durch Beziehungen zwischen den Komponenten eines Systems. Diese Strukturen sind Gegenstand der Betrachtungen im folgenden Abschnitt.

Die Struktur eines Systems zu einem Zeitpunkt kann als gerichteter Graph $\mathfrak{S} = \langle \mathcal{C}, E \rangle$, mit $E \subseteq \mathcal{C} \times \mathcal{C}$, dessen Knoten durch die Komponenten des Systems gebildet werden, dargestellt werden. Die Forderung, dass dieser Graph gerichtet ist, stellt keine Einschränkung der Systemstrukturen dar, da für den Graphen keine Zyklenfreiheit gefordert wird. Anhand der Semantik der Strukturen kann dieser Graph in Subgraphen $\mathfrak{S}_i = \langle \mathcal{C}_i, E_i \rangle$, mit $\mathcal{C}_i \subseteq \mathcal{C}$ und $E_i \subseteq E$, zerlegt werden, wobei jeder Subgraph eine Teilstruktur innerhalb des Systems beschreibt. Die Struktur eines Systems ergibt sich dann als Vereinigung der Subgraphen, also $\mathfrak{S} = \bigcup \mathfrak{S}_i$.

**Beispiel 4.4**
Beispiele für derartige Subgraphen sind die Aufrufbeziehung zwischen Komponenten, d. h. welche Komponente die Entstehung einer anderen Komponente ausgelöst hat, oder die Nutzungsbeziehungen zwischen Komponenten, d. h. welche anderen Komponenten die Dienste einer Komponente nutzen. Eine derartige Situation ist in Abb. 4.3 auf der nächsten Seite dargestellt.

Für die Spezifikation von Systemen ist die Beschreibung der potenziellen Beziehungen der Komponenten notwendig. Diese Beschreibung der Strukturgraphen erfolgt auf der Grundlage der Metatypen.

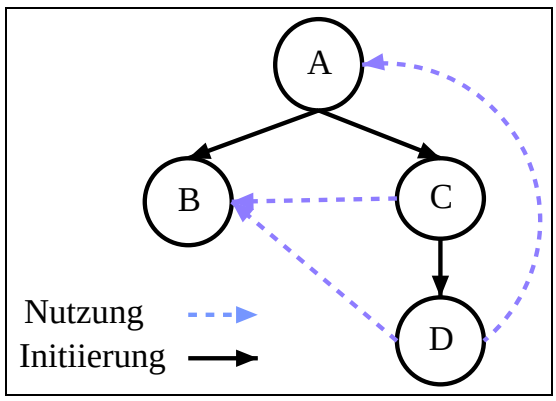



Abbildung 4.3: Strukturen in einem System.

**Definition 4.9 (Strukturdefinition)**
*Seien $M$ und $M'$ Metatypen. Eine Strukturdefinition, $D = (M \times M', E)$ definiert einen Strukturgraphen $\mathfrak{S} = \langle \mathcal{C}_M \cup \mathcal{C}_{M'}, E \rangle$, wobei $\mathcal{C}_M$ und $\mathcal{C}_{M'}$ die Menge der von $M$ bzw. $M'$ abgeleiteten Komponenten sind.*

Jede Strukturdefinition beschreibt eine Teilstruktur des Strukturgraphen eines Systems. Da sich die Gesamtheit der Struktur eines Systems aus der Vereinigung der Teilstrukturen ergibt, wird die Gesamtheit der Definition der Strukturen des Systems aus der Zusammenfassung der Strukturdefinitionen gebildet.

Die Strukturen eines Systems können bei hinreichender Anzahl der Komponenten bzw. der Metatypen sehr komplex werden. Eine Vereinfachung in der Spezifikation der Strukturen, die Beschränkung auf baumartige Strukturen[2], ist ohne Informationsverlust möglich.

Diese Vereinfachung erhöht einerseits die Übersichtlichkeit der Spezifikation, andererseits kann dieses Wissen für die Analyse der Spezifikation, im Generierungsprozess des Managements und bei der Ausführung des Systems gewinnbringend eingesetzt werden, da für baumartige Strukturen einfachere und effizientere Algorithmen zur Verfügung stehen als für beliebige Graphen.

[2] Es handelt sich nicht um tatsächliche Bäume, da erstens die Richtung der Kanten für Bäume üblicherweise in umgekehrter Richtung erfolgt und zweitens für die Strukturen die Forderung der Zyklenfreiheit nicht erhoben wird.

**Satz 4.2 (Zerlegung von Strukturen)**
*Jeder Strukturgraph* $\mathfrak{S} = \langle \mathcal{C}, E \rangle$ *mit* $E \subseteq \mathcal{C} \times \mathcal{C}$ *lässt sich derart in Unterstrukturen* $\mathfrak{S}_i = \langle \mathcal{C}, E_i \rangle$ $E_i \subseteq \mathcal{C} \times \mathcal{C}$ *zerlegen, dass folgendes gilt:*

*a)* $\forall C_1, C_2 \in \mathcal{C} : \ (C_1, C_2) \in E_i \ \Rightarrow \nexists C' \in \mathcal{C}$ *mit* $(C', C_2) \in E_i$

*b)* $S = \bigcup S_i$

**Beweis**

Seien $\mathfrak{S} = \langle \mathcal{C}, E \rangle$ ein Strukturgraph und $C' \in \mathcal{C}$ eine Komponente.

Dann ist $\mathcal{X} = \{X | X \in \mathcal{C} \wedge \exists (C', X) \in E\}$ die Menge aller Komponenten, welche eingehende Kanten besitzen, die von $C'$ ausgehen.

Falls $|\mathcal{X}| \leq 1$ ist, so ist die Forderung erfüllt.

Falls $|\mathcal{X}| > 1$ ist, so wird der Strukturgraph in $n = |\mathcal{X}|$ Strukturgraphen zerlegt, mit

$$\mathfrak{S}_i = \langle \mathcal{C}, E/(X_j, C') \rangle, \text{ wobei } X_j \in \mathcal{X}, j \neq i, \ 1 \leq i, j \leq n.$$

Die Strukturgraphen $\mathfrak{S}_i$ erfüllen die Forderung des Satzes und es gilt $\mathfrak{S} = \bigcup \mathfrak{S}_i$, da die Knotenmenge der Strukturgraphen $\mathfrak{S}$ und $\mathfrak{S}_i$ jeweils $\mathcal{C}$ ist und für die Kantenmenge $E = \bigcup E_i$ gilt. *q.e.d.*

Ein Strukturgraph lässt sich in Unterstrukturen zerlegen, so dass jede Komponente bzgl. einer Unterstruktur maximal *eine* ausgehende Kante enthält. Für die Spezifikation eines Systems ist es aber notwendig, die Strukturdefinitionen entsprechend zu formulieren. Es ist daher notwendig, die Strukturdefinitionen auf baumartige Strukturen zu beschränken. Dabei können zwei Fälle unterschieden werden:

1. $m : n$**-Beziehung mit festem** $m$
   In diesem Fall liegt die Anzahl der baumartigen Strukturen bei der Spezifikation fest. Für jede Struktur kann eine entsprechende Strukturdefinition, $D_i = (M \times M', E)$ mit $1 \leq i \leq m$ angegeben werden.

2. $m : n$**-Beziehung mit dynamischem** $m$
   Bei diesem Fall liegt die Anzahl der Strukturen bei der Spezifikation noch nicht fest. Daher kann keine feste Anzahl an Strukturdefinitionen festgelegt werden. Lässt man dies zu, so ist die Anzahl der Eigenschaften eines Metatyps in der Spezifikation nicht festgelegt. Dies würde die Spezifikation der Metatypen verkomplizieren.

Die Alternative ist die Verwendung einer Technik, die in der Informatik – insbesondere beim Entwurf von Betriebssystemen – eine zentrale Rolle spielt, die *Virtualisierung*. Durch Einführung einer neuen Komponentenart, also einem neuen Metatypen, kann die $m : n$-Beziehung aufgespalten werden in eine $m : 1$- und eine $1 : n$-Beziehung. Dies ist in Abb. 4.4 dargestellt. Abb. 4.4(a) zeigt einen Strukturgraphen mit $m : n$-Beziehung. In Abb. 4.4(b) ist die gleiche Situation nach der Einführung virtueller Komponenten (A', A", B', B", C' und C") dargestellt.

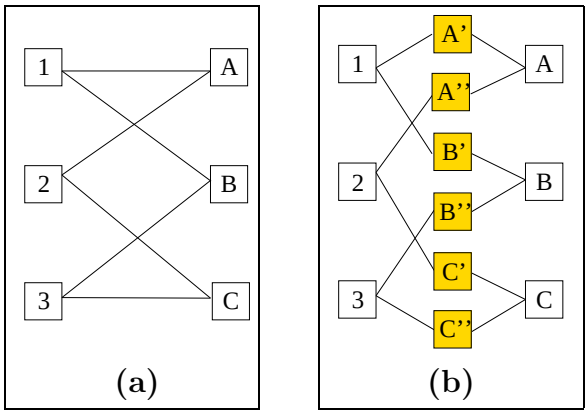



Abbildung 4.4: Virtualisierung von Komponenten.

Eine Systemdefinition lässt sich somit in Unterdefinitionen zerlegen, so dass jede Komponente bzgl. einer Struktur maximal *eine* ausgehende Kante enthält. Diese Unterstrukturen werden als *Relationen* bzw. als *Strukturrelationen* bezeichnet, die dazugehörigen Definitionen als *Relationsdefinition*. Die Spezifikation der Strukturen eines Systems erfolgt durch eine Reihe von Relationsdefinitionen.

Damit kann die Einordnung einer Relation in das System der Eigenschaften der Komponenten erklärt werden. Sei $\mathcal{R} = (M_1, M_2, E)$ eine Relation. Dann ist $E_\mathcal{R}$ eine Eigenschaft von $M_2$, also $E_\mathcal{R} \in \mathcal{E}_{M_2}$ mit der Domäne $dom(E_\mathcal{R}) = M_1$ und $E_{\mathcal{R}_\mathcal{S}}$ eine Eigenschaft von $M_1$, also $E_{\mathcal{R}_\mathcal{S}} \in \mathcal{E}_{M_1}$ mit der Domäne $dom(E_{\mathcal{R}_\mathcal{S}}) = \{M_1\}$, also einer Menge von Komponenten des Metatyps $M_1$. Für einen Metatypen $M$ wird die Menge aller durch die Strukturdefinitionen verbundenen Metatypen als $\mathcal{S}_M$ bezeichnet, d. h. falls $\mathcal{R} = (M_1, M_2, E)$ eine Relation ist, so ist $M_1 \in \mathcal{S}_{M_2}$ und $M_2 \in \mathcal{S}_{M_1}$.

Die Strukturen innerhalb eines System und somit die Strukturdefinitionen repräsentieren einerseits die Abhängigkeiten, die im System aufgrund des Problemlösungsverfahrens bestehen. Diese Strukturen beschreiben also Nutzungs- und Abhängigkeitsbeziehungen, die aus dem Quelltext eines Systems entstehen. Auf der anderen Seite wird durch die Strukturen auch eine Realisierungsentschei-

dung beschrieben. Bestimmte Zustände einer Komponente können nur durch die Verwendung anderer Komponenten realisiert werden.

**Beispiel 4.5**
Im Beispiel 4.2 auf Seite 56 wurden für die Komponenten Akteur und Server die Eigenschaft *Rechenfähigkeit* eingeführt. Diese Eigenschaft kann im realen System nur dann mit dem Zustand *wahr* belegt werden, wenn für die entsprechenden Komponenten mittels eines Threads oder eines Prozesses beispielsweise diese Eigenschaft durchgesetzt wird. Daher steht jede aktive Komponente in einer Beziehung zu einer entsprechenden Ressource.

Eine Struktur, welche der Durchsetzung von Zuständen dient, wird als *Ressourcenstruktur* bezeichnet. Die entsprechende Relationsdefinition als *Ressourcenrelation*.

## 4.4 Abstraktionsebenen

Die Darstellung eines Systems kann auf unterschiedlichen Ebenen stattfinden. Eine Unterscheidung, die statische und die dynamische Sicht, wurde im vorherigen Abschnitt bereits diskutiert. Ein System kann aber auch auf unterschiedlichen Abstraktionsstufen betrachtet werden. Die Beschreibung des Systems, sowohl dynamisch als auch statisch, unterscheidet sich auf den unterschiedlichen Abstraktionsstufen. Im Rahmen des MoDiS-Projektes wurde für die Sprache INSEL (siehe [RW96]) das in [Piz99] beschriebene $STK$-Modell entwickelt. In diesem Modell wird die Struktur eines Systems in drei Dimensionen, nämlich Raum ($S$), Zeit ($T$) und Konkretisierungsniveau ($K$), beschrieben. Das $STK$-Modell liefert einen ersten homogenen Ansatz für die Beschreibung eines Systems in allen Dimensionen. Für die Spezifikation von Systemen ist eine solche einheitliche Beschreibung unabdingbar. Die Dimensionen Zeit und Raum wurden im bisher Gesagten bereits behandelt. Die dritte Dimension, das Konkretisierungsniveau wird, im Folgenden mittels der gegebenen Mittel für die Spezifikation von Systemen beschrieben.

Klassisch unterscheidet man in einem System zwischen Anwendungskomponenten und Ressourcen. Die Verwaltung und Breitstellung von Ressourcen ist die zentrale Aufgabe von Managementsystemen. Diese Sichtweise ist allerdings verkürzend und somit für eine vollständige Spezifikation nicht passend. Wesentlich ist die Einführung von mehreren Ebenen, die eine schrittweise Überführung der Anwendung und ihrer Komponenten auf die Hardware beschreiben. Durch die Einführung mehrerer Ebenen ergibt sich für das Management die Möglichkeit, unterschiedli-

che Pfade für die Realisierung von Komponenten bzw. deren Eigenschaften und Zuständen zu wählen.

Betrachtet man diese Realisierungsaufgabe vom Standpunkt einer einzelnen Komponente aus, so wird zu ihrer Realisierung eine Menge von Ressourcen benötigt. Daher ist zunächst die Frage zu klären, was eine Ressource im Sinne der Spezifikation ist.

**Definition 4.10 (Ressourcenkomponente)**
*Eine Komponente $R$ ist eine Ressourcenkomponente, wenn es eine Komponente $C$ vom Metatyp $M$ gibt, welche zur Durchsetzung der Belegung ihrer Eigenschaften $\mathcal{E}_M$, bzw. einer Teilmenge $\mathcal{E}' \subseteq \mathcal{E}_M$ davon, $R$ irgendwann benötigt, d. h. wenn im Strukturgraphen $\mathfrak{S} = \langle \mathcal{C}, E \rangle$, mit $C, R \in \mathcal{C}$ das Tupel $(C, R)$ in $E$ liegt und es gilt $Z(C, \mathcal{E}', t) \Rightarrow (C, R) \in E$.*

*Die Menge aller Ressourcen von $C$ zum Zeitpunkt $t$ wird als $\mathcal{R}_t(C)$ bezeichnet.*

Die Ressourcen selbst wiederum benötigen eine weitere Menge an Ressourcen. Die Definition der Ressourcen stützt sich auf die Struktur des Systems und somit auf den Zustand der Komponenten ab. Der Zustand der Komponenten ist Veränderungen mit der Zeit unterworfen. Daher sind die Zuordnungen der Komponenten zu ihren Ressourcen flexibel. Diese Flexibilität ermöglicht einerseits die zeitlich befristete Bindung von Ressourcen an eine Komponente. Andererseits kann aber eine Ressource einer Komponente auch durch eine andere Ressource ausgetauscht werden, wenn sich beispielsweise entweder der Zustand der ursprünglichen Ressource verändert hat, oder andererseits sich die Anforderungen der Komponente, welche durch den Zustand der Komponente ausgedrückt werden, verändern.

In der Spezifikation eines Systems muss eine potenzielle Nutzung von Ressourcen beschreibbar sein. Da es sich bei den Ressourcenbeziehungen um Strukturen des Systems handelt, können die Beschreibungsmittel, welche zur Spezifikation der Strukturen eingesetzt werden, hier entsprechend angewandt werden. Die potenzielle Nutzung von Ressourcen kann in zweierlei Hinsicht variabel gestaltet werden:

1. **Zeitlich begrenzte Nutzung von Ressourcen**
   Die zeitlich begrenzte Nutzung von Ressourcen wird durch zwei Zeitpunkte definiert, den Beginn und das Ende der Nutzung von Ressourcen. Dieser Aspekt der Ressourcennutzung wird im Kapitel 6 auf Seite 107 eingehend untersucht.

2. **Alternative Nutzung von Ressourcen**
   Für die alternative Nutzung von Ressourcen muss in der Spezifikation des Systems festgelegt werden, welche Eigenschaften eine Ressource haben muss, um die Ausführung der entsprechenden Komponente zu ermöglichen. Als Ressource muss daher ein Metatyp festgelegt werden, dessen abgeleitete Komponenten diese Eigenschaften besitzen. Wird hierfür ein Metatyp aus der Menge der nutzbaren Metatypen gewählt, so liegen die Eigenschaften der Ressourcen vollständig fest. Wird hingegen ein abstrakter Metatyp für eine Ressource festgelegt, so sind die Eigenschaften nicht vollständig festgelegt. Das Management erhält somit einen weiteren Freiheitsgrad in der Befriedigung der Ressourcenanforderungen.

Die Ressourcen-Beziehungen der Komponenten finden ihren Abschluss nach unten durch die Hardware bzw. die Hardware-Ressourcen:

**Definition 4.11 (Hardware-Komponente)**
*Eine Komponente $H$ ist eine Hardware-Komponente, wenn es keine Komponente $R$ gibt, welche $H$ zu seiner Ausführung irgendwann benötigt, also $\mathcal{R}_t(H) = \emptyset$ für alle $t$.*

Mit anderen Worten, eine Hardware-Ressource ist eine Komponente, deren Existenz nicht von weiteren Ressourcen abhängt. Die Entscheidung, was in diesem Sinne eine Hardware-Ressource ist, ist abhängig von der Tiefe der Spezifikation der Metatypen.

**Beispiel 4.6**
Am Beispiel 4.5 auf Seite 67 wurde gezeigt, dass die Ressource Thread zur Realisierung der Rechenfähigkeit von Aktivitäten verwendet werden kann. Ein Thread ist nach Definition 4.11 Hardware-Ressource, wenn zur Durchsetzung der Zustände seiner Eigenschaften keine weiteren Komponenten benötigt werden. Wird für das Management eines System auf eine bestehende Thread-Implementierung, wie z. B. eine `pthread`-Implementierung nach [ANS96], zurückgegriffen, so kann in der Spezifikation eine Komponente *Thread* als Hardware-Ressource betrachtet werden.

Eine andere Sichtweise ergibt sich, wenn die Spezifikation eines Threads sich auf weitere Komponenten stützt, wie z. B. *Speicher*, *Prozessor* usw. Damit ist ein Thread zwar weiterhin eine Ressource für die Komponenten der Metatypen *Ak-*

*tivität* bzw.- *Server*, aber es handelt sich nicht um Hardware-Ressourcen. Dieses Schema kann mit den Komponenten des Metatyps *Speicher* fortgesetzt werden. Diese können wiederum als Hardware-Ressourcen spezifiziert werden, oder die Spezifikation enthält weitere Komponenten, die zur Realisierung von *Speicher* benötigt werden, wie z. B. *Seiten* oder *Kacheln* usw.

Die Einteilung der Komponenten in Ressourcen kann entsprechend auf Metatypen übertragen werden. Die folgende Definition legt den Begriff der „Ressource“ auf der Basis der Metatypen fest.

**Definition 4.12 (Ressource)**
*Ein Metatyp $R$ wird als Ressource des Metatypen $M$ bezeichnet, wenn es*

*a) eine Strukturdefinition $D = (M \times R, E)$ oder $D = (R \times M, E)$ gibt und*

*b) eine Menge von Eigenschaften $\mathcal{E}' \subseteq \mathcal{E}_M$ existiert, bei welcher für mindestens eine Belegung von $\mathcal{E}'$ gilt, dass es eine Kante im Strukturgraphen $\mathfrak{S}_D$ gibt, welche eine Komponente mit diesem Zustand mit einer Komponente, die aus dem Metatypen $R$ abgeleitet wurde, in Relation setzt.*

*Alle Ressourcen von $M$ werden als $\mathcal{R}_M$ bezeichnet.*

*Ein Metatyp $H$ ist eine Hardware-Ressource, $\mathcal{R}_H = \emptyset$*

*Die Menge aller Hardware-Ressourcen wird als $\mathcal{H}$ bezeichnet.*

Eine Komponente $R$ ist also dann eine Ressource für eine andere Komponente $C$, wenn $R$ für die Realisierung eines der möglichen Zustände der Komponente $C$ benötigt wird. Auf der Grundlage der Definition der Ressource ist eine Zuordnung von Metatypen zu Ebenen möglich:

**Definition 4.13 (Ressourcenebene)**
*Jedem Metatyp $M$ wird eine Ressourcenebene $r(M) \in I\!N$ zugeordnet:*

$$r(M) = \begin{cases} 0 & : \quad M \in \mathcal{H} \\ max(r(R_1)+1, \ldots, r(R_n)+1), \ mit\ \mathcal{R}_M = \bigcup_{i=1}^{n} R_i & : \quad sonst \end{cases}$$

Die Ressourcenebene beschreibt den Abstand eines Metatypen, bzw. der davon abgeleiteten Komponenten zur Hardware, wobei die Definition des Begriffs Hardware von der Spezifikation des Systems abhängig ist. Metatypen bzw. Komponenten können neben den Relationen zu ihren Ressourcen auch mit Komponenten anderer Ebenen in Beziehung stehen. Um eine Schichtung der Metatypen zu erhalten, muss die Definition erweitert werden:

**Definition 4.14 (Abstraktionsebene)**
*Jedem Metatyp $M$ wird eine Abstraktionsebene $a(M) \in I\!N$ zugeordnet:*

$$a(M) = max(r(M), a(M_1), \ldots, a(M_n)),\ mit\ \mathcal{S}_M/\mathcal{R}_M = \bigcup_{i=1}^{n} M_i$$

Die Zuordnung von Metatypen zu Abstraktionsebenen ordnet die Metatypen in einer Hierarchie an. Die unterste Ebene diese Hierarchie wird durch die Hardware gebildet. Jeder Metatyp steht nur mit Komponenten einer niedrigeren Abstraktionsebene (seinen Ressourcen) und Komponenten der selben Abstraktionsebene in Verbindung. Die oberste Ebene in dieser Hierarchie wird durch die Komponenten gebildet, deren Realisierung die Aufgabe des Managements ist. Diese Komponenten entstehen aus dem Problemlösungsverfahren.

## 4.5 Zusammenfassung

Mit den bisher dargestellten Konstrukten zur Spezifikation können die Bestandteile eines Systems, die Komponenten, statisch beschrieben werden. Bei der Betrachtung von Systemen im Sinne von berechnenden Systemen, wie sie im Bereich der Informatik betrachtet werden, ist die entscheidende Fragestellung nicht die Beschreibung existierender Systeme, sondern die Beschreibung von Systemen zur automatisierten Lösung einer Problemstellung. Um diese Aufgabe lösen zu können, muss zunächst die Problemstellung bzw. das entsprechende Lösungsverfahren geeignet beschrieben werden. Für diese Aufgabe, die Beschreibung von Algorithmen, werden in der Informatik (Programmier-)Sprachen eingesetzt. Dabei hat die Praxis gezeigt, dass für unterschiedliche Problemklassen unterschiedliche Sprachen geeignet sind. Hierbei spielen neben objektiven Kriterien aber auch Präferenzen der Anwender, in diesem Fall Programmierer, eine entscheidende Rolle.

Um ein solches Problemlösungsverfahren automatisiert ablaufen zu lassen, ist andererseits aber auch eine Ausführungsumgebung notwendig. Die Ausführungsumgebung stellt die Basismechanismen zur Durchführung der Problemlösung bereit. Dabei ist die Aufgabe des Managements die semantisch äquivalente Abbildung des Problemlösungsverfahrens auf die Ausführungsumgebung.

Im gleichen Maße, wie die Problemlösungsverfahren, die mit einer Sprache beschrieben werden, sich unterscheiden, sind auch die Ausführungsumgebungen für Systeme variabel. Daher ist, analog zu den Programmiersprachen, eine Sprache zur Beschreibung der Ausführungsumgebungen notwendig.

Zur Lösung dieser Aufgabenstellung ist es notwendig, sowohl die Sprache zur Beschreibung der Problemlösung, als auch diejenige für die Ausführungsumgebungen zu kennen. Um die korrekte Ausführung der Problemlösung mittels der Ausführungsumgebung zu ermöglichen, müssen auch die Zusammenhänge zwischen diesen beiden beschrieben werden.

In dieser Arbeit wird, um dies zu ermöglichen, ein Ansatz verfolgt, der die einheitliche Beschreibung sowohl der Elemente der Problemlösung, als auch der Ausführungsumgebung sowie der Zusammenhänge ermöglicht. Die Beschreibung der Möglichkeiten für die Problemlösung, sowie für die Ausführungsumgebung und die Strategien der Realisierung, also der Zusammenhänge zwischen Problemlösung und Ausführungsumgebung, sollte erweiterbar sein.

Dies wird ermöglicht, indem für beide Aspekte eine komponentenbasierte Sichtweise verwendet wird. Auf der Ebene der Problembeschreibung sind die Komponenten als Elemente der Programmiersprache vorstellbar. Diese Elemente können nach bestimmten, in der Syntax und der Semantik der Sprache festgelegten Regeln kombiniert werden. Bei der klassischen Realisierung einer Programmiersprache durch einen Übersetzer, welcher die Elemente der Programmiersprache in Maschinensprache transformiert, wird die Einhaltung dieser Regeln überwacht.

Auf der Ebene der Ausführungsumgebungen können ebenso Komponenten unterschiedlicher Granularität festgelegt werden. Die vollständige Ausführungsumgebung ergibt sich aus der Kombination dieser Komponenten. Ebenso wie bei Programmiersprachen existieren auch für die Kombination dieser Komponenten Regeln.

Von besonderem Interesse sind aber die Zusammenhänge zwischen diesen beiden Welten. Die Abbildung von programmiersprachlichen Konstrukten auf die Komponenten der Ausführungsumgebung wird nach vorgegebenen Regeln durchgeführt. Dabei werden die Komponenten der Sprache auf Ressourcen abgebildet. Diese wiederum werden ebenfalls durch Ressourcen einer niedrigeren Abstraktionsebene realisiert. Dieser Prozess setzt sich fort, bis die Ebene der Hardware-Ressourcen erreicht wird.

**Beispiel 4.7**
In Abb. 4.5 auf der nächsten Seite ist für das Beispiel 4.2 auf Seite 56 der Weg von der Spezifikation über die Komponentenklassen zum laufenden System dargestellt.

Die Abb. 4.5 zeigt im oberen Drittel eine Menge von Metatypen. Die Metatypen können in Ressourcen (*Thread* und *Knoten*) und Sprachelemente unterschieden werden. Aus den Metatypen werden die Komponentenklassen abgeleitet, in der mittleren Ebene der Abb. 4.5 dargestellt. Aus den Sprachelementen wird auf dieser Ebene das Problemlösungsverfahren abgeleitet, von den Ressourcen lassen sich Beschreibungen für mögliche Realisierungen ableiten. Für die Ressource *Knoten* beispielsweise steht die Realisierung *Pc* zur Verfügung. Aus diesen Komponentenklassen entsteht dann zur Ausführungszeit das dynamische System, welches in Abb. 4.5 im unteren Drittel als Schnappschuss dargestellt ist. Das laufende System besteht einerseits aus der Menge der real vorhandenen Hardware-Ressourcen (*Pc - 1* und *Pc - 2*), den Zwischenressourcen der Klasse *pthread* und den Komponenten des Problemlösungsverfahrens.

Zwischen den Komponenten des Systems bestehen eine Menge von Abhängigkeiten, die ebenfalls im unteren Drittel von Abb. 4.5 dargestellt sind. Einerseits gibt es Nutzungsbeziehungen zwischen den Komponenten des Problemlösungsverfahrens, andererseits Realisierungsbeziehungen zwischen diesen und den Ressourcen-Komponenten.

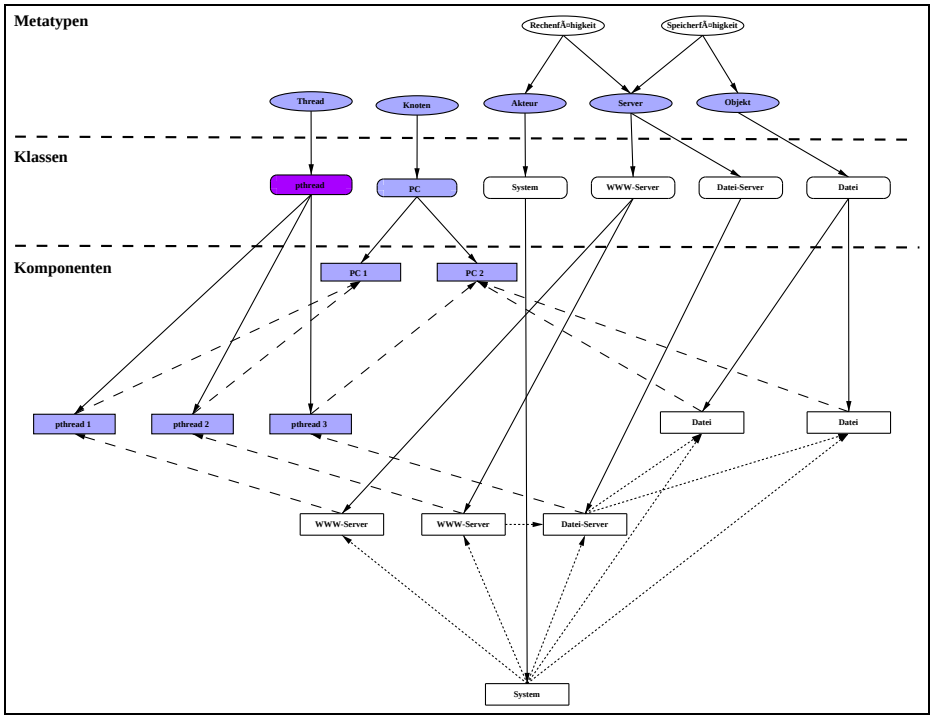



Abbildung 4.5: Von der Spezifikation zum System in Ausführung.

# 5 Spezifikation dynamischer Systeme

Ein zentrales Charakteristikum eines jeden Systems ist seine Dynamik. Diese findet ihren Ausdruck in der ständigen Veränderung des Systems bzw. dem Zustand des Systems. Einerseits entstehen neue Teile des Systems und bestehende Teile werden aufgelöst. Andererseits verändert sich der Zustand des Systems bzw. seiner Teile indem neue Bindungen zwischen Systemteilen entstehen oder Bindungen aufgelöst werden.

Diese Veränderungen eines Systems müssen in geeigneter Weise beschrieben werden. Da für jede Veränderung des Systems ein Auslöser existieren muss, ist es notwendig diese „Auslöser“ zu spezifizieren.

Die Spezifikation der zeitlichen Zusammenhänge ist von zentraler Bedeutung, da die Korrektheit im Ablauf eines Systems nur gewährleistet werden kann, wenn zeitliche Restriktionen eingehalten werden. Diese Aufgabenstellung wird im Allgemeinen als Synchronisation bezeichnet. Es ist daher für das Management von Bedeutung, die Restriktionen zu kennen. Anderseits kann das Management das Wissen über zeitliche Abläufe im System nutzen, um frühzeitig geeignete Maßnahmen ergreifen zu können. Von besonderem Interesse sind dabei Vorhersagen über das Verhalten der Systeme bereits zum Zeitpunkt der Generierung des Managements.

## 5.1 Ereignisse

Um die zeitlichen Aspekte der Veränderungen von Zuständen eines Systems in einer Spezifikation korrekt erfassen zu können, ist es zunächst notwendig, aus der Beobachtung von Systemen die zentralen Aspekte der zeitlichen Abläufe zu analysieren. Die Analyse von Systemen wird unter anderem in [Kra00] ausführlich diskutiert. In dieser Arbeit wurden die Auslöser für Zustandsänderungen ausführlich untersucht. Im Folgenden werden diese Auslöser einer Zustandsveränderung als Ereignisse bezeichnet:

**Definition 5.1 (Ereignis)**
*Ein Ereignis ist eine Änderung von Zuständen eines Systems, welche zu einem Zeitpunkt im Ablauf des Systems stattfindet.*

Die Zeit wird dabei durch das Bezugssystem des Beobachters gebildet. Diese Zeit ist im Allgemeinen eine kontinuierliche Zeit, die sich mit den Elemente aus $\mathbb{R}$ darstellen lässt. Ungeachtet dessen kann innerhalb des Systems ein anderes, eventuell diskretes Zeitmodell verwendet werden, das dem Beobachter aber zunächst verborgen bleibt.

Ein Ereignis ist somit atomar in der Zeit (*t-atomar*) und in seinen Wirkungen (*w-atomar*) (vgl. [Spi98b]). Der gesamte Ablauf eines Systems lässt sich durch eine partiell geordnete Menge von Ereignissen beschreiben. Diese Ereignismenge wird im Folgenden als *Spur* bzw. *Ereignisspur* bezeichnet. Ebenso können einerseits Teilsysteme, aber auch zeitliche Ausschnitte der Lebenszeit eines Systems durch Spuren beschrieben werden.

Jedes Ereignis in einem dynamischen System wird entweder durch eine Aktion innerhalb des Systems ausgelöst oder durch eine Einwirkung von außerhalb des Systems. Daher lassen sich Ereignisse in *interne* und *externe* Ereignisse unterscheiden. Analog lassen sich auch die Ereignisse innerhalb eines Teilsystems in interne und externe Ereignisse differenzieren. Ein System in dem nur interne Ereignisse auftreten, wird als *abgeschlossenes System* bezeichnet. Die Klassifizierung in interne und externe Ereignisse ist dabei willkürlich, insofern sie von der betrachteten Granularität der Teilsysteme abhängt. Mit anderen Worten, betrachtet man ein System in dem nur externe Ereignisse auftreten, so kann dieses System als *minimal* betrachtet werden. Jedes System lässt sich in eine endliche disjunkte Menge von minimalen Teilsystemen zerlegen. Die Teilsysteme bestehen aus den Komponenten, wie sie in Definition 4.1 auf Seite 53 festgelegt wurden. Über die Teilsysteme ist daher eine Gruppierung von Komponenten auf der Basis der Kooperation durch Ereignisse gegeben. Diese Kooperation kann zum Einen horizontal sein, wobei Komponenten derselben Abstraktionsebene zusammenarbeiten. Zum Anderen können aber auch vertikale Kooperationen als Teilsysteme identifiziert werden. Dabei handelt es sich um die Kooperation zwischen Komponenten und ihren Ressourcen.

Ein abgeschlossenes System, bei dem nur interne Ereignisse auftreten, ist von seiner Umwelt unabhängig und nicht beeinflussbar. Allerdings ist dabei keine Aussage darüber enthalten, ob ein abgeschlossenes System nicht Ereignisse in seiner Umwelt erzeugt.

Externe Ereignisse sind ein Kommunikationsmechanismus zwischen einem System und seiner Umwelt bzw. zwischen den Teilen eines Systems. Das Auftretens eines (externen) Ereignisses trägt Information aus der Umgebung eines Systems in das System hinein. Neben der impliziten Information des Auftreten eines Ereignisses können Ereignisse auch explizite Information tragen. Diese Information wird als *Ereignisparameter* bezeichnet.

### 5.1.1 Phasen

Ereignisse sind, wie oben erwähnt, atomar in der Zeit. Für die Beschreibung der Dynamik eines Systems ist die Zeit aber von Bedeutung. Einerseits ist die zeitliche Ordnung der Ereignisse, andererseits die Zeitspanne zwischen zwei Ereignissen von Interesse.

Die zeitliche Ordnung der Ereignisse strukturiert die Lebenszeit eines Systems. Die Zeitspanne zwischen zwei Ereignissen wird als (Lebens-)Phase bezeichnet:

**Definition 5.2 (Phase)**
*Eine (Lebens)Phase eines Systems ist eine Zeitspanne, welche durch zwei Ereignisse begrenzt wird.*

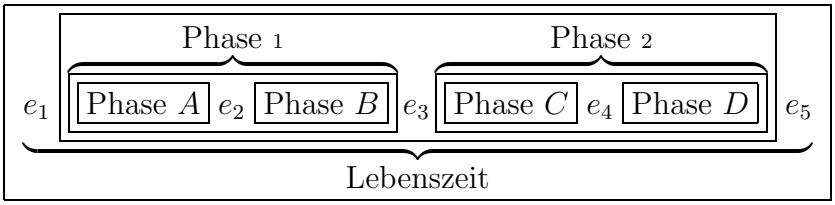



Abbildung 5.1: Phasen eines Systems.

Eine Phase beginnt stets mit einem Ereignis und endet mit einem anderen Ereignis, welches bzgl. der zeitlichen Ordnung später liegt. Somit sind Phasen im Gegensatz zu Ereignissen nicht atomar. Allerdings haben Phasen ebenso wie Ereignisse eine Ordnung. Phasen folgen entweder aufeinander, d. h. das Ende der einen Phase markiert den Beginn der folgenden Phase, wobei das Startereignis der Letzteren gleichzeitig das Endereignis der Ersteren ist. Zudem können Phasen geschachtelt sein, d. h. eine Phase kann in eine Menge von Teilphasen zerlegt werden. Dies ist in Abb. 5.1 dargestellt. In die Phase der Lebenszeit sind die Phasen

1 und 2 eingeschachtelt, welche sequentiell aufeinander folgen. In die Phasen 1 und 2 selbst sind die Phasen $A$ und $B$ bzw. $C$ und $D$ eingeschachtelt.

Aufgrund der Schachtelungsstruktur und der sequentiellen Abfolge der Phasen ergibt sich, dass Ereignisse Start- sowie Endereignis mehrerer Phasen sein können. In Abb. 5.1 auf der vorherigen Seite ist das Ereignis $e_1$ Startereignis der Lebenszeit, sowie der Phasen 1 und $A$. Das Ereignis $e_2$ ist einerseits Endereignis der Phase $A$, andererseits aber Startereignis der Phase $B$. Die Phasen $B$ und 2 werden beide durch das Ereignis $e_3$ beendet. Dieses Ereignis ist aber auch Startereignis für die Phasen $C$ und 2. Analoges gilt für die Ereignisse $e_4$ und $e_5$.

### 5.1.2 Charakteristika von Ereignissen

Die einzelnen Ereignisse, die während der Lebenszeit eines Systems auftreten, besitzen Charakteristika, die eine Einteilung der Ereignisse ermöglichen. Insbesondere sind die folgenden Merkmale von Interesse:

1. **Teilsystem**
   Jedes Ereignis lässt sich einem oder mehreren Teilsystemen zuordnen. Da die Teilsysteme in einer Schachtelungsstruktur geordnet sind, führt die Zuordnung eines Ereignisses zu einem Teilsystem auch automatisch zu einer Zuordnung des Ereignisses in allen dieses Teilsystem umschließenden (Teil-)Systemen.

2. **Phasen**
   Jedes Ereignis liegt entweder in einer Phase seines Teilsystems oder markiert den Übergang von einer Phase in die nächste Phase. Ebenso wie bei der Zuordnung zu Teilsystemen ergibt sich aus der Schachtelungsstruktur der Phasen eine Zuordnung in mehrere Phasen.

3. **Vorbereich**
   Bestimmte Ereignisse können nur auftreten, falls vor dem Ereignis eine bestimmte Ereignisfolge aufgetreten ist.

4. **Zustand des Systems**
   Bestimmte Ereignisse treten nur in einem bestimmten Zustand des Systems oder Teilsystems auf. Da die Veränderungen des Zustandes nur durch Ereignisse hervorgerufen werden können, kann dieses Charakteristikum auf das Vorherige zurückgeführt werden.

5. **Abstraktionsebene**
   Die Ereignisse eines Systems treten jeweils in einer Abstraktionsebene auf. Der Auslöser eines Ereignisses kann aber in einer anderen Abstraktionsebene liegen. Somit dienen Ereignisse der Kommunikation über Abstraktionsebenen hinweg.

Neben den eben beschriebenen Ereignischarakteristika existiert noch ein weiteres Merkmal von Ereignissen. Wie bereits beschrieben, können Ereignisse für ein gegebenes (nicht-minimales) Teilsystem $t$ in interne und externe Ereignisse unterteilt werden. Dieses Teilsystem kann aber derart in zwei Teilsysteme $t_1$ und $t_2$, die nicht notwendig minimal sind, zerlegt werden, so dass ein Ereignis $e$ in einem der Teilsysteme, zum Beispiel $t_2$, als externes Ereignis auftritt. Da $e$ in $t$ ein internes Ereignis war, muss die Ursache für $e$ in $t_1$ liegen.

Die Ursache für das Ereignis selbst liegt in einer Zustandsveränderung von $t_1$, die entweder durch $t_1$ selbst, also den Berechnungsfortschritt von $t_1$, ausgelöst wird, oder aber durch ein anderes, externes Ereignis $e'$. Dieser kausale Zusammenhang zwischen $e$ und $e'$ wird als *Ereigniszusammenhang* bezeichnet.

**Beispiel 5.1**
Sei $A$ eine aktive Komponente, welche während ihrer Laufzeit eine Kindkomponente $K$, ebenfalls eine aktive Komponente, abspaltet und diese vor der Terminierung wieder einfängt. Das Abspalten des Kindes wird durch das Ereignis $k_s$, das Einfangen des Kindes durch $k_e$ modelliert. Initiierung und Terminierung der aktiven Komponenten wird jeweils durch die Ereignisse $i$ und $t$ markiert. Der Ablauf der beiden Komponenten ist in Abb. 5.2(a) auf dieser Seite dargestellt.

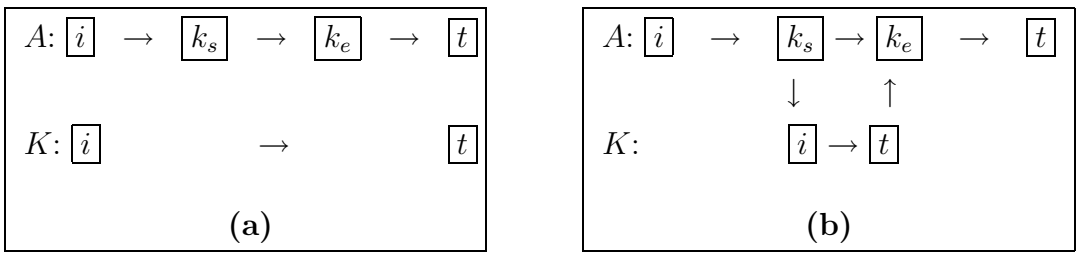



Abbildung 5.2: Eltern-Kind-Zusammenhang mittels Ereignissen.

Das Auftreten des Ereignisses $i$ in $K$ ist aber eine unmittelbare Folge der Auslösung von $k_s$ in $A$. Wohingegen das Ereignis $k_e$ in $A$ nur auftreten kann, wenn das Ereignis $t$ in $K$ auftritt. Diese Ereigniszusammenhänge sind in Abb. 5.2(b) auf dieser Seite dargestellt.

**Definition 5.3 (Ereigniszusammenhänge)**
*Ein Ereignis e in einer Komponente C heißt abhängiges Ereignis, genau dann wenn, es ein Ereignis $e'$ in einer Komponente $C'$ gibt, so dass e dann und nur dann auftritt, wenn $e'$ auftritt.*

*Wenn die Komponenten C und $C'$ in einer Relation $\mathcal{R} \subseteq \mathcal{C} \times \mathcal{C}$ enthalten sind, so wird e auch als $\mathcal{R}$-abhängig von $e'$ bezeichnet.*

Die Ereignisse innerhalb eines Systems sind gemäß ihrer zeitlichen Abfolge partiell geordnet. Ereignisse, die in dieser partiellen Ordnung nicht vergleichbar sind, können gleichzeitig auftreten. Die Ereignisse innerhalb einer Komponente dagegen sind total geordnet, d. h. innerhalb einer Komponente gilt für zwei Ereignisse $e_1$ und $e_2$ stets, dass entweder $e_1$ zeitlich vor $e_2$ aufgetreten ist, oder aber $e_2$ vor $e_1$. Die totale Ordnung der Ereignisse für eine Komponente ergibt sich aus der Wirkungsatomarität der Ereignisse. Die Zustandsänderung, deren Auslöser ein Ereignis war, tritt stets vollständig ein. Damit kann ein zweites Ereignis keinen inkonsistenten Zustand der Komponente vorfinden. Daraus ergibt sich aber auch, dass das Ereignis $e_2$ entweder den Zustand vor dem Ereignis $e_1$ vorfindet und somit $e_1$ zeitlich vor $e_2$ stattfindet oder aber $e_2$ den Zustand nach $e_1$ vorfindet und somit nach $e_1$ stattfindet.

Ereignisse dienen der Kommunikation einerseits zwischen dem Management unterschiedlicher Teilsysteme, aber auch zwischen Management und Anwendung. Eine Zustandsänderung innerhalb des Managements wird entweder durch ein Ereignis innerhalb des Managements selbst ausgelöst. Somit transportiert ein Ereignis Information innerhalb des Managements. Eine Zustandsänderung innerhalb des Managements kann aber auch die Reaktion auf eine Veränderung innerhalb der Anwendung sein. Das entsprechende Ereignis transportiert dann Information aus der Anwendung in das Management. Erstere Art der Ereignisse wird als *managementinternes* Ereignis bezeichnet, zweitere Art als *Anwendungsereignis*.

## 5.2 Spezifikation von Ereignissen

Die bisherigen Betrachtungen über Ereignisse bezogen sich auf die Beschreibung laufender Systeme, also auf die Beobachtung von Systemen von außen. Für die Spezifikation eines Systems müssen Ereignisse bzw. Ereignisklassen statisch beschrieben werden. Diese Beschreibung muss alle möglichen Ereignisse, deren Parameter und Klassifikation, sowie die möglichen Ereignisspuren umfassen.

Im Folgenden wird somit die Spezifikation von zeitlichen Aspekten diskutiert. Dabei werden die Ereignisse zum zentralen Mittel der Beschreibung. Ereignisse sind in dieser Sichtweise nicht mehr die abstrakten Auslöser einer beobachteten Zustandsveränderung, sondern die spezifizierten Ereignisse können entweder in der Anwendung oder dem Management erzeugt werden und bewirken dann eine Zustandsänderung.

Mit dieser veränderten Sichtweise auf Ereignisse verändert sich auch das zeitliche Bezugssystem der Ereignisse. Die kontinuierliche Realzeit spielt in dieser Sichtweise keine Rolle. Die Ereignisse selbst definieren nun eine diskrete Zeit. Leslie Lamport stellte 1978 ein einfaches Schema für eine logische Zeit und deren Synchronisation in verteilten Systemen vor (vgl. [Lam78]). Die Grundlage der logischen Zeit wird durch die *Happened-Before-Relation* gebildet:

**Definition 5.4 (Happened-Before-Relation)**

- *Sind a und b Ereignisse innerhalb einer Komponente und tritt a vor b ein, so gilt* $a \rightarrow b$ *(a happened before b).*
- *Ist b ein durch a ausgelöstes Ereignis, so gilt* $a \rightarrow b$.

Diese Relation entspricht der kausalen Ordnung der Ereignisse. Nachfolgende Ereignisse innerhalb einer Komponente hängen potenziell immer von den vorhergehenden Ereignissen in der gleichen Komponente ab. Ein Ereignis in einer Komponente hängt potenziell immer von den Ereignissen ab, welche dem auslösenden Ereignis vorangehen. Die Happend-Before-Relation ist partiell und transitiv.

Theoretisch führt jede Komponente ihre private logische Uhr $t$ als Eigenschaft. Bei jedem Ereignis $\sigma$ wird diese Uhr erhöht, und somit jedem Ereignis ein Zeitstempel zugeordnet. Der Zeitstempel eines Ereignisses wird im Folgenden als $t(\sigma)$ bezeichnet. Diese Zeitstempel müssen der Happend-Before-Relation entsprechen, was durch folgenden Algorithmus erreicht wird:

a) Sei $\sigma$ ein komponenten-lokales Ereignis:

$$t := t(\sigma) := t + 1$$

b) Sei $\sigma$ ein von $\sigma'$ abhängiges Ereignis:

$$t := t(\sigma) := max(t, t(\sigma')) + 1$$

Die Einheiten dieser logischen Zeit entsprechen unterschiedlichen Zeitspannen in der realen Zeit. Sei $\tau(\sigma)$ der Zeitpunkt des Ereignisses $\sigma$ in Realzeit, dann gilt:

$$t(\sigma) < t(\sigma') \Rightarrow \tau(\sigma) < \tau(\sigma')$$

### 5.2.1 Komponenten und Ereignisse

Für jeden Metatypen kann eine Menge von Ereignisklassen angegeben werden. Die Ereignisklassen des Metatyps $M$ werden als $\Sigma_M$ bezeichnet. Die Ereignisklassen aller Metatypen als $\Sigma$. Die Ereignisse dieser Klassen strukturieren die Lebenszeit der Komponente, welche von diesem Metatypen abgeleitet werden. Die Lebenszeit einer Komponente beginnt mit einem Ereignis einer ausgezeichneten Klasse und endet mit einem Ereignis aus einer Menge von Klassen. Die Erstere wird als *Startklasse*, die Letzteren als *Terminierungsklassen* bezeichnet.

Wesentlich ist hierbei die Unterscheidung zwischen den aufgetretenen Ereignissen und den Ereignisklassen. Die für eine Komponente $C \in \mathcal{C}$ bis zum Zeitpunkt $t \in T$ aufgetretenen Ereignisse werden durch die Menge $\mathcal{H}_C(t), t \in T$ beschrieben. Die Mengen $\mathcal{H}_C(t)$ wachsen monoton mit der Zeit, da ein zum Zeitpunkt $t$ bereits aufgetretenes Ereignis auch zum Zeitpunkt $t' > t$ aufgetreten ist. Es gilt also

$$t < t' \Rightarrow \mathcal{H}_C(t) \subseteq \mathcal{H}_C(t')$$

Jedes aufgetretene Ereignis $e \in \mathcal{H}_C(t)$ ist von einer Ereignisklasse $\sigma \in \Sigma_M$, mit $M \rightsquigarrow C$ abgeleitet. Die Abbildung $\theta : \mathcal{H}_C(t) \rightarrow \Sigma$ ordnet jedem aufgetreten Ereignis seine Klasse zu. Im Folgenden wird der Begriff Ereignis sowohl für Ereignisklassen, als auch für aufgetretene Ereignisse verwendet, falls die Bedeutung aus dem Kontext offensichtlich ist.

Die potenziellen Ereignisfolgen lassen sich als Sprache mit den Ereignisklassen als Alphabet beschreiben. Jede zur Laufzeit eines Systems auftretende Folge von Ereignissen bildet ein Wort dieser Sprache. Aufgrund der Charakteristika aus Abschnitt 5.1.2 auf Seite 78 ist diese Sprache im Allgemeinen kontextsensitiv, da das Auftreten eines Ereignisses abhängig vom seinem Vorbereich ist.

Für die Spezifikation von Ereignisfolgen sind kontextsensitive Sprachen allerdings untauglich. Einerseits ist die Beschreibung der Abhängigkeiten eines Ereignisses vom Zustand des Systems ausschließlich durch die Ereignisfolge für die Erstellung der Spezifikation unverhältnismäßig komplex und somit nicht handhabbar. Die Handhabbarkeit der Systeme ist aber gerade eines der zentralen Ziele einer Spezifikation. Andererseits ist es notwendig, die Ereignisfolgen der Komponenten eines Systems sowohl statisch, als auch dynamisch zu analysieren. Bereits die Entscheidung des Wortproblems auf kontextsensitiven Sprachen ist aber nur mit exponentieller Komplexität entscheidbar.

Für die Festlegung der potenziellen Ereignisfolgen der Komponenten werden daher Zusatzbedingungen eingeführt, mittels welcher die Abhängigkeiten zwischen dem Zustand einer Komponente und der Zulässigkeit eines Ereignisses beschrieben werden können. Sei $M$ ein Metatyp und $C$ eine Komponente, welche von $M$ abgeleitet wurde. Dann können die Ereignisbedingungen für die Ereignisse der Komponente $C$ als Abbildungen der Form $Z(C, \mathcal{E}_M, t) \rightarrow \mathbb{B}$ beschrieben werden. Damit ist es möglich, die Abhängigkeit der Ereignisse vom Zustand der Komponenten eindeutig zu beschreiben.

Für die Beschreibung der Abhängigkeit eines Ereignisses von seinem Vorbereich ist nun eine allgemeinere Form der Spezifikation von Ereignisfolgen ausreichend, d. h. eine Beschreibungsform, welche weniger Einschränkungen bzgl. der Zulässigkeit von Ereignissen impliziert. Die in diesem Sinne allgemeinste Form der Beschreibung von Sprachen sind die regulären Sprachen.

**Lemma 5.1**
*Jede Sprache $L$ über dem Alphabet $A$ ist Teilmenge einer regulären Sprache $L_R$ über dem Alphabet $A$.*

**Beweis**

Sei $A = \{a_1, \ldots, a_n\}$. Die Worte über dem Alphabet $A$ sind dann beliebige Kombinationen der Zeichen aus $A$. Diese können durch den regulären Ausdruck $(a_1|\ldots|a_n)^*$ beschrieben werden.

Da jedes Wort $w \in L$ ebenfalls eine Kombination der Zeichen aus $A$ ist, gilt $L \subseteq (a_1|\ldots|a_n)^*$ und, da $(a_1|\ldots|a_n)^*$ ein regulärer Ausdruck ist, gilt die Behauptung des Satzes $L \subseteq L_R$.

*q.e.d.*

Mit der Kombination der zulässigen Ereignisfolge gemäß einer regulären Sprache und den Ereignisbedingungen lassen sich die zulässigen Ereignisfolgen spezifizieren. Da die regulären Sprachen sich mittels der regulären Ausdrücke beschreiben lassen, kann für die Spezifikation auf diese Beschreibungsform zurückgegriffen werden. Dies führt zu einer leichten Handhabbarkeit der Ereignisspezifikation, da die regulären Ausdrücke einen intuitiven Umgang mit Ereignisfolgen ermöglichen.

Wie in Definition 5.3 auf Seite 80 festgelegt, können Ereignisse äquivalent sein. In der Spezifikation ergibt sich die Notwendigkeit, diese Ereigniszusammenhänge beschreibbar zu machen. Die Ereigniszusammenhänge können als eine Relation $\mathcal{Z} \subseteq \Sigma \times \mathcal{E} \times \Sigma$ beschrieben werden. Jedem Ereignis $\sigma \in \Sigma$ wird durch die Relation $\mathcal{Z}$ eine Menge von Paaren aus Eigenschaften und Ereignissen zugeordnet,

die von $\sigma$ abhängig sind. Die Abhängigkeiten zwischen Ereignissen werden hierbei entlang der Strukturdefinition gemäß Definition 4.9 auf Seite 64 spezifiziert. Diese Festlegung impliziert eine Zuordnung der Ereignisse zu den Komponenten eines Systems. In der Spezifikation wird dies dadurch erreicht, dass die Ereignisse den Metatypen zugeordnet werden. Jedes Ereignis ist für genau einen Metatypen definiert. Da Komponenten direkt nur von nutzbaren Metatypen (vgl. Definition 4.3 auf Seite 56) abgeleitet werden können, ist jedes Ereignis für einen nutzbaren Metatypen definiert. Dieser Zusammenhang wird durch die Abbildung $\varsigma : \Sigma \rightarrow \mathcal{U}$ beschrieben, die jedem Ereignis aus $\Sigma$ einen Metatypen aus $\mathcal{U}$, der Menge der nutzbaren Metatypen, zuordnet. Die Menge der Ereignisse eines nutzbaren Metatypen $U \in \mathcal{U}$ wird als $\Sigma_U = \{\sigma \mid \varsigma(\sigma) = U\}$ bezeichnet.

Zur Vereinfachung der Spezifikation können Ereignisse auch für nicht nutzbare Metatypen spezifiziert werden, wobei diese entsprechend der Vererbungshierarchie der Metatypen an die Untertypen vererbt werden. Damit die Zuordnung von Ereignissen zu nutzbaren Metatypen dabei eindeutig bleibt, wird dabei allerdings festgelegt, dass die geerbten Ereignisse in zwei Untertypen nicht identisch sind.

Seien also $M$ und $M'$ jeweils nutzbare Metatypen und sei $A$ ein abstrakter Metatyp, wobei gilt $M \leftarrow A$ und $M' \leftarrow A$. Sei weiterhin $\sigma$ ein für $A$ definiertes Ereignis, dann gibt es zwei nicht identische Ereignisse $\sigma_M$ und $\sigma_{M'}$ mit $\varsigma(\sigma_M) = M$ und $\varsigma(\sigma_{M'}) = M'$, deren Ereignisparamter und Ereignisbedingung denen des Ereignisses $\sigma$ entsprechen. Analog zur Schreibweise der Untertypbeziehung auf Metatypen wird dieser Zusammenhang zwischen $\sigma$ und $\sigma_M$ bzw. $\sigma_{M'}$ als $\sigma_M \leftarrow \sigma$ bzw. $\sigma_{M'} \leftarrow \sigma$ bezeichnet.

Damit ergibt sich für die Spezifikation der Dynamik der Komponenten eines Metatypen folgende Definition:

**Definition 5.5 (Ereignisspezifikation)**
*Eine Ereignisspezifikation für einen Metatypen ist ein Tupel* $\langle \Sigma, \Pi, B, \mathcal{Z}, R \rangle$ *mit*

- *Menge der Ereignisklassen* $\Sigma_M$
- *Ereignisparameter* $\Pi : \Sigma_M \rightarrow \mathcal{D} \times \ldots \times \mathcal{D}$
- *Ereigniszusammenhängen* $\mathcal{Z} \subseteq \Sigma_M \times \mathcal{E}_M \times \Sigma$
- *Ereignisbedingungen* $B : \Sigma_M \times \mathcal{E}_M \times T \rightarrow \mathbb{B}$
- *Potenzielle Ereignisfolge* $R_M = \begin{cases} \sigma_{start} R' & : \quad M \in \mathcal{U} \\ \epsilon & : \quad sonst \end{cases}$,
  *wobei* $\sigma_{start} R'$ *ein regulärer Ausdruck über* $\Sigma_M$ *ist.*

Mit der Festlegung der potenziellen Ereignisfolgen ist die Klasse des Startereignisses ($\sigma_{start}$)für jeden nutzbaren Metatypen eindeutig festgelegt. Die Klasse der Endereignisse ergibt sich implizit aus dem regulären Ausdruck $R$, da aus dem regulären Ausdruck $R$ durch die bekannten Verfahren (siehe [HMU01]) ein deterministischer endlicher Automat $A = \langle \Sigma, \zeta, \delta, \zeta_0, \phi \rangle$ gebildet werden kann. Dabei ist $\zeta_0 \in \zeta$ der Startzustand des Automaten und $\phi \subseteq \zeta$ die Menge der Endzustände. Die Menge der Endereignisklassen wird durch die Ereignisse gebildet, welche im Automaten in einen der Endzustände aus $\phi$ führen. Für eine Komponente $C$ des Metatypen $M$ wird das Startereignis als $\sigma_{start}(C) \in \Sigma_M$ bezeichnet, das Endereignis als $\sigma_{term}(C) \in \Sigma_M$. Für ein Ereignis $\sigma \in \Sigma$ bezeichnet $pre(\sigma)$ die Menge aller Worte, die Präfixevon Ereignisfolgen bis $\sigma$ sind.

Die Ereignisse, deren zeitliche Abfolge in der Spezifikation der potenziellen Ereignisfolgen festgelegt wurde, werden als *synchrone Ereignisse* bezeichnet. Ereignisse, deren Auftreten unabhängig von der Ereignisfolge ist, welche also keinen Vorbereich besitzen, werden dagegen als *asynchrone Ereignisse* bezeichnet. Asynchrone Ereignisse können somit unabhängig von der bisherigen Ereignisreihenfolge auftreten und sind auch nicht im Vorbereich eines zeitlich späteren Ereignisses enthalten. Asynchrone Ereignisse beeinflussen aber den Zustand der Komponenten und somit den Systemzustand. Ebenso kann die Zulässigkeit des Auftretens asynchroner Ereignisse durchaus vom augenblicklichen Systemzustand abhängen.

Ohne Beschränkung der Allgemeinheit kann auf die Betrachtung asynchroner Ereignisse in den Spezifikationen verzichtet werden, da die asynchronen Ereignisse in die Folge der synchronen Ereignisse integriert werden kann. Dafür ist folgende Definition notwendig:

**Definition 5.6 ($x$-erweiterte Ereignisfolgen)**
*Sei $R$ ein regulärer Ausdruck zur Beschreibung einer Ereignisfolge mit der Ereignismenge $\Sigma$. Sei $x \in \Sigma$ ein Ereignis der Ereignismenge. Der Ausdruck $R\&x$ ergibt sich aus $R$ wie folgt:*

*a) Sei $R = a$, mit $a \in \Sigma$, dann ist $R\&x = ax*$

*b) Sei $R = AB$, wobei $A$ und $B$ reguläre Ausdrücke sind, dann ist $R\&x = (A\&x)(B\&x)$*

*c) Sei $R = A \mid B$, wobei $A$ und $B$ reguläre Ausdrücke sind, dann ist $R\&x = (A\&x) \mid (B\&x)$*

*d) Sei $R = A*$, wobei $A$ ein regulärer Ausdruck ist, dann ist $R\&x = (A\&x)*$*

*$R\&x$ wird als $x$-**erweiterte Ereignisfolge** bezeichnet.*

Für die $x$-erweiterten Ereignisfolgen gilt folgender Satz:

**Satz 5.2 (Elimination asynchroner Ereignisse)**
*Sei $R$ ein regulärer Ausdruck zur Beschreibung einer Ereignisfolge mit der Ereignismenge $\Sigma$, sowie $a \in \Sigma$. Sei weiterhin $b$ ein asynchrones Ereignis, wobei gilt: $\nexists w \in L(aR) : b$ ist in $w$ enthalten. Dann gilt:*

*a)* $u \in L(R) \Rightarrow u \in L(R\&b)$.

*b)* $uv \in L(R\&b) \Rightarrow ubv \in L(R\&b)$, *mit* $|u| \geq 1$.

*c)* $u \notin L(R) \Rightarrow u \notin L(R\&b)$, *falls $b$ nicht in $u$ enthalten ist.*

**Beweis**

a) $u \in L(R) \Rightarrow u \in L(R\&b)$:
Der Beweis von $u \in L(R) \Rightarrow u \in L(R\&b)$ erfolgt per Induktion über die Länge des Wortes $u$ (in Zeichen $|u|$).

1. Sei $|u| = 1$:

   i. Sei $R = c$:
   Dann gilt $R\&b = (cb*)$. Da nach Vorrausetzung gilt $u \in L(R)$ muss $u = c$ sein und $c$ ist offensichtlich in $L(cb*)$.

   ii. Sei $R = AB$, wobei $A$ und $B$ reguläre Ausdrücke sind:
   Dieser Fall ist für $|u| = 1$ nicht möglich, falls $A \neq \epsilon$ und $B \neq \epsilon$, da dann Folgendes gilt:

   $$\forall v \in L(AB) \ : \ |v| \geq 2$$

   Falls $A = \epsilon$ oder $B = \epsilon$, so ist dieser Fall äquivalent zu i.

   iii. Sei $R = A \mid B$, wobei $A$ und $B$ reguläre Ausdrücke sind:
   Da nach Vorrausetzung $|u| = 1$ gilt, gilt entweder $u \in L(A)$ und $A = c$ oder $u \in L(B)$ und $B = c$. In beiden Fällen gilt die Argumentation aus i. analog.

   iv. Sei $R = A*$, wobei $A$ ein regulärer Ausdruck ist:
   Da nach Vorrausetzung $|u| = 1$ gilt, ist für $A$ einer der folgenden Fälle möglich:

   - $A = c$
   - $A = A_1|\ldots|A_n$, wobei mindestens ein $u \in L(A_j)$ und $A_j = c$.

In beiden Fällen gilt die Argumentation aus i. analog.

2. Sei $|u| > 1$ und für alle $v \in L(R)$ mit $|v| < |u|$ gilt $v \in L(r\&b)$:

   i. Sei $R = c$:
   Dieser Fall ist nicht möglich, da $L(R) = \{c\}$ und $|c| = 1$.

   ii. Sei $R = AB$, wobei $A$ und $B$ reguläre Ausdrücke sind:
   Nach der Definition der regulären Ausdrücke existiert eine Zerlegung von $u = xy$, mit $x \in L(A)$ und $y \in L(B)$. Falls $x \neq \epsilon$ und $y \neq \epsilon$, so gilt gemäß der Induktionsvoraussetzung $x \in L(A\&b)$ und $y \in L(B\&b)$. Da $R\&b = (A\&b)(B\&b)$ gilt somit $u \in L(R\&b)$.

   Falls $x = \epsilon$ oder $y = \epsilon$ so ist entweder $u = L(B)$ oder $u = L(A)$. Gemäß den Fällen iii. oder iv. gilt dann die Behauptung.

   iii. Sei $R = A \mid B$, wobei $A$ und $B$ reguläre Ausdrücke sind:
   Da $u \in L(R)$ gilt entweder $u \in L(A)$ oder $u \in L(B)$. Da $A$ und $B$ reguläre Ausdrücke sind, liegt entweder der Fall ii. oder iv. vor.

   iv. Sei $R = A*$, wobei $A$ ein regulärer Ausdruck ist:
   Nach der Definition der regulären Ausdrücke existiert eine Zerlegung von $u = xy$, mit $x \in L(A)$ und $y \in L(A*)$. Falls $y \neq \epsilon$, so gilt $|x| < |u|$ und $|y| < |u|$ und nach Voraussetzung, dass $x \in L(A\&b)$, sowie $y \in L(A * \&b)$. Dann gilt aber nach ii. auch $u \in L(A * \&b)$.

   Falls $y = \epsilon$ gilt, so ist $u = x \in L(A)$. Gemäß den Fällen ii. oder iii. gilt dann die Behauptung.

b) $uv \in L(R\&b) \Rightarrow ubv \in L(R\&b)$, mit $|u| \geq 1$:
Da $uv \in L(R\&b)$ gilt $R\&b = (A\&b)(B\&b)$, wobei $u \in L(A\&b)$ sowie $v \in L(B\&b)$. Nach Definition gilt $ub \in L(A\&b)$. Somit ist auch $ubv \in L((A\&b)(B\&b)) = L(R\&b)$.

c) $u \notin L(R) \Rightarrow u \notin L(R\&b)$, falls $b$ nicht in $u$ enthalten ist:
Der Beweis erfolgt durch Widerspruch und durch Induktion über die Länge des Wortes $u$. Sei also $u \notin L(R)$. Angenommen es gilt $u \in L(R\&b)$.

1. Sei $|u| = 1$:
   Da $u \in L(R\&b)$ gilt, so ist $R\&b$ entweder $c\&b = cb*$ und somit $R = c$. Für $u$ gibt es dann die Möglichkeit $u = c$, also $u \in R$, was der Voraussetzung widerspricht, oder $u = b$, was ebenfalls ein Widerspruch zur Voraussetzung ist.

2. Sei $|u| > 1$ und für alle $v$ mit $|v|+1 = |u|$ gilt $v \notin L(R) \Rightarrow v \notin L(R\&b)$, falls $b$ nicht in $v$ enthalten ist:
   Falls $u = vb \in L(R\&b)$ gilt, so ergibt sich ein Widerspruch zur Voraussetzung, dass $b$ nicht in $u$ enthalten ist.

Sei also $u = vc$. Angenommen es gilt $vc \in L(R\&b)$. Da $c \neq b$, gilt $vc \in L((R'\&b)c)$, aber nicht $vc \in L(Rc)$, mit anderen Worten es gilt $v \in L(R'\&b)$, aber nicht $v \in L(R')$. Dies ist ein Widerspruch zur Induktionsvoraussetzung.

*q.e.d.*

Obiger Satz besagt, dass jede Ereignisfolge um die asynchronen Ereignisse erweitert werden kann, ohne dabei die Restriktionen bzgl. der Reihenfolge der synchronen Ereignisse zu verletzten. Somit können sich alle nachfolgenden Betrachtungen auf synchrone Ereignisse beschränken, da die asynchronen Ereignisse in die Ereignisfolgen integriert werden können. Für die Realisierung der Durchsetzung der zeitlichen Restriktionen durch das Management allerdings sind die asynchronen Ereignisse interessant, da diese nicht in die Kontrolle zur Laufzeit des Systems eingebunden werden müssen.

Mit der Festlegung der Reihenfolge der (synchronen) Ereignisse durch die potenziellen Ereignisfolgen und die Verknüpfung der Zulässigkeit von Ereignissen mit den Zuständen der Komponenten sind die Rahmenbedingungen der zeitlichen Entwicklung eines Systems beschrieben. In einem System in Ausführung, insbesondere in einem verteilten System mit mehreren Ausführungfäden, kann die Situation eintreten, dass Ereignisse für eine Komponente erzeugt werden, welche nach obigen Restriktionen entweder überhaupt nicht oder zum entsprechenden Zeitpunkt nicht zulässig sind. Für diesen Fall sind geeignete Maßnahmen zu treffen.

Für die Komponente ergeben sich zwei Alternativen für den Umgang mit unzulässigen Ereignissen. Die erste Alternative liegt im Verwerfen eines nicht zulässigen Ereignisses. Dieses Ereignis tritt im Ablauf des Systems dann nicht auf, d. h. dieses Ereignis ist ohne Wirkung auf den Zustand des Systems. Diese Sorte von Ereignissen wird als *nicht-blockierende Ereignisse* bezeichnet.

Die alternative Fehlersemantik für Ereignisse wird als *blockierende Ereignisse* bezeichnet. Ein blockierendes Ereignis wird solange zurückgestellt, bis die Restriktionen des Empfängers für das Eintreten des Ereignisses erfüllt sind. Diese Fehlersemantik ist unter anderem dann sinnvoll, wenn ein Ereignis aufgrund unterschiedlicher Laufzeiten in einem verteilten System erzeugt wurde, bevor der Empfänger den Zustand erreicht hat, indem das Ereignis angenommen werden kann.

### 5.2.2 Spezifikation von Phasen

Wie in Abschnitt 5.1.1 auf Seite 77 dargestellt, wird die Lebenszeit einer Komponente durch Phasen strukturiert. Die Phasen einer Komponente sind einerseits

Teil des Zustandsraums der Komponenten, andererseits nützliche Vergröberungen innerhalb der Spezifikation der zeitlichen Zusammenhänge innerhalb der Komponenten. Daher ist es notwendig, die Phasen in der Spezifikation geeignet zu beschreiben. Eine Phase ist nach der Definition 5.2 auf Seite 77 der Zeitraum zwischen zwei Ereignissen. Aus der Konstruktion des endlichen Automaten aus dem regulären Ausdruck der potenziellen Ereignisfolgen ergeben sich die Automatenzustände als einfachste Phasen, d. h. Phasen zwischen zwei direkt aufeinanderfolgenden Ereignissen.

Durch Zusammenfassung von Automatenzuständen können diese minimalen Phasen zu vergröberten Phasen erweitert werden. Die Ereignisse zwischen den Phasen definieren die Phasenübergänge. Damit ist implizit auch die Schachtelung von Phasen gegeben. Sei $A = \langle \Sigma, \zeta, \delta, \zeta_0, \phi \rangle$ der endliche Automat, welcher die potenziellen Ereignisfolgen beschreibt. Dann ist $\zeta(P) \subseteq \zeta$ die Menge der Automatenzustände in der Phase $P$. Eine Phase $Q$ ist dann in eine Phase $Q'$ geschachtelt, wenn $\zeta(Q) \subseteq \zeta(Q')$ gilt. Jede Phase $P$ wird somit durch einen (Teil-)Automaten $A_P = \langle \Sigma, \zeta(P), \delta, \zeta_0(P), \phi(P) \rangle$ beschrieben, wobei $\zeta(P)$ die Menge der Zustände der Phase, $\zeta_0(P) \subseteq \zeta(P)$ die Menge der Startzustände des (Teil-)Automaten und $\phi(P) \subseteq \zeta(P)$ die Menge der Endzustände des (Teil-)Automaten ist.

Die Phasen einer Komponente des Metatypen $M \mapsto C$ können als Teil der Menge der Eigenschaften $\mathcal{E}_M$ interpretiert werden. Für die Spezifikation des Managements ist allerding im Allgmeinen nur eine Teilmenge der Phasen von Interesse. Diese Phasen werden als *benannte Phasen* bezeichnet und bilden die Elemente einer Datensorte. Sei im Folgenden $\mathcal{P}_M$ die Menge der benannten Phasen des Metatypen $M$.

In der Spezifikation werden die Phasen als Teile der regulären Ausdrücke der potenziellen Ereignisfolgen beschrieben. Die potenzielle Ereignisfolge jeder benannten Phase $P$ wird durch einen regulären Ausdruck $R_P$ über der Menge der Ereignisse $\Sigma$ und eine *Phasenübergangsrelation* $\tilde{v} : \Sigma \times \mathcal{P}_M \rightarrow 2^{\mathcal{P}_M}$ spezifiziert. Die Übergangsrelation beschreibt die sequentielle Abfolge der Phasen. Die Zusammenfassung der Phasen ergibt somit die potenzielle Ereignisfolge einer Komponente. Aus den regulären Ausdrücken der Phasen $R_P$ können jeweils (nicht-deterministische) endliche Automaten $A_P$ für jede benannte Phase $P$ gebildet werden. Der gesamt Automat $A$ ergibt sich aus den (Teil-)Automaten durch die Verbindung der End- und Startzustände der (Teil-)Automaten gemäß der Phasenübergangsrelation $\tilde{v}$. Die Phasenübergangsrelation wird somit Teil der Übergangsrelation $\delta$ des nicht-deterministischen endlichen Automaten der vollständigen potenziellen Ereignisfolge.

Bildet man aus dieser Spezifikation der potenziellen Ereignisfolgen der Phasen einen deterministischen endlichen Automaten, so geht durch den Algorithmus im Allgmeinen die Information bezüglich der Phasen verloren, da bei der Überführung des nicht-deterministischen endlichen Automaten in einen determi-

nistischen endlichen Automaten Zustände als Mengen von Zuständen des nichtdeterministischen endlichen Automaten konstruiert werden und diese nicht mehr korrekt den Phasen zugeordnet werden können. Daher ist es notwendig, die Phasenübergänge deterministisch zu beschreiben, so dass die Übergänge zwischen den Phasen eindeutig spezifiziert sind. Ein Phasenübergang ist daher genau dann deterministisch, wenn die folgenden beiden Forderungen erfüllt sind:

a) Die Phasenübergangsrelation $\tilde{\upsilon}$ ist eine partielle Funktion, d. h. es gilt

$$\tilde{\upsilon} : \Sigma \times \mathcal{P}_M \rightarrow \mathcal{P}_M.$$

b) Es gibt keine Übergangsrelation $\delta$ im Automaten $A_P$ dergestalt, dass für ein $\sigma \in \Sigma$ und $\zeta' \in \phi(P)$ gilt

$$\delta(\sigma, \zeta') \neq \emptyset,$$

falls $\tilde{\upsilon}(\sigma, P)$ definiert ist.

Mit anderen Worten ist ein Phasenübergang dann deterministisch, wenn für ein Ereignis und einen Zustand des Automaten die nachfolgende Phase, sei es die bisherige oder eine andere Phase, eindeutig festgelegt ist.

Die Forderung nach deterministischen Phasenübergängen lässt sich erreichen, wenn man die Phasenübergangsrelation erweitert und den Zustand der Komponente $Z(C, \mathcal{E}_M, t)$ mit in die Relation einbezieht. Der Wechsel von einer Phase $Q$ in eine Phase $Q'$ ist nur dann zulässig, wenn die entsprechende Bedingung $B_\upsilon : Z(C, \mathcal{E}_M, t) \rightarrow \mathbb{B}$ erfüllt ist. Daraus ergibt sich die *bedingte Phasenübergangsfunktion* als $\upsilon : \Sigma \times \mathcal{P}_M \times \mathcal{C} \times \mathcal{E}_M \times T \rightarrow \mathcal{P}_M$. Falls die Bedingung für den Phasenwechsel nicht erfüllt ist, so verbleibt die Komponente in der bisherigen Phase. Existiert eine Übergangsfunktion im Teilautomaten für das entsprechende Ereignis, so wird diese angewandt, ansonsten ist das Ereignis an dieser Stelle unzulässig und muss entsprechend der Fehlersemantiken für unzulässige Ereignisse behandelt werden.

Diese Definition ist gleichbedeutend mit der Einführung eines neuen, von $\sigma$ $Id$-abhängigen Ereignisses $\sigma'$ mit der Ereignisbedingung $B_\upsilon$ und einer verwerfenden Fehlersemantik, wobei die bedingte Phasenübergangsfunktion $\upsilon$ zur unbedingten Phasenübergangsfunktion $\upsilon' : \Sigma/\sigma \cup \{\sigma'\} \times \mathcal{P}_M \rightarrow \mathcal{P}_M$ wird, mit

$$\upsilon'(\sigma', P) = \begin{cases} \upsilon(\sigma, P, C, E, t) & : \quad B_\upsilon(C, E, t) = wahr \\ \bot & : \quad \text{sonst} \end{cases}$$

Die Phasenübergangsfunktion $\upsilon'$ „simuliert“ mittels der Ereignisbedingung für $\sigma'$ somit die Bedingung der bedingten Phasenübergangsfunktion $\upsilon$. Da die Ereignisse $\sigma'$ verworfen werden, wenn die Ereignisbedingung nicht erfüllt ist, wird die Ereignisfolge innerhalb einer Phase nicht verändert, $\sigma'$ tritt dann und nur dann auf, wenn der Phasenwechsel zulässig ist.

**Lemma 5.3**
*Die bedingte Phasenübergangsfunktion $\upsilon$ ist deterministisch*

**Beweis**

Die erste Bedingung für den deterministischen Phasenübergang ist durch die Definition der bedingten Phasenübergangsfunktion offensichtlich erfüllt.

Da die nicht-bedingte Phasenübergangsfunktion $\upsilon'$ äquivalent zur bedingten Phasenübergangsfunktion $\upsilon$ ist, das „simulierende“ Ereignis $\sigma'$ aber kein Ereignis des Alphabets des endlichen Automaten ist, existiert kein $\delta(\sigma', \zeta') \neq \emptyset$. *q.e.d.*

**Beispiel 5.2**
Eine mögliche Anwendung für Phasen ist die Feststellung von Fehlerzuständen innerhalb einer Komponente. Mit dem Auftreten eines Fehlers, wechselt die Komponente in eine Fehlerphase. Wurde der Fehlerzustand durch das Management oder die Anwendung behoben, so kann die Komponente aus dem Fehlerzustand wieder in die Arbeitsphase zurückkehren. Tritt nun ein weiterer Fehler auf, während sich die Komponente in der Fehlerphase befindet, so darf die Komponente erst dann in die Arbeitsphase zurückkehren, wenn alle Fehler entsprechend behoben wurden. Der Phasenübergang aus der Fehlerphase in die Arbeitsphase ist also ein bedingter Phasenübergang, dessen Zulässigkeit von der Anzahl der noch nicht behobenen Fehler abhängt. Diese Situation ist in Abb. 5.3 dargestellt.

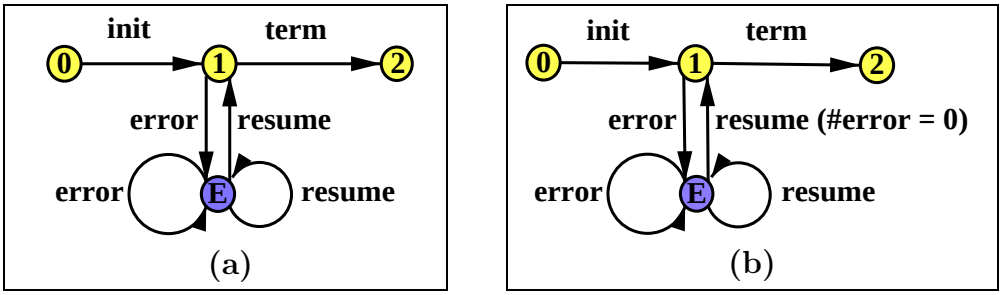



Abbildung 5.3: (a) Nicht-deterministischer und (b) bedingter, deterministischer Phasenübergang.

Für die Spezifikation sind die benannten Phasen als Strukturierungsmittel der zeitlichen Zusammenhänge wie folgt definiert:

**Definition 5.7 (Benannte Phasen)**
*Die Menge der benannten Phasen $\varphi$ eines Metatypen $M$ ist eine Menge von Tupeln der Form $\langle id, R, \upsilon \rangle$, wobei*

*a) $id$ der Bezeichner der Phase*

*b) $R$ der reguläre Ausdruck der potenziellen Ereignisfolgen innerhalb der Phase*

*c) $\upsilon$ die bedingte Phasenübergangsfunktion*

*ist.*

Der reguläre Ausdruck der Ereignisspezifikation nach Definition 5.5 auf Seite 84 ergibt sich dann aus den regulären Ausdrücken der einzelnen Phasen, sowie den Phasenübergangsfunktionen $\upsilon$. Für die Konstruktion des endlichen Automaten zur Erkennung der zulässigen Ereignisfolgen ist es allerdings notwendig, die bedingte Phasenübergangsfunktion $\upsilon$ durch die unbedingte Phasenübergangsfunktion $\upsilon'$ zu ersetzen und die Ereignismenge entsprechend zu erweitern.

Die potenziellen Ereignisfolgen und die Phasen, welche eine zusätzliche Strukturierung der potenziellen Ereignisfolgen ermöglichen, werden jeweils für einen nutzbaren Metatypen spezifiziert, da von abstrakten Metatypen keine Instanzen gebildet werden können. Analog zu den Ereignissen ist es aber sinnvoll, auch die Spezifikation von Phasen für abstrakte Metatypen zuzulassen, um die Spezifikation von Systemen zu vereinfachen. Da die Ereignisse in den Untertypen nicht identisch zu den Ereignissen der abstrakten Metatypen sind, muss allerdings eine Umbenennung der Ereignisse in den Phasen erfolgen. Die Ereignisse aus $\Sigma_M$ werden ersetzt durch die entsprechenden Ereignisse, gemäß $\leftarrow$, aus $\Sigma_{M'}$, in Zeichen $[\Sigma_M / \Sigma'_M]$. Damit ergibt sich folgender Zusammenhang zwischen der Vererbungshierarchie auf Metatypen und den Phasen:

$$M \leftarrow M' \Rightarrow \varphi_M[\Sigma_M / \Sigma'_M] \sqsubseteq \varphi'_M,$$

wobei

- $\langle id, R, \upsilon \rangle \approx \langle id', R', \upsilon' \rangle$
  $\Leftrightarrow$
  $id = id', L(R) \subseteq L(R'), \upsilon(\sigma, P, C, E, t) = \upsilon'(\sigma, P, C, E, t),$

  mit $\sigma \in \Sigma_M$, $P \in \mathcal{P}_M$, $M \rightsquigarrow C$, $E \in \mathcal{E}_M$ und $t \in T$, wobei $\langle id, R, \upsilon \rangle \in \varphi_M$.

- $\varphi \sqsubset \varphi'$
  $\Leftrightarrow$
  $\forall \langle id, R, v \rangle \in \varphi \ : \ \exists \langle id', R', v' \rangle \in \varphi' \ : \ \langle id, R, v \rangle \approx \langle id', R', v' \rangle$
- $\varphi = \varphi'$
  $\Leftrightarrow$
  $\forall \langle id', R', v' \rangle \in \varphi' \ : \ \exists \langle id, R, v \rangle \in \varphi \ : \ \langle id, R, v \rangle \approx \langle id', R', v' \rangle$

Die Phasen des Untertypen müssen eine Erweiterung der Phasen des Obertypen sein, d. h. alle zulässigen Ereignisfolgen des Obertypen müssen auch in dem entsprechenden Untertypen zulässig sein.

Für Ereigniszusammenhänge und Phasenwechsel werden folgende Schreibweisen verwendet:

1. **Ereigniszusammenhänge**
   - $\sigma'$ ist $R$-abhängig von $\sigma$: $\sigma \xrightarrow{R} \sigma'$
   - $\sigma'$ ist $Id$-abhängig von $\sigma$: $\sigma \rightarrow \sigma'$
2. **Phasenwechsel**
   - unbedingter Phasenwechsel mit dem Ereignis $\sigma$ in die Phase $p$: $\sigma \Rightarrow p$
   - bedingter Phasenwechsel mit dem Ereignis $\sigma$ in die Phase $p$ unter der Bedingung $C$: $\sigma \Rightarrow^{[C]} p$

## 5.3 Semantik von Ereignisfolgen

Die Bedeutung der Ereignisfolgen und ihrer Strukturierung in Phasen besteht zum Einen in der Integration des zeitlichen Verlaufs der Lebenszeit einer Komponente in den Zustand der Komponente. Diese Integration ermöglicht die Nutzung dieser Information durch das Management.

Zum Anderen kann die Spezifikation der Ereignisfolgen aber auch zur Überprüfung der Konsistenz einer Spezifikation und der Vorhersage der Lebenszeit von Komponenten dienen. Damit diese Vorhersagen korrekt getroffen werden können, müssen die in den Ereignisfolgen und Ereignisbedingungen festgelegten Restriktionen innerhalb eines Systems durchgesetzt werden. Diese Durchsetzung wird als Synchronisation bezeichnet.

### 5.3.1 Konsistente Ereignisfolgen

Die Spezifikation der Ereignisfolgen basiert auf den Metatypen, aus welchen zur Laufzeit eines Systems die Komponenten des Systems abgeleitet werden. Diese Sichtweise ist insofern einschränkend, als dass die Zusammenhänge zwischen den Ereignisfolgen der Komponenten nur indirekt über die Spezifikation der Ereigniszusammenhänge gegeben ist. Eine Ereignisspezifikation muss aber global konsistent sein, d. h. die Ereigniszusammenhänge müssen erfüllbar sein.

**Definition 5.8 (Konsistente Ereignisfolgen)**
*Sei $\sigma R$ ein regulärer Ausdruck zur Beschreibung der Ereignisfolgen der Komponente eines Metatypen $M$ und $\sigma' R'$ ein regulärer Ausdruck zur Beschreibung der Ereignisfolgen der Komponente eines Metatypen $M'$.*

*Die Ereignisfolgen $\sigma R$ und $\sigma' R'$ heißen zueinander konsistent, wenn für jede potenzielle Ereignisfolge gemäß $\sigma R$ gilt, dass es eine potenzielle Ereignisfolge gemäß $\sigma' R'$ gibt, in welcher die abhängigen Ereignisse in genau der Reihenfolge der auslösenden Ereignisse in $\sigma R$ auftreten.*

*Eine Ereignisspezifikation heißt global konsistent, wenn alle potenziellen Ereignisfolgen zueinander konsistent sind.*

Für die Überprüfung der Konsistenz von Ereignisfolgen ist es hilfreich, die Ereignisfolgen zunächst zu vereinfachen. Da für die Konsistenz nur die abhängigen Ereignisse und deren auslösende Ereignisse von Interesse sind, werden die potenziellen Ereignisfolgen reduziert, indem alle unabhängigen Ereignisse aus $\sigma R$ bzw. $\sigma' R'$ entfernt werden. Diese reduzierten potenziellen Ereignisfolgen werden im Folgenden als $r$ und $r'$ bezeichnet, wobei die abhängigen Ereignisse in $r'$ durch ihre Auslöser ersetzt werden.

Die potenziellen Ereignisfolgen $r$ und $r'$ sind genau dann konsistent, wenn die „Sprache“ des regulären Ausdrucks $r'$ Teilmenge der „Sprache“ des regulären Ausdrucks $r$ ist, also wenn $L(r') \subseteq L(r)$ gilt. Diese Bedingung lässt sich anhand der deterministischen endlichen Automaten für $r$ und $r'$ überprüfen, da

a) $L(r') \subseteq L(r) \Leftrightarrow L(r') \cap L(r) = \overline{(\overline{L(r')} \cup \overline{L(r)})} = L(r')$

b) Zwei reguläre Sprachen genau dann gleich sind, wenn die Minimalautomaten bis auf Umbenennungen gleich sind.

Die Konstruktion eines negierten Automaten, sowie die Vereinigung zweier Automaten sind triviale Aufgabenstellungen der Automatentheorie.

Eine weitere für das Management, insbesondere die Generierung des Managements, interessante Fragestellung lässt sich ebenfalls anhand der reduzierten potenziellen Ereignisfolgen entscheiden: Falls eine Komponente $C$ in einer bestimmten minimalen Phase $P$, d. h. in einem bestimmten Zustand des Automaten, ist, in welcher Menge von minimalen Phasen kann sich eine, durch die Relation $R$ verbundene Komponente $C'$ befinden?

Bildet man für jede minimale Phase $Q$ der Komponente $C'$ die Sprache der Präfixe, d. h. die Sprache der potenziellen Ereignisfolgen, welche in die minimale Phase $Q$ führen und reduziert diese auf die $R$-abhängigen Ereignisse, so erhält man für jede minimale Phase $Q$ einen regulären Ausdruck für die potenziellen Ereignisfolgen, die in $Q$ führen. Der reguläre Ausdruck für das Präfix der potenziellen Ereignisfolgen bis $Q$ sei $r_Q$, wobei wiederum die abhängigen Ereignisse durch ihre jeweiligen Auslöser ersetzt werden.

Gleichermaßen wird für die minimale Phase $P$ das entsprechende Präfix $r_P$ gebildet. Die Komponente $C'$ kann sich in allen minimalen Phasen $Q$ befinden, für die gilt:

$$L(r_Q) \subseteq L(r_P)$$

Es muss bei einer konsistenten Ereignisspezifikation mindestens ein solches $Q$ existieren. Da sich die vergröberten Phasen einer Komponente als Zusammenfassung minimaler Phasen ergeben, sind somit Aussagen über alle Phasen einer Komponente möglich.

Damit sind anhand der Spezifikation weitreichende Aussagen über die zeitlichen Zusammenhänge zwischen den Komponenten möglich, bevor ein System sich in Ausführung befindet, sogar bevor eine Beschreibung des Systems in Form eines Problemlösungsverfahrens vorliegt. Die Aussagen werden nur aufgrund der möglichen Komponenten, d. h. auf der Basis der Metatypen getroffen. Dies ermöglicht die Analyse von Zusammenhängen bereits bei der Erstellung des Managements durch einen generativen Prozess, aber auch die Nutzung dieser Information in den Spezifikationen für das Management.

### 5.3.2 Synchronisation durch Ereignisspezifikation

Neben der Vorhersage der potenziellen Abläufe innerhalb eines Systems ist die Synchronisation von Abläufen innerhalb eines nebenläufigen, verteilten Systems eines der zentralen Ziele der formalen Spezifikation der Ereignisfolgen. Als Synchronisation bezeichnet man im Allgemeinen die Steuerung asynchroner Abläufe mit dem Ziel, eine wohldefinierte Präzedenz der Verarbeitungsschritte zu realisieren. Durch die Festlegung zulässigen Abläufe eines Systems durch Ereignisfolgen

und Ereignisbedingungen ergibt sich für das Management die Aufgabe der Durchsetzung dieser Restriktionen.

Die zu synchronisierenden Abläufe sind dabei allerdings funktionale Abläufe und somit Teil der Anwendungen und nicht des Managements. Durch Auslösen eines Ereignisses innerhalb der Anwendung wechselt ein Ausführungsfaden aus dem Kontext der Anwendung in den Kontext des Managements. Das Auftreten eines Ereignisses innerhalb des Managements führt durch die Ereignisabhängigkeiten entlang der Relationen zu einer Kette von Ereignissen. Der Ausführungfaden verlässt den Kontext des Managements, wenn die Zustandsänderungen aufgrund der Ereignisse einer solchen Kette eingetreten sind (vgl. Abb. 5.4).

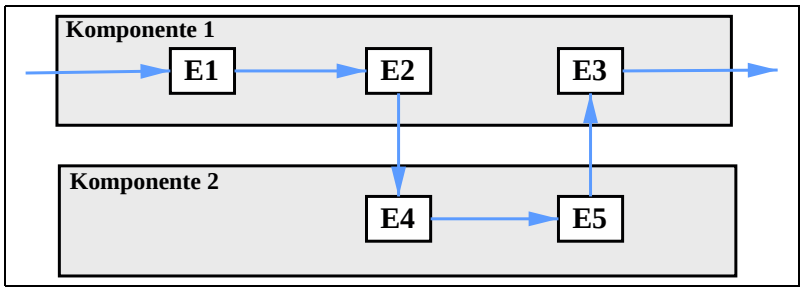



Abbildung 5.4: Kontextwechsel eines Ausführungsfadens.

Für das Management treten Ausführungsfäden nur dann in Erscheinung, wenn durch ein Anwendungsereignis in den Kontext des Managements gewechselt wird. Die Ausführungsfäden an sich sind in der Spezifikation des Managements nicht beschrieben, sondern treten innerhalb des Managements nur implizit auf. Eine explizite Spezifikation der Ausführungsfäden ist einerseits nicht notwendig, da die Synchronisation auf der Basis der Ereignisfolgen und -bedingungen spezifiziert wird. Andererseits sind die Ausführungsfäden Ressourcen der Anwendung, die selbst durch Komponenten repräsentiert werden. Eine Integration der Ausführungsfäden in das Management würde daher zu einer nicht notwendigen Einschränkung der Flexibilität führen.

Werden innerhalb eines Ausführungsfadens zu einem Zeitpunkt mehrere Ereignisse ausgelöst, beispielsweise durch die spezifizierten Ereigniszusammenhänge, so ist es notwendig, diese Ereignisse geeignet zu sequentialisieren, da ein Ausführungsfaden nur maximal ein Ereignis zu einem Zeitpunkt ausführen kann. Da diese Sequentialisierung Einfluss auf die Abläufe innerhalb des Managements hat, muss, wenn das Management beherrschbar sein soll, die Reihenfolge der Sequentialisierung festlegbar sein. Daher wird für die Sequentialisierung der zusammenhängenden Ereignisse die Aufschreibungsreihenfolge als Grundlage verwendet.

Wird innerhalb eines Ausführungsfadens ein nicht zulässiges Ereignis ausgelöst, sei es innerhalb der Anwendung oder durch die Ereigniszusammenhänge, so muss das Management entsprechend der Fehlersemantik für das entsprechende Ereignis reagieren. Wird das Ereignis verworfen, wird mit der Abarbeitung der weiteren Ereignisse innerhalb der Kette fortgefahren oder, falls keine weiteren Ereignisse ausgelöst wurden, so kehrt der Ausführungsfaden in die Anwendung zurück.

Wurde hingegen ein Ereignis mit blockierender Fehlersemantik ausgelöst, so muss das Management sicherstellen, dass der Ausführungsfaden angehalten wird, bis das Ereignis zulässig ist. Das Eintreten der Zulässigkeit des Ereignisses kann nur durch ein von einem weiteren Aktivitätsfaden ausgelöstes Ereignis herbeigeführt werden (vgl. Abb. 5.5). Damit ergibt sich die Möglichkeit zur Synchronisation von Ausführungsfäden durch die Festlegung von Ereignisreihenfolgen und Ereignisbedingungen.

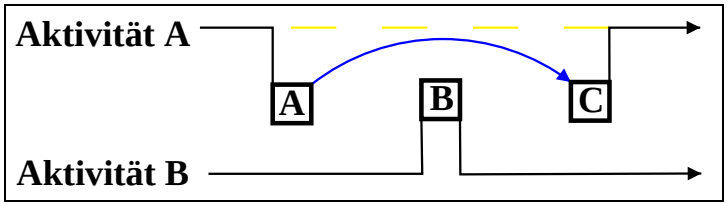



Abbildung 5.5: Blockieren durch Ereignisse.

Die Mächtigkeit dieses Ansatzes ergibt sich dabei aus den Ereignisbedingungen, welche den Zustand der Komponenten in die Synchronisation einbeziehen. Der Zusammenhang zwischen den Zuständen einer Komponente bzw. den Zuständen eines Systems wird in Kapitel 6 auf Seite 107 ausführlich diskutiert.

### 5.3.3 Vergleich zu anderen Synchronisationstechniken

Neben den hier vorgestellten Techniken zur Spezifikation von Synchronisationsbedingungen existieren eine Reihe weiterer Synchronisationsmechanismen und -techniken. Für einen Vergleich des hier vorgestellten Ansatzes zur Synchronisation mit bekannten Mechanismen sind nur Techniken mit hoher Abstraktion von Interesse, mittels derer Synchronisationsbedingungen spezifiziert werden können. Bei diesen Vergleichen ist stets zu beachten, dass die hier diskutierten Ansätze der Spezifikation von Synchronisationsbedingungen auf der Ebene des Managements stattfinden, d. h. ohne Kenntnis des konkreten Systems, die bekannten Synchronisationskonzepte aber für die Spezifikation von Systemen und die Integration in programmiersprachliche Umgebungen entwickelt wurden. Ein zentraler Unterschied zwischen dem in dieser Arbeit beschriebenen Ansatz und den

bekannten Ansätzen, der sich aus dieser Tatsache ergibt, liegt in der Granularität der Synchronisationsbedingungen. Die üblichen Synchronisationskonzepte basieren auf den ablauforientierten Einheiten, welche die Programmiersprachen zur Verfügung stellen, also auf Prozeduren, Methoden und vergleichbaren Konzepten. Mit dem hier vorgestellte Ansatz dagegen werden Synchronisationsbedingungen auf der Basis von Ereignissen beschrieben. Eine Prozedur beispielsweise erzeugt im Allgmeinen mindestens zwei Ereignisse, das Startereignis *start* und das Terminierungsereignis *term*.

Trotz dieses fundamentalen Unterschieds soll hier ein Vergleich mit zwei bekannten Techniken zur Synchronisation versucht werden, einerseits mit dem Monitorkonzept (siehe [Hoa74] und [BH75]), andererseits mit dem Konzept der *Path Expressions* bzw. der Variante der *Predicate Path Expressions* (siehe [And79], [FH76] und [Ree93]).

### Monitore

Monitore sind eine Möglichkeit, Synchronisation auf einer höheren Abstraktionsebene zu beschreiben. Ein Monitor kapselt Prozeduren, Variablen und Datenstrukturen und ermöglicht die Synchronisation dieser Elemente, indem stets nur ein Ausführungsfaden auf den Monitor zugreifen kann. Monitore sind ein typisches Beispiel für Synchronisationskonzepte, die in Programmiersprachen integriert werden können. Aufgrund der Idee der Kapselung von Prozeduren, Variablen und Datenstrukturen in Monitoren kann dieses Konzept beispielsweise in objektorientierte Sprachen, wie Java (vgl. [GJG96]) integriert werden.

Um Wartebedingungen realisieren zu können, wurden Zustandsvaribalen in die Monitore eingeführt. Über diese Zustandsvariablen können Ausführungsfäden koordiniert werden. Diese Zustandsvariablen sind allerdings keine Zähler oder komplexe Datenstrukturen. Es existieren nur zwei Operationen auf diesen, nämlich *wait* um einen Ausführungsfaden zu blockieren und *signal*, um diesen wieder zu reaktivieren. Somit ist es notwendig, dass der Programmierer die Zustandsvaribalen mittels der gegebenen Operationen korrekt einsetzt.

Das hier vorgestellte Konzept zur Spezifikation von Synchronisation ähnelt dem Konzept der Monitore insofern, dass einerseits stets nur ein Ausführungsfaden auf die Eigenschaften einer Komponente zugreifen kann. Diese Tatsache ergibt sich aus der Wirkungsatomarität der Ereignisse. Der Zustand der Komponenten, d. h. die Belegung der Eigenschaften, ist somit stets konsistent.

Eine andere Verwandtschaft zwischen Monitoren und dem hier vorgestellten Konzept liegt in der Formulierung von Synchronisationsbedingungen durch Zustände. Allerdings ist das hier vorgestellte Konzept dafür weitaus mächtiger, da über die Eigenschaften der Komponenten beliebige boolsche Bedingungen für die Zulässigkeit von Ereignissen formuliert werden können.

Der zentrale Unterschied zwischen Monitoren und dem hier vorgestellten Ansatz liegt in der expliziten Festlegung der Reihenfolge von Ereignissen. Bei Monitoren ist die Reihenfolge der Ereignisse nur implizit über die Zustandsvariablen festgelegt. Ein höheres Konzept zur Beschreibung der Reihenfolge ist nicht enthalten, für die Spezifikation des Managements ist diese aber von zentraler Bedeutung.

### Predicate Path Expressions

Pfadausdrücke sind ein anderer bekannter Ansatz zur Beschreibung von Synchronisationsbedingungen. In Pfadausdrücken wird ähnlich zu den regulären Ausdrücken für die potenziellen Ereignisfolgen die Reihenfolge von Operationen festgelegt. Aufgrund der Tatsache, dass durch die Festlegung der zulässigen Reihenfolgen von Operationen nicht alle Synchronisationsprobleme gelöst werden können, wurden die Path Expressions um Prädikate zu den Predicate Path Expressions erweitert.

Die Prädikate ermöglichen die Formulierung von Bedingungen, unter denen ein bestimmter Pfad, d. h. ein Teil der festgelegten Reihenfolgen, betreten werden darf. Insofern entsprechen die Prädikate der Predicted Path Expressions den hier vorgestellten Ereignisbedingungen bzw. den bedingten Phasenwechseln. Allerdings sind die Möglichkeiten zur Formulierung von Prädikaten in den Ausdrücken stark eingeschränkt, da diese nur auf Ereigniszählern, welche implizit mit den Operationen verknüpft sind, formuliert werden können. Ein Zugriff auf den Zustand des Systems oder der zu synchronisierenden Einheit sind nicht vorgesehen.

Die hier vorgestellte Technik zur Spezifikation von Synchronisationsbedingungen kann insofern als Erweiterung der Predicated Path Expressions betrachtet werden, als dass die Ereigniszähler Teil der Eigenschaften der Komponenten sind, aber aufgrund des allgemeinen Zugriffs auf die Eigenschaften der Komponenten und somit auf den Zustand der Komponenten bzw. des Systems ein mächtigeres Instrumentarium zur Formulierung von Bedingungen bzw. Prädikaten gegeben ist.

Ein weiterer wesentlicher Unterschied ergibt sich aus der Tatsache, dass bei den Predicated Path Expressions den Operationen implizit Ereignisse zugeordnet werden. Diese implizite Zuordnung hat nicht immer die gewünschte Granularität. Diese ist nur erreichbar, wenn die Ereignisse frei spezifiziert werden können.

Dieser kurze Vergleich der hier vorgestellten Technik zur Spezifikation von Synchronisationsbedingungen macht einerseits deutlich, dass andere Ansätze zur Synchronisation als Spezialfälle in dem hier vorgestellten Ansatz enthalten sind, dieser aber einerseits eine größere Mächtigkeit, andererseits ein höheres Abstraktionsniveau aufweist. Letzteres ist in der Tatsache begründet, dass Synchronisation

hier auf der Ebene des Managements beschrieben wird und nicht in der Anwendung.

### 5.3.4 Standardprobleme der Synchronisation

Im Folgenden werden einige Standardprobleme der Synchronisation beispielhaft mit den im vorherigen Abschnitt vorgestellten Mitteln diskutiert. Anhand dieser Beispiele wird die Ausdrucksstärke des Ansatzes verdeutlicht.

#### Leser-Schreiber-Problem

Das Leser-Schreiber-Problem simuliert den konkurrierenden Zugriff auf eine Datenbank, wobei der schreibende Zugriff exklusiv erfolgen muss, während durchaus mehrere Prozesse lesend auf die Datenbank zugreifen dürfen (siehe [CHP71]).

Da nur das Management spezifiziert wird, bzw. die für ein System zur Verfügung stehenden Sprachkonzepte und abstrakten Ressourcen, werden hierbei nicht die Leser und Schreiber spezifiziert, sondern die (abstrakte) Ressource „Datenbank“.

Die Menge der Ereignisse wird als

$$\Sigma = \{init, read, ready, write, done\}$$

festgelegt. Das Ereignis *init* ist dabei das Startereignis der Komponenten. Die Menge der benannten Phasen $\varphi$ enthält die Elemente:

- $pre : init \Rightarrow free$
  Nach dem Startereignis wird unbedingt in die Phase $free$ übergegangen.
- $free : write \Rightarrow writing \mid read \Rightarrow reading \mid done \Rightarrow final$
  In der Phase $free$ ist eines der Ereignisse *write*, *read* oder *done* zulässig. Mit jedem dieser Ereignisse wird in die entsprechende Phase *writing*, *reading* oder $final$ gewechselt.
- $writing : ready \Rightarrow free$
  In der Phase *writing* ist nur das Ereignis *ready* zulässig, mit welchem ein Schreiber anzeigt, dass der Schreibzugriff beendet ist.
- $reading : (read|ready \Rightarrow^{[NumReader=0]} free)*$
  In der Phase *reading* sind entweder weitere *read*-Ereignisse oder *ready*-Ereignisse zulässig. Mit dem bedingten Phasenübergang wird, sobald die Anzahl der aktiven Leser Null wird, in die Phase $free$ gewechselt.
- $final$
  In der Phase $final$ sind keine weiteren Ereignisse zulässig. Diese Phase beendet die Komponente. Da nur mit dem *done* Ereignis in der Phase $free$

in $final$ übegegangen werden kann, ist das Ereignis *done* das Terminierungsereignis dieses Metatypen und kann nur dann auftreten, wenn kein Leser bzw. Schreiber die Datenbank noch nutzt.

Dabei ist $NumReader$ eine Eigenschaft, welche die Anzahl der aktiven Leser im Puffer repräsentiert. Diese verändert sich mit den Ereignissen *read* bzw. *ready*.

Eliminiert man den bedingten Phasenübergang durch die Einführung eines neuen, von *read* $Id$-abhängigen und bedingten Ereignis $freeIt$ mit der Bedingung $NumReader = 0$ und einer verwerfenden Fehlersemantik, so wird die Phase *reading* zu

$$reading : (read|ready) * freeIt \Rightarrow free.$$

Der entsprechende deterministische endliche Automat für diese Variante der potenziellen Ereignisfolgen ist in Abb. 5.6 dargestellt.

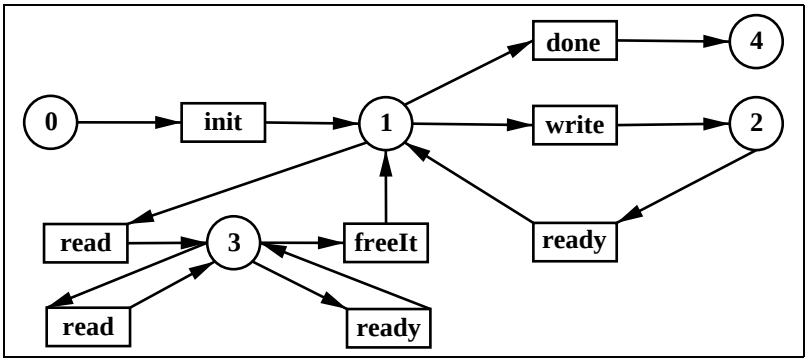



Abbildung 5.6: Potenzielle Ereignisfolge des Leser-Schreiber-Problems.

Diese Lösung des Leser-Schreiber-Problems ist allerdings „unfair“, da die Möglichkeit besteht, dass ein Schreiber „verhungert“, d. h. niemals Zugriff auf die Datenbank erhält. Daher wird im Folgenden eine „faire“ Variante des Leser-Schreiber-Problems präsentiert.

### Leser-Schreiber-Problem – Mit Fairness

Fairness kann in die bestehende Lösung des Leser-Schreiber-Problems eingeführt werden, indem die Leser bzw. Schreiber zunächst Anforderungsereignisse erzeugen, die dem Management mitteilen, dass unerfüllte Lese- bzw. Schreibanforderungen ausstehen.

Daher wird Menge der Ereignisse zu

$$\Sigma = \{init, read, doRead, ready, write, doWrite, done\}$$

erweitert. Dabei ist wiederum *init* das Startereignis und *done* das Terminierungsereignis.

Die Phasen $\varphi$ des Metatypen können wie folgt spezifiziert werden:

- $pre : init \Rightarrow free$
- $free : doWrite \Rightarrow writing \mid doRead \Rightarrow reading \mid done \Rightarrow final$
- $writing : ready \Rightarrow free$
- $reading : (doRead|ready \Rightarrow^{[NumReader=0]} free)*$
- $final :$

Die potenziellen Ereignisfolgen der einzelnen Phasen unterscheiden sich – bis auf die geänderten Ereignisnamen – nicht von den potenziellen Ereignisfolgen der vorherigen Lösung, die Ereignisse *read* und *write* sind allerdings in diesem Fall asynchron. Der zentrale Unterschied besteht in der Einführung von Ereignisabhängigkeiten und Ereignisbedingungen:

1. **Ereignisbedingungen**

   - $read : wantWrite = false$
   - $write : wantRead = false$

   Die Belegung der Eigenschaften $wantWrite$ und $wantRead$ repräsentieren jeweils den Wunsch eines Schreibers bzw. eines Lesers nach Zugriff auf die Datenstruktur. Die Ereignisse sind nur dann zulässig, wenn der entsprechende Wunsch im Augenblick nicht vorliegt bzw. bereits erfüllt wurde.

2. **Ereignisabhängigkeiten**

   - $read \rightarrow doRead$
   - $write \rightarrow doWrite$

   Die Ereignisse *doRead* und $doWrite$ sind jeweils $Id$-abhängig von *read* und *write*. Ist ein Lese- bzw. ein Schreibereignis zulässig, so wird das entsprechende Ereignis erzeugt. Diese Ereignisse selbst werden dann gemäß den Regeln für den wechselseitigen Ausschluss synchronisiert.

### Erzeuger-Verbraucher-Problem

Das Erzeuger-Verbraucher-Problem ist ein weiteres bekanntes Synchronisationsproblem, welches in der Praxis häufig anzutreffen ist. Die Aufgabenstellung gibt einen Puffer statisch begrenzter Größe ($max$) vor. Einerseits greifen Erzeuger auf diesen zu, die den Puffer füllen möchten, andererseits werden die Elemente

des Puffers von den Verbrauchern entnommen. Praktische Bedeutung hat dieses Beispiel unter anderem bei der Nachrichtenkommunikation.

Die Erzeuger dürfen dabei nur solange auf den Puffer zugreifen, wie der Puffer nicht vollständig gefüllt ist, die Verbraucher dürfen nur dann auf den Puffer zugreifen, falls dieser nicht leer ist. Damit der Puffer stets in einem konsistenten Zustand ist, dürfen die Erzeuger und Verbraucher nur wechselseitig ausgeschlossen den Puffer nutzen.

Aus diesen Anforderungen ergibt sich zunächst für den wechselseitigen Ausschluss eine Lösung analog zur Lösung des Leser-Schreiber-Problems. Die Ereignismenge wird wiederum als

$$\Sigma = \{init, read, write, ready, done\}$$

festgelegt, mit dem Startereignis *init* und dem Terminierungsereignis *done*. Die Menge der Phasen des Metatyps $\varphi$ unterscheidet sich im Wesentlichen nicht von den Phasen des Leser-Schreiber-Problems:

- $init : \Rightarrow free$
- $free : write \Rightarrow writing \mid read \Rightarrow reading \mid done \Rightarrow final$
- $writing : ready \Rightarrow free$
- $reading : ready \Rightarrow free$
- $final :$

Der zugehörige deterministische endliche Automat ist in Abb. 5.7 dargestellt.

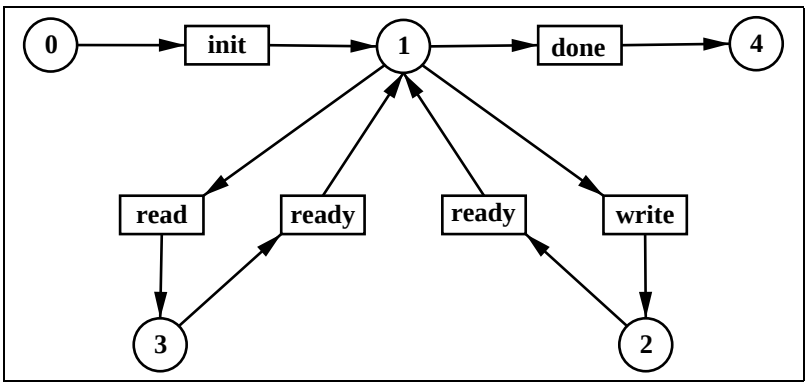



Abbildung 5.7: Potenzielle Ereignisfolge des Erzeuger-Verbraucher-Problems.

Die Durchsetzung der Puffergröße durch das Management wird durch Ereignisbedingungen und eine Eigenschaft *filled*, deren Belegung im Intervall $[0; max]$ den Füllstand des Puffers beschreibt, spezifiziert:

- $read : filled > 0$
- $write : filled < max$

Ein Leseereignis darf nur dann auftreten, wenn der Puffer mindestens ein Element enthält, ein Schreibereignis nur dann, wenn die maximale Füllung des Puffers noch nicht erreicht wurde. Dabei sind sowohl Verhungern als auch Verklemmungen bei dieser Lösung ausgeschlossen.

## 5.4 Zusammenfassung

Ereignisse sind die Auslöser von Zustandsveränderungen in Systemen. In der Spezifikation werden diese Auslöser auf der Ebene der Metatypen unabhängig von den zur Laufzeit existierenden Komponenten beschrieben. Durch die Festlegung von Restriktionen bezüglich der Zulässigkeit von Ereignissen kann das Management Synchronisationsbedingungen durchsetzen, aber auch die Ereignisfolgen analysieren, sowie die Konsistenz der Folgen bereits vor der Existenz eines Systems bzw. sogar vor der Existenz einer Beschreibung des Systems überprüfen.

In diesem Kapitel wurde ein Ansatz zur Spezifikation der potenziellen Ereignisfolgen durch reguläre Ausdrücke und Ereignisbedingungen beschrieben. Die potenziellen Ereignisfolgen können in Phasen strukturiert werden, wodurch eine Vergröberung möglich wird. Über die Spezifikation der Ereignisfolgen und -bedingungen ist auch die Spezifikation von Synchronisation auf abstrakter Ebene möglich.

Dabei ist es wesentlich, die „Qualität“ einer Methodik zur Spezifikation zu untersuchen und anhand gewisser Kriterien zu analysieren.

In [Blo79] werden unterschiedliche Ansätze zur Beschreibung von Synchronisation gegenübergestellt. In dieser Arbeit wurden zwei Kriterien für die Qualität einer Spezifikationsmethode als zentrale Kriterien festgelegt:

1. **Die Einfachheit der Verwendung**
   Eine Methodik zur Spezifikation, insbesondere ein Ansatz auf höherer Abstraktionsebene, sollte einfach zu verwenden sein. Dies betrifft einerseits die Komplexität der Aufschreibung, andererseits die Verständlichkeit. Beide Aspekte sind notwendig, um die Wahrscheinlichkeit von Spezifikationsfehlern zu minimieren.

2. **Die Ausdrucksstärke**
   Jeder Ansatz zur Spezifikation sollte die Möglichkeit eröffnen, alle Anforderungen bezüglich der Spezifikation zu beschreiben. Können bestimmte

Synchronisationsprobleme nicht oder nur teilweise beschrieben werden, so fehlt dem Ansatz die notwendige Ausdrucksstärke.

Die Festlegung, Ereignisfolgen mittels regulärer Ausdrücke zu beschreiben, mag, aufgrund der mangelnden Ausdrucksstärke, zunächst verwundern, da durch Zusatzbedingungen dieser Mangel wieder behoben werden muss. In [DH86] wird ein Ansatz zur Beschreibung von Ereignisfolgen durch kontextfreie Sprachen beschrieben. Auch dieser Ansatz zeigt, dass kontextfreie Sprachen nicht ausreichen, um die potenziellen Ereignisfolgen zu beschreiben. Die dort präsentierte Lösung basiert auf dem Schnitt mehrerer kontextfreier Sprachen. Dieser Ansatz bereitet allerdings Probleme bei der statischen Analyse der potenziellen Ereignisfolgen. Andererseits ist die Komplexität der Spezifikation erheblich, so dass das Kriterium der einfachen Verwendung bei diesem Ansatz als nicht erfüllt angesehen werden muss.

Die regulären Ausdrücke dagegen sind ein Instrumentarium, welches einerseits gut untersucht ist und für das eine Reihe effizienter Algorithmen zur Verfügung steht. Anderseits sind reguläre Ausdrücke intuitiv zu verstehen und somit einfach anwendbar. Die – zunächst als Nachteil erscheinende – Notwendigkeit der Verwendung von Zusatzbedingungen ist bei genauerer Betrachtung eine Methodik der Spezifikation, welche der üblichen informellen Beschreibung von Synchronisationsbedingungen sehr nahe kommt. Daher sind diese Zusatzbedingungen einfach anwendbar und ermöglichen eine Ausdrucksstärke, die mit vergleichbaren Ansätzen, wie den Predicated Path Expressions nur schwer erreichbar ist. Dieser Zuwachs in der Ausdrucksstärke basiert auf der Möglichkeit, den Zustand der Komponenten und somit des Systems in die Bedingungen einzubeziehen.

Somit ergibt sich für die vorgestellte Methode zur Spezifikation von zeitlichen Abhängigkeiten in Systemen einerseits eine einfache und an der Semantik orientierte Beschreibung auf einem hohen Abstraktionsniveau. Andererseits steht mit dieser Form der Spezifikation eine Ausdrucksmächtigkeit zur Verfügung, welche es ermöglicht, alle Restriktionen bezügliche der Reihenfolge von Ereignissen zu beschreiben

# 6 Spezifikation der Eigenschaften abstrakter Systeme

Mit den Festlegungen aus den vorherigen beiden Kapiteln ist es möglich, die Eigenschaften von Komponenten sowie die Struktur bezüglich der Lebenszeit von Komponenten abstrakt zu beschreiben. In diesem Kapitel wird das Zusammenwirken dieser Charakteristika der Komponenten, sowie die Möglichkeiten, daraufaufbauend ein Management für ein System zu generieren, beschrieben.

Ein System in Ausführung ist durch Zustandsänderungen charakterisiert. Diese Veränderungen betreffen einerseits den Zustand, d. h. die Belegung der Eigenschaften der Komponenten des Systems, andererseits aber auch die Menge der Komponenten, die zu einem Zeitpunkt im System vorhandenen sind. Die Zustände werden durch Elemente aus Datensorten repräsentiert. Für die Beschreibung der Berechnung der Zustände ist daher zunächst eine Rechenstruktur zu definieren, welche die Berechnung von Belegungen der Eigenschaften ermöglicht.

## 6.1 Eine Rechenstruktur für Zustandsberechnungen

Im Abschnitt 4.2.3 auf Seite 61 wurden die Datensorten des Managements eingeführt. Die Rechenstruktur, die für die Berechnung der Belegungen der Eigenschaften benötigt wird, beschreibt die auf diesen Daten möglichen Operationen. Diese Rechenstruktur ist daher definiert als Tupel $\langle \mathcal{D}, \mathcal{O} \rangle$, wobei $\mathcal{O}$ eine Menge von Operationen auf $\mathcal{D}$ darstellt und wird als Datentyp bezeichnet. Die Elemente $O$ von $\mathcal{O}$ sind also Funktionen der Form $O : \mathcal{D}^n \times \mathcal{D}$, wobei $n$ als die *Stelligkeit* der Operation $O$ bezeichnet wird. Im Folgenden werden die Elemente von $\mathcal{O}$ für eine Spezifikation festgelegt. Die Festlegung der Rechenoperationen folgt der Definition der vollständigen Datensorten 4.8 auf Seite 62. Zunächst wird für jede Datensorte $D \in \mathcal{D}$ der Gleichheitsoperator mit der üblichen Semantik vorrausgesetzt:

$$= \quad : \quad D \times D \rightarrow I\!B$$

1. **Operationen auf einfachen Datensorten**
   Die einfachen Datensorten entsprechen den Sorten der ausführenden Hardware. Wie in Abschnitt 4.4 auf Seite 67 dargestellt, ist die Definition der Hardware von der jeweiligen Spezifikation abhängig. Damit sind auch die zur Verfügungstehenden einfachen Datensorten, und somit die Operationen darauf, von der Spezifikation abhängig.

   Allerdings wird ein Datentyp $\mathbb{B}$ für die Repräsentation der boolschen Werte vorausgesetzt, mit dem Wertebereich $\{wahr, falsch, \perp\}$, sowie den Operationen mit den üblichen Gesetzen:

$$\begin{array}{rcll} wahr & : & & \rightarrow \mathbb{B} \\ falsch & : & & \rightarrow \mathbb{B} \\ \neg & : & \mathbb{B} & \rightarrow \mathbb{B} \\ \wedge & : & \mathbb{B} \times \mathbb{B} & \rightarrow \mathbb{B} \\ \vee & : & \mathbb{B} \times \mathbb{B} & \rightarrow \mathbb{B} \end{array}$$

2. **Zeit**
   Die Sorte $T$ repräsentiert die Zeit im System. Für diese Sorte stehen folgende Operationen zur Verfügung:

$$\begin{array}{rcll} 0 & : & & \rightarrow T \\ now & : & & \rightarrow T \\ succ & : & T & \rightarrow T \\ pred & : & T & \rightarrow T \\ < & : & T \times T & \rightarrow \mathbb{B} \\ > & : & T \times T & \rightarrow \mathbb{B} \end{array}$$

   Dabei gelten folgende Gesetze:

   - $t \geq 0 = wahr$
   - $succ(t) > t = wahr$
   - $pred(t) \leq t = wahr$
   - $now > 0 = wahr$
   - $now \geq now = wahr$

   Die Funktion *now* gibt jeweils die aktuelle Zeit wieder. Zwei Aufrufe von *now* liefern daher nicht zwingend identische Ergebnisse, die Funktion ist allerdings monoton wachsend. Alle Werte $t$ mit $t < now$ werden als *Vergangenheit* bezeichnet, alle Werte mit $t > now$ als *Zukunft*. Die Zuordnung von Werten zur Zukunft bzw. der Vergangenheit ist nicht statisch, da diese von der Funktion *now* abhängig ist.

**3. Metatypen**

a) Jeder Metatyp $M$ bildet eine Sorte deren Wertebereich die Komponenten dieses Typs umfasst. Die Menge $\mathcal{E}_M$ sind die Eigenschaften des Metatypen $M$. Für jede Eigenschaft $E \in \mathcal{E}_M$ existiert eine Operation $val_E \in \mathcal{O}$ mit

$$val_E : M \times T \to dom(E), \text{ mit } val_E(C,t) = \tilde{Z}(C,E,t)$$

Im Folgenden wird für die Operation $val_E(C, now)$ abkürzend $C \vdash E$ geschrieben.

b) Für jeden Metatyp wird eine Eigenschaft *self* festgelegt, welche konstant mit der Komponente belegt ist. Zu jedem Zeitpunkt $t$ gilt also $val_{self}(C,t) = C$.

c) Mit den Metatypen wird neben den Sorten für die Komponenten auch eine Sorte für die Klassen $K_1, \ldots, K_n$ festgelegt. Mit der Menge $\mathcal{K}_e$ sind gemäß der Definition 4.7 auf Seite 59 die Klasseneigenschaften festgelegt. Für jede Eigenschaft $k \in \mathcal{K}_e$ existiert eine Operation in $val_E \in \mathcal{O}$ mit

$$val_k : K_i \times T \to dom(k), \text{ mit } val_E(X,t) = \begin{cases} \tilde{Z}(C,k,t) & : \exists C \in X \\ \bot & : \text{sonst} \end{cases}$$

Im Folgenden wird für die Operation $val_k(X, now)$ abkürzend $X \models k$ geschrieben.

Sowohl für Komponenten, als auch für Klassen sind Operationen für die Konstruktion neuer Elemente notwendig. Diese Operationen werden im Abschnitt 6.2.2 auf Seite 114 beschrieben.

**4. Tupel**

Aus dem Kreuzprodukt der Sorten $D_1, \ldots, D_n \in \tilde{\mathcal{D}}$ ergibt sich die Tupelsorte $V = \langle D_1 \times D_2 \times \ldots \times D_n \rangle \in \mathcal{D}$. Für diese sind folgende Operatoren definiert:

**a) Konstruktion eines neuen Tupels**
$build_V : D_1 \times \ldots \times D_n \to V$

**b) Selektion eines Elements**
$sel_{V,i} : V \to D_i$, mit $1 \leq i \leq n$

Mit folgender Gesetzmäßigkeit:

$$v = build_V(d_1, \ldots, d_n) \Rightarrow sel_{V,i}(v) = d_i, 1 \leq i \leq n$$

5. **Abstrakte Datentypen**
   Ein abstrakter Datentyp $A = \langle T, F \rangle$ bildet mit jedem Modell $M_A : S \to \tilde{\mathcal{D}}$ für diesen abstrakten Datentyp eine Datensorte. Auf dieser Sorte sind die Operationen definiert, die für $A$ definiert wurden, wobei die Sorten $T$ aus $A$ gemäß dem Modell $M_A$ ersetzt werden:

   Sei $f \in F$, mit $f_{sig}(f) = T_1 \times \ldots \times T_n \times D_1 \ldots \times D_m \to S$, mit $S \in T \cup \mathcal{D}$ dann ist

   $$f_{M_A} : \begin{cases} M_A(T_1) \times \ldots \times M_A(T_n) \times D_1 \ldots \times D_m & \to & M_A(S) & : & S \in T \\ M_A(T_1) \times \ldots \times M_A(T_n) \times D_1 \ldots \times D_m & \to & S & : & S \in \mathcal{D} \end{cases}$$

   eine Operation. Die Gesetzmäßigkeiten des abstrakten Datentyps übertragen sich entsprechend.

Als weitere Sprachelemente stehen die folgenden Ausdrücke zur Verfügung, deren Semantik hier informell beschrieben wird:

1. **Alternativenauswahl**
   $if\ C_1\ :\ E_1\ \ldots if\ C_n\ :\ E_n\ else\ E_{n+1}$, wobei $C_1 \ldots C_n$ boolsche Ausdrücke sind und $E_1 \ldots E_{n+1}$ Ausdrücke mit beliebigem, aber identischem Ergebnistyp sind. Das Ergebnis dieses Ausdrucks wird durch das $E_j$ $1 \leq j \leq n+1$ bestimmt, dessen Bedingung zuerst zu *wahr* ausgewertet wird. Das heißt, falls mehrere Bedingungen $C_i$ zu *wahr* ausgewertet werden, so bestimmt das $E_j$ mit minimalem $j$ das Ergebnis des Ausdrucks. Wird keines der $C_i$ zu *wahr* ausgewertet, so wird das Ergebnis durch $E_{n+1}$ bestimmt.

2. **Variablenbindung**
   $let\ v := E : E'$, wobei $E$ und $E'$ Ausdrücke mit beliebigem Typ sind. Bei der Auswertung des Ausdrucks werden alle Vorkommen von $v$ in $E'$ durch das Ergebnis des Ausdrucks $E$ ersetzt. Dabei darf $v$ in $E$ nicht vorkommen.

3. **Ereigniserzeugung**
   $throw\ \sigma(E_1, \ldots E_n)@E_{n+1}$, wobei der Ergebnistyp von $E_{n+1}$ eine Komponente aus $\mathcal{C}$ ist. Dieser Komponente wird ein Ereignis der Klasse $\sigma$ zugestellt, wobei $E_1 \ldots E_n$ die Ereignisparameter bestimmen. Das Ergebnis dieses Ausdrucks ist von boolschem Typ und gibt an, ob das Ereignis zulässig war.

4. **Komponentenerzeugung**
   $new[C](E_1, \ldots E_n)$, wobei $C$ eine Komponentenklasse ist, für welche eine neue Komponente erzeugt wird. Die Ausdrücke $E_1 \ldots E_n$ bestimmen die Ereignisparameter des Startereignisses. Die Erzeugung neuer Komponenten und Komponentenklassen wird in Abschnitt 6.2.2 ausführlicher diskutiert.

## 6.2 Dynamische Zustandsänderungen von Systemen

Die mittels der Spezifikation beschriebenen Systeme sind dynamisch, also mit der Zeit veränderlich. Wie in Abschnitt 4.1 auf Seite 51 ausführlich diskutiert, ergibt sich die Dynamik der Systeme einerseits aus der Veränderung der Zustände der Komponenten des Systems, andererseits aus der Veränderung der Menge der zu einem Zeitpunkt existierenden Komponenten. Diese beiden Aspekte der Dynamik müssen in der Systemspezifikation ausdrückbar sein.

### 6.2.1 Zustandsveränderung in Komponenten

Der Zustand einer Komponente zum Zeitpunkt $t$ wird durch die Belegungen der Eigenschaften, welche durch den Metatypen $M$, von dem die Komponente abgeleitet wurde, festgelegt. In der abstrakten Spezifikation einer Komponente muss der Zustand durch eine Rechenvorschrift in Abhängigkeit von der Zeit, dem aktuellen Zustand der Komponente, sowie deren Umgebung, d. h. den Zuständen der durch den Strukturgraphen verbundenen Komponenten, beschrieben werden.

**Definition 6.1 (Direkte Umgebung)**
*Sei $C$ eine Komponente, mit den Eigenschaften $\mathcal{E}_C$. Dann ist die Umgebung von $C$ zum Zeitpunkt $t$ wie folgt definiert:*

$$\begin{aligned} \tilde{U}_C(t) &= \mathcal{E}_C \cup \{e \mid e \in \tilde{U}_{C'}(t),\ C' \in F_C(t)\},\ \textit{mit} \\ F_C(t) &= \{x \mid x = Z(C, E, t),\ E \in \mathcal{E}_C\} \cup \\ &\quad \{o(x_1, \ldots x_n) \mid o \in \mathcal{O},\ x_i \in F_C(t),\ 1 \leq i \leq n\} \end{aligned}$$

Die Umgebung einer Komponente besteht zunächst aus der Menge aller Eigenschaften der Komponente. Darüber hinaus gehören auch die Eigenschaften aller Komponenten, die durch die Systemstrukturen mit der Komponente verbunden sind, zur Umgebung der Komponente. Die Umgebung einer Komponente bestimmt die Grundlage für die Berechnung des Zustandes, also die Belegung der Eigenschaften, einer Komponente zu einem Zeitpunkt $t$. Die vollständige Umgebung einer Komponente ist die Erweiterung der direkten Umgebung einer Komponente um deren Vergangenheit:

**Definition 6.2 (Vollständige Umgebung)**
*Sei $C$ eine Komponente, mit den Eigenschaften $\mathcal{E}_C$. Dann ist die Umgebung von $C$ wie folgt rekursiv definiert:*

$$U_C(t) = \begin{cases} \tilde{U}_C(t) & : \quad t = 0 \\ \tilde{U}_C(t) \cup U_C(pred(t)) & : \quad t > 0 \end{cases}$$

Die Belegung der Eigenschaften einer Komponente wird allgemein durch eine Menge von Funktionen spezifiziert. Die zeitlichen Abhängigkeiten der Zustände einer Eigenschaft werden durch eine Zuordnung von Funktionen zu Ereignissen erfasst.

Sei $M$ ein Metatyp mit der Ereignismenge $\Sigma_M$ und den Eigenschaften $\mathcal{E}_M$. Sei weiterhin $E \in \mathcal{E}_M$ eine Eigenschaft mit der Domäne $dom(E) \in \mathcal{D}$ und $F_E$ eine Menge von Funktionen mit $dom(E)$ als Wertebereich. Die partiell definierte Funktion $\delta_E : \Sigma_M \rightarrow F_E \cup \{\rightarrow \perp\}$ ordnet jedem Ereignis eine Funktion aus der Menge $F_E$ oder $\perp$ zu. Zu jedem Zeitpunkt $t$ ist eine dieser Berechnungsfunktionen für die jeweilige Eigenschaft *gültig*. Die gültige Berechnungsfunktion wird gemäß der Abbildung $\gamma_E : \mathcal{C} \times T \rightarrow F_E \cup \{\rightarrow \perp\}$ ausgewählt, wobei $\gamma_E$ wie folgt definiert ist:

$$\gamma_E(C,t) = \begin{cases} \rightarrow \perp & : \quad t < t(\sigma_{start}(C)) \\ \delta_E(\theta(\sigma_{C,E,t})) & : \quad t \geq \sigma_{start}(C) \end{cases},$$

wobei $\sigma_{C,E,t}$ das Ereignis aus der Menge der für die Komponente $C$ aufgetretenen Ereignisse $\mathcal{H}_C(t)$ ist, dass den jüngsten Zeitstempel besitzt und für das die Funktion $\delta_E$ definiert ist, also

$$\begin{array}{l} \sigma_{C,E,t} \in \mathcal{H}_C(t) \text{ mit} \\ \nexists \sigma' \in \mathcal{H}_C(t) \ : \ t(\sigma') > t(\sigma_{C,E,t}),\ \sigma' \neq \sigma_{C,E,t} \wedge \delta_E(\theta(\sigma_{C,E,t})) \neq \perp \end{array}$$

Somit ist stets diejenige Berechnungsfunktion gültig, die für die Klasse des zuletzt aufgetretenen Ereignisses in der jeweiligen Komponente für das die Eigenschaft eine Berechnungsfunktion definiert, spezifiziert wurde.

Die *Berechnungsfunktionen* $f_{E,\sigma} \in F_E$ für eine Eigenschaft $E$ und ein Ereignis $\sigma$ sind Abbildungen aus der Umgebung der Komponente und der Menge der Ereignisparameter $P_\sigma$ auf den Datensorten der Eigenschaft $dom(E)$:

$$f_{E,\sigma} : {P_\sigma}^* \times {U_C}^* \rightarrow dom(E)$$

Durch eine Berechnungsfunktion wird die Belegung einer Eigenschaft in Abhängigkeit von der Zeit und der Umgebung der Komponenten beschrieben, wobei die

zeitliche Abhängigkeit durch die Ereignisse der Komponente festgelegt wird. Mit der Umgebung einer Komponente werden auch die Eigenschaften der durch den Strukturgraphen mit der Komponente in Beziehung stehenden Komponenten erfasst. Eine Zustandsänderung einer Komponente kann damit durch dreierlei Umstände ausgelöst werden:

1. **Auftreten eines Ereignisses**
   Mit dem Eintritt eines Ereignisses $\sigma$ wird mittels der Abbildung $\delta_E$ die entsprechende Berechnungsfunktion aus der Menge der Funktionen $F_E$ ausgewählt und somit der Zustand der Eigenschaft $E$ entsprechend neu belegt.

2. **Veränderung in der Umgebung**
   Eine Zustandsänderung in der Umgebung der Komponente führt zu einer Veränderung der Belegung der Parameter der Berechnungsfunktion $f_E$. Damit verändert sich der Zustand der Eigenschaft $E$. Diese Zusammenhänge, die als *Propagierung* bezeichnet werden, sind in Abb. 6.1 dargestellt.

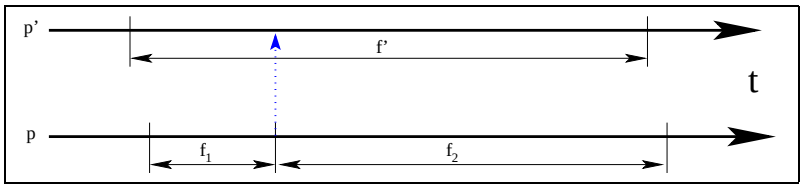



Abbildung 6.1: Propagierung von Eigenschaften.

   Diese Art der Zustandsänderung scheint zunächst im Widerspruch zur Definition 5.1 auf Seite 76 zu stehen, in der ein Ereignis als Auslöser einer Zustandsveränderung gefordert wird, da sich der Zustand einer Komponente verändert, ohne dass ein Ereignis für diese Komponente eingetreten wäre. Der Auslöser für die Zustandsveränderung in diesem Fall ist eine Änderung in der Umgebung der Komponente. Diese Veränderung wird wiederum entweder durch eine Veränderung der Umgebung der Komponenten, oder durch ein Ereignis ausgelöst. Die Ketten, die durch diese fortgesetzte Propagierung von Veränderungen entstehen, müssen aber als Auslöser ein Ereignis haben.

   Ein Ereignis in einer Komponente kann durch die Propagierung von Zustandsänderungen Auswirkungen auf den Zustand einer Vielzahl von Komponenten haben und somit den Zustand des Systems grundlegend verändern.

3. **Veränderung von Klasseneigenschaften**
   In Abschnitt 4.2.2 auf Seite 57 wurden die Eigenschaften eines Metatyps in Komponenteneigenschaften und Klasseneigenschaften unterteilt. Klasseneigenschaften unterscheiden sich demnach von den Komponenteneigenschaf-

ten dadurch, dass sie für eine Klasse von Komponenten zu jedem Zeitpunkt identisch belegt sind. Die Berechnungsfunktionen und die Zuordnung dieser zu den Ereignissen ist für die Klasseneigenschaften analog definiert. Ein Ereignis, welches innerhalb einer Komponente einer Klasse auftritt, kann zur Veränderung der entsprechenden Klasseneigenschaft führen. Da die Belegung dieser Eigenschaft für alle Komponenten der Klasse identisch ist, wird somit die Umgebung aller Komponenten der Klasse verändert. Diese Veränderung kann wiederum zu einer Veränderung der Belegung anderer Eigenschaften der Komponenten führen.

Mit den Berechnungsfunktionen $f_{E,\sigma}$ wird die Belegung einer Eigenschaft zu einem Zeitpunkt beschrieben. Mit diesen Voraussetzungen gilt für die Belegung einer Eigenschaft, also deren Zustand

$$val_C(E,t) = \tilde{Z}(C,E,t) = \gamma_E(C,t)(P_{\sigma_{C,E,t}}, U_C(t)),$$

wobei $C$ eine Komponente ist und $E \in \mathcal{E}_C$.

Da auch die Struktur eines Systems durch die Zustände der Komponenten repräsentiert wird, müssen auch die Strukturrelationen als Teilmenge der Eigenschaften entsprechend berechnet werden. Im Abschnitt 4.3 auf Seite 63 wurde festgelegt, dass mit einer Relation $\mathcal{R} = (M_1, M_2, E)$ zwei Eigenschaften, $E_\mathcal{R}$ in $M_2$ und $E_{\mathcal{R}_\mathcal{S}}$ in $M_1$ definiert sind. Wesentlich ist dabei, dass diese Eigenschaften zu jedem Zeitpunkt in den entsprechenden Komponenten konsistent belegt sind.

Da die durch Relationen beschriebenen (Teil-)Strukturen stets baumartig sind, wird nur für $E_\mathcal{R}$ in $M_2$ eine Menge von Berechnungsfunktionen $F_{E_\mathcal{R}}$ gefordert. Die Menge der Berechnungsfunktionen wird für die Eigenschaft $E_{\mathcal{R}_\mathcal{S}}$ festgelegt als:

$$F_{E_{\mathcal{R}_\mathcal{S}}} = \{f_{E_{\mathcal{R}_\mathcal{S}},\sigma} = \{C \mid M_2 \mapsto C \wedge Z(C, E_\mathcal{R}, t(\sigma)) = \mathit{self}\}\}.$$

Für jede Relation $\mathcal{R}$ gibt es nur eine Berechnungsfunktion für die Zustände der Eigenschaften $E_{\mathcal{R}_\mathcal{S}}$. Daher ist die Zuordnung von Ereignissen zu Berechnungsfunktionen $\delta_{E_{\mathcal{R}_\mathcal{S}}} : \Sigma_{M_1} \rightarrow F_{E_{\mathcal{R}_\mathcal{S}}} \cup \{\rightarrow \perp\}$ konstant als $\delta_{E_{\mathcal{R}_\mathcal{S}}}(\sigma) = f_{E_{\mathcal{R}_\mathcal{S}},\sigma}$ definiert.

### 6.2.2 Komponentenentstehung und Auflösung

Neben der Veränderung des Zustandes eines Systems durch die Belegung der Eigenschaften der Komponenten ist die Entstehung und die Auflösung neuer Komponenten innerhalb des Systems eine zentrale Eigenschaft dynamischer Systeme. Mit den Zeitpunkten der Entstehung und der Auflösung einer Komponente ist die Lebenszeit der Komponente festgelegt.

In der Spezifikation der Ereignisse sind diese Zeitpunkte durch zwei ausgezeichnete Ereignisse beschrieben. Einerseits durch das Startereignis, andererseits durch

die Klasse der Terminungsereignisse (siehe Abschnitt 5.2.1 auf Seite 82). Mittels der Spezifikation dieser Ereignisse ist die Lebenszeit der Komponenten abstrakt festgelegt.

Mit dem Auftreten einer Ereignisses der Terminierungsklasse wird die Komponente aufgelöst. Der Zustand der Komponente zu diesem Zeitpunkt ist durch die entsprechenden Berechnungsfunktionen gemäß den Abbildungen $\delta_E$ und $f_{E,\sigma} \in F_E$ für alle Eigenschaften $E$ der Komponente festgelegt. Der Terminierungszustand der Komponente hat daher Einfluss auf den Zustand der Umgebung der Komponente, sowie auf die Eigenschaften der Klasse der Komponente. Die Umgebung der Komponente ist nach Definition 6.2 auf Seite 112 in Abhängigkeit von den Zuständen der Komponente definiert. Teil des Zustandes sind die Strukturen des Systems. Eine Komponente, die aufgelöst wird, wird aus dem Strukturgraphen entfernt. Dabei können für eine Relation $\mathcal{R} = (M_1, M_2, E)$ zwei Fälle auftreten:

1. **Auflösung von $C_1$ mit $M_1 \mapsto C_1$**
   Alle Komponenten in der Menge $Z(C_1, E_{\mathcal{R}_S}, t)$ haben die Komponente $C_1$ als Belegung für die Eigenschaft $E_{\mathcal{R}}$. Mit der Auflösung von $C_1$ wird die Belegung in diesen Komponenten zu $\perp$.

2. **Auflösung von $C_2$ mit $M_2 \mapsto C_2$**
   In diesem Fall wird die Eigenschaft $E_{\mathcal{R}}$ für $C_2$ mit der Auflösung von $C_2$ mit $\perp$ belegt. Damit wird $C_2$ aus der Belegung der Eigenschaft $E_{\mathcal{R}_S}$ der im Strukturgraphen verbundenen Komponente entfernt.

In beiden Fällen wird die aufgelöste Komponente konsistent aus dem Strukturgraphen entfernt.

Die Entstehung einer neuen Komponente ist mit dem Auftreten eines Ereignisses der Startklasse verbunden. Da die Komponente zu diesem Zeitpunkt aber noch nicht existiert, kann dieses Ereignis zunächst nicht bei der Komponente selbst ausgelöst werden. Daher wird dieses Ereignis über die Klasse, welcher die neue Komponente zugeordnet werden soll, ausgelöst. Damit ist auch die Zuordnung der Komponente zu ihrer Klasse festgelegt. Mit der Erzeugung der Komponente wird ihr initialer Startzustand belegt. Dieser ergibt sich aus den Berechnungsfunktionen für die Eigenschaften der Komponente gemäß der Abbildungen $\delta_E$ und $f_{E,\sigma} \in F_E$. Somit ist auch die Einordnung der neuen Komponente in den Strukturgraphen des Systems erklärt.

Da, wie bereits dargestellt, das Startereignis einer Komponente nicht bei der Komponente selbst ausgelöst werden kann, sondern über die Klasse ausgelöst wird, diese aber als Abstraktum im System selbst keine Ereignisse empfangen kann, wird für die Erzeugung einer neuen Komponente eine zusätzliche Operation benötigt. Sei $\sigma \in \Sigma_M$ ein Startereignis aus der Ereignismenge des Metatypen $M$, mit den Parametern $P_\sigma^1, \ldots, P_\sigma^n$ und $\mathcal{K}_M$ die Menge aller Klassen, welche aus $M$ abgeleitet wurden. Dann gibt es eine Funktion

$$new : \mathcal{K}_M \times P_\sigma^1 \times \ldots \times P_\sigma^n \rightarrow M,$$

welche eine neue Komponente erzeugt, wobei folgende Gesetzmäßigkeiten gelten:

Sei $k \in \mathcal{K}_M$, dann gilt

a) $C = new[k](p_1, \ldots p_n) \Rightarrow M \mapsto C$

b) $C = new[k](p_1, \ldots p_n) \Rightarrow C \in k$

Neben der Entstehung und der Auflösung von Komponenten innerhalb eines Systems ist auch die Frage der Entstehung und der Auflösung von Klassen zu klären. Eine Möglichkeit, die Entstehung und Auflösung von Klassen zu erklären, wäre die Festlegung, dass eine Klasse solange existiert, wie es Komponenten dieser Klasse gibt. Diese Festlegung ist jedoch unbrauchbar, da einerseits Klassen den Informationsfluss von bereits terminierten Komponenten zu neuen Komponenten realisieren. Dieser Informationsfluss wäre unterbrochen, wenn in einem System für eine gegebene Zeitspanne keine Komponente einer gegebenen Klasse existiert. Andererseits wurde bei der Definition der Entstehung der Komponenten vorrausgesetzt, dass die entsprechende Klasse bereits existiert. Die Entstehung einer ersten Komponente wäre daher auf diese Art und Weise nicht möglich.

Die Entstehung einer Klasse wird daher als eine Funktionalität des Managements festgelegt. Die Erschaffung einer neuen Klasse kann durch den expliziten Aufruf einer entsprechenden Managementfunktion realisiert werden. Um eine initiale Belegung der Klasseneigenschaften zu erreichen, durch die sich zwei Klassen unterscheiden können, können dieser Initialisierungsfunktion Parameter $P_1, \ldots, P_n$ übergeben werden, mit $P_i \in \mathcal{D}$, $1 \leq i \leq n$. Die Festlegung der Parameter ist metatyp-spezifisch. Sei also $M$ ein Metatyp mit den Parametern $P_1, \ldots, P_n$, dann gibt es folgende Funktion:

$$M : P_1, \ldots, P_n \rightarrow \mathcal{K}_M$$

Die Lebenszeit einer Klasse wird als unendlich festgelegt. Die Auflösung einer Klasse im System ist daher nicht möglich. Somit stehen die Informationen, welche von den Klasseneigenschaften repräsentiert werden, ab dem Entstehungszeitpunkt einer Klasse dem Management stets zur Verfügung.

Die Festlegung der unendlichen Existenz einer Klasse eröffnet auch die Möglichkeit, Informationen über die Klassen von einer Ausführung eines Systems zur nächsten Ausführung zu erhalten. Somit kann das Management durch die fortlaufende Sammlung von Information über mehrere Ausführungen des Systems hinweg Optimierungen in der Systemausführung vornehmen.

## 6.3 Information und Entscheidung

Die Belegungen der Eigenschaften einer Komponente beschreiben deren Zustand. Der Zustand setzt sich einerseits aus dem Wissen des Managements über die jeweilige Komponente, aber auch aus den Entscheidungen des Managements zur Realisierung der Komponente zusammen. Die Eigenschaften einer Komponente erhalten somit eine duale Bedeutung. Diese Dualität wird im folgenden diskutiert.

### 6.3.1 Eigenschaften als Information

Die einer Komponente zugeordneten Eigenschaften repräsentieren das Wissen des Managements über die jeweilige Komponente. Die Aufgabe der Eigenschaften liegt in dieser Sichtweise also in der Erfassung, Speicherung und Bewertung von Information. Die Berechnungsfunktionen der Eigenschaften repräsentieren somit die Art und Weise, auf welche Informationen gewonnen und gegebenenfalls bewertet werden. Dies ist die klassische Aufgabe des Monitoring (vgl. [BKLW00] oder [Rac01]).

Diese Information kann zu unterschiedlichen Zwecken genutzt werden. Für den Entwickler von Problemlösungsverfahren sind während der Entwicklung Informationen von Interesse, die auf Fehlersituationen hindeuten. Fehlersuche ist bei der Erstellung von Problemlösungsverfahren einer der zentralen Schritte. Die Informationen über die aus der Problembeschreibung entstandenen Komponenten kann wesentlich zur Vereinfachung der Fehlersuche und Fehlerbereinigung beitragen. Die Informationen über die Komponenten des Systems kann aber auch für die Analyse der Performanz des Systems genutzt werden. Das Wissen über Performanz-Parameter kann einerseits vom Entwickler eines Problemlösungsverfahrens genutzt werden. In vielen Sprachkonzepten entscheidet auch die Formulierung der Problemlösung über die Performanz der Ausführung. Andererseits kann diese Information auch vom Management des Systems genutzt werden, um die Entscheidungen für die Realisierung, beispielsweise die Bindung von Ressourcen, verbessern zu können.

Neben der Fehlersuche während der Entwicklung eines Problemlösungsverfahrens ist die Erkennung von Fehlerzuständen auch während der Ausführung von Interesse. Nur wenn Fehlerzustände erkannt werden, kann entsprechend – entweder automatisiert oder durch den Benutzer – auf die Situation reagiert werden. Systeme, die Fehlerzustände nicht erkennen und entsprechende Reaktionen auslösen können, sind nicht beherrschbar und beim Einsatz in kritischen Umgebungen sogar gefährlich. Daher ist die Erkennung von Fehlerzuständen eine zentrale Aufgabe von Managementsystemen.

Für die funktionale Sicherheit sind neben den Zuständen der Komponenten des Problemlösungsverfahrens auch die Zustände der Ressourcenkomponenten von Interesse. Gerade in einem verteilten System ist die Auslastung der einzelnen Ressourcen von Bedeutung, da auf dieser Grundlage vielfältige Managemententscheidungen getroffen werden können. Dabei spielt auch die Bindung zwischen den Ressourcen und den Komponenten des Problemlösungsverfahrens eine zentrale Rolle. Diese Bindungen beschreiben unter anderem die Nutzungsbeziehungen zwischen Komponenten und ihren Ressourcen. Diese Bindungen können über den Strukturgraphen und dessen Teilstrukturen beschrieben und analysiert werden, so dass diese Information dem Management auf feingranularer Ebene zur Verfügung gestellt werden kann. Für die feingranulare, effiziente Verwaltung von Ressourcen allgemein ist ein erhebliches Wissen des Managements über einerseits das ausgeführte Problemlösungsverfahren, andererseits über die Ressourcen selbst notwendig. Als Beispiel seien hier Techniken zur automatischen Speicherbereinigung (Garbage Collecction) genannt (siehe [PS95] für einen Überblick).

Neben der funktionalen Sicherheit muss für ein System auch die Informationssicherheit gewährleistet werden. Aufgabenstellungen wie Authentifikation und die Überprüfung von Zugriffsberechtigungen sind wesentliche Aspekte in diesem Umfeld. Informationssicherheit kann aber nur mit einer ausreichenden, korrekten und feingranularen Menge an Information über das System bzw. seine Komponenten durch das Management sichergestellt werden (vgl. z. B. [Eck00b]).

Bei all diesen Fragestellungen bezüglich des Wissens über den Zustand von Systemen wird klassisch zwischen dem Wissen zur Übersetzungszeit, also der aus dem Problemlösungsverfahren stammenden Information, und der durch Monitoring gewonnenen Information unterschieden. Die Information, die zur Übersetzungszeit gewonnen wird, fließt in einem klassischen System nur indirekt in das ausgeführte System und steht dem Management während der Ausführung somit nicht zur Verfügung. Damit geht eine zentrale Quelle der Information über das System verloren. Ebenso stehen Informationen, welche durch die Beobachtung des Systems während der Ausführung entstehen, dem Übersetzer nicht zur Verfügung. Dieser ist somit bei diversen Entscheidungen auf grobe Abschätzungen angewiesen. Bei klassischer Programmierung ist der einzige Informationsfluss zwischen einem System in Ausführung und dem Übersetzer durch den Programmierer gegeben, welcher durch eine veränderte Programmierung oder Hinweise an den Übersetzer einen Informationsfluss simulieren kann. Dabei ist allerdings die Voraussetzung, dass die zur Verfügung stehenden Ressourcen bekannt sind. Insbesondere bei verteilten Systemen mit Dynamik in der Hardwarebasis ist diese Voraussetzung nicht unbedingt gegeben. Damit kann das System nicht feingranular an die vorhandene Basis angepasst werden.

Durch die einheitliche Spezifikation von Sprache, Ressourcen und Hardware sowie den Strukturen zwischen diesen kann – gemeinsam mit den Informationen

über diese Komponenten – ein automatisiertes Management beschrieben werden, dass sowohl die Information aus der Problemlösungsbeschreibung, als auch aus der Ausführung gleichermaßen berücksichtigt. Da es bei diesem Vorgehen keinerlei Trennung zwischen statischer und dynamischer Analyse sowie statischem und dynamischen Management gibt, entsteht dieser Informationsfluß automatisch. Damit steht dem Management ein umfassendes, feingranulares Wissen zur Verfügung, auf dessen Grundlage eine effiziente und korrekte Realisierung der Problemlösung ermöglicht wird.

### 6.3.2 Repräsentation von Managemententscheidungen

Eine andere Sichtweise auf Eigenschaften und deren Berechnungsfunktionen erhält man, wenn man die Belegung der Eigenschaften als Ergebnis von Managemententscheidungen betrachtet. Die Grundlage der Managemententscheidungen bildet das Wissen, welches durch die Belegung der Eigenschaften repräsentiert wird. Dieses Wissen umfasst die reinen Informationseigenschaften, wie sie in Abschnitt 6.3.1 auf Seite 117 beschrieben wurden, aber auch bereits getroffene Managemententscheidungen.

Die Realisierung einer Komponente bzw. die Realisierung des Zustands der Komponente wird durch die Bindung der Komponente an Komponenten einer niedrigeren Abstraktionebene beschrieben. Jede Realisierungsentscheidung für eine Komponente schränkt die Freiheitsgrade des Managements ein. Durch die Bindung einer Komponente an Hardwareressourcen werden die Freiheitsgrade für Managemententscheidungen zu Null, d. h. es sind keine weiteren Entscheidungen möglich bzw. notwendig.

Die Entscheidungen des Managements können allerdings revidiert werden, wenn eine Veränderung im Zustand der realisierten Komponente dies erforderlich macht. Die Bindung von Ressourcen an eine Komponente ist somit temporär.

Die Bindungen der Komponenten werden durch die Relationen in der Spezifikation repräsentiert. Die Berechnungsfunktionen für die entsprechenden Relationen können daher als Entscheidungsfunktionen betrachtet werden. Für eine Relation hat das Management Entscheidungsfreiheit in mehreren Dimensionen. Zunächst kann das Management zwischen unterschiedlichen Klassen und Komponenten entscheiden, da in der Spezifikation für die zueinander in Relation stehenden Komponenten nur deren Metatyp festgelegt wird. Diese Entscheidungsfreiheit wird als *horizontaler Freiheitsgrad* bezeichnet.

**Beispiel 6.1**
In einem verteilten System mit der Knotenmenge $\{N_1, \ldots, N_n\}$, wobei diese von einem Metatypen $\mathcal{N}$ abgeleitet sind, und den Aktivitäten $\{A_1, \ldots A_m\}$, welche von einem Metatypen $\mathcal{A}$ abgeleitet wurden, kann das Management für jede Aktivität entscheiden, auf welchem Knoten die Aktivität realisiert werden soll. Die Managemententscheidung für die Realisierung einer Aktivität $\mathcal{A} \mapsto A$ ist dann eine Berechnungsfunktion der Form ${P_\sigma}^* \times {U_A}^* \to \mathcal{N}$, wobei $\sigma \in \Sigma_\mathcal{A}$ das Startereignis der aktiven Komponente $A$ ist.

Neben den horizontalen Freiheitsgraden kann das Management auch über den Typ der entsprechenden Realisierung entscheiden. Wird die entsprechende Relation mit einem abstrakten Metatyp als Ressource spezifiziert, so kann das Management aus den Ressourcenkomponenten wählen, welche aus entsprechenden Untertypen abgeleitet sind. Durch die Hierarchie der Metatypen werden die potenziellen Alternativen festgelegt. Diese Art der Freiheitsgrade wird als *vertikaler Freiheitsgrad* bezeichnet.

**Beispiel 6.2**
Gegeben sei eine Systemspezifikation mit den Metatypen $Objekt$, $Speicher$, $Platte$ und $RAM$, wobei die Komponenten des Metatyps $Objekt$ Speicherfähigkeit besitzen, die durch das Management realisiert werden muss. Diese Realisierung wird durch die Komponenten des Metatyps $Speicher$ erreicht, der allerdings als abstrakter Metatyp mit den Untertypen $Platte$ und $RAM$ spezifiziert sei.

Damit kann das Management bei der Realisierung der Relation zwischen einer Komponente $O$ mit $Objekt \mapsto O$ und einer Komponente $S$ mit $Speicher \rightsquigarrow S$ entscheiden, ob $Platte \mapsto S$ oder $RAM \mapsto S$ gelten soll. Diese Entscheidung wird mittels der entsprechenden Berechnungsfunktion ${P_\sigma}^* \times {U_O}^* \to Speicher$ spezifiziert, wobei $\sigma \in \Sigma_{Objekt}$ das Startereignis der Komponente $O$ ist.

Um auf Veränderungen im System geeignet reagieren zu können, dürfen die Entscheidungen des Managements nicht irreversibel sein. Eine einmal getroffene Entscheidung muss unter Umständen verändert werden, falls sich Veränderungen in der Umgebung der Komponente ergeben, oder der Berechnungsfortschritt im System eine erneute Entscheidung erforderlich macht. Diese Veränderungen können in der Spezifikation der Berechnungsfunktionen der Eigenschaften zur Repräsentation von Entscheidungen ausgedrückt werden. Da die Berechnungsfunktionen von der Umgebung der Komponenten abhängig sind, wirkt sich, wie

in Abschnitt 6.2.1 auf Seite 111 diskutiert, diese Veränderung auch auf die Belegung dieser Eigenschaft aus. Somit wird eine Managemententscheidung mit der Veränderung der Umgebung der Komponente entsprechend neu getroffen.

Der Berechnungsfortschritt im System, als zweite Ursache für die Veränderung im System, wird durch das Auftreten von Ereignissen in den Komponenten repräsentiert. Die Berechnungsfunktionen für die Eigenschaften zur Repräsentation von Managemententscheidungen sind, wie alle Berechnungsfunktionen, in Abhängigkeit von den Ereignissen definiert. Mit dem Auftreten eines neuen Ereignisses kann somit eine veränderte Berechnungsfunktion ausgewählt werden. Durch diese Veränderung der Berechnungsfunktion für eine Eigenschaft kann somit eine erneute Managemententscheidung ausgelöst werden.

### 6.3.3 Durchsetzung von Eigenschaften

Mit der oben beschriebenen Entscheidungsfindung des Managements durch die Berechnung von Belegungen für Eigenschaften der Komponenten ist die Politik eines Systems beschrieben. Für die Realisierung eines Systems ist es aber nicht ausreichend, eine Politik festzulegen; diese muss auch durch entsprechende Mechanismen durchgesetzt werden. Bei der Entscheidung, eine Aktivität auf einem ausgewählten Knoten zu plazieren wie in Beispiel 6.1 auf der vorherigen Seite, muss diese Entscheidung durch das Initiieren der Aktivität auf dem entsprechenden Knoten durchgesetzt werden.

Die Durchsetzung von Zuständen kann entweder durch die Verwendung entsprechender Funktionalitäten der Hardware-Ressourcen erreicht werden, oder durch das Auslösen eines Ereignisses in der entsprechenden Komponente, wobei Letzteres auch die Erzeugung einer neuen Komponente beinhalten kann.

Für die Durchsetzung von Entscheidungen wird daher die Spezifikation der Metatypen um *Aktionen* ergänzt. Es obliegt einer Aktion, die Durchsetzung von Managemententscheidungen zu realisieren. Aktionen lösen in Abhängigkeit vom Zustand einer Komponente bestimmte Basisfunktionalitäten aus. Diese Basisfunktionalitäten sind:

1. **Erzeugung von Ereignissen**
   Durch Ereignisse können Komponenten sich untereinander synchronisieren. Sobald eine Komponente einen Zustand erreicht hat, kann diese ein Ereignis bei einer anderen Komponente auslösen.

2. **Die Erzeugung von Komponenten und Klassen**
   Die Realisierung bestimmter Managementfunktionalität erfordert die Erzeugung neuer Komponenten oder Komponentenklassen. Soll eine Akti-

vität beispielsweise durch einen Thread realisiert werden, so muss dafür eine Komponente vom Typ *Thread* erstellt werden.

3. **Ausführen von Funktionalität der Hardware**
   Die Funktionalität der Hardware wird durch die externen Operationen modelliert. Die Auslösung von Funktionalitäten der Hardware wird durch den Aufruf der entsprechenden Funktion initiiert. Bei der Erzeugung einer Komponente des Typs *Thread* zum Beispiel wird mit der Erzeugung einer Instanz dieses Metatyps auch ein realer Thread erzeugt, im Fall einer Realisierung durch *pthreads* beispielsweise durch den Aufruf der `createthread` Funktion.

Damit kann eine Aktion wie folgt definiert werden:

**Definition 6.3 (Aktion)**
*Sei $M$ ein Metatyp mit den Eigenschaften $\mathcal{E}_M = \{E_1, \ldots, E_n\}$. Eine Aktion ist dann eine Abbildung*

$$A : dom(E_1) \times \ldots \times dom(E_n) \rightarrow \bigcup D_i,\ \forall D_i \in \mathcal{D},$$

*mit einer Bedingung für die Durchführung der Aktion*

$$C_A : dom(E_1) \times \ldots \times dom(E_n) \rightarrow I\!B$$

*und der Semantik*

$$A(e_1, \ldots, e_2) = \begin{cases} f_A(e_1, \ldots e_n) & : \quad C_A(e_1, \ldots e_n) = wahr \\ \bot & : \quad sonst \end{cases},$$

*mit der Aktionsfunktion*

$$f_A : dom(E_1) \times \ldots \times dom(E_n) \rightarrow \bigcup D_i,\ \forall D_i \in \mathcal{D}.$$

Die Aktionsbedingung beschreibt die Belegungen der Eigenschaften einer Komponente, die zur Auslösung der Aktion führen. Wird diese Bedingung zu *wahr* ausgewertet, dann und nur dann wird die entsprechende Aktionsfunktion ausgeführt. Die Aktionsfunktion selbst beschreibt die durchzuführenden Berechnungen zur Durchsetzung von Managemententscheidungen.

Aktionen in der hier beschriebenen Art und Weise können neben der Durchsetzung von Managementfunktionalität auch zur Reaktion auf das Eintreten bestimmter Zustände, wie zum Beispiel Fehlerzuständen, verwendet werden. Damit

steht mit den Aktionen ein weiteres mächtiges Mittel zur Beschreibung des Managements, seiner Politik und den entsprechenden Mechanismen zur Verfügung,

## 6.4 Zusammenfassung

Mit den getroffenen Festlegungen kann nun eine Spezifikation von dynamischen Systemen und von Metatypen als zentrales Element für die Spezifikation formal definiert werden.

**Definition 6.4 (Metatyp)**
*Ein Metatyp ist ein Acht-Tupel:* $M = \langle \mathcal{T}, \mathcal{E}, \mathcal{K}, \Sigma, \mathcal{P}, \mathcal{F}, \delta, \mathcal{A} \rangle$, *wobei*

- $\mathcal{T}$ *eine Menge von Metatypen ist, für die gilt:* $\forall M' \in \mathcal{T} \ : \ M \leftarrow M'$
- $\mathcal{E}$ *eine Menge von Eigenschaften mit zugeordneten Domänen ist.*
- $\mathcal{K} \subseteq \mathcal{E}$ *eine Menge von Klasseneigenschaften ist.*
- $\Sigma$ *eine Menge von Ereignissen gemäß Definition 5.5 auf Seite 84 ist.*
- $\varphi$ *eine Menge von Phasen gemäß Definition 5.7 auf Seite 92 ist.*
- $\mathcal{F}$ *eine Menge von Funktionen ist.*
- $\delta$ *eine Abbildung* $\mathcal{E} \times \Sigma \rightarrow \mathcal{F}$ *ist, welche sich aus den Funktionen* $\delta_E$ *für alle* $E \in \mathcal{E}$ *ergibt.*
- $\mathcal{A}$ *eine Menge von Aktionen gemäß Definition 6.3 auf der vorherigen Seite ist.*

Die Metatypen bilden die Grundlage für eine Spezifikation im Allgemeinen. Eine Spezifikation ist formal definiert als:

**Definition 6.5 (Spezifikation)**
*Eine Spezifikation ist ein Drei-Tupel:* $S = \langle \mathcal{D}, \mathcal{M}, \mathcal{H} \rangle$, *wobei*

- $\mathcal{D}$ *eine Menge von vollständigen Datensorten gemäß Definition 4.8 auf Seite 62 ist.*

- $\mathcal{M}$ *eine Menge von Metatypen ist.*
- $\mathcal{H}$ *eine Menge von Hilfsfunktionen, wobei jede Funktion* $H \in \mathcal{H}$ *eine Signatur der Form* $H : \mathcal{D}^* \rightarrow \mathcal{D}$ *besitzt.*

Diese Form der Spezifikation ermöglicht eine abstrakte Beschreibung des Verhaltens von Komponenten. Komponenten repräsentieren sowohl die Problemlösung, als auch die zu deren Ausführung benötigten Ressourcen. Durch diese einheitliche Beschreibung der beiden Anteile eines Systems wird ein Informationsfluss zwischen allen Komponenten beschrieben, welcher als Grundlage für die Entscheidungen des Managements dient. Die nicht funktionalen Zustände der einzelnen Komponenten werden ebenso einheitlich repräsentiert, unabhängig davon ob sie Managemententscheidungen oder durch Beobachtung gewonnenes Wissen darstellen. Die Durchsetzung von Entscheidungen wird über Berechnungsfunktionen und Aktionen beschrieben.

Abschließend soll hier ein Beispiel präsentiert werden, welches die diskutierten Konzepte, sowie ihre Anwendung und das Zusammenspiel der unterschiedlichen Teile einer Spezifikation verdeutlicht. Die Notation erfolgt dabei gemäß der formalen Aufschreibung aus Abschnitt 9.6 auf Seite 178, die hier aber intuitiv verständlich sein sollte.

**Beispiel 6.3**
Gegeben sei ein objektbasiertes System mit aktiven Komponenten. Jede aktive Komponente wird aus einem Generator erzeugt. Jeder Generator kann Komponenten eines bestimmten Typs, festgelegt durch die Problembeschreibung, erzeugen. Dieses Konzept ist in [SEL+96], sowie [Piz99] ausführlich beschrieben. Jede Komponente im System hat darüber hinaus eine wohldefinierte Lebenszeit. Die aktiven Komponenten können über eine Menge von Knoten verteilt werden.

Zunächst wird ein allgemeiner, abstrakter Metatyp *instance* definiert, der alle zur Laufzeit existierenden Komponenten repräsentiert. Diesem Metatypen werden zwei Ereignisse *init* und *done* zugeordnet. Das *init* Ereignis wird mit dem Aufrufer vom Typ *instance* und dem Generator, der die Instanz erzeugt, parametrisiert. Durch die Bedingungen für die Lebenszeit einer Komponente wird festgelegt, dass eine Komponente nur dann terminieren kann, wenn sie selbst keine Kinder mehr enthält. Daher wird in der Spezifikation für das Ereignis *done* festgelegt, dass diese nur auftreten kann, wenn es keine Kinder mehr gibt. Das Ereignis *inst_done* im Generator ist abhängig vom Ereignis *done* der Instanzen und kann

nur gemeinsam mit diesem auftreten. Die Relation für die Lebenszeitabhängigkeiten der Kinder wird durch $\varepsilon$ beschrieben. Jede Instanz, mit Ausnahme der Generatoren, hängt vom jeweiligen Vater ab.

```
META instance

  EVT init (father : instance, gen : Generator)
  EVT done[(count(ε_S)=0)] ⇒ inst_done (self)@generator

  REL ε → instance
    init :                  IF ¬(self ≻ Generator) : father
                            ELSE ⊥
    done :                  ⊥
  PROP used → int
  PROP generator → Generator
    init :                  gen
  PROP Operation → fct
    init :                  IF self ≻ active_inst : gen⊢operation
                            ELSE ⊥
  REL located → node
    init :                  IF self ≻ active_inst :
                              IF (gen⊢time≥10) :
                                LET H← next(getnode()⊢master⊢clients) :
                                  IF (H≠⊥) : H
                                  ELSE
                                    getnode()
                              IF (gen⊢time≥5) :
                                getnode()
                              ELSE ⊥
                            IF (father≠⊥) : father⊢located
                            ELSE
                              getnode()
    done :                  ⊥

  ACT run[((located=⊥)∧(Operation≠⊥))]
    exec(Operation)

END instance
```

Die Eigenschaft „Operation“ verweist auf die Funktion, die durch eine Instanz ausgeführt wird. Wenn es sich bei der Instanz um einen Generator handelt, so besitzt diese keine Operation.

Die Bindung der Aktivitäten an die jeweiligen Knoten wird durch die Relation „located“ beschrieben. Im einfachsten Fall wird eine Instanz auf dem Knoten plaziert, auf dem der Aufruf der Aktivität stattfand. Im Falle einer Aktivität hängt die Platzierungsentscheidung von der vorhergesagten Laufzeit der Aktivität ab, welche durch die Eigenschaft „time“ des Generators repräsentiert wird. Kurzlebige Aktivitäten werden wie Funktionen sequentiell ausgeführt. Aktivitäten, die etwas länger leben, werden auf dem Knoten des Aufrufers als eigenständiger Thread ausgeführt und diejenigen, die langlebig sind, werden über die Knoten verteilt.

Wenn eine Instanz eine Funktion besitzt, aber keinen zugewiesenen Knoten, so wird durch die Aktion „run“ die Funktion ausgeführt.

Jeder Generator im System wird durch den Metatypen „Generator“ repräsentiert. Für diesen Metatyp sind zwei Parameter für die initiale Belegung der Klasseneigenschaften spezifiziert, „operation“ für die auszuführende Operation der erzeugten Aktivitäten und „gens“ für eine Liste von eingeschachtelten Generatoren. Außerdem gibt es für Generatoren eine Klasseneigenschaft „ctime“, welche die vorhergesagte Laufzeit der erzeugten Aktivitäten repräsentiert. Die vorhergesagte Laufzeit für die von *einem* Generator erzeugten Komponenten wird durch die Eigenschaft „time“ dargestellt.

**META** `Generator`[ gens : list⟨`Generator`⟩, operation : `fct` ] ← instance

**EVT** *inst_done* (i : `instance`)

**PROP** ctime → `int` := 1
*done* : *DIV*(*add*(ctime, time), 2)
**PROP** time → `int` := ctime
*inst_done* : *DIV*(*add*(time, i⊢used), 2)

**END** `Generator`

Die Instanzen eines Generators, d. h. die Aktivitäten werden durch den Metatypen `active_inst` repräsentiert. Jede Aktivität beobachtet ihre Laufzeit mittels der Eigenschaft „used“. Diese Eigenschaft der Aktivitäten wird von den Generatoren verwendet, um die vorhergesagte Laufzeit zukünftiger Instanzen zu berechnen. Die Aktion *gens* erzeugt über die Funktion *build_gens* die Generatoren, die in der jeweilige Aktivität enthalten sind.

**META** `active_inst` ← instance

**PROP** used → `int`
*init* : *getptime*()
*done* : *sub*(*getptime*(), used)

**ACT** *gens*[(state=init)]
*build_gens*(*self*, generator⊢gens)

**END** `active_inst`

Die Ressourcen werden in diesem Beispiel durch die Metatypen `thread` und `node` modelliert, wobei die Knoten zur Ausführung von Threads benötigt werden. Die Knoten stellen in diesem Beispiel die Hardware des Systems dar. Jeder reale Knoten wird durch eine Komponente „node“ repräsentiert. Wenn das Management entscheidet, eine Aktivität auf einem Knoten zu starten, so startet der Knoten eine Komponente vom Typ „thread“, um den Code der Aktivität auszuführen. Dies wird über die Aktion *run* erreicht. Die „thread“-Komponente wird durch einen *pthread* realisiert, der mittels `createthread` gestartet wird.

**META** `thread`

**EVT** *init* (f : `fct`)
**EVT** *done*

**PROP** pthread → `int`
*init* : *createthread*(f)

**END** `thread`

**META** `node`

**EVT** *init*
**EVT** *done*

**ACT** *run*[((state=located_in)∧(*count*(located$_S$)>0))]
**LET** H← *current*(located$_S$) :
**IF** H ≻ `active_inst` :
**NEW** *thread*() [H⊢Operation]
**ELSE** *falsch*

**END** `node`

In Abb. 6.2 auf der nächsten Seite ist ein System dargestellt, das mit den oben beschriebenen Metatypen erstellt wurde. Aus dem Metatypen „Aktivität“ sind vier Klassen, nämlich *System*, *Server* und *Berechne* und *Check*, abgeleitet, die ineinander geschachtelt sind. Im laufenden System wurden von einer Komponente *System* zwei *Server*-Komponenten gestartet, welche wiederrum selbst Aktivitäten gestartet haben. Jeder Aktivität ist ein Thread zugeordnet, welcher auf einem Knoten ausgeführt wird.

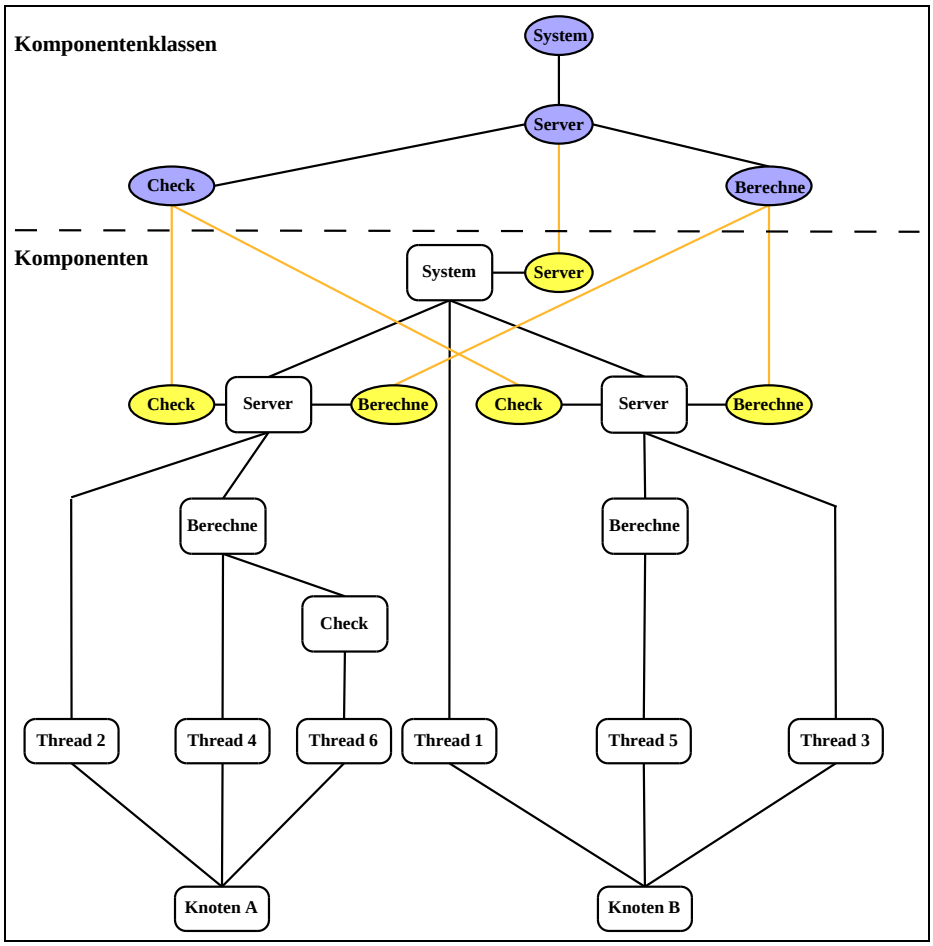



Abbildung 6.2: Ein System mit Generatoren.

# Teil II

# Generierung von Managementsystemen


## Zusammenfassung

Ausgehend von der in Teil I entwickelten Spezifikationssprache wird in diesem Teil der Arbeit die Generierung von Managementsystemen diskutiert. Die Grundlage für die Generierung des Managements wird durch die Analyse der Spezifikation gebildet. In diesem Teil der Arbeit wird auch die Anwendbarkeit der entwickelten Technik anhand mehrerer Beispiele demonstriert.

# 7 Analyse von Spezifikationen

Die effiziente Transformation der abstrakten Spezifikation eines Managementsystems in ein maschinell ausführbares System erfordert auch eine Analyse der impliziten und der expliziten Abhängigkeiten innerhalb der Spezifikation bzw. der spezifizierten Komponenten und Klassen. Eine besondere Rolle spielt dabei die statische Vorhersage dynamischer Abläufe und Zusammenhänge, die durch die Ereignisse und deren Reihenfolge im System beschrieben werden.

Da die dynamische Entwicklung eines Systems jedoch nicht vollständig vorhersagbar ist, sind für die Steuerung des Managements auch dynamische Aspekte von Bedeutung. So entstehen z. B. die Relationen zwischen zwei Komponenten erst zur Laufzeit des Systems. Daher können diese auch erst zur Laufzeit konkret ermittelt und analysiert werden.

In diesem Kapitel wird zunächst die Analyse der Abhängigkeiten zwischen den Ereignissen der Komponenten beschrieben. Daran schließt sich die Analyse der Zusammenhänge zwischen den Eigenschaften der Komponenten des Systems an. Sowohl die Ereignisse als auch die Eigenschaften können dynamisch und statisch analysiert werden, wobei insbesondere die statische Analyse Informationen für den Generierungsprozess bildet.

## 7.1 Ereignisabhängigkeiten

Die Ereignisse, die Phasen und die Ereignisfolgen beschreiben die zeitlichen Abläufe des Systems. Gemeinsam mit den Zusammenhängen der Komponenten durch die Systemstrukturen können die zeitlichen Abläufe des Systems analysiert werden.

Gemäß den Festlegungen aus dem Abschnitt 5.2 auf Seite 80 befindet sich eine Komponente $C \in \mathcal{C}$ zu einem Zeitpunkt in genau einer minimalen Phase.

Für die Analyse der Abhängigkeiten innerhalb der Komponenten sind die zeitlichen Zusammenhänge zwischen den Komponenten von zentraler Bedeutung. Eine Anwendung dieser Zusammenhänge wurde bereits in Abschnitt 5.3.1 auf Seite 94 bei der Einführung der konsistenten Ereignisfolgen aufgezeigt.

**Definition 7.1 (Phasenbeziehungen)**
*Sei $C$ eine Komponente und $C'$ eine durch die Relation $R$ verbundene Komponente. Sei weiterhin $\zeta_C$ die Menge der minimalen Phasen der Komponente $C$, sowie $\zeta_{C'}$ die Menge der minimalen Phasen der Komponente $C'$. Für eine Phase $p \in \zeta_C$ ist die Menge der Phasenbeziehungen zu $C'$ – $\tilde{\mathcal{P}}(p, C') \subseteq \zeta_{C'}$ – definiert, wobei gilt, dass falls $C$ in der Phase $p$ ist, so ist $C'$ in einer Phase $p'$ und $p' \in \tilde{\mathcal{P}}(p, C')$.*

Die Menge $\tilde{\mathcal{P}}(p, C')$ enthält somit alle minimalen Phasen, in denen sich die Komponente $C'$ befinden kann, wenn $C$ in der minimalen Phase $p$ ist.

Die Menge $\tilde{\mathcal{P}}(p, C')$ kann mittels des deterministischen endlichen Automaten der potenziellen Ereignisfolgen der Metatypen $M \mapsto C$ und $M' \mapsto C'$ berechnet werden. Für jede minimale Phase $p$ einer Komponente kann die Sprache der Präfixe bis zu dieser Phase konstruiert werden. Diese wird als $pre(Q, p)$ bezeichnet, wobei $Q$ der reguläre Ausdruck der potenziellen Ereignisfolgen der Komponente ist. Seien die potenziellen Ereignisfolgen von $C$ durch $R$ und die potenziellen Ereignisfolgen von $C'$ durch $R'$ gegeben.

Analog zur Überprüfung der Konsistenz der Ereignisfolgen werden die regulären Ausdrücke der potenziellen Ereignisfolgen reduziert, indem alle unabhängigen Ereignisse aus den regulären Ausdrücken entfernt werden (siehe 5.3.1 auf Seite 94). Diese reduzierten potenziellen Ereignisfolgen werden im Folgenden als $\rho(Q)$ für den regulären Ausdruck $Q$ bezeichnet. Mit diesen Festlegungen kann $\tilde{\mathcal{P}}(p, C')$ wie folgt definiert werden:

$$\tilde{\mathcal{P}}(p, C') = \{p' \in \zeta' \mid L(pre(\rho(R'), p')) \subseteq L(pre(\rho(R), p))\},$$

wobei $L(Q)$ die durch den regulären Ausdruck $Q$ definierte Sprache ist.

Die Menge der vollständigen Phasenbeziehungen einer Komponente $C$, $\mathcal{P}(p)$ in der minimalen Phase $p$ ist dann als

$$\mathcal{P}(p) = \bigcup_{C' \in \mathcal{X}} \tilde{\mathcal{P}}(p, C')$$

definiert, wobei $\mathcal{X} = \{X \in \mathcal{C} \mid M \mapsto X \Rightarrow \exists E \in \mathcal{E}_C : dom(E) = M\}$.

Ausgehend von den Phasenbeziehungen können die Ereignisbeziehungen zwischen den Ereignisklassen der Komponenten bzw. Metatypen gebildet werden. Die Grundlage dafür bildet der Zusammenhang zwischen den Ereignissen und den minimalen Phasen, welcher durch den deterministischen endlichen Automaten der regulären Ausdrücke für die potenziellen Ereignisfolgen beschrieben wird. Für einen Metatypen $M$ wird dieser Automat durch das Tupel $A_M =$

$\langle \Sigma_M, \zeta_M, \delta_M, \zeta_{0,M}, \phi_M \rangle$ beschrieben. Für jede minimale Phase $p \in \zeta_M$ kann die Menge der Ereignisse $\Psi_M(p)$ bestimmt werden, welche in diese Phase führen:

$$\Psi_M(p) = \{\sigma \in \Sigma_M \mid \exists p' \in \zeta_M \ : \ \delta_M(p', \sigma) = p\}.$$

Ebenso kann für jedes Ereignis $\sigma \in \Sigma_M$ die Menge aller minimalen Phasen bestimmt werden, in welche mit dem Ereignis $\sigma$ gewechselt wird:

$$\Omega_M(\sigma) = \{p \in \zeta_M \mid \sigma \in \Psi_M(p)\}.$$

Da für jedes Ereignis und jede minimale Phase eindeutig der entsprechende Metatyp bestimmt werden kann, wird im Folgenden abkürzend $\Omega(\sigma)$ bzw. $\Psi(p)$ geschrieben.

Mit diesen beiden Mengen sowie den vollständigen Phasenbeziehungen kann für jedes Ereignis $\sigma \in \Sigma$ die Menge der potenziell parallel aufgetretenen Ereignisse bestimmt werden.

**Definition 7.2 (parallele Ereignisklassen)**
*Sei $\sigma \in \Sigma_U$ eine Ereignisklasse eines nutzbaren Metatypen $U \in \mathcal{U}$. Die Menge der parallelen Ereignisklassen zu $\sigma$ ist dann*

$$\Xi(\sigma) = \bigcup_{p \in \Omega(\sigma)} \bigcup_{p' \in \mathcal{P}(p)} \Psi(p')$$

Über die Menge $\Xi(\sigma)$ ist auch die Menge der zu einem Zeitpunkt, beschrieben durch das Ereignis $\sigma$, potenziell gültigen Berechnungsfunktionen festgelegt, da gemäß den Festlegungen aus Abschnitt 6.2.1 auf Seite 111 die Ereignisse einer Komponente die gültigen Berechnungsfunktionen bestimmen.

Neben den parallelen Ereignissen ist für die Analyse des Managements auch die Menge der einem Ereignis vorhergehenden Ereignisse von Interesse, da diese die Berechnungsfunktionen für den Zugriff auf die Vergangenheit einer Eigenschaft bestimmen, die gemäß den Festlegungen in Definition 6.2 auf Seite 112 Teil der Umgebung einer Komponente sind.

Diese Ereignismenge kann ebenfalls durch den deterministischen endlichen Automaten der regulären Ausdrücke der potenziellen Ereignisfolgen bestimmt werden. Für jedes Ereignis $\sigma \in \Sigma_M$ für einen Metatypen $M \in \mathcal{M}$ sei $\bar{\Omega}_M(\sigma)$ die Menge aller minimalen Phasen, aus welchen $\sigma$ herrausführt, also

$$\bar{\Omega}_M(\sigma) = \{p \in \zeta_M \mid \delta_M(p, \sigma) = p',\ p' \in \Omega_M(\sigma)\}$$

Die Menge der Ereignisse vor $\sigma \in \Sigma_M$ wird dann bestimmt durch die Mengen der Ereignisse, die in diese minimalen Phasen führen.

**Definition 7.3 (Ereignisvorgänger)**
*Die Menge der Vorgänger eines Ereignisses $\sigma \in \Sigma_M$ eines Metatypen $M \in \mathcal{M}$ wird bestimmt durch*

$$pred(\sigma) = \bigcup_{p \in \bar{\Omega}_M(\sigma)} \Psi_M(p)$$

Um die zukünftige Entwicklung eines Systems vorhersagen zu können ist auch die Menge der Nachfolger eines Ereignisses von Interesse. Diese Menge kann analog durch den deterministischen endlichen Automaten bestimmt werden als

$$succ_M(\sigma) = \bigcup_{p \in \Omega_M(\sigma)} \{\sigma' \in \Sigma_M \mid \exists \delta_M(p, \sigma')\}$$

Die Menge $succ(\sigma)$ enthält also alle Ereignisse, die auf das Ereignis $\sigma$ direkt folgen können. Für Terminierungsereignisse $\sigma_{term}$ gilt

$$succ_M(\sigma_{term}) = \emptyset$$

Ausgehend von den direkten Nachfolgern eines Ereignisses kann die Menge aller potenziell möglichen Ereignisse $\aleph_M(\sigma)$ nach einem Ereignis $\sigma$ rekursiv bestimmt werden:

a) $\aleph_M^0(\sigma) = succ_M(\sigma)$

b) $\aleph_M^n(\sigma) = \aleph_M^{(n-1)}(\sigma) \cup \bigcup_{\sigma' \in \aleph_M^{(n-1)}(\sigma)} succ_M(\sigma')$

Da die Menge der Ereignisse $\Sigma_M$ eines Metatypen $M$ endlich ist, terminiert dieses rekursive Verfahren notwendigerweise und die Menge $\aleph_M(\sigma) = \aleph_M^\infty(\sigma)$ ist endlich.

Die parallelen Ereignisse, die Ereignisvorgänger und die Ereignisnachfolger bilden somit die Grundlage für die Analyse der Zusammenhänge innerhalb des Managements, wie sie in den folgenden Abschnitten beschrieben wird.

## 7.2 Abhängigkeiten zwischen Eigenschaften

Die Berechnungsfunktionen einer Eigenschaft $E$, wie sie mittels der Funktionsmenge $F_E$ festgelegt sind, beschreiben die potenziellen Zustände einer Kompo-

nente bezüglich der Eigenschaft $E$ als Abbildung aus der Umgebung der Komponente. Gemäß der Definition 6.2 auf Seite 112 besteht die Umgebung einer Komponente zu einem Zeitpunkt aus den Eigenschaften der Komponente selbst, den Eigenschaften aller bekannten, d. h. durch die Belegung von Eigenschaften erreichbaren, Komponenten und der Vergangenheit dieser Eigenschaften.

Auf dieser Grundlage kann für eine Berechnungsfunktion $f_{E,\sigma} \in F_E$ die Menge der abhängigen Eigenschaften ermittelt werden.

**Definition 7.4 (Abhängige Eigenschaften)**
*Sei $f_{E,\sigma} \in F_E$ die Berechnungsfunktion für eine Eigenschaft $E \in \mathcal{E}_M$ eines nutzbaren Metatypen $M \in \mathcal{U}$ bei einem Ereignis $\sigma \in \Sigma_M$.*

*Die Menge der abhängigen Eigenschaften dieser Berechnungsfunktion ist definiert als*

$$\tilde{\mathcal{G}}_{f_{E,\sigma}} = \{E' \in \bigcup_{M \in \mathcal{M}} \mathcal{E}_M \mid E' \textit{ ist in } exp(f_{E,\sigma}) \textit{enthalten}\},$$

*wobei $exp(f_{E,\sigma})$ der definierende Ausdruck der Funktion $f_{E,\sigma}$ ist.*

*Die Menge $\tilde{\mathcal{G}}_{f_{E,\sigma}}$ kann in zwei Teilmengen zerlegt werden, die lokalen Abhängigkeiten $\tilde{\mathcal{L}}_{f_{E,\sigma}}$ und die nicht-lokalen Abhängigkeiten $\tilde{\mathcal{N}}_{f_{E,\sigma}}$, wobei eine Eigenschaft $E' \in \tilde{\mathcal{G}}_{f_{E,\sigma}}$ genau dann in $\tilde{\mathcal{L}}_{f_{E,\sigma}}$ enthalten ist, wenn für eine Komponente $C$ mit $M \mapsto C$ gilt, $E' \in \mathcal{E}_C$, sonst ist $E' \in \tilde{\mathcal{N}}_{f_{E,\sigma}}$.*

Die Abhängigkeiten der Eigenschaften gemäß obiger Definition sind statisch festgelegt. Für die Analyse der dynamischen Zusammenhänge der Eigenschaften müssen die Abhängigkeiten verfeinert werden.

Die Berechnungsfunktionen $f_{E,\sigma} \in F_E$ beschreiben die Zustandsveränderungen in Abhängigkeit von den Ereignissen, welche für die Komponente aufgetreten sind. Die Abhängigkeiten einer Eigenschaft zu einem Zeitpunkt $t$ ergibt sich somit aus der entsprechenden gültigen Berechnungsfunktion, die gemäß $\gamma_E$ ausgewählt wird, sowie der Belegung der strukturbeschreibenden Eigenschaften, welche in diese eingehen. Die Analyse der Abhängigkeiten wird daher von den Eigenschaften zu den Berechnungsfunktionen der Eigenschaften verfeinert.

Für die statische Analyse der zeitlichen Abhängigkeiten einer Berechnungsfunktion $f_{E,\sigma}$ lassen sich, wie in Definition 7.4 festgelegt, zwei Fälle unterscheiden:

1. **Lokale Abhängigkeiten**
   Als lokale Abhängigkeiten werden die Abhängigkeiten der Berechnungs-

funktion $f_{E,\sigma}$ bezeichnet, die sich auf Eigenschaften derselben Komponente beziehen, also in der Menge $\mathcal{E}_C$ enthalten sind. Für diese Abhängigkeiten lassen sich wiederum zwei Fälle unterscheiden, wobei im Folgenden $E' \in \mathcal{E}_C$ die abhängige Eigenschaft sei:

**a)** Falls eine Berechnungsfunktion $f_{E',\sigma} \in F_{E'}$ existiert, so heißt diese Abhängigkeit *simultan*. Die Eigenschaft $E'$ wird in diesem Fall gleichzeitig mit $E$ neu belegt.

**b)** Falls keine Berechnungsfunktion $f_{E',\sigma} \in F_{E'}$ existiert, so heißt diese Abhängigkeit *nicht-simultan*.

**2. Nicht-lokale Abhängigkeiten**
Nicht-lokale Abhängigkeiten ergeben sich aus der Verwendung von Eigenschaften anderer Komponenten, die in der Umgebung enthalten sind. Da zum Zeitpunkt der statischen Analyse der Spezifikation die Komponenten nur in Form ihrer Metatypen existieren, können diese Abhängigkeiten nur durch Abschätzungen erfasst werden.

Durch die Spezifikation der Ereigniszusammenhänge und der potenziellen Ereignisfolgen kann die Menge der nicht-lokal abhängigen Funktionen allerdings beschränkt werden.

Mit diesen Festlegungen kann ein Abhängkeitsgraph für die Berechnungsfunktionen konstruiert werden. Die Knoten dieses Graphen werden durch die Berechnungsfunktionen $f_{e,\alpha}$, $\forall e \in \mathcal{E}_M$, $\forall \alpha \in \Sigma$ gebildet. Die Kanten des Graphen ergeben sich aus den Abhängigkeiten $\tilde{\mathcal{G}}$ und den obigen Unterscheidungen.

**Definition 7.5 (Funktionaler Abhängigkeitsgraph)**
*Ein funktionaler Abhängigkeitsgraph $\mathfrak{A}_\mathcal{U} = \langle V, E \rangle$ einer Menge von nutzbaren Metatypen ist definiert als:*

*a)* $V = \{f_{e,\sigma} \neq \perp \mid \forall e \in \bigcup_{U \in \mathcal{U}} \mathcal{E}_U \wedge \forall \sigma \in \bigcup_{U \in \mathcal{U}} \Sigma_U\}$.

*b)* $E \subseteq V \times V$, *wobei* $(f_{e,\sigma}, f_{e',\sigma'}) \in E$, *mit* $e \in \mathcal{E}_U$, $\sigma \in \Sigma_U$ *mit* $U \in \mathcal{U}$ *genau dann, wenn* $e' \in \tilde{\mathcal{G}}_{f_{e,\sigma}}$ *und*

1. *$e'$ eine Eigenschaft derselben Komponente ist, also $e' \in \tilde{\mathcal{L}}_{f_{e,\sigma}}$ und $\sigma = \sigma'$ ist, die Abhängigkeit also simultan ist.*
2. *$e'$ eine Eigenschaft derselben Komponente ist, also $e' \in \tilde{\mathcal{L}}_{f_{e,\sigma}}$ und $\sigma'$ in einem der Präfixe von $\sigma$ – $pre(\sigma)$ – enthalten ist.*
3. *$E'$ eine Eigenschaft eines Metatypen $U' \in \mathcal{U}$ ist und $\sigma' \in \Xi(\sigma)$ ist.*

**Beispiel 7.1**
Sei eine Spezifikation mit drei Metatypen $\mathcal{M} = \{M_1, M_2, M_3\}$ gegeben, mit der Hierarchie $M_2 \leftarrow M_1$ und $M_3 \leftarrow M_1$. Die Ereignismengen seien

$$\Sigma_{M_1} = \Sigma_{M_2} = \Sigma_{M_3} = \{i, r(p \ : \ M_1), t\}, \text{ mit } R_{M_1} = R_{M_2} = R_{M_2} = i \ r * \ t.$$

Als Eigenschaft von $M_1$ sei $E_1$ mit $dom(E_1) = I\!N$ und den Berechnungsfunktionen

$$E_1 = \begin{cases} 1 & : \text{ bei } i \\ E_1 + 1 & : \text{ bei } r \end{cases}.$$

festgelegt.

Als Eigenschaften von $M_2$ seien $E_2$ mit $dom(E_2) = M_1$ und $E_3$ mit $dom(E_3) = I\!N$ und den Berechnungsfunktionen

$$E_2 = \ p \ \ : \ \text{ bei } r \ \text{, sowie } E_3 = E_2 \vdash E_1$$

festgelegt.

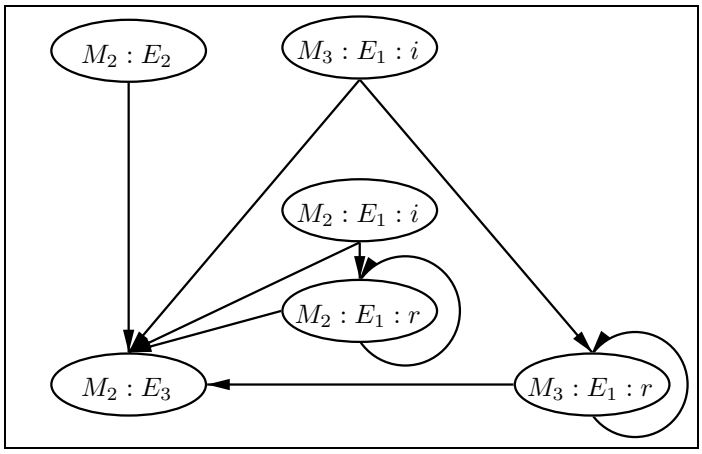



Abbildung 7.1: Der Abhängigkeitsgraph für die Spezifikation aus Beispiel 7.1.

Daraus ergibt sich der in Abb. 7.1 dargestellte funktionale Abhängigkeitsgraph $\mathfrak{A}_{\mathcal{U}}$, wobei die Menge der nutzbaren Metatypen in diesem Fall $\mathcal{U} = \{M_2, M_3\}$ ist. Die Berechnungsfunktionen für $E_1$ sind in den Metatypen $M_1$ und $M_2$ jeweils lokal abhängig. Die Abhängigkeit von $E_3$ in $M_2$ zu $E_1$ ist dagegen nicht-lokal, da über die Eigenschaft $E_2$ zugegriffen wird. Da der Metatyp $M_1$ abstrakt ist, ergibt sich eine Abhängigkeit sowohl zu $E_1$ in $M_2$ als auch zu $E_1$ in $M_3$.

Der funktionale Abhängigkeitsgraph beschreibt die Zusammenhänge der Berechnungsfunktionen für die Eigenschaften der Komponenten des Managements und

somit die Zusammenhänge der Eigenschaften selbst. Diese Zusammenhänge bilden die Grundlage der Propagierung von Veränderungen im System, wie sie in Abschnitt 6.2.1 auf Seite 111 beschrieben wurden.

Der funktionale Abhängigkeitsgraph ist nicht notwendigerweise zyklenfrei. Die Zyklen des Graphen sind potentielle Zyklen in den Berechnungen der Eigenschaften, welche zur Ausführungszeit eines Systems auftreten.

Diese Zyklen müssen geeignet aufgebrochen werden, da eine solche zyklische Berechnung nicht terminiert. Ein Zyklus in einer Eigenschaft $E$ wird durchbrochen, indem die Vergangenheit der Eigenschaft eingesetzt wird, d. h. falls die Berechnung einer Eigenschaft $E$ einer Komponente $C$ zum Zeitpunkt $t$ von der Belegung der Eigenschaft $E$ zum Zeitpunkt $t$ ($\tilde{Z}(C, E, t)$) selbst abhängt, so wird dieser Ausdruck als $\tilde{Z}(C, E, pred(t))$ ausgewertet. Die Zyklen können in drei unterschiedliche Klassen unterschieden werden:

1. **Direkte Zyklen**
   Ein direkter Zyklus ist eine direkte Abhängigkeit einer Eigenschaft $E$ von sich selbst, die Berechnungsfunktion $f_{\sigma,E}$ für eine Ereignis $\sigma$ enthält in diesem Fall direkt die Eigenschaft $E$; dies entspricht einer Schleife im funktionalen Abhängigkeitsgraphen $\mathfrak{A}_U$

2. **Lokale Zyklen**
   Ein lokaler Zyklus ist ein Zyklus, welcher innerhalb der simultanen Berechnungsfunktionen eines Ereignisses $\sigma$ auftritt. Seien $f_{\sigma,E_1}$ und $f_{\sigma,E_2}$ Berechnungsfunktionen zweier Eigenschaften derselben Komponente, mit $f_{\sigma,E_1} = o(E_2)$ und $f_{\sigma,E_2} = o(E_1)$, dann wird diese Situation als lokaler Zyklus bezeichnet.

3. **Nicht-lokale Zyklen**
   Zyklen, welche nicht unter einen der ersten beiden Fälle fallen, werden als nicht-lokale Zyklen bezeichnet.

Die ersten beiden Arten der Zyklen, die direkten und die lokalen Zyklen, können statisch anhand des funktionalen Abhängigkeitsgraphen erkannt werden, die Letzteren sind statisch nur potenziell ermittelbar und müssen daher zur Laufzeit des Systems behandelt werden.

Die Aktionen des Managements, welche der Durchsetzung von Managemententscheidungen dienen, sind von der Belegung der Eigenschaften der Komponenten abhängig und werden bei einer Veränderung in der Belegung der Eigenschaften ausgelöst. Diese Zusammenhänge werden durch die Erweiterung des Graphen der funktionalen Abhängigkeiten erfasst.

**Definition 7.6 (Vollständiger Abhängigkeitsgraph)**
*Ein vollständiger Abhängigkeitsgraph* $\mathfrak{V}_{\mathcal{U}} = \langle V, E \rangle$ *einer Menge von nutzbaren Metatypen ist definiert als:*

*a)* $V = \{f_{e,\sigma} \neq \bot \mid \forall e \in \bigcup_{U \in \mathcal{U}} \mathcal{E}_U \wedge \forall \sigma \in \bigcup_{U \in \mathcal{U}} \Sigma_U\} \cup \bigcup_{U \in \mathcal{U}} \mathcal{A}_U$.

*b)* $E \subseteq V \times V$, *wobei* $(v, v') \in E$, *genau dann, wenn*

1. $(v, v') \in E_{\mathfrak{A}_U}$, *wobei* $\mathfrak{A}_U = \langle V_{\mathfrak{A}_U}, E_{\mathfrak{A}_U} \rangle$.
2. $v = f_{e,\sigma} \notin \bigcup_{U \in \mathcal{U}} \mathcal{A}_U$, $v' \in \bigcup_{U \in \mathcal{U}} \mathcal{A}_U$ *und* $e$ *in* $C_v$ *enthalten ist, wobei* $C_v$ *die Definition der Bedingung der Aktion* $v$ *ist.*

Anhand des vollständigen Abhängigkeitsgraphen können Eigenschaften erkannt werden, deren Zustand nicht genutzt wird. Die Nutzung einer Eigenschaft wird dabei definiert durch den direkten oder indirekten Einfluss auf die Auslösung von Aktionen, die Erzeugung neuer Komponenten und die Erzeugung von Ereignissen. Diese Einflüsse können anhand des vollständigen Abhängigkeitsgraphen und dem Wissen über die Berechnungsfunktionen ermittelt werden.

Ungenutzte Eigenschaften sind im Allgemeinen ein Hinweis auf Fehler in einer Spezifikation, da diese Eigenschaften keinen Beitrag zu den Entscheidungen des Managements liefern.

## 7.3 Analyse dynamischer Aspekte

Neben der Analyse der Abhängigkeiten zum Zeitpunkt der Generierung des Systems ist auch die Erfassung von Abhängigkeiten zur Laufzeit des Systems notwendig. Insbesondere die Beziehungen zwischen den Eigenschaften der Komponenten müssen zur Laufzeit erfasst werden, da diese die Grundlage der Propagierung von Änderungen bilden, wie sie in Abschnitt 6.2.1 auf Seite 111 beschrieben wurde und die ein wesentliches Charakteristikum der beschriebenen Spezifikationstechnik ausmacht.

### 7.3.1 Auswirkungen der Eigenschaftszusammenhänge

Diese Beziehungen basieren auf der Nutzung von Eigenschaften durch Berechnungsfunktionen. Eine Berechnungsfunktion, welche eine Eigenschaft nutzt, muss – nach den Regeln der Propagierung – erneut ausgeführt werden, wenn sich eine der in die Berechnung eingehenden Eigenschaften verändert. Daher muss zur Laufzeit eine Graphstruktur erstellt werden, welche diese Zusammenhänge repräsentiert. Die Knoten dieses Graphen werden durch die Eigenschaften der Komponenten gebildet. Die Menge der Knoten ist dabei aufgrund der Dynamik des Systems variabel. Mit der Entstehung neuer Komponenten oder der Auflösung von Komponenten wird die Menge der Knoten entsprechend verändert.

Die Kanten dieser Struktur ergeben sich durch die Berechnungsfunktionen der Eigenschaften. Wird eine Eigenschaft $E$ berechnet, so wird mit dem Zugriff auf eine andere Eigenschaft $E'$ eine entsprechende gerichtete Kante $(E, E')$ in den Graphen eingefügt. Dieser Graph enthält dabei nur die tatsächliche Nutzung von Eigenschaften. Durch eine Zustandsänderung von $E'$ muss für alle eingehenden Kanten in den Knoten $E'$ die Berechnungsfunktion der entsprechenden Eigenschaften erneut ausgeführt werden.

Wird für eine Eigenschaft $E$ eine Berechnungsfunktion neu ausgeführt, sei es durch das direkte Auftreten eines Ereignisses oder durch Propagierung, so werden die bisher von $E$ ausgehenden Kanten aus dem Graphen entfernt.

Somit ergibt sich für jeden Zeitpunkt $t$ ein Abhängigkeitsgraph über die Eigenschaften der Komponenten des Systems. Durch Veränderung einer Eigenschaft wird ein Knoten dieses Graphen herausgegriffen, welcher als Quelle einer Veränderung des Systemzustandes dient. Abbildung 7.2 auf der nächsten Seite stellt diese Situation für einen Graphen dar.

Für die Propagierung der Veränderungen der Eigenschaften durch den Abhängigkeitsgraphen können zwei Alternativen gewählt werden. Einerseits kann die Propagierung durch Tiefensuche erfolgen. Andererseits kann die Propagierung aber auch durch eine Breitensuche über den Graphen realisiert werden.

Dabei hat die erstere Alternative, die Tiefensuche, den Vorteil, dass sie ohne globales Wissen über den Graphen realisiert werden kann und somit lokal durch die Berechnungsfunktionen erfolgen kann. Allerdings kann dabei der Fall auftreten, dass bei einer Veränderung eventuell Berechnungsfunktionen und Aktionen mehrfach ausgeführt werden. Der einfachste Fall einer derartigen Situation ist in Abb. 7.3 auf der nächsten Seite dargestellt. Die Propagierung einer Veränderung der Eigenschaft 1 durch Tiefensuche führt entweder zu der Reihenfolge $1, 2, 4, 3, 4$ oder zu der Reihenfolge $1, 3, 4, 2, 4$. In beiden Fällen wird die Veränderung der Eigenschaft 4 erst in zwei Schritten erreicht.

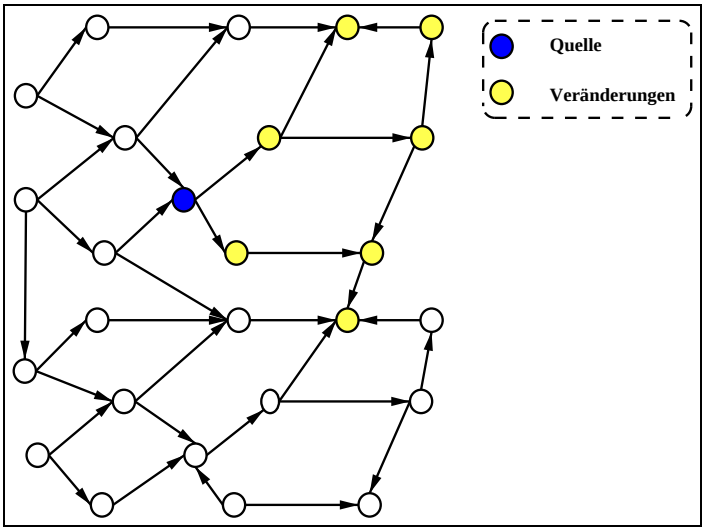



Abbildung 7.2: Propagierung von Veränderungen.

Diese Situation kann durch Breitensuche gelöst werden. Eine Breitensuche würde in diesem Fall zu der Reihenfolge $1, 2, 3, 4$ oder $1, 3, 2, 4$ führen und somit die Berechnung der Veränderung der Eigenschaft 4 in einem Schritt durchgeführt. Der Nachteil einer Breitensuche ist allerdings ein erhöhter Aufwand für die Propagierung der Veränderungen. Daher sollte diese Alternative nur gewählt werden, wenn die Propagierung durch Tiefensuche zu fehlerhaften Berechnungen führt. Es ist daher notwendig, diese als *Rauten* bezeichneten Situationen im Abhängigkeitsgraphen zu erkennen.

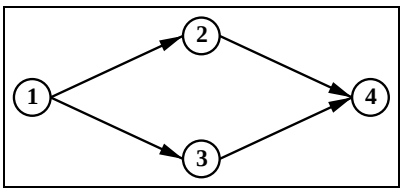



Abbildung 7.3: Ein Abhängigkeitsgraph mit Rauten.

Eine Analyse des Abhängigkeitsgraphen zur Ausführungszeit des Systems zur Erkennung dieser Situationen ist allerdings ebenfalls mit erhöhtem Aufwand verbunden und es ist daher für eine effiziente Realisierung notwendig, diese Analyse bereits zum Zeitpunkt der Generierung des Managements durchzuführen. Diese statische Analyse kann auf dem vollständigen Abhängigkeitsgraphen $\mathfrak{V}$ erfolgen. Der Abhängigkeitsgraph zur Ausführungszeit des Systems ist eine Spezialisierung des vollständigen Abhängigkeitsgraphen der statischen Analyse. Durch den

vollständigen Abhängigkeitsgraphen werden die Berechnungsfunktionen, welche potenziell in Rauten liegen, markiert. Bei der Generierung des Managements wird die Propagierung von Veränderungen der markierten Berechnungsfunktionen dann durch Breitensuche realisiert.

Zwei Knoten $v_1$, $v_2 \in V_{\mathfrak{V}}$ des vollständigen Abhängigkeitsgraphen $\mathfrak{V} = \langle V_{\mathfrak{V}}, E_{\mathfrak{V}} \rangle$ sind genau dann Endpunkte einer Raute, wenn die Anzahl der Wege zwischen $v_1$ und $v_2$ größer Eins ist, also

$$raute(v_1, v_2) \Leftrightarrow |W(v_1, v_2)| \geq 2, \text{ für } v_1 \neq v_2.$$

Wenn zwei Knoten $v_1$, $v_2 \in V_{\mathfrak{V}}$ die Endpunkte einer Raute bilden, so werden alle Knoten auf allen Pfaden von $v_1$ nach $v_2$ markiert. Diese markierten Knoten können potenziell an einer Raute im konkreten Abhängigkeitsgraphen enthalten sein. Sind bei der Propagierung von Veränderungen zur Laufzeit mehrere markierte Knoten in Folge im konkreten Abhängigkeitsgraphen enthalten, so wird an diesen Stellen mittels der Breitensuche propagiert.

### 7.3.2 Analyse von Ausführungsfäden

Bereits in Abschnitt 5.3.2 auf Seite 95 wurde der Zusammenhang zwischen Ereignissen, Komponenten und Ausführungsfäden beschrieben. In den Spezifikationen sind diese Zusammenhänge durch die Ereigniszusammenhänge und die Ereigniserzeugung in Berechnungsfunktionen und Aktionen gegeben. Diese dynamischen Zusammenhänge der Spezifikation werden einerseits zur Laufzeit analysiert, um die Restriktionen bezüglich der Reihenfolge von Ereignissen durchzusetzen. Andererseits können die Zusammenhänge auch statisch analysiert werden, um die Korrektheit der Abläufe zu untersuchen.

#### Dynamische Analyse der Ereignisbedingungen

Neben den Abhängigkeiten der Eigenschaften eines Systems müssen zur Laufzeit auch die Restriktionen bezüglich der Ereignisse durch das Management überprüft werden. Diese Restriktionen werden entsprechend der Definition 5.5 auf Seite 84 durch die regulären Ausdrücke für die Reihenfolge der Ereignisse und die Ereignisbedingungen beschrieben. Für Erstere ist eine Überprüfung auf der Basis des entsprechenden deterministischen endlichen Automaten die effizienteste Lösung. Der Zustand des Automaten wird aus diesem Grund als Eigenschaft der Komponente aufgefasst, deren Berechnungsfunktion sich aus der Zustandsübergangsfunktion des deterministischen endlichen Automaten ergibt.

Die Zulässigkeit eines Ereignisses ist somit in jedem Fall vom Zustand der Komponente abhängig, entweder aufgrund der Ereignisbedingungen oder aber auf-

grund der Reihenfolgerestriktion, welche durch den Automatenzustand repräsentiert wird.

Somit muss für ein aufgetretenes Ereignis die Belegung der Eigenschaften der Komponente überprüft werden und auf dieser Basis eine Entscheidung für die weiteren Berechnungen des Managements getroffen werden. Ist das entsprechende Ereignis zulässig, so werden die entsprechenden Berechnungsfunktionen ausgeführt. Falls aber der Zustand der Komponente das Ereignis nicht zulässt, so muss das Management entsprechend der Semantik des Ereignisses reagieren, d. h. das Ereignis wird entweder verworfen oder der Ausführungsfaden, in dessen Kontext das Ereignis ausgelöst wurde, wird entsprechend blockiert, bis die Komponente einen Zustand erreicht hat, welcher das Ereignis zulässt.

Dazu müssen bei einer Veränderung des Zustandes einer Komponente die Ereignisbedingungen blockierter Ereignisse erneut überprüft werden. Für das Management ist es daher notwendig, die blockierten Ereignisse einer Komponente zu kennen, um bei einer Zustandsänderung eine Prüfung der Ereignisbedingungen durchführen zu können.

### Statische Analyse dynamischer Aspekte

Für die Generierung des Managements sind neben den Abhängigkeiten zwischen den Eigenschaften der Metatypen auch die Abhängigkeiten der Ereignisse von Interesse. Die Ereigniszusammenhänge ergeben sich zum Einen direkt aus der Spezifikation der Ereignisse durch die Relation $\mathcal{Z}$ (vgl. Definition 5.5 auf Seite 84). Mittels dieser Beziehung werden die direkten Ereignisabhängigkeiten der äquivalenten Ereignisse beschrieben. Zum Anderen ergeben sich Ereigniszusammenhänge durch die Verwendung von ereigniserzeugenden Ausdrücken in den Aktionen des Systems. Diese können als bedingte Ereigniszusammenhänge betrachtet werden, wohingegen Erstere als unbedingte Ereigniszusammenhänge betrachtet werden können.

Die Bedingungen dieser Klasse von Ereigniszusammenhängen werden auf der Basis der Eigenschaften und somit als Bedingungen bzgl. des Zustands der Komponenten formuliert. Die Erzeugung eines bedingten Ereignisses ist daher von der Belegung der Eigenschaften abhängig, welche selbst wiederum direkte oder indirekte Folge eines Ereignisses ist. Die Zustandsänderung, die sich als direkte oder indirekte Folge eines Ereignisses ergibt, wird durch die Berechnungsfunktionen für dieses Ereignis beschrieben.

Aus den Berechnungsfunktionen ergeben sich aber auch die Zusammenhänge gemäß der Propagierung von Zustandsänderungen. Damit wird durch ein Ereignis indirekt eine Zustandsänderung ausgelöst.

Diese Zusammenhänge werden in Ereignisgraphen zusammengefasst. Dieser Graph kann für jedes Ereignis $\sigma \in \Sigma$ gebildet werden. Die Knoten dieses Graphen werden durch Paare von Ereignissen und Komponenten gebildet. Dabei wird von der Komponente $C$ ausgegangen, für welche das Ereignis $\sigma$ aufgetreten ist. Ausgehend von diesem Ereignis wird der Graph erweitert um die direkt durch Ereigniszusammenhänge, Zustandsänderungen und Aktionen ausgelösten Ereignisse.

Dieser Graph, der sich ergibt, wenn man von einem gegebenen Ereignis $\sigma \in \Sigma$ rekursiv alle Abhängigkeiten zu anderen Ereignissen verfolgt, repräsentiert die Ereignisfolge, welche durch das Auslösen dieses Ereignisses entsteht und beschreibt die Ereignisfolge der potenziellen Wege eines Ausführungfadens durch das Management mit dem Eintrittspunkt $\sigma$.

**Beispiel 7.2**
Gegeben sei die folgende Spezifikation der Metatypen „Aktivität“ und „Knoten“ zur Erzeugung einer neuen Aktivität auf einem Knoten. Die potenzielle Ereignisfolge der Aktivität ist gegeben durch

$$run\ running\ (start\ cont)^*\ done,$$

wobei das Startereignis *run* mit dem ausführenden Knoten parametrisiert ist und das Ereignis *start* mit einer Aktivitätsklasse. Für die Aktivitäten seien weiterhin folgende Ereigniszusammenhänge spezifiziert:

$$\begin{aligned} start(class) &\rightarrow create@\texttt{node} \\ running &\rightarrow done@\texttt{node}, \end{aligned}$$

wobei `node` eine Eigenschaft mit einer Berechnungsfunktion, welche die Eigenschaft mit dem Parameter des Ereignisses *run* belegt, ist. Über eine Aktion wird für die Aktivität zunächst die Initialisierung durchgeführt und nach deren Abschluss das Ereignis *running* erzeugt.

Die potenzielle Ereignisfolge eines Knotens ist durch

$$init\ (create\ done\ ready)^*\ term$$

gegeben, wobei das Ereignis *create* mit einer Aktivitätsklasse parametrisiert ist. Für die Ereigniszusammenhänge gilt:

$$\begin{aligned} create(class) &\rightarrow ready@self \\ ready &\rightarrow cont@\texttt{caller}, \end{aligned}$$

wobei `caller` eine Eigenschaft mit dem Typ Aktivität ist, die mit dem Ereignis *create* mit dem Aufrufer belegt wird. Weiterhin sei für die Komponenten des Metatypen „Knoten“ eine Eigenschaft `last` gegeben, welche bei dem Ereignis *create*

mit einer neu erzeugten Komponente der Klasse des entsprechenden Parameters belegt wird.

Aus dieser Spezifikation kann nun der Ereignisgraph aus Abb. 7.4 abgeleitet werden. Ausgehend vom Ereignis *start* einer Aktivität wird gemäß der Spezifikation zunächst das Ereignis *create* des Knotens ausgelöst. Durch dieses Ereignis wird die Eigenschaft `last` mit einer neuen Aktivität belegt, wobei die Erzeugung einer Aktivität mit der Auslösung des Startereignisses einhergeht. Außerdem ist durch die Ereigniszusammenhänge festgelegt, dass das Ereignis *ready* des Knotens ausgelöst wird. Damit wird der Ausführungfaden blockiert, bis das Ereignis *done* für den Knoten ausgelöst wird.

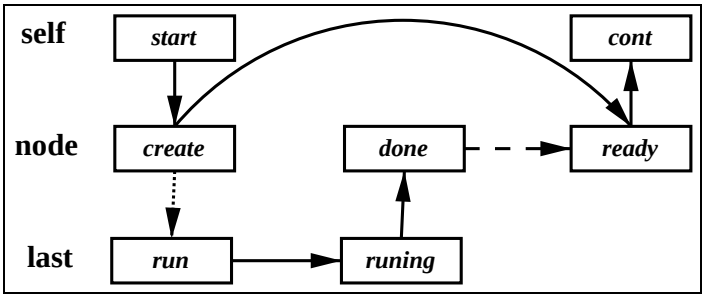



Abbildung 7.4: Ereignisgraph der Spezifikation aus Beispiel 7.2.

Die neu erzeugte Aktivität führt nach dem Startereignis *run* die Aktion zur Initialisierung aus und schließt diese mit dem Ereignis *running* ab, womit durch die Ereigniszusammenhänge das Ereignis *done* für den Knoten ausgelöst und damit die Blockierung des erzeugenden Ausführungsfadens aufgehoben wird, das Ereignis *ready* des Knotens also nun auftritt. Somit wird das Ereignis *cont* für die ursprüngliche Aktivität ausgelöst.

Bei der Konstruktion dieses Graphen wird deutlich, dass die Restriktionen bezüglich der Reihenfolge der Ereignisse erfüllt werden.

Durch die Analyse der Ereigniszusammenhänge können die zeitlichen Zusammenhänge einer Spezifikation untersucht werden. Auf dieser Basis lassen sich beispielsweise Verklemmungen durch die Restriktionen bzgl. der Reihenfolge erkennen. Ebenso werden die Zusammenhänge zwischen den Komponenten dabei nochmals deutlich herausgearbeitet.

Für die Erstellung der Spezifikation eines Managements ist im Allgemeinen der umgekehrte Weg sinnvoll, d. h. ausgehend von einem Ereignisgraphen werden die Ereigniszusammenhänge, Eigenschaften und Aktionen abgeleitet, so dass das gewünschte Verhalten erreicht wird.

## 7.4 Zusammenfassung

Mit den statischen und dynamischen Analysen einer Spezifikation bzw. eines Systems sind die Grundlagen für die Generierung des Managements gelegt. Die entsprechenden Überprüfungen sind einerseits im Generator und andererseits im generierten Management zu realisieren.

Die statischen Analysen der Spezifikation ermöglichen aber auch eine Überprüfung der Korrektheit der Spezifikation, bevor ein System in Ausführung gebracht wird. Die aus den Analysen hervorgehenden Graphen sind für die Erstellung des Managements von Interesse, da diese Zusammenhänge aufdecken und somit die Vorstellungen der Entwickler bezüglich der Abläufe bestätigen oder widerlegen können. Somit sind diese ein wertvolles Hilfsmittel für die Erstellung eines korrekten Managements. Wesentlich ist dabei allerdings, dass die Granularität der Darstellung zum Einen die notwendige Feinheit besitzt, zum Anderen aber auch soweit vergröbert, dass eine Überfrachtung mit Details vermieden wird.

# 8 Generierung flexibler Managementfunktionalität

In diesem Kapitel wird die automatische Transformation einer abstrakten Management-Spezifikation, wie sie in dem Abschnitt I auf Seite 27 beschrieben wurde, in maschinell effizient ausführbaren Code beschrieben. Für diese Aufgabe müssen die Konstrukte der Spezifikationssprache, wie Datentypen, Komponenten, Eigenschaften usw. analysiert und transformiert werden. Für die effiziente Ausführung des generierten Codes auf einer verteilten Hardware-Konfiguration wird aber auch eine Reihe von Basismechanismen benötigt. Der aus der Spezifikation erzeugte Code greift auf die zur Verfügung stehenden Mechanismen zurück, um ein effizientes Management zu realisieren, womit sich eine Trennung von Politik bzw. Managementstrategie und Mechanismus ergibt.

Der Transformationsprozess aus der Spezifikation in maschinell ausführbaren Code wird im ersten Teil dieses Kapitels beschrieben. Die Mechanismen, welche für die Ausführung benötigt werden, sind Gegenstand der Betrachtungen im zweiten Teil dieses Kapitels.

## 8.1 Basismechanismen

Für die Realisierung des Managements eines verteilten, kooperativen Systems sind Basismechanismen notwendig, welche die wesentlichen Fähigkeiten eines Rechensystems bereitstellen. Dabei sind an dieser Stelle die Basismechanismen gemeint, welche notwendig sind, um das generierte Management zur Ausführung bringen zu können. Die Mechanismen, welche dem System bzw. den Problemlösungsverfahren zur Verfügung werden dagegen durch das Management bereitgestellt und sind somit nicht Teil der folgenden Betrachtungen, da diese sich aus der Spezifikation ergeben. Diese Basisfunktionalitäten für das Management entsprechen den grundlegenden Fähigkeiten eines Rechensystems, welche durch die Hardware gegeben sind:

1. **Rechenfähigkeit**
   Die Rechenfähigkeit beschreibt die Möglichkeit, Berechnungen aktiv auszuführen und wird durch den oder die Prozessoren der Hardware realisiert.

2. **Speicherfähigkeit**
   Die Speicherfähigkeit beschreibt die Möglichkeit, Daten in einem System zu halten und geordnet auf diese zuzugreifen. Der Zugriff auf den Speicher des Systems erfolgt über einen Namensraum, die Adressen des Speichers.

3. **Kommunikationsfähigkeit**
   Unter Kommunikationsfähigkeit wird im Allgemeinen die Möglichkeit zum geregelten Datenaustausch zwischen den Teilen eines Systems verstanden. Dabei kann zwischen der stellenlokalen und der stellenübergreifenden Kommunikation unterschieden werden.

Die drei Basisfähigkeiten werden als Grundlage der Realisierung des Managements eines verteilten Systems vorausgesetzt. Für die Nutzung der von der Hardware zur Verfügung gestellten Mechanismen zur Realisierung dieser drei Basisfunktionalitäten durch den generierten Managementcode ist es vorteilhaft, diese Fähigkeiten geeignet zu kapseln und auf einem höheren Niveau zur Verfügung zu stellen.

Die Architektur eines Systems kann somit abstrakt als Schichtenmodell gesehen werden, wie in Abb. 8.1 dargestellt.

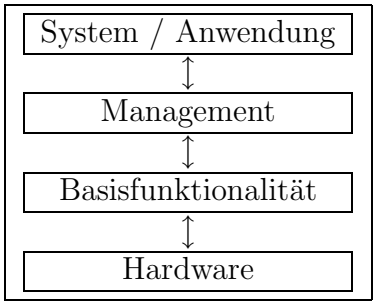



Abbildung 8.1: Schichtenmodell eines Systems.

Diese Architektur ermöglicht den Austausch der Basisfunktionalität und der Hardware des Systems ohne Veränderung des Managements unter der Voraussetzung, dass die Schnittstelle der Basisfunktionalität nicht verändert wird. Diese Architektur kann somit auf verschiedene Weisen realisiert werden. Eine Möglichkeit ist die Verwendung eines Mikrokerns mit Servern zur Realisierung der Basisfunktionalitäten, auf welchen das Management aufsetzt. Eine andere Alternative ist die Verwendung eines klassischen Betriebssystems als Grundlage für die Basisfunktionalität, welche dann als Bibliothek realisiert werden kann.

Die erstere Alternative ist zum Einen performanter und zum Anderen ermöglicht diese Alternative die Beschreibung von klassischen Betriebssystemfunktionalitäten durch die Spezifikation, da bei diesem Ansatz ausschließlich Mechanismen bereitgestellt werden, die Politik des Systems aber veränderlich ist.

Die zweite Alternative dagegen realisiert in einigen Bereichen bereits bestimmte Politiken. So ist das Scheduling der Aktivitäten beispielsweise im Allgemeinen nicht oder nur unwesentlich durch Benutzerprogramme beeinflussbar. Der Vorteil der zweiten Alternative ist dagegen im geringeren Aufwand für die Realisierung zu sehen.

Die Realisierung der Kapselung der notwendigen Basisfähigkeiten wird in den folgenden Abschnitten thematisiert, unabhängig von der gewählten Alternative.

### 8.1.1 Rechenfähigkeit

In klassischen Betriebssystemen wird den aktiven Einheiten ein abstrakter Prozessor zugeordnet. Die abstrakten Prozessoren der einzelnen Aktivitäten werden auf dem realen Prozessor abgebildet, wobei das Betriebssystem den Wechsel zwischen den Aktivitäten durch die als Dispatching und Scheduling bekannten Verfahren realisiert.

Da die Entscheidung über das Scheduling der Prozesse eine der zentralen Aufgaben eines Managementsystems ist, ist es erforderlich, dass diese Entscheidung durch das generierte Management getroffen wird. Zur Basisfunktionalität der Rechenfähigkeit zählt daher der Mechanismus, um abstrakte Prozessoren auf den realen Prozessor abzubilden, also der als Dispatching bekannte Vorgang.

Desweiteren ist für die Aktivitäten eine Basis zur Realisierung der Synchronisation notwendig. Voraussetzung dafür ist eine atomare Operation „Test-and-set-lock“, welche durch die Hardware zur Verfügung gestellt werden muss. Aufbauend auf dieser Hardware-Operation wird auf der Ebene der Basisfunktionalität eine binäre Semaphore realisiert, welche als Synchronisationsmechanismus für das generierte Management dient. Darüber hinaus ist ein Signalisierungsmechanismus notwendig, der es ermöglicht, Veränderungen im System mitzuteilen und dabei blockierte Ausführungsfäden wieder freigibt.

### 8.1.2 Kommunikationsfähigkeit

Die Kommunikationsfähigkeit erfordert einerseits stellenlokale und andererseits stellenübergreifende Kommunikation zwischen den Aktivitäten. Im ersten Fall kann die Kommunikation effizient durch die gemeinsame Nutzung von Speicher-

bereichen realisiert werden, wobei die notwendige Synchronisation mit den Mechanismen aus dem vorherigen Abschnitt gewährleistet werden muss.

Das ISO/OSI-Modell für Netzwerkkommunikation beschreibt sieben Schichten für die Realisierung der stellenübergreifenden Kommunikation (vgl. Abb 8.2). Im Allgemeinen stellt die Kommunikationshardware gemeinsam mit dem Betriebssystem die Schichten 1 mit 5 zur Verfügung. Die darüberliegenden Schichten werden von den Anwendungen gebildet.

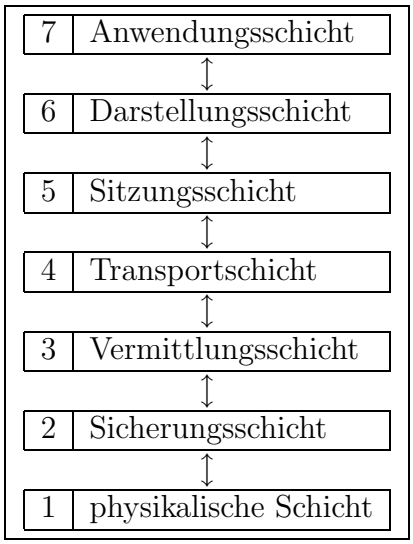



Abbildung 8.2: Schichtenmodell für Netzwerke nach ISO/OSI.

Für das Management eines verteilten Systems wird eine Kommunikationsplattform benötigt, die es ermöglicht, Nachrichten zwischen den Knoten des Systems auszutauschen, also die Schichten bis einschließlich sechs realisiert. Im Rahmen des MoDiS-Projektes wurde durch ein Systementwicklungsprojekt eine nachrichtenorientierte Kommunikationsplattform entwickelt (vgl. [Hag01]), welche einen Nachrichtenaustausch zwischen zwei Knoten über TCP/IP ermöglicht. Diese ist in den Kern des Betriebssystems Linux integriert und somit effizient nutzbar.

Die nachrichtenorientierte Schnittstelle des Mechanismus ermöglicht dabei die Nutzung anderer Kommunikationsplattformen, auch für andere Netzwerk-Topologien, wie sie beispielsweise in Clustern eingesetzt werden. Ein Beispiel für eine derartige Plattform ist MPI (vgl. [SOHL+96]).

### 8.1.3 Speicherfähigkeit

Für die im Management verwaltete Information muss eine geeignete Datenstruktur zur Verfügung gestellt werden. An diese Datenstruktur sind verschiedene Anforderungen zu stellen, die sich aus den Bedingungen des generierten Codes auf der einen Seite und der Verteiltheit auf der anderen Seite ergeben.

1. **Verteiltheit**
   Um in einem verteilten System Information verwalten zu können, müssen bei der Verwaltung der Daten die Aspekte der Verteilung berücksichtigt werden. Eine Lösung mit zentraler Informationshaltung dagegen würde zum Einen einen Flaschenhals darstellen und zum Anderen bei Veränderungen der verteilten Hardware-Konfiguration unter Umständen versagen.

2. **Effizienz**
   Da die Datenstrukturen zur Verwaltung der Managementinformation die zentrale Einheit der Basisfunktionalität darstellen, ist die Performanz der Datenverwaltung von entscheidender Bedeutung. Die Gesamtperformanz des Managements ist stark von der Performanz der Datenverwaltung abängig.

3. **Konsistenz**
   Der Zugriff auf die Information des Managements muss in einem nebenläufigen System derart erfolgen, dass die Konsistenz der gespeicherten Daten durchgesetzt wird. Die notwendige Synchronisation muss dabei allerdings feingranular erfolgen, damit die potenzielle Parallelität nicht eingeschränkt wird. Eine unnötige Beschränkung der Nebenläufigkeit würde die Forderung der Effizienz verletzen.

4. **Verwaltung der Strukturgraphen**
   Einer der zentralen Gesichtspunkte der Spezifikation ist die Strukturierung der Systeme. Einerseits müssen die Strukturgraphen des Systems gespeichert werden. Da die Relationen der Spezifikation als Eigenschaften modelliert werden, kann dabei auf den Mechanismus zur Speicherung von Eigenschaften zurückgegriffen werden.

   Anderseits werden, wie in Abschnitt 6.2.1 auf Seite 111 dargestellt, Veränderungen an der Belegung der Eigenschaften auch durch Veränderungen in der Umgebung einer Komponente ausgelöst, d. h. die Veränderungen werden propagiert. Daher ist es notwendig, bei Veränderungen in der Umgebung eine entsprechende Neuberechnung der semantischen Funktionen der abhängigen Eigenschaften auszulösen. Aus Gründen der Effizienz sollte dieser Vorgang aber nur dann erfolgen, wenn eine Neuberechnung notwendig ist. Dazu muss der konkrete Abhängigkeitsgraph realisiert werden.

Die Speicherung und Verwaltung der Management-Information ist somit der komplexeste Teil der Basisfunktionalität und wird im Folgenden als Wissensbasis bezeichnet.

Die erste Aufgabe der Wissensbasis ist die Speicherung der Zustände einer Komponente. Für jede Eigenschaft $e \in \mathcal{E}_C$ einer Komponente $C$ muss die aktuelle Belegung gespeichert werden. Dies entspricht der Realisierung von $\tilde{Z}(C, e, \sigma_{c,e,now})$. Falls für die Eigenschaft $e$ ein potenzieller Zugriff auf einen Wert der Vergangenheit erfolgt, so muss für diese Eigenschaft auch der vergangene Wert gespeichert werden, also $\tilde{Z}(C, e, \tilde{\sigma}_{c,e,now})$, mit $\tilde{\sigma}_{c,e,now} \in \mathcal{H}_C(now)$ und

- $\delta_e(\theta(\tilde{\sigma}_{c,e,now})) \neq \perp$
- $t(\tilde{\sigma}_{c,e,now}) < t(\sigma_{c,e,now})$ und
- $\nexists \sigma' \neq \sigma_{c,e,now} \ : \ t(\sigma') > t(\tilde{\sigma}_{c,e,now}) \wedge \delta_e(\theta(\sigma') \neq \perp$

Daher müssen zwei Werte gespeichert werden, wobei mit dem Auftreten eines Ereignisses, welches die Belegung der Eigenschaft direkt verändert, die Belegung der entsprechenden Speicherstellen erneuert werden muss.

Der zweite Gesichtspunkt der Verwaltung der Managementinformation ist die Realisierung der Propagierung von Veränderungen. Wenn in einer Berechnungsfunktion einer Eigenschaft $e$ ein Zugriff auf eine weitere Eigenschaft $e'$ erfolgt und sich im weiteren Verlauf $e'$ verändert, so muss die gültige Berechnungsfunktion von $e$ erneut ausgeführt werden, damit der die Belegung von $e$ korrekt bleibt.

In der Wissensbasis wird daher für jede Eigenschaft $e'$ die Menge der abhängigen Eigenschaften $e$ bzw. die Menge deren gültiger Berechnungsfunktionen gespeichert und bei einer Veränderung von $e'$ erneut aufgerufen. Falls durch ein Ereignis die gültige Berechnungsfunktion einer der Eigenschaften $e$ verändert wird und diese Eigenschaft somit nicht mehr von $e'$ abhängt, muss das Ereignis $e$ bzw. dessen Berechnungsfunktion aus der Menge der von $e'$ abhängigen Ereignisse entfernt werden.

Zu jedem Zeitpunkt existiert somit ein konkreter Abhängigkeitsgraph der Eigenschaften, welcher die Propagierung der Veränderungen steuert und der dynamisch an die Veränderungen der Abhängigkeiten angepasst wird. Dieser Graph ist eine Spezialisierung des Abhängigkeitsgraphen, wie er in Abschnitt 7.2 auf Seite 134 eingeführt wurde.

Durch diesen Graphen werden Berechnungsketten im System gebildet, wie in Abb. 8.3 auf der nächsten Seite dargestellt.

Die Berechnung der Ketten kann abgebrochen werden, wenn keine Veränderung in der Belegung der Eigenschaften mehr erfolgt. Dies ist zum Einen der Fall bei der nicht strikten Abfrage von Eigenschaften und zum Anderen, falls eine Berechnung einer Eigenschaft keine Veränderung der Belegung ergibt.

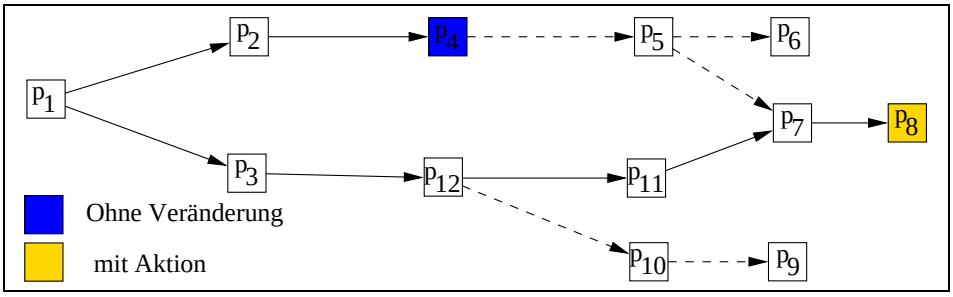



Abbildung 8.3: Berechnungsketten.

Aus der Forderung der Verteiltheit der Wissensbasis ergibt sich, dass dieser Abhängigkeitsgraph ebenfalls verteilt realisiert werden muss. Der Zugriff auf die Eigenschaften, wie auch die Propagierung der Veränderungen von Eigenschaften müssen daher stellenübergreifend durchgesetzt werden. Dazu ist es notwendig, die Eigenschaften und ihre Berechnungsfunktionen in der verteilten Hardware-Konfiguration zu lokalisieren. Für die Eigenschaften ist daher ein global eindeutiger Namensraum erforderlich, in dem auch die Zuordnung der Werte zur Zeit enthalten ist. Die Namen sind aufgebaut aus einem Knotenidentifikator, einem knotenlokalen Identifikator für die Komponente, einem Identifikator für die Eigenschaft und dem Zeitstempel (siehe Abb 8.4). Dabei identifizieren die ersten beiden Anteile eindeutig eine Komponente des verteilten Systems, die Letzteren den Wert einer Eigenschaft zu einem gegebenen Zeitpunkt. Wie aus Abb. 8.4 ersichtlich, ist in den Namen auch der Metatyp der Komponente enthalten. Somit kann anhand des Namens einer Komponente deren Typ ermittelt werden.

| Komponentenidentifikator | | | Eigenschaft | |
|---|---|---|---|---|
| Knoten | lokaler Identifikator | Metatyp | Eigenschaft | Zeitstempel |
| 32 Bit | 32 Bit | | 32 Bit | 32 Bit |

Abbildung 8.4: Realisierung der Identifikatoren für Eigenschaften.

Die Anforderung einer entfernt realisierten Eigenschaft ist ein aufwendiger Vorgang, der auf die Basisfunktionalität zur Kommunikation zurückgreift. Zur Erhöhung der Effizienz der Zugriffe werden die Belegungen der Eigenschaften auf den Knoten repliziert. Die Anforderung des Wertes einer entfernten Komponente wird dann aus den vorhandenen Replikaten erfüllt, falls ein solches bereits auf dem Knoten vorhanden ist. Bei einer Veränderung des Wertes einer Eigenschaft, ist es dann allerdings notwendig, die Replikate über den neuen Wert zu informieren oder diese zu invalidieren. Dieser Vorgang kann in die Realisierung der Propagierung von Veränderungen integriert werden, da mit der Veränderung

der Belegung einer Eigenschaft die abhängigen Eigenschaften erneut berechnet werden müssen.

Im Zusammenhang mit der Propagierung von Veränderungen muss die Wissensbasis auch die in Abschnitt 7.3.1 auf Seite 140 beschriebenen Alternativen für die Propagierung realisieren. Im Allgemeinen werden Veränderungen aus Effizienzgründen durch Tiefensuche verbreitet, für den Fall einer Raute im Abhängigkeitsgraph muss aus semantischen Gründen allerdings auf die Breitensuche gewechselt werden. Für die statisch ermittelten und entsprechend markierten Berechnungsfunktionen der potenziellen Rauten wird durch die Wissensbasis der Wechsel des Propagierungsmechanismus durchgeführt.

Die Entscheidung, auf welchem Knoten eine Komponente realisiert wird, wird durch den Aufruf des Startereignisses der Komponente getroffen. Eine Komponente wird initial auf der Stelle realisiert, auf der das Startereignis auftritt. Das Management hat somit die Möglichkeit, Komponenten auf beliebigen Stellen zu erzeugen. Die Entscheidung, auf welchem Knoten eine Komponente realisiert wird, ist allerdings nicht statisch, sondern kann durch das Management dynamisch verändert werden. Dieser als Migration bezeichnete Vorgang wird durch eine Operation der Wissensbasis realisiert. Mit einer Migration verändert sich aber der Name der Komponente und somit die Namen aller Eigenschaften der Komponente. Andere Komponenten, welche eine Referenz über den Namen auf die migrierte Komponente halten, haben somit einen ungültigen Namen. Daher muss der bisherige Knoten der Komponente eine Abbildung des alten auf den neuen Namen realisieren und bei einer Anfrage an diese Komponente den Anfrager über die Namensänderung informieren.

Die Konzepte einer effizienten Wissensbasis wurden im Rahmen eines Systementwicklungsprojektes (siehe [Pre02]) realisiert und bilden gemeinsam mit der Kommunikationsplattform die Basis des prototypischen Generators, der in Kapitel 9 näher beschrieben wird.

## 8.2 Transformationen

Damit aus einer abstrakten Spezifikation ausführbarer Code erzeugt werden kann, müssen sowohl die Funktionalität als auch das Typsystem in ein semantisches Äquivalent transformiert werden. Für diese Transformation ist eine Übersetzung der funktionalen Anteile der Spezifikationssprache notwendig. Für diese Aufgabe existieren eine Reihe von Ansätzen, wie sie z. B. in [Mac98] beschrieben werden.

Für die Realisierung der abstrakten Spezifikation müssen zunächst die Datensorten der Spezifikationssprache realisiert werden. Bei dieser Aufgabe können drei Arten von Datensorten unterschieden werden:

1. **Einfache Datensorten**
   Die einfachen Datensorten, wie Zahlen, Zeichen bzw. Zeichenketten und Wahrheitswerte, können direkt im Speicher der Hardware-Konfiguration realisiert werden.

2. **Zusammengesetzte Datensorten**
   In die Klasse der zusammengesetzten Datensorten fallen komplexere Datenstrukturen, wie Tupel, Listen, Mengen und andere abstrakte Datensorten.

3. **Komponenten und Komponentenklassen**
   Die Umsetzung der Metatypen bzw. ihrer Ausprägung zur Laufzeit, d. h. die Komponenten und Komponentenklassen müssen zur Laufzeit repräsentiert werden. Sowohl Komponenten als auch Komponentenklassen werden durch die jeweiligen Eigenschaften repräsentiert. Somit ist die Aufgabe bei der Transformation in diesem Fall die Repräsentation der Eigenschaften der Komponenten.

Um eine Management-Spezifikation in Ausführung zu bringen, muss aber auch das Verhalten der Komponenten und ihrer Komponentenklassen in ausführbaren Code überführt werden. Die Dynamik des Systems ergibt sich aus einer Abfolge von Ereignissen, welche eine Veränderung des Zustandes einzelner Komponenten und somit des gesamten Systems auslösen. Der veränderte Systemzustand wiederum veranlasst eine Reaktion des Managements durch die Auslösung von Aktionen. Die Realisierungsaufgabe in diesem Feld entspricht also der Umsetzung der Ereignisse, der Berechnungsfunktionen und der Aktionen.

Beide Realisierungsaufgaben, sowohl der Datensorten, als auch der Funktionalität des Managements, sind unter der Randbedingung der Nebenläufigkeit und der Verteiltheit der Systeme zu betrachten.

### 8.2.1 Zielsprache der Transformation

Für die Transformation der abstrakten Spezifikation eines Managements in semantisch äquivalenten, ausführbaren Code ist es notwendig, eine geeignete Repräsentation dieses Codes zu wählen. Die Zielmaschine, für welche der Code erstellt wird, ist gemäß der von-Neumann-Architektur strukturiert, welche aus den Einheiten Rechenwerk, Leitwerk, Speicherwerk sowie Ein- und Ausgabewerk besteht. Diese Einheiten können jeweils durch entsprechende Befehle gesteuert werden. Die Befehle der Einheiten werden durch Programmiersprachen beschrieben. Der Begriff Programmiersprache umfasst dabei sowohl Maschinensprache als auch höhere Konzepte der Programmierung, wie imperative, objektorientierte, logische oder funktionale Sprachen.

Für die Realisierung der Transformation in ausführbaren Code auf dieser Grundlage stehen somit unterschiedliche Alternativen zur Verfügung, welche jeweils unterschiedliche Vor- und Nachteile aufweisen.

1. **Realisierung durch Hochsprachen**
   Bei der Realisierung des Managements durch eine Hochsprache muss zunächst zwischen Sprachen, welche die Architektur der Maschine wiederspiegeln, und solchen, die von dieser abstrahieren, unterschieden werden. In letztere Kategorie fallen funktionale und logische Sprachen, in Erstere insbesondere die imperativen Sprachen. Die objektorientierten Sprachen nehmen eine Zwitterstellung ein, da diese einerseits durch die Konzepte der Objektorientierung ein maschinenunabhängiges Konzept realisieren, andererseits die Methoden der Objekte aber im Allgemeinen imperativ beschrieben werden.

   Allen Hochsprachen gemeinsam ist zum Einen die Notwendigkeit einer weiteren Transformation in Maschinensprache und zum Anderen die Unabhängigkeit von der exakten Architektur der Zielmaschine. Der erste Punkt stellt dabei den Nachteil der Hochsprachen dar, da die zusätzliche Transformation erstens nur eine eingeschränkte Möglichkeit zur Optimierung des Managements bietet und zweitens eine eventuelle Fehlerquelle darstellt. Diese beiden Nachteile treten umso deutlicher zu Tage, je höher das Konzept der Sprache ist.

   Der zweite Punkt dagegen ist ein Vorteil, den die Transformation in eine Hochsprache bietet, da somit auf triviale Art und Weise die Möglichkeit zur Unterstützung heterogener Architekturen gegeben ist. Diese Unabhängigkeit von der Architektur nimmt dabei mit der Abstraktionsebene der Sprache zu.

2. **Realisierung durch Maschinensprache**
   Eine Realisierung durch Maschinensprache ermöglicht die direkte Ausnutzung der Fähigkeiten der vorhandenen Maschine durch das Management. Somit ist die Transformation der Spezifikation in optimierten, effizienten Code möglich.

   Allerdings ist es dabei notwendig, die Codeerzeugung für jede Plattform entsprechend neu zu beschreiben, wodurch die Unterstützung heterogener Plattformen erschwert wird.

Die beschriebenen Vor- und Nachteile der Realisierungsalternativen sind graphisch in Abb. 8.5 auf der nächsten Seite dargestellt.

Eine weitere zentrale Bedingung für die effiziente Realisierung des Managements ist die Verknüpfung des erzeugten Managementcodes mit dem Code des Pro-

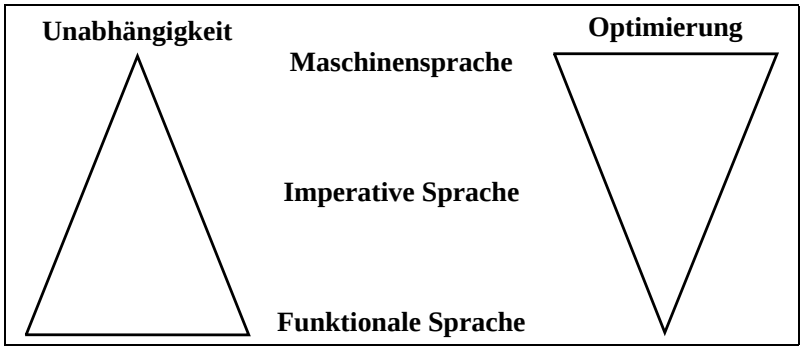



Abbildung 8.5: Realisierungsalternativen der Transformation.

blemlösungsverfahrens. Diese Problemstellung wird im Allgemeinen als *Binden* bezeichnet (vgl. [Lev99]).

Das Lösungsverfahren für eine Problemstellung wird entsprechend der Konzepte durch die Komponentenklassen des Systems beschrieben. Für die Realisierung der Komponenten ist eine Transformation der Komponenten in ausführbaren Code notwendig. Diese Transformation kann als Berechnungsfunktion von Eigenschaften bzw. Klasseneigenschaften beschrieben werden. Der ausführbare Code einer Komponente wird somit durch eine Eigenschaft der Komponente repräsentiert. Aus dieser Tatsache ergibt sich, dass für den Code ein Datentyp definiert sein muss.

Der Code der Komponenten und der Code des Managements müssen strukturiert miteinander in Beziehung gebracht werden, so dass ein Gesamtsystem entstehen kann. Die Kommunikation von dem Komponentencode zum Management wird durch die Ereignisse gebildet. Aus dem Komponentencode muss somit die Auslösung von Ereignissen möglich sein. Die Kommunikation ausgehend vom Management zum Code der Komponenten bedingt die Möglichkeit des Instanzierens von Code, d. h. das Ausführen des Codes und die Einflechtung von Code in den Komponentencode, wie in Abb. 8.6 auf der nächsten Seite dargestellt. Dabei muss auch die Schnittstelle zu den Hardware-Ressourcen geeignet integriert werden.

Daraus ergibt sich die Forderung, dass Managementcode, Komponentencode und die Schnittstelle zur Hardware effizient miteinander in Beziehung gesetzt werden können.

Um dabei die Vor- und Nachteile der Realisierungsalternativen gegeneinander abzuwägen, bietet sich die Transformation in eine maschinennahe, aber Architekturunabhängige Zwischensprache an, welche es einerseits ermöglicht, den effizienten Code zu erzeugen, diese aber andererseits nicht an eine spezielle Architektur gebunden ist. Der gemeinsame Nenner aller Zielarchitekturen ist die Architektur

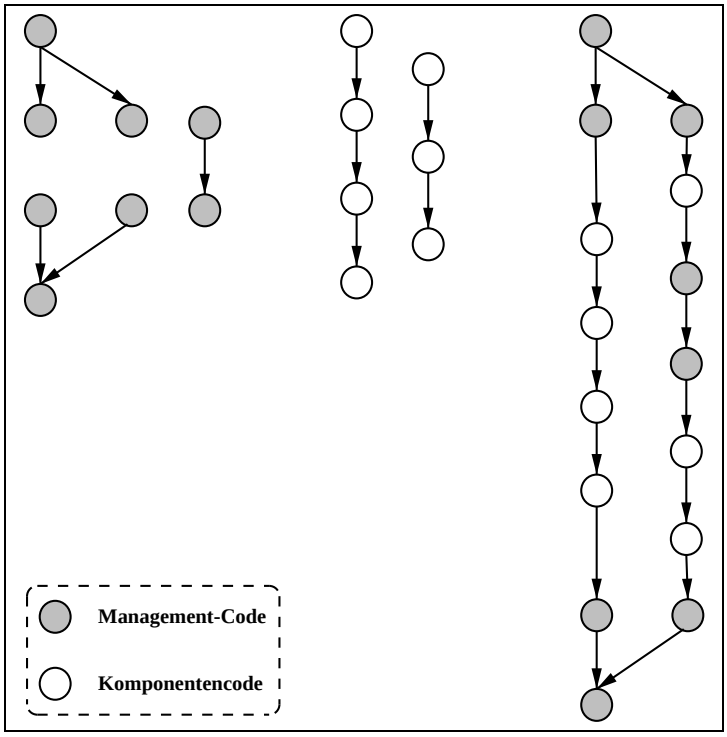



Abbildung 8.6: Synthese von Management und Komponentencode.

nach von Neumann, die sich in den imperativen Sprachen wiederspiegelt. Für die Zielsprache der Transformation wird daher eine imperative Zwischensprache gewählt.

Beispiele für derartige Zwischensprachen sind die Register Transfer Language, die in der GNU-Übersetzer Familie (vgl. [GCC01]) eingesetzt wird, die Intermediate Language des .NET-Frameworks von Microsoft (vgl. [Han02]) oder der Byte-Code von Java (vgl. [LY99]). Letztere ist dabei insbesondere für die plattformunabhängige Ausführung des Codes gedacht und wird im Allgemeinen durch eine virtuelle Maschine ausgeführt.

Die im Folgenden beschriebenen Transformationen gehen von der abstrakten Spezifikation aus und überführen die in dieser enthaltenen Konstrukte in Elemente einer allgemeinen prozeduralen Sprache. Die Übersetzung der Elemente dieser Sprache in ausführbaren Code ist eine Aufgabe, für die effiziente Realisierungen existieren. Eine Transformation auf reale Maschinensprache wäre zwar ebenso möglich, würde aber einen höheren Aufwand in der Realisierung der Transformation bedeuten, sowie einen Verlust an Portierbarkeit auf andere Hardwarekonfigurationen. Der andererseits mögliche Gewinn an Performanz in der Ausführung

des generierten Systems wäre aber nicht von entsprechender Größenordnung, um von Relevanz zu sein.

### 8.2.2 Realisierung von Datensorten

Die Transformation der Spezifikation erfordert zunächst die Realisierung der Datensorten der Spezifikation. Die Inkanationen der Elemente der Datensorten wird durch die Speicherfähigkeit der Maschine, in der von-Neumann-Architektur durch das Speicherwerk repräsentiert, realisiert. Dieser Speicher muss entsprechend der Datensorten durch die einzelnen Elemente geprägt werden.

Der Speicher einer Maschine dieser Architektur ist in Zellen fester Größe strukturiert, wobei jede Zelle durch eine Adresse angesprochen werden kann. Eine Adresse ist dabei ein Element aus $I\!N_0$. Ein Datenobjekt einer bestimmten Größe, die als vielfaches der Zellengröße beschrieben wird, das in diesem Speicher realisiert ist, wird durch die Adresse der ersten Zelle identifiziert. Für jedes Datenobjekt $d$ liefert die Funktion $loc(d)$ die Anfangsadresse des Objektes im Speicher.

Mit den skizzierten Mitteln kann die Realisierung auf einem Knoten der verteilten Hardwarebasis beschrieben werden. Für die Verteilung der Information über die Stellengrenzen hinweg sind zusätzliche Maßnahmen notwendig. Die Basis dafür wird durch die Kommunikationsfähigkeit der Realisierungsbasis (vgl. Abschnitt 8.1.2 auf Seite 149) gebildet. Für die Realisierung der Verteiltheit existieren im Wesentlichen zwei Alternativen. Einerseits kann die Operation, welche auf den Daten ausgeführt wird, auf den Knoten, welcher die Daten hält, verlagert werden. Dies entspricht dem Vorgehen bei einem entfernten Prozeduraufruf. Andererseits können die Daten auf den Knoten transportiert werden, welcher die Operation auf diesem ausführt. Diese Alternative kann beispielsweise durch einen verteilten, gemeinsamen Speicher umgesetzt werden.

Die Datensorten einer Spezifikation werden durch die Menge $\tilde{\mathcal{D}}$, bzw. der Erweiterung der Elemente um das undefinierte Element $\perp$ zur Menge $\mathcal{D}$, beschrieben. In der Realisierung der Datensorten ist es notwendig, für jede Sorte die Größe der Daten dieser Sorte zu kennen. Dazu wird für jede Sorte die Funktion $size : \mathcal{D} \rightarrow I\!N$ definiert. Die Datensorten können folgendermaßen unterschieden werden:

1. **Einfache Datensorten**
   Die einfachen Datensorten einer Spezifikation repräsentieren die Sorten der Maschine. Die Prägung des Speichers gemäß dieser Basissorten ist durch die Maschine festgelegt. In der Realisierung durch eine Zwischensprache müssen diese Datensorten geeignet beschreibbar sein bzw. müssen in der Festlegung der Zwischensprache geeignet repräsentiert werden. Die Größe der Basissorten ist daher maschinenabhängig.

2. **Tupel**
   Tupel repräsentieren eine Zusammenfassung von mehreren Datensorten. Ein Tupel aus $n$-Elementen ist definiert als $\langle D_1 \times \ldots \times D_n \rangle$ mit $D_i \in \mathcal{D}$. Durch ein Tupel wird ein Abschnitt des Speichers der Größe $\sum_{i=1}^{n} size(D_i)$ geprägt. Die Elemente des Tupels werden in diesem Speicherbereich linear aufeinanderfolgend angelegt. Sei $d$ ein Datenobjekt des Typs

   $$V = \langle D_1 \times \ldots \times D_n \rangle.$$

   Dann gilt $loc(sel_{V,i}(d)) = loc(d) + \sum_{j=1}^{i-1} size(D_j)$.

3. **Abstrakte Datentypen**
   Abstrakte Datentypen wurden in die Spezifikationen eingeführt, damit eine effiziente Realisierung neuer Datentypen, insbesondere von Containern für Daten, möglich ist. Die Realisierung der Datenstrukturen und der Funktionen der abstrakten Datentypen liegt daher außerhalb der Spezifikation. Ansätze zur Realisierung von abstrakten Datentypen finden sich zum Beispiel in [OW97].

   Allerdings werden die abstrakten Datentypen und ihre Zugriffsfunktionen derart gekapselt, dass ein transparenter Zugriff in einem verteilten System möglich wird. Bei der Implementierung der Datenstruktur muss daher der Verteilungsaspekt nicht berücksichtigt werden.

4. **Metatypen**
   Ein Metatyp repräsentiert zwei Datensorten, einerseits die Komponenten und andererseits die Komponentenklassen. Eine Komponente wird durch die Wissensbasis und den entsprechenden Identifikator der Komponente (vgl. Abb. 8.4 auf Seite 153) eindeutig beschrieben. Die Größe einer Komponente $k$ ist somit konstant $size(k) = 64\text{Bit}$.

   Die Wertebereiche des Klassenanteils der Metatypen wird analog durch die Wissensbasis repräsentiert. Jede Klasse wird dabei wie eine Komponente realisiert, wobei jede Komponente eine Referenz auf ihre Klasse enthält. Die Ereignisse der Komponenten werden durch Aufrufe der entsprechenden Berechnungsfunktionen der Klassen an diese weitergereicht.

Mit diesen Festlegungen ist die Realisierung der Datensorten durch die Speicherfähigkeit der Knoten festgelegt.

### 8.2.3 Realisierung der funktionalen Ausdrücke

Die Rechenfähigkeit der Hardwarebasis bildet die Grundlage für die Realisierung der funktionalen Ausdrücke der Spezifikationen. Gemäß der Festlegungen der von-Neumann-Architektur können Anweisungen, welche den Programmablauf beeinflussen und Ausdrücke, die Berechnungen durchführen, unterschieden werden.

Die Anweisungen der Zielsprache seien die leere Anweisung, die Zuweisung an Variablen, die bedingte Anweisung und die Anweisungssequenz, mit folgenden Schreibweisen:

```
Stat  ::=  SKIP
       |   Variable := Expr
       |   Stat ; Stat
       |   IF Expr THEN Stat ELSE Stat
```

Die Ausdrücke der Sprache seien entsprechend der Datensorten der Maschine definiert. Dabei werden zumindest die Vergleichsoperatoren $=$, $\neq$, $<$ und $>$ vorausgesetzt, sowie die logischen Operationen $\wedge$, $\vee$ sowie $\neg$. Darüber hinaus sind Aufrufe von Funktionen Ausdrücke.

Für die Transformation der funktionalen Ausdrücke der Spezifikation wird für jeden Ausdruck $E$ eine Anweisung Stat($E$) erzeugt, durch die eine Variable Var($E$) mit dem Ergebnis des Ausdrucks $E$ belegt wird.

1. **Alternativenauswahl**
   Die Alternativenauswahl wird durch den Ausdruck

   $$E \hat{=} if\ C_1\ :\ E_1\ \ldots if\ C_n\ :\ E_n\ else\ E_{n+1}$$

   beschrieben. Die Umsetzung in die imperative Sprache Stat($E$) ist dann:

```
Stat(C1);
IF Var(C1) THEN
  Stat(E1);
  Var(E) := Var(E1)
ELSE
  Stat(C2);
  IF Var(C2) THEN
    ...
  ELSE
    Stat(Cn);
    IF Var(Cn) THEN
      Stat(En);
      Var(E) := Var(En)
```

**ELSE**
Stat($E_{n+1}$);
Var($E$) := Var($E_{n+1}$)

2. **Variablenbindung**
   Die Umsetzung des Ausdrucks $E \hat{=} let\ v := E' : E''$ zu Stat($E$) wird durch

   Stat($E'$);
   Var($v$) **:=** Var($E'$);
   Stat($E''$);
   Var($E$) **:=** Var($E''$);

   beschrieben.

3. **Ereigniserzeugung**
   Die Ereignisse werden in der Realisierung auf Funktionen abgebildet. Sei $U \in \mathcal{U}$ ein nutzbarer Metatyp und $\sigma \in \Sigma_U$ ein Ereignis dieses Metatyps, so existiert eine Funktion $\sigma : \mathcal{C} \times {P_\sigma}^* \rightarrow I\!B$. Diese Funktionen werden in Abschnitt 8.2.4 auf der nächsten Seite eingehend diskutiert. Damit ergibt sich für den ereigniserzeugenden Ausdruck $E \hat{=} throw\ \sigma(E_1, \ldots E_n)@E_{n+1}$ Stat($E$) zu

   Stat($E_1$);
   ⋮
   Stat($E_n$);
   Stat($E_{n+1}$);
   Var(E) **:=** $\sigma$(Var($E_{n+1}$), Var($E_1$), ..., Var($E_n$))

   Falls ein Ereignis für einen abstrakten Metatypen erzeugt wird, so wird in der Realisierung durch eine Typabfrage das konkrete Ziel ermittelt. Diese Abfrage ist trivial, da der Metatyp einer Komponente Teil ihres Namens ist.

4. **Komponentenerzeugung**
   Die Verwaltung der Komponenten des verteilten Systems wird durch die Wissensbasis realisiert. Der Zugriff auf diese geschieht durch spezielle Funktionen. Die Erzeugung einer Komponente ist somit ein Aufruf der entsprechenden Funktion der Wissensbasis.

Die beschriebene Transformation der funktionalen Ausdrücke ist eine vollständige und korrekte Transformation, die allerdings nicht zwangsläufig effizienten Code erzeugt. Der Code der imperativen Sprache kann jedoch optimiert werden, zum Beispiel durch Variablenfaltung oder Umordnung des Codes (vgl. z. B. [WM97]). Die Funktionen der Spezifikation werden entsprechend in Funktionen der imperativen Sprache abgebildet.

### 8.2.4 Realisierung der Dynamik

Für die Transformation der Dynamikbeschreibung des Managementsystems müssen die entsprechenden Anteile der Spezifikation transformiert werden. Die Realisierung der Berechnungen innerhalb des Managements kann, ebenso wie die Realisierung der Eigenschaften im Speicher, auf die Menge der nutzbaren Metatypen $\mathcal{U}$ bzw. die Komponenten und Komponentenklassen als deren Ausprägungen zur Laufzeit, reduziert werden. Für jedes Element der Menge der nutzbaren Metatypen $U \in \mathcal{U} \subseteq \mathcal{M}$ ergibt sich aus der Spezifikation eine Menge von Ereignissen $\Sigma_U$ und die Menge der Eigenschaften $\mathcal{E}_U$.

Für die Eigenschaften eines Metatypen ist zwischen den Eigenschaften der Komponetenklasse und den Eigenschaften der Komponenten zu unterscheiden, so dass die für einen Metatypen definierten Eigenschaften in zwei disjunkte Teilmengen zerfallen.

Die Auswahl der Berechnungsfunktionen der Eigenschaften ist von den Ereignissen bzw. der Folge der Ereignisse abhängig. Für die Realisierung der Ereignisse müssen in Abhängigkeit von den Ereignissen in geeigneter Weise die Berechnungen der entsprechenden Eigenschaftsfunktionen ausgeführt werden.

Die auszuführenden Berechnungsfunktionen der Eigenschaften beim Auftreten eines Ereignisses $\sigma \in \Sigma_U$ für eine Komponente $U \mapsto C$ ergeben sich aus den simultanen Abhängigkeiten der lokalen Eigenschaften (vgl. Definition 7.4 auf Seite 135).

Die Berechnungen, die bei einem Ereignis $\sigma \in \Sigma_U$ ausgeführt werden, sind somit gegeben durch die Menge aller Berechnungsfunktionen für die Eigenschaften, bei denen sich mit dem Eintreten von $\sigma$ die zugeordnete gültige Berechnungsfunktion ändert. Jedes Ereignis $e$ ist somit mit einer Menge von Berechnungsfunktionen verknüpft, die beim Eintritt von $e$ ausgeführt werden müssen. Für eine effiziente Realisierung des Ereignissystems ist die Reihenfolge der Ausführung der Berechnungsfunktionen von Interesse. Die Ausführungsreihenfolge wird durch die Abhängigkeiten zwischen den Berechnungsfunktionen induziert.

Der simultane Abhängigkeitsgraph $\mathfrak{L}_U(\sigma)$ ist ein Teilgraph des funktionalen Abhängigkeitsgraphen $\mathfrak{A}_\mathcal{U}$, welcher alle Berechnungsfunktionen als Knoten enthält, die mit dem Ereignis $\sigma$ gesetzt werden, sowie die Abhängigkeiten zwischen diesen.

**Definition 8.1 (Simultaner Abhängigkeitsgraph)**
*Ein simultaner Abhängigkeitsgraph $\mathfrak{L}_U(\sigma) = \langle V, E \rangle$, mit $\sigma \in \Sigma_U$, eines nutzbaren Metatypen $U \in \mathcal{U}$ ist definiert als:*

*a)* $V = \{f_{e,\sigma} \neq \perp \mid \forall e \in \mathcal{E}_U\}$.

*b)* $E \subseteq V \times V$, *wobei* $(f_{e,\sigma}, f_{e',\sigma'}) \in E$, *mit* $e \in \mathcal{E}_U$, $\sigma \in \Sigma_U$ *genau dann, wenn* $e' \in \tilde{\mathcal{L}}_{f_{e,\sigma}}$ *und* $\sigma = \sigma'$ *ist, die Abhängigkeit also simultan ist.*

Eine topologische Sortierung des simultanen Abhängigkeitsgraphen liefert die optimale Ausführungsreihenfolge für die beim Eintritt eines Ereignisses auszuführenden Berechnungen, unter der Voraussetzung, dass der simultane Abhängigkeitsgraph zyklenfrei ist.

**Beispiel 8.1**
Sei $U$ ein nutzbarer Metatyp mit den Eigenschaften `a`, `b`, `c` und `d` und den in Abb. 8.7(a) auf dieser Seite dargestellten Berechnungsfunktionen beim Ereignis $\sigma$. Der sich daraus ergebende toplogisch sortierte simultane Abhängigkeitsgraph ist in Abb. 8.7(b) dargestellt.

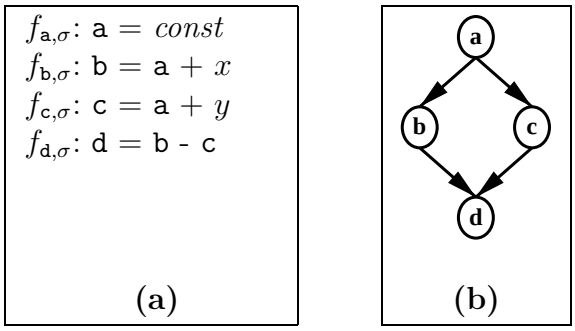



Abbildung 8.7: Toplogisch sortierter, simultaner Abhängigkeitsgraph.

Beim Auftreten des Ereignisses $\sigma$ werden die Berechnungsfunktionen der Eigenschaften gemäß der topologischen Sortierung des simultanen Abhängigkeitsgraphen ausgeführt, im Beispiel 8.1 also in der Reihenfolge $f_{\mathtt{a},\sigma}$, $f_{\mathtt{b},\sigma}$, $f_{\mathtt{c},\sigma}$, $f_{\mathtt{d},\sigma}$ oder der Reihenfolge $f_{\mathtt{a},\sigma}$, $f_{\mathtt{c},\sigma}$, $f_{\mathtt{b},\sigma}$, $f_{\mathtt{d},\sigma}$.

Allgemein ergibt sich für den Code der Berechnungen der Menge der simultanen Berechnungsfunktionen $(\{f_{e,\sigma} \neq \perp \mid \forall e \in \mathcal{E}_U\})$ eines Ereignisses $\sigma \in \Sigma_U$ für einen nutzbaren Metatypen $U \in \mathcal{U}$ folgende Regel:

Stat($f_{e_1,\sigma}$);
⋮
Stat($f_{e_n,\sigma}$);

wobei gilt, dass $\tilde{\mathcal{L}}_{f_{e_1,\sigma}} = \emptyset$ und $f_{e_i,\sigma} \preceq_{\mathfrak{L}_U(\sigma)} f_{e_{i+1},\sigma}$, für $1 < i < n$

Ist die Voraussetzung der Zyklenfreiheit nicht gegeben, so muß der Zyklus durchbrochen werden. Wenn $\mathfrak{L}_U(\sigma)$ in $f_{e,\sigma}$ zyklisch ist, so wird der Zyklus gemäß den Festlegungen aus Abschnitt 7.2 auf Seite 134 durchbrochen, indem für genau ein Auftreten von $f_{e,\sigma}$ in $\mathfrak{L}_U(\sigma)$ die Belegung von $e$ vor dem Eintreten des Ereignisses angenommen wird. Diese Situation ist in Abb. 8.8 dargestellt.

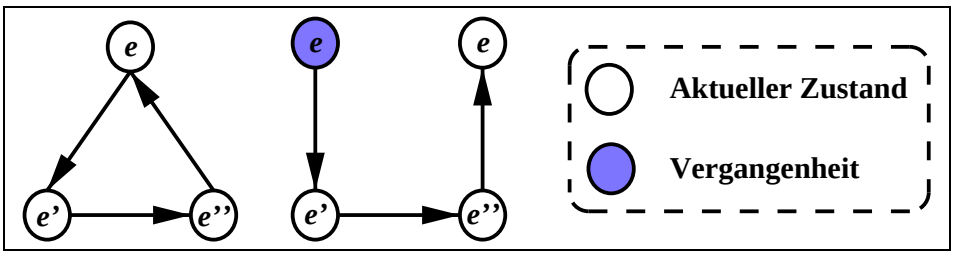



Abbildung 8.8: Zyklus im Abhängigkeitsgraphen.

Mit diesen Festlegungen kann der Code vollständig beschrieben werden, der beim Auftreten eines Ereignisses $\sigma \in \Sigma_U$ für eine Komponente $U \mapsto C$ des Metatypen $U \in \mathcal{U}$ ausgeführt wird. Neben der Erzeugung des Codes der Berechnungsfunktionen $\{f_{e,\sigma} \neq \perp \mid \forall e \in \mathcal{E}_U\}$, ist Code für die Zulässigkeit des Ereignisses zu erzeugen, sowie für die Aktionen.

Zur Durchsetzung der Konsistenz der Belegungen der Eigenschaften einer Komponente müssen die Berechnungsfunktionen atomar ausgeführt werden. Diese Atomarität kann durch die Semaphoren der Basisfunktionalität realisiert werden. Jede Berechnungsfunktion einer Komponente wird durch komponentenspezifische Semaphoren geschützt. Dies ist verklemmungsfrei, da vor dem Verlassen der Berechnungsfunktion die Semaphore freigegeben wird. Die Komponente muss allerdings in jedem Fall in einem konsistenten Zustand sein.

Der vollständige Code der Funktion für ein Ereignis $\sigma$ ergibt sich dann zu

P(sem)
Stat($res(\sigma)$)
Stat($f_{e_1,\sigma}$);
⋮
Stat($f_{e_n,\sigma}$);

Stat($A_1$);
⋮
Stat($A_m$);
V(sem)

wobei $res(\sigma)$ die Ereignisbedingung bezüglich $\sigma$ und die entsprechende Reaktion gemäß der Spezifikation darstellt und $A_1, \ldots A_m$ die Aktionen des Metatypen $U$, für welche gilt:

$$A_i \in \mathcal{A}_U \wedge \exists e \in \{e_1, \ldots, e_n\} : C_{A_i} \text{ enthält } e$$

Für die Realisierung der Ereignisbedingungen und der Reihenfolgenrestriktionen durch den deterministischen endlichen Automaten der potenziellen Ereignisreihenfolge wird auf die Synchronisationsmechanismen der Basisfunktionalität bzw. der Hardware zurückgegriffen, welche die Blockierung von Ausführungsfäden und die Signalisierung von Veränderungen ermöglichen.

Die Berechnungsfunktion für ein Ereignis in einer Komponente ist somit festgelegt. Neben der direkten Veränderung von Eigenschaften durch das Eintreten eines Ereignisses ist allerdings auch die Veränderung von Eigenschaften durch eine Veränderung an einer abhängigen Eigenschaft möglich. Dieser Propagierungsmechanismus wird durch die Wissensbasis unterstützt, indem der entsprechende konkrete Abhängigkeitsgraph verwaltet wird (vgl. Abschnitt 8.1.3 auf Seite 151).

Bei der Propagierung von Veränderungen durch die Wissensbasis ist es notwendig, dass für eine Eigenschaft die letzte Berechnungsfunktion erneut ausgeführt werden kann. Daher wird neben einer Funktion für jedes Ereignis $\sigma \in \Sigma$ eine weitere Funktion für jede Berechnungsfunktion $f_{E,\sigma}$ für alle $E \in \mathcal{E} = \bigcup_{U \in \mathcal{U}} \mathcal{E}_U$ und alle $\sigma \in \Sigma$ erzeugt. Diese Funktionen haben jeweils die Form:

Stat($f_{E,\sigma}$);
Stat($A_1$);
⋮
Stat($A_m$);

wobei $A_1, \ldots A_m$ die Aktionen des Metatypen $U$ sind, für welche gilt:

$$A_i \in \mathcal{A}_U \wedge C_{A_i} \text{ enthält } E$$

Auch diese Funktionen müssen analog zu den Ereignisfunktionen durch Semaphoren zum wechselseitigen Ausschluss geklammert werden. Die Überprüfung der Restriktionen bzgl. der Reihenfolge der Ereignisse ist dagegen in diesem Falle nicht notwendig, da für die Komponente kein neues Ereignis eingetreten ist.

Die Transformation der Aktionen eines Metatypen besteht einerseits aus der Transformation der Aktionsbedingung und andererseits aus dem Code der Aktion selbst. Im Allgemeinen kann für eine Aktion $A \in \mathcal{A}$ also Code der Form

Stat($C_A$);
**IF** Var($C_A$) **THEN** Stat($f_A$) **ELSE SKIP**;

erzeugt werden. Das Ergebnis der Berechnung einer Aktion fließt dabei gemäß der festgelegten Semantik der Spezifikationssprache nicht in die weiteren Berechnungen des Managements ein.

## 8.3 Zusammenfassung

In diesem Kapitel wurde die Transformation von Spezifikationen in den Code einer imperativen Sprache beschrieben. Dieser Code spiegelt die Architektur einer Maschine nach von Neumann wieder, welche die gemeinsame Grundlage der Hardware-Konfigurationen bildet. Dabei wird für jedes Ereignis eines Metatypen eine Funktion generiert, welche die Berechnungen, die mit diesem Ereignis verbunden sind ausführt. Die auszuführenden Berechnungen ergeben sich aus den Restriktionen der Ereignisse, den gültigen Berechnungsfunktionen und den Aktionen der Komponente.

Die Basis des generierten Codes wird durch die Basisfunktionalität bereitgestellt. Der zentrale Aspekt der Basisfunktionalität ist dabei die Verwaltung der Eigenschaften des Managements durch die Wissensbasis, welche auch die Verwaltung der konkreten Systemstrukturen realisiert.

Mit diesen Techniken kann ein Generator realisiert werden, der automatisiert Code für eine gegebene Spezifikation generiert. Dabei ist insbesondere die Verknüpfung von Management und Komponentencode zu beachten. Einerseits muss das Management in der Lage sein, Instanzen aus dem Komponentencode zu bilden und andererseits müssen aus dem Komponentencode heraus Aufrufe in den Managementcode möglich sein, wobei Letzteres durch Ereignisse realisiert wird, welche für das Management externe Ereignisse sind.

# 9 PROMETHEUS – Ein Generator für Managementsysteme

Im Rahmen dieser Arbeit wurde ein prototypischer Generator zur Erzeugung von Managementsystemen aus deren Spezifikation realisiert. In diesem Kapitel wird die von diesem Generator verwendete Spezifikationssprache anhand der Grammatik in BNF-Schreibweise erläutert. Die Sprache folgt dabei den Konzepten aus den Kapiteln 4, 5 und 6. Die vollständige Grammatik sowie die Schlüsselworte der Spezifikationssprache sind in Anhang B auf Seite 253 dargestellt.

Die prototypische Realisierung wird als PROMETHEUS (d. h. „Vorbedacht“) bezeichnet. Die Aufgabe des Programms PROMETHEUS ist die Transformation von abstrakten Spezifikationen in ein effizientes Management. Die dafür notwendigen Umformungen wurden in Kapitel 8 auf Seite 147 diskutiert. Die Zielsprache der Übersetzung von PROMETHEUS ist die Sprache „C“ (vgl. [KR90]), die einerseits als Hochsprache eine einfache Codeerzeugung ermöglicht, andererseits aber die notwendige Maschinennähe aufweist.

Für die Realisierung des Generators wurde das Werkzeug MAX (vgl. [PH93]) eingesetzt, das zur Beschreibung von Programmiersprachen durch eine Kombination aus Prädikatenlogik und Attributgrammatiken entwickelt wurde.

## 9.1 Basisdefintionen

Eine PROMETHEUS Spezifikation besteht aus einer Liste von Deklarationen:

```
Syntaxtree      ::=  [ DeclarationSeq ]
DeclarationSeq  ::=  Declaration { Declaration }
Declaration     ::=  Metatype
                  |  Function
                  |  TypeDef
                  |  AdtDef
```

Eine Deklaration ist entweder ein Metatyp, eine Funktion, eine Typdefinition oder ein abstrakter Datentyp.

## 9.2 Typen in PROMETHEUS

Das Typsystem von PROMETHEUS genügt den in Abschnitt 4.2.3 auf Seite 61 getroffenen Festlegungen. PROMETHEUS kennt als primitive Typen ganze Zahlen (`int`), boolsche Werte (`boolean`) und Zeichenketten (`string`). Weitere Typen können als Typdefinition, abstrakte Datentypen oder Metatypen spezifiziert werden, wobei jede Definition eines neuen Metatypen zwei Datentypen beschreibt, einen Typen für die Komponentenklasse und einen Typen für die Komponenten.

Die Bildung von Strukturen ist durch Tupelbildung möglich.

```
Type   ::=  [ ParameterSeq ]
          |  UsedType
          |  Id < TypeSeq >
```

Alle Typen, die nur über einen Namen angesprochen werden (Basistypen, Metatypen, benutzerdefinierte Typen usw.) werden als UsedType beschrieben. Wird dem Namen das Zeichen „$“ vorangestellt, so bezeichnet der Typ eine Komponentenklasse.

```
UsedType   ::=  [ $ ] Id
```

### Benutzerdefinierte Typen

Benutzerdefinierte Typen sind entweder neue Namen für Typen oder externe Typen. Für externe Typen muss eine Typdefinition außerhalb der PROMETHEUS-Spezifikation existieren.

```
TypeDef   ::=  TYPEDEF Id ( ; | : Type ; )
```

### Abstrakte Datentypen

Abstrakte Datentypen definieren Containertypen mit variablem Inhalt. Der in den Containern enthaltene Typ wird bei der Anwendung des abstrakten Datentyps angegeben, analog zu Templates in objektorientierten Sprachen. In der PROMETHEUS Spezifikation werden nur die Köpfe der Zugriffsfunktionen deklariert. Dies Funktionen müssen extern definiert werden.

```
AdtDef    ::=  ADT Id < IdSeq > AdtBody
```

```
IdSeq      ::=  Id { , Id }
AdtBody    ::=  BEG AdtSeq END ;
AdtSeq     ::=  AdtFunc { AdtFunc }
AdtFunc    ::=  FUNC Id ( ( TypeSeq ) RETURNS Type ; | )
                RETURNS Type ; )
```

## 9.3 Funktionen

Funktionen stellen einerseits Hilfsfunktionalitäten zur Verfügung, anderseits dienen Funktionen als Schnittstelle zu Ressourcen. Interne Funktionen besitzen einen Ausdruck zur Berechnung des Funktionswertes, externe Funktionen definieren nur eine Schnittstelle zur Nutzung externer Funktionalitäten (z. B. Nutzung von Ressourcen). Die externen Funktionen werden in berechnende Funktionen und Messfunktionen unterschieden, wobei Letztere zur Erfassung von Informationen aus der Hardware dienen.

```
Function   ::=  Internal
             |  External
             |  Measure
Internal   ::=  FUNCTION Id ( ( ParameterSeq ) RETURNS
                Type IS Expression ; | ) RETURNS Type IS
                Expression ; )
External   ::=  EXTERN Id ( ( TypeSeq ) RETURNS Type ; |
                ) RETURNS Type ; )
Measure    ::=  MEASURE Id ( ( TypeSeq ) RETURNS Type ;
                | ) RETURNS Type ; )
```

## 9.4 Metatypen

Das zentrale Element einer PROMETHEUS-Spezifikation sind die Metatypen (vgl. Definition 6.4 auf Seite 123). Ein Metatyp definiert einerseits eine Komponentenklasse und andererseits eine Komponente.

```
Metatype   ::=  META Id OptMetaParamSeq OptUsedMetaSeq
                MetaBody
MetaBody   ::=  [ BEG ( BodySeq END | END ) ] ;
```

Ein Metatyp besteht aus Parametern, Ereignissen, potenziellen Ereignisfolgen, Eigenschaften, Relationen und Aktionen.

Eigenschaften, Relationen und Aktionen können sowohl für Komponentenklassen als auch für Komponenten spezifiziert werden.

```
BodySeq    ::=  BodyElem { BodyElem }
BodyElem   ::=  ...
```

### Metatypenhierarchie

Metatypen sind in eine Hierarchie eingeordnet. Diese wird durch eine Liste von übergeordneten Metatypen bei jedem Metatypen definiert. Dabei werden die Bestandteile der Metatypen in der Hierarchie gemäß den Konzepten nach unten weitergegeben. Bei Konflikten bzgl. dieser Vererbung entscheidet die Aufschreibungsreihenfolge in der Liste der übergeordneten Metatypen.

```
OptUsedMetaSeq   ::=  [ EXTENDS UsedMetaSeq ]
UsedMetaSeq      ::=  UsedMeta { , UsedMeta }
UsedMeta         ::=  Id
```

### Parameter von Metatypen

Parameter spezifizieren Informationen, die von außen in eine Komponentenklasse eingebracht werden, z. B. Informationen, die aus dem Quelltext einer Problemlösung gewonnen werden.

Ein Parameter eines Metatypen spezifiziert eine Eigenschaft bzw. eine Relation in der Komponetenklasse. Im Gegensatz zu den explizit definierten Relationen, bei denen die Berechnung in den Kindern spezifiziert wird, wird hier dem Vater eine Menge von Kindern übergeben (als Anwendung eines abstrakten Datentypen). Damit wird bei dem entsprechenden Vater eine Eigenschaft mit dem Typen des (Vater-)Metatypen spezifiziert. Über eine Umbenennung ist es möglich, diese Eigenschaft mit einem neuen Namen und einem anderen Datentypen zu spezifizieren.

```
OptMetaParamSeq  ::=  [ [ ( ] | MetaParamSeq ] ) ]
MetaParamSeq     ::=  MetaParam { , MetaParam }
OptRename        ::=  [ [ Parameter ] ]
MetaParam        ::=  Parameter
                   |  Id -> Type OptRename
```

### Ereignisse

Ereignisse beschreiben, wie in Kapitel 5 auf Seite 75 dargestellt, die Dynamik eines Systems. Bestimmte Ereignisse sind bedingt, d. h. ihr Eintreten ist nur in

bestimmten Zuständen des Systems bzw. der Komponente möglich. Wenn die Bedingung nicht erfüllt ist, so gibt es zwei unterschiedliche Reaktionen, die möglich sind:

a) Verwerfen des Ereignisses („REJECT“).

b) Zurückstellen des Ereignisses („BLOCK“).

Wird das Ereignis blockiert, so wird das Ereignis ausgelöst, sobald sich der Zustand des Systems so verändert hat, dass die Bedingung zu „wahr“ wird. Wird die Reaktion auf ein Ereignis nicht explizit spezifiziert, so wird das Ereignis gemäß der blockierenden Semantik behandelt.

Ein Ereignis kann auch gleichbedeutend mit einem „anderen“ Ereignis (evtl. in einer anderen Komponente) sein. Damit werden Ereigniszusammenhänge beschrieben

Ereignisse transportieren Informationen innerhalb des Systems. Neben der Information, welches Ereignis wann aufgetreten ist, können Ereignisse zusätzliche Information beinhalten, die mittels der Ereignisparameter spezifiziert wird.

```
BodyElem ::= EVENT Id OptParameterSeq Requires
             ConnectEvent ;
           | START Id OptParameterSeq Requires
             ConnectEvent ;
           | TERM Id OptParameterSeq Requires
             ConnectEvent ;
           | SEQUENCE Id : SubSequence OptEndSpec ;
           | SEQUENCE Id : OptEndSpec ;
           | EXACT Id : SubSequence OptEndSpec ;
           | EXACT Id : OptEndSpec ;

Requires          ::= [ [ ( BLOCK ] | REJECT ] | Expression ] |
                      Expression : BLOCK ] | Expression : REJECT ]
                      ) ]
OptParameterSeq   ::= [ ( ( ) | ParameterSeq ) ) ]
ConnectEvent      ::= [ -> UsedEvent OptExpressionSeq @ Expression ]
```

Die Reihenfolge der Ereignisse wird durch eine Sequenz angegeben. Diese Sequenz beschreibt die potenziellen Ereignisfolgen durch reguläre Ausdrücke. Die Spezifikationssprache kennt allerdings keine bedingten Phasenübergänge. Diese müssen durch die in Abschnitt 5.2.2 auf Seite 88 beschriebene Technik simuliert werden.

```
BodyElem ::= SEQUENCE Id : SubSequence OptEndSpec ;
           | SEQUENCE Id : OptEndSpec ;
           | EXACT Id : SubSequence OptEndSpec ;
```

| | | |
|---|---|---|
| | \| | **EXACT** Id **:** OptEndSpec **;** |

| | | |
|---|---|---|
| SubSequence | ::= | Occurunce { Occurunce } |
| Occurunce | ::= | UsedEvent |
| | \| | ( SubSequence ) |
| | \| | Occurunce ∗ |
| | \| | Occurunce \| Occurunce |
| UsedPhase | ::= | Id |

Mit der Festlegung der potenziellen Ereignisfolgen wird auch die Einordnung eines Ereignisses als synchron oder asynchron getroffen. Ein Ereignis ist synchron, wenn es in einer Phase des Metatypen oder in einer der Phase der in der Hierarchie übergeordneten Metatypen enthalten ist. Andernfalls ist das Ereignis asynchron.

Alternativ können auch Start- und Terminierungsereignisse angegeben werden, wobei die weiteren Ereignisse in beliebiger Reihenfolge auftreten können.

### Eigenschaften und Relationen

Sowohl Eigenschaften als auch Relationen (vgl.Kapitel 4 auf Seite 51) bestehen aus einer Datensorte, einem initialen Wert und einer Menge von Berechnungsfunktionen. Diese Funktionen hängen von den Ereignissen ab. Ein Ereignis definiert zugleich eine Eigenschaft in dem als Typ der Relation angegebenen Metatypen, welche in der Spezifikation als „Id_sons“ angesprochen werden kann und die alle Kinder bezüglich dieser Relation enthält. Gemäß den Festlegungen in Abschnitt 4.3 auf Seite 63 sind in PROMETHEUS nur baumartige Relationen zugelassen.

| | | |
|---|---|---|
| BodyElem | ::= | **PROPERTY** Id **RETURNS** Type EmptyExp Updates |
| | \| | **CLASSPROPERTY** Id **RETURNS** Type EmptyExp Updates |
| | \| | **RELATION** Id **RETURNS** UsedType EmptyExp Updates |
| | \| | **CLASSRELATION** Id **RETURNS** UsedType EmptyExp Updates |

| | | |
|---|---|---|
| Updates | ::= | **;** |
| | \| | **IS** UpdateSeq |
| UpdateSeq | ::= | Update { Update } |
| Update | ::= | **ON** UsedEvent **DO** Expression **;** |
| EmptyExp | ::= | [ **:=** Expression ] |

### Aktionen

Aktionen dienen, wie in Abschnitt 6.3.3 auf Seite 121 dargestellt, der Durchsetzung von Managemententscheidungen. Diese Entscheidungen werden durch Eigenschaften der Komponenten bzw. Komponentenklassen repräsentiert. Eine Aktion wird ausgelöst, wenn eine bestimmte Eigenschaft eine Bedingung erfüllt. In diesem Fall wird der Ausdruck der Eigenschaft berechnet. Dieser Ausdruck kann die Erzeugung von Ereignissen oder den Aufruf von (externen) Funktionen beinhalten.

Aktionen lassen sich in ACTION und OBSERVER unterteilen. Erstere werden nur ausgeführt, wenn ein Ereignis eintritt, das den Wert einer Eigenschaft verändert. Observer dagegen werden auch ausgeführt, wenn sich der Wert einer Eigenschaft durch eine Veränderung einer anderen Eigenschaft verändert (Propagierung).

**ACTION** Id [ Expression ] **DO** Expression **;**
| **OBSERVER** Id [ Expression ] **DO** Expression **;**
| **CLASSACTION** Id [ Expression ] **DO** Expression **;**
| **CLASSACTION** Id [ Expression ] **DO** Expression **;**

## 9.5 Ausdrücke

Ausdrücke beschreiben die Berechnungen, die im Management des Systems notwendig sind. Die Konzepte für die Berechnungen des Managements wurden in Abschnitt 6.1 auf Seite 107 eingeführt.

| | | |
|---|---|---|
| ExpressionSeq | ::= | Expression { , Expression } |
| Expression | ::= | ( Expression ) |
| | \| | Constant |
| | \| | IdentNt |
| | \| | BoolExp |
| | \| | PrefixExp |
| | \| | UsedId OptExpressionSeq |
| | \| | Id [ TypeSeq ] |
| | \| | ... |
| UsedId | ::= | IdentNt |
| Id | ::= | IdentNt |

### Konstanten

Konstanten existieren für die vordefinierten Basistypen (`int`, `boolean` und `string`), sowie für die aktuelle Komponente (`SELF`). Außerdem existiert für jeden Datentypen ein undefinierter Wert, wie in Definition 4.8 auf Seite 62 festgelegt, der mit `NIL` bezeichnet wird.

```
Constant ::= INT
           | TRUE
           | FALSE
           | STRING
           | NIL
           | SELF
```

### Zugriff auf Eigenschaften

Der Zugriff auf lokale Eigenschaften erfolgt über den Namen der Eigenschaft. Der Zugriff auf nicht-lokale Eigenschaften geschieht über einen Ausdruck und den Operator „.“, der die Komponente berechnet, welche die Eigenschaft besitzt.

```
Expression ::= Name
             | Expression . IdentNt
```

### Funktionsaufrufe

Funktionen werden über ihren Namen und eine Liste von Parametern aufgerufen. Für die Erzeugung eines neuen (leeren) Containers aus einem abstrakten Datentyp wird eine „Funktion“ aufgerufen, wobei eine Liste der Typen des Containers angegeben wird.

```
Expression ::= UsedId OptExpressionSeq
             | Id [ TypeSeq ]
```

### Bedingte Ausdrücke

Bedingte Ausdrücke bestehen aus einer Liste von Paaren aus Bedingung und Ausdruck. Sobald eine der Bedingungen zu „wahr“ ausgewertet wird, wird der entsprechende Ausdruck berechnet, womit die Berechnung des bedingten Ausdrucks abgeschlossen ist. Sollte keine der Bedingungen zu „wahr“ ausgewertet werden, so wird ein Alternativ-Ausdruck berechnet.

```
Expression ::= ConditionSeq ELSE Expression
```

ConditionSeq ::= Condition { Condition }
Condition ::= IF Expression **:** Expression

Eine besondere Form für die Bedingung in einem bedingten Ausdruck ist die Typabfrage für Metatypen. Da Metatypen in einer Vererbungshierarchie organisiert sind, ist es im Allgemeinen nicht möglich, den exakten Typ eines Ausdrucks statisch zu bestimmen. Daher gibt es einen Ausdruck, der den Typ eines Ausdrucks zur Laufzeit überprüft. Wenn dieser Ausdruck zu „wahr“ ausgewertet wird, so ist im Berechnungsausdruck sichergestellt, dass der Ausdruck aus der Bedingung von dem gegebenen Typ ist.

Expression ::= Expression $-$ > Type

### Variableneinführung

Zur Vereinfachung der Berechnungen können Variablen eingeführt werden. Diese Variablen werden durch die Berechnung eines Ausdrucks mit einem Wert belegt und stehen dann für die Berechnung zur Verfügung.

Expression ::= LET Id := Expression **:** Expression

### Boolsche Ausdrücke

Boolsche Ausdrücke sind als Infix- bzw. Prefix-Ausdrücke definiert. Es existieren die wesentlichen boolschen Operationen ($\wedge$, $\vee$, $\neg$), sowie die Vergleichsoperationen ($=$, $\neq$, $<$, $>$, $\leq$ und $\geq$).

BoolExp ::= Expression ( & Expression | | Expression | >
Expression | < Expression | >= Expression | <=
Expression | = Expression | # Expression )
PrefixExp ::= ! Expression

### Weitere Zugriffsalternativen für Eigenschaften

Neben dem standardmäßigen Zugriff auf Eigenschaften kann auch auf den vergangenen Wert der Eigenschaft ($-$) zugegriffen werden.

Wenn in einem Ausdruck eine Eigenschaft verwendet wird, so wird diese Berechnung bei einer Veränderung der verwendeten Eigenschaft erneut angestoßen (Propagierung). Dieses Verhalten ist nicht immer gewünscht. Soll der Wert einer Eigenschaft nicht strikt abgefragt werden, so wird dem Ausdruck, der die verwendete Eigenschaft beschreibt, das Symbol „^“ vorangestellt.

```
Expression  ::=  Expression −
              |  ^ Expression
```

#### Erzeugung von Ereignissen

Insbesondere in den Berechnungsfunktionen der Aktionen ist es notwendig, Ereignisse zu erzeugen. Die Erzeugung eines Ereignisses kann von einem weiteren Ausdruck gefolgt werden, welcher dann den Wert der Berechnung beschreibt, anderenfalls liefert der Ausdruck den undefinierten Wert `NIL`.

```
Expression  ::=  THROW UsedEvent OptExpressionSeq @
                 Expression
              |  THROW UsedEvent OptExpressionSeq @
                 Expression : Expression
```

#### Erzeugung von Komponenten

Das Management kann auch die Erzeugung einer neuen Komponente veranlassen. Dafür muß die entsprechende Komponentenklasse und eine Liste von Ausdrücken für die Parameter des Startereignisses angegeben werden.

```
Expression  ::=  NEW [ Expression ] OptExpressionSeq
```

Die Erzeugung von neuen Komponentenklassen ist durch eine implizite Funktion mit dem Identifikator des entsprechenden Metatypen möglich.

## 9.6 Notation von Spezifikationen

In dieser Arbeit werden die Spezifikationen im Folgenden in einer an die obige BNF-Grammatik angelehnten Notation dargestellt. Die in dieser Notation verwendeten Schreibweisen sind, sofern nicht selbsterklärend bzw. mit der obigen Grammatik identisch, in der folgenden Tabelle dargestellt.

Einen Ausschnitt einer Spezifikation in dieser Darstellung zeigt die Abb. 9.1 auf der nächsten Seite.

| Symbol | Bedeutung |
|---|---|
| ident ⟨ . . . ⟩ | ein abstrakter Datentyp; |
| [ $\mathrm{id}_1$, . . . $\mathrm{id}_n$ ] | ein Tupel; |
| `ident` | ein Typ; |
| **META** ident . . . ← . . . | ein Metatyp mit Obertypen; |

```
META test
  EVT seta (pa : int) [BLOCK]
  EVT init [BLOCK]
  EVT done [BLOCK]
  SEQ run :
    init (seta)* done
  PROP a → int := ⊥
    init :        0
    seta :        pa
  PROP b → int := ⊥
    init :        add(a, 1)
  PROP c → int := ⊥
    init :        add(a, 1)
  PROP d → int := ⊥
    init :        sub(c, b)
  OBS p[(d≠0)]
    print(d # 0:, b)
  OBS p2[(d=0)]
    print(d = 0:, b)
END test
```

Abbildung 9.1: Beispiel einer Spezifikation

| Symbol | Bedeutung |
|---|---|
| **EVT** *evt* (...) [...] ⇒ $evt_1()$@ident | ein Ereignis mit Parametern, Ereignisbedingung und abhängigem Ereignis; |
| ident | eine Komponentenklasse oder eine Komponenteneigenschaft bzw. Relation; |
| $\text{ident}_{\mathcal{S}}$ | die Menge der Komponenten, welche durch die Relation „ident“ verbunden sind; |
| $\text{ident}_{\mathcal{F}}$ | die Komponenten, welche durch die Relation „ident“ verbunden ist; |
| $\perp$ | das undefinierte Element; |
| $\downarrow$ Ausdruck | der vergangene Wert einer Eigenschaft; |
| $\nearrow$ [ Ausdruck ] | der nicht strikte Wert einer Eigenschaft; |
| Ausdruck $\succ$ Typ | der Operator Ist-Typ-Von. |

## 9.7 Zusammenfassung

Mit dem Werkzeug PROMETHEUS und seiner Sprache steht eine prototypische Realisierung für die automatische Generierung des Managements verteilter, kooperativer Systeme zur Verfügung. Die Realisierung folgt den in dieser Arbeit dargestellten Konzepten. Neben der eigentlichen Transformation einer Spezifikation in ein ausführbares Management gemäß der in Kapitel 8 erarbeiteten Regeln, werden durch den Generator auch eine Reihe von Analysen, wie sie in den Kapiteln 7 und 11 beschrieben wurden, durchgeführt. Die Ergebnisse dieser Analysen fließen einerseits in den generierten Code ein, andererseits werden diese aber auch graphisch ausgegeben, so dass dem Entwickler als Hilfsmittel ein Abhängigkeitsgraph des Managements zur Verfügung stehen. Für die Darstellung der graphischen Ausgaben wird das Werkzeug VCG (**V**isualization of **C**ompiler **G**raphs, vgl. [San95]) verwendet. Die Abbildung 9.2 auf der nächsten Seite zeigt die Komplexität eines solchen Abhängigkeitsgraphen.

Damit diese Komplexität handhabbar bleibt, können Teile dieses Graphen ausgeblendet werden, so dass der Entwickler des Managements nur relevante Teilaspekte betrachten kann. Daneben werden durch den Generator auch graphische Darstellungen der deterministischen endlichen Automaten für die potenziellen Ereignisfolgen eines Metatypen erzeugt und somit die minimalen Phasen der Komponenten sichtbar gemacht.

Somit steht für die Entwicklung einer Spezifikation eines Managementsystems mit PROMETHEUS eine Reihe von Werkzeugen zur Verfügung, welche eine effiziente Arbeitsweise erlauben. Daneben stehen aber auch Werkzeuge zur graphischen Darstellung und Analyse von Ereignisfolgen laufender Systeme zur Verfügung, die eine Suche nach konzeptionellen Fehlern in der Spezifikation ermöglichen.

Die Entscheidung, das Werkzeug MAX zur Realisierung des Prototypen zu verwenden, ermöglichte einerseits eine einfache und effiziente, an den Konzepten orientierte Entwicklungsarbeit und eröffnet andererseits die Möglichkeit, durch die syntaktische Ähnlichkeit von MAX-Beschreibungen und PROMETHEUS-Spezifikationen den Generator durch sich selbst zu beschreiben und somit in einem Bootstrap-Verfahren weiterzuentwickeln.

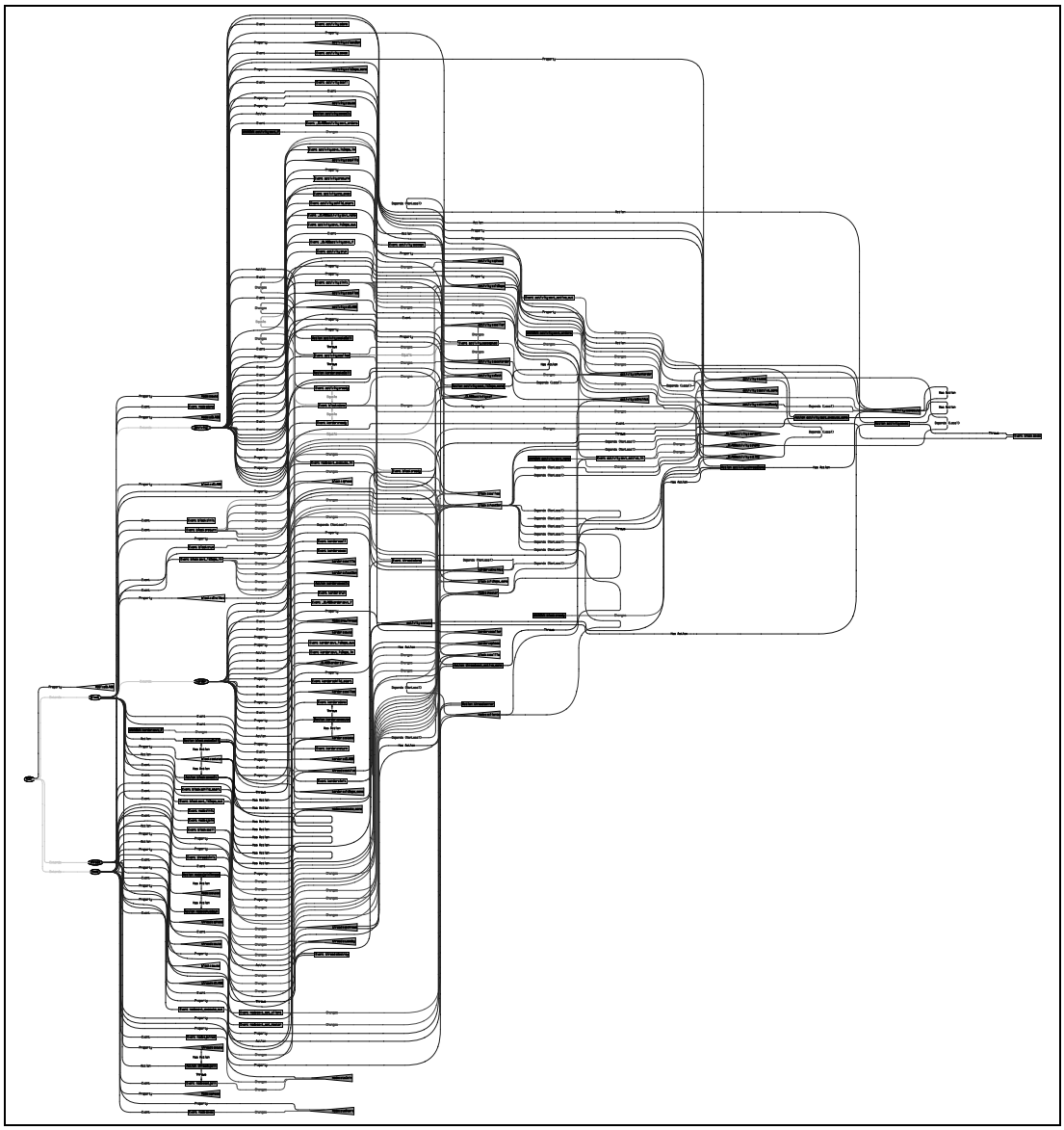


Abbildung 9.2: Ein komplexer Abhängigkeitsgraph.

# 10 Fallbeispiele

Dieses Kapitel diskutiert den Einsatz der erarbeiteten Spezifikationstechniken und des Werkzeugs PROMETHEUS anhand einiger Beispiele. Dabei gliedert sich das Kapitel in drei Teile. Im ersten Teil wird die Spezifikation von Basisdiensten beschrieben, welche in klassischen Systemen durch das Betriebssystem ausgeführt werden. Im zweiten Teil des Kapitels wird MoDiS beschrieben, ein stark strukturierter Ansatz für verteilte kooperative Systeme. Im letzten Teil wird dann ein Laufzeitsystem für CDSL, einer nebenläufigen, verteilten Programmiersprache, als Beispiel einer PROMETHEUS-Spezifikation vorgestellt. Auch dabei wird die Bedeutung der Strukturierung von Systemen nochmals deutlich heraus gearbeitet.

## 10.1 Fallbeispiel Spezifikation von Basisdiensten

Dieser Abschnitt beschreibt die Spezifikation einiger Basistechniken zur Bereitstellung von Rechen- und Speicherfähigkeit, welche üblicherweise durch Betriebssysteme realisiert werden. Zunächst wird die Bereitstellung von virtuellen Prozessoren durch einen Scheduling-Algorithmus dargestellt. Da die prototypische Realisierung der Generierung von Managementfunktionalität auf dem Betriebssystem Linux aufsetzt, kann diese Spezifikation nur auf der Basis einer Simulation realisiert werden. Diese Simulation allerdings kann visualisiert werden und somit beispielsweise in der Lehre zum Einsatz kommen.

### 10.1.1 Scheduling von Prozessen

Der Scheduling-Algorithmus, welcher im Folgenden spezifiziert wird, folgt dem in Linux (vgl. [Bec97]) realisierten Algorithmus für die Threads, welche nach dem *Timesharing*-Verfahren verwaltet werden.

Jeder Thread besitzt eine Priorität (*priority*). Diese kann zur Laufzeit des Threads verändert werden. Außerdem besitzt jeder Thread ein Quantum (*quantum*) an

Rechenzeit. Aus diesen beiden Werten wird die Güte (*goodness*) des Threads berechnet:

$$goodness = \begin{cases} quantum + priority & : \quad quantum > 0 \\ 0 & : \quad \text{sonst} \end{cases}$$

Wenn der Scheduler nun eine Entscheidung trifft, so wählt er den Prozess mit dem höchsten Wert für die Güte aus. Während der Ausführung des Threads wird mit jedem Tick der Uhr das Quantum um Eins erniedrigt. Dem Thread wird der Prozessor entzogen, wenn er entweder kein verbleibendes Quantum mehr hat, der Thread blockiert oder ein blockierter Prozess mit höherer Güte wieder rechenbereit wird.

Die Güte wird neu berechnet, wenn kein rechenbereiter Thread mehr über ein Quantum verfügt. Die Quanten der Prozesse werden dann neu berechnet, wobei sich das neue Quantum als

$$quantum = (quantum/2) + priority$$

ergibt.

Die Spezifikation dieses Scheduling-Algorithmus basiert auf zwei Metatypen, nämlich `node` und `thread`, wie in Abb. 10.1 auf der nächsten Seite dargestellt, wobei die potenziellen Ereignisfolgen nicht spezifiziert wurden.

Der Metatyp `node` repräsentiert den Rechenknoten, auf dem die Komponenten, welche von `thread` abgeleitet wurden, ausgeführt werden. Die Ereignisse des Metatypen `node` sind neben dem Startereignis *init* und dem Terminierungsereignis *done*, das Ereignis *tick*, welches den Ablauf einer Zeitscheibe anzeigt, und das Ereignis *act*, das dem Knoten die Existenz eines neuen Threads anzeigt. Mittels der Eigenschaft „active“ wird aus der Menge der rechenbereiten Threads derjenige ausgewählt, welcher über die höchste Güte verfügt. Die Überprüfung der Güten findet entweder mit dem Erscheinen eines neuen Threads statt, oder aber mit dem Ablauf einer Zeitscheibe. Im ersten Fall wird nur überprüft, ob der neue Thread eine höhere Güte als der laufende Thread besitzt. Im zweiten Fall wird mittels der rekursiven Funktion *getnext* aus der Liste der rechenbereiten Thread derjenige mit höchster Güte ausgewählt. Die Aktion $x$ teilt dem entsprechenden Thread mit, dass er den Prozessor erhalten hat. Falls kein Thread über ein weiteres Quantum verfügt, so veranlasst die Aktion *doRecalc* eine Neuberechnung der Quanten mittels der Funktion *calcquantum*.

Für den Metatypen `thread` sind die Ereignisse *init* als Startereignis, *go* als Signal, dass der Thread den Prozessor erhalten hat, *stop* als Signal, dass dem Thread der Prozessor entzogen wurde, *suspend* als Signal, dass der Thread blockiert wurde, *continue* zum Entblockieren, *nice*, um die Priorität neu zu setzen, sowie *reset* zum Neuberechnen des Quantums definiert. Jeder Thread steht mittels *my_node* in Relation zu seinem Knoten und, falls er rechenbereit ist, auch in der Relation

```
FCT calcquantum : (sons : rel⟨thread⟩, last : thread) → boolean :
  LET T← next(sons) :
    IF (last=⊥) : calcquantum(sons, T)
    IF (T=last) : reset ()@T : true
    ELSE reset ()@T : calcquantum(sons, last)
END calcquantum
FCT getnext : (sons : rel⟨thread⟩, last : thread) → thread :
  LET T← next(sons) :
    IF (T=⊥) : ⊥
    IF (T=last) :
      IF (↗ [T⊢quantum]>0) : T
      ELSE ⊥
    ELSE
      IF (↗ [T⊢quantum]>0) : T
      ELSE getnext(sons, last)
END getnext
META node
  EVT init, done, act (t : thread) ⇒ stop ()@active, tick ⇒ stop ()@active
  PROP active → thread := ⊥
    act :    IF (↗ [t⊢goodness]≤↗ [↗ [active]⊢goodness]) : ↗ [active]
             ELSE t
    tick :   getnext(exec_node_S, current(exec_node_S))
  ACT x[(active≠⊥)]
    go ()@active
  ACT doRecalc[(((active=⊥))∧(count(my_node_S)>0))]
    LET T← current(my_node_S) : calcquantum(my_node_S, T)
END node
META thread
  EVT init (n : node, p : int)⇒ act (self)@n, go, stop, suspend, continue,
  terminate, reset, nice (p : int)
  PROP priority → int := ⊥
    init :   p
    nice :   p
  PROP quantum → int := ⊥
    init :   p
    stop :   add(quantum, −1)
    reset :  add(DIV(quantum, 2), ↗ [priority])
  PROP goodness → int := ⊥
    init :   IF (quantum=0) : 0 ELSE add(quantum, ↗ [priority])
  REL exec_node → node := ⊥
    init :   n
    suspend :  ⊥
    continue :  my_node
    terminate :  ⊥
  REL my_node → node := ⊥
    init :   n
    terminate :  ⊥
END thread
```

Abbildung 10.1: Spezifikation eines Schedulers

*exec_node*. Die Güte wird durch Propagierung automatisch mit der Veränderung der Quanten angepasst.

### 10.1.2 Leichtgewichtige Prozesse und Verteilungsmechanismen

Das folgende Beispiel beschreibt ein System, das Aktivitäten auf eine Menge von Knoten verteilt und jeweils durch Threads realisiert, wobei `pthreads` (vgl. [ANS96]) als „Hardware"-Ressourcen verwendet werden. In bestimmten, durch die Aktivitäten festgelegten, Fällen kann eine Realisierung als sequentieller Ablauf erlaubt sein. Die Metatypen werden in diesem Beispiel nur noch in ihren relevanten Ausschnitten dargestellt.

Für die Auswahl der Realisierung stehen somit drei Alternativen zur Verfügung:

a) Sequentielle Ausführung

b) Ausführung als Thread auf dem lokalen Rechner

c) Ausführung auf einem entfernten Rechner

Die Auswahl zwischen diesen Alternativen wird in dem hier dargestellten Beispiel aus dem Verhältnis von Overhead und vorhergesagter Laufzeit der Aktivität berechnet. Die vorhergesagte Laufzeit einer Aktivität wird als Klasseneigenschaft der Aktivitäten modelliert und über die terminierten Aktivitäten berechnet. Die Berechnungsfunktion dieser Eigenschaft $\boxed{runtime}$ ist:

$$\boxed{runtime} = (\boxed{runtime} + usedtime) \ / \ 2$$

Dabei wird durch *usedtime* die verbrauchte Zeit der zuletzt beendeten Aktivität repräsentiert. Durch diese Berechnungsfunktionen wird ein Durchschnitt über die Laufzeiten der Aktivitäten gebildet, wobei diese gemäß ihrem Alter gewichtet werden.

Die Entscheidung, ob eine Aktivität sequentiell, als Thread oder auf einem entfernten Knoten gestartet wird, wird beim Aufrufer für die neu zu erzeugende Aktivität getroffen und erfolgt durch die Eigenschaft *childNode*. Die externe Funktion *getnode*() liefert dabei den aktuellen, d. h. lokalen Knoten. Im Falle der entfernten Ausführung der Aktivität wird der Knoten nach dem Round-Robin-Prinzip aus der Liste der benachbarten Knoten gewählt, die jeder Knoten als Eigenschaft besitzt.

Durchgesetzt wird die Entscheidung durch die Aktionen *exec* bzw. *startThread* (siehe Abb. 10.2 auf der nächsten Seite). Die Erstere realisiert die sequentielle Ausführung durch Erzeugung der neuen Aktivität und anschließendem Zustellen des Ereignisses *sequential*, wodurch die Aktivität ihre Funktion ausführt. Die

```
META activity[name : string, f : fct]
        ⋮
  PROP [runtime] → int := constant
    terminate :    DIV(ADD(runtime, usedtime), 2)
  PROP usedtime → int := ⊥
    init :         getptime()
    terminate :    SUB(getptime(), usedtime)
  PROP childNode → node := ⊥
    createChild :  IF (↗ [child⊢runtime]≥const_distributed) :
                     next(getnode()⊢clients)
                   IF (↗ [child⊢runtime]≥const_concurrent) :
                     getnode()
                   ELSE
                     IF child⊢seqAllowed :
                       ⊥
                     ELSE
                       getnode()
        ⋮
  ACT exec[((childNode=⊥)∧(state=createChild))]
    sequential ()@NEW child []
  ACT startThread[(childNode≠⊥)]
    runThread (child)@childNode
END activity
```

Abbildung 10.2: Leichtgewichtige Prozesse und Verteilungsmechanismen

zweite Aktion fordert den gewählten Knoten durch das Ereignis *runThread* auf, einen Thread für die Aktivität zu erzeugen.

```
META node
        ⋮
  PROP toStart → activity := ⊥
    runThread :    NEW what []
        ⋮
  ACT run[(toStart≠⊥)]
    NEW thread() [toStart]
END node
```

Abbildung 10.3: Spezifikation von Rechenknoten

Als letzter relevanter Metatyp wird an dieser Stelle der Metatyp *Thread* vorgestellt. Eine Instanz dieses Metatypen erzeugt einen `pthread` mit der Funktion

einer Aktivität und führt diese mittels der externen Funktion *createpthread* aus (siehe Abb. 10.4).

```
META Thread
        ⋮
  PROP toStart → fct := ⊥
    init :          act⊢f
        ⋮
  ACT exec[(toStart≠⊥)]
    createpthread(toStart)
END node
```

Abbildung 10.4: Spezifikation von Threads

### 10.1.3 Ein verteilter gemeinsamer Speicher

Mit den beiden obigen Fallbeispielen wurde die Spezifikation von Basisdiensten zur Realisierung der Rechenfähigkeit eines Systems demonstriert. Das nun folgende Beispiel zeigt eine Spezifikation, welche Speicherfähigkeit in einem verteilten System realisiert.

Durch die gemeinsame Nutzung von Speicherbereichen durch die Aktivitäten eines Systems steht eine einfach zu benutzende Technik für die Interprozesskommunikation zur Verfügung. Diese Technik kann auf verteilte Systeme erweitert werden, wobei man dann von einem verteilten gemeinsamen Speicher (DSM – **D**istributed **S**hared **M**emory) spricht.

In diesem Abschnitt wird die Spezifikation eines solchen verteilten gemeinsamen Speichers diskutiert. Die Spezifikation orientiert sich dabei an den in [RP01] vorgestellten Konzepten für einen seitenbasierten DSM, der auf POSIX-Threads basiert. Dabei wird jede Seite des verteilten Adressraums einem Knoten zugeordnet, welcher für die Verwaltung der Seite verantwortlich ist. Lesende Anforderungen von Aktivitäten anderer Knoten werden durch Replikation erfüllt, wobei die Seite für schreibende Zugriffe gesperrt wird. Erfolgt ein schreibender Zugriff, so werden zunächst alle Replikate der Seite als ungültig markiert, und die Verantwortung für die Seite sowie die Seite selbst wird auf den Knoten, welcher die schreibende Aktivität realisiert, übertragen.

Bei diesem Ansatz wird davon ausgegangen, dass die Anzahl der am System beteiligten Knoten initial bekannt ist. Der für jeden Knoten des Systems bereitstehende virtuelle Adressraum wird zunächst in einen Teil, welcher durch den DSM verwaltet wird, und einen Teil, der durch den Knoten lokal verwaltet wird,

aufgeteilt. Der Anteil, welcher durch den verteilten gemeinsamen Speicher verwaltet wird, wird zunächst zu gleichen Teilen an die beteiligten Knoten zugewiesen. Diese Situation ist in Abb. 10.5 dargestellt.

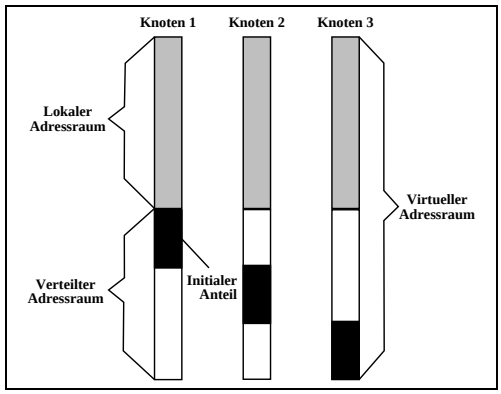



Abbildung 10.5: Die initiale Konfiguration des DSM.

Die Speicheranforderungen der Aktivitäten werden entsprechend der Art der Anforderung aus dem lokalen Adressraum oder dem Anteil des Knotens am verteilten Adressraum realisiert.

Wird von einer Aktivität eine Seite lesend oder schreibend angefordert, die nicht auf dem lokalen Knoten vorhanden ist, so wird diese angefordert. Für diesen Vorgang gibt es eine Reihe von Alternativen:

1. **Broadcast**
   Die Anfrage wird an alle Knoten geschickt. Der Besitzer meldet sich daraufhin beim anfragenden Knoten. Der Nachteil dieses Verfahrens liegt in der Behandlung der Anfrage durch alle Knoten, obwohl nur ein Knoten gesucht wird.

2. **Zentraler Manager**
   Ein zentraler Manager verwaltet die Zuordnungen von Seiten und Besitzern. Alle Anfragen werden dabei an den zentralen Manager gestellt und entweder von diesem beantwortet, oder es wird der Besitzer durch den Manager benachrichtigt. Dieser Ansatz kann zu einem Flaschenhals werden, der die Performanz des Systems beeinträchtigt, da jede Anfrage und jede Änderung der Besitzverhältnisse vom Manager verwaltet werden muss.

3. **Verteilte Verfahren**
   Bei einem verteilten Verfahren kennt jeder Prozess den potenziellen Besitzer einer Seite. Die Anfragen werden an diesen potenziellen Besitzer gerichtet und von diesem an dessen potenziellen Besitzer weitergeleitet, falls er nicht der Besitzer ist. Damit die Listen potenzieller Besitzer nicht zu lang werden

bzw. die Anfrageketten verkürzt werden, kann in periodischen Abständen durch einen Broadcast über die tatsächlichen Besitzverhältnisse informiert werden.

Der in [RP01] vorgestellte DSM *Murks* verwendet eine Abart der zweiten Alternative. In der hier vorgestellten Spezifikation wird dagegen die erste Alternative gewählt, damit ein möglicher Flaschenhals vermieden wird.

Wie in [RP01] dargestellt, kann es bei der Einlagerung von Seiten zu Race-Conditions kommen, wenn während der Einlagerung der Seite eine andere Aktivität auf diese Seite zugreift. Um dies zu verhindern, müssen für den Zeitraum der Einlagerung alle Aktivitäten gestoppt werden.

In der Spezifikation werden die Seiten nicht als eigene Metatypen modelliert. Stattdessen wird ein Typ definiert, welcher Seitenbereiche repräsentiert (vgl. Abb. 10.6).

**TYPE** `addr_t` → `int`
**TYPE** `page_t` → [addr : `addr_t`, size : `int`]

Abbildung 10.6: Typen für die Spezifikation eines DSM.

Jeder Knoten besitzt drei Listen von Seitenbereichen. Die erste Liste repräsentiert die lokalen, nicht-belegten Seiten, entspricht also der Liste der Freibereiche. Die zweite Liste repräsentiert die Seiten, für die der Knoten verantwortlich ist. Die dritte Liste enthält die lokal vorhandenen Seiten, also die Seiten, für die der Knoten verantwortlich ist und die Replikate, die der Knoten besitzt. Die Markierung, welche Seiten schreibbar sind und welche nur zum Lesen zugegriffen werden dürfen, wird durch den entsprechenden Mechanismus des Betriebssystems bzw. der Hardware realisiert. Dieser Teil der Spezifikation ist in Abb. 10.7 auf der nächsten Seite dargestellt.

Für die Speicherverwaltung existieren zunächst drei externe Ereignisse nämlich Speicheranforderung (*alloc*), Speicherfreigabe (*free*) sowie *fault*, das beim Zugriff auf eine nicht vorhandene oder gesperrte Seite erzeugt wird.

Letzteres wird mit der Art des Zugriffs – lesend oder schreibend – parametrisiert (vgl. Abb 10.8 auf Seite 192). War der Zugriff, welcher dieses Ereignis ausgelöst hat, ein schreibender Zugriff und wird die Seite von dem Knoten verwaltet, so wird allen Knoten durch ein Ereignis mitgeteilt, dass diese Seite nicht mehr zugreifbar ist (*invalid*) und anschließend als schreibbar markiert. Falls die entsprechende Seite nicht von diesem Knoten verwaltet wird, so wird allen Knoten über das Ereignis (*broadcast*) mitgeteilt, dass der Knoten die entsprechende Seite benötigt. Falls die Seite nur zum Lesen benötigt wird, erhält der Knoten die Seite vom entsprechenden Verwaltungsknoten als Replikat. Falls die Seite zum Schreiben

```
META DSMnode
  PROP free_list → list⟨page_t⟩ := ⊥
    init :      appfront(free_list, area)
    alloc :     ...
    free :      appfront(free_list, area)
  PROP resp_list → list⟨page_t⟩ := ⊥
    init :      appfront(resp_list, area)
    getPage :   appfront(resp_list, page_t(page, ADD(page, pagesize)))
    broadcast : IF (hasPage≠⊥) :
                  remove(resp_list, hasPage)
                ELSE resp_list
  PROP local_list → list⟨page_t⟩ := ⊥
    init :      appfront(free_list, area)
    getPage :   appfront(resp_list, page_t(page, ADD(page, pagesize)))
    getRepl :   appfront(resp_list, page_t(page, ADD(page, pagesize)))
    broadcast : IF (hasPage≠⊥) :
                  remove(local_list, hasPage)
                ELSE local_list
    Invalid :   IF (hasLocal≠⊥) :
                  remove(local_list, hasLocal)
                ELSE local_list
      ⋮
END DSMnode
```

Abbildung 10.7: Spezifikation eines verteilten gemeinsamen Speichers (Teil I).

angefordert wurde, so sendet der bisherige Verwaltungsknoten zunächst allen anderen Knoten die Mitteilung, die Seite zu invalidieren und überträgt sie dann an den anfordernden Knoten, der damit auch zu deren Verwalter wird.

Mit dieser Spezifikation eines verteilten gemeinsamen Speichers wird auch die Verwendung von Hardware-Ressourcen in Spezifikationen nochmals verdeutlicht. In der Spezifikation werden die verwalteten Einheiten nicht explizit spezifiziert, da ihr Management von der Hardware übernommen wird. Die – hier nicht dargestellten – Zugriffe auf die Funktionalität der Hardware kann durch externe Funktionen erreicht werden.

## 10.2 Fallbeispiel MoDiS-OS

MoDiS (**Mo**del **o**riented **Di**stributed **S**ytems), das am Lehrstuhl XIII der Fakultät der Technischen Universität München entwickelt wird, stellt einen Ansatz zur Entwicklung verteilter kooperativer Systeme da. Das MoDiS gliedert sich technisch in eine Sprache zur Beschreibung von Systemen. Diese als INSEL (**IN**tegration and **SE**peration **L**anguage) bezeichnete Sprache spiegelt die

```
META DSMnode
  EVT alloc (size : int)
  EVT free (area : page_t)
  EVT fault (page : addr_t) ⇒ ready()@SELF
  EVT broadcast (page : addr_t, recv : DSMnode, write : boolean)
  EVT getPage (page : page_t)
  EVT getRepl (page : page_t)
  EVT ready
  EVT Invalid (page : addr_t)
  SEQ all :
    init (fault (getPage | getRepl) ready)* done
          ⋮
  PROP hasPage → page_t := ⊥
    broadcast :   IF write :
                    FindInList(page, ↗ [reps_list])
                  ELSE ⊥
  PROP hasRepl → page_t := ⊥
    broadcast :   IF ¬(write) :
                    FindInList(page, ↗ [reps_list])
                  ELSE ⊥
  PROP hasLocal → page_t := ⊥
    Invalid :     FindInList(page, ↗ [local_list])
  PROP FindPage → page_t := ⊥
    fault :       FindInList(page, ↗ [reps_list])
  PROP InvalidPage → page_t := ⊥
    fault :       IF (FindPage≠⊥) : FindPage
                  ELSE ⊥
  PROP BroadcastPage → page_t := ⊥
    fault :       IF (FindPage=⊥) : page
                  ELSE ⊥
  ACT SendPage[(hasPage≠⊥)]
    getPage (hasPage)@recv
  ACT SendRepl[(hasRepl≠⊥)]
    getRepl (hasPage)@recv
  ACT Invalid[((hasPage≠⊥)∨(InvalidPage≠⊥))]
    sendALLinvalid(hasPage)
  ACT search[(FindPage≠⊥)]
    sendALLbroadcast(hasPage)
END DSMnode
```

Abbildung 10.8: Spezifikation eines verteilten gemeinsamen Speichers (Teil II).

abstrakten Konzepte von MoDiS wieder. Die Sprache INSEL kann als Weiterentwicklung der Sprache ADA (vgl. [Nie90]) verstanden werden. Der zweite Bestandteil des Systems ist die Laufzeitumgebung bzw. das Management, welches als MoDiS-OS bezeichnet wird.

Die Konzepte von MoDiS werden in [SEL+96] und in [Piz99] ausführlich beschrieben. Ein wesentliches Merkmal dieses Ansatzes ist die starke Strukturierung der Systeme.

### 10.2.1 Ziele von MoDiS

Mit dem MoDiS-Projekt wird ein Ansatz verfolgt, der sich zum Ziel gesetzt hat, Systeme gemäß einem festgelegten Konzeptevorrat, welcher die freie Kombinierbarkeit der Eigenschaften und damit ein hohes Maß an Flexibilität bei der Konstruktion verteilter kooperativer Systeme ermöglicht (vgl. [SEL+96]) zu beschreiben. Die Konzepte von MoDiS werden durch die Programmiersprache INSEL realisiert. Diese Sprache zeichnet sich durch die folgenden Eigenschaften aus:

1. **Objektbasiertheit**
   Die Komponenten eines MoDiS-Systems werden objektbasiert beschrieben. Damit ist für jede Komponente innen und außen festgelegt. Die Komponenten sind somit gekapselt und interagieren über festgelegte Schnittstellen.

2. **Schachtelung**
   Die Komponenten sind ineinander geschachtelt. Damit werden die Sichtbarkeiten der Komponenten untereinander beschrieben. Dies führt zu einer starken Strukturierung der formulierten Problembeschreibungen.

3. **Hohes Abstraktionsniveau**
   Die Problembeschreibungen werden auf einem hohen Abstraktionsniveau beschrieben. Dabei werden Realisierungsaspekte nicht berücksichtigt; diese werden durch das Management des Systems entschieden.

Die Komponentenarten in MoDiS werden entsprechend ihrer Bedeutung für die Berechnungen der Problemlösung bzw. der Formulierung der Problembeschreibung unterschieden (vgl. Abb. 10.10 auf Seite 197). Dabei werden zunächst DE-Komponenten (einfache Komponenten mit nur einem **De**klarationsteil) und DA-Komponenten (mit **D**eklarations- und **A**nweisungsteil) unterschieden.

Die DE-Komponenten können in Wert-orientierte Komponenten und Generatoren unterschieden werden. Die Wert-orientierten Komponenten stellen elementare Speicher bereit. Zu dieser Komponentenart zählen z. B. Variablen und Zeiger. Generatoren legen die Eigenschaften einer Teilmenge der Komponenten fest. Jede Komponente wird aus einem Generator erzeugt. Dieses Konzept ist mit dem Klassen und Instanzen-Konzept objektorientierter Sprachen vergleichbar. Allerdings werden Generatoren, anders als üblicherweise Klassen, im laufenden System dynamisch erzeugt und aufgelöst und sind in die Strukturen des Systems eingebunden.

DA-Komponenten sind aus DE- und DA-Komponenten zusammengesetzte Komponenten mit der Fähigkeit, Informationen zu speichern. Für jede DA-Komponente ist eine implizite äußere Operation definiert, in Abhängigkeit von der Komponentenart wird diese als „führe_aus“, „initialisiere“ oder „starte“ bezeichnet. Desweiteren ist für jede DA-Komponente eine explizite innere Operation definiert, die als kanonische Operation bezeichnet wird. Neben dieser impliziten äußeren Operation können für DA-Komponenten weitere explizite äußere Operationen festgelegt werden.

Eine DA-Komponente befindet sich zu jedem Zeitpunkt ihrer Existenz in einem der folgenden Zustände: Vorbereitet (V), in Ausführung (A), Rechnend (R), Wartend (W) oder Terminiert (T).

Eine DA-Komponente setzt sich aus benannten Komponenten, d. h. Komponenten, welche über einen Namen angesprochen werden, anonymen Komponenten, d. h. Komponenten, welche über Zeiger angesprochen werden, sowie dem Anweisungsteil der kanonischen Operation zusammen. Die in einer Komponente enthaltenen benannten und anonymen Komponenten werden im Deklarationsteil der Komponente definiert.

Mit der Ausführung der impliziten äußeren Operation einer Komponente werden im Wesentlichen die folgenden Schritte vollzogen:

a) ein Übergang von außen nach innen;

b) die Ausführung der expliziten inneren Operation, die aus der Erarbeitung des Deklarationsteils und der Ausführung des Anweisungsteils besteht;

c) nach Beendigung der kanonischen Operation ein Übergang von innen nach außen.

Die DA-Komponenten können weiter untergliedert werden in aktive und passive Komponenten. Als passive Komponenten werden dabei Ordern und Depots betrachtet, aktive Komponenten werden durch die Akteure gebildet. Die im Deklarationsteil einer Komponente $a$ deklarierten Komponenten werden als lokale Komponenten von $a$ bezeichnet.

Die in den MoDiS Konzepten beschriebenen und in der Sprache INSEL realisierten Komponentenarten sind im Wesentlichen also die Folgenden:

1. **Akteure**
   Die Akteure bilden die aktiven Komponenten des Systems und besitzen somit neben der Speicherfähigkeit auch Rechenfähigkeit. Ein Akteur wird durch die implizite äußere Operation „starte“ erzeugt und führt so dann seine kanonische Operation unabhängig von seinem Erzeuger aus. Die Akteure lassen sich weiter unterscheiden in **m**ono-operationale Akteure (M-Akteure) und Akteure mit **K**ommunikationsfähigkeiten (K-Akteure). M-Akteure be-

sitzen nur eine äußere Operation, die kanonische Operation, und führen ihre Operation ohne Beeinflussung aus.

K-Akteure dagegen definieren weitere explizite äußere Operationen, die so genannten Kommunikationsoperationen. Diese können von außen aufgerufen werden und ermöglichen eine Kommunikation nach dem Operationen-orientierten Rendezvous-Konzept. Die Kommunikationsoperationen werden durch den anbietenden K-Akteur mittels spezieller Anweisungen explizit angenommen und dann sequentiell als K-Order in die kanonische Operation des K-Akteurs eingebettet. Dieser Vorgang wird als „akzeptieren“ bezeichnet. Dabei ermöglichen die akzeptierenden Anweisungen die Annahme einer K-Order aus einer Teilmenge der äußeren Operationen. Bei diesem Vorgang wird der K-Akteur als Auftragnehmer durch die akzeptierende Anweisung solange blockiert, bis ein entsprechender K-Order-Aufruf eines Auftragnehmers vorliegt. Falls ein Auftragnehmer einen K-Order-Aufruf absetzt, bevor der entsprechende K-Akteur seine akzeptierende Anweisung ausführt, so wird dieser ebenfalls blockiert. Diese Situation ist in Abb. 10.9 dargestellt.

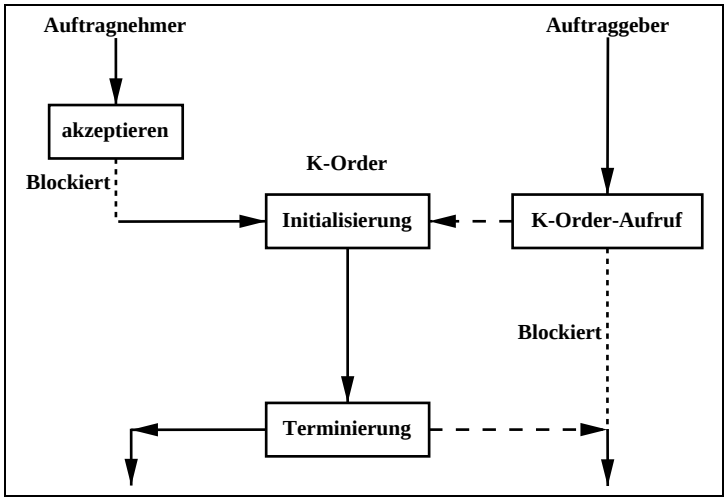



Abbildung 10.9: Das Operationen-orientierte Rendezvous.

**2. Ordern**

Die Ordern definieren Operationen und sind mit Inkarnationen und Blöcken in imperativen Programmiersprachen vergleichbar. Die implizite äußere Operation einer Order ist „führe_aus“. Die Definition von weiteren äußeren Operationen ist für Ordern nicht zulässig.

Die Ordern werden in S-Ordern und K-Ordern unterschieden. Die S-Ordern untergliedern sich in FS-, PS- und BS-Ordern, welche Funktions-, Prozedur- und Block-Inkanationen imperativer Sprachen entsprechen. Die K-Ordern

dagegen sind Kommunikationsordern, welche als äußere Operationen von K-Akteuren definiert werden.

3. **Depots**
   Die Depots stellen Speicher-definierende Komponenten dar. Ein Depot definiert lokale, benannte Komponenten, welche Speicher und Zugriffsoperationen auf diesen bereitstellen. Ein Depot ist eine passive Komponente, die von anderen Komponenten genutzt werden kann. Die implizite äußere Operation der Depots ist „initialisiere“; weitere äußere Operationen zum Zugriff auf ein Depot müssen explizit definiert werden.

4. **Generatoren**
   Die Generatoren sind Komponenten, welche Klassen von Komponenten definieren. Die äußere implizite Operation der Generatoren ist „erzeuge“, durch die eine Komponente der entsprechenden Klasse erzeugt wird. Die erzeugte Komponente kann eine DE-Komponente, eine DA-Komponente und auch wieder ein Generator sein. Letzteres wird als Generator zweiter Ordnung bezeichnet und entspricht dem Template-Konzept, welches beispielsweise aus der Sprache C++ bekannt ist (vgl. [Str97]). Damit wird eine weitere Abstraktionsstufe in der Beschreibung von Komponenten ermöglicht. Die Generatoren zweiter Ordnung sind nur für Depots und K-Akteure zulässig.

   Über die Generatoren wird die Information über die Klassen der Komponenten in das ausgeführte System eingebracht und nutzbar gemacht.

In Abb. 10.10 auf der nächsten Seite sind die Komponentenarten von MoDiS mit ihren definitorischen Beziehungen graphisch dargestellt.

### 10.2.2 Systemstrukturen in MoDiS

Die Strukturierung beschränkt sich in MoDiS nicht allein auf die Strukturierung der Problembeschreibung durch die Schachtelung. Vielmehr sind die Komponenten zu ihrer Laufzeit in Strukturen organisiert. Diese Strukturen sind der zentrale Ansatzpunkt für das Management von MoDiS und werden im Folgenden betrachtet.

Die Einordnung der Komponenten in die Strukturen ist dabei von den Zuständen der Komponenten abhängig. Der Übergang zwischen den Zuständen wird durch Ereignisse modelliert, deren Reihenfolge strikt festgelegt ist:

1. **Erzeugung** $e$
   Mit diesem Ereignis tritt die Komponente in den Zustand „Vorbereitet“ ein.

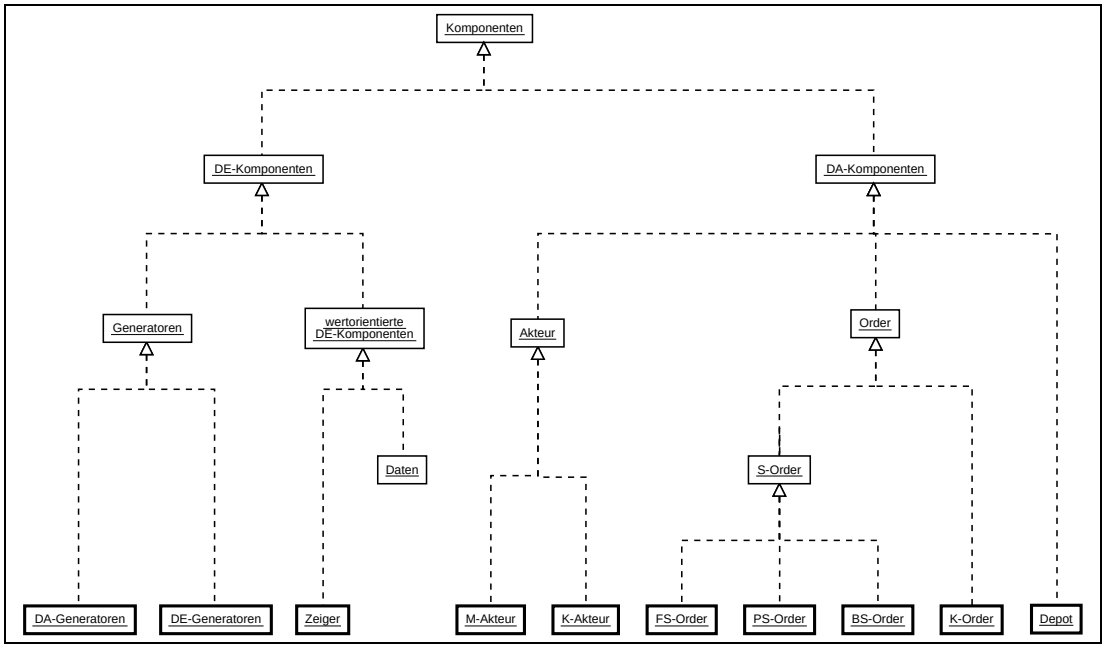



Abbildung 10.10: Komponenten in MoDiS.

2. **Initiierung** $i$
   Dieses Ereignis beschreibt den Übergang von „Vorbereitet“ in den Zustand „Ausführung“, der die Zustände „Rechnend“ und „Wartend“ umfasst.

3. **Terminierung** $t$
   Wenn die Komponente ihre kanonische Operation beendet so betritt sie den Zustand „Terminiert“.

4. **Auflösung** $a$
   Mit diesem Ereignis wird die Komponente aufgelöst.

Die Systemstrukturen in MoDiS sind durch die im Folgenden beschriebenen Relationen festgelegt:

### Definitionsstruktur $\tilde{\delta}$

Eine DA-Komponente, die einen Generator enthält, bestimmt die Menge der Komponenten, welche von den Inkanationen dieses Generators genutzt werden können. Dieser Zusammenhang wird durch die Strukturrelation $\tilde{\delta}$ beschrieben.

Eine Komponente $a$ ist unmittelbar-$\delta$-innen zu einer Komponente $b$ – $(a, b) \in \tilde{\delta}$ – genau dann, wenn der Generator, dessen Inkanation $a$, ist eine lokale Komponente von $b$ ist.

Durch die Einordnung in die Defintionsstruktur wird festgelegt, welche Komponenten des Systems aufgrund der Komponentenschachtelung von $a$ genutzt werden können. Die Ausführungsumgebung ergibt sich aus den lokalen Komponenten von $a$ sowie der Ausführungsumgebung der Komponente $b$, in die $a$ geschachtelt wurde. Die Definitionsstruktur ordnet jede Komponente bei ihrem *statischen Vorgänger* ein.

#### Lokalitätsstruktur $\tilde{\lambda}$

Die Lokalitätsstruktur $\tilde{\lambda}$ ordnet benannte DA-Komponenten bei ihrem Erzeuger ein, d. h. der Komponente, welche die Deklaration enthält. Durch die Einordnung $(a, b) \in \tilde{\lambda}$ in die Lokaliätsstruktur wird festgelegt, dass $a$ über seinen Namen lokal innerhalb $b$ nutzbar ist. Da Depots erst dann nutzbar sind, wenn sie terminiert sind, erfolgt die Einordnung von Depots in die $\tilde{\lambda}$ Struktur mit der Terminierung.

#### Zeigerstruktur $\tilde{\zeta}$

Die Zeigerstruktur $\tilde{\zeta}$ ordnet einer anonymen DA-Komponente die Komponenten zu, welche einen Zeiger auf die anonyme Komponente haben. Durch die Einordnung in die Zeigerstruktur wird festgelegt, welche Komponenten des Systems Zeigervariablen enthalten und über diese zugreifen können. Das Management muss einerseits den Zugriff auf nutzbare Komponenten und anderseits den Zugriffsschutz für unbenutzbare Komponenten realisieren.

#### Zeigerstruktur $\gamma$

Über die Zeigerstruktur $\gamma$ wird jeder anonymen DA-Komponente, die über einen Zeiger identifizierbar ist, diejenige DA-Komponente zugeordnet, die den entsprechenden Zeigergenerator für diesen Zeiger enthält.

#### Sequentielle Ausführungsstruktur $\tilde{\sigma}$

Mit der sequentiellen Ausführungsstruktur $\tilde{\sigma}$ wird die sequentielle Einordnung von DA-Komponenten beschrieben. Eine Order $a$ im Zustand A ist unmittelbar-$\sigma$-innen zu einer Komponente $b$ – $(a, b) \in \tilde{\sigma}$ –, genau dann, wenn $a$ eine S-Order ist, die bei der Ausführung der kanonischen Operation von $b$ erzeugt wurde oder wenn $a$ eine K-Order des K-Akteurs $b$ ist, die bei der Ausführung der kanonischen Operation von $b$ erzeugt wurde.

#### Parallele Ausführungsstruktur $\tilde{\pi}$

Die parallelen Kontrollflüsse von Akteuren werden durch die parallele Ausführungsstruktur $\tilde{\pi}$ erfasst. Ein M-Akteur $m$ ist unmittelbar-$\pi$-innen zu einer Komponente $b$ – $(m, b) \in \tilde{\pi}$ –, genau dann, wenn $m$ bei der Ausführung der kanonischen Opera-

tion von $b$ erzeugt wurde und die kanonische Operation von $m$ parallel ausgeführt wird.

Die K-Akteure werden, im Gegensatz zu den M-Akteuren, nicht immer $\pi$-innen in die erzeugende Komponente eingeordnet. In der Definition der Einordnung von K-Akteuren wird die Abbildung $\eta(x)$ verwendet, die jeder Komponente $x$ entweder einen Akteur oder eine Order zuordnet:

- Falls $x$ ein Akteur oder eine Order ist, so gilt: $\eta(x) = x$.
- Falls $x$ ein benanntes Depot im Zustand A ist: $\eta(x) = \eta(y)$, wobei $y$ die DA-Komponente ist, für die gilt $(x, y) \in \tilde{\sigma}$.
- Falls $x$ ein benanntes Depot im Zustand A ist: $\eta(x) = \eta(y)$, wobei $y$ die DA-Komponente ist, die $x$ als lokale Komponente enthält.
- Falls $x$ ein anonymes Depot ist: $\eta(x) = \eta(\gamma(x))$.

Damit ist ein benannter K-Akteur $k$ unmittelbar-$\pi$-innen zu $b$ – $(k, b) \in \tilde{\pi}$ – , genau dann wenn $b = \eta(x)$, wobei $x$ die DA-Komponente ist, deren Deklarationsteil die Deklaration von $k$ enthält. Sei $k$ ein anonymer K-Akteur, so ist $k$ unmittelbar-$\pi$-innen zu $b$, genau dann wenn $b = \eta(\gamma(k))$.

#### Kommunikative Ausführungsstruktur $\tilde{\kappa}$

Die kommunikative Ausführungsstruktur $\tilde{\kappa}$ ordnet die K-Ordern bei ihrem Aufrufer ein. Eine K-Order $c$ ist unmittelbar-$\kappa$-innen zu x – $(c, x) \in \tilde{\kappa}$ –, genau dann, wenn ein Akteur $a$ existiert, der Auftraggeber für $c$ ist und der $x$ ausführt.

Die Relationen $\tilde{\kappa}$ und $\tilde{\sigma}$ beschreiben somit für eine K-Order die Auftraggeber- und Auftragnehmer-Beziehung.

#### Ausführungsstruktur $\tilde{\alpha}$

Die Ausfühungsstruktur $\tilde{\alpha}$ beschreibt die Abhängigkeiten zwischen parallelen Ausführungsfäden, sowie die Einbettung sequentieller Operationen in diese. Die Relation $\tilde{\alpha}$ kann als Zusammenfassung der Relationen $\tilde{\sigma}$, $\tilde{\pi}$ und $\tilde{\kappa}$ betrachtet werden. Durch die Einordung von $(a, b) \in \tilde{\alpha}$ werden drei Festlegungen getroffen:

a) Die Ausführung der Komponente $a$ startet nach dem Beginn der Ausführung der Komponente $b$.

b) Die Komponente $b$ setzt ihre Ausführung frühestens nach dem Ereignis $\mathtt{i}_a$ fort.

c) Die Ausführung der Komponente $a$ endet vor dem Ende der Ausführung der Komponente $b$.

Daher muss die Komponente $b$ auf die Terminierung der Komponente $a$ warten, bevor sie selbst terminieren kann. Dieser Vorgang wird als *Abschlußsynchronisation* bezeichnet. Somit weiß der Erzeuger, dass nach dem Erzeugungsaufruf die erzeugte Komponente ihr Initiierungsereignis `i` durchlaufen hat. Des Weiteren ist sichergestellt, dass bei der Terminierung des Erzeugers auch die erzeugten Komponenten ihre Berechnungen abgeschlossen haben und ihre Ergebnisse vorliegen.

#### Lebenszeitstruktur $\tilde{\epsilon}$

Die Lebenszeitstruktur $\tilde{\epsilon}$ stellt sicher, dass eine bekannte Komponente, d. h. Komponenten deren Name bekannt ist oder die über einen Zeiger ansprechbar ist, auch tatsächlich existiert. Die Lebenszeitstruktur $\tilde{\epsilon}$ setzt sich aus den Strukturen $\sigma$, $\pi$, $\kappa$, $\lambda$ und $\gamma$ zusammen. Durch die Einordung von $(a, b) \in \tilde{\epsilon}$ werden folgende Festlegungen getroffen:

a) Die Erzeugung der Komponente $a$ findet nach der Erzeugung der Komponente $b$ statt.

b) Die Auflösung der Komponente $a$ findet vor der Auflösung der Komponente $b$ statt.

Daher muss das Management vor der Auflösung der Komponente $b$ sicherstellen, dass die Komponente $a$ aufgelöst wird.

Die Zusammenhänge zwischen den Strukturen $\tilde{\alpha}$ bzw. $\tilde{\epsilon}$ und den weiteren Strukturen ist in folgender Tabelle dargestellt, wobei $\eta$(x) der erste (reflexive) $\epsilon$-Vorfahre von x ist, der kein Depot ist.

| | | Wird eingeordnet bei | |
|---|---|---|---|
| Komponente | | $\alpha$ | $\epsilon$ |
| S-Order | unbenannt | Erzeuger [$\sigma$] | Erzeuger [$\sigma$] |
| K-Order | unbenannt | Erzeuger [$\kappa$] und bei Generator-Anbieter [$\sigma$] | Erzeuger [$\kappa$] |
| Depot | benannt | Erzeuger [$\sigma$] | Erzeuger [$\sigma$ bzw. $\lambda$] |
| | anonym | Erzeuger [$\sigma$] | Zeigergenerator-Anbieter [$\gamma$] |
| M-Akteur | unbenannt | Erzeuger [$\pi$] | Erzeuger [$\pi$] |
| K-Akteur | benannt | $\eta$(Erzeuger) [$\pi$] | Erzeuger [$\lambda$] |
| | anonym | $\eta$(Zeigergenerator-Anbieter) [$\pi$] | Zeigergenerator-Anbieter [$\gamma$] |

### 10.2.3 Spezifikation des Managements

Die Strukturen des vorherigen Abschnitts bilden die Grundlage für das Management eines Systems. Die Aufgaben des Managements sind dabei im Wesentlichen:

a) Entscheidung über die sequentielle, nebenläufige oder verteilte Ausführung von Akteuren;

b) Durchsetzung der Terminierungs- und Auflösungsbeziehungen;

c) Realisierung des Operationen-orientierten Rendezvous-Konzeptes für die Kommunikation.

Für den ersten Punkt kann dabei auf die Überlegungen im Abschnitt 10.1.2 auf Seite 186 zurückgegriffen werden. Die Diskussion der Spezifikation des Managements wird sich hier daher insbesondere mit den beiden letzten Punkten beschäftigen.

Die Hierarchie der Metatypen der Spezifikation von MoDiS ergibt sich aus den in Abb. 10.10 auf Seite 197 dargestellten Zusammenhängen der Komponentenarten. Für die Spezifikation des Managements auf dieser Grundlage ist zunächst die Spezifikation der Strukturrelationen notwendig. Die in MoDiS definierten Strukturen genügen, mit Ausnahme der Beziehung unmittelbar-$\alpha$-innen, alle der Forderung, baumartig zu sein. Die Ausführungsstruktur $\tilde{\alpha}$ verletzt diese Forderung durch die Einordnung von K-Ordern sowohl beim Auftraggeber als auch beim Auftragnehmer. In der Spezifikation wird daher die Struktur $\tilde{\alpha}$ ohne den Anteil der Struktur $\tilde{\kappa}$ betrachtet. Die kommunikative Ausführungsstruktur unmittelbar-$\kappa$-innen wird separat behandelt. Somit sind alle Strukturen eines Systems baumartig. Ein Ausschnitt aus der Spezifikation ist in Abb. 10.11 auf der nächsten Seite dargestellt. Die Zuordnung der Komponenten zur Definitionsstruktur $\delta$ wird über die statische Schachtelungstiefe der Komponenten berechnet. Diese Berechnung wird durch die Funktion *pstDiff* durchgeführt. Die Funktionen *walkBackDelta* und *walkBackDeltaLambda* setzen die korrekte Zuordnung der Komponenten in der Definitionsstruktur durch.

Die Terminierungs- und Auflösungsbedingungen für die Komponenten werden als Ereignisbedingungen der entsprechenden Ereignisse spezifiziert. Die Durchsetzung dieser Bedingungen durch das Management ist somit gewährleistet.

Damit verbleibt dem Management die Aufgabe der Realisierung des Operationen-orientierten Rendezvous-Konzeptes. Dafür werden die potenziellen Ereignisfolgen der beteiligten Komponenten, nämlich K-Akteur und die, die K-Order aufrufenden, DA-Komponente verfeinert.

Der relevante Ausschnitt der potenziellen Ereignisfolge für K-Akteure wird durch den regulären Ausdruck

```
META Comp
  EVT init (caller : DA)
  EVT run
  EVT terminate [((numsons(α_S)=0)∧(numsons(κ_S)=0))]
  EVT destroy [(numsons(ε_S)=0)]
  SEQ lifetime :
    init run terminate destroy
        ⋮
  REL ε → comp := ⊥
    init :        caller
    destroy :     ⊥
END Comp
META DA ← Comp
        ⋮
  REL α → DA := ε
    terminate :  ⊥
  REL δ → DA := ⊥
    init :        IF ((EnclDepot≠⊥)∧(EnclDepot≠caller⊢EnclDepot)) :
                    walkDeltaLambda(caller, EnclDepot)
                  ELSE
                    IF (pstDiff(caller, self)>0) :
                      walkBackDelta(caller, pstDiff(caller, self))
                    ELSE
                      IF (pstDiff(caller, self)<0) : caller
                      ELSE caller⊢δ
      ⋮
END DA
```

Abbildung 10.11: Ausschnitt der MoDiS-Spezifikation.

$$called * \ accept\ done$$

festgelegt. Das Ereignis *called* zeigt den Aufruf einer K-Order an und wird mit der Komponentenklasse der aufgerufenen K-Order parametrisiert. Wird die K-Order durch den K-Akteur akzeptiert, so wird ein *accept*-Ereignis mit der entsprechenden Klasse von K-Ordern erzeugt. Mit dem Ereignis *accept* wird die K-Order erzeugt und sequentiell in den K-Akteur eingeordnet. Das Ende der Ausführung der K-Order wird dem Akteur durch das Ereignis *done* mitgeteilt.

Für die DA-Komponente, welche die K-Order aufruft, wird der relevante Ausschnitt der potenziellen Ereignisfolge durch den regulären Ausdruck

$$call\ return$$

beschrieben. Der Aufruf von *call* ist gleichbedeutend mit dem Aufruf des Ereignisses *called* des K-Akteurs, das Ereignis *return* entspricht dem *done* Ereignis des K-Akteurs. Der Ablauf eines K-Order Aufrufs ist in Abb. 10.12 dargestellt.

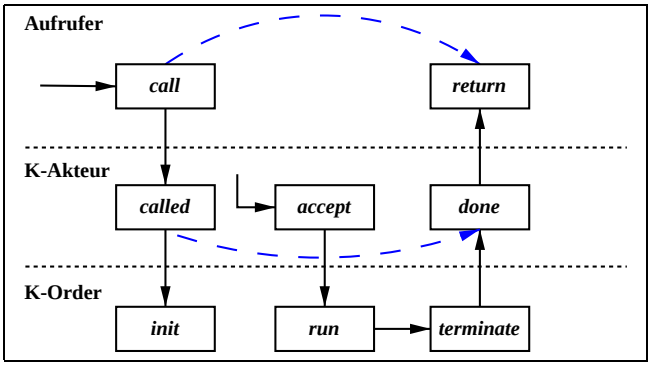



Abbildung 10.12: Ablauf des Operationen-orientierten Rendezvous.

Die Beobachtung des Systems, aber auch das Fällen von Entscheidungen durch das Management wird in MoDiS nicht allein auf die Komponenten und ihre Komponentenklassen abgestützt, sondern die Generatoren können als zwischen Komponenten und ihre Klassen geschaltete Komponenten betrachtet werden. Informationen werden zunächst durch die Komponenten gesammelt und dann bei den entsprechenden Generatoren gesammelt. Diese Information fließt dann von den Generatoren zu den entsprechenden Komponentenklassen. Bei der Erzeugung eines neuen Generators werden die Belegungen der Eigenschaften wieder von der Komponentenklasse in den Generator als Vorbelegung der Eigenschaften eingebracht. Diese Information wiederum steht dann für die Erzeugung der Komponenten zur Verfügung. Damit kann das Management eines Systems feingranularer Entscheidungen finden. Dieses Vorgehen ist in Abb. 10.13 auf der nächsten Seite dargestellt.

Für die Realisierung des Managements steht mit dem Übersetzer *gic* (**G**NU **I**NSEL **C**ompiler, vgl. [Piz97]) ein Werkzeug zur Verfügung, das Programme in der Sprache INSEL in Maschinencode transformiert. Der Übersetzer wurde so erweitert, dass neben der Übersetzung der INSEL-Programme in ausführbaren Code auch die entsprechenden Aufrufe in das generierte Management, d. h. die Erzeugung der Ereignisse, realisiert werden. Die zur Laufzeit gewonnenen Informationen über das System fließen über eine wohldefinierte Schnittstelle über das Dateisystem in den Übersetzer. Damit wird der Zusammenhang zwischen der statischen und der dynamischen Erarbeitung und Nutzung von Klasseneigenschaften simuliert. Durch diese Technik konnte darauf verzichtet werden, die Analyse von INSEL-Programmen und die notwendige Codeerzeugung neu zu erstellen.

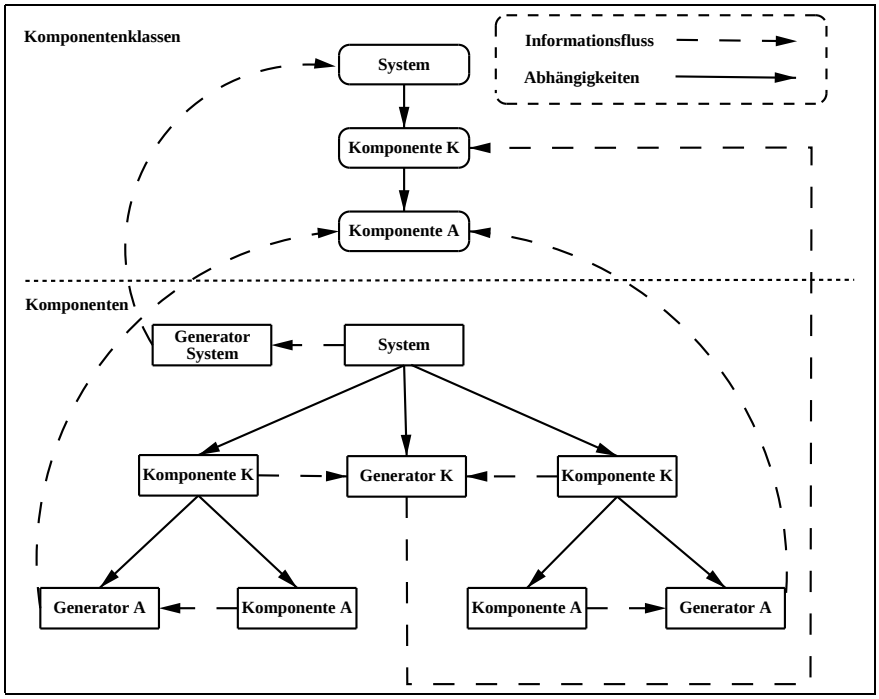



Abbildung 10.13: Informationsfluss zwischen Komponenten, Generatoren und Komponentenklassen.

### 10.2.4 Visualisierung des Systems

Neben der Spezifikation und Realisierung des Managements von Systemen auf der Grundlage der MoDiS-Konzepte ist auch die Visualisierung von Systemen von Interesse, um diese Systeme und ihr Verhalten analysieren zu können. Daher wurden für die Darstellung des Systems bzw. seiner Komponenten und deren Eigenschaften zwei Visualisierungswerkzeuge entwickelt. Beide Werkzeuge ermöglichen eine optische Aufbereitung eines laufenden MoDiS-Systems.

#### JaVis

Das Werkzeug JaVis kann die Struktur eines Schnappschusses des Systems zum augenblicklichen Zeitpunkt darstellen. Dabei werden die Komponenten als Knoten eines Graphen und Strukturrelationen als Kanten dieses Graphs dargestellt (siehe Abb. 10.14). Zusätzlich besteht die Möglichkeit, die Belegung der Eigenschaften einer Komponente im augenblicklichen Schnappschuß darzustellen.

Um die Entwicklung des Systems betrachten zu können, besteht die Möglichkeit, die Visualisierung zu einem beliebigen Zeitpunkt zu unterbrechen und dann in Einzelschritten fortzusetzen.

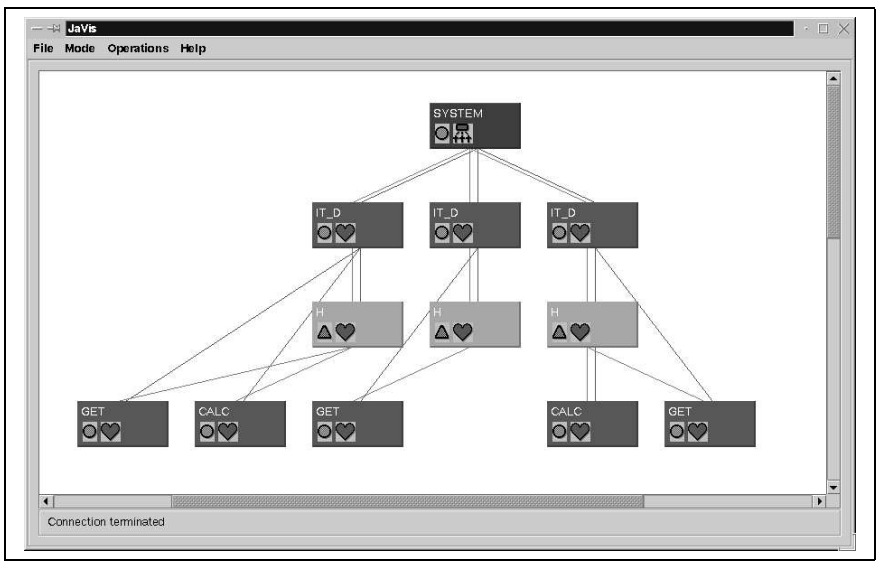



Abbildung 10.14: Visualisierung eines Systems mit JaVis

### DyVis

Das zweite Visualisierungswerkzeug DyVis stellt ein System in seinem zeitlichen Ablauf dar. Dabei werden die Ereignisse `i` und `t` jeder Komponente als Knoten eines Graphen dargestellt. Die Kanten des Graphen werden durch die $\alpha$ Relation des Systems gebildet (Siehe Abb. 10.15). Da aufgrund der zeitlichen Schachtelung des Systems eine Vorhersage der Terminierungsreihenfolge der Komponenten möglich ist, wird zu jedem Initiierungsknoten auch der korrespondierende Terminierungsknoten dargestellt, wobei Terminierungsereignisse, welche noch nicht eingetreten sind, durch andere Symbole dargestellt werden als bereits eingetretene Ereignisse. Damit wird eine Darstellung des Systems in der Zukunft möglich. Aus dieser Art der Darstellung ergeben sich die Ereignisse und ihre Zusammenhänge als mathematischer Verband.

Um die Übersichtlichkeit des dargestellten Ereignisverbandes zu erhöhen, ermöglicht das Werkzeug DyVis eine Vergröberung der Darstellung durch Faltung von Teilverbänden.

Analog zu JaVis besteht auch hier die Möglichkeit, die Eigenschaften der Komponenten und deren Belegung darzustellen und die Visualisierung zu unterbrechen.

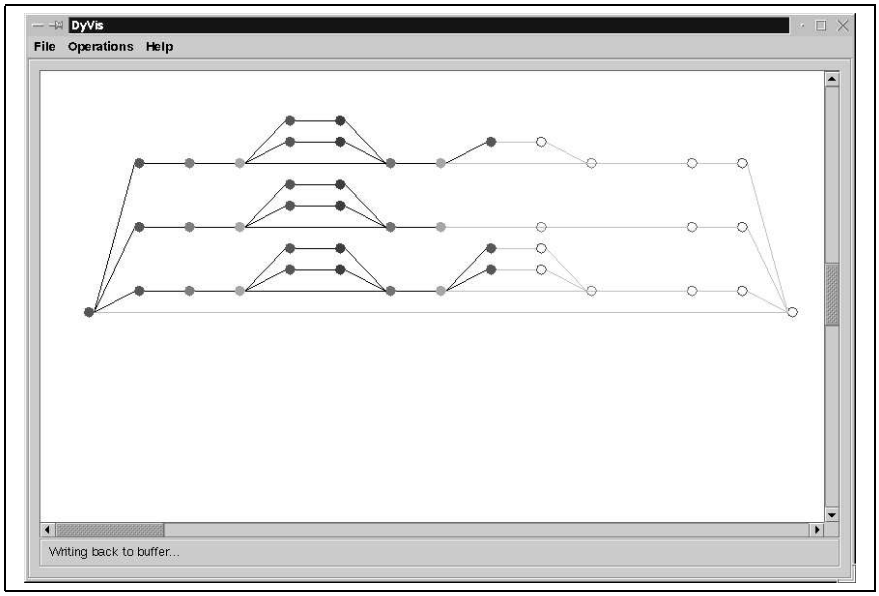



Abbildung 10.15: Visualisierung eines Systems mit DyVis

## 10.3 Fallbeispiel CDSL

Die Sprache CDSL (**C**oncurrent **D**istributed **S**ynchronized **L**anguage, vgl. [Hüt02] und [Spr02]) ist eine Programmiersprache für verteilte, nebenläufige Systeme, bei welcher besonderes Augenmerk auf die Synchronisation gelegt wird.

### 10.3.1 Ziele und Strukturen von CDSL

Die Konzepte der Sprache CDSL lassen sich in drei wesentlichen Aspekten zusammenfassen:

1. **Trennung von Rechen- und Speicherfähigkeit**
   Die Komponenten von CDSL werden in rechenfähige, speicherfähige und Operationen definierende Komponenten unterschieden. Erstere werden in CDSL als *Activities*, Zweitere als *Storages* und Letztere als Methoden bezeichnet.

2. **Objektbasiertheit**
   Die Komponenten in CDSL werden objektbasiert beschrieben. Die Komponenten sind somit gekapselt und interagieren über festgelegte Schnittstellen. Die Schnittstellen eines Storages stellen Zugriffsoperationen auf den gekapselten Speicher dar, die Schnittstellen eines Activity sind Kommunikationsoperationen.

3. **Referenzierung**
   Eine CDSL-Komponente kann Referenzen auf andere Komponenten enthalten. Über diese Referenzen ist zum Einen eine Zusammenarbeit der Komponenten möglich, zum Anderen werden über die Referenzen die Abhängigkeiten der Komponenten untereinander beschrieben.

Die aktiven Komponenten eines Systems realisieren die Nebenläufigkeit in CDSL. Das Konzept der Nebenläufigkeit bedingt die Notwendigkeit zur Synchronisation. In CDSL können prinzipiell drei Arten der Synchronisation unterschieden werden:

a) die Kommunikation zwischen Activities;

b) der Zugriff auf ein Storage über dessen Schnittstelle;

c) die Auflösungsabhängigkeiten, auch als Abschlusssynchronisation bezeichnet.

Für die Kommunikation zwischen den rechenfähigen Komponenten wird auch in CDSL das Operationen-orientierte Rendezvous-Konzept eingesetzt. Der Aufruf einer Kommunikationsoperation einer Activity $a$ durch eine andere aktive Komponente $b$ muss durch die Komponente $a$ akzeptiert werden.

Die Synchronisation des Zugriffs auf eine speicherfähige Komponente muss zunächst den wechselseitigen Ausschluss der lesenden und der schreibenden Operationen sicherstellen. Die Aufgabe des Managements ist es an diesem Punkt, die Zugriffe auf ein Storage atomar durchzuführen, bzw. die Atomarität der Zugriffe zu gewährleisten.

Das zentrale Konzept der Sprache von CDSL ist die Referenzierung von Komponenten. In CDSL wird jeder Zugriff auf eine andere Komponente über eine Referenz modelliert. Dabei wird zwischen starken und schwachen Referenzen unterschieden. Für jede Komponente existiert genau eine starke Referenz, welche die Auflösungsabhängigkeiten der Komponenten beschreibt. Die starke Referenz einer Komponente wird im Allgemeinen bei der Erzeugung der Komponente beim jeweiligen Erzeuger gesetzt. Eine weitere Komponentenart, welche bei der Definition der starken Referenzen und somit der Lebenszeit-Struktur eingeführt wird, sind die so genannten Blöcke. Ein Block markiert in diesem Sinn eine Zusammenfassung von Anweisungen, die auch Referenzen enthalten kann. Die Synchronisationsbedingungen gelten entsprechend auch für Blöcke. Eine derartige Struktur ist in Abb. 10.16 auf der nächsten Seite beispielhaft dargestellt.

Die Lebenszeit-Struktur dient der Ermittlung des Zeitpunktes der Auflösung einer Komponente. Ein Komponente kann aufgelöst werden, wenn sie ihren Beitrag zu den Berechnungen des Systems vollständig erbracht hat. Eine Aktivität kann somit aufgelöst werden, wenn sie erstens ihre Berechnungen abgeschlossen hat und zweitens die Ergebnisse der Berechnung an den Auftraggeber übergeben

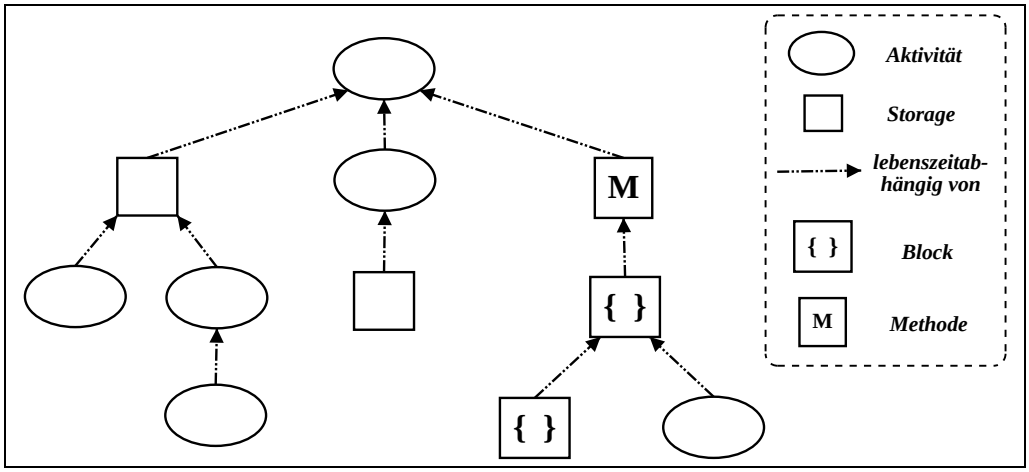



Abbildung 10.16: Lebenszeitabhängigkeiten in CDSL

wurden. Ein Storage dagegen kann aufgelöst werden, wenn die darin gekapselten Daten im weiteren Verlauf der Berechnungen nicht mehr benötigt werden.

Diese Überlegungen führen zu folgenden Festlegungen:

- Eine Aktivität $a$ kann aufgelöst werden, wenn alle Aktivitäten $b_1 \ldots b_n$ aufgelöst wurden, auf die $a$ eine starke Referenz besitzt und die Aktivität $a$ ihre Berechnungen abgeschlossen hat.
- Ein Storage $s$ kann aufgelöst werden, wenn die Komponente $k$, welche die starke Referenz auf $s$ hält, aufgelöst wird.

Ein weiterer zentraler Aspekt eines CDSL-Programms ist die Semantik der Parameterübergabe. In CDSL werden alle Parameter und Rückgabewerte gemäß der kopierenden Semantik übergeben. Somit ist sichergestellt, dass die Parameter bzw. die über die Parameter referenzierten Komponenten als starke Referenz übergeben werden. Die in anderen Sprachen übliche Vermischung von Semantiken bei der Parameterübergabe wird somit vermieden und die Auflösung der Objekte, welche durch Parameter angesprochen werden, wird in die generelle Politik zur Auflösung von Komponenten einbezogen.

### 10.3.2 Spezifikation von CDSL

Aus den Sprachkonzepten von CDSL und der entsprechenden abstrakten Syntax (vgl. [Spr02]) können für den Sprachanteil des Managements für CDSL-Programme die in Abb. 10.17 auf der nächsten Seite dargestellten Metatypen abgeleitet werden.

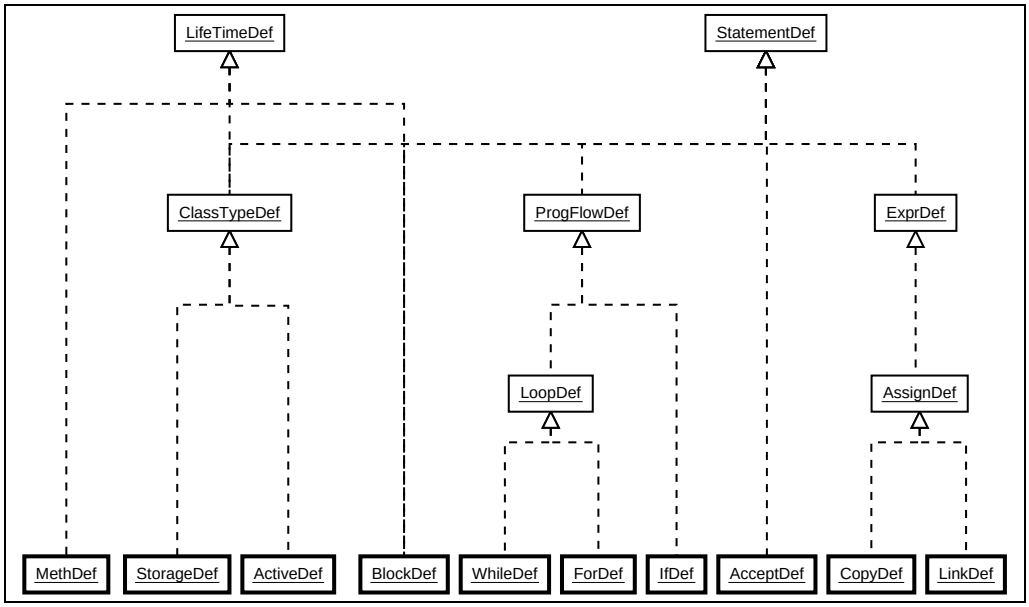



Abbildung 10.17: Metatypen der Spezifikation von CDSL

Der abstrakte Metatyp `LifeTimeDef` kapselt die Eigenschaften, Ereignisse und potenziellen Ereignisfolgen, die zur Durchsetzung der Lebenszeit-Struktur notwendig sind und ist gemäß den Konzepten von CDSL daher Obertyp zu den Metatypen für Activities, Storages, Methoden und Blöcken. Die Realisierung der Lebenszeiten ist analog zur Realisierung in MoDiS (vgl. Abschnitt 10.2.3), wobei die Einordnung in die Lebenszeit-Struktur mit der Entstehung einer Komponente durchgeführt wird.

Ebenso ist die Realisierung des Operationen-orientierten Rendezvous an die entsprechende Realisierung des Managements in MoDiS angelehnt und analog spezifiziert.

Die dritte zentrale Aufgabe des Managements, die Durchsetzung atomarer Zugriffe auf die speicherfähigen Komponenten mittels der Zugriffsoperationen, kann entsprechend der in Abschnitt 5.3.4 auf Seite 100 (Standardprobleme zur Synchronisation) beschriebenen Verfahren gelöst werden.

Im Rahmen der Spezifikation des Managements von CDSL wurde eine neue Strategie für die Verteilung der aktiven und passiven Komponenten über die verteilte Hardware-Konfiguration entwickelt. Die Grundlage dafür bildet zunächst eine statische Analyse der potenziellen Kommunikationsbeziehungen und -häufigkeiten der Komponenten. Das Ergebnis dieses Prozesses ist eine statische Einteilung der Komponenten in so genannte *Cluster*, welche Komponentenklassen enthalten, deren Instanzen zur Laufzeit mit einer hohen Wahrscheinlichkeit häufig mitein-

ander kommunizieren. Ein Cluster ist dabei ein Metatyp der Spezifikation und die Komponentenklassen der Activities bzw. Storages werden Instanzen dieses Metatypen zugeordnet.

Zur Laufzeit des Systems wird jedem Cluster eine Teilmenge der vorhandenen Knoten der verteilten Hardwarebasis zugewiesen. Das Cluster wählt nun für jede Instanz einen Knoten aus dieser Menge aus, auf welchem die Komponente platziert wird. Durch Monitoring der Kommunikationsbeziehungen wird das reale Kommunikationsverhalten der Komponenten ermittelt und in den Komponentenklassen verwaltet. Somit kann die initiale Zuordnung von Komponentenklassen zu Clustern verbessert werden, indem Komponentenklassen zwischen Clustern wandern. Diese Situation ist in Abb. 10.18 dargestellt.

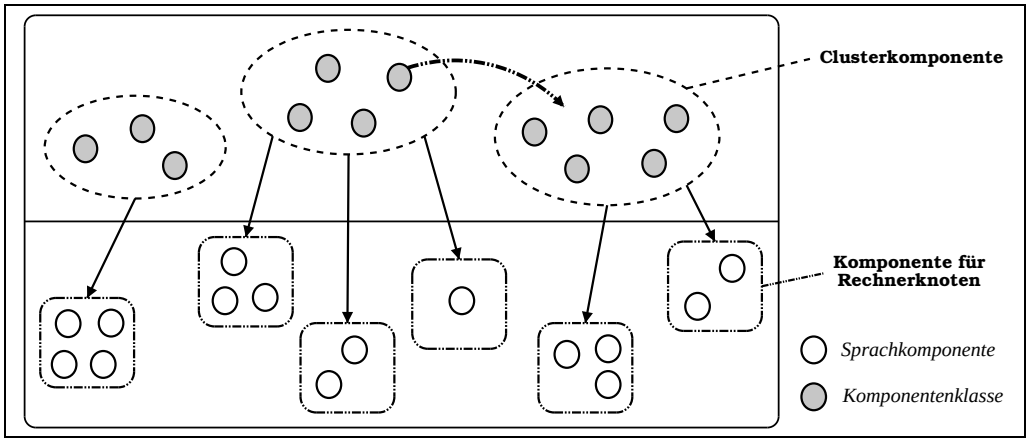



Abbildung 10.18: Zuordnung von Komponentenklassen zu Clustern in CDSL.

Damit diese Form der Migration nicht zu einem ständigen Flattern der Komponentenklassen zwischen den Clustern führt, muss eine entsprechende Hürde in die Entscheidung der Migration eingebaut werden. Daher wird die Entscheidung über die Cluster-Migration einer Komponente gemäß der folgenden Formel getroffen:

$$\kappa(C_{eigen}) \cdot \frac{\sigma(C_{activ})}{\sigma(C_X)} < \kappa(C_X) \cdot \alpha,$$

wobei

- $\kappa(C_{eigen})$ : Kommunikationsgrad dieser Aktivitätsklasse mit anderen Aktivitätsklassen und Storageklassen im eigenen Cluster.
- $\frac{\sigma(C_{activ})}{\sigma(C_X)}$ : Verhältnis von Synchronisation der Aktivitätsklasse mit Aktivitätsklassen im eigenen Cluster und mit Aktivitätsklassen im fremden Cluster X.
- $\kappa(C_X)$ : Kommunikationsgrad der Aktivitätsklasse mit Aktivitätsklassen und Storageklassen im fremden Cluster X.

- $\alpha$ : Gewichtung des Kommunikationsgrades mit fremden Clustern.

Diese Form der Verteilung von Komponenten ermöglicht einerseits die sukzessive Verbesserung der Platzierungsentscheidungen der Komponenten des Managements, ohne andererseits für jede Komponente die Platzierungsentscheidung über die gesamte verteilte Hardware-Konfiguration treffen zu müssen.

Darüber hinaus kann die zur Laufzeit gewonnene Information über das Kommunikationsverhalten der Komponenten wieder in die statische Analyse des Systems einfließen und somit eine verbesserte Transformation des Programm-Codes in Maschinensprache erreicht werden.

Die vollständige Spezifikation der Laufzeitumgebung für die Sprache CDSL, wie sie in [Ben03] entworfen wurde, ist im Anhang D dargestellt.

## 10.4 Zusammenfassung

Die in diesem Kapitel präsentierten Beispiele für die Spezifikation des Managements verteilter Systeme demonstrieren zum Einen die Machbarkeit des Ansatzes der abstrakten Beschreibung des Managements und dessen Generierung. Der Ansatz ist sowohl für Basismechanismen, wie das Scheduling oder die Speicherverwaltung, als auch für das Management komplexer Systeme einsetzbar.

Zum Anderen zeigen die Beispiele, dass durch geeignete Strukturierungen und die Konzentration auf die Konzepte des Managements einfache, verständliche und somit handhabbare Realisierungen ermöglicht werden.

Insbesondere mit der Spezifikation des Managements von CDSL konnte demonstriert werden, dass die Verwendung von wiederkehrenden Standardaufgaben des Managements durch geeignete Kapselung der entsprechenden Ereignisse, Eigenschaften und Aktionen in abstrakten Metatypen, sowie deren Verwendung durch die Hierarchie auf den Metatypen einerseits möglich ist und andererseits zu einer Vereinfachung der Spezifikation des Managements führt.

# 11 Performanz in generierten Systemen

In diesem Kapitel wird die Performanz der generierten Systeme diskutiert. Bei diesen Betrachtungen sind folgende Erkenntnisse von zentraler Bedeutung:

1. **Overhead**
   Die notwendige Kommunikation in einem verteilten, kooperativen System bewirkt stets einen Overhead. Dabei ist die Leistungsfähigkeit des Netzwerks der begrenzende Faktor. Eine Reduzierung der Kommunikation erhöht daher die Leistung eines Systems.

2. **Zusammenhang zwischen Speicher und Rechenzeit**
   Bei der Optimierung eines Systems muss zwischen der Optimierung im Bezug auf den Speicher und der Optimierung im Bezug auf die Rechenzeit unterschieden werden, wobei diese beiden Aspekte gegenläufige Kräfte darstellen. Eine Steigerung der Rechengeschwindigkeit muss mit einem erhöhten Speicherbedarf erkauft werden und umgekehrt wird eine Verringerung des Speicherbedarfs durch einen höheren Rechenaufwand bezahlt.

Ein weiterer zentraler Punkt in diesem Zusammenhang ist die Feststellung, dass Sicherheit, sowohl im Sinne von funktionaler Sicherheit, als auch im Sinne des Datenschutzes mit zusätzlichem Aufwand verbunden ist. Dabei ist die Frage der Abwägung, inwiefern ein Verlust an Rechengeschwindigkeit zugunsten erhöhter funktionaler Sicherheit und Datensicherheit in Kauf genommen wird.

## 11.1 Auswirkungen der Systembeobachtung

*„Jede Systembeobachtung verändert das beobachtete System.“*

Diese fundamentale Erkenntnis aus Physik und Systemtheorie kann auch auf die in der Informatik betrachteten Systeme übertragen werden.

Die Beeinflussung durch die Beobachtung eines Systems in Black-Box-Sicht ist dabei noch relativ gering, da nur die Ein- und Ausgaben des Systems beobachtet

werden. Allerdings kann bereits die Aufbereitung dieser Daten in eine lesbare Form, sowie die Protokollierung der Daten, Einfluss auf das Verhalten der Systems haben.

Wird die Systembeobachtung auf die Glass-Box-Sicht eines Systems erweitert, so wird die Beeinflussung des Systems durch Beobachtung verstärkt. Mit dem Grad der Detaillierung der Informationserfassung steigt der Aufwand für das Monitoring. Allerdings bildet das gewonnene Wissen die Grundlage für Managemententscheidungen. An dieser Stelle muss daher eine Abwägung zwischen einer breiten Basis an Information für das Management und dem Aufwand für die Gewinnung dieser Information getroffen werden. Diese Abwägung kann dynamisiert werden, da das Wissen über ein System mit dessen Lebensdauer steigt; somit verringert sich der Zuwachs an Wissen mit der Zeit. Ab einem gewissen Punkt ist das Wissen über das System oder Teile des Systems ausreichend für die Entscheidungen des Managements, so dass der Aufwand für die Systembeobachtung reduziert werden kann. Die Festlegung dieses Punktes kann in die Berechnungsfunktionen der Eigenschaften integriert werden.

Die Gewinnung von Information ist also mit Kosten in Bezug auf die zur Verfügung stehende Rechenleistung der Hardware verbunden. Durch das Management wird also ein zusätzlicher Overhead durch die Gewinnung von Informationen und deren Verwaltung erzeugt. In der folgenden Tabelle sind Messwerte für die Erzeugung von Aktivitäten, sowie deren Abschlusssynchronisation aus dem Fallbeispiel MoDiS (vgl. Abschnitts 10.2 auf Seite 191) dargestellt. Dem gegenüber werden die Messwerte für die Lösung dieser Aufgabe durch die `pthread`-Bibliothek gestellt. Dabei ist zu beachten, dass das generierte Management wesentliche Monitoring-Aufgaben realisiert.

| Anzahl der Aktivitäten | generiertes Management | `pthread`-Bibliothek |
|---:|---:|---:|
| 3 | 7 ms | 1 ms |
| 39 | 107 ms | 6 ms |
| 155 | 451 ms | 22 ms |
| 399 | 1366 ms | 73 ms |
| 1110 | 6009 ms | 560 ms |
| Durchschnitt | 4,6 ms | 0,4 ms |

Die Messwerte stellen den Overhead dar, der sich aus dem Einsatz des generierten Managements ergibt. Da die prototypische Realisierung des generierten Managements auf der `pthread`-Bibliothek aufbaut, ergibt sich der Overhead des Managements aus der Differenz der Messwerte.

Eine weitere Quelle für den erhöhten Aufwand stellt die Netzwerkkommunikation dar. In der folgenden Tabelle sind die Messwerte für eine verteilte Ausführung des obigen Beispiels dargestellt.

| Anzahl der Aktivitäten | 2 Knoten | 3 Knoten | 4 Knoten |
|---|---|---|---|
| 3 | 132 ms | 99 ms | 87 ms |
| 39 | 364 ms | 232 ms | 348 ms |
| 155 | 898 ms | 820 ms | 672 ms |
| 399 | 2633 ms | 1454 ms | 1159 ms |
| 1110 | 9654 ms | 4087 ms | 3327 ms |
| Durchschnitt | 8,0 ms | 3,9 ms | 3,3 ms |

## 11.2 Strategien zur Steigerung der Performanz in generierten Systemen

Die grundlegende Idee zur Steigerung der Performanz in generierten Systemen ist die Eliminierung nicht benötigter Berechnungen des Managements bzw. die zeitliche Verlagerung aufwändiger Berechnungen. Für Berechnungen stehen drei unterschiedliche Zeiträume bzw. Zeitpunkte zur Verfügung.

1. **Generierungszeit**
   Zur Generierungszeit des Managements liegt die Spezifikation vollständig vor. Durch eine entsprechende Analyse der Berechnungsfunktionen können Berechnungen eliminiert bzw. vereinfacht werden.

2. **Startzeit des Systems**
   Mit dem Beginn der Ausführung eines Systems werden Teile des Managements konfiguriert und verbleiben über die gesamte Ausführungszeit des Systems invariant. Berechnungen auf dieser Basis können zu diesem Zeitpunkt durchgeführt werden und somit aus den dynamischen Analysen herausgezogen werden.

3. **Ausführungszeit des Systems**
   Teile der spezifizierten Berechnungen können zeitlich oder räumlich verschoben ausgeführt werden. In einem verteilten System ergeben sich für einzelne Knoten Zeiten, in welchen keine Berechnungen durchzuführen sind. Diese Zeiten können für die Berechnungen des Managements genutzt werden.

Die Grundlage für die Steigerung der Performanz generierter Systeme bildet die Datenflussanalyse, welche auf den Abhängigkeiten der Eigenschaften, also dem funktionalen Abhängigkeitsgraphen $\mathfrak{A}_{\mathcal{U}}$ basiert.

### 11.2.1 Invariante Berechnungen

Anhand des funktionalen Abhängigkeitsgraphen können Eigenschaften ermittelt werden, deren Belegung nicht veränderlich ist, d. h. die bereits mit der Spezifikation festgelegt sind. Formal ist eine invariante Eigenschaft folgendermaßen definiert

**Definition 11.1 (Invariante Eigenschaft)**
*Sei $M \in \mathcal{M}$ ein Metatyp. Eine Eigenschaft $E \in \mathcal{E}_M$ heißt invariante Eigenschaft, genau dann, wenn gilt*

$$\forall C \in \mathcal{C} : M \rightsquigarrow C \Rightarrow \forall t, t' \in T : \tilde{Z}(C, E, t) = \tilde{Z}(C, E, t)$$

*Die Menge aller invarianten Eigenschaften wird als $\mathcal{I}$ bezeichnet.*

Eine Eigenschaft ist genau dann invariant, wenn sie über die gesamte Lebenszeit der Komponente konstant belegt ist,d. h. die Belegung der Eigenschaft wird durch Ereignisse nicht verändert. Die konstante Belegung einer Eigenschaft erfolgt durch direkte Zuweisung eines konstanten Ausdrucks mit der Berechnungsfunktion des Startereignisses.

Ausgehend von den invarianten Eigenschaften können durch den funktionalen Abhängigkeitsgraphen alle Berechnungsfunktionen ermittelt werden, in deren Berechnung nur invariante Eigenschaften und konstante Ausdrücke eingehen. Diese Funktionen können statisch analysiert und das Ergebnis ihres definierenden Ausdrucks kann bereits zur Generierungszeit ermittelt werden. Diese Berechnungsfunktionen sind dann konstant. Ausgehend von diesen konstanten Funktionen können weitere Funktionen statisch ausgewertet werden, da jeder Berechnungsfunktion einer Eigenschaft eindeutig ein Ereignis zugeordnet ist und für jedes Ereignis die Menge der simultanen Ereignisse definiert ist. Eine Berechnungsfunktion $f_{E,\sigma}$ einer Eigenschaft $E$ bei Auftreten des Ereignisses $\sigma$ kann also statisch ausgewertet werden,

a) $\forall E \in \tilde{\mathcal{G}}_{f_{E,\sigma}} : E \in I$ oder

b) $\forall f_{E',\sigma'} : (f_{E,\sigma}, f_{E',\sigma'}) \in V$, $f_{E',\sigma'}$ ist statisch auswertbar, wobei $\mathfrak{A}_{\mathcal{U}} = \langle E, V \rangle$

Sind dagegen nur Teile einer Funktion konstant, so können die definierenden Ausdrücke der Berechnungsfunktionen vereinfacht werden. Auch diese Vereinfachungen erhöhen die Performanz des generierten Managements. Diese Optimierungen bedingen einen erhöhten Analyseaufwand zur Generierungszeit des Managements. Dies gilt auch für die im Folgenden beschriebene Abschwächung der Invarianz von

Eigenschaften. Eine semiinvariante Eigenschaft ist eine Eigenschaft, welche ab einem Zeitpunkt in der Lebenszeit der Komponente invariant ist.

**Definition 11.2 (Semiinvariante Eigenschaft)**
*Sei $M \in \mathcal{M}$ ein Metatyp. Eine Eigenschaft $E \in \mathcal{E}_M$ heißt semiinvariante Eigenschaft, genau dann, wenn gilt*

$$\forall C \in \mathcal{C} : M \rightsquigarrow C \Rightarrow \exists t \in T : \forall t' \geq t \in T : \tilde{Z}(C, E, t) = \tilde{Z}(C, E, t')$$

Falls eine semiinvariante Eigenschaft eine Klasseneigenschaft ist, so ist diese Eigenschaft für alle Komponenten einer Klasse ab dem Zeitpunkt $t$ unveränderlich, ab dem diese Eigenschaft für eine Komponente unveränderlich ist.

Semiinvariante Eigenschaften sind insbesondere dann von Interesse, wenn der Zeitpunkt, ab dem die Eigenschaft invariant ist, statisch ermittelt werden kann. Dieser Fall liegt vor, wenn der Übergang zur Invarianz einer Eigenschaft mit einem Ereignis verknüpft ist, d. h. alle Ereignisse $\sigma' \in \aleph_M(\sigma)$ die Belegung der Eigenschaft nicht verändern. Dieser Fall liegt insbesondere dann vor, wenn

a) $\forall \sigma' \in \aleph_M(\sigma) : \nexists f_{E,\sigma'} \in F_E$ und

b) alle Eigenschaften $E' \in \tilde{\mathcal{G}}_{f_{E,\sigma}}$ unveränderlich sind.

Ist eine Eigenschaft semiinvariant und kann dieses Merkmal statisch ermittelt werden, so sind Optimierungen des Managements ab diesem Zeitpunkt möglich. Von besonderem Interesse sind dabei die Klasseneigenschaften. Durch semiinvariante Klasseneigenschaften können Berechnungen für Managemententscheidungen bereits zu einem sehr frühen Zeitpunkt fixiert werden.

Klasseneigenschaften repräsentieren beispielsweise auch die Berechnungen, welche in klassischen Systemen durch den Übersetzer erfolgen, wie die Generierung von Code. Die Codeeigenschaft einer Komponente kann in vielen Fällen berechnet werden, sobald die Komponentenklasse durch das Problemlösungsverfahren vorliegt. Ausgehend von dieser Festlegung können weitere Eigenschaften der Komponenten fixiert werden.

Dies gilt auch für die Hardware-Ressourcen des Systems, falls diese nicht dynamisch sind. Die Konfiguration kann dann mit dem Start des Systems ermittelt werden und ist ab diesem Zeitpunkt invariant. Managemententscheidungen, welche von der vorhandenen Hardware abhängen, werden dann zu diesem Zeitpunkt fixiert.

Mit den invarianten und semiinvarianten Eigenschaften werden Berechnungen des Managements aus der Ausführungszeit in die Generierungszeit des Managements bzw. in die Startzeit des Systems verlagert, wodurch die notwendigen Berechnungen des Managements zur Laufzeit vereinfacht werden können.

### 11.2.2 Optimierung von Aktionen

Wird die Analyse der Spezifikation nicht nur auf die Eigenschaften beschränkt, sondern auf die Aktionen des Managements erweitert, so entsteht ein weiteres Optimierungspotenzial. Einerseits können, gemäß der Festlegungen im vorhergehenden Abschnitt, die Berechnungen der Aktionen vereinfacht werden. Diese Vereinfachungen verringern den Aufwand für die Überprüfung und Durchführung von Aktionen.

Andererseits können aber Aktionen weiter optimiert werden, wenn der Einfluss der Berechnungsfunktionen der Eigenschaften auf die Aktionen feingranularer analysiert wird. Für jede Eigenschaft $E$ kann eine Menge von Aktionen ermittelt werden, für welche gilt

$$A_i \in \mathcal{A}_U \wedge C_{A_i} \text{ enthält } E.$$

Gemäß der Festlegungen bezüglich der Codeerzeugung für eine Eigenschaft $E$ und Ereignis $\sigma$ in Abschnitt 8.2.4 auf Seite 163, fließt der Code für obige Aktionen in diese generierte Funktion ein. Sowohl der Ausdruck der Ereignisbedingung als auch der Ausdruck der Aktionsberechnung kann durch Einsetzen des definierenden Ausdrucks der Berechnungsfunktion für die entsprechende Eigenschaft umgeformt und anschließend vereinfacht werden. Somit können Teile der Aktionsausdrücke statisch berechnet werden.

Diese Vereinfachungen der Aktionsbedingungen können zu zwei extremen Situationen führen.

1. **Konstant wahre Aktionsbedingung**
   In diesem Fall kann die Überprüfung der Aktionsbedingung zur Laufzeit entfallen, der Code für die Aktion wird in jedem Fall ausgeführt.

2. **Konstant falsche Aktionsbedingung**
   In diesem Fall kann die Aktion zur Laufzeit nie ausgeführt werden, der Code für die Aktion kann aus der generierten Funktion eliminiert werden.

Diese Technik kann erweitert werden auf die Aktionen in den generierten Funktionen für das Auftreten eines Ereignisses. Dabei werden alle Eigenschaften, welche durch das entsprechende Ereignis eine veränderte gültige Berechnungsfunktion erhalten, in der Aktion durch den entsprechenden definierenden Ausdruck ersetzt, wobei diese Ersetzung auch die Anwendung einer dieser Eigenschaften in

den definierenden Ausdrücken beinhaltet, wobei zu beachten ist, dass dieses Vorgehen nur für zyklenfreie simultane Abhängigkeitsgraphen terminiert. Im Falle eines Zyklus im simultanen Abhängigkeitsgraphen muss der Ersetzungsvorgang abgebrochen werden.

**Beispiel 11.1**
In Beispiel 8.1 auf Seite 164 wurde ein Metatyp $U$ beschrieben, mit den Eigenschaften `a`, `b`, `c` und `d`, sowie den Berechnungsfunktionen

- $f_{\mathtt{a},\sigma}$: $\mathtt{a} = const$
- $f_{\mathtt{b},\sigma}$: $\mathtt{b} = \mathtt{a} + x$
- $f_{\mathtt{c},\sigma}$: $\mathtt{c} = \mathtt{a} + y$
- $f_{\mathtt{d},\sigma}$: $\mathtt{d} = \mathtt{b} - \mathtt{c}$

Für diesen nutzbaren Metatypen sei nun eine Aktion $A$ definiert mit $C_A \hat{=} d > a$. Dann kann $C_A$ zu $x - y > const$ umgeformt werden. Da $x$, $y$ und $const$ konstante Werte sind, kann in diesem Fall die Bedingung $C_A$ statisch ausgewertet werden.

Kann eine Aktionsbedingung statisch ausgewertet werden, so können die Berechnungsfunktionen $f_{E,\sigma}$ der Eigenschaften eliminiert werden, wenn diese Eigenschaft nicht in weitere Berechnungen eingeht, d. h. es keine Berechnungsfunktion $f_{E',\sigma'}$ gibt, für die gilt:

$$f_{E',\sigma'} \in \mathcal{G}_{f_{E,\sigma}}.$$

Eine weitere Möglichkeit zur Optimierung der Aktionen ergibt sich, wenn die Bedingungen zweier Aktionen genau entgegengesetzt spezifiziert wurden, also für $A_1, A_2 \in \mathcal{A}$ gilt, dass $C_{A_1} = \neg C_{A_2}$. In diesem Fall kann der Code für diese Aktionen zusammengefasst und zu

Stat($C_{A_1}$);
**IF** Var($C_{A_1}$) **THEN** Stat($f_{A_1}$) **ELSE** Stat($f_{A_2}$)

transformiert werden.

Durch die statische Analyse der Aktionen kann einerseits der Rechenaufwand für die Durchführung von Aktionen verringert werden. Andererseits kann diese Analyse auch Aufschlüsse über die Verwendung der Aktionen im Management liefern. So können beispielsweise nie ausgeführte Aktionen ermittelt werden, welche auf eine fehlerhafte Spezifikation hindeuten.

### 11.2.3 Verzögerte Auswertung

Die Eigenschaften bzw. deren Belegung bilden die Grundlage für die Entscheidungen des Managements, welche selbst wieder durch Eigenschaften repräsen-

tiert werden. Dabei fällt der Zeitpunkt der Erarbeitung von Wissen durch das Management und der Zeitpunkt der Anwendung des Wissens auseinander, im Extremfall wird das erarbeitete Wissen für die Entscheidungen des Managements bei bestimmten Abläufen des Systems nicht benötigt. In diesen Situationen kann die Erarbeitung des Wissens entweder zeitlich gestreckt oder vollständig aus den Berechnungen des Managements eliminiert werden. Dies wird im Folgenden als *verzögerte Auswertung* von Eigenschaften bezeichnet.

Für die verzögerte Auswertung ist zunächst die Frage zu klären, welche Eigenschaften verzögert ausgewertet werden können. Die Kriterien für eine verzögerte Auswertung sind die Folgenden:

a) Verzögert ausgewertete Eigenschaften dürfen keine Managemententscheidungen repräsentieren, welche der Durchsetzung bedürfen. Diese Eigenschaften sind genau diejenigen, welche in den bedingten Ausdruck einer Aktion eingehen.

b) Keine Nutzung von zeitkritischen externen Funktionen. Derartige Funktionen dienen der Erfassung von Informationen aus der Hardware, die für zukünftige Managemententscheidungen genutzt werden sollen. Ein typisches Beispiel ist die Ermittlung der Laufzeit einer Komponente durch Auslesen der Hardwareuhr.

Die verzögerte Auswertung von Eigenschaften muss spätestens bei dem Zugriff auf die entsprechende Eigenschaft durchgeführt werden. Wird während der Zeitspanne, in welcher eine Berechnungsfunktion für eine Eigenschaft gültig ist, diese Eigenschaft nie abgefragt, so kann die Berechnung dieser Funktion unterbleiben.

Für die Realisierung der verzögerten Auswertung ist eine weitere Unterstützung durch die in Abschnitt 8.1.3 auf Seite 151 beschriebene Wissensbasis erforderlich, da beim Zugriff auf eine verzögert ausgewertete Eigenschaft die Belegung zunächst durch die entsprechende Berechnungsfunktion ermittelt werden muss. Die Wissensbasis muss daher in der Lage sein, neben der Belegung von Eigenschaften auch eine Referenz auf die Berechnungsfunktion der verzögert ausgewerteten Eigenschaft zu speichern und diese bei Bedarf auszuführen.

Somit ergeben sich unter Umständen Ketten von verzögert ausgewerteten Eigenschaften, falls in die Berechnung einer solchen Funktion wiederum andere, noch nicht ausgewertete, Eigenschaften einfließen. Damit die verzögerte Auswertung von Eigenschaften zu einer Steigerung der Performanz des Systems führt, ist es notwendig, dass diese Ketten in ihrer Länge begrenzt bleiben. Dies kann durch Nutzung von Rechenzeiten, welche nicht für die Berechnungen des Systems benötigt werden, für die Auswertung solcher Eigenschaften erreicht werden.

Ein wesentliches Kriterium für die verzögerte Auswertung von Eigenschaften ist die semantische Äquivalenz zur Auswertung von Eigenschaften mit der Verände-

rung der Berechnungsfunktion oder deren Parametern. Diese semantische Äquivalenz ist ohne zusätzliche Maßnahmen dann gewährleistet, wenn der funktionale Abhängigkeitsgraph $\mathfrak{A}_{\mathcal{U}}$ zyklenfrei ist.

Für den Fall eines zyklischen Abhängigkeitsgraphen ist dies nicht zwingend gewährleistet, wie das folgende Beispiel zeigt.

**Beispiel 11.2**
Sei ein nutzbarer Metatyp $U$ gegeben mit den Eigenschaften `a` und `b`, sowie den Ereignissen $\sigma_1$ und $\sigma_2$ und den Berechnungsfunktionen $f_{\mathtt{a},\sigma_1} = 1$, $f_{\mathtt{a},\sigma_2} = \mathtt{b} + 1$ und $f_{\mathtt{b},\sigma_1} = \mathtt{a}$.

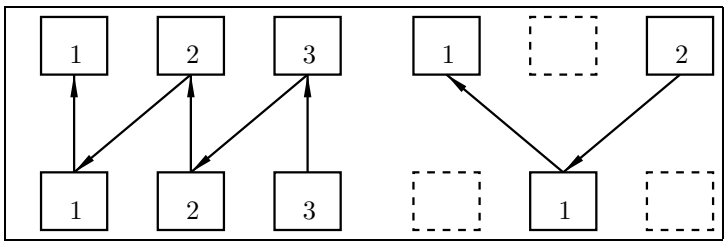



Abbildung 11.1: Verzögerte Auswertung zyklischer Eigenschaften.

Abbildung 11.1 zeigt auf der linken Seite den korrekten Ablauf der Auswertungen für das Auftreten der Ereignisse in der Reihenfolge $\sigma_1, \sigma_2, \sigma_2$ und auf der rechten Seite das Verhalten bei identischer Ereignisfolge und verzögerter Auswertung.

Die Problemstellung bei der verzögerten Auswertung zyklischer Abhängigkeiten liegt in der Löschung der Belegung der Eigenschaften durch vergangene Ereignisse. Würde in der Wissensbasis die vollständige Vergangenheit der Komponente mit allen Ereignissen gespeichert, wäre eine korrekte Ausführung der verzögerten Auswertung möglich.

Die Speicherung der vollständigen Historie einer Eigenschaft stellt allerdings einen erheblichen Aufwand im Bezug auf die Speicherkapazität des Managements dar und ist daher nicht praktikabel. Eine erster Ansatz zur Reduzierung des Speicheraufwands wird durch die Beschränkung auf Eigenschaften, welche an Zyklen beteiligt sind, erreicht. Eine Eigenschaft ist an einem Zyklus beteiligt, wenn sie

a) eine zyklische Berechnungsfunktion enthält oder

b) als Parameter in eine zyklische Berechnungsfunktion eingeht.

Auch die Speicherung der vollständigen Vergangenheit der an einem Zyklus beteiligten Eigenschaften ist mit einem erheblichen Aufwand in Bezug auf die Speicherfähigkeit verbunden. Eine weitere Reduktion des Speicheraufwands kann durch die Analyse des funktionalen Abhängigkeitsgraphen und der potenziellen Ereignisfolgen erreicht werden.

Für die an einem Zyklus beteiligten Eigenschaften müssen nur die Anteile an der Vergangenheit gespeichert werden, welche tatsächlich in einen Zyklus eingehen können. Ein Zyklus in den Berechnungsfunktionen der Eigenschaften ist durch Ereignisse begrenzt. Hat eine Komponente durch ein Ereignis den zyklischen Anteil der Berechnungsfunktionen verlassen, so muss die Vergangenheit der an diesem Zyklus beteiligten Funktionen nicht mehr gespeichert werden. Die Menge der Ereignisse einer Komponente, welche den Zyklus einer Eigenschaft der Komponente beenden, ist durch die Berechnungsfunktionen bestimmt. Die Vergangenheit der bis zum Eintreten eines solchen Ereignisses zyklischen Eigenschaft kann somit mit dem Auftreten des Ereignisses aus dem Speicher entfernt werden.

Für die weiteren an einem Zyklus beteiligten Eigenschaften ist die Festlegung des Endes des Zyklus durch zwei Alternativen möglich. Einerseits können dynamisch zur Laufzeit des Systems diese Komponenten vom Ende des Zyklus benachrichtigt werden. Durch diese Variante wird der belegte Speicher zum frühest möglichen Zeitpunkt freigegeben. Allerdings bedingt diese Alternative unter Umständen einen erhöhten Kommunikationsaufwand für die Benachrichtigung.

Alternativ können für die an einem Zyklus beteiligten Eigenschaften statisch die Ereignisse ermittelt werden, mit welchen die Beteiligung an einem Zyklus zwingend beendet ist. Dies ist der Fall, wenn im funktionalen Abhängigkeitsgraphen keine Kante die Berechnungsfunktion der Eigenschaft mit einer zyklischen Eigenschaft verbindet.

Die letztere Alternative führt unter Umständen zur Speicherung nicht mehr benötigter Belegungen der Eigenschaften, ist im Gegensatz zur dynamischen Variante aber ohne zusätzlichen Kommunikationsaufwand realisierbar.

### 11.2.4 Optimierung der Codeerzeugung

Weitere Optimierungen ergeben sich aus der Optimierung des aus der Spezifikation generierten Codes. Die Ansatzpunkte für diese Optimierungen liefert die Beobachtung, dass die Wissensbasis und die Kommunikationsinfrastruktur des Managements als wesentliche Bestandteile der Basisfunktionalität zeit- und rechenintensive Teile des Systems darstellen. Neben den üblichen Optimierungen des ausführbaren Codes, wie sie in [ASU88b] beschrieben werden, ist die Reduzierung der Zugriffe auf die Wissensbasis und die Kommunikation ein zentraler Ansatzpunkt für effizienten Managementcode.

Die Atomarität der Berechnungen der Belegungen der Eigenschaften ermöglicht eine Reduzierung der Zugriffe auf die Wissensbasis, in dem in jeder generierten Ereignisfunktion das Ergebnis eines Zugriffs auf eine Eigenschaft lokal gespeichert wird und diese lokale Kopie der Belegung im weiteren Verlauf der Berechnungen verwendet wird. Ebenso kann die eine berechnete Belegung einer simultanen Eigenschaft lokal gespeichert werden, falls diese in weitere Berechnungen innerhalb dieser Funktion einfließt.

Der zweite zentrale Ansatzpunkt zur Steigerung der Performanz in einem verteilten System ist die Kommunikation über Stellengrenzen hinweg. Wird die Anzahl der Kommunikationsvorgänge verringert, so erhöht sich die Performanz des Systems. Ein Ansatz zur Reduktion der Kommunikation ist die in Abschnitt 8.1.3 auf Seite 151 im Zusammenhang mit der Realisierung der Wissensbasis bereits diskutierte Replizierung von Informationen.

Ein anderer Ansatz ist die Bündelung von Kommunikationsvorgängen. Werden für die Berechnungen in einer generierten Ereignisfunktion mehrere nicht lokale Eigenschaften angefordert, so können diese Kommunikationsvorgänge zu einer gemeinsamen Kommunikation gebündelt werden, falls die Eigenschaften auf dem selben Knoten realisiert sind.

## 11.3 Zusammenfassung

Durch Strategien zur Erhöhung der Performanz der generierten Systeme kann die Leistungsfähigkeit des Managements erhöht werden. Diese Optimierungen basieren auf der weiteren Analyse der Spezifikationen und des generierten Codes.

Wesentlich ist dabei die Erhaltung der Semantik der Spezifikation durch die Optimierungen. Eine Verletzung dieser Forderung würde den großen Gewinn der Spezifikation in Verbindung mit der Generierung des Managements, die Reduzierung der Fehlerwahrscheinlichkeit und die Handhabbarkeit ad absurdum führen.

Die Bedeutung der Performanz eines Systems darf allerdings nicht überbewertet werden, da die korrekte Ausführung eines Systems ebenfalls von zentraler Bedeutung ist. Im Gegensatz zur Performanz eines Systems kann die Fehleranfälligkeit allerdings nicht quantifiziert werden, wodurch ein objektiver Vergleich zwischen Systemen unter diesem Aspekt erschwert wird.

# 12 Zusammenfassung und Ausblick

In diesem Kapitel werden die wesentliche Aspekte dieser Arbeit zusammengefasst und Ergebnisse dargestellt und bewertet. Im Anschluss daran werden weitere offene Fragestellungen und weitere Forschungsaufgaben aufgezeigt.

## 12.1 Abstrakte Spezifikation und Generierung

Zu Beginn der Arbeit wurden bestehende Ansätze für verteilte Systeme und deren Betriebssysteme dargestellt. Dabei wurden die Schwächen der bestehenden Ansätze herausgearbeitet, die insbesondere aus der hohen Komplexität solcher Systeme resultieren. Dabei wurde die Strukturierung des Managements als zentraler Ansatzpunkt für die Handhabbarkeit von Systemen und deren Management betont. Diese Strukturen ermöglichen die Erarbeitung und Verwaltung von Wissen über die ausgeführten Systeme durch das Management und bilden somit die Grundlage für die Entscheidungen des Managements. Aus dieser Erkenntnis resultiert die Verbindung klassischer Betriebssystemfunktionalität mit Funktionalitäten, die in einer klassischen Sicht nicht dem Betriebssystem zugeordnet werden – wie der Übersetzung von Problembeschreibungen in Code – zu einem Management verteilter Systeme.

Aufbauend auf diesen Erkenntnissen wurde die Notwendigkeit eines Ansatzes begründet, der ausgehend von einer abstrakten Beschreibung des Managements die Generierung des ausführbaren Managements ermöglicht.

Daher stellt der folgende Teil der Arbeit existierende Spezifikationstechniken für verteilte Systeme dar. Dabei zeigt sich, dass bestehende formale Beschreibungsmodelle nicht für die Spezifikation des Managements verteilter Systeme geeignet ist. Für die Spezifikation des Managements mit dem Ziel der Generierung eines ausführbaren, anwendungsangepassten Managements ist somit eine spezielle Spezifikationstechnik notwendig.

Diese Beschreibungsform basiert auf Komponenten, die ein System in Ausführung bilden. Jede Komponente wird von einem Metatypen abgeleitet, der die Eigenschaften der Komponente und der Komponentenklasse spezifiziert. Die Belegung

der Eigenschaften der Komponenten beschreibt den Zustand des Systems. Dabei sind auch die Strukturen des Systems, welche die Grundlage für die Informationsflüsse im System bilden, Teil der Eigenschaften. Die Dynamik der Systeme wird durch Ereignisse und eine potenzielle Folge von Ereignissen beschrieben. Diese Anteile der Spezifikation ermöglichen die Beschreibung von Synchronisationsbedingungen.

Die Verbindung zwischen den Ereignissen und den Eigenschaften der Komponenten wird durch die Berechnungsfunktionen der Eigenschaften gebildet. Mit dem Auftreten von Ereignissen wird so die Belegung der Eigenschaften verändert. Eigenschaften repräsentieren dabei einerseits den Zustand des Systems, andererseit die Entscheidungen des Managements. Die Entscheidungen werden durch Aktionen durchgesetzt.

Die Grundlage für die generative Transformation dieser Spezifikationen in ein ausführbares Management wird durch die Analyse der Beschreibungen gelegt. Diese Analysen ermöglichen auch die Optimierung des erzeugten Managements durch verschiedene Techniken. Der generierte Code stützt sich auf die Basisfähigkeiten einer verteilten Hardwarekonfiguration – Rechen-, Speicher- und Kommunikationsfähigkeit.

Die Anwendbarkeit des Ansatzes wurde durch die prototypische Realisierung des Generators gezeigt. Mit diesem Prototypen wurde eine Reihe von Managementaufgaben spezifiziert und der entsprechende Code generiert. Anhand der Beispiele wurde gezeigt, dass der Ansatz auf allen Ebenen anwendbar ist. Sowohl die Basis eines Systems, wie die Verwaltung von Aktivitäten und Speicher können durch Spezifikationen beschrieben werden, als auch Konzepte von Programmiersprachen auf hohem Niveau für verteilte Systeme. Durch die einheitliche Spezifikation aller Aspekte des Managements eines verteilten, kooperativen Systems werden die Zusammenhänge innerhalb der Systeme deutlich. Diese Ausarbeitung der Strukturen von Systemen ist nicht nur die Grundlage für die Generierung des Managements, sondern bietet auch die Möglichkeit, das Verständnis komplexer Systeme in der Informatik zu erweitern.

Die Performaz eines generierten Managements liegt dabei allerdings unter der einer händisch optimierten Verwaltung. Durch die beschriebenen Optimierungstechniken kann diese Einbuße an Leistungsfähigkeit verringert werden. Dabei ist auch zu beachten, dass diesen Performanzeinbußen erhebliche Gewinne im Bezug auf die Funktionssicherheit und die strukturierte Erweiterbarkeit des Managements gegenüber stehen. Diese sind allerdings nicht messbar und entziehen sich somit einer objektiven Bewertung, wodurch aber die Bedeutung der Zuverlässigkeit insbesondere für verteilte Systeme nicht gemindert wird.

## 12.2 Weiterführende Forschungsaufgaben

Die in dieser Arbeit präsentierten Ansätze für die Spezifikation des Managements verteilter, kooperativer Systeme und deren generative Transformation bilden die Grundlage für eine Reihe weiterführender Forschungsaufgaben. Diese werden im Folgenden kurz diskutiert.

### Analyse von Systemstrukturen

In der vorliegenden Arbeit wurde an verschiedenen Stellen die Bedeutung der Strukturen eines Systems für dessen Handhabbarkeit betont. Durch die Spezifikation eines Systems werden diese Strukturen verdeutlicht und Zusammenhänge sichtbar gemacht. Die Untersuchung dieser Strukturen eröffnet die Möglichkeit, die Zusammenhänge innerhalb des Managements eingehend zu untersuchen und die Abhängigkeiten der einzelnen Teilaspekte zu erkennen. Derartige Erkenntnisse können die Basis für zukünftige Entwicklungen in dem Bereich des Managements verteilter Systeme und dem Bereich der Anwendungen für verteilte Systeme sein. Da in heutigen Systemen der Zusammenhang zwischen Anwendung bzw. der Sprachkonzepte, welche für die Entwicklung von Anwendungen verwendet werden, und dem Management nur wenig formal erfasst ist, bietet die einheitliche Spezifikation aller Aspekte verteilter Systeme die Chance, diese Zusammenhänge umfassend zu untersuchen.

### Visuelle Spezifikation

Die Komplexität des Managements verteilter, kooperativer Systeme, die in den Strukturen der Systeme deutlich wird, wird durch den Einsatz von formalen Spezifikationstechniken handhabbar. Allerdings zeigt sich, dass für den Menschen die rein textuelle Darstellung dieser Zusammenhänge nicht optimal ist. Die graphische Darstellung komplexer Strukturen erhöht die Übersichtlichkeit erheblich, solange die Sicht auf die Strukturen auf wesentliche Aspekte begrenzt wird. Die Festlegung der wesentlichen Aspekte ist dabei von den Zielen, die erreicht werden sollen, abhängig und wechselt in den verschiedenen Stadien der Erstellung von Spezifikationen.

Die gegenwärtige prototypische Realisierung des Generators unterstützt dies durch die Generierung von graphischen Darstellungen der Spezifikation, wie zum Beispiel der Darstellung des funktionalen Abhängigkeitsgraphen. Für die Erstellung von Spezifikationen wäre es allerdings hilfreich, graphische Editoren zur Verfügung zu haben, welche die Erstellung durch unterschiedliche Sichtweisen auf die Spezifikation ermöglichen. Notwendig ist dazu der Entwurf einer bzw. mehrerer graphischer Beschreibungssprachen, die zu der textuellen Darstellung äquivalent sind. Von zentraler Bedeutung ist dabei die semantische Konsistenz

der verschiedenen Sichtweisen auf die Spezifikation, d. h. die Sichtweisen müssen widerspruchsfrei sein und die eindeutige Semantik der Spezifikation erhalten.

### Formalisierung der Semantik

Ein entscheidendes Kriterium für eine Spezifikationssprache ist die Festlegung der Semantik der Spezifikationen. Die in dieser Arbeit erarbeitete Spezifikationssprache besitzt eine wohldefinierte Semantik, so dass die Eigenschaften des spezifizierten Managements vollständig festgelegt sind.

Als Grundlage für eine vollständige Verifikation des spezifizierten Managements und eines formalen Nachweises der Korrektheit des Generators ist allerdings eine vollständige formale Beschreibung der Semantik der Spezifikationssprache erforderlich. Im Gegensatz zu den Semantiken von Programmiersprachen für sequentielle Systeme muss eine derartige Semantik auch die zeitlichen Aspekte der Spezifikation erfassen.

Als Ansatz für die Formalisierung der Semantik bietet sich ein hybrider Ansatz aus den Techniken für die Beschreibung der Semantik von Programmiersprachen und Kalkülen für nebenläufige Systeme an, wobei letztere so erweitert werden müssen, dass die hohe Dynamik des Managements erfassbar wird.

### Formale Verifikation von Spezifikationen

Die Formalisierung der Semantik der Spezifikationen bildet die Grundlage für den formalen Nachweis weiterer Eigenschaften des spezifizierten Managements. Erste Ansätze für diese Verifikation der Spezifikationen wurden in dieser Arbeit mit dem Nachweis der Konsistenz der Ereignisfolgen und den Erkenntnissen, welche aus dem funktionalen Abhängigkeitsgraphen abgeleitet werden können, erbracht.

Insbesondere sind dabei die zeitlichen Zusammenhänge innerhalb des Managements und zwischen dem Management und der Anwendung von Interesse. Die besondere Schwierigkeit liegt dabei in der Tatsache, dass für die Verifikation nur die Spezifikation des Managements zur Verfügung steht und das Verhalten der Anwendungen nicht erfasst werden kann. Allerdings ermöglichen die Metatypen und die in der Spezifikation enthaltenen Strukturen die Erarbeitung von Teilaspekten, welche durch die Anwendung erfüllt werden müssen.

Diese Ableitungen ermöglichen auf der anderen Seite aber auch den Nachweis der Korrektheit von Problemlösungsbeschreibungen. Eine Problemlösungsbeschreibung ist korrekt, wenn diese aus Komponenten bzw. Komponentenklassen aufgebaut ist, die aus Metatypen entstehen und die Ausführung die in der Spezifikation der Metatypen enthaltenen Restriktionen erfüllt. Insbesondere erfordert dies die Erzeugung von Ereignissen in einer Ordnung, welche die Reihenfolgerestriktionen des Managements erfüllen kann.

Die Basis dieser Analyse bildet der generierte Code für die Komponenten bzw. die Komponentenklassen. Die Erzeugung dieses Codes wird durch die Berechnungsfunktionen der entsprechenden Eigenschaften gesteuert. Somit besteht ein weiterer Zusammenhang zwischen den Berechnungsfunktionen des Managements und dem korrekten Ablauf eines Systems.

### Verbesserung der prototypischen Realisierung

Neben dem eigentlichen Generator für die Erzeugung des Managements eines verteilten, kooperativen Systems stellt die Basisfunktionalität und insbesondere die Wissensbasis eine zentrale Komponente der Realisierung dar. In der gegenwärtigen Realisierung baut die Basisfunktionalität auf einem LINUX-System auf. Eine erhebliche Steigerung der Performanz des Systems, sowie eine Flexibilisierung des Managements kann erreicht werden, wenn die Basisfunktionalität auf einer minimalen Abstraktion der Hardware aufbaut. Dies ließe sich durch den Einsatz eines Mikro- oder Nanokerns als Grundlage der Basisfunktionalität erreichen.

Die Realisierung der abstrakten Datentypen stellt ebenfalls ein Gebiet für weitere Forschungsaufgaben dar, da diese wesentlich zur Ausdrucksstärke der Spezifikationen beitragen. Von besonderem Interesse ist dabei die Auflösung von abstrakten Datentypen. Für diese Aufgabe würde sich der Einsatz einer automatischen Speicherbereinigung (Garbage Collection) anbieten.

# Anhang

# A Mathematische Grundlagen

In diesem Anhang werden kurz die mathematischen Grundlagen der Graphentheorie und der Automatentheorie erläutert, die in der vorliegenden Arbeit vorausgesetzt werden. Detaillierte Einführungen in die genannten Bereiche finden sich in [Die00] und [HMU01].

## A.1 Grundlagen der Graphentheorie

Eine der zentralen mathematischen Strukturen in dieser Arbeit sind Graphen. Im Folgenden sind die zentralen Definitionen für Graphen zusammengefasst.

**Definition A.1 (ungerichteter Graph)**
*Sei $V$ eine endliche Menge von Knoten und $E \subseteq \{\{v_1, v_2\} \mid v_1, v_2 \in V\}$ eine Menge von (ungerichteten) Kanten. Dann heißt $\mathfrak{G} = \langle V, E \rangle$ ein (ungerichteter) Graph.*

Aufgrund der Festlegung, dass $V$ eine endliche Menge ist, ist auch die Menge $V \times V$ endlich und somit die Menge $E$. Gemäß dieser Definition gibt es in einem ungerichteten Graphen keine mehrfachen Kanten.

**Definition A.2 (gerichteter Graph)**
*Sei $V$ eine endliche Menge von Knoten und $E \subseteq V \times V$. Dann heißt $\mathfrak{G} = \langle V, E \rangle$ ein ungerichteter Graph; $(v_1, v_2) \in E$ heißt gerichtete Kante von $v_1$ nach $v_2$.*

In beiden Fällen wird eine Kante $(v, v)$ wird als Schleife bezeichnet.

Graphen können auf einfache Weise visualisiert werden. Sei $\mathfrak{G} = \langle V, E \rangle$ ein gerichteter Graph, mit $V = \{1, 2, 3, 4\}$ und $E = \{(1, 2), (1, 3), (3, 4)(1, 4)$ dann ergibt sich die Visualisierung aus Abb. A.1

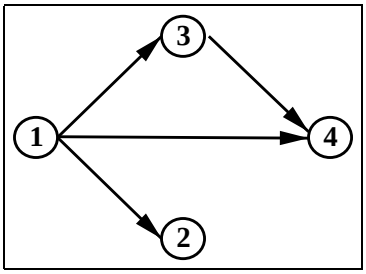



Abbildung A.1: Visualisierung von Graphen.

In Graphen können für bestimmte Anwendungen Unterstrukturen bestimmt werden.

**Definition A.3 (Teilgraph)**
*Sei $\mathfrak{G} = \langle V, E \rangle$ ein gerichteter (ungerichteter) Graph. Sei $V' \subseteq V$ und $E' \subseteq E$, sowie $\mathfrak{G}' = \langle V', E' \rangle$ ein gerichteter (ungerichteter) Graph. Dann heißt $\mathfrak{G}'$ Teilgraph von $\mathfrak{G}$.*

Eine typische Aufgabestellung bei Graphen ist das Durchlaufen von Kanten von einem Knoten zu einem anderen bzw. die Fragestellung, ob eine Verbindung zwischen zwei Knoten durch Kanten existiert.

**Definition A.4 (Kantenzug)**
*Sei $\mathfrak{G} = \langle V, E \rangle$ ein gerichteter (ungerichteter) Graph. Eine Folge $v_1, \ldots, v_n$ von Knoten in $V$ heißt Kantenzug von $v_1$ nach $v_n$ in $\mathfrak{G}$, wenn entweder $(v_i, v_{i+1}) \in E$ oder $(v_{i+1}, v_i) \in E$ für alle $i = 1, \ldots, n-1$ gilt. Die Länge des Kantenzuges ist $n - 1$.*

In einem Kantenzug dürfen Kanten mehrfach durchlaufen werden. In gerichteten Graphen auch entgegen der Orientierung.

**Definition A.5 (Weg)**
*Sei $\mathfrak{G} = \langle V, E\rangle$ ein ungerichteter Graph. Ein Kantenzug $v_1, \ldots, v_n$ von Knoten in $V$ heißt ein Weg von $v_1$ nach $v_n$ in $\mathfrak{G}$, wenn die Kanten paarweise verschieden sind.*

*Sei $\mathfrak{G} = \langle V, E\rangle$ ein gerichteter Graph. Ein Kantenzug $v_1, \ldots, v_n$ von Knoten in $V$ heißt ein Weg von $v_1$ nach $v_n$ in $\mathfrak{G}$, wenn die Kanten paarweise verschieden sind und wenn $(v_i, v_{i+1} \in E$ für alle $i = 1, \ldots, n-1$ gilt.*

*Die Länge des Weges ist $n-1$. Der Knoten $v_n$ ist in $\mathfrak{G}$ erreichbar, wenn es einen Weg von $v_1$ nach $v_k$ gibt.*

*Die Menge aller Wege von $v_1$ nach $v_k$ in $\mathfrak{G}$ wird als $W(v_1, v_n)$ bezeichnet.*

Im Gegensatz zu einem Kantenzug sind die benutzten Kanten eines Weges paarweise verschieden und die Kanten dürfen im Falle eines gerichteten Graphen nur vom Start zum Ziel hin durchlaufen werden.

**Definition A.6 (Zyklus, Kreis)**
*Sei $\mathfrak{G} = \langle V, E\rangle$ ein ungerichteter Graph. Ein Weg $v_1, \ldots, v_n$ heißt Kreis in $\mathfrak{G}$, wenn $v_1 = v_n$ gilt.*

*Sei $\mathfrak{G} = \langle V, E\rangle$ ein gerichteter Graph. Ein Weg $v_1, \ldots, v_n$ heißt Zyklus in $\mathfrak{G}$, wenn $v_1 = v_n$ gilt.*

**Definition A.7 (Azyklischer Graph)**
*Ein gerichteter Graph $\mathfrak{G} = \langle V, E\rangle$ ohne Zyklen heißt azyklischer oder zyklenfreier Graph.*

Eine besondere Bedeutung haben Graphen, bei denen es zwischen je zwei Knoten $v_i$, $v_j$, $v_i \neq v_j$ einen Kantenzug gibt. Ein solcher Graph wird als zusammenhängend bezeichnet.

Eine ausgezeichnete Klasse der Graphen sind die *Bäume*. Ein zyklenfreier zusammenhängender Graph $\mathfrak{G} = \langle V, E\rangle$ wird als Baum bezeichnet, wenn es genau eine

Wurzel $w \in V$ gibt, d. h. es existiert kein Knoten $v \in V$, so dass $(v, w) \in E$. Formal ist ein Baum folgendermaßen definiert:

**Definition A.8 (Baum)**
*Ein ungerichteter Graph ist ein ungerichteter Baum, wenn er kreisfrei ist und je zwei Knoten durch genau einen Weg verbunden sind.*

*Ein gerichteter Graph ist ein gerichteter Baum, wenn er eine Wurzel $w$ besitzt und von der Wurzel zu jedem Knoten genau ein Weg existiert.*

Zusammenhängende Graphen können ausgehend von einem Knoten durchlaufen werden. Für das Durchlaufen eines Graphen existieren zwei alternative Verfahren:

1. **Tiefensuche (DFS)**
   Kanten werden ausgehend von dem zuletzt entdeckten Knoten, der mit noch unerforschten Kanten inzident ist, erforscht. Erreicht man von $v$ aus einen noch nicht erforschten Knoten $v'$, so verfährt man mit $v'$ analog zu $v$. Wenn alle mit $v'$ inzidenten Kanten erforscht sind, erfolgt ein Backtracking zu $v$. Die Knoten erhalten in der Reihenfolge ihres Erreichens eine DFS-Nummer $t(v)$. Zu Beginn gilt für alle $v \in V$: $t(v) = \infty$.

   Algorithmisch kann die Tiefensuche folgendermaßen definiert werden:

   Sei $\mathfrak{G} = \langle V, E \rangle$ ein Graph. Für einen Knoten $v$ sei die Prozedur `DFSEARCH(`$v$`)` wie folgt definiert:

   $i$:=$i+1$; $t(v)$:=$i$; $N(v)$:=$\{w|(v,w) \in E\}$;
   `while` $\exists w \in N(V)$ mit $t(w) = \infty$ `do`
   $N(v)$:=$N(v)/\{w\}$;
   $B$:=$B \cup \{(v,w)\}$;
   `DFSEARCH(`$w$`)`
   `end`

   Der Algorithmus zur Tiefensuche ist dann:

   $B$:=$\emptyset$; $i$:=0;
   `for all` $v \in V$ `do` $t(v)$:=$\infty$;
   `while` $\exists v \in V$ mit $t(v) = \infty$ `do`
   `DFSEARCH(`$v$`);`

2. **Breitensuche (BFS)**
   Bei der Breitensuche werden zunächst alle noch nicht erreichten Nachbarn

eines Knotens besucht, bevor die Suche in der Tiefe fortgesetzt wird. Dazu werden alle besuchten Knoten in eine Warteschlange eingereiht. Ein Knoten gilt als erreicht, wenn er in die Warteschlange aufgenommen wurde. Solange die Warteschlange nicht leer ist, selektiert man einen Knoten aus der Warteschlange und geht wie oben beschrieben vor. Die Knoten $v$ erhalten in der Reihenfolge ihres Herausnehmens aus der Warteschlange eine BFS-Nummer $b(v)$.

Algorithmisch kann die Breitensuche folgendermaßen definiert werden:

Sei $\mathfrak{G} = \langle V, E \rangle$ ein Graph. Für einen Knoten $v$ sei die Prozedur `BFSEARCH(`$v$`)` wie folgt definiert:

$i$:=$i+1$; $b(v)$:=$i$; $N(v)$:=$\{w|(v,w) \in E\}$;
**while** $\exists w \in N(v)$ mit $r(w) = false$ **do**
    $N(v)$:=$N(v)/\{w\}$;
    $B$:=$B \cup \{(v,w)\}$;
    füge $w$ an $W$ an;
    $r(w)$:=$true$;
**end**

Diese Prozedur wird für den jeweils ersten Knoten der Warteschlange $W$ ausgeführt.

$B$:=$\emptyset$; $i$:=0; $W$:=();
**for all** $v \in V$ do
    $b(v)$:=$\infty$; $r(v)$:=$false$;
**end**
**while** $W = ()$ **and** $\exists v \in V$ mit $b(v) = \infty$ **do**
    $W(v)$:=$(v)$;
    $r(v)$:=$true$;
    **while**$W \neq ()$ **do**
        wähle ersten Knoten $w$ aus $W$;
        entferne $w$ aus $W$;
        `BFSEARCH(`$v$`)`;
    **end**
**end**

Die Knoten gerichteter azyklischer Graphen können durch eine Halbordnung geordnet werden:

**Lemma A.1**
*Sei $\mathfrak{G} = \langle V, E \rangle$ ein gerichteter, azyklischer Graph. Die Relation $R \subseteq V \times V$ mit*

$$(v, w) \in R \Leftrightarrow W(v, w) \neq \emptyset$$

*definiert eine Halbordnung auf $V$, d. h. es gilt:*

1. $\forall v \in V : (v, v) \in R$ *(Reflexivität)*
2. $\forall v, w \in V : (v, w) \in R \wedge (w, v) \in R \Rightarrow v = w$ *(Antisymmetrie)*
3. $\forall u, v, w \in V : (u, v) \in R \wedge (v, w) \in R \Rightarrow (u, w) \in R$ *(Transitivität)*

Die topologische Sortierung erzeugt eine vollständige Ordnung, die nicht im Widerspruch zur partiellen Ordnung steht.

**Definition A.9 (Topologische Sortierung)**
*Sei $\mathfrak{G} = \langle V, E \rangle$ ein gerichteter azyklischer Graph mit $V = \{v_1, \ldots, v_n\}$. Eine topologische Sortierung ist eine Abbildung*

$$ord : V \to \{1, \ldots n\} \text{ mit } |V| = n,$$

*so dass mit $(u, v) \in E$ auch $ord(u) < ord(v)$ gilt.*

*Für die Knoten $u, v$ wird dann $u \prec_{\mathfrak{G}} v$ geschrieben.*

**Lemma A.2**
*Sei $\mathfrak{G} = \langle V, E \rangle$ ein azyklischer gerichteter Graph, dann und nur dann existiert eine topologische Sortierung von $\mathfrak{G}$.*

**Beweis**

a) $\Leftarrow$
Folgt aus der Definition.

b) $\Rightarrow$
Durch Induktion über $|V|$:

Indunktionsanfang: $|V| = 1$
Da $\mathfrak{G}$ zyklenfrei ist, ist $\mathfrak{G}$ insbesondere auch schleifenfrei. Somit ist $E = \emptyset$.

Indunktionsschritt: $|V| = n$

- Da $\mathfrak{G}$ azyklisch ist, muss es einen Knoten $v$ ohne Vorgänger geben. Setze $ord(v) = 1$.
- Durch Entfernen von $v$ erhält man einen azyklischen Graphen $\mathfrak{G}'$ mit $|V'| = n-1$, für den es nach Induktionsvoraussetzung eine topologische Sortierung $ord'$ gibt.
- Die gesuchte topologische Sortierung für $\mathfrak{G}$ ergibt sich durch $ord(v') = ord'(v') + 1$, für alle $v' \in V'$.

*q.e.d.*

Für die Realisierung von Graphen in Rechnern existieren eine Reihe von Möglichkeiten. Die Speicherung kann durch Matritzen oder verzeigerte Datenstrukturen erfolgen. Im Rahmen dieser Arbeit wird eine Bibliothek der Universität Passau zur Verarbeitung von Graphen eingesetzt (vgl. [FPM]).

## A.2 Grundlagen der regulären Sprachen

Reguläre Ausdrücke sind ein zentrales Element der Spezifikationen, die in dieser Arbeit entwickelt wurden. Reguläre Ausdrücke $R$ über einem Alphabet $\Sigma$ sind folgendermaßen definiert:

1. **Leere Menge**
   $R = \emptyset$ ist ein regulärer Ausdruck.
2. **Leeres Wort**
   $R = \varepsilon$ ist ein regulärer Ausdruck.
3. **Sei** $a \in \Sigma$
   $R = a, \quad a \in \Sigma$
4. **Seien** $a$ **und** $b$ **reguläre Ausdrücke**

   **a)** $R = a + b$ ist ein regulärer Ausdruck.

   **b)** $R = ab$ ist ein regulärer Ausdruck.

**c)** $R = a*$ ist ein regulärer Ausdruck.

Mit diesen Definitionen kann eine Grammatik für reguläre Ausdrücke angeben werden:

| <R> ::= $\varepsilon$ \| a \| <R><R> \| <R>+<R> \| <R>* |
|---|

### A.2.1 Generierung von NEA aus regulären Ausdrücken

Reguläre Ausdrücken sind äquivalent zu den nicht deterministischen endlichen Automaten (NEA):

Ein endlicher Automat ist ein 5-Tupel $A = (Z, \Sigma, \delta, z_0, F)$, wobei

- $Z$ eine endliche Menge von Zuständen
- $\Sigma$ eine endliche Zeichenmenge, das Alphabet
- $\delta$ eine Relation, die einem Paar aus $Z \times (\Sigma \cup \varepsilon)$ eine Menge von Folgezuständen zuordnet ($\delta : Z \times (\Sigma \cup \varepsilon) \rightarrow 2^Z$)
- $z_0$ der Startzustand
- $F$ eine Menge von Endzuständen ($F \subseteq Z$)

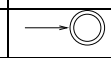
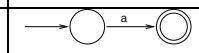
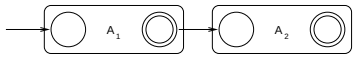
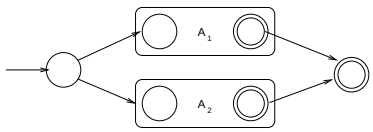
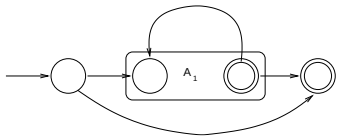
ist.

Ein NEA befindet sich zunächst im Startzustand $z_0$. Wenn ein Eingabezeichen aus $\Sigma$ gelesen wird, so ist mit der Funktion $\delta$ die Menge der Nachfolgezustände festgelegt. Zustandsübergänge können auch stattfinden, wenn kein Zeichen gelesen wird, wenn für mindestens einen der aktuellen Zustände $z$ gilt: $\delta(z, \varepsilon) \neq \emptyset$.

Nicht-deterministische endliche Automaten können als gerichtete Graphen dargestellt werden. Dabei sind die Knoten die Zustände; die Kanten repräsentieren die Zustandsübergänge gemäß der Relation $\delta$. Es existiert eine mit dem Zeichen $a$ markierte und gerichtete Kante von Knoten $k_n$ zu $k_m$, genau dann, wenn $z_m \in \delta(z_n, a)$.

Nicht-deterministische endliche Automaten können nach folgendem Verfahren aus einem regulären Ausdruck erzeugt werden:

Jeder Produktion wird eine semantische Aktion gemäß der folgenden Tabelle zugeordnet, die angibt, wie der entsprechende Automat zu konstruieren ist.

| Produktion | Automat | Bemerkungen |
|---|---|---|
| <R>::=$\varepsilon$ |  | |
| <R>::=a |   | |
| <R>::=<R><R> |   | Alle Endzustände des Automaten des linken Ausdrucks ($A_1$) werden mit dem Startzustand des rechten Ausdrucks ($A_2$) mit einer $\varepsilon$-Kante verbunden. |
| <R>::=<R>+<R> |   | Ein neuer Startzustand wird erzeugt, von dem zwei neue $\varepsilon$-Kanten in die Startzustände der Automaten des linken ($A_1$) und rechten ($A_2$) Ausdrucks führen.<br>Ebenso wird ein neuer Endzustand eingeführt. Alle Endzustände des linken und rechten Ausdrucks werden mit ihm mit $\varepsilon$-Kanten verbunden. |
| <R>::=<R>* |   | Ein neuer Startzustand wird eingeführt, von dem eine $\varepsilon$-Kante in den Startzustand des Automaten ($A_1$) der rechten Seite führt.<br>Dazu kommt noch ein neuer Endzustand, der mit allen Endzuständen von $A_1$ mit $\varepsilon$-Kanten verbunden wird.<br>Zusätzlich werden $\varepsilon$-Kanten von allen Endzuständen zum Startzustand von $A_1$ eingeführt. |

Ein regulärer Ausdruck $a{+}(bc)*$ sieht als Syntaxbaum aus, wie in Abbildung A.2 auf der nächsten Seite dargestellt.

Der entsprechende Automat wird nach den Regeln aus obigen Tabelle erzeugt, indem sie aufsteigend von den Blättern angewendet werden (Abbildung A.3 auf der nächsten Seite).

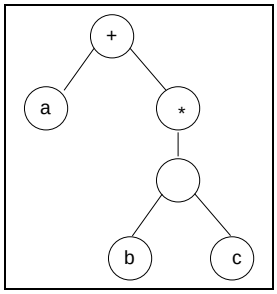



Abbildung A.2: Syntaxbaum für $a + (bc)*$

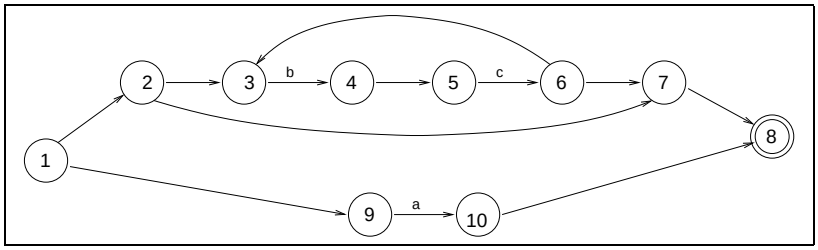



Abbildung A.3: endlicher Automat für $a + (bc)*$

### A.2.2 Umwandlung von NEA in DEA

Die nicht deterministischen endlichen Automaten lassen sich, wie eben gesehen, sehr einfach aus regulären Ausdrücken generieren. Allerdings befindet sich ein solcher Automat im Allgemeinen in mehreren Zuständen gleichzeitig. So befindet sich der Automat aus Abbildung A.3 zum Beispiel in den Zuständen 1,2,3,7,8 und 9, noch bevor ein Zeichen verarbeitet wurde.

Die deterministischen endlichen Automaten (DEA) befinden sich dagegen immer nur in einem Zustand. Deshalb ist dort die $\delta$-Funktion eine Abbildung $Z \times \Sigma \rightarrow Z$. Beide Formen der endlichen Automaten sind äquivalent. Das heißt, dass es zu jedem NEA einen DEA gibt, der dieselbe Sprache akzeptiert und umgekehrt.

Daher kann aus einem NEA ein äquivalenter DEA konstruiert werden:

Der NEA wird mit immer länger werdenden Wörtern simuliert. Dabei ergeben sich mit jedem Schritt Teilmengen, in denen sich der NEA nach dem Lesen des Wortes befindet. Diese Zustandsmengen werden die neuen Zustände des DEA. Irgendwann kommen keine neuen Teilmengen hinzu. Spätestens dann, wenn jede mögliche Teilmenge der NEA-Zustände zu den DEA-Zuständen hinzugefügt wurde, stoppt der Algorithmus.

Nachdem das grobe Funktionsprinzip geklärt ist, kann der Algorithmus konstruiert werden. Dazu wird zunächst eine Abbildung benötigt, die $\varepsilon$-Hülle, oder einfach $\varepsilon^* : 2^Z \rightarrow 2^Z$. Dabei enthält $\varepsilon^*(P)$ all jene Zustände, die von einem Zustand aus $P$ über einen mit $\varepsilon$ markierten Pfad erreichbar sind. Dafür die eignet sich der Algorithmus aus Abb. A.4.

Eingabe: $P \subseteq Z$
Ausgabe: $Ergebnis \subseteq Z$

$Ergebnis$:=$P$
Loop
    für alle $z \in Ergebnis$:
        $Ergebnis := Ergebnis \cup \delta(z, \varepsilon)$
    Ende
Bis keine neuen Zustände zu $Ergebnis$ hinzukamen

Abbildung A.4: Berechnung von $\varepsilon^*$

Um die Korrektheit dieses Algorithmus nachzuweisen, müssen drei Fragen beantwortet werden:

a) ist $Ergebnis \subseteq \varepsilon^*(P)$ ?

b) ist $\varepsilon^*(P) \subseteq Ergebnis$ ?

c) terminiert der Algorithmus?

**Zu 1.** (Induktion über die Schleifendurchläufe):

- $P \subseteq \varepsilon^*(P)$ ist trivialerweise richtig
- Angenommen, dass $Ergebnis \subseteq \varepsilon^*(P)$ im n-ten Schleifendurchlauf, dann gilt für alle hinzugefügten Zustände $z \in \varepsilon^*(P)$
- Es geht allerdings auch einfacher: Die Behauptung ist eine Schleifeninvariante. Sie gilt also bei jedem Schleifendurchlauf und damit auch nach der Schleife.

**Zu 2.**:
Angenommen $\varepsilon^*(P) \not\subseteq Ergebnis$. Das heißt, dass mindestens ein Zustand $z$ existiert, der in $\varepsilon^*(P)$ ist, aber nicht in $Ergebnis$. Dass $z \in \varepsilon^*(P)$, bedeutet auch, dass es einen mit $\varepsilon$ markierten Pfad von einem Zustand $p \in P$ zu $z$ der Länge $n \in I\!N$ gibt. Mit Induktion läßt sich zeigen, dass dann $z$ sich eben *doch* in $Ergebnis$ befindet:

Behauptung: Für eine gegebene (aber beliebig große) Pfadlängen $n$ aus $\varepsilon$-Kanten befinden sich alle Zustände, die sich über diese Pfade von $P$ aus erreichen lassen, in $Ergebnis$.

Induktionsanfang:
Aus $Ergebnis{:=}P \Rightarrow P \subseteq Ergebnis$. Das sind gerade alle Pfade der Länge 0.

$n \to n+1$:
Angenommen, dass die Behauptung für die Pfadläge $n$ richtig ist. Dann werden alle $\varepsilon$-Kanten, die von einem Zustand ausgehen, der über einen Pfad der Länge $n$ erreichbar ist, mit $Ergebnis := Ergebnis \cup \delta(z, \varepsilon)$ hinzugefügt. Das sind genau jene, die sich über einen Pfad der Länge $n+1$ erreichen lassen.

**Zu 3.** (Terminierung):
Bei jedem Schleifendurchlauf wird mindestens ein Zustand in $Ergebnis$ eingefügt. Der Algorithmus terminiert spätestens dann, wenn $Ergebnis = Z$

Mit der $\varepsilon$-Hülle kann nun eine Funktion $\hat{\delta} : 2^Z \times \Sigma^* \to 2^Z$ definiert werden, die für eine Teilmenge von Zuständen und ein Wort $w = va, \quad a \in \Sigma$ über $\Sigma$ die Zustandsmenge berechnet, in der sich der Automat nach dem Lesen dieses Wortes befindet:

$$\hat{\delta}(P, \varepsilon) = \varepsilon^*(P)$$

und

$$\hat{\delta}(P, va) = \varepsilon^* \left( \bigcup_{p \in \hat{\delta}(P,v)} \delta(p, a) \right)$$

Die Behauptung, dass diese Funktion genau die Zustandsmenge berechnet, in der sich der Automat nach dem Lesen von $w$ befindet, ist zu beweisen.

Induktion über die Wortlänge:

Für $w = \varepsilon$ ist die Behauptung sicher richtig.

Angenommen, die Behauptung ist richtig für ein Wort $v \in \Sigma^*$. Dann gilt für $\hat{\delta}(P, va)$ $(a \in \Sigma)$: In der Ergebnismenge befinden sich zunächst alle Zustände, die von $\hat{\delta}(P, v)$ mit $a$ unmittelbar erreicht werden können. Dazu kommen noch die Zustände, die man von dort aus erreicht, ohne ein weiteres Zeichen zu lesen, also die $\varepsilon$-Hülle. Da die Behauptung für $\hat{\delta}(P, v)$ richtig ist (nach Annahme), befinden sich in der Ergebnismenge genau jene Zustände, in denen sich der Automat nach Lesen von $va$ befindet.

Jetzt kann der (rekursive) Algorithmus konstruiert werden, der aus einem NEA $A = (Z, \Sigma, \delta, z_0, F)$ einen DEA $A' = (Z', \Sigma, \delta', z_0', F')$ erzeugt (siehe Abb. A.5 auf der nächsten Seite).

Eingabe: ein NEA $A = (Z, \Sigma, \delta, z_0, F)$
Ausgabe: ein äquivalenter DEA $A' = (Z', \Sigma, \delta', z_0', F')$

$z_0' := \varepsilon^*(\{z_0\})$
$Z' := \{z_0'\}$
wenn $z_0' \cap F \neq \emptyset$
    $F' := F' \cup \{z_0'\}$
ende
findeWeitereZustände($z_0'$)

Prozedur findeWeitereZustände($zustand \in 2^Z$)
Anfang
    für alle $a \in \Sigma$:
        für alle $z \in zustand$ :
            $neuerZustand := neuerZustand \cup \delta(z, a)$
        Ende
        $neuerZustand := \varepsilon^*(neuerZustand)$
        wenn $neuerZustand \neq \emptyset$
            $\delta' := \delta' \cup \{(zustand, a) \rightarrow neuerZustand\}$
            wenn $neuerZustand \notin Z'$
                $Z' := Z' \cup \{neuerZustand\}$
                findeWeitereZustände($neuerZustand$)
                wenn $neuerZustand \cap F \neq \emptyset$
                    $F' := F' \cup \{neuerZustand\}$
                Ende
            Ende
        Ende
    Ende
Ende

Abbildung A.5: Umwandlung eines NEA in einen DEA

Der konstruierte DEA $A'$ ist äquivalent zum übergebenen NEA $A$. Für jedes Wort $w \in \Sigma*$ gilt also $w \in L(A') \Leftrightarrow w \in L(A)$.

### Beweis

Die Prozedur `findeWeitereZustände` berechnet die $\hat{\delta}$-Funktion. Der übergebene Zustand ist also die Teilmenge, in der sich der NEA nach dem Lesen eines Wortes $v$ befindet. Dann wird die $\hat{\delta}$-Funktion für alle Wörter $va$, $a \in \Sigma$ berechnet. Dabei ergibt sich jeweils eine Zustandsmenge. Ist diese neu, also noch nicht in

$Z'$ enthalten, wird sie in $Z'$ hinzugefügt und `findeWeitereZustände` rekursiv aufgerufen. Der $\delta'$-Funktion wird also genau dann ein Übergang $(zustand, a) \rightarrow neuerZustand$ hinzugefügt, wenn $\varepsilon^* \left(\bigcup_{z \in zustand} \delta(z, a)\right) = neuerZustand$. Ebenso gilt, dass $neuerZustand$ genau dann $F'$ hinzugefügt wird, wenn in $neuerZustand$ mindestens ein Endzustand des NEA enthalten ist.

Terminierung: Die Prozedur wird nur aufgerufen, wenn ein neuer Zustand hinzukommt. Da die Menge aller Teilmengen der NEA-Zustände $\left|2^Z\right| = 2^{|Z|}$ beträgt, terminiert der Algorithmus in endlicher Zeit spätestens dann, wenn alle diese Teilmengen gefunden wurden. *q.e.d.*

### A.2.3 Minimierung

Die nach dem eben beschriebenen Verfahren erzeugten deterministischen Automaten enthalten im Allgemeinen mehr Zustände und Übergänge, als eigentlich notwendig wären. Das heißt, dass es häufig Automaten gibt, die mit weniger Zuständen und Übergängen die selbe Sprache akzeptieren. So zeigt Abbildung A.6, das Ergebnis für den Fall, dass man den DEA für den Ausdruck $a * b$ konstruiert.

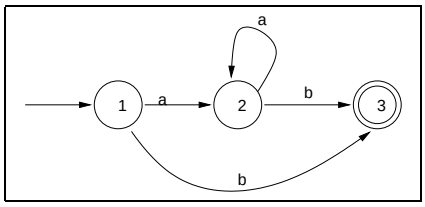



Abbildung A.6: Automat für $a * b$

Es ist ziemlich offensichtlich, dass hier ein Zustand überflüssig ist.

Die Tatsache, dass für Automaten oft eine kleinere Version existiert, wäre an sich noch keine nähere Betrachtung wert. Ob nun ein Automat ein paar Zustände mehr oder weniger hat, dürfte in der Praxis nur wenige hundert Bytes Speicherplatz ausmachen. Wichtig ist die Minimierung aber zum Beispiel dann, wenn zwei Sprachen anhand der zugehörigen Automaten auf Gleichheit überprüft werden sollen. Minimiert man diese nämlich, sind sie bis auf Umbenennung der Zustände (Isomorphie) gleich ([HMU01]).

In diesem Abschnitt soll das Verfahren zur Minimierung von deterministischen endlichen Automaten vorgestellt werden. Die Grundlage bildet der Satz von Myhill-Nerode:

**Satz A.3 (Satz von Myhill-Nerode)**
*Die folgenden drei Aussagen sind äquivalent:*

*a) Die Sprache $L \subseteq \Sigma*$ wird von einem endlichen Automaten akzeptiert, ist also regulär.*

*b) $L$ ist die Vereinigung von Äquivalenzklassen einer rechtsinvarianten[1] Äquivalenzrelation mit endlichem Index[2].*

*c) Die Äquivalenzrelation*

$$R_L = \{(x, y) \in \Sigma * \times \Sigma * | \forall z \in \Sigma* : xz \in L \Leftrightarrow yz \in L\}$$

*hat einen endlichen Index.*

Der Beweis dieses Satzes findet sich in [HMU01]. Wichtig ist an dieser Stelle die Tatsache, dass es eine solche Äquivalenzrelation gibt und dass die Menge der Äquivalenzklassen endlich ist.

Seien $x, y \in \Sigma*$ zwei Wörter, für die $(x, y) \in R_L$ gilt. Dann gilt für eine beliebige Zeichenfolge $z \in \Sigma*$: Wenn $xz$ in $L$ ist, ist $yz$ ebenfalls in $L$. $x$ und $y$ verhalten sich auf jeden Fall hinsichtlich ihrer Akzeptanz durch den Automaten gleich. Für den Automaten aus Abbildung A.6 auf der vorherigen Seite wären das zum Beispiel $\varepsilon$ und $a$.

Der Automat befindet sich nach dem Lesen von $x$ und $y$ im Zustand $z_i$ beziehungsweise $z_j$ (im Beispiel $\varepsilon$:1 und $a$:2). Diese Zustände kann man auch zu einem zusammenfassen. Die Äquvalenzklassen von $R_L$ implizieren Äquivalenzklassen der Zustände bezüglich einer Relation $R_Z \subseteq Z \times Z$:

$$(x, y) \in R_L \Leftrightarrow (\forall z \in \Sigma* : xz \in L \Leftrightarrow yz \in L)$$

$$\Leftrightarrow (\delta(z_0, xz) \in F \Leftrightarrow \delta(z_0, yz) \in F)$$

$$\Leftrightarrow (\delta(\delta(z_0, x), z) \in F \Leftrightarrow \delta(\delta(z_0, y), z) \in F)$$

$$\Leftrightarrow (\delta(z_0, x), \delta(z_0, y)) \in R_Z$$

Das bedeutet, wenn $y$ in der Äquivalenzklasse von $x$ ist, ist der Zustand, in dem sich der Automat nach dem Lesen von $y$ befindet, in der Äquivalenzklasse des Zustands nach dem Lesen von $x$.

Der Minimalautomat wird dann konstruiert, indem die Zustände durch ihre Äquivalenzklassen ersetzt werden. Bleibt, ein Verfahren zum Bestimmen der Äquivalenzklassen der Zustände zu finden. Aus den eben gezeigten Formeln kann einfach ein solcher Algorithmus konstruiert werden.

[1] $\forall w \in \Sigma* : (u, v) \in R \Rightarrow (uw, vw) \in R$
[2] endlicher Index = endliche Menge von Äquvalenzklassen

Dazu ist es zweckmäßig, die Relation $R_Z$ als Matrix aufzuschreiben. Da $R_Z$, wie es sich für eine Äquivalenzrelation gehört, symmetrisch und reflexiv ist, reicht es, nur die Hälfte unter der Diagonale der Matrix zu betrachten. Für den Beispielautomaten ergibt dies:

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | - | - | - |
| 2 | | - | - |
| 3 | | | - |

Im ersten Schritt werden alle Zustandspaare $(p, q)$, bei denen ein Zustand ein Endzustand ist und der andere nicht, als nicht äquivalent markiert. Warum diese Zustände nicht äquivalent sind, kann wie folgt gezeigt werden:

Sei (ohne Beschränkung) $p \in F$ und $q \notin F$. Angenommen, $p$ und $q$ wären äquivalent. Dann muss gelten:

$$(p, q) \in R_Z \Rightarrow \forall z \in \Sigma* : (\delta(p, z) \in F \Leftrightarrow \delta(q, z) \in F)$$

Ohne Zweifel ist $\varepsilon \in \Sigma*$.

Dann ergibt sich:

$$\delta(p, \varepsilon) = p \in F \Leftrightarrow \delta(q, \varepsilon) = q \in F$$

Und das steht im Widerspruch zur oben genannten Festlegung, dass $p \in F$ und $q \notin F$.

Die Beispielmatrix stellt sich dann wie folgt dar:

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | - | - | - |
| 2 | | - | - |
| 3 | x | x | - |

Im nächsten Schritt wird für alle unmarkierten Paare von Zuständen $(p, q)$ und jedes Zeichen $k \in \Sigma$ überprüft, ob $(\delta(p, k), \delta(q, k))$ markiert ist. Ist das der Fall, wird das Paar $(p, q)$ ebenfalls markiert, weil $p$ und $q$ dann auch nicht äquvalent sind.

**Beweis**

Ist $(\delta(p,k),\delta(q,k))$ markiert, gilt (ohne Beschränkung) $\delta(p,k) \in F$ und $\delta(q,k) \notin F$ (oder umgekehrt). Wären $p$ und $q$ äquivalent, würde das zu folgendem Widerspruch führen:

$$(p,q) \in R_Z \Leftrightarrow (\delta(p,k) \in F \Leftrightarrow \delta(q,k) \in F)$$

Im Grunde genommen wird in diesem Schritt überprüft, ob Zustandspaare, bei denen ein Zustand eine Kante von einem Endzustand „entfernt“ ist, *nicht* äquivalent sind (also $|z| = 1$). Wiederholt man diesen Schritt, gilt $|z| = 2$ usw. Daher muss dieser wiederholt werden, bis sich keine Änderungen mehr ergeben. Das Verfahren terminiert sicher, weil es zu jedem Zeitpunkt nur endlich viele unmarkierte Paare in der Matrix gibt und in jedem Durchgang mindestens eines markiert wird. *q.e.d.*

Für das Beispiel ergibt sich (einziges unmarkiertes Paar $(2,1)$):

$$\delta(2,a) = 2, \quad \delta(1,a) = 2, \quad \delta(2,b) = 3 \quad \text{und} \quad \delta(1,b) = 3$$

Die Paare $(2,2)$ und $(3,3)$ sind nicht markiert (Reflexivität von $R_Z$). Deshalb wird $(2,1)$ nicht markiert. Damit gibt es bereits im ersten Durchgang keine Änderung der Matrix und der Algorithmus terminiert. Die Zustände 1 und 2 sind also äquivalent und können zu einem Zustand verschmolzen werden.

Die Äquivalenzklassen sind:

| | |
|---|---|
| [1] | $\{1,2\}$ |
| [2] | $\{1,2\}$ |
| [3] | $\{3\}$ |

Für die Überführung der $\delta$-Funktion wird jeweils $\delta(z_i,k) = z_j \quad (k \in \Sigma, z_i \in Z)$ durch $\delta([z_i],k) = [z_j]$ ersetzt. Damit ergibt sich der in Abbildung A.7 gezeigte Automat.

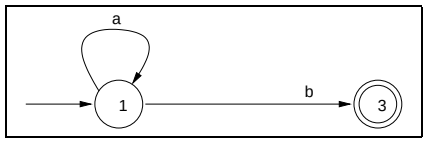



Abbildung A.7: Minimierter Automat für $a * b$

# B Grammatik der Spezifikationssprache

## B.1 Die Schlüsselworte der Spezifikationssprache

| | | | |
|---|---|---|---|
| ACTION | END | LET | RETURNS |
| ADT | EVENT | MEASURE | SELF |
| BEGIN | EXACT | META | SEQUENCE |
| BLOCK | EXTENDS | NEW | START |
| CLASSACTION | EXTERN | NIL | TERM |
| CLASSOBSERVER | FALSE | OBSERVER | THROW |
| CLASSPROPERTY | FUNC | ON | TRUE |
| CLASSRELATION | FUNCTION | PROPERTY | TYPEDEF |
| DO | IF | REJECT | |
| ELSE | IS | RELATION | |

## B.2 Vollständige Grammatik

| | | |
|---|---|---|
| Syntaxtree | ::= | |
| Syntaxtree | ::= | DeclarationSeq |
| DeclarationSeq | ::= | Declaration |
| DeclarationSeq | ::= | Declaration DeclarationSeq |
| Declaration | ::= | Metatype |
| Declaration | ::= | Function |
| Declaration | ::= | TypeDef |
| Declaration | ::= | AdtDef |
| TypeDef | ::= | **TYPEDEF** Id **;** |
| TypeDef | ::= | **TYPEDEF** Id **:** Type **;** |
| Metatype | ::= | **META** Id OptMetaParamSeq OptUsedMetaSeq MetaBody |
| MetaBody | ::= | BEG BodySeq **END ;** |
| MetaBody | ::= | BEG **END ;** |

| | | |
|---|---|---|
| MetaBody | ::= | **;** |
| BodySeq | ::= | BodyElem |
| BodySeq | ::= | BodyElem BodySeq |
| Requires | ::= | |
| Requires | ::= | [ **BLOCK** ] |
| Requires | ::= | [ **REJECT** ] |
| Requires | ::= | [ Expression ] |
| Requires | ::= | [ Expression **: BLOCK** ] |
| Requires | ::= | [ Expression **:** REJECT ] |
| ConnectEvent | ::= | |
| ConnectEvent | ::= | **−>** UsedEvent OptExpressionSeq **@** Expression |
| BodyElem | ::= | **EVENT** Id OptParameterSeq Requires ConnectEvent **;** |
| BodyElem | ::= | **START** Id OptParameterSeq Requires ConnectEvent **;** |
| BodyElem | ::= | **TERM** Id OptParameterSeq Requires ConnectEvent **;** |
| BodyElem | ::= | **SEQUENCE** Id **:** SubSequence OptEndSpec **;** |
| BodyElem | ::= | **SEQUENCE** Id **:** OptEndSpec **;** |
| BodyElem | ::= | **EXACT** Id **:** SubSequence OptEndSpec **;** |
| BodyElem | ::= | **EXACT** Id **:** OptEndSpec **;** |
| BodyElem | ::= | **PROPERTY** Id **RETURNS** Type EmptyExp Updates |
| BodyElem | ::= | **CLASSPROPERTY** Id **RETURNS** Type EmptyExp Updates |
| BodyElem | ::= | **RELATION** Id **RETURNS** UsedType EmptyExp Updates |
| BodyElem | ::= | **CLASSRELATION** Id **RETURNS** UsedType EmptyExp Updates |
| BodyElem | ::= | **ACTION** Id [ Expression ] **DO** Expression **;** |
| BodyElem | ::= | **OBSERVER** Id [ Expression ] **DO** Expression **;** |
| BodyElem | ::= | **CLASSACTION** Id [ Expression ] **DO** Expression **;** |
| Updates | ::= | **;** |
| Updates | ::= | **IS** UpdateSeq |
| UpdateSeq | ::= | Update |
| UpdateSeq | ::= | Update UpdateSeq |
| Update | ::= | **ON** UsedEvent **DO** Expression **;** |
| UsedEvent | ::= | Id |
| OptUsedMetaSeq | ::= | |
| OptUsedMetaSeq | ::= | **EXTENDS** UsedMetaSeq |
| UsedMetaSeq | ::= | UsedMeta |
| UsedMetaSeq | ::= | UsedMeta **,** UsedMetaSeq |
| UsedMeta | ::= | Id |

| | | |
|---|---|---|
| OptEndSpec | ::= | |
| OptEndSpec | ::= | { EndSpecList } |
| EndSpecList | ::= | EndSpec |
| EndSpecList | ::= | EndSpec \| EndSpecList |
| EndSpec | ::= | UsedEvent −> UsedPhase |
| SubSequence | ::= | Occurunce |
| SubSequence | ::= | Occurunce SubSequence |
| Occurunce | ::= | UsedEvent |
| Occurunce | ::= | ( SubSequence ) |
| Occurunce | ::= | Occurunce ∗ |
| Occurunce | ::= | Occurunce \| Occurunce |
| UsedPhase | ::= | Id |
| Function | ::= | Internal |
| Function | ::= | External |
| Function | ::= | Measure |
| Internal | ::= | **FUNCTION** Id ( ParameterSeq ) **RETURNS** Type **IS** Expression **;** |
| Internal | ::= | **FUNCTION** Id ( ) **RETURNS** Type **IS** Expression **;** |
| External | ::= | **EXTERN** Id ( TypeSeq ) **RETURNS** Type **;** |
| External | ::= | **EXTERN** Id ( ) **RETURNS** Type **;** |
| Measure | ::= | **MEASURE** Id ( TypeSeq ) **RETURNS** Type **;** |
| Measure | ::= | **MEASURE** Id ( ) **RETURNS** Type **;** |
| UsedType | ::= | Id |
| UsedType | ::= | **$** Id |
| OptType | ::= | |
| OptType | ::= | **:** UsedType |
| Type | ::= | [ ParameterSeq ] |
| Type | ::= | UsedType |
| Type | ::= | Id < TypeSeq > |
| TypeSeq | ::= | Type |
| TypeSeq | ::= | Type **,** TypeSeq |
| AdtDef | ::= | **ADT** Id < IdSeq > AdtBody |
| IdSeq | ::= | Id |
| IdSeq | ::= | Id **,** IdSeq |
| AdtBody | ::= | BEG AdtSeq **END ;** |
| AdtSeq | ::= | AdtFunc |
| AdtSeq | ::= | AdtFunc AdtSeq |
| AdtFunc | ::= | **FUNC** Id ( TypeSeq ) **RETURNS** Type **;** |
| AdtFunc | ::= | **FUNC** Id ( ) **RETURNS** Type **;** |
| Parameter | ::= | Id **:** Type |
| OptMetaParamSeq | ::= | |
| OptMetaParamSeq | ::= | [ ] |

| | | |
|---|---|---|
| OptMetaParamSeq | ::= | [ MetaParamSeq ] |
| MetaParamSeq | ::= | MetaParam |
| MetaParamSeq | ::= | MetaParam , MetaParamSeq |
| OptRename | ::= | |
| OptRename | ::= | [ Parameter ] |
| MetaParam | ::= | Parameter |
| MetaParam | ::= | Id −> Type OptRename |
| OptParameterSeq | ::= | |
| OptParameterSeq | ::= | ( ) |
| OptParameterSeq | ::= | ( ParameterSeq ) |
| ParameterSeq | ::= | Parameter |
| ParameterSeq | ::= | Parameter , ParameterSeq |
| EmptyExp | ::= | |
| EmptyExp | ::= | := Expression |
| ExpressionSeq | ::= | Expression |
| ExpressionSeq | ::= | Expression , ExpressionSeq |
| OptExpressionSeq | ::= | ( ) |
| OptExpressionSeq | ::= | ( ExpressionSeq ) |
| Expression | ::= | ( Expression ) |
| Expression | ::= | ^ Expression |
| Expression | ::= | Expression + |
| Expression | ::= | Expression − |
| Expression | ::= | Constant |
| Expression | ::= | IdentNt |
| Expression | ::= | Expression . IdentNt |
| Expression | ::= | BoolExp |
| Expression | ::= | PrefixExp |
| Expression | ::= | UsedId OptExpressionSeq |
| Expression | ::= | Id [ TypeSeq ] |
| Expression | ::= | ConditionSeq **ELSE** Expression |
| Expression | ::= | **LET** Id OptType := Expression : Expression |
| Expression | ::= | **THROW** UsedEvent OptExpressionSeq @ Expression |
| Expression | ::= | **THROW** UsedEvent OptExpressionSeq @ Expression : Expression |
| Expression | ::= | **NEW** [ Expression ] OptExpressionSeq |
| Expression | ::= | Expression −> Type |
| ConditionSeq | ::= | Condition |
| ConditionSeq | ::= | Condition ConditionSeq |
| Condition | ::= | **IF** Expression : Expression |
| Constant | ::= | INT |
| Constant | ::= | **TRUE** |
| Constant | ::= | **FALSE** |

| | | |
|---|---|---|
| Constant | ::= | STRING |
| Constant | ::= | **NIL** |
| Constant | ::= | **SELF** |
| BoolExp | ::= | Expression & Expression |
| BoolExp | ::= | Expression \| Expression |
| BoolExp | ::= | Expression > Expression |
| BoolExp | ::= | Expression < Expression |
| BoolExp | ::= | Expression >= Expression |
| BoolExp | ::= | Expression <= Expression |
| BoolExp | ::= | Expression = Expression |
| BoolExp | ::= | Expression # Expression |
| PrefixExp | ::= | ! Expression |
| UsedId | ::= | IdentNt |
| Id | ::= | IdentNt |

# C Hinweise zur Verwendung von PROMETHEUS

## C.1 Schnittstelle zu C

Der PROMETHEUS-Generator erzeugt aus der Spezifikation den entsprechenden Managementcode als C-Code. Die Datentypen werden direkt auf die entsprechenden C-Typen abgebildet, wobei für die Komponenten und Komponentenklassen ein C-Typ (`unique_t`) verwendet wird. Über externe Funktionen und externe Datentypen ist eine Schnittstelle zu C gegeben. Für die externen Funktionen wird ein entsprechender Funktionskopf erzeugt.

Für jeden externen Datentypen (`typ`) muss neben dem Typen selbst auch ein Wert für das nicht definierte Element (`_no_typ_`) angegeben werden.

Die Funktionen der abstrakten Datentypen müssen ebenfalls extern definiert werden. Auch für diese werden entsprechende Funktionsköpfe erzeugt (evtl. als seperate Datei).

Es besteht die Möglichkeit C-Code direkt in die Spezifikation aufzunehmen, indem der Code mit `%{` und `%}` geklammert wird. Eine Ergänzung des erzeugten Headers ist mit `%[` und `%]` möglich.

## C.2 Starten eines Systems

In jedem System wird implizit eine Meta-Klasse `node` erzeugt, falls keine Meta-Klasse diesen Namens existiert. Der Meta-Klasse `node` werden automatisch – gleichgültig ob implizit oder explizit spezifiziert – eine Eigenschaft `master` und eine Eigenschaft `clients`, sowie die Ereignisse `set_master` und `set_client` hinzugefügt. Bei der Initialisierung des Systems werden diese Werte automatisch gesetzt. Der Master (der Knoten, auf dem das System gestartet wurde) kennt alle Clients; jeder Client kennt den Master.

Der Start des Systems geschieht durch den Aufruf der Funktion `boot`, die als Parameter die Anzahl der Argumente und ein NULL-terminiertes Feld der Argumente erhält (analog zur Funktion `main` in C). Die Funktion liefert für den Master 1 und für einen Client 0 zurück und sollte stets als erste Funktion in `main` aufgerufen werden.

## C.3 Aufruf von PROMETHEUS

### Aufruf

`prometheus` [ options ] <spezifikation>

### Optionen

**-h** Hilfstext anzeigen.

**-I** <**Pfad**> Pfad für Input-Dateien

**-e** Auswertung aller Eigenschaften erzwingen.

**-l** Verzögerte Auswertung der Eigenschaften.

**-a** Separate Header für die abstrakten Datentypen.

**-o** <**name**> Name der Ausgabedateien.

**-m** Erzeugung eines Makefiles für die Übersetzung des Projektes

**-d** Erzeugen einer LaTeX Beschreibung des Projektes, sowie eines Abhängigkeitsgraphen (als Eingabe für VCG).

# D Eine Laufzeitumgebung für CDSL

## D.1 PROMETHEUS Spezifikation

### D.1.1 Typen

**TYPE** `Type` → `string`
**TYPE** `Float` → `int`
**TYPE** `fct`
**TYPE** `P_VOID`
**TYPE** `LOGFILE`

### D.1.2 Abstrakte Datatypen

**ADT** list⟨t⟩
  **FCT** *appfront* : (`t` × list⟨`t`⟩) → list⟨`t`⟩
  **FCT** *appback* : (list⟨`t`⟩ × `t`) → list⟨`t`⟩
  **FCT** *conc* : (list⟨`t`⟩ × list⟨`t`⟩) → list⟨`t`⟩
  **FCT** *front* : (list⟨`t`⟩) → `t`
  **FCT** *rest* : (list⟨`t`⟩) → list⟨`t`⟩
  **FCT** *childs* : (list⟨`t`⟩) → `int`
  **FCT** *isempty* : (list⟨`t`⟩) → `boolean`
  **FCT** *first* : (list⟨`t`⟩) → `t`
  **FCT** *next* : (list⟨`t`⟩) → `t`
  **FCT** *elems* : (list⟨`t`⟩) → `int`
  **FCT** *iselem* : (list⟨`t`⟩ × `t`) → `boolean`
**END** list
**ADT** elemCount⟨t⟩
  **FCT** *increment* : (elemCount⟨`t`⟩ × `t`) → elemCount⟨`t`⟩
  **FCT** *getMax* : (elemCount⟨`t`⟩) → `t`
  **FCT** *getMaxVal* : (elemCount⟨`t`⟩) → `int`
  **FCT** *getSecondMax* : (elemCount⟨`t`⟩) → `t`
  **FCT** *getSecondMaxVal* : (elemCount⟨`t`⟩) → `int`
  **FCT** *getThirdMax* : (elemCount⟨`t`⟩) → `t`
  **FCT** *getThirdMaxVal* : (elemCount⟨`t`⟩) → `int`
  **FCT** *getCount* : (elemCount⟨`t`⟩ × `t`) → `int`
**END** elemCount

### D.1.3 Funktionen

**FCT** *extractClus* : (`string`) → `int`

**FCT** *add* : (`int` × `int`) → `int`

**FCT** *sub* : (`int` × `int`) → `int`

**FCT** *incr* : (`int`) → `int`

**FCT** *decr* : (`int`) → `int`

**FCT** *equals* : (`string` × `string`) → `boolean`

**FCT** *createthread* : (`fct` × `P_VOID` × `P_VOID`) → `int`

**FCT** *exec* : (`fct` × `ActiveDef`) → `boolean`

**FCT** *joinThread* : (`int`) → `boolean`

**FCT** *terminateSystem* : → `boolean`

**FCT** *averageCommValues* : (elemCount⟨`ROOT`⟩ × elemCount⟨`ROOT`⟩ × `string` × `string` × `string`) → elemCount⟨`ROOT`⟩

**FCT** *setClusCl* : (`ROOT`) → `Cluster`

**FCT** *getClusCl* : → `Cluster`

**FCT** *checkMigration* : (elemCount⟨`Cluster`⟩ × elemCount⟨`Cluster`⟩ × `Cluster` × `int` × `Float`) → `Cluster`

**FCT** *calculateWeight* : (`int` × `int` × `Float`) → `int`

**FCT** *compWeight* : (`ActiveDef`) → `int`

**FCT** *initCluster* : (clNum : `int`, cl : list⟨`Cluster`⟩, c : `Cluster`) → list⟨`Cluster`⟩ :
  **IF** (clNum≤0) : cl
  **ELSE**
    **LET** H2← **NEW** c [clNum] :
      **LET** H3← *decr*(clNum) :
        *initCluster*(H3, *appfront*(H2, cl), c)
**END** *initCluster*

**FCT** *mapNodeCluster* : (cl_li : list⟨`Cluster`⟩, nodes : rel⟨`node`⟩, fi : `node`) → `boolean` :
  *setCluster* (*front*(cl_li))@*current*(nodes) :
    **IF** (*next*(nodes)=fi) : *true*
    **ELSE**
      *mapNodeCluster*(cl_li, nodes, fi)
**END** *mapNodeCluster*

**FCT** *setClusterComp* : (clusId : `int`, l : rel⟨`Cluster`⟩, n : `int`) → `Cluster` :
  **IF** (n≤0) : ⊥
  **IF** (*first*(l)⊢Id=clusId) :
    *first*(l)
  **ELSE**
    *setClusterComp*(clusId, *remove*(l, *first*(l)), *decr*(n))
**END** *setClusterComp*

**FCT** *isEntryCall* : (entries : list⟨`MethDef`⟩, what : `MethDef`) → `boolean` :
  *iselem*(entries, what)
**END** *isEntryCall*

**FCT** *agingValues* : (classHash : elemCount⟨`ROOT`⟩, compHash : elemCount⟨`ROOT`⟩) → elemCount⟨`ROOT`⟩
:
  classHash
**END** *agingValues*

## D.1.4 Metatypen

**META** `Project`[id : `string`, compList : list⟨`ClassTypeDef`⟩, clusNum : `int`] ← LifeTime-
Def

  **EVT** *init* (sysThread : `P_VOID`) [**BLOCK**]
  **EVT** *done* [**BLOCK**]
  **EVT** *startSystem* (sysComp : `ActiveDef`) [**BLOCK**]

  **SEQ** *all* :
    *init*
    *startSystem*
    *done*

  **PROP** Id → `string` := id
  **PROP** compClassList → list⟨`ClassTypeDef`⟩ := compList
  **PROP** clCluster → `Cluster` :=
      **LET** H← *Cluster*(clusNum, compClassList) :
        **LET** H2← *setClusCl*(H) :
          H
  **PROP** systemComp → `ActiveDef` := ⊥
    *startSystem* : sysComp
  **PROP** systemThread → `P_VOID` := ⊥
    *init* : sysThread
  **PROP** clusterList → list⟨`Cluster`⟩ := ⊥
    *init* : **LET** H← *initCluster*(clusNum, *rest*(⊥), clCluster) :
      H

  **ACT** *AC_NodesToCluster*[(state=init)]
    **LET** H3← *getnode*()⊢master⊢clients :
      **LET** H4← *mapNodeCluster*(clusterList, H3, *first*(H3)) :
        *true*
  **ACT** *AC_startSystemThread*[(((state=startSystem)∧(systemThread≠⊥))∧(systemComp≠⊥))]
    **LET** H2← **NEW** *Thread*() [systemComp, systemThread] :
      *true*

**END** `Project`

**META** `ClassTypeDef` ← AttribDef, StatementDef

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]
**EVT** *commStoEvt* (s : `StorageDef`) [**BLOCK**]
**EVT** *partnerComm* (s : `StorageDef`) [**BLOCK**]

**END** `ClassTypeDef`

**META** `ActiveDef`[clname : `string`, id : `int`, main : `fct`, mlist : list⟨`MethDef`⟩] ← ClassTypeDef, DeclarationDef, LifeTimeDef

**EVT** *init* (coName : `string`, obj : `P_VOID`, fath : `DeclarationDef`, ltFath : `LifeTimeDef`) [**BLOCK**]
⇒ *placeMe* ()@*self*
**EVT** *setNode* (n : `node`) [**BLOCK**]
**EVT** *initCl* (cl : `Cluster`) [**BLOCK**]
**EVT** *terminated*
[(*count*(ltdeps$_S$)=0) |**BLOCK** ]
⇒ *done* ()@*self*
**EVT** *termStorages* [**BLOCK**]
**EVT** *startSon* [**BLOCK**]
**EVT** *placeMe* [**BLOCK**]
**EVT** *done* [**BLOCK**]
**EVT** *call* (act : `ActiveDef`, what : `MethDef`) [**BLOCK**]
⇒ *called* (what, *self*)@act
**EVT** *wait_accept* [**BLOCK**]
**EVT** *return* [**BLOCK**]
**EVT** *accept* (what : list⟨`MethDef`⟩) [**BLOCK**]
⇒ *accepted* ()@*self*
**EVT** *called* (what : `MethDef`, by : `ActiveDef`)
[*isEntryCall*(accEntry, what) |**BLOCK** ]
**EVT** *accepted* [**BLOCK**]
⇒ *wait_accept* ()@caller
**EVT** *commStoEvt* (s : `StorageDef`) [**BLOCK**]
**EVT** *syncClusEvt* (p : `Cluster`) [**BLOCK**]
**EVT** *commClusEvt* (p : `Cluster`) [**BLOCK**]

**SEQ** *work* :
*init*
*initCl*
(
*startSon*
*evt_ltdeps_in*
|
*call*
*wait_accept*
*return*
|
*accept*
*called*
*accepted*
)*

*terminated*
*done*

**PROP** Id → `int` := id
**PROP** className → `string` := clname
**REL** actCluster → `Cluster` := ⊥
*init* : **IF** (↗ [actCluster]=⊥) :
*setClusterComp*(clusId, clCluster⊢components$_{\mathcal{S}}$,
*count*(clCluster⊢components$_{\mathcal{S}}$))
**ELSE** ↗ [actCluster]
*done* : *checkMigration*(commClus, syncAct, actCluster, weight, $\omega$)
**PROP** $\omega$ → `int` := 70
**PROP** weight → `Float` := 0
*done* : *compWeight(self)*
**PROP** clusId → `int` :=
**IF** (className≠⊥) :
*extractClus*(className)
**ELSE** ⊥
**PROP** methList → list⟨`MethDef`⟩ := mlist
**PROP** clCluster → `Cluster` :=
*getClusCl*()
**PROP** syncAct → elemCount⟨`ActiveDef`⟩ := ⊥
*done* : *averageCommValues*(↗ [syncAct], ↗ [syncActComp], ↗ [className], "`syncAct`", ↗ [compName])
**PROP** commSto → elemCount⟨`StorageDef`⟩ := ⊥
*done* : *averageCommValues*(↗ [commSto], ↗ [commStoComp], ↗ [className], "`commSto`", ↗ [compName])
**PROP** commClus → elemCount⟨`Cluster`⟩ := ⊥
*done* : *averageCommValues*(↗ [commClus], ↗ [commClusComp], ↗ [className], "`commClus`", ↗ [compName])
**PROP** syncClus → elemCount⟨`Cluster`⟩ := ⊥
*done* : *averageCommValues*(↗ [syncClus], ↗ [syncClusComp], ↗ [className], "`syncClus`", ↗ [compName])
**REL** execute → `node` := ⊥
*setNode* : n
*terminated* : *done* ()@*first*(active$_{\mathcal{S}}$) : ⊥
**PROP** accEntry → list⟨`MethDef`⟩ := ⊥
*accept* : what
*called* : ⊥
**PROP** calledEntry → `MethDef` := ⊥
*called* : what
*accepted* : ⊥
**PROP** caller → `ActiveDef` := ⊥
*called* : by
*accepted* : ⊥
**PROP** callee → `ActiveDef` := ⊥
*call* : act
*return* : ⊥

**PROP** compName → `string` := ⊥
  *init* : coName
  *terminated* : ↗ [compName]
**REL** components → `ActiveDef` := ⊥
  *init* : CLASS
**PROP** sysObj → `P_VOID` := ⊥
  *init* : obj
**REL** ltdeps → `LifeTimeDef` := ⊥
  *init* : ltFath
  *terminated* : ⊥
**REL** active → `DeclarationDef` := ⊥
  *init* : fath
**PROP** syncActComp → elemCount⟨`ActiveDef`⟩ := ⊥
  *init* : ⊥
  *call* : **IF** (↗ [act]⊢actCluster⊢Id=actCluster⊢Id) :
    **LET** H← *increment*(↗ [syncActComp], act) :
      H
    **ELSE**
      *syncClusEvt* (act⊢actCluster)@*self* : ↗ [syncActComp]
  *called* : **IF** (↗ [by]⊢actCluster⊢Id=actCluster⊢Id) :
    **LET** H← *increment*(↗ [syncActComp], by) :
      H
    **ELSE**
      *syncClusEvt* (by⊢actCluster)@*self* : ↗ [syncActComp]
**PROP** commStoComp → elemCount⟨`StorageDef`⟩ := ⊥
  *init* : ⊥
  *commStoEvt* : **IF** (↗ [s]⊢stoCluster⊢Id=actCluster⊢Id) :
    **LET** H← *increment*(↗ [commStoComp], s) :
      H
    **ELSE**
      *commClusEvt* (s⊢stoCluster)@*self* : commStoComp
**PROP** commClusComp → elemCount⟨`Cluster`⟩ := ⊥
  *init* : ⊥
  *commClusEvt* : *increment*(↗ [commClusComp], p)
**PROP** syncClusComp → elemCount⟨`Cluster`⟩ := ⊥
  *init* : ⊥
  *syncClusEvt* : *increment*(↗ [syncClusComp], p)

**ACT** *AC_placeActivity*[((state=placeMe)∧(actCluster≠⊥))]
  **IF** *equals*(compName, "`system`") : *true*
  **ELSE**
    *placeActivity* (*self*)@actCluster

**END** `ActiveDef`

**META** `StorageDef`[clName : `string`, id : `int`, mlist : list⟨`MethDef`⟩] ← ClassTypeDef, DeclarationDef, LifeTimeDef

**EVT** *init* (coName : `string`, fath : `DeclarationDef`, ltFath : `LifeTimeDef`) [**BLOCK**]
**EVT** *done* [**BLOCK**]
**EVT** *gotThread* (th : `Thread`) [**BLOCK**]
**EVT** *newAct* (a : `ActiveDef`) [**BLOCK**]
**EVT** *newSto* (s : `StorageDef`) [**BLOCK**]

**EVT** *initCl* (cl : `Cluster`) [**BLOCK**]
**EVT** *gotClClus* [**BLOCK**]
**EVT** *ready* [**BLOCK**]
**EVT** *clear* [**BLOCK**]
**EVT** *gotThreadId* [**BLOCK**]
**EVT** *referenced* [**BLOCK**]
**EVT** *setNode* (n : `node`) [**BLOCK**]
**EVT** *comm* (p : `ClassTypeDef`) [**BLOCK**]
**EVT** *partnerComm* (s : `StorageDef`) [**BLOCK**]
**EVT** *commClusEvt* (p : `Cluster`) [**BLOCK**]

**PROP** Id → `int` := id
**PROP** className → `string` := clName
**PROP** methList → list⟨`MethDef`⟩ := mlist
**REL** stoCluster → `Cluster` := ⊥
*init* : **IF** (↗ [stoCluster]=⊥) :
*setClusterComp*(clusId, clCluster⊢components$_{\mathcal{S}}$,
*count*(clCluster⊢components$_{\mathcal{S}}$))
**ELSE** ↗ [stoCluster]
**PROP** clusId → `int` :=
**IF** (className≠⊥) :
*extractClus*(↗ [className])
**ELSE** ⊥
**PROP** commClass → elemCount⟨`ClassTypeDef`⟩ := ⊥
*done* : *averageCommValues*(↗ [commClass], ↗ [commComp], ↗ [className], "`commAct`", ↗ [compName])
**PROP** commClus → elemCount⟨`Cluster`⟩ := ⊥
*done* : *averageCommValues*(↗ [commClus], ↗ [commClusComp], ↗ [className], "`commClus`", ↗ [compName])
**PROP** compName → `string` := ⊥
*init* : coName
**REL** components → `StorageDef` := ⊥
*init* : CLASS
**REL** passive → `DeclarationDef` := ⊥
*init* : fath
**PROP** ltdeps → `LifeTimeDef` := ⊥
*init* : ltFath
**PROP** commComp → elemCount⟨`ClassTypeDef`⟩ := ⊥
*init* : ⊥

*comm* : **IF** p ≻ `ActiveDef` :
**LET** pVar← p :
*commStoEvt* (*self*)@p :
**IF** (p⊢clusId=clusId) :
*commStoEvt* (*self*)@p :
*increment*(↗ [commComp], p)
**ELSE**
*commClusEvt* (p⊢actCluster)@*self* : ↗ [commComp]
**ELSE**
**IF** p ≻ `StorageDef` :
**LET** pVar← p :
*partnerComm* (*self*)@p :
**IF** (p⊢clusId=clusId) :
*increment*(↗ [commComp], p)
**ELSE**
*commClusEvt* (p⊢stoCluster)@*self* : ↗ [commComp]
**ELSE** ↗ [commComp]

*partnerComm* : *increment*(↗ [commComp], s)

**PROP** commClusComp → elemCount⟨`Cluster`⟩ := ⊥

*init* : ⊥

*commClusEvt* : *increment*(↗ [commClusComp], p)

**END** `StorageDef`

**META** `ObjDef`

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**END** `ObjDef`

**META** `MethDef`[id : `string`, callP : list⟨`ObjDef`⟩, retP : list⟨`ObjDef`⟩, bl : `BlockDef`] ← LifeTimeDef

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**PROP** ident → `string` := id
**PROP** callParams → list⟨`ObjDef`⟩ := callP
**PROP** retParams → list⟨`ObjDef`⟩ := retP
**PROP** methblock → `BlockDef` := bl

**END** `MethDef`

**META** `AcceptDef`[methlist : list⟨`MethDef`⟩] ← StatementDef

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**PROP** acceptedMethList → list⟨`MethDef`⟩ := methlist

**END** `AcceptDef`

**META** `StatementDef`

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**END** `StatementDef`

**META** `BlockDef`[id : `string`, stmts : list⟨`StatementDef`⟩] ← LifeTimeDef, DeclarationDef

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]
**EVT** *terminated*
[(*count*(ltdeps$_{\mathcal{S}}$)=0) |**BLOCK** ]
**EVT** *gotThread* (th : `Thread`) [**BLOCK**]
**EVT** *destroy* [**BLOCK**]

**PROP** ident → `string` := id
**PROP** statements → list⟨`StatementDef`⟩ := stmts
**PROP** phase → `int` := ⊥
*init* : 1
*terminated* : 2
*destroy* : 3
**PROP** actChildrenReady → `int` := ⊥
*init* : 0

**END** `BlockDef`

**META** `ExprDef` ← Statement

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**END** `ExprDef`

**META** `BoolExprDef` ← ExprDef

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**END** `BoolExprDef`

**META** `AssignDef` ← ExprDef

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**END** `AssignDef`

**META** `CopyDef` ← AssignDef

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**PROP** lValue → `ObjDef` := ⊥

**END** `CopyDef`

**META** `MoveDef` ← AssignDef

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**PROP** lValue → `ObjDef` := ⊥

**END** `MoveDef`

**META** `LinkDef` ← AssignDef

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**PROP** lValue → `ObjDef` := ⊥
**PROP** source → `ClassTypeDef` := ⊥

**ACT** *AC_incrRefCount*[(source≠⊥)]
**IF** source ≻ `StorageDef` :
*referenced* ()@source
**ELSE** *true*

**END** `LinkDef`

**META** `ProgFlowDef` ← Statement

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**END** `ProgFlowDef`

**META** `IfDef`[ifs : list⟨`IfDefElem`⟩] ← ProgFlowDef

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**PROP** ifelems → list⟨`IfDefElem`⟩ := ifs

**END** `IfDef`

**META** `IfDefElem`[b : `BoolExprDef`, bl : `BlockDef`] ← ProgFlowDef

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**PROP** boolExpr → `BoolExprDef` := b
**PROP** body → `BlockDef` := bl

**END** `IfDefElem`

**META** `LoopDef` ← ProgFlowDef

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**END** `LoopDef`

**META** `WhileDef`[boolExpr : `BoolExprDef`, bl : `BlockDef`] ← LoopDef

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**PROP** condition → `BoolExprDef` := boolExpr
**PROP** body → `BlockDef` := bl

**END** `WhileDef`

**META** `ForDef`[boolExpr : `BoolExprDef`, bl : `BlockDef`] ← LoopDef

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**PROP** condition → `BoolExprDef` := boolExpr
**PROP** body → `BlockDef` := bl

**END** `ForDef`

**META** `LifeTimeDef`

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**END** `LifeTimeDef`

**META** `DeclarationDef`

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**END** `DeclarationDef`

**META** `Bool`

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**END** `Bool`

**META** `Int`

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**END** `Int`

**META** `node`

**EVT** *init* [**BLOCK**]
**EVT** *setJoin* (jt : `Thread`) [**BLOCK**]
**EVT** *join*
[(toJoin≠⊥) |**BLOCK** ]
**EVT** *joined* [**BLOCK**]
**EVT** *placeSto* (sto : `StorageDef`) [**BLOCK**]
**EVT** *setCluster* (clus : `Cluster`) [**BLOCK**]
**EVT** *done*
[(*count*(execute$_{\mathcal{S}}$)=0) |**BLOCK** ]

```
  SEQ all :
    init
    setCluster
    (
      evt_execute_in | evt_execute_out |
        setJoin
        joined
    )*
    done

  PROP toJoin → Thread := ⊥
    setJoin :              jt
    joined :               ⊥
  PROP newThread → Thread := ⊥
    evt_execute_in :       IF ¬(equals(comp⊢className, "system")) :
                             NEW Thread() [comp, ⊥]
                           ELSE ⊥
  REL nodelist → Cluster := ⊥
    setCluster :           clus
  PROP cpuLoad → Float := ⊥
    init :                 0
  PROP memoryLoad → int := ⊥
    init :                 0
  PROP maxMem → int := ⊥
    init :                 0
  PROP cpuPerformance → int := ⊥
    init :                 0
  PROP stoCompList → list⟨ObjDef⟩ := ⊥
    placeSto :             appback(stoCompList, sto)

  ACT AC_joinThread[(toJoin≠⊥)]
    LET H← joinThread(toJoin⊢pthread) :
      destroy ()@toJoin :
        joined ()@self

END node

META Thread

  EVT init (act : ActiveDef, thr : P_VOID) [BLOCK]
  EVT done [BLOCK]
  EVT destroy [BLOCK]

  SEQ all :
    init
    done
    destroy

  PROP pthread → int := ⊥
    init :                 LET P1← createthread(act⊢main, act⊢sysObj, thr) :
                             P1
```

**REL** activity → `ActiveDef` := ⊥
*init* : act
*destroy* : ⊥
**PROP** usedby → `ActiveDef` := ⊥
*init* : act

**ACT** *AC_join*[(state=done)]
*setJoin* (*self*)@*getnode*()
**ACT** *AC_error*[(pthread=0)]
*terminateSystem*()

**END** `Thread`

**META** `Cluster`[num : `int`, Typedefs → list⟨`ClassTypeDef`⟩]

**EVT** *init* (id : `int`) [**BLOCK**]
**EVT** *done* [**BLOCK**]
**EVT** *placeActivity* (act : `ActiveDef`) [**BLOCK**]
**EVT** *placeStorage* (sto : `StorageDef`) [**BLOCK**]

**SEQ** *pre* :
*init*
(
*placeActivity* | *placeStorage*
)*
*done*

**REL** components → `Cluster` := ⊥
*init* : **LET** H← CLASS :
H
**PROP** Id → `int` := ⊥
*init* : id
**PROP** nodeId → `int` := ⊥
**PROP** newActivity → `ActiveDef` := ⊥
*init* : ⊥
*placeActivity* : act
**PROP** newStorage → `StorageDef` := ⊥
*init* : ⊥
*placeStorage* : sto
**PROP** averActNum → `int` := ⊥
**PROP** averStoNum → `int` := ⊥
**PROP** averMemUse → `int` := ⊥
**PROP** syncRate → `Float` := ⊥

**ACT** *AC_placeAct*[(state=placeActivity)]
**LET** H← *first*(nodelist$_{\mathcal{S}}$) :
**IF** (*count*(nodelist$_{\mathcal{S}}$)>0) :
*setNode* (H)@newActivity
**ELSE** *true*

**ACT** *AC_placeSto*[(state=placeStorage)]
**LET** H← *first*(nodelist$_{\mathcal{S}}$) :
**IF** (*count*(nodelist$_{\mathcal{S}}$)>0) :
*setNode* (H)@newStorage
**ELSE** *true*

**END** `Cluster`

**META** `CommManager`[avClList : list⟨`ClusterComm`⟩, avCoList : list⟨`ComponentComm`⟩]

**EVT** *init* (clList : list⟨`ClusterComm`⟩, coList : list⟨`ComponentComm`⟩, father : `ClassTypeDef`) [**BLOCK**]
**EVT** *done* [**BLOCK**]

**PROP** averClusList → list⟨`ClusterComm`⟩ := avClList
**PROP** averCommList → list⟨`ComponentComm`⟩ := avClList
**PROP** clusCompList → list⟨`ClusterComm`⟩ := ⊥
*init* : clList
**PROP** fatherObj → `ClassTypeDef` := ⊥
*init* : father
**PROP** commCompList → list⟨`ClusterComm`⟩ := ⊥
*init* : coList

**END** `CommManager`

**META** `ComponentComm`[avCommVal : `Float`]

**EVT** *init* (partner : `ClassTypeDef`, cVal : `int`) [**BLOCK**]
**EVT** *done* [**BLOCK**]

**PROP** averCommValue → `Float` := avCommVal
**PROP** CommPartner → `ClassTypeDef` := ⊥
*init* : partner
**PROP** commValue → `Float` := ⊥
*init* : cVal

**END** `ComponentComm`

**META** `ClusterComm`[clusid : `int`, avcVal : `Float`]

**EVT** *init* (cVal : `Float`) [**BLOCK**]
**EVT** *done* [**BLOCK**]

**PROP** clusterId → `int` := clusid
**PROP** averCommValue → `Float` := avcVal
**PROP** commValue → `Float` := ⊥
*init* : cVal

**END** `ClusterComm`

**META** `SyncElem`

**EVT** *init* [**BLOCK**]
**EVT** *done* [**BLOCK**]

**END** `SyncElem`

## D.2 Implementierung ausgewählter externer Funktionen

```
/*******************************************************
 *** Erzeugung der Komponenten und deren Klassen ***
 *** Ort, um Testfaelle zu integrieren           ***
 *******************************************************/

int createthread(fct  f , P_VOID obj, P_VOID thr)
{

  if (thr == NULL)
    {
      pthread_t thread;
      if (pthread_create(&thread, NULL, f, obj) == 0)
        {
          return thread;
        }
      else
        {
          fprintf(stderr , "Bei Thread-Erzeugung ist ein Fehler aufgetreten !!! \n");
          fprintf(stderr , "this-Objekt hat den Zeiger-Wert : %p\n", obj);
          fprintf(stderr , "Funktions-Zeiger hat den Wert : %p\n", f);
          return 1;
        }
    }
  /* system thread has own thread pointer from CDSL_System */
  else
    {
      pthread_t* thread = (pthread_t*) thr;
      if (pthread_create(thread, NULL, f, obj) == 0)
        {
          return *thread;
        }
      else
        {
          fprintf(stderr , "Bei Thread-Erzeugung ist ein Fehler aufgetreten !!! \n");
          fprintf(stderr , "this-Objekt hat den Zeiger-Wert : %p\n", obj);
          fprintf(stderr , "Funktions-Zeiger hat den Wert : %p\n", f);
          return 1;
        }
    }
}

int terminateSystem()
{
  exit (0);
}
```

```
int extractClus(char* clName)
{
  int MAX_CLUS = 3;              /* Anzahl Cluster */
  int clusId = 1;                /* DEFAULT IS 1 */
  const char delim = '_';
  char* tmp = strdup(clName);
  char* end = strrchr(tmp, delim);

  if  (end != NULL)
    {
      end++;
      if ( (end != NULL) && (strlen(end) == 1) )
        {
          clusId = atoi(end);
          if (1 <= clusId <= MAX_CLUS)
            {
              return clusId;
            }
        }
    }

  return clusId;
}

/*******************************************************
 *** Load Balancing                                  ***
 *******************************************************/

void updateValues(void *data, void **classH)
{
  elCount_data *d = NULL;

  if ((data != NULL) && (classH != NULL))
    {
      unique_t *pComp = NULL; /* aktuelle Komp. mit der komm. wurde */
      unique_t *pCla = NULL; /* Klassenelement der aktuellen Komponente */
      unique_t cla;
      HASHTABLE *clH = (HASHTABLE*) (*classH); /* hashtable der
                                                  Klassenkomponenten (hier wird eingefuegt) */
      int coVal; /* Zaehlerwert in Komponenten-Hashtable */
      ENTRY *pEntry; /* Eintrag in der Komponenten-Hashtable */


      d = (elCount_data*)data;
      pComp = d->root;

      cla = _get_ROOT_CLASS(*pComp);
      pCla = &cla;

      coVal = (*d->pCo);
```

```
        /* ist die Klasse der Komponente bereits in Klassen-Hash registriert */
        pEntry = hsearch(clH, (void*)pCla);

        /* komp bereits in Hashtable der Klasse, Aging anwenden */
        if (pEntry != NULL)
          {
            int *pClVal;          /* Zeiger auf aktuellen Zaehlerwert der
                                     (!) Klasse */
            int tmp;

            pClVal = (int *)( ((elCount_data*)pEntry->data)->pCo );
            fprintf(stderr, "Alter Wert aus dem KlassenHash <%p>: <%d>.\n", clH, (*pClVal));

            /*** AGING, Herz des ganzen ***/
            tmp = ( ((*pClVal) * 0.5) + (coVal * 0.5) );
            (*pClVal) = tmp;

          }

        /* Komponentenklasse neu Einfuegen, kein Aging noetig */
        else
          {
            ENTRY newEntry; /* darf lokal sein, wird nochmal von Hash angelegt */
            ENTRY *dummy;
            elCount_data *newECD = NULL; /* Zeiger auf neuen Wrapper fuer
                                            Zaehlerwert */

            newECD = (elCount_data*) malloc(sizeof(elCount_data));
            newECD->pCo = (int*) malloc(sizeof(int));
            (*newECD->pCo) = coVal;

            /* Neuen unique_t anlegen */
            newECD->root = (unique_t*) malloc(sizeof(unique_t));
            (newECD->root)->ip = pCla->ip;
            (newECD->root)->count = pCla->count;

            newEntry.key = (void*) newECD->root;
            newEntry.data = (void*) newECD;

            hinsert(clH, newEntry);
            dummy = hsearch(clH, newECD->root);
            if (dummy == NULL)
              {
                fprintf(stderr, "Fehler beim Insert in KlassenHash <%p>\n", clH);
              }

          }
      }
}
```

```
/* fuege alle Klassen statt den Komponenten in neue Hash, weil wir beim Aging
   auf Klassen arbeiten */
void toClass(void *data, void **classHash)
{
  if ((data != NULL) && (classHash != NULL))
    {
      elCount_data *actECD = (elCount_data*) data;
      HASHTABLE *clH = (HASHTABLE*) (*classHash); /* hashtable der
                                                     Klasse (hier wird eingefuegt) */

      if (actECD != NULL)
        {
          elCount_data *newECD = (elCount_data*) malloc(sizeof(elCount_data));
          unique_t *pComp = actECD->root; /* aktuelle Komp. mit der
                                             komm. wurde */
          int *pActVal = (actECD->pCo);
          ENTRY entry;

          newECD->pCo = pActVal;
          newECD->root = pComp;

          entry.key = (void*) pComp;
          entry.data = (void*) newECD;

          hinsert(clH, entry);
        }
    }
}


/* return cluster component with maximum weight */
unique_t checkMigration(elemCount_p_t commClus, elemCount_p_t syncClus,
                        unique_t actClus, int actVal, int omega)
{

  unique_t *res = NULL;
  ENTRY *syncValRes = NULL; /* Suche nach SyncVal fuer einen der drei Maxwerte
                               der Kommunikation */
  /* Sync-Werte fuer maximum Cluster */
  int maxVal = 0;
  int secMaxVal = 0;
  int thiMaxVal = 0;
  int m = 0; /* PLatzhalter fuer groessten der drei */
  ENTRY *maxE = NULL;

  _elemCount_t *commEl = (_elemCount_t*)(commClus.data);
  _elemCount_t *syncEl = (_elemCount_t*)(syncClus.data);
  /* Komm-Werte fuer maximum Cluster */
  int maxCommVal = (int)((commEl->max)->data);
  int secMaxCommVal = (int)((commEl->max)->data);
  int thiMaxCommVal = (int)((commEl->max)->data);
```

```
/* Berechnung fuer maximum Cluster */

/* gibts einen SyncWert fuer Cluster mit maximalem Kommunikationswert ? */
syncValRes = hsearch( syncEl->h, (void*)((commEl->max)->key) );

if (syncValRes != NULL)
  {
    int syn = (int) (syncValRes->data);
    maxVal = calculateWeight(maxCommVal, syn, omega);
  }
else
  {
    /* minimaler Wert fuer syn ist 1 */
    maxVal = calculateWeight(maxCommVal, 1, omega);
  }

/* Berechnung fuer second maximum Cluster */

/* gibts einen SyncWert fuer Cluster mit maximalem Kommunikationswert ? */
syncValRes = hsearch( syncEl->h, (void*)((commEl->secMax)->key) );

if (syncValRes != NULL)
  {
    int syn = (int) (syncValRes->data);
    secMaxVal = calculateWeight(secMaxCommVal, syn, omega);
  }
else
  {
    /* minimaler Wert fuer syn ist 1 */
    secMaxVal = calculateWeight(secMaxCommVal, 1, omega);
  }

/* Berechnung fuer third maximum Cluster */

/* gibts einen SyncWert fuer Cluster mit maximalem Kommunikationswert ? */
syncValRes = hsearch( syncEl->h, (void*)((commEl->thiMax)->key) );

if (syncValRes != NULL)
  {
    int syn = (int) (syncValRes->data);
    thiMaxVal = calculateWeight(thiMaxCommVal, syn, omega);
  }
else
  {
    /* minimaler Wert fuer syn ist 1 */
    thiMaxVal = calculateWeight(thiMaxCommVal, 1, omega);
  }
```

```
  /* entweder aktuellen Cluster zurueck oder aber Cluster mit maximalem Wert */
  /* (kann wegen Sync-Wert auch secMax-Cluster oder third-MaxCluster sein */

  /* finde groessten aus drei  */
  if (maxVal > secMaxVal) {
    if      (maxVal > thiMaxVal)
      {
        m = maxVal ; maxE = commEl->max;
      }
    else
      {
        m = thiMaxVal; maxE = commEl->thiMax;
      }
  }
  else
    {
      if      (secMaxVal > thiMaxVal)
        {
          m = secMaxVal; maxE = commEl->secMax;
        }
      else
        {
          m = thiMaxVal; maxE = commEl->thiMax;
        }
    }

  /* aktueller Wert ist trotzdem groesser */
  if (actVal > m)
    {
      return actClus;
    }
  /* max Cluster ist groesser, also Migration !!! */
  else
    {
      res = (unique_t*) (maxE->data);
      return *res;
    }
}

int calculateWeight(int commVal, int syncVal, int omega)
{
  return (commVal * syncVal * (omega/100));
}
```

# Abbildungsverzeichnis

# Definitionsverzeichnis

## Definitionen

# Literaturverzeichnis


[Agh86] Agha, G. A.
*Actors. A Model of Concurrent Computation in Distributed Systems.*
MIT Press, 1986.

[AH99] Au, A. und Heiser, G.
*L4 User Manual* 1999.

[And79] Andler, S.
*Predicate path expressions.*
In: *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* Seiten 226–236. ACM Press, 1979.

[ANS96] ANSI/IEEE.
*Information technology – Portable Operating System Interface – Part 1* IEEE Std 1003.1, 1996 Auflage 1996.

[ASU88a] Aho, A. V., Sethi, R. und Ullmann, J. D.
*Compilerbau, Teil 1.*
Addison-Wesley, Bonn/Deutschland, 1988.

[ASU88b] Aho, A. V., Sethi, R. und Ullmann, J. D.
*Compilerbau, Teil 2.*
Addison-Wesley, Bonn/Deutschland, 1988.

[BBH+98] Bal, H. E., Bhoedjang, R., Hofman, R. et al.
*Performance Evaluation of the Orca Shared-Object System.*
In: *ACM Transactions on Computer Systems* Band 16(1):1–4 1998.

[BBSS97] Broy, M., Breitling, M., Schätz, B. und Spies, K.
*Summary of Case Studies in Focus - Part II.*
Technischer Bericht TUM-I9740 1997.

[BDD+92] Broy, M., Dederich, F., Dendorfer, C. et al.
*The Design of Distributed Systems - An Introduction to FOCUS.*
Technischer Bericht TUM-I9202 1992.

[Bec97] Beck, M.
*Linux–Kernelprogrammierung.*
Addison-Wesley, 1997.

[Ben03] Benedek, T.
*Formale Beschreibung einer nebenläufigen, synchronisierten Sprache für verteilte Systeme.*
Diplomarbeit Technische Unisversität München Januar 2003.

[Ber72] Bertelsmann, L.-I., Herausgeber.
*Das moderne Lexikon.*
Nummer 18 in Bertelsmann Lexikon. Verlagsgruppe Bertelsmann, Güthersloh/Deutschland, 1972.

[BG97] Becker, C. R. und Geihs, K.
*MAQS: management for adaptive QoS-enabled services.*
In: *Proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems and Services.* 1997.

[BG98] Becker, C. und Geihs, K.
*Quality of Service: Aspects of Distributed Programs.*
In: *Proceedings of the Aspect-Oriented Programming Workshop (ICSE'98).* 1998.

[BGS89] Bal, H. E., G., S. J. und S., T. A.
*Programming Languages for distributed computing systems.*
In: *ACM Computing Surveys* Band 21(3):261–322 September 1989.

[BH75] Brinch Hansen, P.
*The programming language Concurrent Pascal.*
In: *IEEE Transaction on Software Engineering* Band SE-1(2):199–207 Juni 1975.

[BHR84] Brookes, S. D., Hoare, C. A. R. und Roscoe, A. W.
*A Theory of Communicating Sequential Processes.*
In: *Journal of the ACM (JACM)* Band 31(3):560–599 1984.

[BIR97] Booch, G., Ivar, J. und Rumbaugh, J., Herausgeber.
*UML Distilled Applying the Standard Object Modeling Language.*
Addison-Wesley Langman, Inc., 1997.

[BKLW00] Bode, A., Karl, W., Ludwig, T. und Wismüller, R.
*Monitoring Technologies for Parallel On-Line Tools.*
In: *SFB 342 Final Colloquium: Methods and Tools for the Efficient Use of Parallel Systems.* TU München, Deutschland, 2000.

[BKT92] Bal, H. E., Kaashoek, M. F. und Tanenbaum, A. S.
*Orca: a language for parallel programming of distributed systems.*
In: *IEEE Transactions on Software Engineering*
Band 18(3):190–205 1992.

[Blo79] Bloom, T.
*Evaluating synchronization mechanisms.*
In: *Proceedings of the seventh symposium on Operating systems principles* Seiten 24–32. 1979.

[Bly96] Blythe, J.
*Event-Based Decomposition for Reasoning about extrenal Change in Planners.*
American Association for Artificial Intelligence 1996.

[BN84] Birrell, A. D. und Nelson, B. J.
*Implementing remote procedure calls.*
In: *ACM Transactions on Computer Systems (TOCS)*
Band 2(1):39–59 1984.

[Bro94] Brown, C.
*UNIX Distributed Programming.*
Prentice Hall, New York, 1994.

[BS02] Broy, M. und Siedersleben, J.
*Objektorientierte Programmierung und Softwareentwicklung.*
In: *Informatik Spektrum* Februar 2002.

[Bus95] Buss, A. H.
*A Tutorial on discrete-event Modeling with Simulation Graphs.*
In: C. Alexopoulos, R. L., K. Kang und Goldsman, D., Herausgeber, *Proceedings of the 1995 Winter Simulation Conference* Seiten 74–81. 1995.

[Bus96] Buss, A. H.
*Modeling with Event Graphs.*
In: J. M. Charnes, D. T. B., D. J. Morrice und Swain, J. J., Herausgeber, *Proceedings of the 1996 Winter Simulation Conference* Seiten 153–160. 1996.

[Bus00] Buss, A. H.
*Component-based Simulation Modeling.*
In: J. A. Joines, K. K., R. R. Barton und Fishwick, P. A., Herausgeber, *Proceedings of the 2000 Winter Simulation Conference* Seiten 964–971. 2000.

[CDK01] Coulouris, G., Dollimore, J. und Kindberg, T.
*Distributed Systems. Concepts and Design.*
Pearson Education, 2001 3. Auflage.

[CE00] Czarnecki, K. und Eisenecker, U. W.
*Generative Programming – Methods, Tools, and Applications.*
Addison-Wesley, 2000.

[CHJ86] Cohen, B., Harwood, W. T. und Jackson, M. I.
*The Specification of Complex Systems.*
Addison-Wesley, 1986.

[CHP71] Courtois, P. J., Heymans, F. und Parnas, D. L.
*Concurrent control with "readers" and "writers".*
In: *Communications of the ACM* Band 14(10):667–668 1971.

[CJK94] Coulouris, G., Jean, D. und Kindberg, T.
*Distributed Systems – Concepts and Design.*
Addison-Wesley, Workingham/UK, 1994.

[CJS88] Cleaveland, R., J., P. und Steffen, B.
*The Concurrency Workbench.*
Report of LFCS Edinburgh University 1988.

[CKF+01] Coady, Y., Kiczales, G., Feeley, M. et al.
*Structuring operating system aspects: using AOP to improve OS structure modularity.*
In: *Communications of the ACM* Band 44(10):79–82 2001.

[Cor91] Corbato, F. J.
*On Building Systems That Will Fail.*
In: *Communications of the ACM* Band 34:72–81 1991.

[CP85] Cardelli, L. und Peter, W.
*On Understanding Types, Data Abstraction, and Polymorphism.*
In: *ACM Computing Surveys* Band 17(4):471–522 Dezember 1985.

[CPS+95] Cen, S., Pu, C., Staehli, R. et al.
*Demonstrating the Effect of Software Feedback on a Distributed Real-Time MPEG Video Audio Player.*
In: *ACM Multimedia* Seiten 239–240. 1995.

[Dah97] Dahlheimer, M. K.
*Java Virtual Machine – Sprache, Konzept, Architektur.*
O'Reilly Essentials, Köln/Deutschland, 1997.

[DH86] Dixon, R. D. und Hemmendinger, D.
*Analyzing synchronization problems by using event histories as languages.*
In: *Proceedings of the 1986 ACM fourteenth annual conference on Computer science* Seiten 183–188. ACM Press, 1986.

[DHT95] Doorn, L. V., Homburg, P. und Tanenbaum, A. S.
*Paramecium: An Extensible Object-Based Kernel.*
Seiten 86–89. 1995.

[Die00] Diestel, R.
*Graph Theory.*
Springer-Verlag, New York/USA, 2000.

[Dij68] Dijkstra, E. W.
*The Structure of THE Multiprogramming System.*
In: *Communications of the ACM* Band 11:341–346 1968.

[Dij76] Dijkstra, E. W.
*A Discipline of Programming.*
Prentice Hall, Englewood Cliffs, NJ/USA, 1976.

[Eck96] Eckert, C.
*Issues in the Design of Modern Distributed Computing Environments.*
In: *Proceedings of the Eighth IASTED International Conference on Parallel and Distributed Computing and Systems* Seite 188.
Chicago/USA, 1996.

[Eck00a] Eckel, B.
*Thinking in Java.*
Prentice Hall, 2000 2. Auflage.

[Eck00b] Eckert, C.
*IT-Sicherheit – Konzepte, Verfahren, Protokolle.*
R. Oldenbourg Verlag, 2000.

[ECS93] Engesser, H., Claus, V. und Schwill, A., Herausgeber.
*DUDEN - Informatik.*
DUDENVERLAG, Mannheim, Leipzig, Wien, Zürich, 1993 2. Auflage.

[EHS97] Ellsberger, J., Hogrefe, D. und Sarma, A.
*SDL - Formal Object-oriented Language for Communicating Systems.*
Prentice Hall Europe, 1997.

[Eme90] Emerson, E. A.
*Temporal and Modal Logic.*
In: *Handbook of Theoretical Computer Science* Band B Seiten 997–1067. Elsevier Science Publishers B. V., 1990.

[EP99a] Eckert, C. und Pizka, M.
*Improving Resource Management in Distributed Systems using Language-level Structuring Concepts.*
In: *The Journal of Supercomputing, Kluwer Academic Publishers, ISSN 0920-8542* Band 13:35–55 1999.

[EP99b] Eckert, C. und Pizka, M.
*Improving Resource Management in Distributed Systems using Language-level Structuring Concepts.*
In: *The Journal of Supercomputing* (13) 1999.

[EW95a] Eckert, C. und Windisch, H.-M.
*A new approach to match operating systems to application needs.*
In: *In IASTED - ISMM International Conference on Parallel and Distributed Computing and Systems.* 1995.

[EW95b] Eckert, C. und Windisch, H.-M.
*A top-down driven, object-based approach to application-specific operating system design.*
In: *In IEEE International Workshop on Object-Orientation in Operating Systems.* 1995.

[Exe01] *Executable Specifications: Creating Testable, Enforceable Designs.*
http://www.modeled-computaion.com Februar 2001.

[FBB+97] Ford, B., Back, G., Benson, G. et al.
*The Flux OSKit: a substrate for kernel and language research.*
In: *Proceedings of the sixteenth ACM symposium on Operating systems principles* Seiten 38–51. ACM Press, 1997.

[FH76] Flon, L. und Habermann, A. N.
*Towards the construction of verifiable software systems.*
In: *Proceedings of the 1976 conference on Data : Abstraction, definition and structure* Seiten 141–148. 1976.

[FHL+96] Ford, B., Hibler, M., Lepreau, J. et al.
*Microkernels Meet Recursive Virtual Machines.*
In: *Operating Systems Design and Implementation* Seiten 137–151.
1996.

[Fon97] Fontane, T.
*Effi Briest.*
Ullstein, Berlin/Deutschland, 1997 4. Auflage.

[FPM] Forster, M., Pick, A. und Marcus, R.
*The Graph Template Library (GTL).*
Universität Passau.

[GBD+94] Geits, A., Beguelin, A., Dongarra, J. et al., Herausgeber.
*PVM: Parallel Virtual Machine A Users Guide and Tutorial for Networked Parallel Computing.*
MIT press, 1994.

[GCC01] *Using and Porting the GNU Compiler Collection (GCC).*
http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc.html Februar 2001.

[GFWK02] Golm, M., Felser, M., Wawersich, C. und Kleinöder, J.
*The JX Operating System.*
In: *2002 USENIX Annual Technical Conference* Seiten 44–58.
Monterey, CA/USA, 2002.

[GJG96] Gosling, J., Joy, B. und Guy, S.
*The Java Language Specification.*
Addison-Wesley, Reading, MA/USA, 1996.
Http://java.sun.com/docs/books/jls/html/index.html.

[GLH+92] Gray, R. W., Levi, S. P., Heuring, V. P. et al.
*Eli: a complete, flexible compiler construction system.*
In: *Communications of the ACM* Band 35(2):121–130 1992.

[Gor88] Gordon, M. J.
*Programming Language Theory and Its Implementation* Seiten 13–55.
Prentice Hall International (UK) Ltd., 1988.

[Gos91] Goscinski, A.
*Distributed Operating Systems – The Logical Design.*
Addison-Wesley, Sydney, Australien, 1991.

[GP97] Groh, S. und Pizka, M.
*A Different Approach to Resource Management for Distributed Systems.*
In: *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications - PDPTA'97* Seiten 202 – 206. Las Vegas, NV/USA, 1997.

[GR97a] Groh, S. und Rudolph, J.
*Distributed Operating Systems based on Manager Agents.*
In: *Proc. of the 11th Annual International Symposium on High Performance Computing Systems - HPCS'97* Seiten 623 – 632. Winnipeg, Manitoba/Canada, 1997.

[GR97b] Groh, S. und Rudolph, J.
*On the Efficient Distribution of a Flexible Resource Management.*
In: *Proceeding of the IASTED International Conference on Parallel and Distributed Systems EuroPDS'97* Seiten 265 – 268. Barcelona/Spanien, 1997.

[GR00] Gurevich, Y. und Rosenzweig, D.
*Partially Ordered Runs: a Case Study.*
In: Gurevich, Y., Kutter, P., Odersky, M. und Thiele, L., Herausgeber, *Proceedings of ASM'2000* Band 1912 von *Abstract State Machines: Theory and Applications.* Springer-Verlag, 2000.

[Gra96] GrammaTech New York/USA.
*The Synthesizer Generator Reference Manual* 5. Auflage 1996.

[Gri81] Gries, D.
*The Science of Programming.*
Springer Inc., New York/USA, 1981.

[Gro96] Groh, S.
*Designing an Efficient Resource Management for Parallel Distributed Systems by the Use of a Graph Replacement System.*
In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA'96* Seiten 215 – 225. Sunnyvale/USA, 1996.

[Gro98] Groh, S.
*Ein agentenbasiertes, flexibel anpassungsfähiges Ressourcenmanagement für verteilte, parallele und kooperative Systeme.*
Dissertation Technische Universität München, Deutschland 1998.

[Gro00a] Grosch, J.
*Ast – A Generator for Abstract Syntax Trees.*
Technischer Bericht 15 CoCoLab - Datenverarbeitung 2000.

[Gro00b] Grosch, J.
*Puma – A Generator for the Transformation of Attributed Trees.*
Technischer Bericht 26 CoCoLab - Datenverarbeitung Karlsruhe/Deutschland 2000.

[GT96] Geppert, A. und Tombros, D.
*Event-based Distributed Workflow Execution with EVE.*
Technischer Bericht ifi-96.05 1996.

[GZ99] Gaspari, M. und Zavattaro, G.
*An Algebra of Actors.*
In: *Proc. 3nd IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS)* Seiten 3–18. Kluwer Academic Publishers, 1999.

[Hag01] Hagenmaier, T.
*Integration stellenübergreifender Kommunikation in den MoDiS-Kern.*
Systementwicklungsprojekt Technische Universität München 2001.

[Han02] Hanson, D. R.
*Lcc.NET: Targeting the .NET Common Intermediate Language from Standard C.*
Technischer Bericht MSR-TR-2002-112 Microsoft Research November 2002.

[HDS+95] Homburg, P., van Doorn, L., van Steen, M. et al.
*An Object Model for Flexible Distributed Systems.*
In: *Proceedings 1st Annual ASCI Conference* Seiten 69–78. Heijen/Niederlande, 1995.

[HM84] Hull, M. E. C. und McKeag, R. M.
*Communicating Sequential Processes for Centralized and Distributed Operating System Design.*
In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* Band 6(2):175–191 1984.

[HMU01] Hopcroft, J. E., Motwani, R. und Ullman, J. D.
*Introduction to Automata Theory, Languages, and Computation.*
Addison-Wesley, 2001 2. Auflage.

[Hoa74] Hoare, C. A. R.
*Monitors: an operating system structuring concept.*
In: *Communications of the ACM* Band 17(10):549–557 1974.

[Hoa85] Hoare, C. A.
*Communicating Sequential Processes.*
Series in Computer Science. Prentice Hall, 1985.

[Hof94] Hoffner, Y.
*Monitoring in Distributed Systems.*
Technischer Bericht Architecture Projects Management Limited Cambridge/UK 1994.

[HS97] Huber, F. und Schätz, B.
*Rapid Prototyping with AutoFocus.*
In: Wolisz, A., Schieferdecker, I. und Rennoch, A., Herausgeber, *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG'97 Fachgespräch* Seiten 343–352. GMD Verlag (St. Augustin)/Deutschland, 1997.

[HSSS96] Huber, F., Schätz, B., Schmidt, A. und Spies, K.
*AutoFocus - A Tool for Distributed Systems Specification.*
In: *Proceedings FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems, P. 467-470.* 1996.

[Hud91] Hudson, S. E.
*Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update.*
In: *ACM Transactions on Programming Languages and Systems* Band 13 Seiten 315–341. ACM Press, 1991.

[Hüt02] Hütter, B.
*CDSL: A concurrent, distibuted and synchronized language.*
http://www13.in.tum.de/ huetter/CDSL/ 2002.

[HV02] Hinze, A. und Voisard, A.
*A Parameterized Algebra for Event Notification Services* 2002.

[Int01] International, E.
*Common Language Infrastructure (CLI) , Standard ECMA-335.*
http://www.ecma.ch/ecma1/STAND/ecma-335.htm Dezember 2001.

[ITU99] ITU-T.
*Specification and description language (SDL)* 1999.

[JH02] Jähnichen, S. und Herrmann, S.
*Was, bitte, bedeutet Objektorientierung.*
In: *Informatik Spektrum* August 2002.

[Jon90] Jones, L. G.
*Efficent Evaluation of Circular Attribute Grammars.*
In: *ACM Transactions on Programming Languages and Systems* Band 12 Seiten 429–462. ACM Press, 1990.

[Kai93] Kaiser, K. S. M., Gail E.
*Parallel and Distributed Incremental Attribute Evaluation Algorthms for Multiuser Software Development Environments.*
In: *ACM Transactions on Software Engeneering and Methodology* Band 2 Seiten 47–92. ACM Press, 1993.

[Kas80] Kastens, U.
*Ordered Attribute Grammars.*
In: *Acta Informatica* Band 13(3):229–256 1980.

[Kat84] Katayama, T.
*Translation of Attribute Grammaras into Procedures.*
In: *ACM Transactions on Programming Languages and Systems* Band 6 Seiten 345–369. ACM Press, 1984.

[KDCZ94] Keleher, P., Dwarkadas, S., Cox, A. L. und Zwaenepoel, W.
*TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems.*
In: *Proc. of the Winter 1994 USENIX Conference* Seiten 115–131. 1994.

[KHH+01] Kiczales, G., Hilsdale, E., Hugunin, J. et al.
*An Overview of AspectJ.*
In: *Lecture Notes in Computer Science* Band 2072:327–355 2001.

[KLM+97] Kiczales, G., Lamping, J., Menhdhekar, A. et al.
*Aspect-Oriented Programming.*
In: Akşit, M. und Matsuoka, S., Herausgeber, *Proceedings European Conference on Object-Oriented Programming* Band 1241 Seiten 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[KR90] Kerninghan, B. W. und Ritchie, D. M.
*Programming in C.*
Addision-Wesley, 1990.

[Kra00] Kranzelmüller, D.
*Event Graph Analysis for debugging massively prallel Programs.*
Dissertation Johannes Kepler Universität Linz, Österreich 2000.

[Lam78] Lamport, L.
*Time, Clocks and the Ordering of Events in a Distributed System.*
In: *Communications of the ACM* Band 21. 1978.

[Lam83] Lampson, B. W.
*Hints for Computer System Design.*
In: *ACM Symposium on Operating System Principles* Operating Systems Review Seiten 33–48. 1983.

[LC75] Lauer, P. E. und Campbell, R. H.
*A description of path expressions by Petri nets.*
In: *Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages* Seiten 95–105. ACM Press, 1975.

[Lee90] van Leeuwen, J., Herausgeber.
*Handbook of Theoretical Computer Sience, Volume B.*
Elsevier Science Publishers B.V., Amsterdam/Niederlande, 1990.

[Lev99] Levine, J. R.
*Linkers and Loaders.*
Morgan-Kauffman, 1999.

[Lie96] Liedtke, J.
*Toward Real Microkernels.*
In: *Communications of the ACM* Band 39(9):70–77 1996.

[LV01] Lorenz, D. H. und Vlissides, J.
*Desining Components versus Objects: A Transformational Approach.*
IEEE 2001.

[LY99] Lindholm, T. und Yellin, F.
*The Java$^{TM}$ Virtual Machine Specification.*
Sun Microsystems, Inc., 1999 2. Auflage.

[LYI95] Lea, R., Yokote, Y. und ichiro Itoh, J.
*Adaptive Operating System Design using Reflection.*
In: *OBPDC* Seiten 205–218. 1995.

[Mac98] Macos, D.
*Implementation funktionaler Programmiersprachen durch Quelltexttransformation.*
Dissertation Humboldt-Universität Berlin/Deutschland 1998.

[Mar99] Martin Leucker and Thomas Noll.
*Rapid Prototyping of Specification Language Implementations.*
In: *IEEE International Workshop on Rapid System Prototyping* Seiten 60–65. 1999.

[MBKS96] McKusick, M. K., Bostic, K., Karels, M. J. und S., Q. J.
*The Design and Implementation of the 4.4 BSD Operating System.*
Addison-Wesley, Reading, MA/USA, 1996.

[MDJ93] Menon, S., Dasgupta, P. und Jr., R. J. L.
*Asynchronous Event Handling in Distributed Object-Based Systems.*
In: *International Conference on Distributed Computing Systems* Seiten 383–390. 1993.

[Mei00] Meier, R.
*State of the Art Review of Distributed Event Models* 2000.

[Mey88] Meyer, B.
*Object-oriented Sofware Construction.*
Series in Computer Sience. Prentice Hall International, Hertfordshire/UK, 1988.

[Mil80] Milner, R.
*A Calculus of Communicating Systems* Band 92 von *Lecture Notes in Computer Science.*
Springer–Verlag, 1980.

[Mil89] Milner, R.
*Communication and Concurrency.*
Series in Computer Sience. Prentice Hall Int., New York, London, 1989.

[Mil93] Milner, R.
*The Polyadic pi–Calculus: A Tutorial.*
In: Hamer, F., Brauer, W. und Schwichtenberg, H., Herausgeber, *Logic and Algebra of Specification.* Springer–Verlag, 1993.

[MK94] Micallef, J. und Kaiser, G. E.
*Extending attribute grammars to support programming-in-the-large.*
In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* Seiten 1572–1612. 1994.

[Mul89] Mullender, S., Herausgeber.
*Distributed Systems.*
ACM Press, 1989.

[MV93] Mandayam, R. und Vemuri, R.
*Performance Specification Using Attributed Grammars.*
In: *30th ACM/IEEE Design Automation Conference* Seiten 661–667. 1993.

[Nie90] Nielsen, K.
*Ada in Distributed Real-Time Systems.*
Intertext Publications/Multiscience Press, Inc., New York/USA, 1990.

[OMG01a] Object Management Group, Inc.
*OMG Unified Modeling Language Specification* September 2001.

[OMG01b] *The Common Object Request Broker: Architecture and Specification* September 2001.

[OMG02] *CORBA Components* Juni 2002.

[OW97] Odersky, M. und Wadler, P.
*Pizza into Java: Translating Theory into Practice.*
In: *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France* Seiten 146–159. ACM Press, New York (NY), USA, 1997.

[Paa95] Paakki, J.
*Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation.*
In: *ACM Computing Surveys* Band 27(2):196–255 1995.

[PE97] Pizka, M. und Eckert, C.
*A language-based approach to construct structured and efficient object-based distributed systems.*
In: El-Rewini, H. und Patt, Y., Herausgeber, *Proc. of 30th Hawaii International Conference on System Sciences - HICSS* Band 1 Seiten 130 – 139. IEEE CS Press, 1997.

[PEG97] Pizka, M., Eckert, C. und Groh., S.
*Evolving Software Tools for New Distributed Computing Environments.*
In: *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications - PDPTA'97* Seiten 87 – 96. Las Vegas, NV/USA, 1997.

[Pet62] Petri, C. A.
*Kommunikation mit Automaten.*
Dissertation Rheinisch-Westfälisches Institut für Instrumentelle Mathematik 1962.

[PFH02] Peschel-Findeisen, T. und Hütter, B.
*Structuring concepts in operating system design.*
In: Hamza, M. H., Herausgeber, *International Symposium on Parallel and Distributed Computing and Networks* Applied Informatics Seiten 161–166. IASTED ACTA Press, Anaheim, Calgary, Zurich, 2002.

[PFH03] Peschel-Findeisen, T. und Hütter, B.
*A Generated Management for Distributed Systems.*
In: Hamza, M. H., Herausgeber, *International Symposium on Parallel and Distributed Computing and Networks* Applied Informatics Seiten 845 — 850. IASTED ACTA Press, Anaheim, Calgary, Zurich, 2003.

[PH93] Poetzsch-Heffter, A.
*Programming Language Specification and Prototyping Using the MAX System.*
In: Bruynooghe, M. und Penjam, J., Herausgeber, *Programming Language Implementation and Logic Programming* LNCS 714 Seiten 137–150. Springer-Verlag, 1993.

[Piz97] Pizka, M.
*Design and Implementation of the GNU INSEL Compiler (GIC).*
Technischer Bericht TUM–I9713, SFB–Bericht 342/09/97 A Technische Universität München/Deutschland Oktober 1997.

[Piz99] Pizka, M.
*Integriertes Management erweiterbarer verteilter Systeme.*
Dissertation Technische Universität München/Deutschland 1999.

[Pre02] Preissinger, J.
*Realisierung einer verteilten Wissensbasis für kooperative Systeme.*
Systementwicklungsprojekt Technische Universität München 2002.

[PS95] Plainfosse, D. und Shapiro, M.
*A Survey of Distributed Garbage Collection Techniques.*
In: *International Workshop on Memory Management.* 1995.

[Rac01] Rackl, G.
*Mointoring and Managing Heterogeneous Middleware.*
Dissertation Technische Universität München Deutschland 2001.

[RDAS02] Rajogopalan, M., Debray, S. K., A., H. M. und Schlichting, R. D.
*Profile-Directed Optimization of Event-Based Programs.*
In: *Proceedings of the PLDI'02* Seiten 106–116. ACM, 2002.

[RDD99] Ram, P., Do, L. und Drew, P.
*Distributed transactions in practice.*
In: *ACM SIGMOD Record* Band 28(3):49–55 1999.

[Ree93] Rees, O.
*Using Path Expressions as Concurrency Guards.*
Technischer Bericht Architecture Projects Management Limited Cambridge/UK 1993.

[Rei97] Reimer, N.
*Untersuchung von Strategien für verteiltes Last- und Ressourcenmanagement.*
Technischer Bericht SFB-Bericht 342/08/97 A TUM-I9712 Technische Universität München Deutschland April 1997.

[RHT96] Reimer, N., Hänssgen, S. U. und Tichy, W. F.
*Dynamically Adapting the Degree of Parallelism with Reflexive Programs.*
In: *In Proceedings of the Third International Workshop on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR).* 1996.

[RP01] Rehn, C. und Pizka, M.
*Murks - A POSIX Threads Based DSM System.*
In: *Proc. of the thirteenth IASTED International Conference on Parallel and Distributed Computing and Systems - PDCS 2001.* Anaheim, CA/USA, 2001.

[RT89] Reps, T. W. und Teitelbaum, T.
*The Synthesizer Generator: A System for Constructing Language-Based Editors.*
Springer Inc., New York/USA, 1989.

[RW96] Radermacher, R. und Weimar, F.
*INSEL Syntax-Bericht.*
Technischer Bericht TUM–I9617, SFB–Bericht 342/08/96 A Technische Universität München Deutschland März 1996.

[San95] Sander, G.
*VCG – Visualization of Compiler Graphs.*
Universität des Saarlandes Saarbrücken/Deutschland Februar 1995.

[Sar99] Saraiva, J. a. A. B. V.
*Purely Functional Implementation of Attribute Grammars.*
Proefschrift Universiteit Utrecht/Niederlande 1999.

[Sch83] Schruben, L.
*Simulation Modeling with Event Graphs.*
In: *Communications of the ACM* Band 26(11):957–963 November 1983.

[Sch98] Schwoon, S.
*Übersetzung von SDL-Spezifikationen in Petri-Netze.*
Diplomarbeit Unisversität Hildesheim 1998.

[Sch00] Schruben, L. W.
*Mathematical Programming Models of discrete Event System Dynamics.*
In: J. A. Joines, K. K., R. R. Barton und Fishwick, P. A., Herausgeber, *Proceedings of the 2000 Winter Simulation Conference* Seiten 381–384. 2000.

[SEL+96] Spies, P. P., Eckert, C., Lange, M. et al.
*Sprachkonzepte zur Konstruktion verteilter Systeme.*
Technischer Bericht TUM-I9618, SFB 342/09/96 A Technische Universität München/Deutschland März 1996.

[SEP+00] Spies, P., Eckert, C., Pizka, M. et al.
*Flexible, Distributed and Adaptive Resource Management in MoDiS.*
In: *SFB 342 Final Colloquium: Methods and Tools for the Efficient Use of Parallel Systems.* Technische Universität München, München/Deutschland, 2000.

[SG98] Silberschatz, A. und Galvin, P. B.
*Operating System Concepts.*
Addison-Wesley, Reading, MA/USA, 1998 5. Auflage.

[SGS95] SGS-THOMPSON Microelectronics Limited.
*occam 2.1 reference manual* 1995.

[SH99] Schätz, B. und Huber, F.
*Integrating Formal Description Techniques.*
In: Wing, J. M., Woodcock, J. und Davies, J., Herausgeber, *FM'99 – Formal Methods, Proceedings of the World Congress on Formal Methods in the Development of Computing Systems, Volume II* Band 1709 von *Lecture Notes in Computer Science* Seiten 1206–1225. Springer Verlag, 1999.

[Slo95] Sloane, A. M.
*An evaluation of an automatically generated compiler.*
In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* Band 17(5):691–703 1995.

[SM97] Shields Michael, W.
*Semantics of Parallelism – Non-Interleaving Representation of Behauviour.*
Springer, Berlin Heidelberg New York, 1997.

[SOHL+96] Snir, M., Otto, S., Huss-Lederman, S. et al., Herausgeber.
*MPI: The Complete Reference.*
MIT Press, 1996.

[Son93] Sonntag, S.
*Adaptierbarkeit durch Reflektion.*
Dissertation Technische Universität Chemnitz Deutschland 1993.

[Spi98a] Spies, K.
*Eine Methode zur formalen Modellierung von Betriebssystemkonzepten.*
Dissertation Technische Universität München 1998.

[Spi98b] Spies, P. P.
*Ereignisverbände – Ein flexibles Beschreibungsinstrumentarium für die Entwicklung Verteilter Systeme.*
In: *FBT'98–Fachgespräch.* Cottbus/Deutschland, 1998.

[Spr02] Sprenger, T.
*Implementierung eines Compilers für die verteilte Sprache CDSL.*
Systementwicklungsprojekt Technische Universität München 2002.

[SR00] Solomon, D. A. und Russinowich, M. E.
*Inside Windows 2000.*
Microsoft Press, 2000 3. Auflage.

[SRC84] Saltzer, J. H., Reed, D. H. und Clark, D. D.
*End-to-End Arguments in System Design.*
In: *Transactions on Computer Systems* 2. 1984.

[SS95] Small, C. und Seltzer, M.
*Structuring the Kernel as a Toolkit of Extensible, Reusable Components* 1995.

[SSE02a] Schröter, C., Schwoon, S. und Esparza, J.
*The Model-Checking Kit.*
In: *Proceedings of Toolsday-Workshop (satellite event of CONCUR'02)* Report Series FIMU-RS-2002-05 Seiten 22–31.
Masaryk University, Brno/Ungarn, 2002.

[SSE02b] Schröter, C., Schwoon, S. und Esparza, J.
*The Model-Checking Kit.*
http://www7.informatik.tu-muenchen.de/gruppen/theorie/KIT/ 2002.

[SSS95] Seltzer, M., Small, C. und Smith, K.
*The Case for Extensible Operating Systems* 1995.

[Sta95] Stansifer, R.
*Theorie und Entwicklung von Programmiersprachen.*
Prentice Hall, München/Deutschland, 1995.

[Str97] Stroustrup, B.
*The C++–Programming Language.*
Addison-Wesley, 1997 3. Auflage.

[Sul01] Sullivan, G. T.
*Aspect-oriented programming using reflection and metaobject protocols.*
In: *Communications of the ACM* Band 44(10):95–97 2001.

[Sun97] Sunha, P. K.
*Distributed Operating Systems. Concepts and Design.*
IEEE Press, 1997.

[Tan95] Tanenbaum, A. S.
*Distributed Operating Systems.*
Prentice Hall, 1995.

[Tan01] Tanenbaum, A. S.
*Modern Operating Systems.*
Prentice Hall, New Jersey/USA, 2001 2. Auflage.

[TH90] Tassel, J. V. und Hemmendinger, D.
*Specifying and automatically generating Ada tasks in Prolog.*
In: *Proceedings of the 1990 ACM annual conference on Cooperation* Seiten 121–127. ACM Press, 1990.

[Wal88] Walz, J. A.
*Incremental Evaluation for a General Class of Circular Attribute Grammars.*
In: *Conference on Programming Desgin and Implementation* Seiten 209–221. ACM Press, 1988.

[Web98] Weber, M.
*Verteilte Systeme.*
Spektrum Akademischer Verlag, Heidelberg, Berlin, 1998.

[Wei97] Weimer, F.
*Davit: Ein System zur interaktiven Ausführung und zur Visualisierung von Insel-Programmen.*
Technischer Bericht SFB-Bericht 342/15/97 A TUM-I9721
Technische Universität München Deutschland April 1997.

[Wer92] Werner, D.
*Theorie der Betriebssysteme.*
Hanser, München, Wien, 1992.

[WG92] Wirth, N. und Gutknecht, J.
*Project Oberon – The Design of an Operating System and Compiler.*
ACM Press, 1992.

[Win94] Winskel, G.
*The Formal Semantics of Programming Languages: An Introduction.*
The MIT Press, Cambrige, London, 1994 2. Auflage.

[Win96] Windisch, H.-M.
*The distributed programming language insel - concepts and implementation.*
In: *In First International Workshop on High-Level Programming Models and Supportive Environments* Seiten 17 – 24. 1996.

[Wir90] Wirsing, M.
*Algebraic Specification.*
In: van Leeuwen, J., Herausgeber, *Handbook of Theoretical Computer Science, Volume B* Seiten 675–788. Elsevier Science Publishers B.V., Amsterdam/Niederlande, 1990.

[WM97] Wilhelm, R. und Maurer, D.
*Übersetzerbau.*
Springer-Verlag, Berlin Heidelberg New York, 1997.

[Yok92] Yokote, Y.
*The Apertos reflective operating system: The concept and its implementation.*
In: Paeypcke, A., Herausgeber, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* Band 27 Seiten 414–434. ACM Press, New York/USA, 1992.

[Yok93] Yokote, Y.
*Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach.*
In: *Object Technologies for Advanced Software, First JSSST International Symposium* Band 742 Seiten 145–162.
Springer-Verlag, 1993.

[ZK93] Zimmermann, C. und Krass, A. W.
*Mach: Konzepte und Programmierung.*
Springer-Verlag, 1993.

# Index

## Symbole



## A

## B



## C



## D



## E

## O



## P



## R



## S