Konfliktbehandlung im policy–basierten Management mittels a priori Modellierung

Dissertation

an der

Fakultät für Mathematik, Informatik und Statistik
der
Ludwig-Maximilians-Universität München

vorgelegt von

Bernhard Kempter

Tag der Einreichung: 09. Juli 2004
Tag der mündlichen Prüfung: 02. August 2004

1. Berichterstatter: **Prof. Dr. Heinz-Gerd Hegering**, Universität München
2. Berichterstatter: **Prof. Dr. Johann Schlichter**,Technische Universität München

Danksagung

Die vorliegende Arbeit entstand während meiner Zeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Kommunikationssysteme und Systemprogrammierung der Ludwig–Maximilians–Universität München.

Ein besonderer Dank gebührt meinem Doktorvater Prof. Dr. Heinz-Gerd Hegering, der mir in all den Jahren stets den richtigen Weg aufgezeigt hat, darauf vertraute, dass ich ihn auch gehen werde und mir den Freiraum für Nebenpfade gab. Ebenso danke ich Herrn Prof. Dr. Johann Schlichter für seine wertvollen Anmerkungen, welche die Arbeit spürbar bereicherten.

Meinen Kolleginnen und Kollegen am Lehrstuhl und MNM–Team sei für die angenehme Arbeitsumgebung und das herzliche Arbeitsklima gedankt, ohne die die vorliegende Arbeit gar nicht wachsen und gedeihen hätte können. In intensiver und ausdauernder Weise haben sich um das Düngen und Stutzen Markus Garschhammer und Harald Rölle bemüht — was wäre ohne Euch daraus geworden?

Nicht zuletzt danke ich meiner Familie und vor allem Stephanie für die große Unterstützung und stetigem Rückhalt in allen Phasen der Doktorarbeit.

Zusammenfassung

Das policy–basierte Management nimmt sowohl in der Forschung als auch in der Industrie einen steigenden Stellenwert ein. Durch die verteilte Spezifikation und aufgrund divergenter Ziele können Policies zueinander in Konflikt stehen. Die bereits existierenden Ansätze zur Policy–Konfliktbehandlung sind nur begrenzt einsetzbar, da sie häufig auf eine dediziert Policy–Sprache limitiert sind, wichtige Konfliktarten per se nicht erkennen können oder für neuartige Konfliktarten keine Methodik zur Integration bieten.

Diese Arbeit zeigt, dass unter Berücksichtigung von Managementmodellen neue Konfliktarten nachgewiesen werden können, die bis jetzt mit den Ansätzen in der Literatur nicht behandelbar sind. Dazu werden Managementmodelle als *a priori* Modelle aufgefasst. Ein a priori Modell beschreibt den Sollzustand eines Systems und definiert somit eine Menge von einzuhaltenden Bedingungen. Unter dieser Prämisse werden neuartige Konflikte — Konflikte zwischen Beziehungen von Managementobjekten — nachgewiesen.

Den Kern der Lösungsidee bildet eine Methodik zur Ableitung von Konfliktdefinitionen aus Modellaspekten. Dabei werden ausgehend von Modellaspekten Invarianten abgeleitet, mit Policy–Aktionen verknüpft und schließlich Vorbedingungen definiert, deren Einhaltung Konflikte verhindert. Die breite Anwendbarkeit der Methodik wird anhand eines statischen Beziehungsmodells für die Beziehungen der funktionalen Abhängigkeit und Enthaltenseinsrelationen gezeigt. Ebenso wird die Anwendbarkeit der Methodik für Vertreter von dynamischen Modellen, den endlichen Automaten demonstriert.

Zur Konfliktbehandlung wurde ein neuer Algorithmus entwickelt, der aus den Phasen Konfliktlokalisierung, Konflikterkennung und Konfliktlösung besteht. In der ersten Phase wird durch Teilmengenbildung die Anzahl der zu betrachtenden Policies schnell reduziert. In der letzten Phase werden für die einzelnen Konfliktarten Strategien entwickelt, die eine optimale Konfliktlösung gewährleisten. Der Algorithmus ist sowohl für die präventive als auch die reaktive Konfliktbehandlung anwendbar.

Damit eine generische Lösung erreicht wird, sind wichtige Designziele für die Methodik und dem Algorithmus: die Unabhängigkeit von einer dedizierten Policy–Sprache, die Breite der behandelbaren Konfliktarten sowie die Unabhängigkeit von einem spezifischen Managementinformationsmodell. Die Anwendbarkeit der Lösung in der Praxis wird durch eine exemplarische Abbildung der Konfliktdefinitionen in das Common Information Model gezeigt.

Summary

Policy based management plays an important role in the research community and in the industry. Policies can be in conflict to each other due to specification of policies by several persons or when targeting divergent goals. Existing approaches for policy detection and resolution are limited: they are only applicable for distinct policy languages. They are only able to detect certain conflict types or offer no methodology to integrate handling of new conflicts types.

The assessment of conflicts shows that by analyzing management models, new kinds of conflicts can be identified, which are not addressed by existing solutions in a generic sense. Therefore management models are used as *a priori* models. An a priori model describes a target state of a system and can be seen as set of conditions a policy system has to observe. Under this premise a new conflict type — a conflict between related managed objects — can be identified.

The core of the solution presented is a methodology to derive conflict definitions out of (a priori) model properties. Starting with the definition of invariants of model properties, policy actions are linked with these invariants and in the end preconditions are specified which prevent policy conflicts from occurring. The general applicability of the methodology is demonstrated for functional dependencies, containment hierarchies and finite automata.

A new algorithm was developed for conflict handling, which consisting of the phases conflict localization, conflict detection and conflict solution. In the first phase, the algorithm quickly reduces the set of policies to be considered. After having performed conflict detection by means of invariant evaluation in the second phase, in the last phase, for each conflict type a distinct strategy is developed to ensure optimal conflict solution. The algorithm is applicable both for preventive and reactive conflict handling.

To achieve a generic solution, important design goals for the methodology and the algorithm were independence of specific policy languages, support for conflict types not known at design time and independence of a specific management information model. The applicability of the approach in practice is demonstrated in an exemplary manner by the mapping of conflict definitions to the Common Information Model.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Kapitelüberblick

1.1 Motivation

Das Management eines Unternehmens lässt sich nach [HAN 99a] in folgende fundamentale Managementdisziplinen einteilen:

Die unteren vier Ebenen werden als technisches Management zusammengefasst, während die oberen Ebenen sich den Geschäftsprozessen, also eher den unternehmerischen Aspekten widmen.

Die vorliegende Arbeit konzentriert sich auf das technische Management. Wesentlich für diese Arbeit ist, dass das Management einer höheren Ebene zu erreichenden Ziele als Anforderungen an die unterliegende Ebene definiert und weiterreicht.

Unternehmensziele sollen in den Ebenen des technischen Managements umgesetzt werden, ohne dass diese bis jetzt hinreichend formal gefasst sind. Das technische Management versucht diese Ziele in dem von ihm zu beeinflussenden Managed System durchzusetzen.

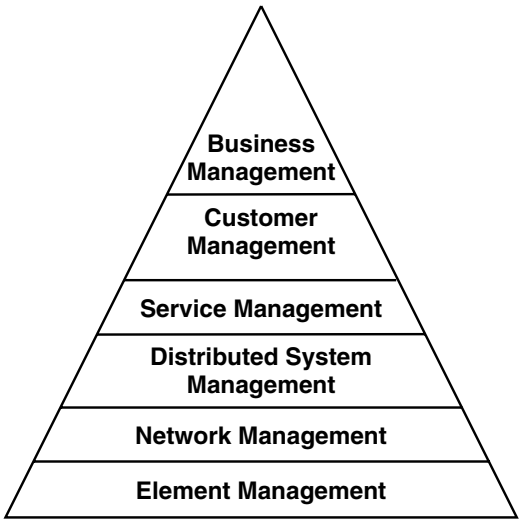

Abbildung 1.1: Managementpyramide nach [HAN 99a]

In [HAN 99a] wird das technische Management wie folgt definiert:

> *„ Das Management vernetzter Systeme umfasst in seiner allgemeinsten Definition alle Maßnahmen, die einen effektiven und effizienten, an Zielen des Unternehmens ausgerichteten Betrieb der Systeme und ihrer Ressourcen sicherstellen. “*

Damit die Ziele erreicht werden ist es notwendig, die Managementaufgaben gemäß den Zielen auszurichten und die korrekte Durchführung zu gewährleisten.

Typische Ziele des Business Management sind beispielsweise: *Alle Dienste müssen sicher, performant und billig betrieben werden.* Hierbei sieht man bereits, dass sich diese Ziele gegenseitig wiedersprechen. Man wird nicht alle drei Ziele gemeinsam mit ihrem Maximum erreichen können, sondern muss Kompromisse eingehen. Das technische Management versucht, diese Ziele so optimal wie möglich umzusetzen. Da das technische Management reale Ressourcen betreibt, werden durch die divergenten Unternehmensziele konkrete, auf den beschränkten Ressourcen auftretende Konflikte induziert. In Abschnitt 1.1.1 wird dieses Verhalten an einem konkreten Beispiel demonstriert.

Die unterschiedlichen Ziele, denen das IT-Management gerecht werden muss, und die organisatorische Aufteilung des Managements in Domänen, z.B. in Abteilungen, erschweren den reibungslosen Betrieb der gesamten IT-Infrastruktur.

Ein vielversprechender Ansatz, um diese unterschiedlichen Aufgaben einheitlich zu erfüllen, ist das Policy-basierte Management. Policies sind sowohl geeignet, in

den verschiedenen Managementbereichen angewendet zu werden, als auch Managementaufgaben in unterschiedlicher Granularität ausdrücken zu können.

Eine Policy ist eine aus den Managementzielen abgeleitete Regel, die durch auszuführende Aktionen aktiv das zu managende System beeinflusst. Policies werden in der Regel von mehreren Personen (Administratoren etc.) spezifiziert, wobei unterschiedliche Ziele verfolgt werden können. Werden mehrere Policies gleichzeitig ausgeführt, so können die beteiligten Policies in Konflikt zueinander stehen. Allgemein gesprochen ist ein Konflikt eine Situation, in der ein festgelegtes Ziel aufgrund anderer Ziele nicht erreichbar ist. Diese Konfliktsituation kann zu einem nicht vorhersehbaren bzw. inkonsistenten Systemverhalten führen.

Im heutigen Managementbetrieb werden Konflikte meistens dadurch gelöst, dass die beteiligten Organisationen sich an einen Tisch setzen, das Problem besprechen und durch geeignete Maßnahmen beheben. Durch die fortschreitende Automatisierung der Aufgaben im Managementbereich werden aber auch die Konflikte mit in den Managementprozess hineingetragen, ohne dass diese gelöst werden.

Dadurch wächst das Bedürfnis, eine automatisierte Konfliktbehandlung zu realisieren. Diese Konfliktbehandlung soll allgemein gehalten sein, damit eine möglichst breite Palette an Konflikten gelöst (oder zumindest erkannt) werden kann.

Policies sind für ein breites Anwendungsfeld im IT–Management geeignet. Beispielhaft werden wichtige Vertreter aus unterschiedlichen Bereichen des Managements genannt: Im Bereich des Sicherheitsmanagement ist die Policy–Sprache Ponder [DDLS 00], welche am Imperial College London entwickelt wurde, ein wichtiger Vertreter. Ponder eignet sich besonders, um Pflichten und die dazu nötige Autorisierung von Aufgaben auszudrücken. Im Bereich des Abrechnungsmanagement existiert mit ProPolis eine prozessorientierte Policy–Sprache [Radi 03, DaKe 04], welche den Prozess der Abbrechnung steuert. Im Bereich des QoS–Managements existieren eine Vielzahl von Policy–Sprachen wie Hierarchical QoS Markup Language (HQML) [GNY$^+$ 01] oder die Quality of Service Modeling Language (QML) [FrKo 98]. In [Heil 00] werden Policies im Bereich nomadischer Systeme eingesetzt. Ein Überblick und Vergleich von Policy–Sprachen ist in [DDGH 02] zu finden.

Im Folgenden wird ein Szenario zur Verdeutlichung der Problematik von Policy–Konflikten vorgestellt.

1.1.1 Beispiel eines Policy–Konflikts

Ein Unternehmen verfolgt u.a. zwei Ziele. Einerseits sollen die Betriebskosten minimiert werden und andererseits optimale Arbeitsbedingungen für die Mitarbeiter

gewährleistet werden (siehe Abbildung 1.2).

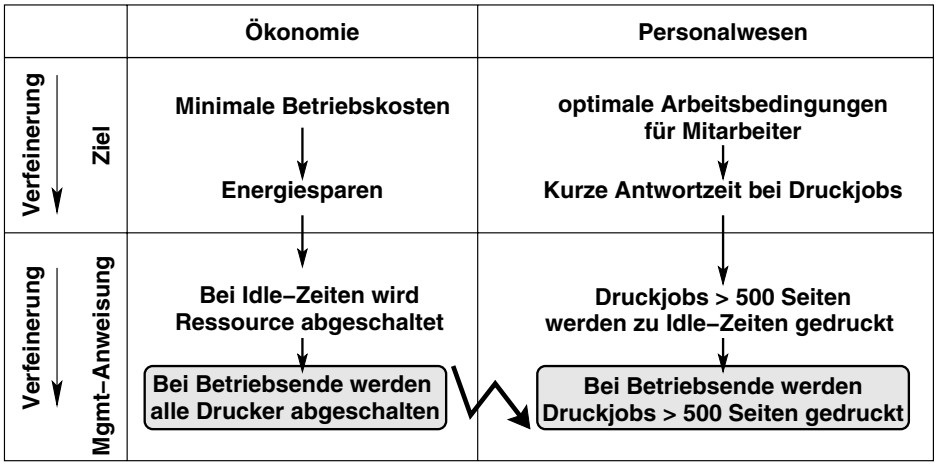

Abbildung 1.2: Unterschiedliche Ziele und daraus resultierender Konflikt

Eine der möglichen Folgerungen des Ziels der minimalen Betriebskosten ist das (Unter–)Ziel Energie zu sparen. Dieses Ziel wird z.B. durch die Managementanweisung, dass bei längeren Idle-Intervallen alle stromverbrauchenden Ressourcen abgeschaltet werden, umgesetzt.

Von dem zweiten Ziel, optimale Arbeitsbedingungen für Mitarbeiter, wird u.a. das Ziel abgeleitet, dass man auf Druckjobs nicht lange warten muss. Konsequent wurde dies u.a. umgesetzt, indem Druckaufträge, die größer als 500 Seiten lang sind, erst nach Betriebsende gedruckt werden.

Die obersten Ziele wurden jeweils von der Unternehmensleitung für alle Abteilungen vorgegeben. Die nachgelagerten Ziele wurden von der IT-Abteilung erarbeitet. Die Umsetzung der Ressourcenabschaltung wurde von der Gruppe Systemmanagement erarbeitet, während die Entscheidung große Druckaufträge nach Betriebsende zu drucken, von der Gruppe Anwendungsmanagement getroffen wurde.

Der Konflikt zeigt sich hier bei den konkreten Managementanweisungen. Einerseits sollen alle Drucker nach Betriebsende abgeschaltet werden und andererseits sollen in dieser Zeit große Druckjobs bearbeitet werden. Interessant ist, dass auf der Ebene der Ziele noch keine Aussage getroffen werden kann, ob ein Konflikt aufgetreten ist oder nicht.

1.1.2 Defizite bestehender Ansätze

In der Literatur existiert eine Reihe von Ansätzen, um Policy–Konflikte zu behandeln. Die wesentlichen Defizite dieser Arbeiten werden in Kapitel 3 detailiert untersucht. Eine Zusammenfassung dieser Ansätze und damit die Motivation für die vorliegende Arbeit wird nun kurz vorgestellt.

Spezifische Voraussetzungen für Policy–Konflikte In der Literatur wird in der Regel eine sehr enge Konfliktdefinition gewählt. Die meisten Konfliktdefinitionen gehen davon aus, dass folgende notwendige Bedingungen gegeben sein muss, damit ein Policy–Konflikt auftreten kann: es muss eine *Domänenüberlappung* der Policies vorliegen (siehe Abschnitt 3.2). Eine Domäne ist eine Menge von Objekten, die von einer Policy betroffen ist. Eine Domänenüberlappung ist gegeben, wenn die Schnittmenge zweier Domänen nicht leer ist. In Abschnitt 2.5 wird gezeigt, dass weitere wichtige Konfliktearten existieren, diediese Voraussetzung nicht erfüllen. Daraus folgt, dass neben einer neuen Konfliktdefinition auch ein Konfliktbehandlung entwickelt werden muss, die mit Konflikten dieser Art umgehen kann.

Konfliktbehandlung auf dedizierte Policy–Sprache limitiert Die Konfliktbehandlungsansätze in der Literatur sind häufig auf eine Policy–Sprache zugeschnitten. Beispielsweise basiert die Konfliktbehandlung bei dem Ansatz von Sloman und Lupu (siehe Abschnitt 3.2.2) auf der Typisierung von Policies, die bereits in der Grammatik der Sprache *Ponder* enthalten ist. Will man die Konfliktbehandlung für eine weitere Policy–Sprache einsetzen, so müsste man die bestehende Grammatik dieser Policy–Sprache anpassen, was insgesamt zu einem hohen Änderungsaufwand führen würde.

Fehlende Methodik zur Konfliktbehandlung Bis jetzt existiert in der Literatur keine allgemeine Methodik, die den Prozess der Konfliktbehandlung von der Definition des Konflikts über die Konflikterkennung bis zur Konfliktlösung adäquat unterstützt. Der Vorteil einer Methodik liegt in der schnelleren und weniger fehleranfälligen Spezifikation einer Konfliktbehandlung. In Abschnitt 4.2 wird eine Methodik vorgeschlagen, die für unterschiedliche Konfliktarten eine Konflikterkennung und Konfliktlösung systematisch erschließt.

Aus den betrachteten Defiziten lassen sich eine Reihen von Fragestellungen ableiten, die im Rahmen dieser Arbeit behandelt werden, um die geschilderten Probleme zu lösen.

1.2 Fragestellung

Die Fragestellung lässt sich in die Bereiche Konflikterkennung, Konfliktlösung, Konfliktkategorisierung sowie übergreifende Fragen aufteilen.

1. Konflikterkennung und Konfliktdefinition

   (a) Existiert ein einheitliches Merkmal für Policy–Konflikte? Ist es möglich, trotz unterschiedlicher Anwendungsbereiche und Policy–Sprachen ein einheitliches Merkmal zu definieren?

   (b) Wo sind Policy–Konflikte in einem Managementsystem lokalisierbar? Treten Policy–Konflikte immer an der gleichen Stelle auf bzw. kann man vom dem Auftretungsort Rückschlüsse auf den Policy–Konflikt ziehen?

   (c) Zu welchem Zeitpunkt (bezogen auf den Policy–Lebenszyklus) kann ein Konflikt erkannt werden?

2. Konfliktlösung

   (a) Müssen für in dieser Arbeit betrachteten Konfliktarten neue Konfliktlösungsstrategien erarbeitet werden?

   (b) Ist es möglich, dass Konflikte erst gar nicht auftreten, also vermieden werden können? Ist dieser Vorgang automatisierbar?

   (c) Ist es möglich, die Konfliktlösung so zu gestalten, dass sie für beliebige Policy–Sprachen einsetzbar ist?

3. Konfliktkategorisierung

   (a) Ist die bis jetzt in der Literatur bekannte Konfliktkategorisierung für die in dieser Arbeit betrachteten Konflikte ausreichend?

   (b) Welche neuen Konfliktkategorien lassen sich ableiten?

4. Übergreifende Fragen beschäftigen sich mit der Konfliktbehandlung als Ganzes und deren Eingliederung in das Managementumfeld:

   (a) Existiert ein allgemeines methodisches Vorgehen, um Konflikte zu behandeln?

   (b) Wie kann die Konfliktbehandlung geeignet in das IT–Management integriert werden, da Konflikte funktionsübergreifend auftreten?

(c) Ist die Konfliktbehandlung für Policies grundsätzlich unterschiedlich zu Managementansätzen, die keine Policies verwenden? (Das Auftreten von Konflikten ist kein Phänomen, das nur Policies betritt, sondern generell beim Management vorkommt.)

1.3 Vorgehen

Der in diesem Kapitel vorgestellte Beispielkonflikt und die Fragestellungen werden im folgenden Kapitel 2 allgemein analysiert (siehe Abbildung 1.3). In der Problemanalyse werden die Fragestellung und die relevanten Begriffe präzisiert und anhand eines formalisierten Konfliktmodells, das in Ebenen unterteilt ist, diskutiert. Anschließend wird eine Analyse der Policy–Konflikte auf jeder Ebene durchgeführt.

Im anschließenden Kapitel 3 werden die in der Literatur existierende Ansätze zur Policy–Konfliktbehandlung diskutiert und anhand der in Kapitel 2 identifizierten Defizite bewertet. Verwandte Ansätze, die eine Relevanz zur vorliegenden Arbeit aufweisen, werden ebenfalls vorgestellt: das Goal–Oriented Requirements Engineering und Verklemmungen in Betriebssystemen.

In Kapitel 4 wird eine Methodik vorgestellt, die basierend auf Managementmodellen zu einer Konfliktdefinition und schließlich zu einer Konfliktlösung gelangt. Dabei werden Modelle als *a priori* Modelle verstanden. Ein a priori Modell beschreibt den Sollzustand eines Systems und besitzt damit einen planerischen Charakter. Unter dieser Sichtweise kann ein a priori Modell als eine Sammlung von Bedingungen angesehen werden. Die Anwendbarkeit der Methodik wird anhand verschiedener statischer und dynamischer Modelle gezeigt.

Aufbauend auf diesen Lösungsansatz wird in Kapitel 5 ein Konfliktbehandlungsalgorithmus entwickelt, der effizient eine Konfliktlokalisierung, Konflikterkennung und Konfliktlösung realisiert. Dabei kann der Algorithmus sowohl in der Spezifikationsphase von Policies als auch während der Policy–Ausführung eingesetzt werden.

Kapitel 6 zeigt die Integration der generischen Modelle und Konfliktdefinitionen in das Management–Informationsmodell der Distributed Management Task Force, dem Common Information Model.

In Kapitel 7 wird die Arbeit zusammengefasst, wichtige Ergebnisse herausgearbeitet und weiterführende Forschungsfragestellungen diskutiert.

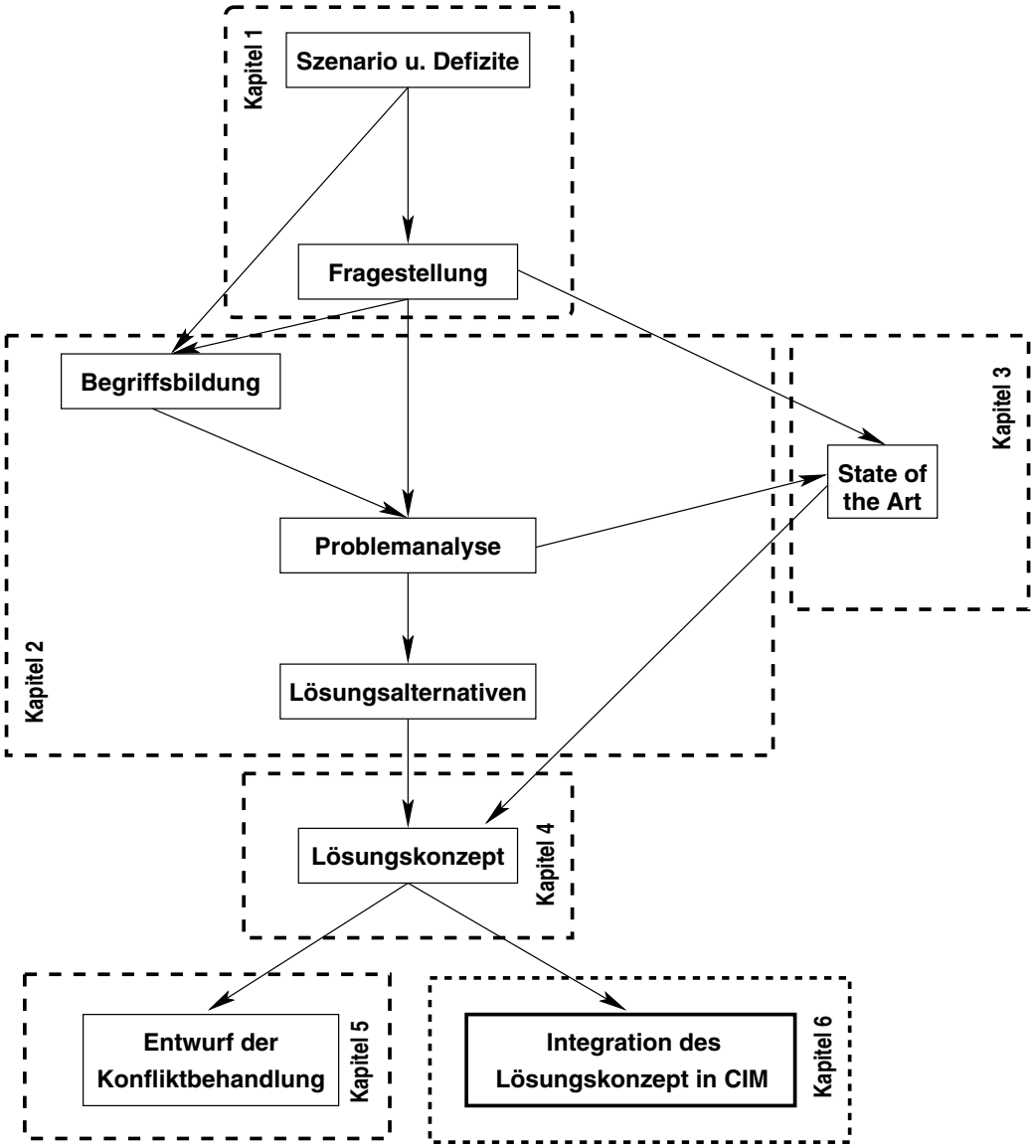

Abbildung 1.3: Vorgehensmodell

1.4 Ergebnisse der Arbeit

Diese Arbeit zeigt, dass unter Berücksichtigung von Managementmodellen eine Reihe von Konflikten existieren, die bis jetzt in der Literatur nicht behandelt werden. Als wesentliche neue Konfliktart werden Konflikte in Beziehungen von Managementobjekten nachgewiesen.

Da in der Literatur keine Vorgehensweise existiert, um methodisch Policy–Konflikte zu behandeln, wird in dieser Arbeit ein allgemeine Methodik entwickelt. Die Anwendbarkeit der Methodik wird anhand eines Beziehungsmodells (für funktionale Abhängigkeiten sowie Enthaltenseinsrelationen) und für einen Vertreter von dynamischen Modellen, den endlichen Automaten vorgeführt. Diese Modelle werden als a priori Modelle aufgefasst. Für jede dieser Anwendungsfälle lassen sich systematisch Konfliktdefinitionen und Vorbedingungen ableiten. Damit sind in dieser Arbeit Policy–Konflikte analysiert und gelöst worden, die bis jetzt in der Literatur nicht betrachtet wurden.

Der entwickelte Algorithmus zur Konfliktbehandlung ist ein effizienter, in weiten Teilen parallelisierbarer Algorithmus, der durch Teilmengenbildung die Anzahl der zu betrachtenden Policies schnell reduziert. Es wurden Strategien zur Konflikterkennung und Konfliktlösung entwickelt, die individuell auf die betrachteten Konfliktarten abgestimmt sind. Der Algorithmus kann sowohl präventiv, in der Phase der Policy–Spezifikation als auch reaktiv, in der Phase der Policy–Ausführung eingesetzt werden.

Damit die Allgemeinheit des Lösungsansatzes erreicht werden kann, sind wichtige Designziele: die Unabhängigkeit von einer dedizierten Policy–Sprache, die Uneingeschränktheit der behandelbaren Konfliktarten sowie die Unabhängigkeit von einem spezifischen Managementinformationsmodell .

Am Beispiel des Common Information Model wird gezeigt, dass sich das generische Beziehungsmodell und die Konfliktdefinitionen in ein existierendes Management–Informationsmodell einfach integrieren lassen und die Methodik dort ohne Modifikation anwendbar ist.

Kapitel 2

Problemanalyse

Kapitelüberblick

2.1 Einleitung

Dieses Kapitel stellt als erstes die wesentlichen Konzepte des policy–basierten Managements vor (siehe Abschnitt 2.2). Dazu zählen u.a. Policy–Architektur, Policy–Hierarchien sowie Policy–Sprachen. Damit wird das Fundament für die spätere Analyse gelegt. Im Anschluss daran werden wichtige Begriffe im Konfliktumfeld definiert (siehe Abschnitt 2.3). Aufgrund der Managementdisziplinen wird ein formales Konfliktmodell entwickelt (siehe Abschnitt 2.4), das als Basis für die anschließende Analyse der Policy–Konflikte dient (siehe Abschnitt 2.5). Die jeweiligen Analyseschritte werden exemplarisch anhand von Beispielpolicies durchgeführt.

2.2 Policy–basiertes Management

Dieser Abschnitt führt in das policy-basierte Management ein. Da die Konfliktbehandlung auf ein möglichst breites Fundament gestellt werden soll, werden im Folgenden allgemeingültige Konzepte des policy-basierten Managements vorgestellt.

2.2.1 Policy–Architektur

Die wesentlichen konzeptuellen Komponenten des policy-basierten Managements zur Verarbeitung von Policies sind nach [RFC 3060, Verm 00]:

Policy Repository : ist die Verwaltungskomponente für alle Policies und dient somit als Wissensbasis für das Management. In ihre werden die Policies persistent gespeichert.

Policy Decision Point (PDP) : entscheidet, ob eine Policy ausgeführt werden soll. Dazu wird der Condition-Teil der Policy ausgewertet. Es existiert in den meisten Ansätzen ein zentraler PDP innerhalb einer Policy Architektur.

Policy Enforcement Point (PEP) : führt Policies aus. Der PEP muss den Action-Teil der Policies auf das Managed System umsetzen. Der PEP ist die homogene Schnittstelle für die Policy–Spezifikation. Es verschattet die Heterogenität der unterliegenden Ressourcen.

Abbildung 2.1 zeigt den Zusammenhang der Komponenten einer Policy–Architektur.

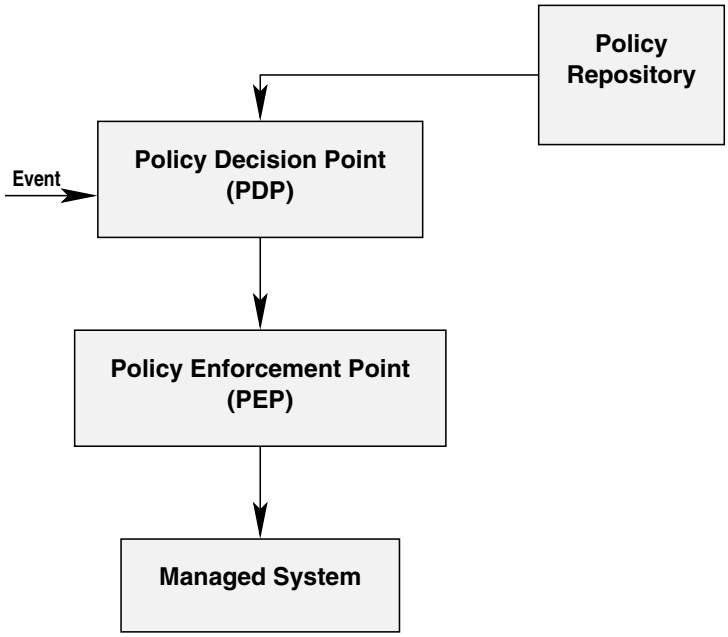

Abbildung 2.1: Allgemeine Policy-Architektur angelehnt an [RFC 3060]

Zur Kommunikation zwischen PDP und PEP wird von der Internet Engineering Task Force (IETF) Network Working Group das Common Open Policy Service (COPS) Protocol [RFC 2748] vorgeschlagen.

Weiterführende Ansätze wie beispielsweise [DLSD 01] unterstützen eine Verteilung von PDPs, um einen effiziente Auswertung von Policies in großen IT–Umgebungen zu gewährleisten. Ebenso ist möglich, dass mehrere PEPs existieren, beispielsweise könnte jedes Koppelelement (Router, Switch, etc.) einen PEP besitzen. Daraus folgt, dass der PDP zusätzlich entscheiden muss, welcher PEP die ausgewertete Policy ausführen soll. Dies geschieht anhand der spezifizierten Target-Objekte der Policy (siehe Abschnitt 2.2.3).

Es existieren Policy–Architekturen in unterschiedlichen Ausprägungen, was beispielsweise die Art der Aktivierung von Policy betrifft. Für die vorliegende Arbeit werden folgende charakteristischen Eigenschaften einer policy–basierten Architektur angenommen:

Ereignisgesteuert: wie oben bei der Verarbeitung von Policies beschrieben, reagieren die in dieser Arbeit betrachteten policy-basierten Managementsysteme nur auf Ereignisse.

Parallele Ausführbarkeit von Policies: in jeder operationalen Policy wird

ein Ereignis spezifiziert, wann sie ausgeführt werden soll. Damit ist es möglich, dass zu einem Ereignis u. U. eine Menge von Policies feuern können. Ist dies der Fall, so können diese Policies parallel ausgeführt werden und unterliegen keiner Abarbeitungsreihenfolge.

Verteilung: es kann ein beliebige Anzahl von PEPs existieren. Damit können Policies echt verteilt abgearbeitet werden.

Gedächtnislosigkeit des PDPs: die Auswertung des Condition–Teils einer Policy erfolgt nur unter Berücksichtigung der booleschen Terme, die im Condition–Teil enthalten sind. Ob und welche Policies vorher (oder anschließend) ausgeführt wurden spielt dabei keine Rolle.

Da es in großen Managementumgebungen nicht praktikabel ist für jedes Managementobjekt eine dedizierte Policy zu spezifizieren, ist es notwendig, die Adressierung einer Menge von Managementobjekten zu ermöglichen. Die beiden wichtigsten Strukturierungmechanismen im Managementumfeld sind Domänen und Rollen. Diese werden nun vorgestellt.

Domänen Eine Domäne ist Menge von Managementobjekten, die aufgrund von charakteristischen Aspekten gebildet wird [SlTw 94]. Mögliche Aspekte sind der Typ des Managementobjekts (Router, Printer), organisatorische Einteilung (IT-Abteilung, Verkaufsabteilung) oder geographische Zuordnung (3. Stock, alle bayerischen Niederlassungen). Zur weiteren hierarchischen Strukturierung ist die Möglichkeit der Subdomänenbildung gegeben (alle Access Router aus der Domäne der Router). Wird ein Managementobjekt in eine andere Domäne verschoben, so werden automatisch die Policies, die an diese Domäne gebunden sind, für das Managementobjekt aktiv.

Rollen Eine Rolle definiert die Funktion eines Akteurs unabhängig von einer konkreten Instanz. Die Funktion fasst Aufgaben oder Verantwortlichkeiten einer Rolle zusammen. Beispielsweise kann die Rolle 'Sicherheitsbeauftragter' die Aufgabe haben, alle Firewall-Regeln zu spezifizieren und die Root-Passwörter monatlich zu ändern. Ein Akteur kann gleichzeitig in mehreren Rollen auftreten und eine natürliche Person oder ein Agent sein.

2.2.2 Policy–Hierarchie

Ein wichtiges Merkmal von Policies ist die Möglichkeit, Managementanweisung in verschiedenen Abstraktionsniveaus und Detaillierungsgraden ausdrücken zu

können. Damit eignen sich Policies prinzipiell für alle Managementdisziplinen wie sie auf der rechten Seite der Abbildung 2.2 gegenübergestellt sind. Da jede Managementdisziplin einer Ebene Vorgaben für die darunterliegende Managementdisziplin macht, ergibt sich für die korrespondierenden Policies eine hierarchische Struktur [Wies 95, Koch 96]. Dabei werden die Policies einer niedereren Ebene durch Verfeinerung von Policies der höheren Ebene gewonnen. Verfeinerung einer Policy bedeutet, dass die betroffenen Entitäten bzw. die auszuführenden Aktionen präzisiert oder weiter eingegrenzt werden.

Corporate bzw. *High-Level Policies* kann man dem Business und Customer Management zuordnen. Diese Policies werden direkt von den allgemeinen Unternehmenszielen abgeleitet und haben im Allgemeinen eine strategische Ausrichtung. Diese Art der Policies haben normalerweise einen geringen Detailgrad und technologische Aspekte haben kaum Bedeutung. Diese Policies sind meist nicht formalisiert, sondern werden üblicherweise informell in Prosa beschrieben.

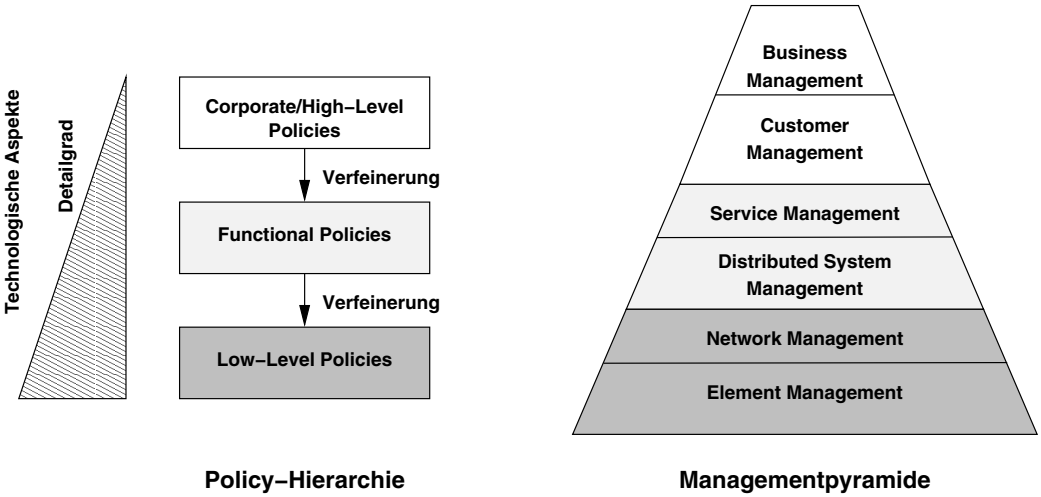

Abbildung 2.2: Policy-Hierarchie angelehnt an [Wies 95] und korrespondierende Managementpyramide nach [HAN 99a]

Functional Policies sind eine Verfeinerung der Corporate Policies und sind auf der Ebene des Service und Distributed System Management anzusiedeln. Policies dieser Ebene operieren auf Managementplattformen und nutzen die dort angebotenen Dienste [Wies 95]. Beispiele für diese Dienste sind CORBA Services der Object Management Group [OMG 98-12-09] oder die OSI Systems Management Functions der ISO [ISO 10164-x]. Policies dieser Ebene haben bereits eine formale Syntax und sind ausführbar. Ab der Ebene der Functional Policies werden die Ziele durch Aktionen repräsentiert, das heißt Ziele sind nicht explizit in einer

Policy ausgedrückt.

Die unterste Stufe der Hierarchie bilden die Low-Level Policies, die dem Network und Element Management zuzuordnen sind. Wesentlich für diese Stufe ist, dass die Policies direkt auf Managementobjekten oder Mengen von Managementobjekten operieren. Dabei werden als Aktionen Methoden der Managementobjekte, die an der Managed Object Boundary angeboten werden, aufgerufen.

Diese Aufteilung der Policies in die Hierarchie wird in Abschnitt 2.4 zur Strukturierung des formalen Konfliktmodells wieder aufgegriffen.

2.2.3 Policy–Sprachen

Aus den wesentlichen Arbeiten im Bereich des policy-basierten Managements ([DDLS 00, Radi 02d, LBN 99, DSP 0108b, RFC 3060] und auch beschrieben in [DaKe 04]) lässt folgende allgemeine Form einer Policy ableiten:

Subject Legt fest, wer die Policy ausführt und gibt somit die Verantwortlichkeit an. Das Subject wird häufig als Rolle spezifiziert.

Target Legt die Zielobjekte fest, die von Policy betroffen sind. Die Zielobjekte werden häufig als Domäne spezifiziert.

Event Legt das Ereignis fest, wann die Policy ausgewertet wird. Der Ereignisbegriff wird dabei in der vorliegenden Arbeit sehr allgemein gefasst. Beispielsweise kann ein Zeitpunkt (12:00:00 Uhr) ein Ereignis darstellen.

Action Legt die auszuführende Aktion auf dem Target–Teil fest.

Condition Legt eine Bedingung fest, ob die Policy ausgeführt wird. Der Condition–Teil kann damit als Vorbedingung angesehen werden. Existiert kein Condition–Teil, so wird die Policy beim Eintreten des Events immer ausgeführt.

Wichtige Vertreter von Policy–Sprachen für das IT–Management, wie Ponder [DDLS 00] und ProPoliS [DaKe 04] halten sich an diese Form.

Im Folgenden wird die Ausdrucksmächtigkeit der einzelnen Policy–Teile diskutiert:
Subject und Target einer Policy kann als ein einzelnes Objekt, eine Domäne oder als Rolle spezifiziert sein. Die Ausdrucksmächtigkeit von Domänen und Rollen wurde in Abschnitt 2.2.1 beschrieben. Der Condition–Teil ist ein logischer Ausdruck, der oft als Normalform angegeben ist, und besitzt somit die Ausdrucksmächtigkeit von regulären Ausdrücken. Der Action–Teil wird im Allgemeinen nicht als Teil der Policy–Sprache spezifiziert, denn die dort definierten Aktionen sind Operationsaufrufe der Managed Object Boundary. Ein Ereignis kann neben dem Ereignisnamen auch noch weitere Information tragen. Diese am COR-

BA Notification Service [OMG 02-08-04] angelehnte Konzept wird *rich event* genannt. Folgende Beispielpolicy ist in Ponder spezifiziert und leicht abgewandelt [Dami 02] entnommen:

```
inst oblig printFail
{
  on      printfail(jobID, userID);
  subject s = printManager;
  target  ms = /servers/mailServer;
  do      ms.mailto(userID, "printing job:"
                    +jobid+" failed.");
}
```

Die Policy mit dem Namen `printFail` ist eine Verpflichtungspolicy (`oblig`). Wenn das Ereignis `printfail()` (mit konkreter Wertebelegung der beiden Parameter `jobID, userID`) eintritt, so wird eine Email mit Fehlermeldung an den Benutzer `userID` gesendet. Der Mailserver wird aus der Domäne `/servers/mailServer` entnommen. Der Event–Teil trägt Informationen, die innerhalb einer Policy adressiert werden können, wie im Beispiel zu sehen ist.

2.2.4 Policy–Lebenszyklus und Zustandsmodell

Jeder Policy kann ein Lebenszyklus zugeordnet werden. In Anlehnung an [Wies 95] wird der folgende Policy–Lebenszyklus für diese Arbeit zu Grunde gelegt:

- Spezifikation: der Lebenszyklus einer Policy beginnt mit der Spezifikation derselben.
- Aktiv: die fertiggestellte Policy wird in einem Repository gespeichert und aktiviert, das heißt zur Ausführung freigegeben.
- Inaktiv: die Policy darf nicht ausgeführt werden.
- Test: der Policy Decision Point wertet den Condition–Teil aus.
- Ausführung: die Policy wird von einem Policy Enforcement Point ausgeführt.
- Gelöscht: die Policy ist gelöscht.

Abbildung 2.3 repräsentiert den Policy–Lebenszyklus durch ein Zustandsmodell. Der Policy–Lebenszyklus ist nicht streng linear, sondern enthält Zyklen. Der erste Zyklus betrifft die Änderung einer Policy. Muss eine Policy geändert werden, so wird die Policy wieder in die Spezifikationsphase überführt. Dort werden die Änderungen vorgenommen. Der zweite Zyklus betrifft die Beendigung der Ausführungsphase. Nach Beendigung der Ausführungsphase geht die Policy wieder in den Zustand *Aktiv* über.

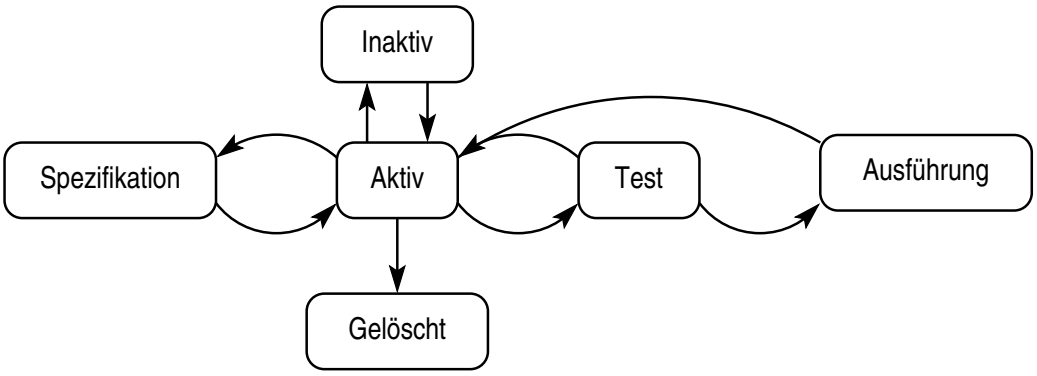

Abbildung 2.3: Policy–Lebenszyklus als Zustandsdiagramm

2.3 Begriffsbildung im Konfliktumfeld

Zur Analyse von Policy–Konflikten ist es notwendig, die wesentlichen Begriffe im Konfliktumfeld exakt zu definieren. Neben dem eigentlichen Konfliktbegriff müssen die Phasen der Konfliktbehandlung sowie die einzuhaltenden Zielen definiert werden.

Ziel — *Ein gewünschtes Verhalten eines Systems, das erreicht werden soll.*

Das Ziel muss sich in messbaren Zielgrößen oder Invarianten ausdrücken lassen. Ziele sind beispielsweise:
Ziel 1: Die Datenschutzbestimmungen müssen eingehalten werden,
Ziel 2: Die Verfügbarkeit des Webservers muss mindestens 98 Prozent betragen,
Ziel 3: Die Verzögerung des Voice over IP Dienstes darf höchstens 100 Millisekunden betragen.
Ziele sind in Policies entweder wie in den Beispielen explizit angegeben, oder implizit durch Aktionen repräsentiert (siehe Abschnitt 2.2.3). Beispielsweise ist

in der Policy *Setze die Zugriffsrechte für einen Email–Account so, dass nur der Empfänger darauf zugreifen kann* das Ziel 1 implizit enthalten.

In Abschnitt 1.1.2 wurde aufgezeigt, dass Policy–Konflikte existieren, die bis jetzt in der Literatur noch nicht betrachtet wurden. Deshalb werden an dieser Stelle der Arbeit Konfliktdefinitionen herangezogen, die möglichst allgemein gehalten sind und eine breite Akzeptanz besitzen. Im Folgenden werden zwei für diese Arbeit wichtige Definition genauer untersucht: In [Wies 95] werden zwei Arten von Policy–Konflikten definiert:

> *„ Policy conflicts are conflicts that are either defined whithin the policies, ..., trying to achieve contrasting goals,...,*
> *or they are caused or uncovered during transformation process when high level abstract policies (with no obvious conflict) are refined and split into smaller less complex policies that have conflicting goals. “*

Der erste Teil der Definition bezieht sich auf gegensätzliche Ziele von Policies. Diese Konfliktdefinition ist für High–Level Policies, wie sie in Abschnitt 2.2.2 beschrieben wurden, nützlich. Allerdings führt die Festlegung auf konträre Ziele zu einer Einschränkung der Allgemeinheit, da Policies das gleiche Ziel verfolgen können (z.B. exklusive Nutzung einer Ressource) und sich trotzdem in Konflikt zueinander befinden. Der zweite Teil der Definition bezieht sich auf den Verfeinerungsprozess von Policies. Konflikte, die durch den Verfeinerungsprozess von Policies entstehen, werden in dieser Arbeit nicht explizit betrachtet. Dieses Problem wird in [BLR 03] eingehend untersucht, eine Lösung ist in diesem Bereich allerdings noch nicht abzusehen.

In [RFC 3198] wird ein Policy–Konflikt definiert als:

> *„Occurs when the actions of two rules (that are both satisfied simultaneously) contradict each other. The entity implementing the policy would not be able to determine which action to perform. The implementers of policy systems must provide conflict detection and avoidance or resolution mechanisms to prevent this situation. “*

Diese Konfliktdefinition geht davon aus, dass *Aktionen* von Policies der wesentliche Ausgangspunkt eines Konflikts sind und somit der Ebene der Functional und Low-Level Policies zuzuordnen. In der vorliegenden Arbeit wird diese Sichtweise für operationale Policies übernommen (siehe Abschnitt 2.5.4).

In den Arbeiten des Imperial College [Dami 02, LuSl 99, MoSl 93] wird ein Policy–Konflikt nur hinsichtlich einer konkreten Konfliktart — dem Modalitätskonflikt — definiert (siehe Abschnitt 3.2.2) und kann deshalb nicht zu einer allgemeinen Konfliktdefinition für diese Arbeit beitragen. Die allgemeine Konfliktdefinition für diese Arbeit lautet:

Konflikt *Sind mindestens zwei Ziele, die durch Policies repräsentiert sind, nicht gleichzeitig und gemeinsam erreichbar, so ist ein Konflikt aufgetreten.*

Diese Definition ist allgemeiner als die beiden zitierten Definitionen, da sie unabhängig von der Policy–Hierarchie ist bzw. die Ziele beliebig zueinander in Beziehung stehen können. Stehen genau zwei Policies in Konflikt, so ist diese Beziehung reflexiv. Entfernt man eine Policy, so existiert der Konflikt nicht mehr. Diese Definition ist allgemeiner gefasst als die Definitionen in [MoSl 93, LuSl 99]. Jene Definitionen gehen von der notwendigen Bedingung der Domänenüberlappung von Subject oder Target aus für Konflikten aus. Dies trifft nicht für jede Konfliktklasse, die in dieser Arbeit behandelt wird zu.

Situation *alle Policies, die gleichzeitig ausgeführt werden, befinden sich in einer Situation*

Mit dem Situationsbegriff, wird eine Menge von Policies charakterisiert, die sich im Falle von operationalen Policies auf das gleiche Event triggern.

Deterministischer Konflikt *ist ein Konflikt, der bei Ausführung der Policies immer eintreten wird.*

Ein Konflikt, bei dem die Ausführungsreihenfolge von Policies für das Auftreten des Konflikts keine Rolle spielt, ist ein deterministischer Konflikt.

Potentieller Konflikt *ist ein Konflikt, der eintreten kann, aber nicht notwendigerweise auch eintreten muss.*

Ein Konflikt, bei dem die Ausführungsreihenfolge von Policies für das Auftreten des Konflikts eine Rolle spielt, ist beispielsweise ein potentieller Konflikt.

Konfliktbehandlung *besteht aus den drei Phasen Konfliktlokalisierung, Konflikterkennung und Konfliktlösung.*

Die Konfliktbehandlung stellt den umfassenden Algorithmus dar, der sich mit Policy–Konflikten befasst.

Präventive Konfliktbehandlung *findet in der Phase der Policy–Spezifikation statt.*

Die präventive Konfliktbehandlung soll den Policy–Ersteller unterstützen, indem eine neu spezifizierte Policies getestet werden kann, ob ein Konflikt mit anderen Policies vorliegt.

Reaktive Konfliktbehandlung *findet in der Phase der Policy–Ausführung statt.*

In dieser Phase kann ein Konflikt tatsächlich auftreten. Die reaktive Konfliktbehandlung ist Teil der Policy–Architektur und wird vom Policy Decision Point ausgeführt.

Konfliktlokalisierung *die Eingrenzung der Menge der zu untersuchenden Policies bei einem (potentiellen) Konfliktfall*

Die wichtigste Aufgabe der Konfliktlokalisierung ist es effizient die Menge von Policies zu minimieren, die in einem (potentiellen) Konfliktfall zu untersuchen sind.

Konflikterkennung *die Feststellung des Eintritts eines Konflikts*

Die Konflikterkennung schließt an die Phase der Konfliktlokalisierung an. In diesem Schritt muss entschieden werden, ob ein Konflikt in einer Menge von Policies vorliegt oder nicht. Dazu müssen auswertbare Kriterien existieren um diese Entscheidung treffen zu können.

Konfliktlösung *die Auflösung eines Konflikts*

Es existieren vier grundsätzliche Strategien, wie mit der Menge der in Konflikt befindlichen Policies in der reaktiven Konfliktlösung verfahren wird:

- Alle ausführen: der Konflikt wird ignoriert, alle Policies werden ausgeführt.
- Alle verwerfen: alle an einem Konflikt beteiligten Policies werden nicht ausgeführt.
- Eine ausführen: die Policy mit der höchsten Priorität wird ausgeführt. Priorisierung ist die am häufigsten in der Literatur angewandte Konfliktlösungsstrategie (siehe Kapitel 3).
- aus der Menge der in Konflikt befindlichen Policies wird ein größtmögliche Teilmenge ausgewählt, die konfliktfrei ausführbar ist.

2.4 Formales Konfliktmodell anhand eines abstrahierten Szenarios

Die in Kapitel 1.1 dargestellte Problematik und die im vorigen Abschnitt eingeführten Begrifflichkeiten werden nun strukturiert und die Beziehungen zueinander explizit mit Hilfe eines UML Klassendiagramm [uml 03] dargestellt. Dieses Diagramm dient im weiteren Verlauf der Arbeit als Basis für die Problemanalyse.

Abbildung 2.4 zeigt das formale Konfliktmodell, welchen in drei Domänen unterteilt ist. Die oberste Domäne, die Unternehmensdomäne wird den Managementdiziplinen Business– und Customer–Management gemäß der Managementpyramide aus Abbildung 1.1 zugeordnet. Die mittlere Domäne zeigt das technische Management und repräsentiert die vier Managementdiziplinen (Service, Distributed System, Network sowie Element–Management) der Managementpyramide. Die unterste Domäne stellt die Modellsicht des Managements dar. Die drei Domänen spiegeln neben der Aufteilung des Managements auch die Aufteilung der Policies in eine Hierarchie (siehe Abschnitt 2.2.2) und die Aufteilung der Konfliktbehandlungsansätze wider (wie in Kapitel 3 gezeigt wird). Die einzelnen Domänen werden nun ausgehend von der Unternehmensdomäne vorgestellt.

Aufgaben der Domänen

Der wesentliche Aufgabe des Unternehmensdomäne für diese Arbeit ist die Definition der Unternehmensziele. Diese Ziele werden in High–Level Policies ausgedrückt. Diese High–Level Policies stellen die Basis für die durchzusetzenden Ziele für die Domäne des technischen Managements dar.

Das technische Management wird durch ein policy–basiertes Managementsystem realisiert. Das heißt, alle wesentlichen Managementvorgänge werden von Policies gesteuert. Die High–Level Policies werden in der Domäne des technischen Managements verfeinert bzw. konkretisiert, bis eine ausführbare, operationale Policy spezifiziert ist. Operationale Policies sind dadurch gekennzeichnet, dass im Action–Teil Managementoperationen enthalten sind, die auf einer Menge von Managementobjekten ausgeführt werden können.

Die unterste Domäne zeigt die Modelldomäne, auf der das policy-basierte Management operiert. Das technische Management manipuliert nicht direkt die Ressourcen, sondern greift auf die Managementobjekte, die eine Abstraktion aus Managementsicht der Ressourcen darstellen, zu. Die Managementobjekte selbst und die Beziehungen der Managementobjekte zueinander, sind in Managementmodellen eingebettet. Typische Managementmodelle sind der Enthaltenseinsbaum, der den sogenannten Management Information Tree (MIT) eines Managed Systems

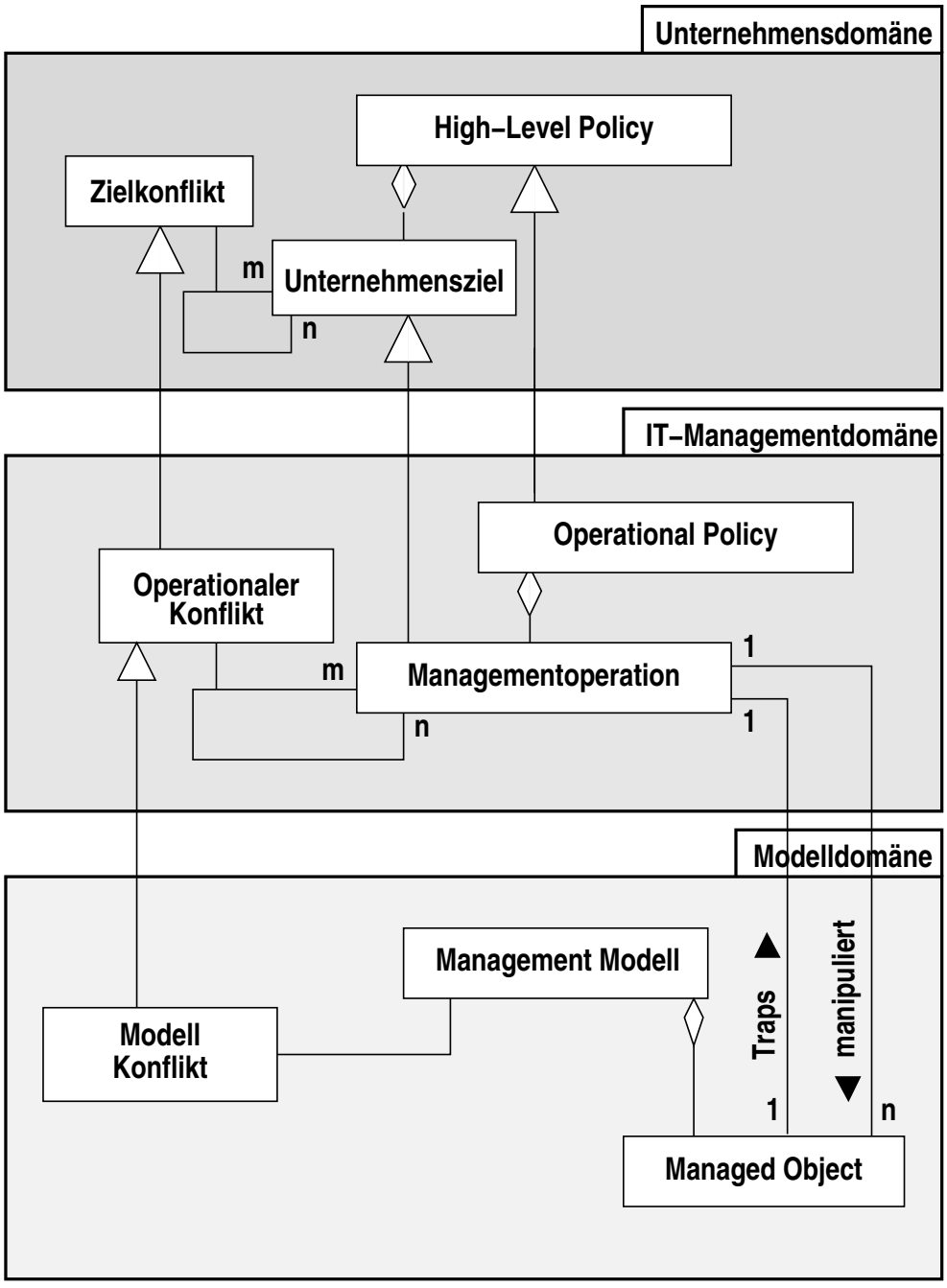

Abbildung 2.4: Formales Konfliktmodell mit den wesentlichen Beziehungen

repräsentiert, oder Modelle aus dem Bereich des Funktionsmodells wie beispielsweise Zustandsmodelle oder Dienstabhängigkeitsgraphen.

Modellierung der Konflikte

Die linke Seite des Konfliktmodells (Abbildung 2.4) modelliert Konflikte und ihre wesentlichen Zusammenhänge. In der Unternehmensdomäne sind Unternehmensziele als High–Level Policies spezifiziert. Divergente Unternehmensziele stehen in Konflikt zueinander und stellen somit einen Zielkonflikt dar. Mit dem Verfeinerungsprozess der High–Level Policies zu operationalen Policies werden der Zielkonflikte mit in die operationalen Policies übertragen.

Der Übergang von High–Level Policies zu Operationalen Policies ist auch dadurch gekennzeichnet, dass bei Operationalen Policies Ziele nicht explizit spezifiziert sind, sondern in Aktionen umgewandelt worden sind. Die wesentliche Eigenschaft der Policies dieser Domäne ist es, dass Managementoperationen ausgeführt werden. In dieser Domäne sind die Managementoperationen in Konflikt zueinander. Somit kann man den Konflikt dieser Domäne als operationalen Konflikt charakterisieren.

Mit der Ausführung der operationalen Policies manifestiert sich der Konflikt schließlich in der Modelldomäne. Operationale Policies manipulieren über Managementoperation Managementobjekte. Die Managementobjekte sind wiederum in Managementmodellen eingebettet. Das heißt, dass sich ein operationaler Konflikt in den Managementmodellen niederschlägt. Dieser wird als Modellkonflikt bezeichnet.

Betrachtet man die drei Domänen im Zusammenhang, so wird ein Konflikt von der obersten Domäne bis in die unterste Domäne weitergereicht bzw. vererbt.

Die genaue Analyse der verschiedenen Konfliktformen für jede Domäne wird im folgenden Abschnitt dargestellt.

2.5 Analyse von Konflikten anhand des Konfliktmodells

Das Konfliktmodell aus dem letzten Abschnitt hat gezeigt, dass Konflikte in unterschiedlichen Domänen existieren. In jeder Domänen zeigt sich ein Konflikt in einer unterschiedlicher Ausprägung (Zielkonflikt, operationaler Konflikt und Modellkonflikt) und jede Domäne besitzt unterschiedliche Informationsquellen (Ziele, Managementoperationen, Managementmodelle), um einen Konflikt behandeln

zu können. Deswegen wird jede Domäne für sich auf mögliche Konfliktbehandlungsansätze hin untersucht.

Policies werden hinsichtlich ihres Konfliktverhaltens für jede Domäne des formalisierten Konfliktmodells systematisch untersucht. Dazu werden konkrete Policies aus dem Bereich des Sicherheits– und Leistungsmanagement als Untersuchungsgegenstand herangezogen. Zuerst werden High–Level–Policies und davon abgeleitete operationale Policies, wie in Abschnitt 2.4 dargestellt, beschrieben. Anschließend werden die Policies hinsichtlich ihres Konfliktverhaltens untersucht. Dazu wird das formalisierte Modell aus Abbildung 2.4 herangezogen, um eine systematische Analyse vorzunehmen.

2.5.1 Beispiel–Policies aus dem Sicherheits– und Performancemanagement

Die Policies in diesem Abschnitt werden in einer leicht lesbaren und allgemein gehaltenen Syntax, wie in Abschnitt 2.2.3 dargestellt, beschrieben. Diese allgemeine Syntax lässt sich auf die gängigen Sprachen im Managementbereich einfach abbilden.

Folgende zwei High–Level Policies (`high-level policy1` und `high-level policy2`) seien gegeben:

Policy	`high-level policy1`
Event	`none`
Subject	`Geschäftsleitung`
Target	`Alle kritischen Systeme`
Action	`Zum Erhalt eines hohen Sicherheitsstandard müssen empfohlene Security-Updates des CERT nach Bekanntgabe umgesetzt werden.`

Policy	`high-level policy2`
Event	`none`
Subject	`Geschäftsleitung`
Target	`Alle für den Betriebsablauf notwendigen Dienste`
Action	`Dienst müssen jederzeit während den Arbeitszeiten verfügbar sein.`

Die erste Policy (`high-level policy1`) aus dem Bereich des Sicherheitsmanagement verfolgt das Ziel, einen möglichst hohen Sicherheitsstandard in der IT–Infrastruktur der Firma zu etablieren. Deshalb müssen den Empfehlungen des CERT Folge geleistet werden und Sicherheits–Updates auf sicherheitskritischen Systemen umgesetzt werden.

Die zweite Policy (`high-level policy2`) aus dem Bereich des Leistungsmanagements verfolgt das Ziel, dass alle für das Unternehmen wichtige Dienste während den Arbeitszeiten verfügbar sein müssen. Deshalb stehen den Mitarbeiter alle notwendigen IT–Dienste während der Arbeitszeiten zur Verfügung.

Beide Policies zeichnen sich dadurch aus, dass sowohl der Target– als auch der Action–Teil der Policy nicht exakt spezifiziert ist. Die exakte Ausformulierung wird den entsprechenden Fachabteilungen überlassen. Auch werden keine Ereignisse spezifiziert, wann die Policy ausgeführt werden soll. Policies dieser Domäne besitzen meist eine Allgemeingültigkeit und werden in diesem Sinne immer ausgeführt.

Aus diesen beiden High–Level–Policies werden drei operationale Policies abgeleitet:

Policy	`operational policy1.1`
Subject	`Security Manager`
Event	`beginOfWork`
Target	`/resource/hardware/server`
Action	`lock(); update(secManager.getPackage());` `reboot(); unlock();`
Condition	`secManager.updateAvailable();`

Die Policy `operational policy1.1` wurde aus der Sicherheits–Policy `high-level policy1` abgeleitet. Wenn das Ereignis `beginOfWork` eintritt, wird getestet, ob ein neues Update vorhanden ist, das auf allen Servern der Firma (repräsentiert durch die Domäne `/resource/hardware/server`) eingespielt wird. Zum eigentlichen Update müssen eine Reihe von Aktionen ausgeführt werden: Zuerst muss der Server zur Nutzung gesperrt (`lock();`) und das Update eingespielt (`update(swPackage);`) werden. Anschließend muss der Server gebootet und zur Nutzung freigegeben werden (`reboot(); unlock();`).

Die beiden folgenden Policies leiten sich aus der `high-level policy2` ab:

Policy	`operational policy2.1`
Event	`beginOfWork`
Subject	`Performance Manager`
Target	`/resource/services/work`
Action	`unlock();`

Policy	`operational policy2.2`
Event	`endOfWork`
Subject	`Performance Manager`
Target	`/resource/services/work`
Action	`lock();`

Die Policy `operational policy2.1` gibt alle Dienste der Domäne `/resource/services/work` zu Beginn der Arbeitszeit (angezeigt durch das Ereignis `beginOfWork`) zur Nutzung durch die Aktion `unlock()` frei. Die Policy `operational policy2.2` ist der Gegenpart und sperrt alle Dienste der Domäne `/resource/services/work` am Ende der Arbeitszeit (angezeigt durch das Ereignis `endOfWork`) durch die Aktion `lock()`.

2.5.2 Konfliktanalyse in der Unternehmensdomäne

High–Level Policies werden auf Unternehmensebene spezifiziert. Dementsprechend hoch ist das Abstraktionsniveau in dem die Ziele formuliert werden. Typische Eigenschaften von Zielen sind, dass sie meist *grobgranular*, *nicht technisch* und *in Prosa* angegeben sind [KKK 96]. Ebenso ist *Zeitraum der Gültigkeit* eines Zieles meist nicht exakt spezifiziert. Die Grobgranularität kann sich sowohl auf die Beschreibung eines Ziels selbst als auch die vom Ziel betroffenen Objekte beziehen. Die Konfliktdefinition aus Abschnitt 2.3 wird für die Unternehmensdomäne angepasst und lautet:

Zielkonflikt *sind mindestens zwei Ziele, die durch Policies repräsentiert sind, nicht gleichzeitig und gemeinsam erreichbar, so ist ein Konflikt aufgetreten.*

Das zweite Problem neben der Schwierigkeit der exakten Spezifikation von Zielen ist die Anforderung der gemeinsamen Erreichbarkeit von Zielen zu bestimmen. Ob ein Ziel erreichbar ist, hängt im Allgemeinen von der gegebenen Infrastruktur ab. Dasselbe gilt auch für das Problem der gemeinsamen Erreichbarkeit.

Mit der Formalisierung von Zielen beschäftigt sich das Goal–oriented Requirements Engineering [LAMS 01] (siehe Abschnitt 3.3.1). Dort wird für den Bereich des Software Engineering versucht, durch eine formale Darstellung von Zielen auf untergeordnete Ziele zu schließen und auch Konflikte zu erkennen. Da aber auf Unternehmensebene die Formalisierung von Zielen schwer fällt (u.a. durch den nicht technische Fokus derjenigen, welche die High–Level Policies formulieren), erscheint dem Autor eine Konfliktbehandlung durch Analyse von Zielen in dieser Domäne als schwierig.

Konfliktanalyse am Beispiel

Auf der obersten Ebene des abstrakten Szenarios (siehe Abbildung 2.4), der Unternehmensebene werden, wie in Abschnitt 2.4 dargestellt, Ziele formuliert und in High–Level–Policies abgelegt. Die zugehörigen Policies im Beispiel sind `high-level policy1` und `high-level policy2`. Policy `policy1` formuliert das Ziel einen hohen Sicherheitsstandard einzuhalten. Policy `policy2` formuliert das Ziel, die betriebsnotwendigen Dienste während der Arbeitszeit zur Verfügung zu stellen.

Stehen diese Policies in Konflikt zueinander? Beide Ziele (*Sicherheit* und *Verfügbarkeit von Diensten*) sind abstrakt und nicht exakt formuliert. Ein Zusammenhang beider Ziele, oder auch die Divergenz beider Ziele ist nicht bestimmbar. Weder die genau betroffenen Entitäten der Policies (*kritische Systeme* und *Dienste*) noch die konkret auszuführenden Aktionen (*Software–Update umsetzen* und *Dienste betreiben*) sind auf diesem Abstraktionsniveau absehbar.

Ergebnis Es ist auf Unternehmensebene in der Regel nicht entscheidbar, ob diese Policies in Konflikt zueinander stehen. Dies liegt zum Einen am Abstraktionsniveau der Aussagen der Policies und zum Anderen sind die konkreten Verfeinerungsschritte hin zu operationalen Policies auf dieser Ebene nicht absehbar. Welche Dienste und Entitäten genau von den Policies betroffen sind, ist nicht direkt erkennbar.

Allgemein kann für High–Level–Policies festgestellt werden, dass sie ein hohes Abstraktionsniveau besitzen und die Formulierung der Policy meist in Prosa vorliegt. Die Ziele werden weder exakt spezifiziert, noch existiert eine Formalisierung derselben. Aus diesen Gründen ist eine Konfliktanalyse auf Unternehmensebene kaum durchführbar, da keine exakte Beziehung der Ziele zueinander hergestellt werden kann.

2.5.3 Konfliktanalyse in der IT–Managementdomäne

Der Verfeinerungsprozess von High–Level Policies zu operationalen Policies bringt im Wesentlichen eine Konkretisierung in drei Bereichen: die abstrakten Ziele werden in *konkrete Managementoperationen* umgewandelt, die von einer Policy *betroffenen Objekte* werden identifiziert (Targets) und es wird ein *Event* spezifiziert, wann eine Policy auszuführen ist.

Diese Konkretisierungen eröffnen neue Möglichkeiten der Konfliktbehandlung. Als erstes wird die Konfliktdefinition aus dem Abschnitt 2.5.2 auf die neue Situation angepasst:

Operationaler Konflikt	*sind mindestens zwei Managementoperationen, die durch Policies repräsentiert sind, nicht gleichzeitig ausführbar, so ist ein Konflikt aufgetreten.*

Das heißt in der IT–Managementdomäne sind Managementoperationen der wesentliche zu untersuchende Faktor. Die Aussage „nicht gleichzeitig ausführbar“ bedeutet, dass sich die Managementoperationen gegenseitig so beeinflussen, dass die zu erzielende Wirkung auf das Managed System nicht mehr gewährleistet ist.

Eine weit verbreitete Möglichkeit, um die Wirkung von Managementoperationen zu beschreiben, ist die Verwendung von Vor– und Nachbedingungen (beispielsweise durch die Object Constraint Language OCL [uml 03]). Damit kann ausgedrückt werden, wie sich der Zustand eines Systems hinsichtlich einer Operation verändert.

Um zu entscheiden, ob Managementoperation zueinander in Konflikt stehen, werden die formalisierten Auswirkungen der Managementoperationen miteinander verglichen.

Da in der IT–Management Domäne nur die Managementoperationen an sich zur Konfliktanalyse zur Verfügung stehen ist es für diese Domäne notwendig, weitere Randbedingungen zu spezifizieren, damit Konflikte erkannt werden können. Um bei einem Vergleich von Managementoperationen tatsächlich Konflikte identifizieren zu können wird die Randbedingungen der Domänenüberlappung bei der Policy–Ausführung (siehe Abschnitt 3.2) eingeführt.

Durch die Spezifikation der Targets als einzelne Objekte oder Domänen kann man auch eine Domänenüberlappung zweier Policies feststellen (siehe Abschnitt 2.2.1). Überlappen sich die Target–Domänen zweier Policies und überführen die Managementoperationen das System in unterschiedliche Zustände (abzulesen an den spezifizierten Vor– und Nachbedingungen) so ist unter der Randbedingung, dass die Policies gleichzeitig ausgeführt werden sollen (also dasselbe Event spezifiziert haben) ein Konflikt eingetreten.

Konfliktanalyse am Beispiel

Die Beispielpolicies auf Managementebene aus Abschnitt 2.5.1 sind:

Policy	`operational policy1.1`
Subject	`Security Manager`
Event	`beginOfWork`
Target	`/resource/hardware/server`
Action	`lock(); update(secManager.getPackage());` `reboot(); unlock();`
Condition	`secManager.updateAvailable();`

Policy	`operational policy2.1`
Event	`beginOfWork`
Subject	`Performance Manager`
Target	`/resource/services/work`
Action	`unlock();`

Policy	`operational policy2.2`
Event	`endOfWork`
Subject	`Performance Manager`
Target	`/resource/services/work`
Action	`lock();`

Operationale Policies zeichnen sich dadurch aus, dass alle Felder soweit konkretisiert sind, dass sie von einem PDP entschieden und einem PEP umgesetzt werden können. Beispielsweise sind alle Targets durch konkrete Domänen angegeben. Die zugehörige Domänenstruktur ist in Abbildung 2.5 dargestellt. Die größte Domäne ist die Domäne `Resource`. Sie ist unterteilt in zwei Unterdomänen `Service` und `Hardware`. Die Domäne `Work` enthält ein Element `database`, die Domäne `Server` enthält den `serverA`.

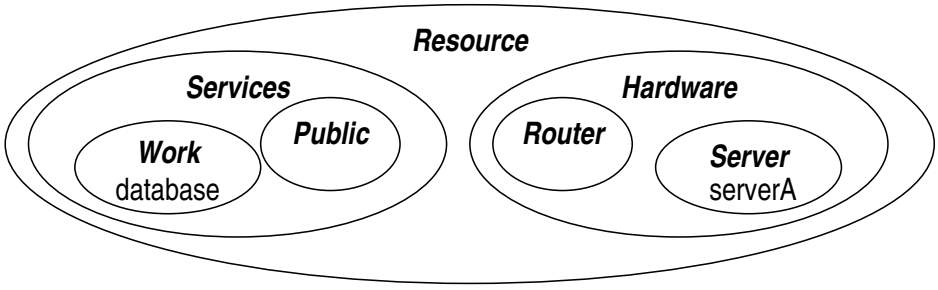

Abbildung 2.5: Domänenstruktur des Beispielszenario

Die Subjects werden durch unterschiedliche Rollen repräsentiert. Für die Policy `operational policy1.1` ist der `Configuration Manager` verantwortlich, während für die beiden Policies `operational policy2.1` und `policy 2.2` der Performance Manager zuständig ist.

Stehen die Policies in Konflikt zueinander? Da auf dieser Ebene drei Policies gegeben sind, existieren drei Paarbeziehungen zwischen den Policies. Diese werden nun einzeln betrachtet.

Konfliktanalyse zwischen Policy `policy1.1` und `policy2.1` Nimmt man die Definition von Konflikten von [MoSl 93, LuSl 99] an, so muss eine Domänenüberlappung vorliegen, damit ein Konflikt eintreten kann (siehe auch Abschnitt 3.2.1). Die Schnittmenge der Domänen `/resource/services/work` und `/resource/hardware/server` ist leer. Ein Domänenüberlappung der Targets von `policy 1.1` und `policy 2.1` liegt somit nicht vor. Die Aktionen werden damit auf unterschiedlichen Ressourcen ausgeführt. Auf dieser Ebene werden die beiden Policy als nicht in Konflikt befindlich eingestuft.

Konfliktanalyse zwischen Policy `policy1.1` und `policy2.2` Bezüglich der Domänenüberlappung gilt dieselbe Aussage wie im vorigen Absatz. Auch hier ist kein Konflikt erkennbar.

Konfliktanalyse zwischen Policy `policy2.1` und `policy2.2` Es existiert eine vollständige Überlappung von Subject und Target beider Policies. Die Aktionen der Policies sind gegensätzlich (`lock()` und `unlock()`). Wenn man davon ausgeht, dass die beiden Ereignisse `beginOfWork` und `endOfWork` zeitlich weit auseinander liegen, so können diese beiden Policies ebenfalls nicht in Konflikt zueinander stehen, da die Bedingung der Gleichzeitigkeit der Policy–Ausführung nicht gewährleistet ist.

Ergebnis Die Policies dieser Ebene sind in allen Teilen exakt spezifiziert und auswertbar. Die Domänenüberlappung ist das wesentliche Merkmal der Konfliktbehandlung in der Literatur. Ausgehend von diesem Merkmal treten in dieser Ebene keine Konflikte auf.

2.5.4 Konfliktanalyse in der Modelldomäne

Die Modelldomäne unterscheidet sich wesentlich von den anderen beiden Domänen, denn es stehen dort durch die vorhandenen Modelle wesentlich mehr Informationsquellen zur Verfügung, Die Managementobjekte sind in unterschiedlichen Modellen eingebettet. Beispiele für solche Modelle sind

- der Enthaltenseinsbaum, wie er beispielsweise im OSI Informationsmodell als Managent Information Tree (MIT) existiert
- Funktionale Abhängigkeitsgraphen, wie sie beispielsweise durch Protokollschichten gegeben sind.
- Dienstabhängigkeitsgraphen [EnKe 01a, Ense 02], wie sie bei Application Service Providern oder in Outsourcing Szenarien vorkommen.

Das Common Information Model (CIM) der Distributed Management Taskforce (DMTF) [CIM 2.2] modelliert zahlreiche Beziehung zwischen Managementobjekten. Dabei werden drei wesentliche Beziehungsarten unterschieden: Assoziationen, Abhängigkeiten und Aggregationen. Alleine im Core–Modell, welches Oberklassen von Managementobjekten für alle Managementbereiche zur Verfügung stellt, werden über 30 verschiedene Assoziationen, über 20 verschiedene Abhängigkeiten und über 20 verschiedene Aggregationen unterschieden. Damit zeigt sich die Wichtigkeit, Beziehungen von Managementobjekten zu betrachten.

Im Hinblick auf die Anwendung von Modellen existieren zwei Anwendungsfälle: Entweder man erfasst mit einem Modell den Istzustand eines Systems oder man nimmt eine Modellierung im vorhinein vor, um damit einen gewünschten Sollzustand eines Systems zu beschreiben. Alle planerischen Modelle besitzen den zweiten Charakter. Dieser Anwendungsfall wird in [Ense 02] als „*a priori* Modellierung“ bezeichnet. Alle Modellierungen in Standards sind typische Vertreter der *a priori* Modellierung. Die *a priori* Modellierung kann als eine Sammlung von Bedingungen interpretiert werden, die von dem zu manageneden System eingehalten werden müssen.

A priori Modelle können im Kontext der Konfliktbehandlung als Ziel aufgefasst werden: Ein a priori Modell muss eingehalten werden, damit das Managed Systems im Sinne des Ziels funktioniert.

Damit werden zwei entscheidende Fortschritte gegenüber den Konfliktbehandlungsansätzen in den anderen Domänen erreicht: Das Ziel (ein a priori Modell) ist bereits formalisiert bzw. lässt sich leicht formalisieren und die Auswirkungen von Managementoperationen lassen sich bezüglich eines Modells relativ einfach darstellen. Daraus wird für die Modelldomäne ein Modellkonflikt definiert.

Modellkonflikt *sind mindestens zwei Managementoperationen, die durch Policies repräsentiert sind, nicht gleichzeitig und gemeinsam ausführbar,* ***da Modellbedingungen von a priori Modellen nicht eingehalten werden****, so ist ein Konflikt aufgetreten.*

Managementoperationen beeinflussen Managementobjekte (in a priori Modellen). Deshalb werden im Folgenden mögliche Konfliktsituationen aufgrund von Managementoperationen aufgezeigt.

Ein Managementobjekt (bei einem objektorientierten Ansatz) besteht aus einer Menge von Attributen und Methoden, die auf den Attributen operieren oder das ganze Objekt betreffen. Der Zustand eines Managementobjekts wird durch die aktuelle Wertebelegung der Attribute gekennzeichnet.

In einer Konfliktsituation, in der mehrere Managementoperationen gleichzeitig auf Attributen operieren, kann es zu folgenden grundlegenden Konfliktfällen kommen:

- Attributkonflikt: dasselbe Attribut einer Instanz soll innerhalb der Situation mehrfach modifiziert werden.
- Interattributkonflikt: verschiedene Attribute einer Instanz sollen in einer Situation modifiziert werden, die zueinander in Beziehung stehen.
- Instanzkonflikt: Attribute verschiedener Instanzen sollen modifiziert werden und die Instanzen stehen in Beziehung zueinander. Im folgenden Abschnitt wird ein Konflikt dieser Art vorgestellt.

Die beiden ersten Konfliktarten sind in [Verm 00] beschrieben. Aus den drei Konfliktarten folgt, dass man die Wirkung der Methoden auf die Attribute spezifizieren muss und die Beziehungen der Attribute und Instanzen zueinander kennen muss, damit eine komplette Konfliktanalyse erfolgen kann.

Der folgende Abschnitt präsentiert ein Beispiel, wie Modellbedingungen zur Konfliktanalyse verwendet werden können. Kapitel 4 stellt eine Methodik auf Basis von Modellbedingungen vor.

Konfliktanalyse am Beispiel

Die Konfliktanalyse auf Modellebene betrachtet die Modelle, in der die Managementobjekte eingebettet sind, die in den Policies spezifiziert sind.

Das Domänenmodell (Abbildung 2.5) wurde bereits auf Policyebene zur Konfliktanalyse herangezogen. Es wird nun je ein Element der Domäne `/resource/services/work` und `/resource/hardware/server` genauer betrachtet.

Es soll angenommen werden, dass von Policy `policy 2.1` und `policy 2.2` eine Datenbank betroffen ist. Das bedeutet, dass sich die Datenbank in der Domäne `/resource/services/work` befindet. Ein Server `serverA` soll von Policy `policy 1.1` betroffen sein. Der `serverA` befindet sich in der Domäne `/resource/hardware/server`. Zusätzlich existiert ein a priori Modell, das eine funktionale Abhängigkeit der Datenbank `database` zu `serverA` repräsentiert, wie in Abbildung 2.6 zu sehen.

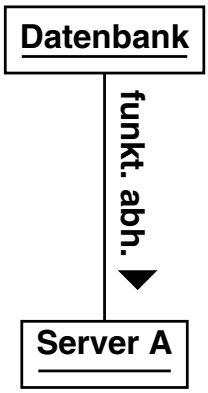

Abbildung 2.6: Modell einer funktionalen Abhängigkeit

Stehen die Policies in Konflikt zueinander? Unter Rücksichtnahme des a priori Modells der funktionalen Abhängigkeit ergibt sich eine neu zu analysierende Situation. Managementoperationen können Beziehungen zwischen Managementobjekten beeinflussen. Funktional abhängig bedeutet, dass die Funktion einer Ressource nur voll erbracht werden kann, wenn die Ressource von der sie abhängt, funktionsfähig und zugreifbar ist.

Policy `policy 1.1` beeinflusst die funktionale Abhängigkeit zwischen `database` und `serverA`, da `Server A` durch die Policy–Aktion `lock()` für die `database` nicht mehr verfügbar ist. Policy `policy 2.1` gibt die `database` durch die Policy–Aktion `unlock()` zur Nutzung frei. Die `database` kann nur korrekt, funktionieren wenn `serverA` verfügbar ist. Damit impliziert die `unlock()`–Operation die Nutzbarkeit von `serverA`. Wird Policy `policy 2.1` und anschließend Policy `policy 1.1` ausgeführt, so ist

ein Konflikt eingetreten. Das Ziel, dass die `database` während der Arbeitszeit jederzeit nutzbar ist (`policy 2.1`), wird durch die Policy `policy 1.1` verletzt.

Eine detaillierte Analyse der Konflikte bei funktionaler Abhängigkeit wird in Abschnitt 4.5 vorgenommen.

Ergebnis Unter Berücksichtigung des a priori Modells der funktionalen Abhängigkeit kann ein Konflikt zwischen Policy `policy 2.1` und `policy 1.1` identifiziert werden. Dieser Konflikt hat die folgenden Eigenschaften:

- Es ist **keine Domänenüberlappung** notwendig, damit ein Konflikt existiert.
- Die beteiligten Policies gehören **unterschiedlichen Managementfunktionsbereichen** an.
- Die **Reihenfolge** der Ausführung der Policies ist wichtig, damit der Konflikt eintritt. Das heißt, wenn die Events der Policies keiner Ordnung unterliegen würden, so trete der Konflikt nicht deterministisch auf.

Es ist nicht möglich diesen Konflikt ohne die zusätzliche Information der funktionalen Abhängigkeit der Managementobjekte zueinander zu identifizieren.

2.5.5 Analyse von Konflikten in der Policy–Hierarchie

In den vorigen Abschnitten wurden Policy–Konflikte für jede Domäne isoliert analysiert. Dieser Abschnitt betrachtet Konflikte, die durch den Zusammenhang der Domänen entstehen.

Es existieren zwei mögliche Ursachen, dass Policies in einer Policy–Hierarchie in Konflikt zueinander stehen:

- Widersprüchliche Spezifikation von Policies in einer Hierarchiestufe: d.h. Policies wurden in der Spezifikationsphase (beispielsweise durch Administratoren) so definiert, dass sie sich in Konflikt zueinander befinden.
- Fehlerhafte Verfeinerung von Policies: Während des Verfeinerungsprozesses werden neue Policies erzeugt, die nicht exakt dem Ziel der übergeordneten Policy entsprechen.

Das Beispiel hat gezeigt, dass die Konflikt–Analyse umso besser ist, je mehr Informationsquellen zur Verfügung stehen. Betrachtet man in der Hierarchiestufe der High–Level Policies nur die dort spezifizierten Ziele, so kann man häufig dort keine Aussage über in Konflikt befindliche Policies treffen.

Auf der Hierarchiestufe der operationalen Policies und unter Berücksichtigung der a priori Modelle können mehr Konflikte erkannt werden als in einer höheren Stufe. Im Beispiel konnte ein Konflikt zwischen den operationalen Policies aufgezeigt werden. Die Frage die sich jetzt stellt, ist: Befinden sich die beiden High–Level Policies in Konflikt zueinander?

Hier können drei Sichtweisen eingenommen werden:

- Die High–Level Policies stehen in Konflikt zueinander, da die jeweils korrekt abgeleiteten operationalen Policies in Konflikt zueinander stehen.
- Die High–Level Policies stehen nicht in Konflikt zueinander, da auf der Hierarchiestufe der High–Level Policies keine Rücksicht auf die vorhandene IT-Infrastruktur genommen werden soll.
- Die High–Level Policies stehen nur in Bezug auf ein konkretes a priori Modell in Konflikt zueinander und sonst nicht.

Im weiteren Verlauf der Arbeit wird der Verfeinerungsprozess von Policies in der Policy–Hierarchie nicht weiter betrachtet. Beispielsweise beschäftigt sich [BLR 03] mit der Policy–Verfeinerung.

2.5.6 Fazit: Verallgemeinerung der Analyse

Es wurde in den drei identifizierten Domänen des Konfliktmodells (Abbildung 2.4) der Unternehmens–, Management– und Modelldomäne für gegebene Beispielpolicies jeweils eine Konfliktanalyse betrieben. Die Analyse hat gezeigt, dass nur mit den Informationen, die auf Unternehmens– und Managementebene zur Verfügung stehen es nicht möglich ist, bestimmte Konfliktarten zu erkennen. Dies liegt entscheidend daran, dass keine Information über die Beziehung der Managementobjekte zueinander vorliegt.

Die Analyse der Modelldomäne hat gezeigt, dass unter Beachtung von a priori Modellen eine neue Qualität der Konfliktanalyse erreicht wird. Die wesentlichen Konfliktbehandlungsansätze in der Literatur gehen von einer Domänenüberlappung als notwendiges Kriterium für einen Konflikt aus. Es konnte gezeigt werden, dass unter Berücksichtigung der funktionalen Abhängigkeit ein Konflikt zwischen Policies gezeigt werden kann, ohne dass eine Domänenüberlappung vorliegt.

Daraus folgt, dass a priori Modelle eine wichtige Informationsquelle zur Konfliktanalyse darstellen.

2.6 Zusammenfassung

Dieses Kapitel beschäftigte sich detailiert mit der Problemanalyse. Dazu wurden zuerst wichtige Konzepte des policy–basierten Managements vorgestellt und Begrifflichkeiten aus dem Konfliktumfeld definiert. Die Definition eines formalen Konfliktmodells ermöglichte die Analyse von Policy–Konflikten in drei unterschiedlichen Domänen. Das wesentliche Ergebnis der Analyse ist, dass die Konfliktbehandlung der Domänen unterschiedlich mächtig sind und in der Modelldomäne Policy–Konflikte unter der Verwendung von a priori Modellen erkannt werden können, die bis jetzt in der Literatur nicht betrachtet wurden.

Der erste Teil der Fragestellung bezogen auf die Konflikterkennung aus Abschnitt 1.2 kann nun beantwortet werden:

1. Konflikterkennung und Konfliktdefinition

   (a) Existiert ein einheitliches Merkmal für Policy–Konflikte? Ist es möglich, trotz unterschiedlicher Anwendungsbereiche und Policy–Sprachen ein einheitliches Merkmal zu definieren?

   Im Beispielszenario wurde eine funktionale Abhängigkeitsbeziehung verletzt. Eine funktionale Abhängigkeitsbeziehung stellt ein Modell dar. Allgemein ausgedrückt kann als Merkmal eines Policy–Konflikts die *Verletzung eines a priori Modelleaspekts* definiert werden.

   Modelle werden in praktisch allen Anwendungsbereichen eingesetzt. Wenn Modelle als Ausgangspunkt der Konfliktdefinition betrachtet werden, ist eine breite Anwendbarkeit der zu entwickelnden Konfliktbehandlung gewährleistet.

   (b) Wo sind Policy–Konflikte in einem Managementsystem lokalisierbar? Treten Policy–Konflikte immer an der gleichen Stelle auf bzw. kann man vom dem Auftretungsort Rückschlüsse auf den Policy–Konflikt ziehen?

   Die Analyse in diesem Kapitel hat gezeigt, dass Konflikte sich in der Modellebene, genauer in den Modellen manifestieren müssen. Modelle stellen eines der wesentlichen Hilfsmittel im Management dar und sind für unterschiedlichste Anwendungsfälle verfügbar.

   (c) Zu welchem Zeitpunkt (bezogen auf den Policy–Lebenszyklus) kann ein Konflikt erkannt werden?

> Die Konfliktanalyse auf Modellebene im Beispielszenario hat gezeigt, dass eine Konfliktanalyse präventiv möglich ist, wenn die Modelle im vorhinein gegeben sind.

Aus den getroffenen Aussagen bezüglich der Fragestellung, lassen sich weitere Fragen ableiten:

1. Welche Art von Modellen sind im IT-Management relevant?
2. Wie können Aspekte von Modellen geeignet formal dargestellt werden?
3. Wie kann aus einem Modell eine Konfliktdefinition abgeleitet werden?
4. Wie kann aus der Konfliktdefinition eine Konfliktbehandlung abgeleitet werden?

Diese Fragen werden ausführlich in Kapitel 4 behandelt.

Im nächsten Kapitel werden existierende Ansätze in der Forschung untersucht.

Kapitel 3

Existierende Ansätze zur Konfliktbehandlung in der Forschung

Kapitelüberblick

3.1 Einleitung

Dieses Kapitel stellt für die vorliegende Arbeit die wesentlichen Ansätze vor, die in der Literatur existieren. Dabei werden die einzelnen Arbeiten detailliert beschrieben und bei einer abschließenden Bewertung werden die Vor– und Nachteile dargestellt sowie der Bezug zur vorliegenden Arbeit hergestellt.

Strukturiert ist das Kapitel folgendermaßen: Zuerst werden die Ansätze vorgestellt, die sich im Kern mit der Darstellung und Behandlung von Policy–Konflikten beschäftigen. Im zweiten Teil werden verwandte Problemstellungen aus anderen Forschungsbereichen vorgestellt, die im weiteren Verlauf der Arbeit Einfluss auf die vorgestellte Lösung haben: das Goal–Oriented Requirements Engineering und Verklemmung in Betriebssystemen.

3.2 Ansätze zur Policy–Konfliktbehandlung

Es existieren bis jetzt keine Bemühungen der großen Standardisierungsgremien, um Policy–Sprachen zu standardisieren. Lediglich Standards zum Management von Policies sind bei der IETF und DMTF vorgeschlagen worden. In Bezug auf Policy–Konflikte existieren keine Aktivitäten der Standardisierungsgremien.

Seit Anfang der 90er Jahre beschäftigt sich die Forschung mit dem Thema der Policy–Konflikte. Ein wesentlicher Beitrag in diesem Bereich ist am Imperial College in London entstanden.

3.2.1 Moffett, Sloman: Policy Conflict Analysis in Distributed System Management (1993)

Diese Arbeit [MoSl 93] liefert als wesentlichen Beitrag eine Klassifikation von Policy–Konflikten wie in Abbildung 3.1 zu sehen ist. Die beiden wichtigsten Arten von Konflikten sind dabei der Modalitätskonflikt und der Zielkonflikt. Der Modalitätskonflikt wird in Abschnitt 3.2.2 genauer untersucht und deswegen hier nicht weiter betrachtet. Alle Zielkonflikte werden in Tabelle 3.1 beschrieben. Das Kriterium zur Einteilung der Klassifikation ist die Überlappung von Domänen. In der rechten Spalte sind die notwendigen Domänen–Überlappungen beschrieben. Diese sind ein notwendiges Kriterium für den Eintritt eines Konflikts.

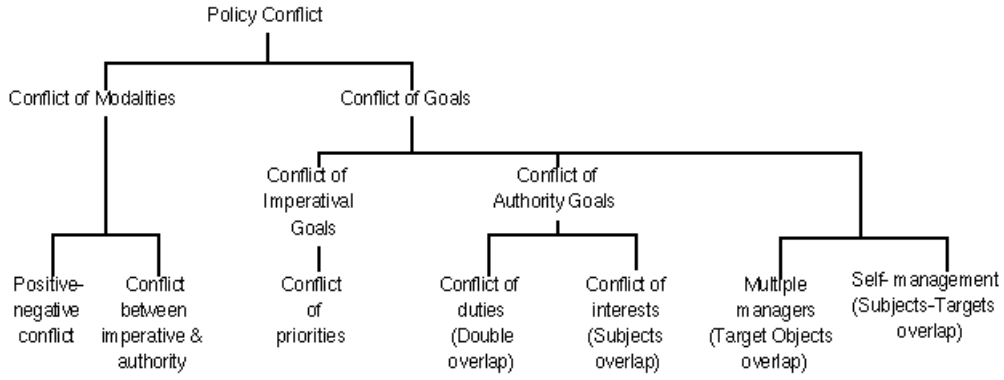

Abbildung 3.1: Klassifikation von Policy–Konflikten aus [MoSl 93]

Konfliktklasse	Beschreibung	Domänen–Überlappung
Prioritätenkonflikt	dieser Konflikt bezieht sich auf die Nutzung beschränkter Ressourcen. Policies befinden sich in dieser Konfliktklasse, wenn eine Policy die Nutzung der Ressource der anderen Policy verweigert oder einschränkt	Targets
Pflichtenkonflikt	dabei wird das Kontrollprinzip der Trennung von Aufgaben verletzt. Beispiel hierfür aus dem Finanzwesen ist, dass nicht derselben Person erlaubt ist Schecks auszustellen und einzulösen [MoSl 93]	Subjects und Targets
Interessenskonflikt	Wenn beispielsweise eine Bank für den einen Kunden eine Übernahmeempfehlung für ein Unternehmen ausspricht und einen anderen Kunden zum Aktienkauf dieses Unternehmens beraten soll [MoSl 93].	Subjects
Mehrfacher Managerkonflikt	Dieser Konflikt tritt ein, wenn zwei Policies unterschiedliche Ziele auf den selben Targets verfolgen.	Targets
Selbstmanagementkonflikt	Sind Subject und Target einer Policy identisch, besteht bspw. die Möglichkeit, dass ein Manager sich selbst das Gehalt erhöht.	Subject überlappt Target

Tabelle 3.1: Mögliche Zielkonflikte nach [MoSl 93]

Bewertung

Die drei Konfliktklassen Selbstmanagementkonflikt, Pflichtenkonflikt und Interessenskonflikt spielen im IT–Management kaum eine Rolle. Die Konfliktklasse Prioritätenkonflikt und der mehrfache Managerkonflikt sind praktisch identisch. Somit ist die aufgestellte Klassifikation im Bereich der Zielkonflikte nicht besonders hilfreich, um Konfliktarten zu unterscheiden.

In Bezug zu dieser Arbeit ist die Klassifikation wenig hilfreich, da sie entlang der möglichen Überlappungen gegliedert ist. Wie in Abschnitt 4.2 dargestellt, ist die Domänenüberlappung kein Kriterium für die dort diskutierten Konflikte.

3.2.2 Lupu, Sloman: Conflicts in Policy-based Distributed Systems Management (1999)

Ziel der Arbeit [LuSl 99] ist es, Verfahren und Werkzeuge für die präventive Konfliktbehandlung im Bereich der Autorisierung anzugeben.

Es werden zwei (Modalitäts–) Typen, entsprechend der Klassifikation wie in Abschnitt 3.2.1 dargestellt, von Policies jeweils in positiver und negativer Ausprägung spezifiziert, die auch in der Policy–Sprache Ponder [DDLS 00] übernommen wurden:

- **Authorization Policies** spezifizieren, welche Aktionen für eine Menge von Zielobjekten einem Subject erlaubt oder verboten sind (positive bzw. negative Authorization Policies, A+ bzw. A-) und sind mit Access Control Lists vergleichbar.
- **Obligation Policies** spezifizieren, welche Aktionen für eine Menge von Zielobjekten ein Subject ausführen muss bzw. nicht ausführen darf (positive bzw. negative Obligation Policies, O+ bzw. O-) und definieren somit die Pflichten und Verbote eines Subject.

Der Aufbau einer Policy–Beschreibung ist wie folgt:

- Identifikator, der die Policy eindeutig identifiziert.
- Modalitätstyp der Policy (A+, A-, O+, O-)
- Trigger: entspricht einem Ereignis und wird nur bei O+ verwendet.

- Subject: ist für die Durchführung der Aktion verantwortlich. Dies wird meist durch Rollen ausgedrückt.
- Target: spezifiziert die Objekte, auf denen die Aktionen ausgeführt werden. Dies wird durch Rollen oder Domänen ausgedrückt.
- Action: spezifiziert die ausführbaren Aktionen auf den Zielobjekten.
- Constraints: entsprechen der Condition, welche die Ausführbarkeit von Policies einschränken.

Es existieren drei unterschiedliche Modalitätskonflikte. Diese können zwischen Paaren von Policies eintreten. Dabei müssen sich jeweils die Subject– und Target–Domänen sowie die Aktionsmengen überlappen:

- O+ und O-: drückt den Konflikt aus, dass ein Subject die Pflicht hat, eine bestimmte Aktion auf einem Object auszuführen, und das Verbot, die Aktion auszuführen.
- O+ und A-: beschreibt den Konflikt, dass ein Subject die Pflicht hat, eine bestimmte Aktion auf einem Object auszuführen, aber genau für die Aktion keine Autorisierung besitzt.
- A+ und A-: ein Subject ist gleichzeitig autorisiert und nicht autorisiert eine Aktion auf einem Target auszuführen. Dieser Konflikt kommt in einem policy–basierten Managementsystem erst zum Tragen, wenn eine O+ Policy existiert, die eine Überlappung mit beiden A+ und A- Policies aufweist.

Bewertung

Durch die Typisierung der Policies gelingt es, durch syntaktische Analyse und Feststellung der Überlappung von Subject, Target und Action Modalitätskonflikte zu erkennen. Modalitätkonflikte beschränken sich auf das Gebiet der Autorisierung und Pflichten. Diese Konfliktklasse kann völlig unabhängig von Gegebenheiten des Managed Systems betrachtet werden, da der Augenmerk alleine auf Recht und Verbot bzw. Pflicht und Verbot liegt. Somit ist die Typisierung der Policies auf diese Merkmale hin naheliegend. Die Typisierung der Policies ist aber nur für die Klasse der Modalitätskonflikte von Nutzen. Andere Konflikte, die nicht aus dem Bereich des Sicherheitsmanagements kommen, wie in Abschnitt 2.5 dargestellt, profitieren von der Typisierung nicht, da alle Policies vom Typ O+ sind. Auch zwingt der Ansatz der Typisierung der Policies Ponder als Policy–Sprache

für alle Managementbereiche einzusetzten. Wie in Abschnitt 1.1 dargestellt, existieren aber eine Reihe von spezialisierten Policy–Sprache für unterschiedliche Anwendungsbereiche.

Weiterhin werden in [LuSl 99] Vorrangregeln bei Auftreten von Modalitätskonflikten diskutiert:

- Negative Policies (O-, A-) haben immer Vorrang: damit wird eine defensive Sicherheitspolitik zum Ausdruck gebracht.
- Zuweisung von Prioritäten: jeder Policy wird eine Priorität zugeordnet. Bei einem Konfliktfall wird nur die Policy mit der höchsten Priorität ausgeführt. Diese Regelung wirft zwei Probleme auf: zum Einen kann der Fall eintreten, dass beide Policies die gleiche Priorität haben und zum Anderen ist eine sinnvolle Prioritätenvergabe für ein große Anzahl von Policies bei verteilter Policy–Erstellung schwierig.
- Distanzmaß zwischen Policy und Target–Objekten: unter der Annahme, dass Policies und Target–Objekte hierarchisch gegliedert sind, kann eine Distanz zwischen beiden ermittelt werden. Diejenige Policy, welche die kürzeste Distanz aufweist, wird ausgeführt. Es besteht die Möglichkeit, dass beide den gleichen Distanzwert besitzen. Für diese Fälle kann die Vorrangregel keine Aussage treffen.
- Spezifität der Domänen: die Policy wird ausgeführt, welche die spezifischere Domäne adressiert. [LuSl 99] favorisieren diesen Ansatz. Allerdings existieren auch bei diesem Vorgehen Fälle, in denen keine Entscheidung getroffen werden kann, weil beispielsweise die Domänenangaben gleich spezifisch sind.

Bewertung

Jeder der vorgestellten Vorrangregeln erreicht, dass eine deterministische Auswahl der in Konflikt befindlichen Policies erfolgt. Keine der Regeln liefert für alle möglichen Modalitätskonflikte eine Lösung.

3.2.3 Damianou: A Policy Framwork for Management of Distributed System (2002)

Die Dissertation von Damianou [Dami 02] präsentiert ein komplettes policy–basiertes Management Framework inklusive einer Spezifikation der Policy–Sprache Ponder. Interessant für die vorliegende Arbeit ist die Spezifikation von

Meta–Policies in der Policy–Sprache Ponder, die in dieser Arbeit vorgenommen werden. Eine Meta–Policy ist eine Policy bezüglich einer Menge von Policies. In dieser Arbeit werden Meta–Policies zur Spezifikation von Bedingungen über eine Menge von Policies [Dami 02] verwendet. Meta–Policies können u.a. dazu verwendet werden, um Konflikte zwischen Policies zu beschreiben und damit auszuschließen. Die Eignung von Meta–Policies zur Beschreibung von Konflikten wird anhand konkreter Beispielekonflikte gezeigt. Eines der Beispiele sieht folgendermaßen aus [Dami 02]:

```
type role nurseT(set <patient> p) {
  inst auth+ mealSchedule {
    target p;
    action updateMealSchedule;
  }

  inst oblig administer {
    target p;
        on Time.at( 08:00:00 );
        do administerDrugs() -> checkTemperature();
  }

  inst meta maxNoOfPatients raises errorInPatients(p) {
    p->size < 10;
  }
}
```

In dem Beispiel wird für eine Rolle `nurseT` eine positive Autorisierungspolicy `auth+ mealSchedule`, eine Verpflichtungspolicy `oblig administer` und eine Meta–Policy `meta maxNoOfPatients` spezifiziert. Eine Krankenschwester hat die Pflicht um 8 Uhr für alle ihre Patienten die Arzneimittel zu verabreichen und anschließend die Temperatur zu überprüfen. Die Meta–Policy `meta maxNoOfPatients` erzwingt, dass die Anzahl der Patienten in der Menge `p` neun nicht übersteigt. Das bedeutet, dass die Meta–Policy hier die Wirkung einer Randbedingung hat, die immer eingehalten werden muss.

Weiterhin wird für die in Abschnitt 3.2.1 beschriebene Konfliktart des Selbstmanagementkonflikts eine Metapolicy spezifiziert.

Bewertung

Meta–Policies sind ein probates Mittel zur Spezifikation von Policy–Konflikten. Die betrachteten Konflikte beschränken sich auf einzelne Beispiele aus der Kon-

fliktkategorisierung aus Tabelle 3.1. Da der Weitere Konfliktarten werden nicht betrachtet. Da Meta–Policies Teil der Policy–Sprache Ponder sind, ist dieses Konzept nicht ohne größeren Aufwand als Konfliktbehandlung auf eine andere Policy–Sprache übertragbar. Meta–Policies haben in der praktischen Anwendung den Nachteil, dass sie als Invariante, also ständig einzuhaltende Bedingung für Policies, gelten. Das heißt, dass ihre Einhaltung kontinuierlich gewährleistet und damit überprüft werden muss. Es wird in der Arbeit keine Methodik beschrieben, wie man systematisch Meta–Policies ableiten kann.

3.2.4 Bandara, Lupu, Russo: Using Event Calculus to Formalise Policy Specification and Analysis

Ziel der Arbeit von Bandara, Lupu und Russo [BLR 03] ist es, den Prozess der Policy–Verfeinerung zu unterstützten. Die Konflikterkennung wird als Teilschritt der Policy–Verfeinerung angesehen. Dazu wird als Formalismus der *Event Calculus* [KoSe 86] eingeführt.

Event Calculus ist eine formale Sprache deren Ziel es ist, dynamisches Systemverhalten zu repräsentieren und über Systemzustände zu schließen.

Die Grundbestandteile des Event Calculus sind die *Zeit*, als eine Menge von diskreten Zeitpunkten, eine Menge von *Eigenschaften* (fluents), die sich über die Zeit verändert, und eine Menge von *Ereignissen* (events). Folgende Basisprädikate werden über den Bestandteilen definiert [BLR 03]:

```
Base predicates:
 1 initiates(A,B,T)  event A initiates fluent B
 2                   for all time > T.
 3 terminates(A,B,T) event A terminates fluent B
 4                   for all time > T.
 5 happens(A,T)      event A happens at time point T
 6 holdsAt(B,T)      fluent B holds at time point T.
 7                   This predicate is useful for
 8                   defining static rules.
 9 initiallyTrue(B)  fluent B is initially true.
10 initiallyFalse(B) fluent B is initially false.
```

Mittels `initiate` (Zeile 1) wird eine Eigenschaft `B` in das System bei dem Eintritt des Ereignisses `A` eingebracht. Das Prädikat `terminate` (Zeile 2) löscht eine Eigenschaft aus dem System. Die Verknüpfung von Ereignissen zu Zeitpunkten geschieht mittels `happens` (Zeile 5).

Damit Aussagen über Policy–Konflikt getroffen werden können, müssen Policies, das Policy Enforcement, das Managed System und Policy–Konflikte als Event Calculus dargestellt werden. Dazu ist es erforderlich, zusätzlich Bestandteile in den Event Calculus aufzunehmen. Folgendes Beispiel einer Policy ist in der Policy–Sprache Ponder spezifiziert [BLR 03]:

```
// Upon system shutdown, any jobs owned by running
// processes should be cancelled
oblig shutdownCancellation {
  on      systemShutdown;
  subject process/;
  target  printManager;
  action  cancelDoc(Job);
  when    Job.owner == process.owner; }
```

Die Darstellung dieser Policy als Event Calculus [BLR 03] lautet:

```
1 initiates(systemEvent(systemShutdown),
2           oblig(Process,
3           operation(printMgr, cancelDoc(Job))),
4           T) <-
5 validSpec(Process,
6           operation(printMgr, cancelDoc(Job))) and
7 holdsAt(pos(state(Job, owner, Process)), T).
```

Tritt das Ereignis `systemShutdown` (Zeile 1) zum Zeitpunkt `T` ein, wird geprüft, ob der Prozess der Eigentümer des Jobs ist (Zeile 7) und die Operation `cancelDoc()` tatsächlich existiert (Zeile 5 und 6). Daraufhin wird die eigentliche Operation ausgeführt (Zeile 3).

Das Managed System (und die darin enthaltenen Managementobjekte) wird als Zustandsmodell modelliert und ebenfalls als Event Calculus dargestellt. Nur am Beispiel des Zustandsdiagramm eines Printmanagers wird die Überführung von Managementmodellen in Event Calculus dargestellt.

Die Policy–Konflikte werden ebenfalls als Event Calculus dargestellt. Der Modalitätskonflikt O+/A- aus Abschnitt 3.2.2 hat als Event Calculus folgende Darstellung [BLR 03]:

```
1 holdsAt(unauthObligConflict(Subj, Op), Tm) <-
2 holdsAt(oblig(Subj, Op), Tm) and
3 holdsAt(deny(Subj, Op), Tm).
```

Wenn zu einem Zeitpunkt `Tm` eine Obligation–Policy ausgeführt werden soll (Zeile 2) und gleichzeitig eine negative Authorization–Policy existiert (Zeile 3), so ist ein Modalitätskonflikt O+/A- eingetreten.

Sind alle Bestandteile als Event Calculus spezifiziert, so kann mit Hilfe eines Beweisers getestet werden, ob für eine Menge von Policies in einem gegebenen Managed System ein Policy–Konflikt auftritt. Wird ein Konflikt erkannt, so werden die am Konflikt beteiligten Policies ausgegeben (nicht die Konfliktart selbst). Es findet als eine Konflikterkennung statt. Über die Komplexität dieses Vorgangs werden keine Aussagen getroffen.

Bewertung

Die vorgestellte Arbeit beschreibt, wie Policy–Konflikte durch einen streng formalen Ansatz spezifiziert und in einem weiteren automatisierten Beweisschritt erkannt werden können.

Der Ansatz geht von der notwendigen Bedingung der Domänenüberlappung der Target–Teile für Policy–Konflikte aus. Damit können Modellkonflikte wie in Abschnitt 2.5.4 dargestellt mit diesem Ansatz nicht erkannt werden. In Kapitel 4 wird für eine Reihe von Modellkonflikten eine Konfliktbehandlung vorgestellt.

Problematisch bei diesem Lösungsansatz ist der hohe Spezifikationsaufwand. Die Policy–Sprache, die Policies, das Policy Enforcement, das Managed System und die Policy–Konflikte müssen als Event Calculus spezifiziert werden. Vor allem der Aufwand, das Managed System in der nötigen Genauigkeit zu spezifizieren, wird als hoch eingeschätzt. Die Modellierung des Managed Systems wurde in der Arbeit nur am Beispiel eines einfachen Zustandsautomaten vorgenommen. Es wird nicht diskutiert, ob und wie weitere Arten von Modellen, die im Management verwendet werden, als Event Calculus darstellbar sind. Die Praxistauglichkeit des Ansatzes konnte damit nicht schlüssig gezeigt werden. Der Aufwand zur Übertragung des Konfliktbehandlungsansatzes auf eine andere Policy–Sprache ist hoch, da jede Policy–Sprache und die zugehörigen Policies als Event Calculus spezifiziert werden müssen. Der Ansatz ist nur zur präventive Konfliktbehandlung einsetzbar.

3.2.5 Verma: Policy-Based Networking

In dem Buch „Policy–Based Networking“ [Verm 00] beschäftigt sich Dinesh C. Verma mit dem Einsatz des policy–basierten Managements in IT–Infrastrukturen. Wesentliche Anwendungsgebiete, die betrachtet werden, sind SLA– und QoS–Management.

Der Autor wählt einen einfachen, geräte–zentrierten Aufbau einer Policy. Eine Policy besteht aus einem Condition–Teil und einem Action–Teil (Event, Subject und Target entfallen). Ein Policy wird ausgeführt, wenn der Condition–Teil zu „wahr“ ausgewertet wird.

In Bezug auf die Fragestellung dieser Arbeit beschäftigt sich der Autor mit der Erkennung von Policy–Konflikten. Die Konflikterkennung wird im Kontext der Policy–Validierung behandelt. Die Policy–Validierung besteht aus den Phasen:

- Syntax–Validierung: es wird getestet, ob eine Policy im Sinne der Policy–Grammatik wohlgeformt ist.
- Attribut–Validierung: es wird getestet, ob die Attributbelegung gültig ist (beispielsweise ob ein Wert innerhalb des angegebenen Intervalls liegt).
- Inter–Attribut–Validierung: es wird die Gültigkeit eines Attributwertes bezüglich eines anderen Attributwertes innerhalb einer Policy getestet. Als Beispiel einer Inter–Attribut–Abhängigkeit wird der Bezug eines DNS–Namen zu der zugehörigen IP–Adresse angegeben.

Ein Policy–Konflikt wird indirekt über ein Konsistenz–Kriterium von Policies wie folgt definiert: werden zwei Policies gleichzeitig ausgeführt, so müssen die Aktionen (in ihrer Wirkung) eindeutig identifizierbar sein. Die Definition setzt also als wesentliche Eigenschaft eines Konflikt die Gleichzeitigkeit voraus.

Im weiteren Verlauf wird die Definition des Policy–Konfliktes nur anhand des Beispiel diskutiert, dass zwei Policies gleichzeitig denselben Attributwert setzen wollen. Im Sinne der in Abschnitt 3.2.1 vorgestellten Klassifikation von Policy–Konflikten handelt es sich dabei um einen Mehrfachen–Manager–Konflikt. Damit ein Konflikt in diesem Fall eintreten kann, muss getestet werden, ob beide Policies tatsächlich gleichzeitig ausgeführt werden, also ihr Condition–Teil zu „wahr“ ausgewertet wird.

Zur Konflikterkennung werden topologische Räume (topological spaces) vorgeschlagen. Ein Condition–Teil besteht aus n (logischen) Termen. Jeder Term wird einer Achse in einem Koordinatensystem zugeordnet. Jeder Condition–Teil beschreibt dann einen gewissen Raum in dem Koordinatensystem. Überlappen sich die Räume zweier Policies, so kann potentiell ein Konflikt eintreten. Folgendes leicht abgeänderte Beispiel aus [Verm 00] soll den Ansatz verdeutlichen. Gegeben sind zwei Policies:

Policy	`policy1`
Action	`serviceClass='Silver';`
Condition	`port==80;`

Policy `policy1` besagt, dass jedem Nutzer zur Nutzung des Webservers (angezeigt durch `port==80`) die Serviceklasse `Silver` zugeordnet wird.

Die zweite Policy ordnet einem Nutzer, dessen IP–Adresse in einem gewissen Intervall liegt, die Serviceklasse `Gold` zu.

Policy	`policy2`
Action	`serviceClass='Gold';`
Condition	`IPAddress` $\in \{10.11.12.0, ..., 10.11.12.30\}$;

Ein Konflikt entsteht, wenn ein Nutzer, dessen IP–Adresse in dem Intervall liegt, auf den Webserver zugreifen will. Beide Condition–Teile der Policies `policy1` und `policy2` werden gleichzeitig zu „wahr“ ausgewertet und beide Action–Teile greifen schreibend auf dasselbe Attribut `serviceClass` zu.

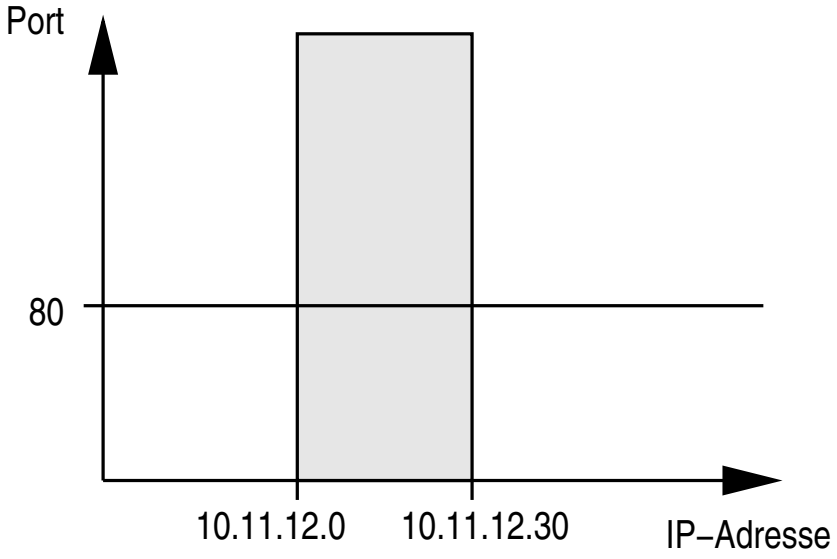

Abbildung 3.2: Topologische Räume zu den Beispielpolicies aus [Verm 00]

Jeder Condition–Teil besteht aus einem Term und es existieren damit die beiden Achsen Port und IP–Adresse (siehe Abbildung 3.2). Policy `policy1` adressiert Port 80 im Condition–Teil, dementsprechend zeigt die horizontale Linie den zugehörigen topologischen Raum an. Der Condition–Teil von Policy `policy2` ist durch das Intervall der IP–Adressen gekennzeichnet. Der topologische Raum wird damit durch das Rechteck repräsentiert. Der Überlappungsbereich der beiden topologischen Räume zeigt den potentiellen Konflikt an.

Der Ansatz kann zur Erkennung potentieller Konflikte eingesetzt werden. Zur Konfliktlösung werden Vorrangregeln durch Prioritäten vorgeschlagen, aber nicht weiter diskutiert.

Dieser Ansatz setzt implizit Expertenwissen bei der Beurteilung der potentiellen Konflikte voraus. Ist eine Überlappung erkannt, so muss untersucht werden, ob die Attributwerte der einzelnen Terme gleichzeitig eingenommen werden können. Dies kann nur durch Expertenwissen beurteilt werden, wie an folgenden Beispiel demonstriert wird.

Ein weiteres Defizit ist, dass dem Ansatz die zeitliche Dimension fehlt. Angenommen zu den beiden spezifizierten Policies `policy1` und `policy2` existiert noch eine weitere `policy3`:

Policy	`policy3`
Action	`serviceClass='Bronze';`
Condition	`congestionAlert==true;`

Kommt es zu einer Stausituation im Netz, so bekommen alle Dienste die Serviceklasse `'Bronze'` zugeteilt.

Bezogen auf den Ansatz der topologischen Räume wird nun eine weitere Dimension eingeführt. Der neue topologische Raum würde die beiden bereits vorhanden Räume schneiden (siehe Abbildung 3.2). Damit wird ein potentieller Konflikt zwischen den drei Policies angezeigt. Es besteht aber kein Konflikt zwischen der Policy `policy3` und den anderen beiden Policies: Die ersten beiden Policies können der Initialisierungsphase, der initialen Belegung des Attributs `serviceClass` zugeordnet werden. Policy `policy3` ist hingegen der Operationsphase des Dienstes anzusiedeln ist. Demnach ist es korrekt, dass Policy `policy3` in der späteren Phase den Wert des Attributs `policy3` ändert, ein Konflikt liegt nicht vor.

Bewertung

Der Ansatz der topologischen Räumen ist für Attribute, die als Zahlenwert oder Intervall ausdrückbar sind, gut anwendbar. Je nach Anzahl der Terme in einem Condition–Teil einer Policy steigt die Dimension des Koordinatensystems und damit auch die Komplexität, Überlappungen zu identifizieren an. Bei n Policies mit k Dimensionen ist die Komplexität im worst–case $O(kn^2)$. Wie gezeigt ist der Ansatz für Policies, die unterschiedlichen Betriebsphasen zuordenbar sind nicht geeignet, da ein Zeitbegriff fehlt. Darüberhinaus kann mit diesem Ansatz kein Konflikt behandelt werden, der sich nicht auf das gleiche Attribut im Action–Teil bezieht. Somit kann beispielsweise der Konflikt wie er aus einer funktionalen Abhängigkeit (Abschnitt 2.5) mit diesem Ansatz nicht erkannt werden, in Abschnitt 4.5 stellt die vorliegenden Arbeit eine Konfliktbehandlung für diesen Konflikt vor.

3.2.6 Dunlop et al: Dynamic Conflict Detection in Policy-Based Management Systems

In [Dira 02] wird eine reaktive Konflikterkennung basierend auf deontischer Logik [Hilp 71] entwickelt. Die Autoren gehen davon aus, dass eine große Anzahl von Policy–Konflikten erst dynamisch zur Laufzeit der Policies erkannt werden kann.

Policies werden mit Hilfe der deontischen Logik formal beschrieben. Die deontische Logik, deren Ursprünge in der formalen Erfassung von Gesetzestexten liegt [WiMe 93], befasst sich mit formalen Beschreibung von Rechten, Privilegien, Pflichten, Verboten etc.

Mit Policies können Pflichten (Obligations), Rechte (Permissions) und Verbote (Prohibitions) ausgedrückt werden und formal in deontischer Logik dargestellt werden. Mit Hilfe der deontischen Logik können Schlüsse über Sequenzen von Ereignissen gezogen werden.

Es wird eine Anzahl von Ereignissen definiert, die als Anstoß für die Konflikterkennung dienen: Hinzufügen einer Policy, Versuch einer Aktionsausführung, Zustandsänderung einer Entität oder ein externer Event.

Die Konflikterkennung überprüft als erstes die notwendige Bedingung der Überlappung von Subject–, Target–, und Action–Feld. Somit betrachtet dieser Ansatz grundsätzlich die gleichen Policy–Konflikte wie in Abschnitt 3.2.2 beschrieben.

Es wird eine Konfliktklassifizierung angegeben mit den vier Klassen:

Interner Policy–Konflikt Diese Konfliktart tritt ein, wenn Policies sich in Konflikt befinden, die an dieselbe Rolle geknüpft sind.

Externer Policy–Konflikt Diese Konfliktart tritt ein, wenn ein Nutzer an mehrere Rollen geknüpft ist und sich die Policies für unterschiedliche Rollen in Konflikt zueinander befinden.

Policy–Raum Konflikt Ein Policy–Raum stellt eine Gruppierung von Policies (ähnlich dem Domänengedanken) dar. Existiert eine Subject–Überlappung unterschiedlicher Policy–Räume, so kann diese Konfliktart eintreten.

Rollenkonflikt Ein Rollenkonflikt tritt ein, wenn ein Nutzer zueinander inkompatible Rollen einnimmt.

Bewertung

Diese Arbeit präsentiert eine Erkennung von Policy–Konflikten durch formales Schließen. Dabei werden Policies formal in deontischer Logik ausgedrückt.

Schwerpunkt der Analyse sind Pflichten, Verbote und Rechte die durch Policies ausgedrückt werden. Eine Überlappung von Subject–, Target–, und Action–Feld wird als notwendige Voraussetzung zum Zustandekommen eines Policy–Konflikts angesehen und reduziert damit die erkennbaren Konfliktarten wesentlich. Beispielsweise alle Modellkonflikte, die über eine Beziehung definiert werden (siehe Abschnitt 2.5.4), sind von diesem Ansatz nicht behandelbar. In Kapitel 4 werden für Modellkonflikte dieser Art eine Konfliktbehandlung präsentiert. Eine Konfliktlösung wird nicht präsentiert. Der Aufwand für die Übertragung des präsentierten Ansatzes auf eine andere Policy–Sprache ist hoch, da von einer typisierten Policy–Sprache wie beispielsweise Ponder ausgegangen wird. Trifft dies nicht zu, so muss die Grammatik der Ziel–Policy–Sprache geändert werden.

3.2.7 Chomicki: Conflict Resolution with PDL

Diese Arbeit entwickelt eine Konflikterkennung und Konfliktlösung für die Policy Description Language $\mathcal{PDL}$ [CLN 00, CLN 03].

$\mathcal{PDL}$ ist eine regelbasierte Sprache und besteht aus den Teilen Event, Condition, Action und kennt also keine Subject– und Target–Felder.

Zur Konflikterkennung wird ein sogenannter Monitor eingeführt, welcher Action–Constraints auswertet. Ein Action–Constraint hat die Form:
never `Action1` $\wedge$ `...` $\wedge$ `ActionN` $\wedge$ `C`
Ein Action–Constraint sagt aus, dass die Aktionen `1` bis `N`, wenn die Bedingung `C` gilt, nicht gleichzeitig ausgeführt werden dürfen, da sonst ein Konflikt eintritt.

Ein Monitor überwacht die Einhaltung aller Action–Constraints und kann dabei folgende Strategien auswählen, falls ein Policy–Konflikt eingetreten ist:

Blockieren von Aktionen Es wird eine Teilmenge an Aktionen blockiert, also nicht ausgeführt. Dabei wird dem Monitor überlassen, welche Aktionen blockiert werden.

Verzögern von Aktionen Ein Teilmenge von Aktionen wird verzögert ausgeführt. Für welche Arten von Policy–Konflikt eine verzögerte Ausführung eine Mögliche Lösung des Problems darstellt, wird nicht betrachtet.

Blockieren von Ereignissen Anstatt Aktionen zu blockieren können auch Ereignisse blockiert werden und somit die Ausführung einer kompletten Policy–Menge, die auf dieses Ereignis triggert, unterbunden werden.

Verzögern von Ereignissen Es wird davon ausgegangen, dass mehrere Ereignisse gleichzeitig auftreten können. Durch die Verzögerung von Ereignis-

sen wird eine Sequentialisierung der Ereignisse erreicht und somit eventuell Policy–Konflikte verhindert.

Bewertung

Diese Arbeit präsentiert eine Konflikterkennung und Konfliktlösung für die Policy Description Language $\mathcal{PDL}$. Die Konfliktbehandlung fokussiert auf die IT–Managementdomäne (siehe Abschnitt 2.4) und ist für die Policy–Sprache $\mathcal{PDL}$ konzipiert. Da der Ansatz die Modellebene nicht betrachtet, können auch keine Modellkonflikte behandelt werden. Darüberhinaus ist der Aufwand den Konfliktbehandlungsansatz auf eine andere Policy–Sprache zu übertragen hoch, da die Konzepte der Konfliktbehandlung direkt auf $\mathcal{PDL}$ aufbauen. Ebenso wird keine Unterstützung für die Spezifikation von Action–Constraints angegeben.

3.2.8 Gegenüberstellung der Ansätze

Zur Einordnung und Bewertung der vorgestellten Ansätze werden die nachfolgenden charakteristischen Eigenschaften herangezogen:

Betrachtete Domänen Die betrachteten Domänen beziehen sich auf die drei Domänen des Konfliktmodells (siehe Abschnitt 2.4).

Einschränkung Es wird beschrieben, ob der Konfliktbehandlungsansatz Einschränkungen bezüglich der Allgemeinheit aufweist:

- **bezüglich Policy–Sprache** Ist der Ansatz für beliebige Policy–Sprachen einsetzbar?
- **bezüglich Konfliktarten** Werden spezielle Annahmen getroffen, welche die erkennbaren Konfliktarten im vorhinein einschränkt?

Konflikterkennung Bei der Konflikterkennung können zwei Phasen unterschieden werden:

- **präventiv** Kann die Konflikterkennung zur Spezifikationsphase von Policies stattfinden?
- **reaktiv** Kann die Konflikterkennung in der Ausführungsphase von Policies stattfinden?
- **algorithmische Umsetzung** Wird ein Algorithmus angegeben, wie die Konflikterkennung umgesetzt werden kann?

Konfliktlösung Bei der Konfliktlösung können zwei Phasen unterschieden werden:

präventiv Kann die Konfliktlösung zur Spezifikationsphase von Policies stattfinden?

reaktiv Kann die Konfliktlösung in der Ausführungsphase von Policies stattfinden?

algorithmische Umsetzung Wird eine Algorithmus angegeben, wie die Konfliktlösung umgesetzt werden kann?

Methodik Wird eine Methodik für die Konfliktbehandlung angegeben, insbesondere wie systematisch Konflikte abgeleitet werden können?

Abbildung 3.3 bewertet die in diesem Kapitel vorgestellten Ansätze zur Policy–Konfliktbehandlung anhand der beschriebenen Kriterien.

Auffallend sind folgende Defizite der bestehenden Ansätze:

- Die Modelldomäne wird kaum betrachtet. In Kapitel 2 konnte gezeigt werden, dass Konflikte existieren, die nur durch die Betrachtung der Modelldomäne behandelt werden können.
- Die Konfliktbehandlung ist meist an eine dedizierte Policy–Sprache gebunden. Es hat sich bis jetzt keine Policy–Sprache in der Praxis durchgesetzt. Damit führt die Festlegung der Konfliktbehandlung u.U. zu einer Insellösung.
- Die betrachtbaren Konfliktarten sind meist im vorhinein eingeschränkt, da von einer Domänenüberlappung als notwendiges Kriterium ausgegangen wird. Wie in Kapitel 2 gezeigt, erfüllen nicht alle Konflikte dieses Kriterium.
- Es existieren kaum Algorithmen zur Konfliktbehandlung. Effiziente Algorithmen sind aber notwendig, wenn die Konfliktbehandlung reaktiv erfolgen muss oder wenn die Anzahl der Policies im System sehr groß ist.
- Es existiert kein methodisches Vorgehen, um neue Konfliktarten ableiten zu können. Damit muss für jede neue Konfliktart eine Konfliktbehandlung erarbeitet werden, ohne eine Anleitung dafür zu besitzen.

Für die genannten Defizite werden in den folgenden Kapiteln Lösungsansätze entwickelt.

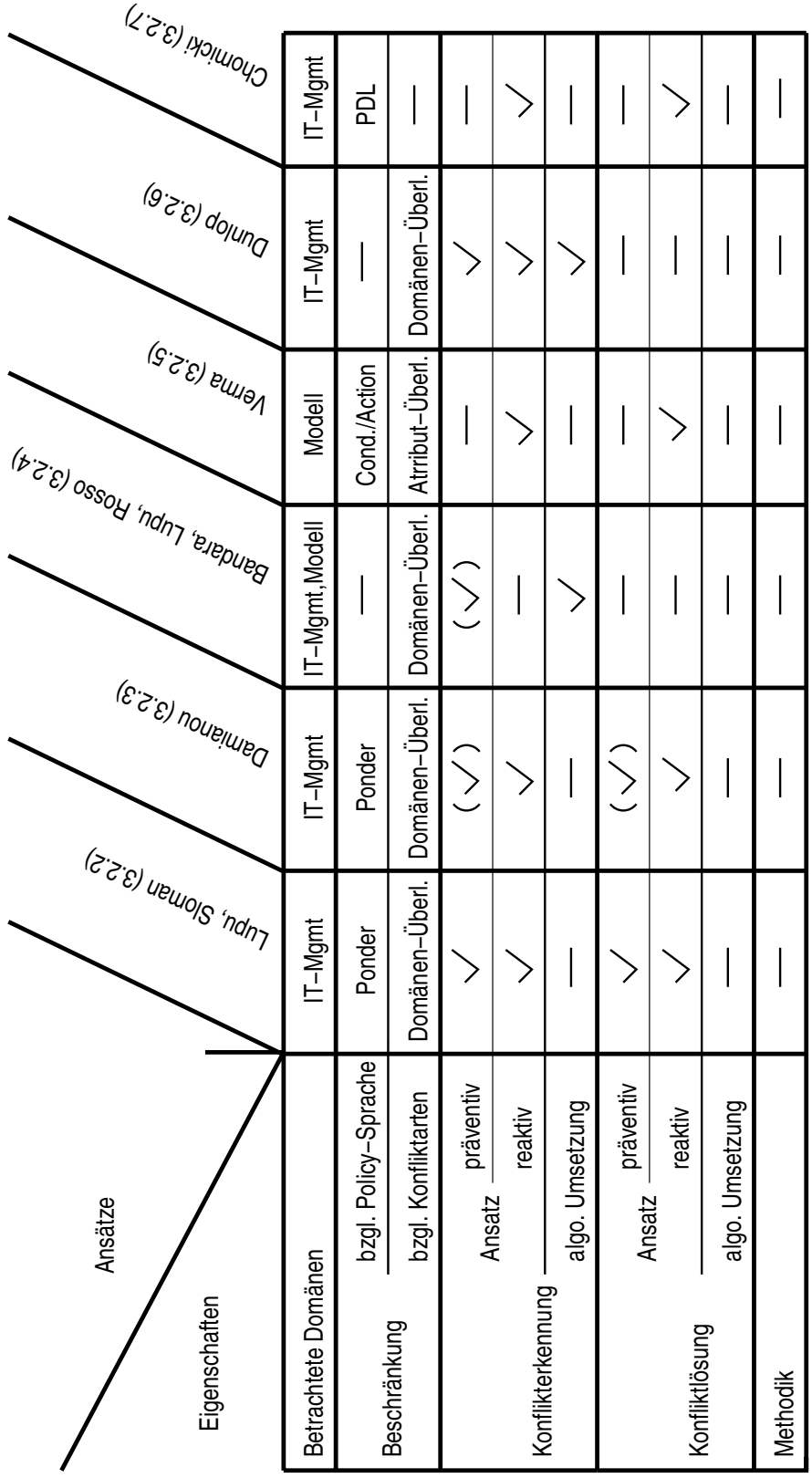

Abbildung 3.3: Charakterisierung der Ansätze

3.3 Verwandte Problemstellungen

In diesem Abschnitt werden zwei Forschungsbereiche vorgestellt, in denen Probleme behandelt werden, die zu Policy–Konflikten ähnlich sind und Ideen für die vorliegende Arbeit liefern. Analysiert werden die Bereiche des Goal–Oriented Requirements Engineering und Verklemmungen in Betriebssystemen.

3.3.1 Goal–Oriented Requirements Engineering

Das Goal–Oriented Requirements Engineering ist eine für die vorliegende Arbeit relevantes Forschungsgebiet, da es sich mit der Zielbeschreibungen der Unternehmensdomäne befasst [LAMS 01] (siehe Abschnitt 2.4 und Abbildung 2.4). Goal–Oriented Requirements Engineering beschäftigt sich mit der Strukturierung, Analyse, Verfeinerung und Ableitung von Zielen [LAMS 01].

Ein *Ziel* ist die Beschreibung einer Eigenschaft, welches ein betrachtetes System erreichen soll [vLD 98]. Dabei kann das System bereits existieren oder erst konstruiert werden. Ziele können unterschiedlich abstrakt formuliert sein und reichen von strategischen Zielen bis implementierungsnahen Zielen. Ziele können durch Analyse von Unternehmensinteressen oder des zu Grunde liegenden Systems extrahiert werden.

Der Verfeinerungsprozess der Ziele endet auf unterster Ebene mit der Spezifikation von Anforderungen. Damit ist die Formulierung von Zielen der erste wichtige Schritt im Prozess des Requirements Engineering.

Ziele sind für den Prozess des Requirements Engineering aus folgenden Gründen wichtig:

- Die Erfüllung der Anforderungen (gemäß den Zielen) kann erfasst werden.
- Irrelevanter Anforderungen können vermieden werden.
- Die Verfeinerung von Zielen dient als Strukturierungshilfe von Anforderungen.
- Die Bewältigung von Konflikten, die durch unterschiedliche Sichtweisen der Akteure hervorgerufen werden.

Eine allgemeine Klassifikation von Zielen ist in Abbildung 3.4 zu sehen. Ziele können funktionale und nicht funktionale Eigenschaften eines Systems beschreiben. Funktionale Ziele beziehen sich auf den von einem System zu erbringenden Dienst, während nicht funktionale Ziele die Qualität des Systems bezüglich Leistung, Sicherheit etc. beschreiben.

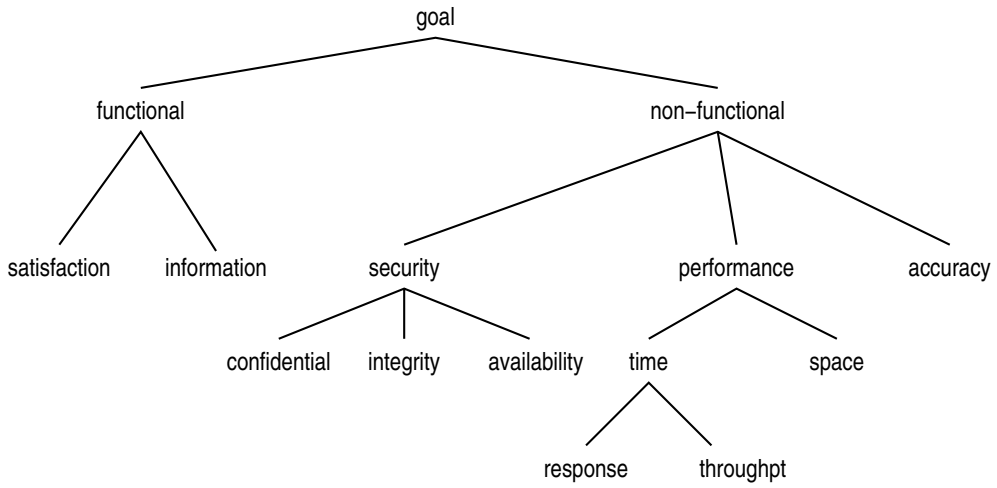

Abbildung 3.4: Klassifikation von Zielen nach [LAMS 01]

Ziele und Systeme, auf die Ziele angewendet werden sollen, besitzen gegensätzlichen Charakter [LAMS 01]: Ziele sind abstrakt, deklarativ und identifizieren gewünschte Eigenschaften explizit. Hingegen sind Systeme Szenario–spezifisch, werden meist prozedural dargestellt und gewünschte Eigenschaften sind implizit.

Ziele werden im Goal–Oriented Requirements Engineering formalisiert dargestellt. Folgende Definition ist ein Beispiel für eine formale Ziel–Spezifikation [LAMS 01]:

Goal	`Achieve [TrainProgress]`
FormalDef	$(\forall$`tr:Train, b:Block`$)$`[On(tr,b)`$\Rightarrow \Diamond$`On(tr, b+1)]`

Das Beispiel ist aus dem Bereich des Zugverkehrs genommen. Züge (`trains`) befahren Streckenabschnitte (`block`), die nummeriert sind. Das Ziel (`TrainProgress`) drückt aus, dass wenn sich ein Zug auf einem Streckenabschnitt `b` befindet, wird es sich in Zukunft (Operator $\Diamond$) auf Abschnitt `b+1` befinden. Das Attribut `Achieve` beschreibt, dass es sich um ein erreichbares Ziel handelt.

Zu dem Attribut Erreichbarkeit (`achieve`) existiert das gegensätzliche Attribut Unterlassen (`cease`). Ein weiteres gegensätzliches Attributepaar ist Vermeiden (`avoid`) und Einhalten (`maintain`).

Ziele können zueinander in Beziehung gesetzt werden. Typische Beziehungen zwischen Zielen sind: Verfeinerung, ein Unterziel kann zu einem Oberziel beitragen, oder eine Konfliktbeziehung (conflict). Abbildung 3.5 zeigt einen Graphen,

in dem Ziele zueinander in Beziehung gesetzt werden. Dabei werden in diesem Graphen weiche und harte Ziele unterschieden. Harte Ziele können (technisch) nachgewiesen werden bzw. in Zahlen ausgedrückt werden, während weiche Ziele nicht eindeutig zu verifizieren sind. Beispielsweise kann das Ziel „mehr Passagiere zu befördern“ (siehe linke Seite der Abbildung 3.5) entweder dadurch erreicht werden, dass das Ziel „neue Gleise bauen“ verwirklicht wird, oder die Ziele „Geschwindigkeit der Züge maximieren“ bzw. die „Distanz zwischen Zügen minimieren“ umgesetzt werden. Die beiden Ziele „Minimierung der Entwicklungskosten“ und „neue Gleise bauen“ stehen in einem strengen Konflikt zueinander. Wobei strenger Konflikt bedeutet, dass diese beiden Zielen unter allen Umständen zu einem Konflikt führen.

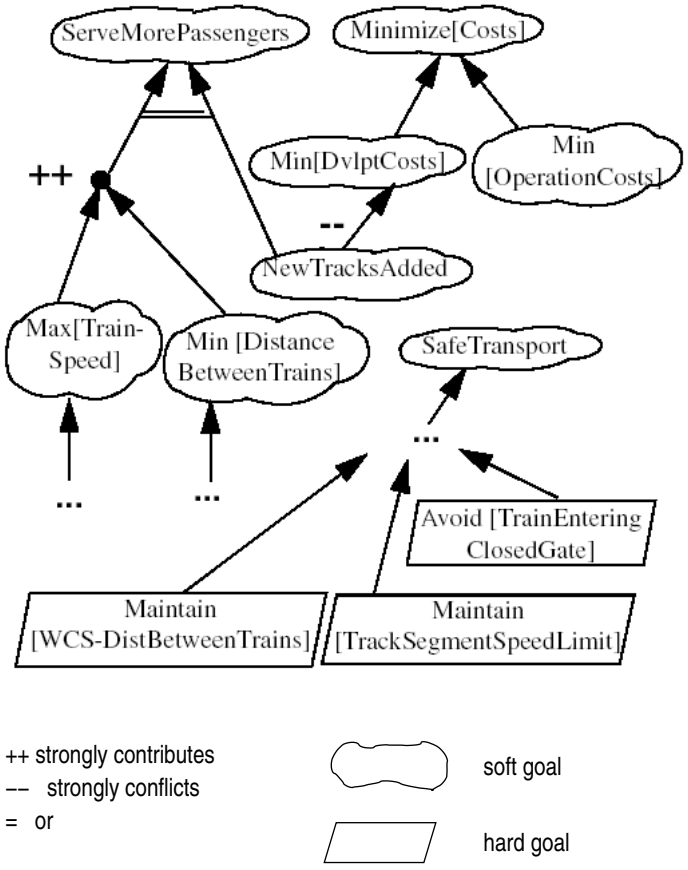

Abbildung 3.5: Zielgraph aus [LAMS 01]

Letzter und wichtiger Schritt in der Verfeinerung von Zielen zu Anforderungen ist die Ableitung von Vor– und Nachbedingungen für (System–)Operationen

[LAMS 01]. Dabei werden Operationen Ziele zugeordnet und aus den Zielen Vor– und Nachbedingungen abgeleitet.

In [vLD 98] wird eine neue Konfliktart identifiziert: die Divergenz. Eine Divergenz tritt im Gegensatz zu einem Konflikt nur unter gewissen Randbedingungen ein. Zur Erkennung von Konflikten und Divergenzen werden zum einen Methoden des formalen Schließens und zum anderen Heuristiken eingesetzt, die für gegebene Divergenztypen eine Lösung vorschlagen. Zur Lösung von Konflikten und Divergenzen werden Ziele gelöscht, neue erzeugt oder bestehende modifiziert. Ebenso werden für Divergenzen Heuristiken zur Lösung angegeben.

Zur Werkzeugunterstützung existieren grafische Editoren zur Spezifikation und Verfeinerung von Zielen. Weitere Werkzeug–Unterstützung beispielsweise zum automatischen Schließen, speziell zum Erkennen und Lösen von Konflikten und Divergenzen, existiert bis jetzt nicht [LAMS 01].

Bewertung

Goal–Oriented Requirements Engineering ist für die vorliegende Arbeit in vielerlei Hinsicht interessant: es ist der einzige Forschungsbereich, der sich mit den Zielen in der Unternehmendomäne (siehe Abschnitt 2.4) befasst. Die Idee, Vor– und Nachbedingungen für Operationen aus Zielen abzuleiten, wird bei der im folgenden Kapitel vorgestellten Methodik aufgegriffen. Die Problematik, dass abstrakte Ziele in Konflikt zueinanderstehen können, *abhängig* von dem betrachteten System, wird nicht betrachtet. Das ist aber wie in Abschnitt 2.5.4 gezeigt eine wichtige Eigenschaft in der Beziehung von Konflikten zu Systemen (die durch Modelle dargestellt sind).

3.3.2 Verklemmungen in Betriebssystemen

Im Bereich der Betriebssysteme existiert ein dem Policy–Konflikt ähnliches Phänomen — die Verklemmung (Deadlock). Zuerst werden die an einer Verklemmung beteiligten Objekte beschrieben, bevor anschließend die eigentliche Verklemmung definiert und beschrieben wird. In den einzelnen Abschnitten werden die Gemeinsamkeiten zwischen der Verklemmung und Policy–Konflikten dargestellt. Dabei lehnt sich die Darstellung der Verklemmung in Betriebssystemen in diesem Abschnitt an [Stal 97] an.

Ein Betriebssystem ist eine abstrakte Maschine, deren Aufgabe es ist die effiziente Bereitstellung der Ressourcen (Betriebsmittel) eines Computersystems zu gewährleisten. Für die Verklemmung wesentliche Konzepte eines Betriebssystems sind:

Prozess Ein Prozess ist die Abstraktion eines in Ausführung befindlichen Programms. Ein Prozess ist unterbrechenbar.

Betriebsmittel Ein Betriebsmittel ist die Abstraktion einer nutzbaren Ressource eines Computersystems.

Ein Prozess kann Betriebsmittel (z.B. Speicher Ein– und Ausgabegeräte) vom Betriebssystem anfordern und Betriebsmittel belegen.

Vergleich von Prozessen und Policies Das Konzept eines Prozesses und einer operationalen Policy sind ähnlich. Beide führen Operationen aus und zu einem Zeitpunkt können mehrere Prozesse bzw. Policies ausgeführt werden. Ein Prozess nutzt und manipuliert Betriebsmittel, während eine Policy Managementobjekte manipuliert. Von diesem abstrakten Standpunkt aus können die Prozesse und Policies als in ihren charakteristischen Eigenschaften als ähnlich betrachtet werden.

Definition und Modellierung von Verklemmungen

In [Stal 97] werden drei notwendige Eigenschaften definiert, die gegeben sein müssen, damit ein Verklemmung eintreten kann:

Wechselseitiger Ausschluss (Mutual Exclusion) Betriebsmittel unterliegen einer exklusiven Nutzung eines Prozesses zu einem Zeitpunkt.

Belegen und Warten (Hold and Wait) Ein Prozess behält seine Betriebsmittel, während er weitere Betriebsmittel anfordern kann.

Ununterbrechenbarkeit (No Preemption) Einem Prozess kann ein Betriebsmittel, welches es hält, nicht entzogen werden.

Diese drei Eigenschaften resultieren aus grundsätzlichen Designentscheidungen von Betriebssystemen — alle gängigen Betriebssysteme besitzen diese Eigenschaften.

Gelten diese drei Eigenschaften und tritt die folgende Bedingung auf:

Zyklische Wartebedingung (Circular Wait) Existiert eine zyklische Wartebedingung, so dass jeder Prozess mindestens ein Betriebsmittel hält, dass von einem anderen Prozess angefordert wird,

so wird von einer *Verklemmung* (Deadlock) gesprochen. Die zyklische Wartebedingung ist unter der Voraussetzung der drei Eigenschaften nicht ohne zusätzliche Hilfe lösbar.

Zur Darstellung von Verklemmungen werden Betriebsmittelgraphen eingesetzt. Abbildung 3.6 zeigt eine Verklemmung zwischen den Prozessen P1 und P2, da sie die zyklische Wartebedingung erfüllen: Prozess P1 belegt Betriebsmittel BM1 und fordert BM2 an. Prozess P2 belegt Betriebsmittel BM2 und fordert BM1 an. In dieser Verklemmungssituation kann keiner der beiden Prozesse mit der Ausführung fortfahren.

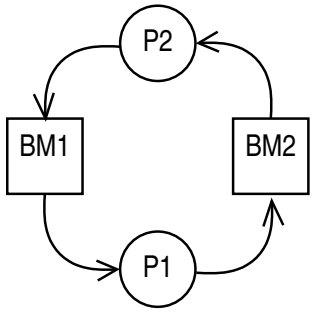

Abbildung 3.6: Modellierung einer Verklemmung als Betriebsmittelgraph

Vergleich zwischen Verklemmungen und Policy–Konflikten Es ist zu untersuchen, ob und wie die Eigenschaften von Verklemmungen auch im policy–basierten Management vorhanden sind.

Wenn man die Definition einer Verklemmung mit ihren vier Eigenschaften und die Policy–Konfliktdefinition der Modellebene gegenüberstellt (siehe Abschnitt 2.5.4), ist festzustellen, dass sich die Definitionen zum großen Teil ähnlich sind:

- es sind mindestens zwei oder mehr Prozesse bzw. Policies beteiligt.
- In beiden Fällen wird der Fortschritt (Prozessfortschritt bzw. Ausführbarkeit von Policies) betrachtet.
- die zyklische Wartebedingung von Prozessen wird im allgemeinen durch einen Betriebsmittelgraphen dargestellt. Die Eigenschaft, dass in diesem Graphen keine Zyklen existieren dürfen, kann als Modellbedingung gemäß der Modellkonfliktdefinition aus Abschnitt 2.5.4 betrachtet werden.

Da die Modellkonfliktdefinition auf beliebige Modelle abzielt, die Verklemmungsdefinition sich nur auf Betriebsmittelgraphen stützt, kann in diesem Sinn

die Verklemmungsdefinition als ein Spezialfall der Modellkonfliktdefinition aufgefasst werden.

Im Weiteren werden Strategien zur Deadlock–Behandlung untersucht: Deadlock–Verhinderung, Deadlock–Vermeidung und Deadlock–Lösung.

Deadlock–Behandlung

Es existieren zwei prinzipielle Strategien, mit dem Verklemmungsproblem zu verfahren: Es wird ein Betriebssystem konstruiert, in dem eine der drei obigen Eigenschaft nicht existiert bzw. die zyklische Wartebedingung per se nicht auftreten kann (Deadlock–Verhinderung). Eine andere Strategie besteht darin, dynamisch beim Ablauf der Prozesse das Eintreten der zyklischen Wartebedingung zu unterbinden (Deadlock–Vermeidung).

Deadlock–Verhinderung Diese Strategie verhindert durch ein entsprechendes Design des Betriebssystems, eine der genannten Eigenschaften und Bedingung durch Konstruktion im vorhinein nicht auftreten zu lassen [Stal 97].

Die Einschränkung der Eigenschaft des *wechselseitigen Ausschlusses* kann nur für Betriebsmittel aufgehoben werden, die eine entsprechende Charakteristik aufweisen (z.B. Lesezugriff auf eine Datei). Da nicht alle Betriebsmittel diesen Charakter haben, kann diese Eigenschaft nicht aufgehoben werden.

Die Eigenschaft *Warten und Belegen* kann dadurch aufgehoben werden, dass ein Prozess alle Betriebsmittel, die es benötigt, bei seiner Initialisierung bekannt gibt.

Die Eigenschaft der *Ununterbrechenbarkeit* kann aufgehoben werden, indem beispielsweise alle Prozesse, die ein bestimmtes Betriebsmittel nicht zugeteilt bekommen, alle Betriebsmittel abgeben müssen.

Damit die *zyklische Wartebedingung* aufgehoben werden kann, wird in Hinblick auf die zeitliche Anforderbarkeit eine lineare Ordnung über den Betriebsmitteln definiert. Die Möglichkeit, dass eine zyklische Wartebedingung eintritt ist damit nicht mehr gegeben.

Alle vorgestellten Möglichkeiten, eine der vier Eigenschaften zu verhindern sind nicht besonders praktikabel und werden auch in den gängigen Betriebssystemen nicht eingesetzt. Gründe hierfür sind hohe Behinderung der Nebenläufigkeit von Prozessen und eine ineffiziente Betriebsmittelnutzung.

Deadlock–Vermeidung Bei der Vermeidung von Verklemmungen geht es darum, bei Ausführung der Prozesse zu überprüfen, ob einem Prozess weitere Betriebsmittel zugewiesen werden können. Die Deadlock–Vermeidung kann somit

der präventiven Deadlock–Erkennung zugeordnet werden. Dafür wurde beispielsweise der Bankiers-Algorithmus entwickelt. Alle entwickelten Algorithmen funktionieren nur, wenn die Betriebsmittelanforderungen der einzelnen Prozesse im vorhinein bekannt sind, was in der Praxis nicht realistisch ist.

Bei der reaktiven Erkennung von Verklemmungen muss geprüft werden, ob ein Zyklus im Betriebsmittelgraph vorliegt. Dies kann beispielsweise bei jeder Betriebsmittelanforderung eines Prozesses geschehen.

Deadlock–Lösung Zur Lösung von eingetretenen Verklemmungen werden folgende Strategien vorgeschlagen

Abbrechen aller Prozesse im Verklemmungsfall Dies ist die am einfachsten umzusetzende Strategie. Alle Prozesse die an einem Zyklus beteiligt sind werden terminiert.

Iteratives Abbrechen von Prozessen im Verklemmungsfall Es werden sukzessive Prozesse abgebrochen und nach jeder Terminierung getestet, ob noch ein Zyklus vorliegt.

Iteratives Freigeben von Betriebsmitteln Es werden sukzessive belegte Betriebsmittel den Prozessen entzogen. Um eine ordnungsgemäße Abarbeitung der Prozesse zu gewährleisten, sind Rollback–Mechanismen erforderlich, die diesen Prozess in den Zustand vor der Anforderung des Betriebsmittels versetzen.

Vergleich zwischen der Deadlock– und Policy–Konfliktbehandlung Die Deadlock–Verhinderung besitzt kein Pendant im Policy–Bereich, da die ersten drei Eigenschaften in einem policy–basierten Managementsystem nicht zwingend vorausgesetzt werden können.

Die Konflikterkennung funktioniert im Prinzip bei Prozessen und Policies analog (siehe Kapitel 4 und Abschnitt 5.3.2). Beide Mal wird die Einhaltung von Modelleigenschaften als Basis von Verklemmungen bzw. Policy–Konflikten angesehen.

Bei der Konfliktlösung sind die Strategien aus den beiden Bereichen analog (siehe Abschnitt 5.3.3). Es werden alle oder Teilmengen von Prozessen bzw. Policies terminiert bzw. nicht ausgeführt, um einen Konflikt zu lösen.

Bewertung

Das Problem von Verklemmungen und von Policy–Konflikten ist, wie in diesem Abschnitt gezeigt, ähnlich gelagert. Dies gilt sowohl für die Definition des

Problems als auch seiner Behandlung. Es konnte gezeigt werden, dass Verklemmungen auf einer abstrakten Ebene als ein Spezialfall von Policy–Konflikten betrachtet werden können, da die Verklemmungsdefinition eine Spezialisierung der Policy–Konfliktdefinition darstellt. Die Strategien zur Behandlung von Verklemmungen können auf den Bereich von Policy–Konflikten großteils übertragen werden.

3.4 Zusammenfassung

In diesem Kapitel wurden die existierenden Ansätze zur Policy–Konfliktbehandlung und verwandte Problemstellungen untersucht. Dabei wurden jeweils der Konfliktbegriff, die Konflikterkennung sowie Konfliktlösungsansätze analysiert. Wichtige Kriterien bei der Bewertung der Ansätze waren: Allgemeinheit des Ansatzes, Beschränkungen bezüglich der behandelbaren Konfliktarten, Limitierung auf eine dedizierte Policy–Sprache und methodische Anleitung zur Konfliktbehandlung.

Keiner der untersuchten Ansätze erfüllt alle diese Kriterien auf befriedigende Art und Weise. Die meisten Konfliktbehandlungen sind jeweils für genau eine Policy–Sprache konzipiert worden und setzten direkt auf die Grammatik der Sprache auf. Soll die Konfliktbehandlung auf eine andere Policy–Sprache übertragen werden, so müsste die Grammatik der Zielsprache geändert werden, was einem hohen Aufwand mit sich bringt. Da sich aber bis jetzt noch keine Policy–Sprache in der Praxis durchgesetzt hat bzw. für unterschiedliche Managementbereiche unterschiedliche Policy–Sprachen entwickelt wurden, bleiben somit diese Ansätze eine Insellösungen. Wie in diesem Abschnitt beschrieben, gehen fast alle Ansätze von einer Domänenüberlappung als notwendiges Kriterium für einen Konflikt aus. In Kapitel 2 konnte gezeigt werden, dass wichtige Konfliktarten existieren, die dieses Kriterium nicht erfüllen. Somit sind diese Ansätze beschränkt in der Anzahl der Konflikte, die behandelt werden können. Desweiteren geben die vorgestellten Arbeiten keine Methodik an, wie bei der Entdeckung einer neuen Konfliktart eine Konfliktbehandlung abgeleitet werden kann. Zusammenfassend muss festgestellt werden, dass die Defizite der Ansätze es erfordern, einen neuen Ansatz zur Konfliktbehandlung zu entwickeln. Dieser muss, aufbauend auf den existierenden Ansätzen, die aufgezeigten Defizite so gut wie möglich eliminieren.

Das nachfolgende Kapitel schlägt eine Methodik vor, wie basierend auf den Erkenntnissen aus den Kapiteln 2 und 3, eine Konfliktbehandlung für Konflikte abgeleitet werden kann. Zahlreiche Beispiele illustrieren die Anwendbarkeit und Nützlichkeit dieser Methodik. Dabei darf sich der Ansatz weder auf eine dedizierte Policy–Sprache abstützen, noch einen zu engen Konfliktbegriff wählen.

Kapitel 4

Entwicklung des Lösungskonzeptes

Kapitelüberblick

4.1 Einleitung

Im Kapitel 2 wurde gezeigt, dass in der Modellebene des formalen Konfliktmodells (Abschnitt 2.5.4) die Konflikte tatsächlich auftreten. Unter Zuhilfenahme der existierenden Modelle des Managements wird in diesem Kapitel ein Lösungskonzept vorgestellt.

Die Kernidee von Konfliktbehandlungsansätzen auf der Modellebene ist, die Managed Objects zu beobachten und unter Zuhilfenahme der a priori Modelle, in der Managed Objects eingebettet sind, Rückschlüsse auf den eingetretenen Konflikt zu ziehen. Modelle, wie sie beispielsweise das Common Information Model (CIM) spezifiziert, können als einzuhaltende Sammlung von Bedingungen betrachtet werden.

Die Verletzung einer Bedingung kann die Auswirkung eines Konflikts sein und ist somit Gegenstand der Konfliktbehandlung. Das heißt, dass die Verletzung von Bedingungen im weiteren Verlauf der Arbeit als initiales Ereignis betrachtet wird, um mit einer Konfliktbehandlung zu beginnen (siehe auch Abschnitt 2.5.4).

Im folgenden Abschnitt wird die Kernidee dieser Arbeit vorgestellt. Es wird eine allgemeine Methodik beschrieben, wie man systematisch von einem Modell zur zugehörigen Konfliktbehandlung gelangt. In Abschnitt 4.3 wird ein generisches Beziehungsmodell entwickelt, da Konflikte in Managementbeziehungen als wesentliche neue Konfliktart in dieser Arbeit identifiziert wurde. In den Abschnitten 4.4 bis 4.6 werden für Klassen des Beziehungsmodells (Assoziation, Komposition und funktionale Abhängigkeit) die Methodik angewandt. Für einen Vertreter von dynamischen Modellen, den endlichen Automaten, wird in Abschnitt 4.7 die Methodik angewandt.

4.2 Kernidee: Allgemeine Methodik zur Konfliktbehandlung auf Basis von a priori Modellen

Die Analyse des Konflikts in Abschnitt 2.5 hat gezeigt, dass Modelle einen wesentlichen Beitrag zur Konfliktanalyse leistet. Dieser Abschnitt beschreibt eine allgemeine Methodik zur Konfliktbehandlung, die für beliebige Modelle (beispielsweise Dienstabhängigkeitsgraphen oder endliche Automaten) anzuwenden ist.

Die Kernidee der Methodik ist:

Aus Modellen sind Invarianten ableitbar,
deren Nichteinhaltung können einen Konflikt anzeigen.

Eine allgemeine Methodik unterstützt dabei das Vorgehen von der Auswahl der Modelle bis zur Konfliktbehandlung. Die Methodik im Überblick stellt sich folgendermaßen dar:

Schritt 1 Auswahl eines Modells

Schritt 2 Invarianten aus dem Modellaspekt extrahieren

Schritt 3 Bezug der Aktionen zu den Invarianten spezifizieren

Schritt 4 Kritische Aktionen identifizieren

Schritt 5 Konfliktdefinition

Schritt 6 Konfliktlösung festlegen

Schritt 1: Auswahl eines Modellaspekts Im ersten Schritt der Methodik muss der zu betrachtende Modellaspekt ausgewählt werden. Eine Modellaspekt beschreibt ein Managementobjekt (z.B. das Verhalten) oder die Beziehung von Managementobjekten zueinander (z.B. ein Abhängigkeitsgraph). Für welche Modellaspekte man eine Konfliktbehandlung entwerfen will, kann von verschiedenen Kriterien abhängen. Beispielsweise kann ein Kriterium sein, ob und wie häufig eine Modellart in einem IT–Managementsystem eingesetzt wird. Ein anderes Kriterium könnte die Wichtigkeit eines Modellart hinsichtlich komplexer voneinander abhängiger Dienste sein. Hier wäre ein Dienstabhängigkeitsgraph ein wichtiger Modellaspekt.

Ein vereinfachtes Beispiel soll die Methodik in jedem Schritt illustrieren:

> **Beispiel:** Bei einem Provider sollen Enthaltenseinsgraphen eine wichtige Rolle spielen, deshalb soll eine Konfliktbehandlung für Enthaltenseinsmodelle entwickelt werden.

Schritt 2: Invarianten aus dem Modellaspekt extrahieren Eine Invariante beschreibt in einer formalen, auswertbaren Form einen Aspekt eines Modells. Die Invariante kann zu dem booleschen Wert `true` oder `false` ausgewertet werden. Die Invariante muss solange gültig sein, wie das Modell (z.B. ein funktionaler Abhängigkeitsgraph) gültig ist.

> **Beispiel:** Eine Invariante für eine Enthaltenseinsrelation lautet: das umfassende Managementobjekt muss mindestens genauso lange wie alle enthaltenen Managementobjekt existieren.

Diese Invarianten zu Modellaspekten sind allgemein gefasst und haben keinen Bezug zu Policies. Dieser Bezug wird im folgenden Schritt hergestellt.

Schritt 3: Bezug von Aktionen zu den Invarianten herstellen Policies führen Aktionen aus. Invarianten enthalten in ihrer Definition typischerweise keine Aktionen. Deshalb wird in diesem Schritt der Zusammenhang von Aktionen zu Invarianten hergestellt. Dazu müssen die Auswirkungen der Aktionen spezifiziert werden.

> **Beispiel:** Alle Aktionen, die umfassende Managementobjekte löschen oder enthaltene Managementobjekte erzeugen, haben einen Bezug zur Invariante aus Schritt 2.

Schritt 4: Kritische Aktionen identifizieren Nachdem der Bezug der Aktionen zu den Invarianten festgelegt wurde, werden in diesem Schritt 4 die Aktionen identifiziert, die zu einer Nichteinhaltung der Invariante beitragen. Diese Aktionen werden im weiteren Verlauf der Arbeit als *kritisch* bezeichnet und im sogenannten Konfliktraum zusammengefasst.

> **Beispiel:** Kritisch ist es, umfassende Managementobjekte zu löschen.

Schritt 5: Konfliktdefinition Alle wesentlichen Aspekte eines Konflikts sind durch Schritt 1 bis Schritt 4 ermittelt worden. Diese werden nun in der Konfliktdefinition zusammengefasst.

Beispiel: Ein Konflikt tritt ein, wenn ein umfassendes Managementobjekt gelöscht wird und darin noch Managementobjekte enthalten sind.

Schritt 6: Konfliktlösung festlegen Für jede Konfliktdefinition wird eine Konfliktlösung erarbeitet. Darin werden die spezifischen Merkmale der Konfliktdefinition berücksichtigt.

Beispiel: Bevor ein umfassendes Managementobjekt gelöscht werden kann wird überprüft, ob darin noch Managementobjekt enthalten sind.

Die dargestellte Methodik beschreibt, wie für einen Modellaspekt Konfliktbedingungen ableitetet werden, die als Basis für eine weiterführende Konfliktbehandlung dienen. Die Methodik ist generisch, da sie auf beliebige Modellaspekte anwendbar ist.

Jedes Modell hat charakteristische Eigenschaften, die sie von anderen Modellen unterscheidet. Aus den Modellaspekten werden Invarianten und schließlich Konfliktbedingungen für Policies abgeleitet. Damit eignen sich Modellaspekte auch als Charakteristikum für Policy–Konflikte. Das heißt, verschiedene Modellarten können als Klassifizierungsschema von Policy–Konflikten dienen.

Anhand von Enthaltenseinsmodellen, Modellen funktionaler Abhängigkeit und endlicher Automaten (Abschnitt 4.6, 4.5 und 4.7) wird die Tragfähigkeit der generischen Methodik demonstriert (Kapitel 4).

Modelle im Bereich des Managements sind Abstraktionen der realen Welt in Hinblick auf managementrelevante Aspekte. Es existieren zwei grundlegende Modellanwendungsfälle. Modelle können einen Ausschnitt der gegebenen realen Welt darstellen und haben damit einen abbildenden (passiven) Charakter. Man erfasst also den Istzustand eines Systems. Die zweite Möglichkeit Modelle einzusetzen besteht darin, eine Modellierung im vorhinein vorzunehmen, um damit einen gewünschten Sollzustand eines Systems zu beschreiben. Alle planerischen Modelle besitzen diesen Charakter. Dieser Anwendungsfall wird in [Ense 02] als '*a priori* Modellierung' bezeichnet und bildet in dieser Arbeit den Schwerpunkt. Alle Modellierungen in Standards sind typische Vertreter der *a priori* Modellierung. Die *a priori* Modellierung kann als eine Sammlung von Bedingungen interpretiert werden, die von dem zu manageneden System eingehalten werden müssen.

Ein *a priori* Modell beschreibt den Sollzustand eines Systems. Der Sollzustand ist in der Lage, Bedingungen für das gesamte System sowie seine Teile zu spezifizieren. Beispielsweise kann man eine Enthaltenseinsrelation auch als einzuhaltende Bedingung auffassen: ein Objekt A muss in einem Objekt B enthalten sein, weil das *a priori* Modell es vorschreibt. Ein Modell kann unterschiedliche Arten von Bedingungen enthalten. Aus den Bedingungsarten wird mit Hilfe der vorgestellten generischen Methodik eine Konfliktbehandlung entwickelt.

Modellarten im IT–Management Modelle im Management legen wie beschrieben das Managementobjekt selbst, oder die Beziehungen zwischen Managementobjekten fest. Managementmodellen können weiterhin *statische* Eigenschaften von Managementobjekt oder *dynamische* Eigenschaften beschreiben. Zu den statischen Modellen gehören beispielsweise Klassenmodelle und Objektmodelle. Beide Modellarten spielen bei Informationsmodellen, die einen objektorientierten Ansatz verfolgen, wie das OSI–Informationsmodell [ISO 10165-1] der ISO oder das Common Information Model der DMTF [CIM 2.2] eine wichtige Rolle. Andere Modellarten wiederum beschreiben das dynamische Verhalten von Managementobjekten. Vertreter dieser Modellart im IT–Management sind Prozessmodelle wie die Enhanced Telecom Operations Map eTOM des TeleManagement Forum [GB 921] oder Zustandsautomaten [ISO 10164-2].

In den folgenden Abschnitten wird die Anwendbarkeit der Methodik an je einem statischen und einem dynamischen Modellarten vorgestellt. Als Vertreter von statischen Modellarten werden Beziehungen zwischen Managementobjekten herangezogen und aus dem Bereich dynamischer Modellarten wird anhand von Zustandsautomaten die Methodik erläutert.

4.3 Entwicklung eines Beziehungsmodells

Die zu betrachtenden Modelle sind wie im letzten Abschnitt beschrieben, das Managementobjekt selbst oder Beziehungen zwischen Managementobjekten. Dieser Abschnitt untersucht Beziehungen im Management. Es wird ein hierarchisches Beziehungsmodell entwickelt. Dabei wird jede Beziehung als Klasse modelliert. Durch die Hierarchie ist es möglich, Eigenschaften der Beziehungsklassen zu vererben. Damit wird ein systematisches Ableiten von Modelleigenschaften erreicht.

Die Modelleigenschaften werden mit Hilfe der Object Constraint Language (OCL) ausgedrückt. Mit OCL kann man für UML Modelle in einfacher Art und Weise Constraints definieren (Abschnitt 4.3.2).

Für die modellierten Beziehungsklassen wird in nachgelagerten Schritten die Methodik angewandt (ab Abschnitt 4.4).

4.3.1 Generisches Beziehungsmodell

Der Sinn des generischen Beziehungsmodells ist es, Beziehungen so allgemeingültig zu definieren, dass es für eine möglichst breite Basis an Management–Informationsmodellen einsetzbar ist. Aktuell existieren im wesentlichen zwei Informationsmodelle, die eine explizite Modellierung von Beziehungen vorsehen: das Common Information Model (CIM) [CIM 2.2] und das General Relationship Management von OSI [ISO 10165-7]. Das generische Beziehungsmodell ist so angelegt, dass es für diese beiden Informationsmodelle genutzt werden kann.

Das Beziehungsmodell befindet sich vom Abstraktionsniveau auf der Ebene des CIM Metamodells. Das Beziehungsmodell beschreibt also nicht das Informationsmodell, sondern wie ein wohlgeformtes Informationsmodell (für Beziehungsklassen) aufgebaut ist.

Abbildung 4.1 verdeutlicht diesen prinzipiellen Zusammenhang der Modelle. In der rechten Spalte der Abbildung wird am Beispiel der Komposition die Abstraktionsniveaus dargestellt. In der untersten Ebene, der Ressourcenebene ist ein Router gezeigt, der 3 Schnittstellenkarten enthält (von maximal 6 möglichen). Das Instanzmodell der Komposition auf der Objektebene besteht aus den drei Schnittstellen (`if1`, `if2` und `if3`), und dem Router `r1`. Verbunden sind diese Objekte durch eine Kompositionsbeziehung, an deren Enden durch `Whole` bzw. `Part` die entsprechenden Beziehungsenden gekennzeichnet sind. Die Abstraktion des Instanzmodells entspricht auf der Modellebene einem Klassenmodell. Das Klassenmodell sagt aus, dass ein Router prinzipiell mit Interfaces in einer Kompositionsbeziehung steht und maximal 6 Interfaces enthalten kann. Auf der Metamodell–Ebene wird festgelegt, dass jede Kompositionsbeziehung aus einem Whole und einem Part bestehen muss.

Die Metamodellebene definiert eine formale Sprache (Syntax und Semantik), die zur Spezifikation von Modellaspekten geeignet ist. Beispiel für Modelle sind UML Core Model oder CIM Meta Model. Auf der Modellebene werden Modelle, beispielsweise Klassenmodelle spezifiziert. Diese sind (korrekte) Instantiierungen der Metamodelle. Auf der Objektebene werden Instanz– oder Implementierungsmodelle verwendet. Diese sind (korrekte) Instantiierungen der Klassenmodelle. Auf der Ressourcenebene werden keine Modelle angewandt, das Implementierungsmodell ist den tatsächlichen physischen Ressourcen am nächsten.

Das generische Beziehungsmodell wird aus Gründen der Übersichtlichkeit zweimal mit unterschiedlichem Detailgrad dargestellt. Das allgemeine Modell kon-

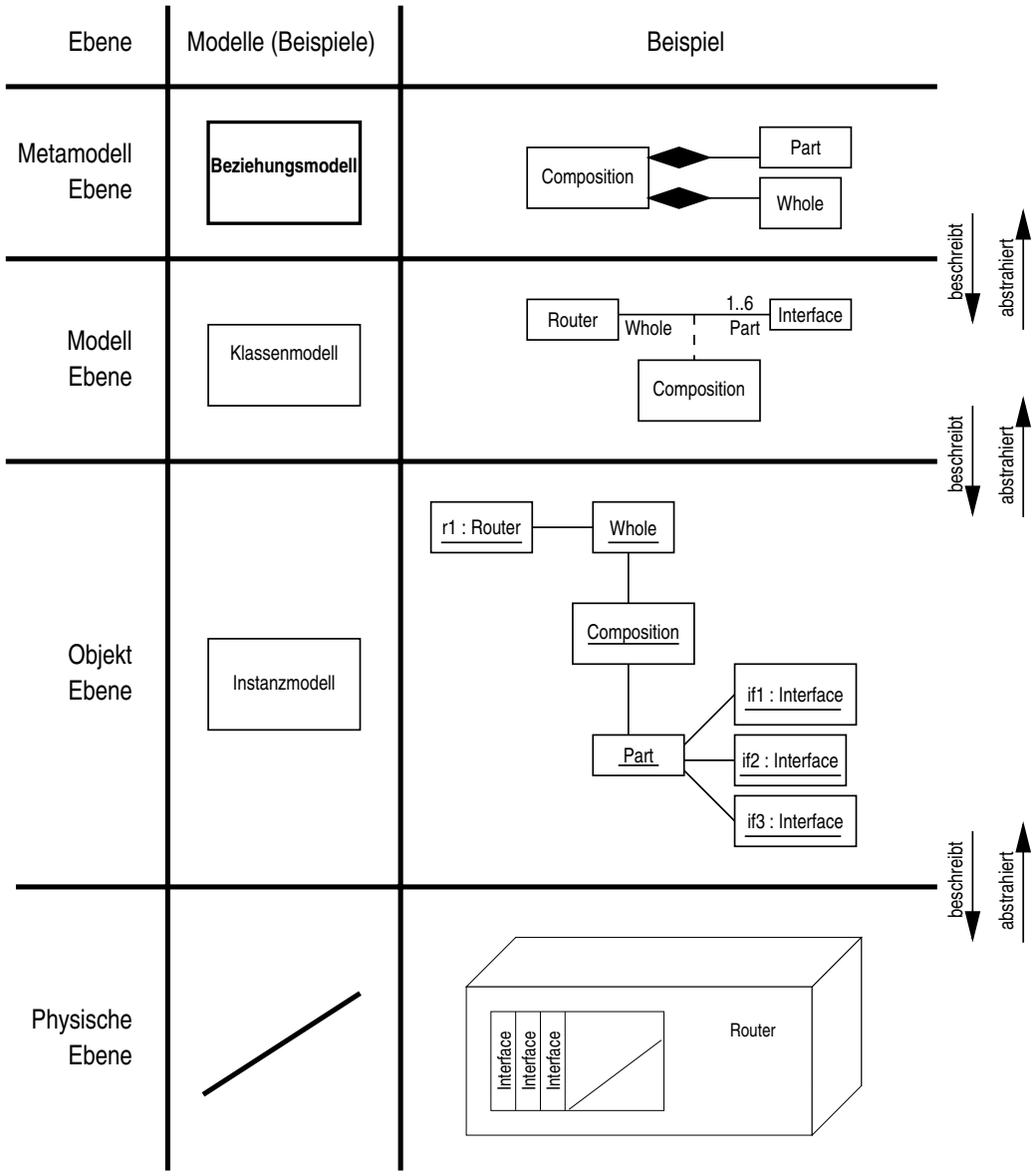

Abbildung 4.1: Zusammenhang der Modellarten

zentriert sich auf die hierarchische Strukturierung des Beziehungsmodells. Das detaillierte Modell erweitert das allgemeine Modell um die Modellierung der Verbindungsenden. Beide Modelle werden in UML dargestellt. Abbildung 4.2 hebt die hierarchische Struktur der Beziehungsklassen zueinander hervor, während Abbildung 4.3 alle Details des Beziehungsmodells darstellt.

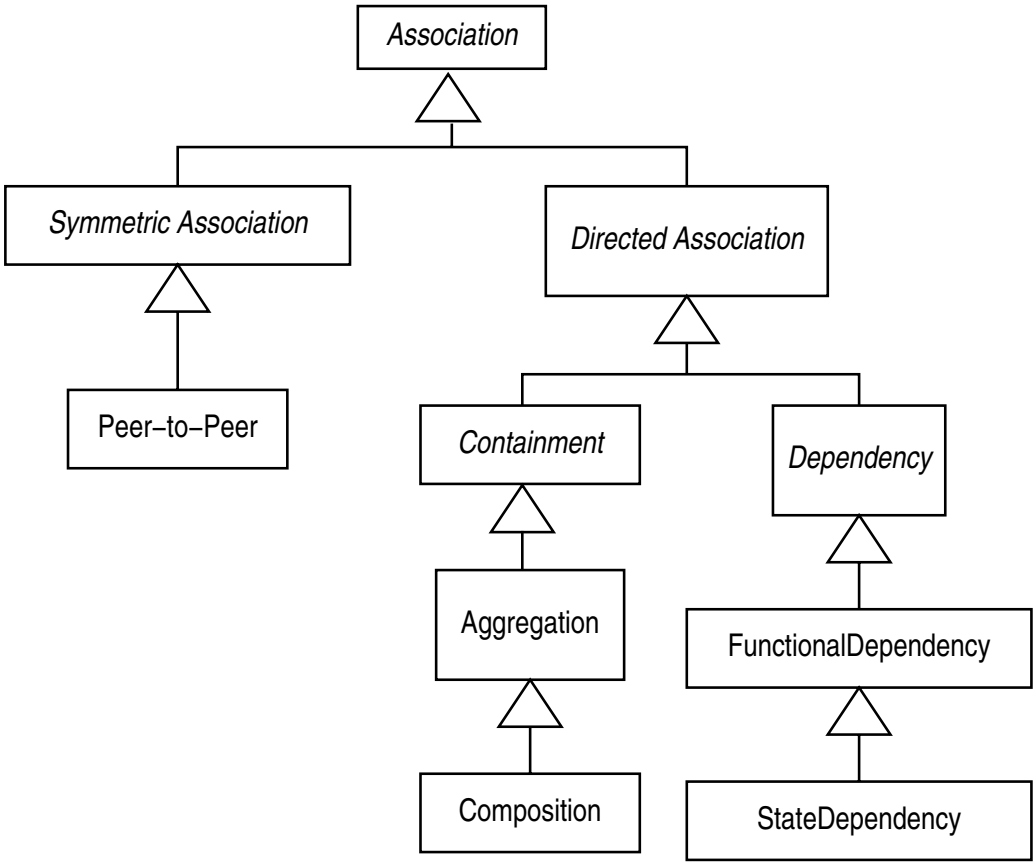

Abbildung 4.2: Hierarchische Struktur des Beziehungsmodells

Hierarchische Struktur der Beziehungsklassen Die abstrakte Klasse `Association` ist die Wurzel der Vererbungshierarchie (siehe Abbildung 4.2). Damit diese Klasse als Oberklasse aller Beziehungen fungieren kann, darf sie nur die allgemeinen Eigenschaften, die jede Beziehung auszeichnet, besitzen. Es werden in dieser Klasse beispielsweise keine Aussagen über eine mögliche Richtung der Beziehung getroffen.

Die beiden abstrakten Klassen `Symmetric Association` und `Directed Association` dienen der Strukturierung der Hierarchie. Die Klasse

`Symmetric Association` stellt Beziehungen dar, in der die Eigenschaft der Symmetrie der Beziehung wesentlich ist. Beispielsweise sind Peer–to–Peer Beziehungen symmetrisch. Gerichteten Beziehungen, wie sie durch die Klasse `Directed Association` repräsentiert werden, lässt sich eine Richtung zuordnen.

Die abstrakte Klasse `Containment` ist die Oberklasse aller Enthaltenseinsbeziehungen. Die Klasse `Aggregation` ist die allgemeinste instantiierbare Klasse einer Enthaltenseinsbeziehung. Die Klasse `Composition` ist eine Verfeinerung der Klasse `Aggregation`, denn sie stellt spezifischere Anforderungen an die korrekte Verwendung, wie in Abschnitt 4.6 bei der Definition der Klasse beschrieben wird.

Die abstrakte Klasse `Dependency` ist die Oberklasse aller Abhängigkeiten. Eine Ableitung der Klasse ist die abstrakte Klasse `FunctionalDependency`. Diese abstrakte Klasse fungiert als Oberklasse für alle funktionalen Abhängigkeiten zwischen Objekten. Die Klasse `StateDependency` betrachtet als wesentliche Eigenschaft der funktionalen Abhängigkeit den Zustand der gebundenen Instanzen.

Detaillierte Darstellung des Beziehungsmodells Abbildung 4.3 stellt das Beziehungsmodell inklusive der Modellierung der Beziehungsenden dar.

Die Klasse `Association` besitzt Beziehungsenden, welche durch die abstrakte Klasse `AssociationEnd` dargestellt werden. Diese Klasse ist die Oberklasse aller Verbindungsenden und besitzt das Attribut `multiplicity`, welches die mögliche Anzahl der mit dem Verbindungsende verknüpfte Instanzen anzeigt. Die Klasse `Association` besitzt die generischen Methoden `create()` und `delete()` (geerbt von der Klasse `ManagedObject`, sowie `bind()` und `unbind()`. Die Methode `create()` erzeugt eine Instanz der Klasse, `delete()` löscht die Instanz. Während der Lebensdauer einer Beziehung kann mittels `bind()` ein Objekt an ein Assoziationsende gebunden werden und mit der Methode `unbind()` kann ein Objekt aus der Assoziation getrennt werden.

An die Assoziationsenden können beliebige Instanzen der Klasse `ManagedObject` gebunden werden.

Die abstrakte Klasse `NavigableEnd` ist eine Verfeinerung der Klasse `AssociationEnd` und ordnet der Beziehung eine Richtung zu. Die Beziehungsenden `Whole` und `Part`, eine Verfeinerung der Klasse `NavigableEnd` werden der Klasse `Aggregation` zugeordnet. Der Klasse `Dependency` werden die Verbindungsenden `Antecedent` und `Dependent` zugeordnet. Diese zeigen, an welchem Ende einer Beziehung der Abhängige und Bereitsteller einer funktionalen Abhängigkeitsbeziehung angesiedelt sind.

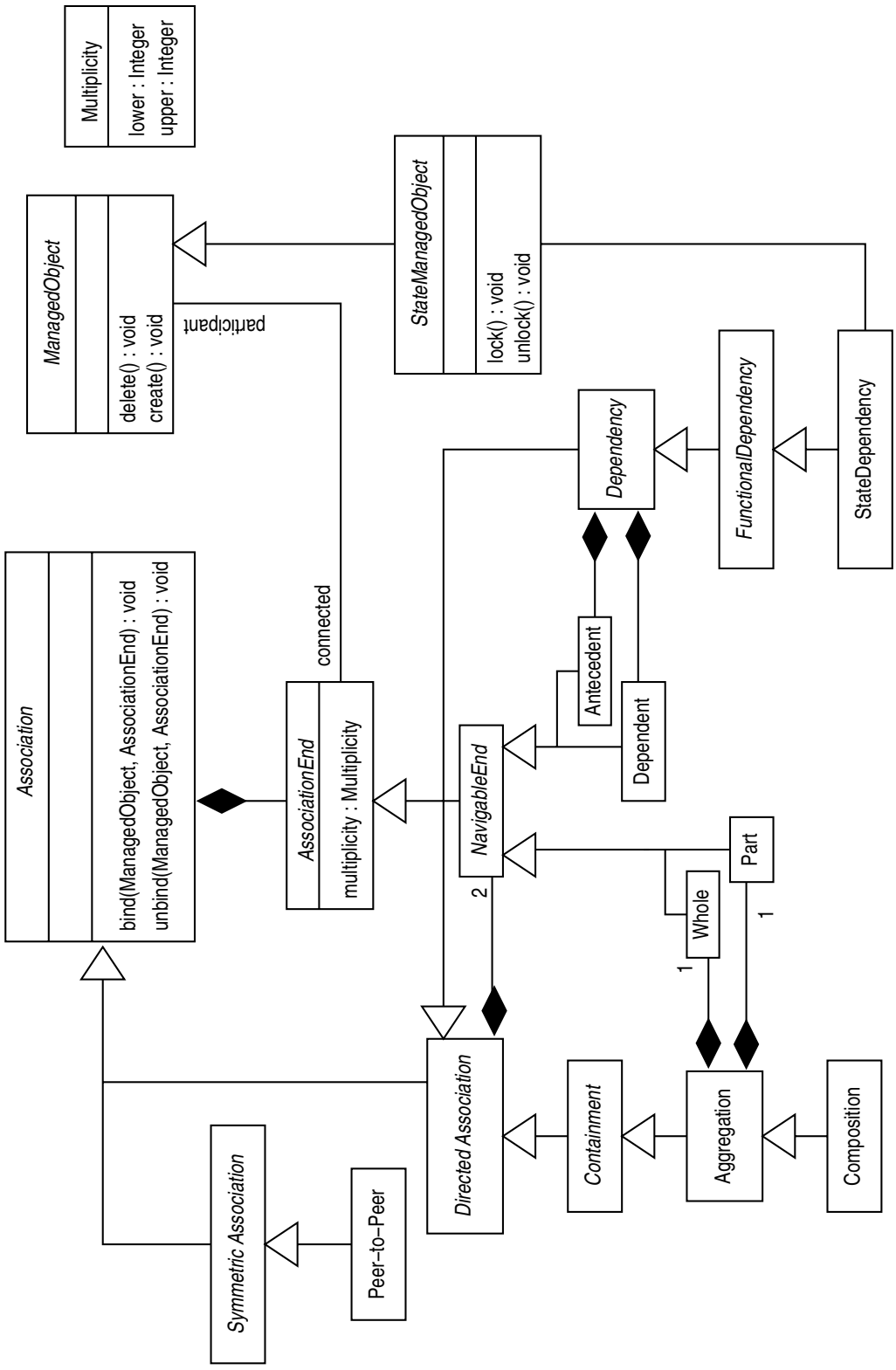

Abbildung 4.3: Beziehungsmodell mit Verbindungsenden–Modellierung

Das Beziehungsmodell zeigt, dass Beziehungsklassen zueinander in einer Vererbungsbeziehung stehen. Die hierarchische Struktur ist für die Ableitung von Konflikten nützlich, da auch Konflikte vererbt werden.

4.3.2 Die Object Constraint Language

Die Object Constraint Language (OCL) [uml 03] ist eine formale Sprache, deren Ziel es ist, Constraints ausdrücken zu können. OCL, welches Teil des UML Standard ist, wurde entwickelt, um in UML Modellen Constraints ausdrücken zu können. Beispielsweise werden die 'well-formedness rules' des UML Metamodells in OCL ausgedrückt. OCL wird verwendet, um Invarianten für Klassen sowie Vor– und Nachbedingungen für Methoden einer Klasse spezifizieren zu können. An einem UML Klassendiagramm werden die wesentliche Eigenschaften von OCL exemplarisch vorgestellt.

Abbildung 4.4 zeigt die beiden Klassen `Person` und `Bank`, die über die Beziehungsklasse `Konto` miteinander verbunden sind. Das `Konto` hat einen aktuellen `stand` und mithilfe der Methode `abheben()` kann eine `Person` Geld abheben.

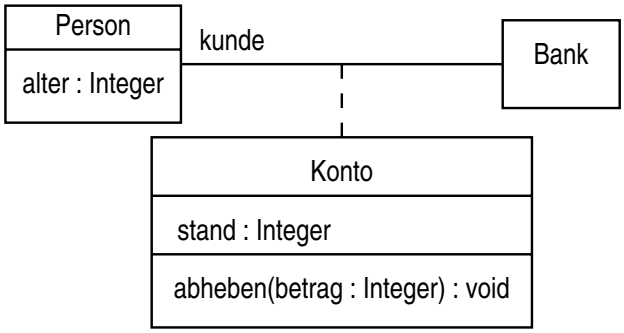

Abbildung 4.4: Beispiel eines UML–Klassendiagramm

Jedes Constraint muss angeben, worauf es sich bezieht. Dieser sogenannte Kontext kann ein Klasse oder die Methode einer Klasse sein. Folgende Invariante drückt aus, dass das Alter einer Person größer gleich Null ist:

```
context  Person
inv:     self.alter >= 0
```

Das Konstrukt **`context`** `Person` gibt an, dass sich das Constraint auf die Klasse `Person` bezieht. Das Schlüsselwort `inv` zeigt an, das es sich um eine Invariante

handelt. Eine Invariante muss während der gesamten Lebensdauer einer Instanz einer Klasse gültig sein. Das Schlüsselwort `self` bezieht sich auf die Instanz der Klasse. Der Ausdruck `self.alter >= 0` bedeutet, dass bei keiner Instanz der Klasse `Person` das Attribut `alter` negativ sein darf.

Ein Beispiel einer Nachbedingung für die Methode `abheben()` der Klasse `Konto` lautet:

```
context  Konto::abheben(betrag : integer) : void
post:    self.stand= (self.stand)@pre - betrag
```

Eine Nachbedingung wird durch das Schlüsselwort `post` angezeigt. Eine Nachbedingung drückt eine Bedingung aus, die direkt nach der Terminierung der Methode gelten muss. Mit dem Schlüsselwort `@pre` bezieht man sich auf den Wert eines Attributs, den es vor der Ausführung der Methode hatte. So gibt die Nachbedingung `self.stand= (self.stand)@pre - betrag` den Kontostand nach Ausführung der Methode `abheben()` an.

Ein Beispiel einer Vorbedingung für die Methode `abheben()` der Klasse `Konto` lautet:

```
context  Konto::abheben(betrag : integer) : void
pre:     self.stand - betrag >= 0
```

Eine Vorbedingung (angezeigt durch `pre`) drückt aus, das die Methode `abheben()` nur ausgeführt wird, wenn der Kontostand nicht negativ wird.

Es ist auch möglich, ausgehend von einem Kontext durch das Modell entlang der Assoziationen zu navigieren:

```
context  Bank
inv:     self.kunde.alter >= 18
```

Die Navigation von der Klasse `Bank` zur Klasse `Person` erfolgt durch den angegebenen Rollennamen (`kunde`). Ist kein Rollenname angegeben, so wird direkt der Klassename benutzt. Das Ergebnis einer Navigation (`self.kunde`) ist ein Menge. Im Beispiel ist es die Anzahl der Instanzen der Klasse (`Person`) die als `kunde` in Beziehung zur Klasse `Bank` stehen. Jeder Kunden der Bank muss mindestens 18 Jahre alt sein.

Mit Hilfe von OCL ist es in einfacher und kompakter Weise möglich, für das in Abschnitt 4.3.1 definierte Beziehungsmodell für jede Beziehungsklasse die einzuhaltenden Bedingungen zu spezifizieren.

4.4 Anwendung der Methodik für die Beziehungsklasse `Association`

Zur Spezifikation der Constraints wurde der Ansatz gewählt, dass nur die unbedingt notwendigen Constraints spezifiziert werden. Das erhöht zum Einen die Übersichtlichkeit, denn es werden möglichst wenige Constraints spezifiziert und zum Anderen wird eine breite Anwendbarkeit der Klassen erreicht, da Constraints immer auch eine Einschränkung der Nutzbarkeit bedeuten.

Die Methodik beginnt hier mit Schritt 2, das Schritt 1 bereits durch die Auswahl des Modellaspekts `Association` geschehen ist.

Schritt 2: Invarianten aus dem Modellaspekt extrahieren Für die Oberklasse `Association` werden Invarianten bezüglich der Kardinalitätsgrenzen spezifiziert. Weitere Einschränkungen werden für diese abstrakte Oberklasse nicht getroffen.

```
context          Association
inv upperLimit:  self.AssociationEnd.participant →size() <=
                 self.AssociationEnd.multiplicity.upper
```

Invariante 1

Die Invariante 1 besagt, dass die Anzahl der an ein Assoziationsende gebundenen Instanzen die Obergrenze zu keinem Zeitpunkt der Lebensdauer der Assoziation überschreiten darf.

Die symmetrische Invariante für die Untergrenze lautet:

```
context          Association
inv lowerLimit:  self.AssociationEnd.participant →size() >=
                 self.AssociationEnd.multiplicity.lower
```

Invariante 2

Die Invariante 2 besagt, dass die Anzahl der an ein Assoziationsende gebundenen Instanzen die Untergrenze zu keinem Zeitpunkt der Lebensdauer der Assoziation unterschreiten darf.

Schritt 3: Bezug von Aktionen zu den Invarianten herstellen Die Aktionen sind in diesem Fall die Methoden der Klassen. Die Klasse `Association` besitzt die beiden Methoden `bind()` und `unbind()`. Um den Bezug der Methoden zu den Invarianten zu beschreiben muss deren Wirkung beschrieben werden. Dies geschieht durch die Spezifikation der Nachbedingungen der Methoden.

Die Nachbedingung der Methode `bind()` lautet:

```
context  Association::bind(mo : ManagedObject, ae :
         AssociationEnd)
  post:  mo.connected.ae and
         (self.ae→size() = (self.ae.→size())@pre +1)
```

Nachbedingung 1

Die Methode `bind()` bindet eine Instanz (`mo`) an ein Assoziationsende (`ae`), die Anzahl der an diesem Assoziationsende gebundenen Instanzen erhöht sich nach Ausführung der Methode um eins. Die Methode hat somit einen Bezug zu beiden Invarianten 1 und 2, da sie die Anzahl der gebundenen Instanzen verändert.

Die Nachbedingung der Methode `unbind()` lautet:

```
context  Association::unbind(mo : ManagedObject, ae :
         AssociationEnd)
  post:  not (mo.connected.ae) and
         (self.ae→size() = (self.ae.→size())@pre -1)
```

Nachbedingung 2

Die Methode `unbind()` trennt eine Instanz (`mo`) aus dem Assoziationsende (`ae`), die Anzahl der an diesem Assoziationsende gebundenen Instanzen erniedrigt sich um eins. Die Methode hat somit einen Bezug zu beiden Invarianten 1 und 2, da sie die Anzahl der gebundenen Instanzen verändert.

Schritt 4: Kritische Aktionen identifizieren Die Identifikation von kritischen Aktionen erfolgt je Invariante. Die Frage, die in diesem Schritt beantwortet werden muss, lautet: 'Existiert eine Abfolge von Aktionen, so dass die Invariante verletzt wird?'. Für Invariante 1 sind Aktionenfolgen zu suchen, welche die Invariante verletzt, also die Obergrenze überschritten wird. Das geschieht, wenn durch den Aufruf von `bind()` Methoden mehr Instanzen an einem Assoziationsende gebunden werden sollen, als von der Obergrenze erlaubt. Damit befinden sich `bind()` Methoden in einem Konfliktraum.

Analoges gilt für die Invariante 2. Es dürfen zu keinem Zeitpunkt mehr Instanzen aus einem Assoziationsende durch die Methode `unbind()` getrennt werden als in der Untergrenze spezifiziert wurde. Damit befinden sich `unbind()` Methoden in einem Konfliktraum.

Schritt 5: Konfliktdefinition Die Konfliktdefinition findet wiederum getrennt für jede Invariante statt.

Die Invariante 1 wird verletzt (zu ´falsch´ ausgewertet), wenn mehr Instanzen als durch die Obergrenze erlaubt an ein Assoziationsende gebunden werden. Das bedeutet, dass die Methode `bind()` in einem potentiellen Konflikt bezüglich der Invariante 1 steht.

conflict	`upperLimit`
context	`Association`
conflict space:	`bind(mo1 : ManagedObject, ae1 : AssociationEnd)` `bind(mo2 : ManagedObject, ae2 : AssociationEnd)`
pre:	`self→exists(ae : AssociationEnd | ae=ae1 and ae=ae2)`
refers to:	`invariant upperLimit`

Konflikt 1

Um den Konflikt zu beschreiben, wird die OCL um die Konstrukte `conflict`, `conflict space` und `refers to` erweitert. Das Konstrukt `conflict` gibt den Namen des Konflikts an. Das Feld `conflict space` gibt alle am Konflikt beteiligten Methoden an. Das Feld `refers to` gibt an, auf welche Invariante sich die Konfliktdefinition bezieht.

Für Konflikt 1 bedeuten die einzelnen Konstrukte:

`conflict upperLimit` ist der Name des Konflikts.

`context Association` Der Konflikt bezieht sich auf die Klasse `Association`.

`conflict space :`
`bind(mo1 : ManagedObject, ae1 AssociationEnd)`
`bind(mo2 : ManagedObject, ae2 : AssociationEnd)`
Der Konfliktraum besteht aus `bind()` Methoden.

`pre: self→exists(ae : AssociationEnd | ae=ae1 and ae=ae2)`
In diesem Feld wird eine Einschränkung des Konfliktraums definiert. Er besagt, dass nur solche Methoden aus dem Konfliktraum potentiell in Konflikt stehen, welche die spezifizierte Bedingung erfüllen. Die Bedingung besagt, dass es sich bei den `bind()`–Methoden um die identischen Assoziationsenden handeln muss (`ae=ae1 and ae=ae2`).

`refers to: Invariante UpperLimit`
Diese Konfliktdefinition bezieht sich auf die Invariante `UpperLimit` (Invariante 1).

Nun wird die Konfliktdefinition für die Invariante 2 spezifiziert:

conflict	`lowerLimit`
context	`Association`
conflict space:	`unbind(mo1 : ManagedObject, ae1 : AssociationEnd)` `unbind(mo2 : ManagedObject, ae2 : AssociationEnd)`
pre:	`self→exists(ae : AssociationEnd | ae=ae1 and ae=ae2) and self.multiplicity.lower > 0`
refers to:	`invariant lowerLimit`

Konflikt 2

`unbind()` Methoden sind in einen potentiellen Konflikt, wenn das Assoziationsende der `unbind()` Methoden identisch ist sowie die Untergrenze größer Null ist. Ist die Untergrenze gleich Null, so kann kein Konflikt eintreten, da nicht mehr Instanzen entbunden werden können als ursprünglich eingebunden wurden.

Schritt 6: Konfliktlösung festlegen Eine Konfliktlösung wird hier nicht ausführlich erörtert, da es sich bei der Klasse `Association` um eine abstrakte Klasse handelt, die nicht instantiierbar ist. In Abschnitt 4.6 werden die beiden Konflikte in der Konfliktlösung genau untersucht.

Eine Art der Konfliktlösung, nämlich die Konfliktvermeidung kann bereits hier spezifiziert werden. Bei der Konfliktvermeidung wird der Aufruf der Methoden so gesteuert, dass kein Konflikt auftreten kann. Das wird dadurch erreicht, dass zu jeder Methode eine Vorbedingung spezifiziert wird. In dieser Arbeit wird jede Vorbedingung so spezifiziert, dass nach Ausführung der Methode die betroffenen Invarianten eingehalten werden. Für die Methode `bind()` beispielsweise muss die Vorbedingung darauf achten, dass die Obergrenze (Invariante 1) durch die Ausführung der Methode nicht erreicht ist. Zur Spezifikation der Vorbedingungen werden einerseits die bereits definierten Nachbedingungen und andererseits die einzuhaltenden Invarianten beachtet.

Die Vorbedingung der `bind()` Methode der Klasse `Association` lautet:

```
context Association::bind(mo : ManagedObject, ae :
        AssociationEnd)
   pre: self.ae.participant →size() <
        self.ae.multiplicity.upper
```

Vorbedingung 1

Diese Vorbedingung 1 der Methode `bind()` Klasse `Association` besagt, dass die Anzahl der an einem Assoziationsende gebundenen Instanzen die Obergrenze noch nicht erreicht haben darf. Mit dieser Vorbedingung ist sichergestellt, dass noch mindestens eine Instanz eingebunden werden kann. Wird die Vorbedingung vor der Ausführung der Methode zu 'falsch' ausgewertet, so wird die Aktion nicht ausgeführt.

Die Einhaltung der Untergrenze (Invariante 2) wird durch die Methode `unbind()` gefährdet. Deshalb definiert man folgende Vorbedingung:

```
context Association::unbind(mo : ManagedObject, ae
        : AssociationEnd) : void
   pre: self.ae.participant →size() >
        self.ae.multiplicity.lower
```

Vorbedingung 2

Die Vorbedingung der Methode `unbind()` der Klasse `Association` besagt, dass die Anzahl der an einem Assoziationsende gebundenen Instanzen die Untergrenze noch nicht erreicht haben darf, um ein Instanz aus dem Assoziationsende zu trennen.

Fazit Für die Oberklasse `Association` konnten zwei Konflikte identifiziert werden. Beide Konflikte behandeln die Einhaltung der Kardinalitätsgrenzen. Hier zeigt sich deutlich der Vorteil des Aufstellen eines objektorientierten Modells von Beziehungsklassen (Abbildung 4.3). Man kann sich bei jeder Klasse auf die Spezifika dieser Klasse konzentrieren. Denn man erbt neben den Attributen und Methoden der Oberklasse auch die bereits definierte Konfliktdefinition. Das heißt, für alle von der Klasse `Association` abgeleiteten Klassen (die alle die `bind()` und `unbind()` Methoden besitzen) ist das Problem der Einhaltung der Kardinalitätsgrenzen bereits gelöst. Somit wurde durch die Gliederung der Beziehungsklassen in eine Vererbungshierarchie auch das Konfliktproblem untergliedert und damit handhabbarer gemacht.

4.5 Anwendung der Methodik bei funktionaler Abhängigkeit

Wie in der Einleitung zu diesem Kapitel festgestellt (Abschnitt 4.1), werden Beziehungen zwischen Managementobjekten analysiert. Da funktionale Abhängigkeiten eine der fundamentalen Beziehung zwischen Managementobjekten darstellt, sind sie auch für die Konfliktbehandlung von besonderem Interesse.

Das grundlegende Prinzip, komplexe Aufgaben in überschaubare und beherrschbar Einzelteile zu gliedern, wird in den meisten Bereichen der Wissenschaft und Produktion von technischen Systemen angewendet. Die Einzelteile erbringen zusammen die gewünschte Funktionalität und stehen somit in Relation zueinander. Funktionale Abhängigkeiten sind eine der wichtigsten Beziehungen, die im Management modelliert und überwacht werden. Durch die Aufteilung sowohl der Hardware als auch der Software einer IT-Infrastruktur in einzelne Komponenten, die gemeinsam eine Funktion erbringen, sind funktionale Abhängigkeiten ein elementarer Bestandteil eines Systems. An den bekannten Paradigmen wie Client—Server bzw. Schnitt- und Schichtbildung, lässt sich eine funktionale Aufteilung ablesen.

Funktional abhängig bedeutet, dass die Funktion einer Ressource nur voll erbracht werden kann, wenn die Ressource, von der sie abhängt, funktionsfähig und zugreifbar ist. Beispielsweise ist der verbindungsorientierte Punkt-zu-Punkt-Dienst TCP von IP durch den Dienstschnitt, der beide trennt, funktional abhängig.

4.5.1 Beispielszenario eines funktionalen Abhängigkeitsgraphen

Abbildung 4.5 zeigt eine Instanzmodell mit expliziter Darstellung der Beziehung durch Assoziationsklassen.

Die Objekte `database` und `serverA` sind Instanzen der Klasse `StateManagedObject` , haben einen Zustand und können mittels den Methoden `unlock()` und `lock()` den Zustand ändern (siehe Abbildung 4.5). Die Assoziation ist eine Instanz der Klasse `StateDependency`. Im Folgenden werden die Begriffe *Abhängiger* für `Dependency` und *Bereitsteller* für `Antecedent` nach [Ense 02] verwendet.

Demnach ist die Datenbank `database` der Abhängige und der Bereitsteller der Server `serverA`. Die korrekte Funktion der Datenbank ist von der Funktion des Servers `serverA` abhängig. Eine Instanz `sd1` der Klasse `StateDependency` verbindet Bereitsteller und Abhängiger und macht die Abhängigkeit explizit.

Die Managementobjekte `database` und `serverA` sind in die Domänenstruktur eingebettet, wie in Abbildung 4.6 zu sehen.

Folgende drei Policies aus Abschnitt 2.5.1 sind zur Steuerung der Knoten des Graphen für den funktionalen Abhängigkeitsgraphen gegeben und werden nochmals kurz erläutert:

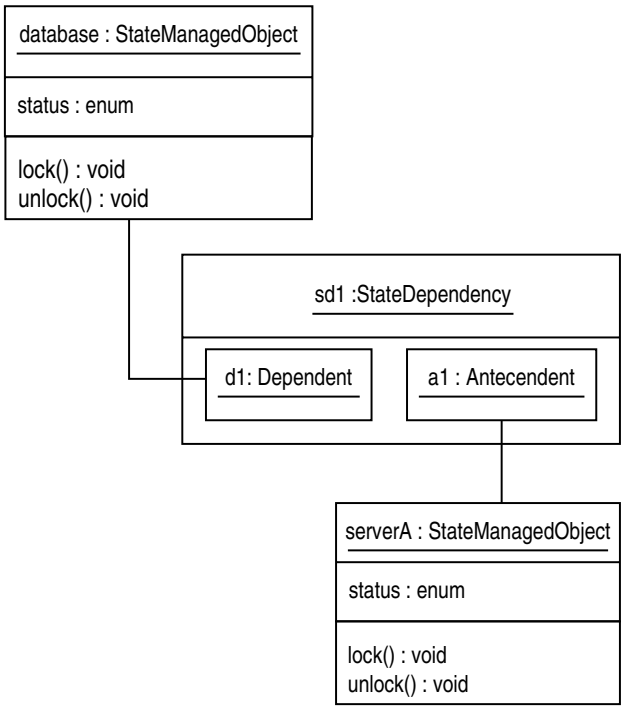

Abbildung 4.5: Beispiel eines Instanzmodells mit funktionalen Abhängigkeiten der Klasse `StateDependency`

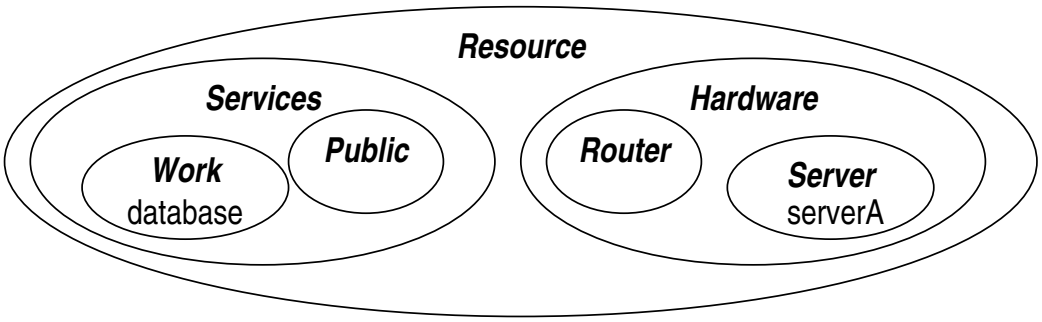

Abbildung 4.6: Domänenzugehörigkeit der Managementobjekte

Policy	`operational policy1.1`
Subject	`Security Manager`
Event	`beginOfWork`
Target	`/resource/hardware/server`
Action	`lock(); update(secManager.getPackage()); reboot(); unlock();`
Condition	`secManager.updateAvailable();`

Policy	`operational policy2.1`
Event	`beginOfWork`
Subject	`Performance Manager`
Target	`/resource/services/work`
Action	`unlock();`

Policy	`operational policy2.2`
Event	`endOfWork`
Subject	`Performance Manager`
Target	`/resource/services/work`
Action	`lock();`

Die Policy `operational policy1.1` hat zusammenfassend folgende Bedeutung: Tritt das das Ereignis `beginOfWork` ein, so werden auf allen Servern der Firma (angezeigt durch die Domäne `/resource/hardware/server`) das Update eingespielt, wenn ein neues Update vorhanden ist. Für den Update müssen mehrere Aktionen ausgeführt werden: Zuerst muss der Server zur Nutzung gesperrt `lock();` und anschließend das Update eingespielt `update(swPackage)` werden. Dann wird der Server gebootet und zur Nutzung freigegeben (`reboot(); unlock();`).

Die Policy `operational policy2.1` gibt zu Beginn der Arbeitszeit (angezeigt durch das Ereignis `beginOfWork`) die Dienste der Domäne `/resource/services/work` zu Beginn der Arbeitszeit durch die Aktion `unlock()` frei. Die Policy `operational policy2.2` sperrt alle Dienste der gleichen Domäne am Ende der Arbeitszeit (angezeigt durch das Ereignis `endOfWork`) und die Aktion `lock()`.

4.5.2 Anwendung der entwickelten Methodik für die Klasse `StateDependency`

Da die dynamischen Aspekte, also die Änderung des Systemzustandes durch Managementoperationen, von Interesse sind, werden jene Managementoperationen untersucht, die einen gegebenen funktionalen Abhängigkeitsgraphen beeinflussen.

Schritt 2: Invarianten aus dem Modellaspekt extrahieren Zur Beschreibung funktionaler Abhängigkeiten existieren in der Literatur unterschiedliche Möglichkeiten. [Grus 99] stellt eine Klassifikation funktionaler Abhängigkeiten auf. Die Klasse `StateDependency` stellt *eine* der Möglichkeiten dar, eine funktionale Abhängigkeit zu beschreiben. Für die Klasse `StateDependency` werden folgende Invarianten spezifiziert:

```
context          StateDependency
inv stateUpper: self.Antecedent.multiplicity.upper=1
```

Invariante 3

Es kann höchstens eine Instanz Bereitsteller (`Antecedent`) gebunden werden.

```
context       StateDependency
inv stateDep: self.Dependent.participant→exists( mo |
              mo.status='Unlocked')
              implies
              self.Antecedent.participant.status='Unlocked'
```

Invariante 4

Sind Instanzen voneinander funktional abhängig, so wird gefordert, falls sich ein Abhängiger im Zustand `Unlocked` befindet, muss sich auch der Bereitsteller im Zustand `Unlocked` befinden. Die Invariante 4 bezieht sich auf die Zustände der Instanzen, die in an den Assoziationsenden gebunden sind.

Schritt 3: Bezug von Aktionen zu den Invarianten herstellen Die Invariante 4 bezieht sich auf die Zustände der Instanzen, die an den Assoziationsenden gebunden sind. Das bedeutet, dass die Methoden der Klasse `StateManagedObject` zu untersuchen sind, ob sie einen Beitrag bezüglich der Invariante 4 leisten.

Bezüglich der Invariante 3 müssen keine weiteren Schritte der Methodik ausgearbeitet werden, da Konflikte bezüglich den Kardinalitätsgrenzen bereits in der Oberklasse `Association` (Abschnitt 4.4) behandelt wurde.

Die zu betrachtenden Methoden der Klasse `StateManagedObject` sind `lock()` und `unlock()` . Somit müssen Nachbedingungen für die Methoden spezifiziert werden:

```
context StateManagedObject ::lock() : void
  post: self.status='Locked'
```

Nachbedingung 3

Nach Terminierung der Methode `lock()` befindet sich die Instanz im Zustand `Locked` .

```
context StateManagedObject ::unlock() : void
  post: self.status='Unlocked'
```

Nachbedingung 4

Nach Terminierung der Methode `unlock()` befindet sich die Instanz im Zustand `Unlocked` .

Beide Methoden der Klasse `StateManagedObject` beeinflussen die Invariante 4.

Schritt 4: Kritische Aktionen identifizieren Kritische Aktionen sind jene Aktionen, die dazu beitragen, dass die Invariante 4 zu 'falsch' ausgewertet wird. Dies sind in diesem Fall die Methode `unlock()` , die ein Abhängiger ausführt und die Methode `lock()` , die der Bereitsteller ausführt. Somit sind die Methoden `lock()` und `unlock()` unter diesem Bedingungen kritische Aktionen.

Nun müssen die Policies identifiziert werden, die kritische Aktionen enthalten. Dazu werden die angegebenen Domänen der Policies ausgewertet. Für jedes Element der Domäne wird getestet, ob das Managementobjekt ein Abhängiger (`Dependent`) bzw. Bereitsteller (`Antecedent`) einer `StateDependency` ist und ob die entsprechenden kritischen Aktionen in dem `Action`–Teil der Policy spezifiziert ist. Bezogen auf das Beispielszenario sind somit folgende Policies im Konfliktraum:

$Konfliktraum_{inv\ stateDep}$ ={`operational policy 1.1`, `operational policy 2.1`}

Die Policy `operational policy 2.1` ist nicht im Konfliktraum vertreten, da die `lock()` –Operation von einem Abhängigen ausgeführt wird.

Schritt 5: Konfliktdefinition Ein Konflikt tritt ein, wenn die Invariante 4 zu 'falsch' ausgewertet wird. Die dort definierte Implikation wird durch die im Konfliktraum definierten Methoden zu 'falsch' ausgewertet, wenn die spezifizierte Vorbedingung zutrifft:

conflict	StateDep
context	StateDependency
conflict space:	ManagedObject [mo1]::lock() ManagedObject [mo2]::unlock()
pre:	self.Antecedent =mo1 and self.Dependent =mo2
refers to:	invariant stateDep

Konflikt 3

Die Vorbedingung reduziert alle möglichen `ManagedObject` auf diejenigen die durch eine `StateDependency` verbunden sind und der `Antecedent` die `lock()` –Operation aufruft und der `Dependency` die `unlock()` –Operation. Der Konflikt tritt unabhängig von der Ausführungsreihenfolge der im Konfliktraum befindlichen Methoden auf:
Fall 1: `mo1.lock(); mo2.unlock();` oder
Fall 2: `mo2.unlock(); mo1.lock();`
Beide Reihenfolgen der Ausführung erzeugen einen Konflikt.

Zwei weitere Konflikte sind bezogen auf die spezifizierten Nachbedingungen aus dem letzten Schritt möglich. Es können gleichzeitig `lock()` oder `unlock()` auf den beiden Assoziationsenden aufgerufen werden. Beidemale kann ein Konflikt eintreten:

conflict	StateDepLock
context	StateDependency
conflict space:	ManagedObject [mo1]::lock() ManagedObject [mo2]::lock()
pre:	self.Antecedent =mo1 and self.Dependent =mo2
refers to:	invariant stateDep

Konflikt 4

Werden `lock()` –Operationen an beiden Assoziationsenden aufgerufen so tritt der Konflikt 4 auf.

Werden hingegen `unlock()` –Operationen an beiden Assoziationsenden aufgerufen so tritt der Konflikt 5 auf:

conflict	StateDepUnlock
context	StateDependency
conflict space:	ManagedObject [mo1]::unlock() ManagedObject [mo2]::unlock()
pre:	self.Antecedent =mo1 and self.Dependent =mo2
refers to:	invariant stateDep

Konflikt 5

Schritt 6: Konfliktlösung festlegen Zur Konfliktvermeidung werden die Vorbedingungen der Methoden `lock()` und `unlock()` spezifiziert. Die Vorbedingung muss sicherstellen, dass die Invariante nach der Terminierung der Methoden sicher eingehalten wird. Zur Definition der Vorbedingung muss die Invariante der Klasse `StateDependency` und die Nachbedingung der jeweiligen Methoden betrachtet werden. Da sich alle drei definierten Konflikte auf dieselbe Invariante beziehen und innerhalb der Konfliktdefinition dieselbe Vorbedingung definieren, können alle drei Konfliktdefinition durch eine Vorbedingung ausgedrückt werden.

```
context ManagedObject::lock() : void
pre:    self.connected.Antecedent.StateDependency
        implies
        self.connected.Antecedent.StateDependency.
        Dependent.forall(mo | mo.status='Locked')
```

Vorbedingung 3

Damit bei einem Bereitsteller die Methode `lock()` aufgerufen werden kann, müssen sich alle Abhängigen im Zustand `Locked` befinden.

```
context ManagedObject::unlock() : void
pre:    self.connected.Dependent.StateDependency
        implies
        self.connected.Dependent.StateDependency.
        Antecedent.status='Unlocked'
```

Vorbedingung 4

Damit bei einem Abhängigen die Methode `unlock()` aufgerufen werden kann, muss sich der Bereitsteller im Zustand `Unlocked` befinden.

Die beiden definierten Vorbedingungen werden nun als (Vor)–Bedingung in den `Condition`–Teil der Policies mit der jeweiligen Aktion aus dem Konfliktraum übernommen:

```
    Policy operational policy2.1
     Event beginOfWork
   Subject Performance Manager
    Target /resource/services/work
    Action unlock();
 Condition self.connected.Dependent.StateDependency
           implies
           self.connected.Dependent.StateDependency.
           Antecedent.status='Unlocked'
```

Die Vorbedingung 4 kann direkt in den `Condition`–Teil der Policies übernommen werden, wenn das Schlüsselwort `self` als:
„Element der Domäne aus dem `Target`–Teil der Policy“ interpretiert wird. Der erste Ausdruck der Implikation kann auch weggelassen werden, da dieser (bezogen auf das Instanzmodell 4.5) immer zu 'wahr' ausgewertet wird.

Da bei der zweiten Policy bereits ein `Condition`–Teil existiert, wird dieser mit der neu Vorbedingung durch ein logisches „und“ verknüpft:

Policy	`operational policy1.1`
Subject	`Security Manager`
Event	`beginOfWork`
Target	`/resource/hardware/server`
Action	`lock(); update(secManager.getPackage());` `reboot(); unlock();`
Condition	`secManager.updateAvailable() and;` `self.connected.Antecedent.StateDependency` **`implies`** `self.connected.Antecedent.StateDependency.` `Dependent.forall(mo | mo.status='Locked')`

4.5.3 Bewertung

Funktionale Abhängigkeiten sind ein wesentliches Merkmal heutiger (verteilter) Systeme. Damit ist die Überwachung und Einhaltung der Abhängigkeiten eine wichtige Aufgabe des Managements. Mit der gezeigten Konfliktbehandlung ist man in der Lage Konflikte zu vermeiden, die sich auf den Zustand von Dienstinstanzen beziehen.

Vergleich mit dem State of the Art Mit der dargestellten Konfliktbehandlung der funktionalen Abhängigkeiten lässt sich eine neue Klasse von Konflikten behandeln. Die domänenorientierte Vorgehensweise zur Konfliktbehandlung nach Lupu, Sloman kann diese Klasse von Konflikten nicht behandeln, da dort keine Beziehungen zwischen Managed Objects beachtet werden bzw. Abhängiger und Bereitsteller sich nicht notwendigerweise in derselben Domäne befinden. Somit kann diese Arbeit einen neuen Beitrag zur Konfliktbehandlung leisten.

4.6 Anwendung der Methodik für Enthaltenseinsbeziehungen

Eine weitere, wesentliche Beziehung zwischen Managementobjekten ist die Enhaltenseinsrelation. Die Enhaltenseinsrelation stellt dar, dass ein Managementobjekt in einem anderen Managementobjekt enthalten ist. Dies kann sowohl eine physische Enhaltenseinsrelation (ein Prozessor befindet sich in einem Computer) als auch eine logische Enhaltenseinsrelation sein (ein Drucker befindet sich in der Office–Domäne).

Das Konzept der logischen Enhaltenseinsrelation spiegelt sich bei der Domänenbildung im Management wider. Gründe für eine Domänenbildung sind beispielsweise organisatorischer, verwaltungstechnischer oder funktionaler Natur. Damit werden unterschiedliche Sichtweisen auf Managementobjekte realisiert. Eine Domäne kann untergliedert werden, also eine Teilmengenbildung vorgenommen werden. Damit wird auch eine Enhaltenseinsrelation etabliert. Domänen selbst werden wiederum als Managementobjekt aufgefasst.

Domänen sind im Bereich des Policy-basierten Managements ein oftmals angewandtes Konzept, mit einer Policy viele Managed Objects gleichzeitig adressieren zu können.

4.6.1 Beispielszenario für Enthaltenseinsbeziehungen

Als Beispielszenario soll die Errichtung und Betreiben von Virtual Private Network (VPN) in einem Unternehmen dienen. Ein VPN wird in [Sail 02] definiert:

VPN *Ein virtuelles privates Netz (VPN) ist ein Netz von logischen Verbindungen innerhalb einer geschlossenen Benutzergruppe mit den Eigenschaften einer privaten Kommunikation. ...*

Das Unternehmen definiert Abteilungen, die aus ihrer Sicht kritisch sind (beispielsweise Finanzabteilung, Personalbabteilung) als geschlossene Benutzergruppen und entscheidet, dass für jede dieser Abteilungen ein VPN eingerichtet werden soll. Die Aufnahme eines neuen Mitglieds in einen kritischen VPN muss überprüft werden. Jeder Mitarbeiter darf sich aus Sicherheitsgründen nur in einem VPN gleichzeitig befinden. Damit eignet sich die Komposition zur a priori Modellierung der Beziehung zwischen VPN und deren Mitgliedern.

Abbildung 4.7 zeigt das Klassendiagramm, welches die Beziehung zwischen VPN und deren Mitgliedern modelliert.

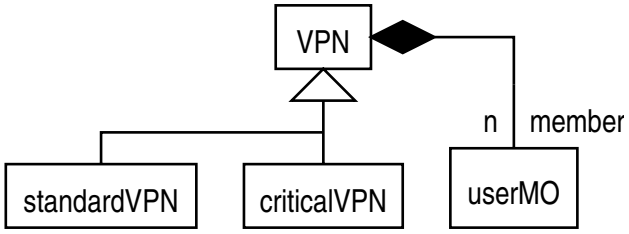

Abbildung 4.7: A priori Modell der Beziehung von Mitglied zu VPN als Klassendiagramm

Folgende Policies sind für die Domäne des VPN zuständig:

Policy	`newVPN`
Event	`newVPN(groupName)`
Subject	`Configuration Manager`
Target	`/resource/vpn/`
Action	`create(groupName);`

Policy	`newUserInStandardVPN`
Event	`newUserInVPN(user)`
Subject	`Configuration Manager`
Target	`/resource/vpn/standard`
Action	`bind(user, part);`

Policy	`newUserInCriticalVPN`
Event	`newUserInVPN(user)`
Subject	`Configuration Manager`
Target	`/resource/vpn/critical`
Action	`user.getVPN().bind(user, part);`
Condition	`criticalGroupMember(user)==true`

Die Policy `newVPN` erzeugt ein neues VPN. Dabei wird implizit ein Kompositionsklasse erzeugt und das VPN als Ganzes gebunden. Die Policy `newUserInStandardVPN` bindet ein neues Mitglied in den Standard–VPN, in denen jeder Mitglied sein kann. Die Policy `newUserInCriticalVPN` bindet ein neues Mitglied in ein VPN, das als kritisch eingestuft wurde. Abbildung 4.8 zeigt die Domänenstruktur der VPNs.

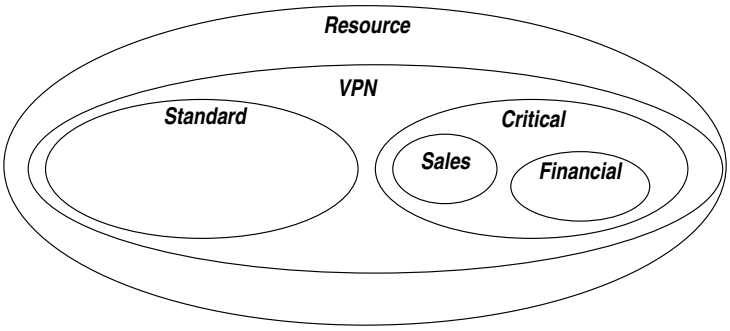

Abbildung 4.8: Domänenstruktur der VPNs

4.6.2 Anwendung der entwickelten Methodik für die Klasse `Aggregation`

Die erste Klasse im Enthaltensein–Ast ist die Klasse `Aggregation` (siehe Abbildung 4.2).

Schritt 2: Invarianten aus dem Modellaspekt extrahieren Für die Klasse `Aggregation` wird folgende Invariante spezifiziert:

```
context      Aggregation
inv aggUpper: self.Whole.multiplicity.upper=1
```

Invariante 5

Die Invariante 5 der Klasse `Aggregation` besagt, dass das Verbindungsende, welches das Ganze repräsentiert, von höchstens einem Objekt belegt werden darf. Die Klasse `Aggregation` enthält nur die notwendigsten Anforderungen, die man für eine Enthaltenseinrelation fordern muss. Diese Klasse kann verwendet werden, um über eine bereits existierende Struktur von Instanzen eine beliebige hierarchische Gruppierung vorzunehmen, wie es beispielsweise bei der Domänenbildung benötigt wird.

Schritt 3: Bezug von Aktionen zu den Invarianten herstellen Für die beiden Methoden `bind()` und `unbind()` ist zu untersuchen, ob Bedingungen für diese Methoden bezüglich der Invariante 5 zu spezifizieren sind. Da die Invariante lediglich die Obergrenze beschränkt, die Einhaltung von Kardinalitätsgrenzen durch `bind()` und `unbind()` aber bereits durch die Oberklasse (Klasse `Association`) gewährleistet wird, muss keine weitere Bedingung für die Klasse `Aggregation` spezifiziert werden.

Das bedeutet, die weiteren Schritte der Methodik müssen nicht mehr bearbeitet werden, da bereits die Konfliktbehandlung der Oberklasse `Association` für die Invariante 5 übernommen wird. Hier zeigt sich, dass die Vererbungshierarchie der Beziehungsklassen aus Abbildung 4.2 sich auch positiv auf die Konfliktbehandlung auswirkt. Jede Subklasse übernimmt („erbt“) die Konfliktbehandlung für die dort betrachteten Aktionen. Das bedeutet, die Klasse `Aggregation` erbt die Konflikte und die Vorbedingungen für die Aktionen bzw. Methoden von der Klasse `Association`. Das bedeutet für die Methoden `bind()` und `unbind()`, die auf einer Instanz der Klasse `Aggregation` aufgerufen wird, dass die gleichen Vorbedingungen (Vorbedingung 1 bzw 2) gelten und somit die Konflikte 1 und 2 nicht auftreten können.

4.6.3 Anwendung der entwickelten Methodik für die Klasse `Composition`

Schritt 2: Invarianten aus dem Modellaspekt extrahieren Die Klasse `Composition` ist eine Verfeinerung der Klasse `Aggregation`. Sie ist für starke Enthaltenseinsbeziehungen gedacht, wie es beispielsweise für den Managent Information Tree (MIT), dem Instanzenbaum des OSI Informationsmodell benötigt wird. Der MIT wird u.a. zur eindeutigen Namensgebung von Instanzen benutzt. Der Instanzname ist ein zusammengesetzter Name, der ausgehend von dem Namen der Wurzelinstanz bis zur eigentlichen Instanz gebildet wird.

Es muss also gewährleistet sein, dass das Intervall der Lebensdauer des `ManagedObject`, welches das Ganze repräsentiert, alle Intervalle der Lebensdauer der Teile beinhaltet. Folgende beiden Invarianten gewährleisten diese Eigenschaft:

```
context            Composition
inv compWholePart: self.Part.participant →size()> 0 implies
                   self.Whole.participant →size()=1
```

Invariante 6

Sind Instanzen einer Kompositionsbeziehung als Teil gebunden, so muss auch eine Instanz als Ganzes gebunden sein.

```
context      Composition
inv compPart: self.Part.participant →forAll(mo :
              ManagedObject | mo.connected.Part=self.Part)
```

Invariante 7

Jedes `ManagedObject`, welches ein `Part` in einer Kompositionsbeziehung ist, darf in keiner anderen Kompositionsbeziehung als `Part` fungieren. Damit wird gewährleistet, dass ein eindeutiger Enthaltenseins*baum* entsteht und kein Enthaltenseins*graph*.

Schritt 3: Bezug von Aktionen zu den Invarianten herstellen Die Klasse `Composition` spezifiziert keine neuen Methoden. Deshalb müssen die geerbten Methoden `bind()` und `unbind()` aus der Oberklasse `Association` bezüglich der beiden neuen Invarianten untersucht werden.

Die Invariante 6 verlangt, dass die Lebensdauer des Ganzen alle Teile beinhaltet. Somit ist der Aufrufzeitpunkt der `bind()` sowie der `unbind()` Methoden von Interesse für diese Invariante.

Die Invariante 7 verlangt, dass ein Managementobjekt nur als `Part` in einer einzigen Kompositionsbeziehung gebunden sein darf. Es muss keine neue Nachbedingung für die Methode `bind()` spezifiziert werden, da die Nachbedingung beschreibt, was eine Methode bewirkt. Die Wirkung der Methode bleibt unverändert. Es wird aber, wie in Schritt 6 der Methodik zu sehen, eine neue Vorbedingung für die Methode `bind()` spezifiziert.

Schritt 4: Kritische Aktionen identifizieren Kritische Aktionen sind diejenigen Aktionen, die dazu beitragen, dass die Invarianten 6 oder 7 nicht eingehalten werden. Die zu untersuchenden Aktionen für jede Invariante sind `bind()` und `unbind()` (geerbt von der Oberklasse `Association`).

Die Invariante 7 wird von der Methode `bind()` beeinflusst, da die Invariante eine Einschränkung der Einbindung von Managementobjekten an Assoziationsenden darstellt. Die Methode `bind()` ist somit bezüglich Invariante 7 kritisch. Bezogen auf die Beispielpolicies aus Abschnitt 4.6.1 bedeutet dies, dass die Policies `newUserInStandardVPN` und `newUserInCriticalVPN` sich im Konfliktraum bezüglich Invariante 7 befindet:

$$Konfliktraum_{inv\ compPart} = \{\texttt{newUserInStandardVPN}, \texttt{newUserInCriticalVPN}\}$$

Schritt 5: Konfliktdefinition Da das Ganze die Lebensdauer der Teile umschließen muss, existieren zwei Möglichkeiten, die Invariante 6 zu brechen. Einmal werden Teile vor dem Ganzen eingebunden oder das Ganze wird entbunden, obwohl noch Teile in der Beziehung existieren. Demzufolge werden zwei Konfliktbedingungen spezifiziert:

conflict	compWholePart
context	Composition
conflict space:	bind(mo1 : ManagedObject, ae1 : AssociationEnd) unbind(mo2 : ManagedObject, ae2 : AssociationEnd)
pre:	ae1.oclIsTypeOf(Part) and ae2.oclIsTypeOf(Whole)
refers to:	invariant compWholePart

Konflikt 6

Wird in einer Instanz der Klasse das Managementobjekt, welches das Ganze repräsentiert, entbunden (`unbind()`) und sind aber noch Teile gebunden (`bind()`), so stehen diese Aktionen in Konflikt zueinander.

conflict	compPartBeforeWhole
context	Composition
conflict space:	bind(mo1 : ManagedObject, ae1 : AssociationEnd) bind(mo2 : ManagedObject, ae2 : AssociationEnd)
pre:	ae1.oclIsTypeOf(Part) and ae2.oclIsTypeOf(Whole)
refers to:	invariant compWholePart

Konflikt 7

Wird eine Instanz, die einen Teil repräsentiert vor der Instanz gebunden, die das Ganze repräsentiert, so wird die Invariante `compWholePart` verletzt.

Die nächste Konfliktdefinition bezieht sich auf die Invariante 7 mit dem zugehörigen Konfliktraum:

conflict	compOnePart
context	Composition
type:	deterministic
conflict space:	Composition[comp1].bind(mo1 : ManagedObject, ae1 : AssociationEnd) Composition[comp2].bind(mo2 : ManagedObject, ae2 : AssociationEnd)
pre:	comp1<>comp2 and mo1=mo2 and ae1.oclIsTypeOf(Part) and ae2.oclIsTypeOf(Part)
refers to:	invariant compOnePart

Konflikt 8

Wird in zwei verschieden Instanzen (`comp1 <> comp2`) der Klasse `Composition` dasselbe Managementobjekt (`mo1=mo2`) jeweils als `Part` gebunden (`ae2.oclIsTypeOf(Part)`), so verletzen die beiden `bind()`–Methoden die Invariante 7, welche aussagt, dass ein Managementobjekt als `Part` in höchstens einer Kompositionsbeziehung fungieren darf.

Konfliktanalyse anhand der Beispielpolicies Tritt das Event `newUserInVPN(user)` auf, so werden die beiden Policies `newUserInStandardVPN` und `newUserInCriticalVPN` getriggert. Die erste Policy bindet den `user` in den Standard–VPN, die zweite Policy bindet den `user` in den zugehörigen kritischen VPN ein. Damit ist `user` in zwei VPNs als Teil gebunden und somit tritt der Konflikt `compOnePart` auf (Konflikt 8).

Schritt 6: Konfliktlösung festlegen Ausgehend von den neuen Konfliktdefinitionen aus Schritt 5 müssen dementsprechend Vorbedingungen für die Methoden `bind()` und `unbind()` spezifiziert werden.

Die Methode `bind()` ist Teil jeder Konfliktdefinition aus Schritt 5. Für jede Konfliktdefinition wird eine Vorbedingung spezifiziert. Für die Methode `bind()` existiert bereits eine Vorbedingung aus der Oberklasse `Association`. Diese muss natürlich erhalten bleiben, die neuen Vorbedingung werden angehängt, indem sie mit dem logischen UND verknüpft werden: Folgende Vorbedingung wird für die Methode `bind()` bezüglich der Konfliktdefinition 8 spezifiziert:

```
context Composition::bind(mo : ManagedObject, ae :
        AssociationEnd) : void
   pre: -- precondition conflict upperLimit
        self.ae.participant →size() <
        self.ae.multiplicity.upper
        -- precondition conflict compPartBeforeWhole
        and
        (ae.oclIsTypeOf(Part) and
        ae.Association.oclIsTypeOf(Composition) implies
        mo.connected.Part.Composition.Whole.
        participant→size()=1)
        -- precondition conflict compOnePart
        and
        (ae.oclIsTypeOf(Part) and
        ae.Association.oclIsTypeOf(Composition) implies
        mo.connected.Part.Composition→size()=0)
```

Vorbedingung 5

Vorbedingung 5 drückt aus, dass eine Instanz nur als `Part` gebunden werden kann, wenn sie nicht bereits als `Part` gebunden ist, das Ganze bereits existiert und die Obergrenze noch nicht überschritten ist.

Die Methode `unbind()` ist nur in der Konfliktdefinition 6 im Konfliktraum vorhanden. Analog zur `bind()` Methode werden alle Vorbedingung, die für die Methode `unbind()` spezifiziert wurden durch eine logische Und–Verknüpfung

zusammengeführt: Die Methode `unbind()` darf nur ausgeführt werden, wenn keine Instanzen mehr als `Part` gebunden ist:

```
context Composition::unbind(mo : ManagedObject, ae :
        AssociationEnd) : void
   pre: -- precondition conflict lowerLimit
        (self.ae.participant →size() >
        self.ae.multiplicity.lower)
        -- precondition conflict compWholePart
        and
        (self.Part.participant→size()=0)
```

Vorbedingung 6

4.6.4 Bewertung

Die Enthaltenseinsrelation ist im Bereich des Managements eine wichtige Beziehung zwischen Managementobjekten , da jedes Managementobjekt bei seiner Erzeugung in einen Enthaltenseinsbaum eingegliedert wird. Auch die im Bereich des Policy-basierten Managements eingesetzten Domänen repräsentieren im Kern eine Enthaltenseinsrelation.

Die Anwendung der Methodik hat gezeigt, dass es möglich ist systematisch aus Invarianten Konfliktdefinitionen abzuleiten. Dabei erleichtert die hierarchische Strukturierung der Beziehungsklassen die Konzentration auf das Wesentliche einer Klasse. Durch die Vererbung von Konfliktbedingungen und Nachbedingungen ist ein Mechanismus etabliert worden, der eine einfache Konfliktdefinition ermöglicht. Vorbedingungen für Aktionen lassen sich einfach aus den Konfliktdefinitionen ableiten. Diese Vorbedingungen sorgen für eine Konfliktvermeidung, das heißt, ein Konflikt kann nicht auftreten. Diese Vorbedingungen können automatisch, wie das folgenden Kapitel zeigen wird, für Policies erzeugt werden. Damit wird die Aufgabe für den Policy–Ersteller entscheidend vereinfacht.

4.7 Anwendung der Methodik für endliche Automaten

Endliche Automaten repräsentieren einen Vertreter von dynamischen Modellarten im Gegensatz zu den in diesem Kapitel bis jetzt behandelten statischen Modellen von Beziehungen.

Zuerst muss geklärt werden, welche Ausdrucksmächtigkeit endliche Automaten im Bereich des IT–Managements besitzen müssen. Das Standardisierungsgre-

mium ISO hat einen generischen endlichen Automaten [ISO 10164-2] definiert, der als Ausgangsbasis zur Definition von spezifischen Automaten dienen soll.

[ISO 10164-2] definiert für die Modellierung einer generischen Zustandsfunktion für alle Managementobjekte drei generische endliche Automaten: administrativ, operational und Nutzung.

- Administrativ: besitzt die Zustände Unlocked, Locked und Shutting down sowie die Transitionen Unlock, Lock, Shut Down, User Quit, Last User Quit
- Operational: besitzt die Zustände Enabled und Disabled sowie die zugehörigen Transitionen Enable und Disable.
- Nutzung: besitzt die Zustände Idle, Active und Busy sowie die Transitionen User Quit, New User, Last User Quit, Capacity Increase und Capacity Decrease.

Das Management kann nur den administrativen endlichen Automaten beeinflussen. Die Zustände des operationalen und des nutzungsbezogenen endlichen Automaten sind als nur lesbar (read–only) definiert. Die Beziehung der drei Automaten zueinander wird ebenfalls definiert (siehe Abbildung 4.9).

Der endliche Automat besitzt als wesentliches neues Merkmal gegenüber den endlichen Automaten, wie sie aus der theoretischen Informatik bekannt sind, die Möglichkeit der Schachtelung von Zuständen, den sogenannten zusammengesetzten Zuständen.

Um die Methodik anwenden zu können, ist eine formalisierte Darstellung eines endlichen Automaten notwendig. Korrespondierend zum Metamodell der Beziehungen (siehe Abschnitt 4.3.1) ist eine Metamodell eines endlichen Automaten notwendig. UML [uml 03] spezifiziert ein Metamodell von endlichen Automaten (*State Machines*), das nun dargestellt wird.

4.7.1 UML State Machine Metamodell

UML führt mit dem State Machine Metamodell das Konzept eines ausdrucksmächtigen endlichen Automaten ein. Er unterstützt zusammengesetzte und nebenläufige Zustände. Außerdem sieht das Modell bereits vor, dass Transitionen Operationen repräsentieren können.

Abbildung 4.10 zeigt das Metamodell der State Machines. Ein endlicher Automat (`StateMachine`) besteht aus einem Oberzustand (`State`) und Transitionen (`Transition`). Ein Zustand kann einfach (`SimpleState`), ein Endzustand (`FinalState`) oder zusammengesetzt (`CompoundState`) sein. Eine

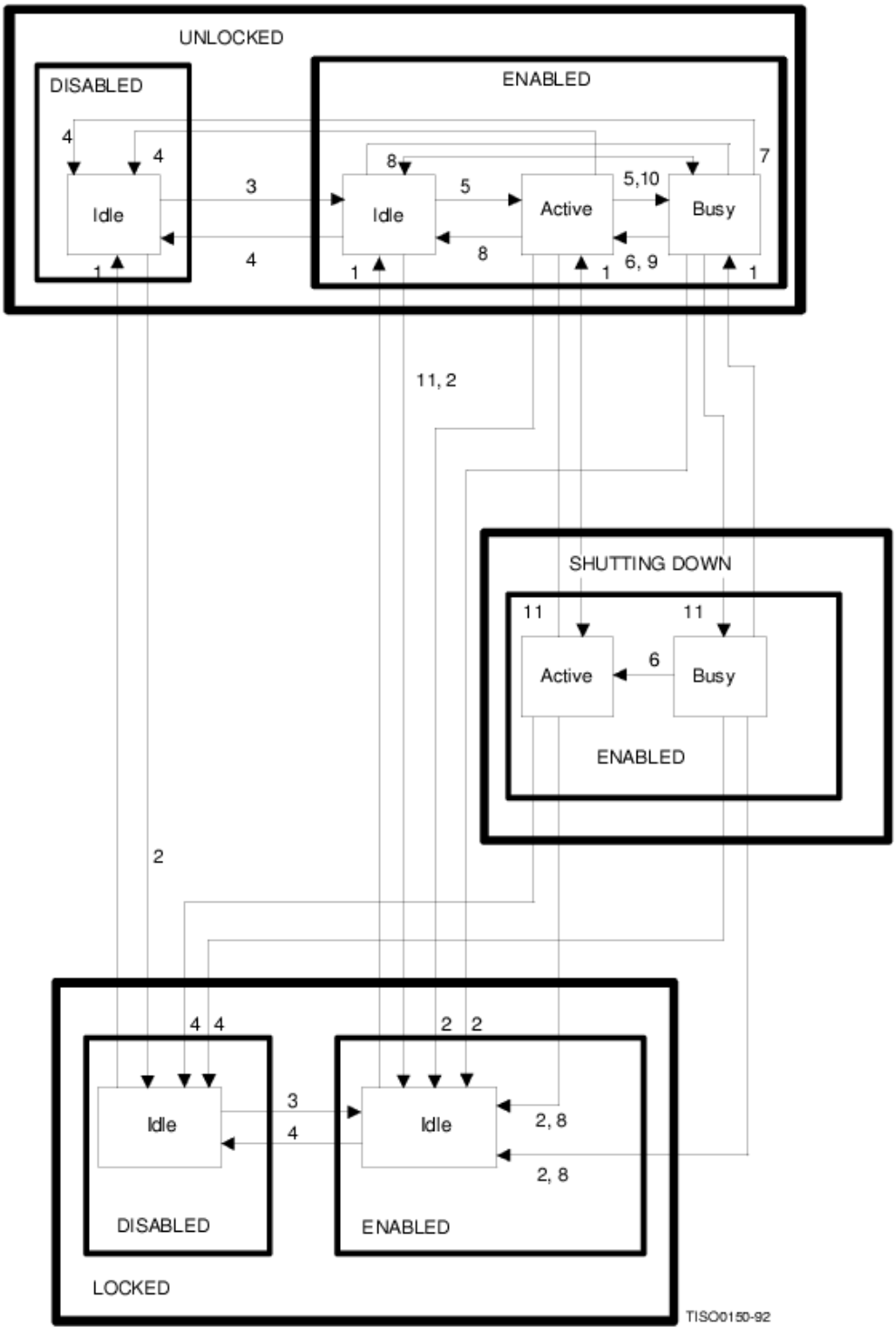

Abbildung 4.9: Kombiniertes Zustandsdiagramm aus [ISO 10164-2]

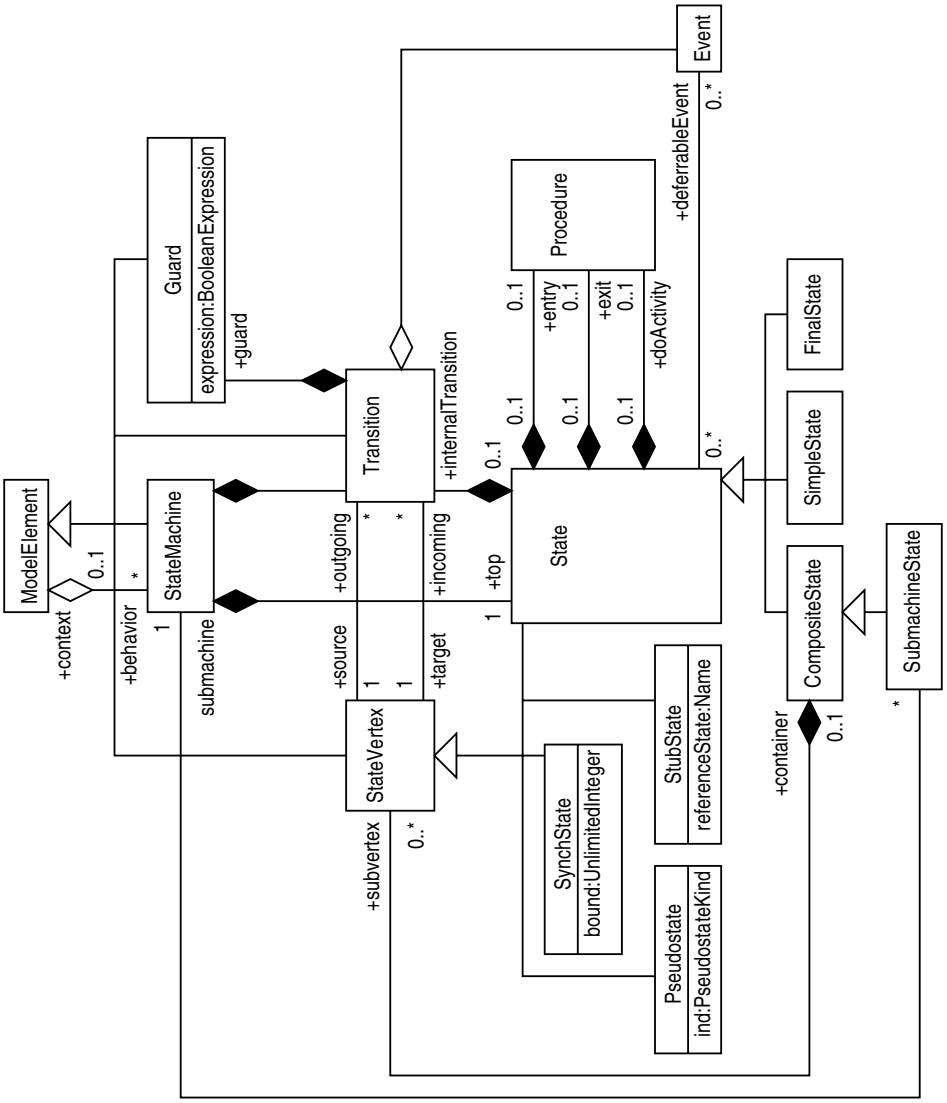

Abbildung 4.10: UML Metamodell der State Machine [uml 03]

Transition besteht aus einem Ereignis (`Event`), einem Wächter (`Guard`) und einer Methode (`Procedure`). Transitionen werden eindeutig genau einem Zustand als Quelle und einem Zustand als Ziel (`StateVertex`) zugeordnet. Es existieren gesondert ausgezeichnete Zustände (`SynchState`, `StubState` und `PseudoState`) um Nebenläufigkeit und Anfangszustände zu modellieren.

Der Beispielautomat in Abschnitt 4.7.3 ist eine Instanz des UML Metamodells der State Machine.

Die Abarbeitung eines endlichen Automaten, welche im Metamodell nicht dargestellt ist, wird von einer hypothetischen Maschine übernommen, die sich innerhalb der `StateMachine` befindet. Die hypothetische Maschine verarbeitet Ereignisse in einem sogenannten *run-to-completion-step*, der folgendermaßen aussieht:

1. Ein Ereignis (*currentEvent*) wird aus der Event–Queue genommen.
2. Es wird die Menge der freigegebenen (*enabled*) Transitionen bestimmt:
   (a) die Transitionen müssen auf das Ereignis triggern.
   (b) die Transitionen müssen ihre Quelle in einem zu diesem Zeitpunkt aktiven Zustand haben.
   (c) der Wächter der Transition muss zu „wahr“ ausgewertet werden.
3. Die Untermenge der Transitionen wird zur Ausführung getestet.
4. Aus der Menge der freigegebenen Transitionen wird durch den Transitionsauswahlalgorithmus eine Untermenge tatsächlich ausgeführt. Diese Untermenge wird so bestimmt, dass keine Konflikte innerhalb der Menge auftreten können.
5. Die Menge der nach der Ausführung aktiven Zustände wird bestimmt.
6. Es wird mit Schritt 1 fortgefahren.

Beim *run-to-completion-step* wird immer nur ein Ereignis gleichzeitig ausgeführt.

4.7.2 Zusammenhang von endlichen Automaten, Managementobjekten und policy–basiertem Management

Im letzten Abschnitt wurde das Metamodell eines endlichen Automaten eingeführt. In diesem Abschnitt wird der Zusammenhang zwischen den endlichen Automaten, den Managementobjekten, dessen Bestandteil sie sind, und

dem policy–basierten Management, welches Managementobjekte manipuliert, bestimmt. Zuerst werden die statischen Zusammenhänge von Policy, Managementobjekt und endlichen Automaten aufgezeigt und anschließend der Zusammenhang zwischen der Ausführungsmaschine des policy–basierten Managements (PDP und PEP) und der hypothetischen Maschine eines endlichen Automaten bestimmt. Es muss untersucht werden, wie Policies auf endliche Automaten wirken, damit die anschließende Anwendung der Methodik korrekt verwendet werden kann.

Die Abbildbarkeit der statischen und dynamischen Strukturen von endlichen Automaten zu policy–basierten Management ist von Interesse, da eine hypothetische Maschine im Allgemeinen nicht von Managementobjekten implementiert wird.

Statischer Zusammenhang Abbildung 4.11 zeigt die Zusammenhänge zwischen den Metamodellen der Entitäten. Auf der linken Seite der Abbildung ist eine Policy mit ihren Bestandteilen, in der Mitte ein Managementobjekt mit den Bestandteilen und auf der rechten Seite ist ein Ausschnitt aus dem UML Metamodell für State Machines abgebildet.

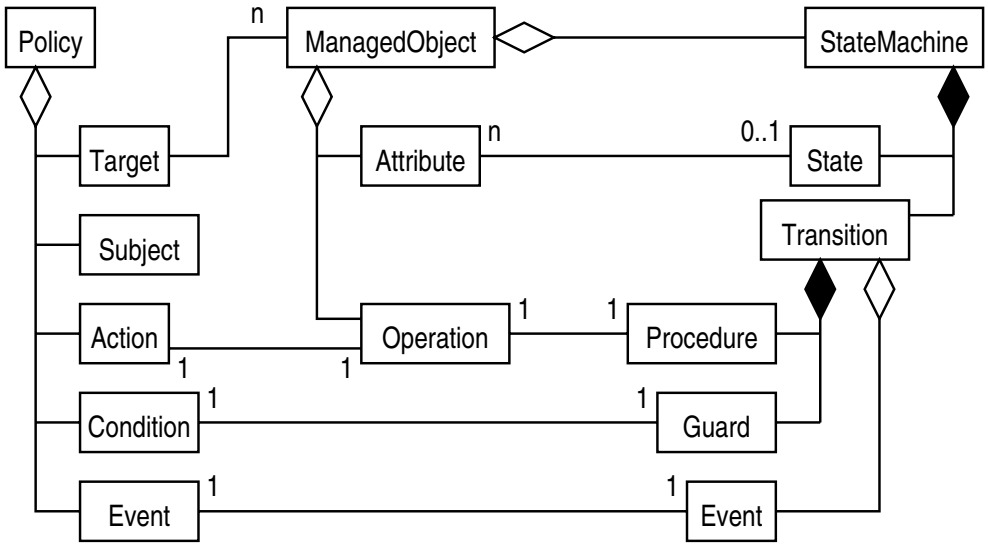

Abbildung 4.11: Zusammenhang von Policy, Managementobjekt und endlichem Automat

Unter der Vorrausetzung, dass der Target–Teil einer Policy ein Managementobjekt adressiert und der Action–Teil einer Policy den Procedure–Teil eines endlichen Automaten repräsentiert, kann der Event der Policy auf den Event des endlichen Automaten abgebildet werden. Genauso wird der Condition–Teil einer Policy auf den Guard des endlichen Automaten abgebildet.

Da man die Bestandteile von Policy und endlichen Automaten weitgehend aufeinander abbilden kann, folgt daraus, dass man die statischen Bestandteile von endlichen Automaten (bis auf die Zustände) auf die statischen Bestandteile von Policies abbilden kann.

Dynamischer Zusammenhang Abbildung 4.12 verdeutlicht den Zusammenhang zwischen den Policy–Architekturkomponenten zur Ausführung von Policies und der hypothetischen Maschine zur Ausführung eines endlichen Automaten.

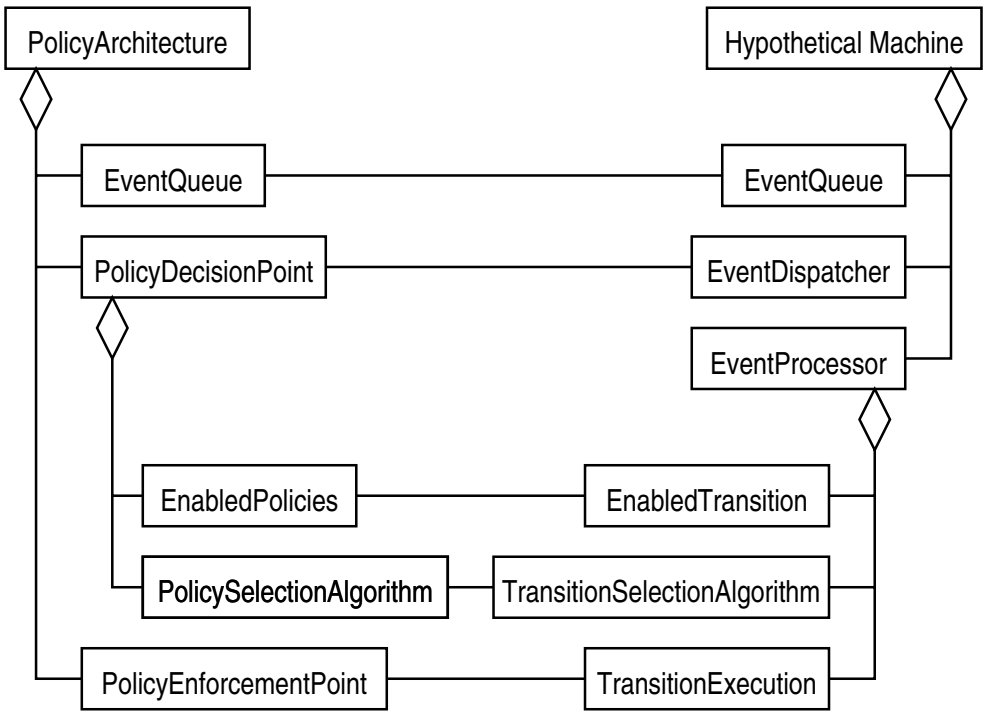

Abbildung 4.12: Zusammenhang der Ausführungsmaschinen des policy–basierten Managements und und endlichem Automat

Dabei zeigt sich, dass alle Komponenten der hypothetischen Maschine auf die Komponenten einer Policy–Architektur abgebildet werden können. Beide Ausführungsmaschinen sind ereignisgesteuert und besitzen somit eine Event–Queue. Die Aufgabe des Event–Dispatchers kann auf den Policy Decision Point (PDP) abgebildet werden. Die Auswahlalgorithmen sowohl für die freigegebenen Policies als auch für die Transitionen arbeiten analog (beide müssen Untermengen von konfliktfreien Policies finden). Ebenso existiert in beiden Ausführungsmaschinen ein Algorithmus, um die tatsächlich auszuführenden Policy bzw. Tran-

sitionen auszuwählen. Der Policy Enforcement Point (PEP) übernimmt die Aufgabe, die Policies auszuführen, für den endlichen Automaten ist das Gegenstück die Transition Execution.

Der Policy Enforcement Point ist der einzige Teil der hypothetischen Maschine, welches im Allgemeinen von Managementobjekten implementiert wird.

Folgerung Die Analyse der Zusammenhänge von Policies, Managementobjekt und endlichen Automaten hat gezeigt, dass sich die wesentlichen Entitäten des endlichen Automaten und der zugehörigen hypothetischen Maschine auf Policies bzw. Policy–Architektur abbilden lassen. Lediglich die aktiven Zustände müssen zusätzlich dem PDP bekannt gemacht werden. Damit kann das Policy–System die Abarbeitung und Konfliktbehandlung für endlichen Automaten übernehmen.

Dies ist besonders wichtig vor dem Hintergrund, dass Managementobjekte im Allgemeinen keine hypothetische Maschine implementieren, sondern lediglich Transitionen, welche durch Managementoperationen repräsentiert sind, ausführen und somit auf der Ebene der Managementobjekten von keiner Konfliktbehandlung bei endlichen Automaten im IT–Management ausgegangen werden kann.

4.7.3 Beispielszenario mit endlichen Automaten

Der deterministische, endliche Automat in Abbildung 4.13 ist von dem generischen Automaten aus [ISO 10164-2] abgeleitet und stellt einen möglichen Zustandsautomaten eines Druckers dar. Er besteht aus dem zusammengesetzten Zustand `Unlocked` , mit den Unterzuständen `Idle` und `Queueing` sowie den Zuständen `Shutdown` und `Locked` .

Im Zustand `Unlocked` ist der Drucker zur Nutzung freigegeben und kann mittels der Transition `queueJob()` entweder vom Zustand `Idle` oder dem Zustand `Queueing` aus Aufträge annehmen. Zur Wartung des Druckers, beispielsweise um Toner nachzufüllen, kann mittels der Transition `lock()` in den Zustand `Locked` gebracht werden, indem keine Aufträge angenommen oder verarbeitet werden können. Im Zustand `Shutting Down` werden die bereits eingetretenen Aufträge noch verarbeitet. Sind alle Aufträge abgearbeitet, so wird in den Zustand `Locked` asynchron übergegangen, indem das (interne) Ereignis `idle` asynchron ausgelöst wird. Der endliche Automat soll sich im Unterzustand `Queueing` und damit auch im übergeordneten Zustand `Unlocked` (angezeigt durch die dicke Umrandung der Zustände) befinden.

Die Klasse `Printer` stellt das Managementobjekt dar, welches als Bestandteil den endlichen Automaten besitzt. Das Verhalten der Klasse `Printer` wird vom endlichen Automaten dargestellt. Die Klasse `Printer` repräsentiert somit das

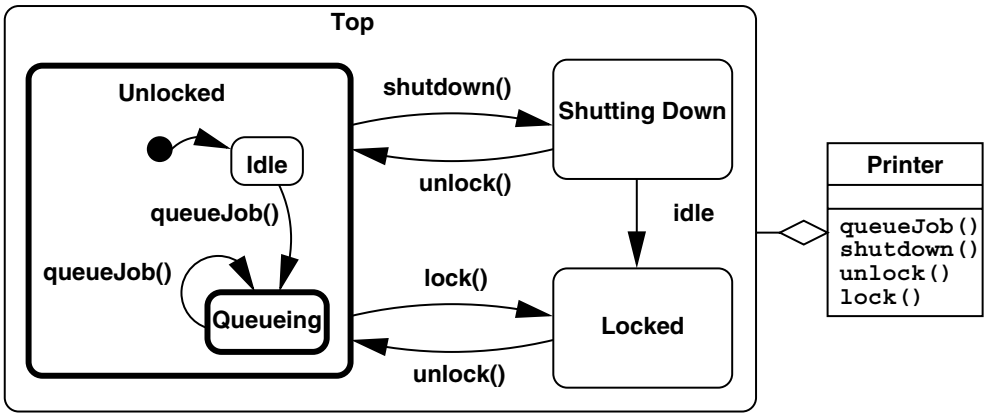

Abbildung 4.13: Endlicher Automat eines Druckers und zugehöriges Managed Object

`ModelElement` im UML Metamodell (siehe Abbildung 4.10). Die Managementoperationen der Klasse bilden die Transitionen des endlichen Automaten.

Als nächstes muss die Frage beantwortet werden, wie die hypothetische Ausführungsmaschine im Management umgesetzt wird. Das Managed Object führt die Transitionen sequentiell durch, steuert diese aber nicht. Diesen Teil übernimmt im Szenario dieser Arbeit das Policy-basierte Managementsystem (siehe Abschnitt 4.7.2).

Um das Beispiel zu vervollständigen, wird der endliche Automat durch drei Policies gesteuert:

Policy `policy1`
Event `afterWork`
Target `/company/office/device/printer`
Action `queueJob(Job > 500)`

Policy `policy2`
Event `afterWork`
Target `/company/office/device/printer`
Action `shutdown()`

Policy `policy3`
Event `beginWork`
Target `/company/office/device/printer`
Action `unlock()`

Diese Beispielpolicies greifen das Beispiel aus Abschnitt 1.1 auf und erweitern es: Die ersten beiden Policies feuern auf das Ereignis `afterWork`

und haben als Target–Domain (Target) alle Drucker der Domöne `/company/office/device/printer`. Policy `policy1` drückt aus, dass alle zurückgehaltenen Druckaufträge, welche größer als 500 Seiten sind, nach Betriebsschluss gedruckt werden sollen. Policy `policy2` leitet nach Betriebsende einen `shutdown()` ein, damit noch alle sich in der Queue befindlichen Aufträge abgearbeitet werden können. Die letzte Policy `policy3` veranlasst, dass alle Drucker der Domäne `/company/office/device/printer` bei Betriebsbeginn zur Nutzung freigegeben werden.

Im Beispiel ist zu sehen, dass erst durch die Spezifikation der Policies die Ereignisse, wann die Transitionen eines endlichen Automaten feuern, definiert werden. Somit kann unter Berücksichtigung der Ereignisse ein erweiterter Beispielautomat spezifiziert werden. Dazu werden zu den Transitionen neben den Operationen die dazugehörigen Ereignisse mit eingetragen in der Form:
`event/operation`
Abbildung 4.14 stellt den erweiterten Beispielautomaten dar.

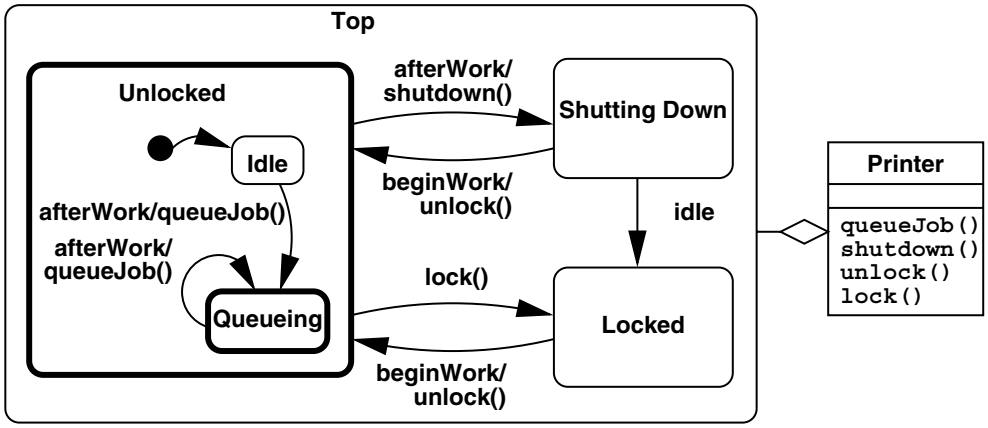

Abbildung 4.14: Um die aus den Policies abgeleiteten Ereignisse erweiterter endlicher Automat eines Druckers

4.7.4 Anwendung der entwickelten Methodik für die Klasse `StateMachine`

Im UML Standard [uml 03] ist bereits eine Konfliktdefinition gegeben. Die Konfliktdefinition erfolgt nur in Prosa und ist nicht formalisiert. Damit muss dieser Punkt im weiteren Verlauf bei der Anwendung der Methodik noch erarbeitet werden. Die Konfliktdefinition lautet [uml 03]:

Two transitions are said to conflict if they both exit the same state, or, more precisely, that the intersection of the set of states they exit is non-empty.

Die Konfliktdefinition gilt für Transitionen, die freigegeben sind. Die Überprüfung, ob eine Schnittmenge vorliegt, ist nur im Fall, dass ein endlicher Automat nebenläufige Zustände beinhaltet, notwendig. Sind keine nebenläufigen Zustände vorhanden, so ist die Schnittmenge der aktiven Zustände nicht leer, da sich dann ein endlicher Automat immer nur in genau einem (zusammengesetzten) Zustand gleichzeitig befinden kann.

Schritt 2: Invarianten aus dem Modellaspekt extrahieren Die formalisierte Darstellung eines endlichen Automaten ist durch das UML Metamodell (siehe Abschnitt 4.7.1) gegeben. Das UML Metamodell formalisiert nur den endlichen Automaten (Menge der Zustände und Menge der Transitionen), aber nicht die hypothetische Ausführungsmaschine. Ein Konflikt tritt aber erst bei der Abarbeitung von Ereignissen durch die hypothetische Maschine auf.

Die Konfliktdefinition des UML Standards bezieht sich auf die von Ereignissen freigegebenen Transitionen. Da das Metamodell die Markierung einer Transition als freigegeben (*enabled*) nicht unterstützt (es fehlt das entsprechende Attribut in der Klasse `Transition`), wird das Metamodell um diese Möglichkeit erweitert (siehe Abbildung 4.15).

Das erweiterte Metamodell enthält zusätzlich die Klasse `EnabledTransition`, die das Attribut `enabled` enthält. Jede Transition, bei der das Attribut `enabled` zu „wahr“ ausgewertet wird, ist bezüglich eines Events freigegeben.

Mit Hilfe dieser Erweiterung lässt sich die Invariante für die Klasse `StateMachine` definieren:

```
context           StateMachine
inv stateMachine: let enabledTrans : Set =
                  self.transitions->forAll(t|t.enabled=true)

                  self.enabledTrans->size() > 1
                  implies
                  self.enabledTrans->forAll(t1, t2| t1<>t2
                  implies
                  not( t1.source->includes(t2.source) or
                  t2.source->includes(t1.source)))
```

Invariante 8

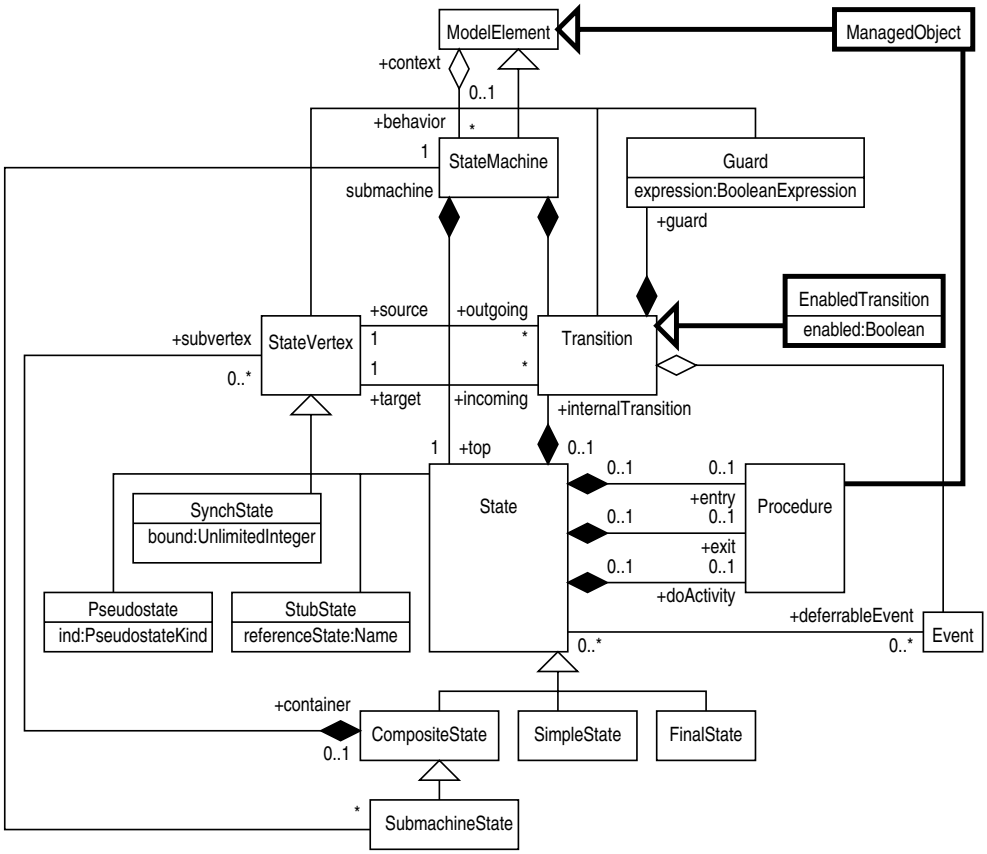

Abbildung 4.15: Erweitertes Metamodell zur Markierung von freigegebenen Transitionen

Die erste Zeile der Invariante definiert eine Variable `enabledTrans`, welche die Menge der freigegebenen Transitionen repräsentiert (`Set` ist ein in OCL vordefinierter Typ, der eine Menge im mathematischen Sinn repräsentiert). Die Invariante lautet: Falls mehr als eine Transition freigegeben ist, so muss die Schnittmenge aller Ursprungszustände leer sein.

Besitzt ein endlicher Automat keine nebenläufigen Zustände, dann genügt die Überprüfung des ersten Terms der Invariante, da der zweite Term immer „wahr“ liefert.

Bezogen auf den Beispielautomaten bedeutet dies, falls sich der Automat im Zustand `Queueing` befindet, dass beim Ereignis `afterWork` sich die Transitionen `shuttingDown()` und `queueJob()` in der Menge der freigegebenen Transitionen befinden und diese Menge damit mehr als ein Element enthält. Die Schnittmenge der beiden Quellzustände ist der Zustand `Queueing`.

Schritt 3: Bezug von Aktionen zu den Invarianten herstellen In diesem Schritt der Methodik werden die Nachbedingungen der Methoden spezifiziert, die mit der Klasse der Invariante zusammenhängen. Im Metamodell der State Machines werden keine expliziten Methoden definiert. Die Methoden sind in den Transitionen `Transition`, genauer in der Klasse `Procedure`, abstrakt modelliert. Somit können auch keine expliziten Nachbedingungen in dieser Modellebene, der Metamodellebene, spezifiziert werden.

Eine Instanz des Metamodells stellt einen konkreten endlichen Automaten dar. Alle diejenigen Transitionen, an denen Aktionen gekoppelt sind, können in ihrer Wirkung beschrieben werden. Die Aktionen sind Methoden einer Klasse (im Beispiel ist die Transition `shutdown()` des endlichen Automaten eine Methode der Klasse `printer`). Dieser Bezug ist auch im erweiterten Metamodell (siehe Abbildung 4.15) durch die Assoziation von `Procedure` zur Klasse `ManagedObject` dargestellt. Also befinden sich alle Operationen einer Klasse, die einen endlichen Automaten implementiert, in Bezug zur Invariante `stateMachine`.

Im Beispielszenario haben alle Aktionen der Policies einen Bezug zum endlichen Automaten der Klasse `printer`. Da die Aktionen der drei Policies Methodenaufrufe von Instanzen der Klasse `printer` sind und diese wiederum alle auch in den Transitionen des zugehörigen endlichen Automaten vorkommen, sind alle Aktionen in Bezug zur Invariante.

Schritt 4: Kritische Aktionen identifizieren Kritisch sind diejenigen Aktionen, die sich in der Menge der freigegebenen Transitionen befinden. Diese Menge

wird für jeden möglichen Event im endlichen Automaten bestimmt und ändert sich bei der Ausführung der hypothetischen Maschine somit dynamisch.

Im Allgemeinen ist es nicht möglich alle Mengen von freigebenende Transitionen vor dem Ablauf der hypothetischen Maschine zu bestimmen, da die Auswertung der Wächter erst zur Laufzeit vorgenommen werden kann.

Schritt 5: Konfliktdefinition Die Vorarbeiten von Schritt 2 bis Schritt 4 ermöglichen nun die Konfliktdefinition für die Invariante `stateMachine`:

conflict	`StateMachine`
context	`StateMachine`
conflict space:	`conflictOperations : Set =` `self.enabledTrans.effects`
refers to:	`invariant stateMachine`

Konflikt 9

Ein Konflikt tritt ein, wenn die Invariante `stateMachine` gebrochen wird, dass heißt, falls sich mehr als eine Operation in der Menge der freigegebenen Transitionen `self.enabledTrans.effects` befindet und die Schnittmenge aller Ursprungszustände nicht leer ist.

Am Beispielautomat muss der Ablauf der hypothetischen Maschine betrachtet werden wie in Abschnitt 4.7.1 dargestellt. Der endliche Automat soll sich in dem Zustand `Queueing` und, da dieser Zustand ein Unterzustand ist, auch im Zustand `Unlocked` (siehe Abbildung 4.13) befinden. Da der Beispielautomat keine nebenläufigen Zustände besitzt, muss die Schnittbildung nicht überprüft werden, da die Schnittmenge nicht leer sein kann.

Wird in diesem zusammengesetzten Zustand das Ereignis `afterWork` getriggert, so befinden sich die beiden Transitionen `queueJob()` und `shutdown()` in der Menge der freigegebenen Transitionen. Somit ist die Konfliktdefinition erfüllt und ein Konflikt zwischen den beiden Policies `policy1` und `policy2` eingetreten.

Schritt 6: Konfliktlösung festlegen Die Konfliktlösung kann aufgeteilt werden abhängig davon, ob der endliche Automat nebenläufige Zustände enthält oder nicht.

Endlicher Automat ohne nebenläufige Zustände Bei einem endlichen Automaten ohne nebenläufige Zustände genügt bereits die Anzahl der freigegebenen Transitionen um entscheiden zu können, ob ein Konflikt eintritt oder nicht. Somit

kann eine Vorbedingung definiert werden, die vor Ausführung einer Operation die Anzahl der freigegebenen Transitionen überprüft:

```
context ManagedObject::operation() : void
pre:    self.enabledTrans.effect
        ->includes(self.operation) implies
        self.stateMachine.enabledTrans->size() <=1
```

Vorbedingung 7

Dabei fungiert der Ausdruck `ManagedObject::operation()` für eine beliebige Operation, die an der Managed Object Boundary zur Verfügung gestellt wird. Wie Abbildung 4.15 zeigt, kann ein Managementobjekt (Klasse `ManagedObject`) über die Aggregationsbeziehung zu seinem endlichen Automaten navigieren und die Anzahl der freigegebenen Transitionen ermitteln.

Für den Beispielautomaten und die zugehörigen Policies bedeutet dies, dass jede Policy diese Vorbedingung in den Condition–Teil übernimmt, da jeder Action–Teil der Policies eine Transition auf einem Managementobjekt repräsentiert.

Policy	`policy1`
Event	`afterWork`
Target	`/company/office/device/printer`
Action	`queueJob(Job > 500)`
Condition	`stateMachine.enabledTrans->size() <=1`

Policy	`policy2`
Event	`afterWork`
Target	`/company/office/device/printer`
Action	`shutdown()`
Condition	`stateMachine.enabledTrans->size() <=1`

Policy	`policy3`
Event	`beginWork`
Target	`/company/office/device/printer`
Action	`unlock()`
Condition	`stateMachine.enabledTrans->size() <=1`

Endlicher Automat mit nebenläufigen Zuständen Für endliche Automaten mit nebenläufigen Zuständen muss der gesamte Term der Invariante bei der Ausführung getestet werden (am Beispiel von Policy `policy1`):

Policy	`policy1`
Event	`afterWork`
Target	`/company/office/device/printer`
Action	`queueJob(Job > 500)`
Condition	`stateMachine.enabledTrans->size() >=1 implies statemachine.enabledTrans ->forAll(t1, t2| t1<>t2 implies not( t1.source->includes(t2.source) or t2.source->includes(t1.source)))`

4.7.5 Bewertung

Das Modell eines endlichen Automaten ist ein Vertreter von dynamischen Modellen, an dem die Methodik angewandt wurde. Eine Vielzahl von Managementobjekten im IT–Management ist zustandsbehaftet (siehe beispielsweise Common Information Model [CIM 2.2]). Es konnte mit Hilfe der Methodik eine Konfliktbehandlung entworfen werden, die für einfache wie komplexe Automaten Konflikte zwischen Policies verhindern kann. Durch die Definition einer Vorbedingung, die unabhängig von einer konkreten Policy–Sprache ist, kann die Konfliktbehandlung in beliebigen Policy–Sprachen eingesetzt werden.

Vergleich mit dem State of the Art Eine Analyse von Policy–Konflikten in Bezug auf die Ausführung von endliche Automaten ist nur in [BLR 03] exemplarisch erörtert worden. Dort werden für einen Beispielautomaten die konfliktären, konkreten Transitionen als Event Calculus (siehe Abschnitt 3.2.4) dargestellt. Eine allgemeine Darstellung für beliebige Automaten wurde nicht vorgenommen. Aus dem Bereich des Model–Checking wurde für die Verifikationsphase der im UML Standard dargestellte Konflikt formalisiert [LiPa 99].

4.8 Zusammenfassung

In diesem Kapitel wurde ein Lösungsansatz zur Behandlung von Policy–Konflikten vorgestellt. In Kapitel 2 wurde festgestellt, dass Konfliktarten existieren, die bis jetzt in der Literatur nicht betrachtet wurden und dass zum Erkennen dieser Konfliktarten a priori Modelle des IT–Managements ein Mittel zur Konflikterkennung darstellen. Ausgehend von dieser Beobachtung wurde eine Methodik angegeben, die eine Systematik vorgibt, wie man durch die Formalisierung

von Metamodellaspekten zu einer Konfliktdefinition und schließlich zu einer Konfliktlösung gelangt.

Die Anwendbarkeit der Methodik wurde für Vertreter von statischen Modellen (funktionale Abhängigkeiten und Enthaltenseinsbeziehung) und einem Vertreter von dynamischen Modellen (endlicher Automat) gezeigt. Die Methodik beruht darauf, Modellaspekte zu formalisieren (in Form von Invarianten), Beziehungen der operationalen Policies zu den Invarianten zu identifizieren und schließlich eine Konfliktdefinition abzuleiten. Zur Konfliktlösung wird die Konfliktvermeidung vorgeschlagen, indem Policies mit Bedingungen angereichert werden, die eine Konflikt nicht eintreten lassen.

Die Anwendbarkeit der Methodik ist nicht auf die Konfliktbehandlung für Policies beschränkt (obwohl dies natürlich der Fokus dieser Arbeit ist), sondern kann für Konfliktbehandlung beliebiger Aktions–basierter Programmiertechniken angewandt werden.

An dieser Stelle kann ein weiterer Teil der Fragestellung aus Abschnitt 1.2 beantwortet werden:

2. Konfliktlösung

   (a) *Es müssen für in dieser Arbeit betrachtete Konfliktarten neue Konfliktlösungsstrategien erarbeitet werden.*
   Es müssen keine neuen Konfliktlösungsstrategien erarbeitet werden, da die in der vorliegenden Arbeit behandelten Konflikte durch Konfliktvermeidung gelöst werden können.

   (b) *Ist es möglich, dass Konflikte erst gar nicht auftreten, also vermieden werden können? Ist dieser Vorgang automatisierbar?*
   Konflikte können durch Anwendung des letzten Schritts der Methodik vermieden werden, indem Bedingungen spezifiziert werden. Der erste Teil der Methodik kann nur teilweise automatisiert werden. Allerdings kann der Aufwand der Konfliktdefinition durch Ausnutzung von Vererbungsbeziehungen, wie am Beispiel des Beziehungsmodells gezeigt, erheblich reduziert werden. Der letzte Teil der Methodik, die Abbildung von Nachbedingungen auf die betroffenen Policies, lässt sich hingegen automatisieren.

   (c) *Ist es möglich, die Konfliktlösung so zu gestalten, dass sie für beliebige Policy–Sprachen einsetzbar ist?*
   Die in der Methodik angewandte Konfliktvermeidung ist für jede Policy–Sprache anwendbar, die einen Condition–Teil unterstützt. Dies ist bei den wichtigen Policy–Sprachen (z.B. Ponder) im Bereich des IT–Managements der Fall.

3. Konfliktkategorisierung

   (a) *Ist die bis jetzt in der Literatur bekannte Konfliktkategorisierung für die in dieser Arbeit betrachteten Konflikte ausreichend?*
   Die Konfliktkategorisierung, die bis jetzt in der Literatur bekannt ist, verwendet als wesentliches Unterscheidungskriterium die Überlappung von Domänen. Wie für Konflikte bei funktionale Abhängigkeit und Enthaltenseinsbeziehung gezeigt wurde, tritt ein Konflikt auch ohne Domänenüberlappung auf. Deswegen ist es notwendig eine neue Konfliktkategorisierung zu finden.

   (b) *Welche neuen Konfliktkategorien lassen sich ableiten?*
   Die Konflikte zwischen Managementobjekten, wie sie in diesem Kapitel als neuartig identifiziert wurden, lassen sich zu neuen Konfliktarten zusammenfassen. In Abschnitt 5.5 wird eine Kategorisierung vorgenommen, in der zusätzlich Aspekte der Konfliktlösung mit aufgenommen werden.

Kapitel 5

Entwurf der Konfliktbehandlungsanwendung

Kapitelüberblick

5.1 Einleitung

Dieses Kapitel zeigt, wie man unter Verwendung der in Kapitel 4 gewonnenen Konfliktdefinitionen eine Konfliktbehandlung durchführen kann. Die dort spezifizierten Vorbedingungen zur Konfliktvermeidung können nur unter der Randbedingung der seriellen Ausführung von Policies direkt eingesetzt werden. Da aber, wie in Abschnitt 2.2.1 beschrieben, Policies normalerweise parallel ausgeführt werden sollen, muss eine Konfliktbehandlung entworfen werden, die den Aspekt der Parallelität berücksichtigt.

Dazu wird als erstes beschrieben werden, wie sich die Konfliktbehandlung in das policy–basierte Management einfügt. Daher stellt Abschnitt 5.2 den Ausführungsprozess eines Policy Decision Points (PDPs) dar. Daran anschließend werden die reaktive Konfliktbehandlung (Abschnitt 5.3) und die präventive Konfliktbehandlung (Abschnitt 5.4) vorgestellt. Die Konfliktbehandlung wird dabei in die Phasen Konfliktlokalisierung, Konflikterkennung und Konfliktlösung unterteilt. Durch die Analyse der Konfliktbehandlungsphasen kann abschließend eine Konfliktkategorisierung stattfinden, die sich entlang der Phasen orientiert (Abschnitt 5.5).

5.2 Ausführungsprozess eines PDPs

Der Policy Decision Point (PDP) ist die Komponente des policy–basierte Management (siehe Abschnitt 2.2.1), die entscheidet, ob und wann Policies ausgeführt werden. Somit ist dies auch die wichtigste Komponente für die Konfliktbehandlung. Im Folgenden wird der Ausführungsprozess eines PDPs beschrieben und diskutiert, wie die Konfliktbehandlung in diesen Prozess integriert werden kann.

Abbildung 5.1 zeigt die Ausführungsschritte eines PDPs, der eine Erweiterung von [Danc 03] um die Komponente der Konfliktbehandlung darstellt. Die Abbildung gliedert sich in drei Spalten: die mittlere Spalte stellt die einzelnen Schritte dar, die ein PDP von dem Eintreten eines Ereignisses bis zur Weitergabe der Policies an die PEPs auszuführen hat. Die rechte Spalte zeigt die notwendigen Dienste und Informationsquellen, die der PDP für die Ausführung seiner Aufgabe benötigt, und auf der linken Seite der Abbildung sind die Ein– und Ausgaben der einzelnen Prozessschritte abgebildet, die zu betrachtenden Policy–Mengen. Diese Mengen werden Schritt für Schritt bezüglich Ausführbarkeit und Konfliktfreiheit überprüft bzw. gebildet. Aus der Menge der aktiven Policies (siehe Abschnitt 2.2.4) wird sukzessive die Menge der tatsächlich ausgeführten Policies gewonnen.

Die Prozessschritte des PDP sehen im Einzelnen folgendermaßen aus (siehe auch Abbildung 5.1):

`Schritt 1:` **Warten auf Ereignis**
Der Prozess beginnt mit dem Eintreten eines Ereignisses und der Meldung, dass die Policies des letzten Verarbeitungsschrittes abgearbeitet wurde. Damit ist gewährleistet, dass sich das Managed System immer in einem konsistenten Zustand (bezogen auf die Policy–Abarbeitung) befindet, da die Auswertung des Condition–Teils (Schritt 3) sich auf den Zustand des Managed Systems beziehen kann.

`Schritt 2:` **Auswahl der Policies, die auf Ereignis triggern**
Das Ereignis wird ausgewertet und aus der Menge der aktiven Policies

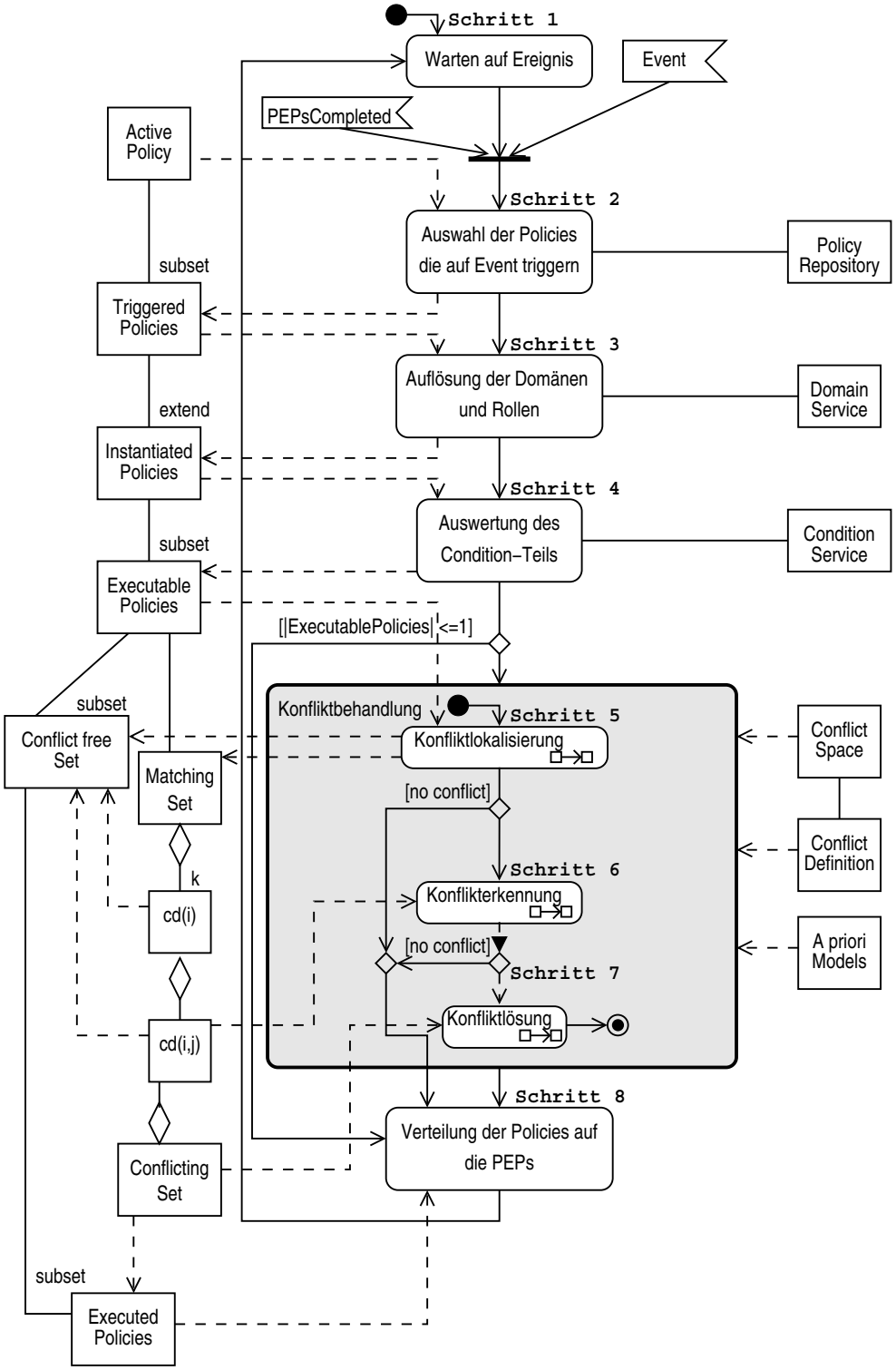

Abbildung 5.1: Allgemeine, prozessorientierte Darstellung der Ausführungsschritte eines PDPs

(die im Policy Repository gespeichert ist) werden diejenigen Policies ausgewählt, die auf dieses Ereignis triggern.

`Schritt 3:` **Auflösung der Domänen und Rollen**
Die Domänen und Rollen im Target–Teil werden mit Hilfe des Domain–Service aufgelöst, das bedeutet, dass die Domänen und Rollen in Managementobjekte aufgelöst werden und für jedes Managementobjekt die Policy instantiiert wird. Das heißt, wenn der Target–Teil durch eine Domäne, in der t Objekte enthalten sind, repräsentiert wird, dann werden t Policies instantiiert.

`Schritt 4:` **Auswerten des Condition–Teils**
Der Condition–Teil jeder Policy aus der Menge der `InstantiatedPolicies` wird ausgewertet und nur die Policies, deren Condition–Teil zu „wahr“ ausgewertet wird, kommen in die Menge der `ExecutablePolicies`. Dazu wird ein Condition–Service benutzt, der die Terme, die nicht direkt vom PDP auswertbar sind (beispielsweise der Attributwert eines Managementobjekts), auswertet.

`Schritt 5:` **Konfliktlokalisierung**
Dieser Teilschritt ist der erste Schritt der Konfliktbehandlung und betreibt die Konfliktlokalisierung. Sie stellt fest, welche Policies an einem Konflikt potentiell beteiligt sind und welche nicht. Erstere werden in der Menge der `MatchingSet` zusammengefasst und der Konflikterkennung (Schritt 6) zugeführt. Die restlichen Policies werden in der Menge `conflictFreePolicies` zusammengefasst.

`Schritt 6:` **Konflikterkennung**
Die Konflikterkennung prüft, ob sich in der Menge der `MatchingSet` Policies tatsächlich in Konflikt zueinander befinden, und legt diese in der Menge `ConflictingSet` ab. Ist kein Konflikt feststellbar, so wird im Prozess mit Schritt 8 fortgefahren.

`Schritt 7:` **Konfliktlösung**
Die Konfliktlösung wählt gemäß einer vorgegebenen Strategie einen Konfliktlösungsansatz aus und entscheidet, welche Policies aus der Menge der `ConflictingSet` tatsächlich ausgeführt werden und führt diese in die Menge `executedPolicies` über.

`Schritt 8:` **Verteilung der Policies auf die PEPs**
Die Verteilung der Policies auf die PEPs erfolgt anhand der spezifizierten Managementobjekte in den Target–Teilen der Policies.

Die Schritte 5 bis 7, also die Konfliktlokalisierung, die Konflikterkennung und die Konfliktlösung werden im folgenden Abschnitt detailliert dargestellt.

5.3 Reaktive Konfliktbehandlung

Die Vorbedingungen zur Konfliktvermeidung, wie sie in Kapitel 4 beschrieben wurde, kann nur unter der Randbedingung der seriellen Ausführung von Policies direkt eingesetzt werden. Deswegen muss eine Konfliktbehandlungskomponente die Einhaltung der Konfliktdefinitionen bei paralleler Ausführung von Policies überprüfen.

Wie im letzten Abschnitt beschrieben besteht eine Konfliktbehandlung aus den drei Phasen Konfliktlokalisierung, Konflikterkennung und Konfliktlösung. Diese Phasen werden nun in den folgenden Abschnitten beschrieben.

5.3.1 Konfliktlokalisierung

Bei der Konfliktlokalisierung wird die Menge der `executablePolicies` in Teilmengen aufgeteilt. Die Aufteilung erfolgt iterativ bis die Teilmenge alle Policies enthält, die konfliktfrei sind (`ConflictFreeSet`), und in Teilmengen, die sich auf die Konfliktdefinitionen beziehen und potentiell in Konflikt befindliche Policies enthalten. Dabei werden die Konfliktdefinitionen aus Kapitel 4 zur Analyse herangezogen.

Die einzelnen Teilschritte der Konfliktlokalisierung sind (siehe auch Abbildung 5.2):

`Schritt 5.1` **Abgleichen der Policies mit Konfliktraum**
In den Konfliktdefinitionen in Kapitel 4 wurde der sogenannte Konfliktraum definiert. Darin sind die Aktionen enthalten, die potentiell zu einem Konflikt führen können. Die Entscheidung, ob der Actions–Teil einer Policy mit Aktionen aus einem Konfliktraum (`conflict space`) übereinstimmt, kann durch syntaktische Prüfung getroffen werden. Dabei wird jede Policy auf alle Konflikträume hin untersucht. Policy–Aktionen, die in keinem Konfliktraum vorkommen, werden in die Menge der konfliktfreien Policies gespeichert (`conflictFreeSet`). Alle anderen Policies werden in der Menge der `matchingSet` gespeichert. Es gilt:
$\texttt{conflictFreeSet} \cup \texttt{matchingSet} = \texttt{executablePolicies}$

`Schritt 5.2` **Gruppieren nach Konfliktdefinitionen**
Dieser Schritt bildet Teilmengen aus der Menge der `matchingSet` dergestalt, dass die Gruppierung anhand der Konfliktdefinition–Zugehörigkeit

vorgenommen wird. Jede gebildete Teilmenge $cd(i)$ bezieht sich auf eine Konfliktdefinition i. Es können maximal soviele Teilmengen wie Konfliktdefinition auftreten. Teilmengen mit nur einem Element werden in die Menge der konfliktfreien Policies übertragen, da an einem Konflikt mindestens zwei Policies beteiligt sein müssen. Es gilt:
$\bigcup_{i}^{k} cd(i) \subseteq$ `matchingSet`
mit k als Anzahl der auftretenden Konfliktdefinitionen.

`Schritt 5.3` **Gruppieren nach Managementobjekten**
Jede einzelne Teilmenge $cd(i)$ wird nach den zur Konfliktdefinition zugehörigen Managementobjekten nochmals unterteilt in $cd(i, j)$, wobei j jeweils bezüglich der Konfliktdefinition zusammengehörige Managementobjekte darstellt. Dazu wird das Target–Objekt, das in den Policies spezifiziert ist, ausgewertet und mit den Vorbedingungen der Konfliktdefinition verglichen. Wiederum werden alle einelementigen Teilmengen in die Menge der konfliktfreien Policies übernommen. Ist eine Policy in mehreren Teilmengen enthalten, so werden diese Teilmengen zu einer gemeinsamen Teilmenge verschmolzen. Der Grund dafür ist, dass beide Mengen beide Konfliktdefinition erfüllen müssen, da ein Elemente in beiden Konfliktträumen vorkommt. Alle so gebildeten Teilmengen $cd(i, j)$ sind voneinander unabhängig. Es gilt:
$\bigcup_{j}^{m} cd(i, j) \subseteq cd(i)$
mit m als Anzahl der unterschiedlichen Managementobjekte, auf die sich eine Konfliktdefinition i bezieht.

Abbildung 5.3 zeigt die Teilmengenbildung der Konfliktlokalisierung an einem Beispiel: Es sind 10 Policies in der Menge `executablePolicies` gegeben. In `Schritt 5.1` werden die beiden Teilmengen `conflictFreeSet` und `MatchingSet` gebildet. Anschließend in `Schritt 5.2` werden innerhalb des `MatchingSet` die Policies bezüglich der Konfliktdefinitionen $cd(i)$ unterteilt und im letzten Schritt innerhalb der dieser Teilmengen nochmals in die Mengen der Managementobjekt–abhängigen $cd(i, j)$. Bei dieser letzten Unterteilung ist Policy `p10` die einzige Policy in der Menge und kann deswegen in die Menge `conflictFreeSet` eingeordnet werden.

Nach den Schritten 5.1 bis 5.3 ist die Anzahl und Größe der Mengen mit potentiell an einem Konflikt beteiligten Policies minimal.

Komplexitätsbetrachtungen Der skizzierte Algorithmus muss hinsichtlich seines Laufzeitverhaltens untersucht werden. Deshalb wird eine *worst case* Abschätzung hinsichtlich der Zeitkomplexität durchgeführt.

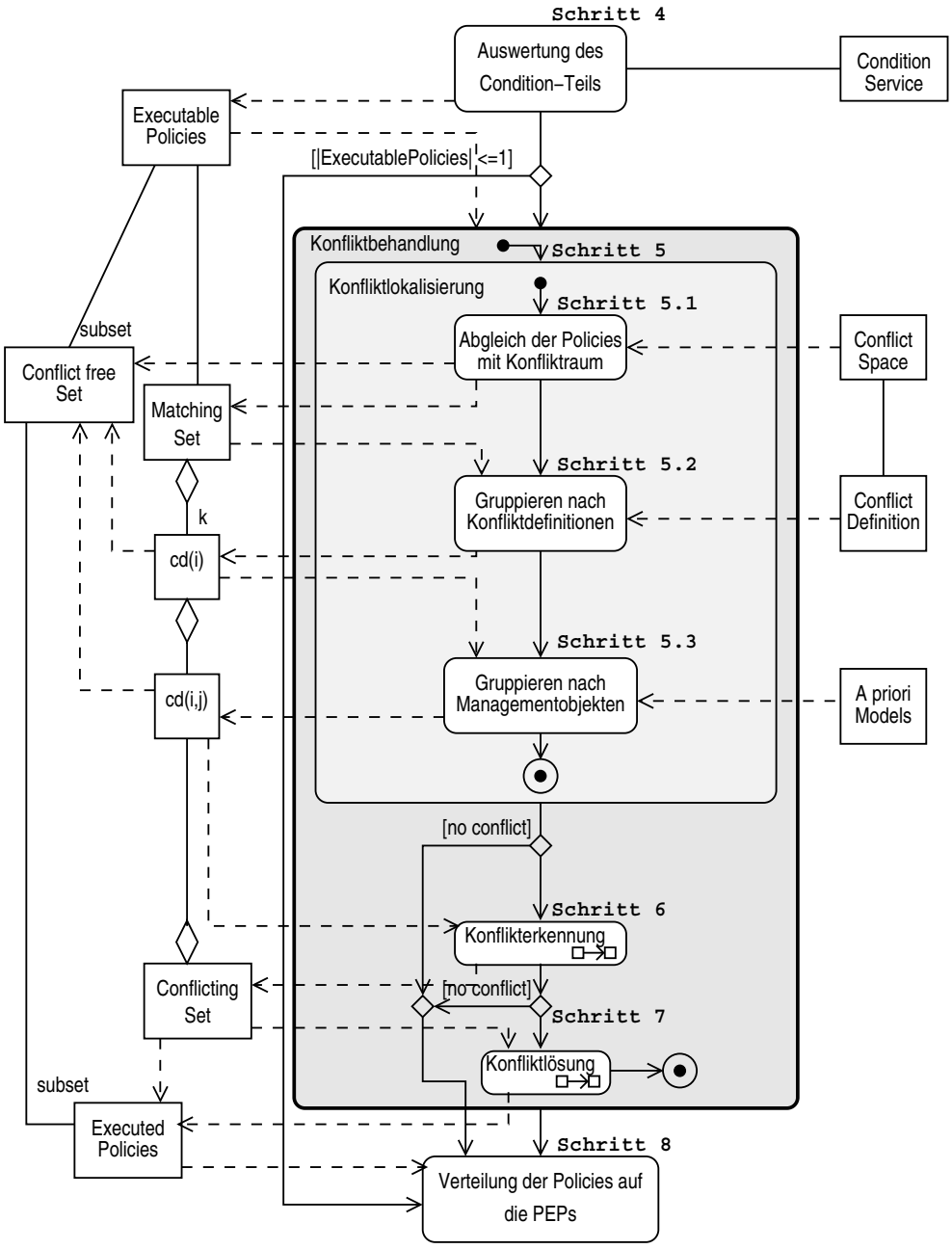

Abbildung 5.2: Der Teilprozess der Konfliktlokalisierung

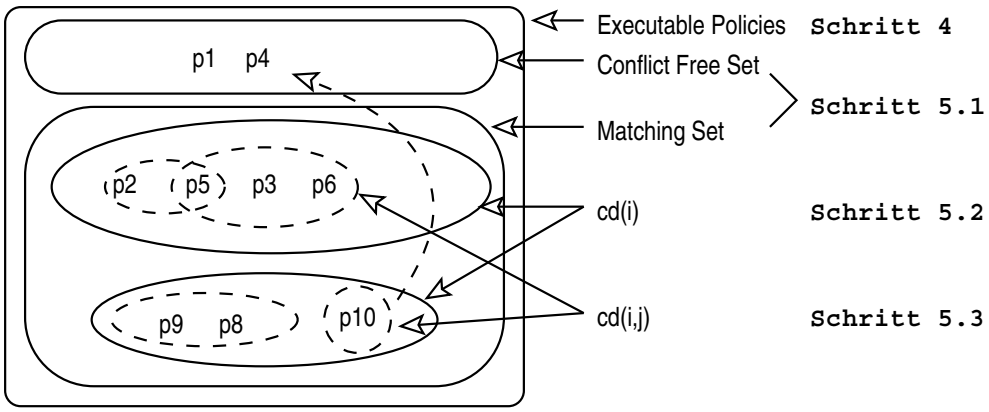

Abbildung 5.3: Teilmengenbildung bei der Konfliktlokalisierung

Sei n Anzahl der Policies in der Menge der `executablePolicies`.

`Schritt 5.1`: Jede Policy–Action muss gegen alle Aktionen in den Konfliktträumen getestet werden. Da diese aber vorab bekannt und fest sind, können diese effizient in Hashtables gehalten werden und in konstanter Zeit verglichen werden. Damit ist die Komplexität: $n * O(1) = O(n)$

`Schritt 5.2`: Die Zuordnung der Aktionen zu Konfliktträumen und damit zu Konfliktdefinitionen ist ebenfalls vorab bekannt und kann in konstanter Zeit ermittelt werden: $n * O(1) = O(n)$

`Schritt 5.3`: In diesem Schritt wird die Zuordnung der Policies zu Managementobjekten unter Zuhilfenahme der spezifizierten Vorbedingungen durchgeführt. Für jede Policy wird einmal die Vorbedingung geprüft, was einen konstanten Faktor ergibt. Damit ist die Komplexität: $n * O(1) = O(n)$

Dies stellt eine *worst case* Abschätzung dar, denn nach jedem Schritt kann sich die Anzahl der zu betrachteten Policies n deutlich verringern. Es ergibt sich somit für den Schritt der Konfliktlokalisierung eine lineare *worst case* Komplexität von $O(n)$ mit n als Anzahl der zu analysierenden Policies. Der Schritt 5.3 kann effizient parallelisiert werden, da die einzelnen Teilmengen $cd(i)$ unabhängig voneinander sind.

5.3.2 Konflikterkennung

Nachdem die Konfliktlokalisierung die Anzahl der zu untersuchenden Policies und Konflikte eingeschränkt hat, muss die Konflikterkennung nun für alle Teilmengen $cd(i, j)$ entscheiden, ob ein Konflikt vorliegt oder nicht.

Die einzelnen Teilschritte der Konflikterkennung sind folgende (siehe Abbildung 5.4):

`Schritt 6.1`: **Strategie für cd(i,j) festlegen**
Es existieren prinzipiell zwei Strategienarten:

Optimistische Strategie Man geht von der Annahme aus, dass *kein* Konflikt vorliegt. Es werden alle möglichen Reihenfolgen der Policies getestet. Liegt kein Konflikt vor, so können die Policies im weiteren parallel ausgeführt werden. Liegt ein Konflikt vor, so wird eine positiv getestete Reihenfolge der Policies ausgeführt.

Pessimistische Strategie Man geht von der Annahme aus, dass *ein* Konflikt vorliegt, dann sucht man eine Reihenfolge, in der kein Konflikt auftritt, und führt genau diese Reihenfolge aus.

Für welche Strategie man sich entscheidet kann von verschiedenen Faktoren abhängen: Die Anzahl der Policy innerhalb einer Teilmenge $cd(i,j)$ ist ein Faktor. Ist die Anzahl gering, so kann die optimistische Strategie angewandt werden, da die Anzahl der zu testenden Permutationen gering ist. Ein weiteres Kriterium ist die Komplexität der zur Auswertung einer Invarianten für eine gegebene Teilmenge $cd(i,j)$. Bei großer Komplexität verwendet man die pessimistische Strategie.

`Schritt 6.2`: **Ordnung der Policies bestimmen**
Es wird eine Ordnung (willkürlich) für die Policies einer Teilmenge $cd(i,j)$ festgelegt und gespeichert. Je nach Strategie wird dieser Vorgang für eine Teilmenge so oft wiederholt bis alle Permutationen einer Teilmenge getestet wurden.

`Schritt 6.3`: **Konflikttest durch Simulation**
Die Policies einer Menge $cd(i,j)$ werden gemäß einer der beiden obigen Strategien geprüft. Bei der Prüfung wird in einer Simulation bei der aufeinanderfolgenden Abarbeitung der Policies die betroffenen Managementobjekte gemäß der spezifizierten Nachbedingung der Aktion einer Policy manipuliert und anschließend die zugehörigen Invarianten überprüft. Wird eine Invariante verletzt, so ist ein Konflikt eingetreten; wird bei einem kompletten Durchlauf für eine Menge $cd(i,j)$ keine Invariante verletzt, so tritt kein Konflikt ein.

Ist gemäß einer Strategie eine Menge $cd(i,j)$ als konfliktfrei bewertet worden, so wird diese Menge in die Menge der konfliktfreien Policies eingefügt (unter Berücksichtigung einer eventuellen Ausführungsordnung der Policies).

Findet man für eine Teilmenge $cd(i, j)$ keine Reihenfolge, so dass kein Konflikt eintritt, so muss diese Teilmenge der Konfliktlösung zugeführt werden.

Komplexitätsbetrachtung Sei m die Anzahl der zu betrachtenden Teilmengen $cd(i, j)$ (mit maximal n Policies).

`Schritt 6.1`: Die Auswahl einer Strategie für eine Teilmenge ist von konstantem Faktor, damit gilt: $m * O(1) = O(m)$

`Schritt 6.2`: Dieser Schritt übergibt eine Teilmenge, deshalb gilt für diesen Schritt: $m * O(1) = O(m)$

`Schritt 6.3`: Beide Strategien, die optimistische und die pessimistische Strategie, brauchen in der *worst case* Abschätzung $n!$ viele Schritte, da alle möglichen Reihenfolgen getestet werden müssen. Die Komplexität ist somit: $O(n^n)$.
Hier kann durch die Zuhilfenahme der Konfliktdefinitionen wesentlich bessere Ergebnisse erzielen. Beispielsweise genügt für die Konfliktdefinition `upperLimit` der Test *einer* Reihenfolge für `bind()` Aktionen, jede weiter Reihenfolge liefert dasselbe Ergebnis. Deshalb wird in Abschnitt 5.5 die Konfliktdefinitionen aus Kapitel 4 bezüglich ihres Verhalten bei der Konfliktbehandlung untersucht und erweitert.

5.3.3 Konfliktlösung

Durch die Konflikterkennung wurde bereits eine wesentliche Vorarbeit geleistet. Die Konfliktmengen $cd(i, j)$ sind bereits darauf überprüft, dass keine Reihenfolge der Policies existiert, in der kein Konflikt eintritt.

Schritt 7.1: **Strategie für cd(i,j) wählen**
Es existieren folgende grundlegende Strategien zur Konfliktlösung:

A: Keine Policy wird ausgewählt. Der Konflikt ist für diese Policy–Menge nicht lösbar. Der Konflikt wird dem Managementsystem bekanntgegeben, indem man die beteiligten Policies, den Konflikt und die konkret beteiligten Managementobjekte weitergibt. Eine Konfliktanalyse kann dann von Hand erfolgen.

B: Alle Policies werden ausgewählt. Diese Strategie ist das Gegenteil der vorhergehenden Strategie. Der Konflikt wird ignoriert, er tritt unter der Voraussetzung, dass die Konflikterkennung wie in Abschnitt 5.3.2 beschrieben, durchgeführt wurde sicher ein.

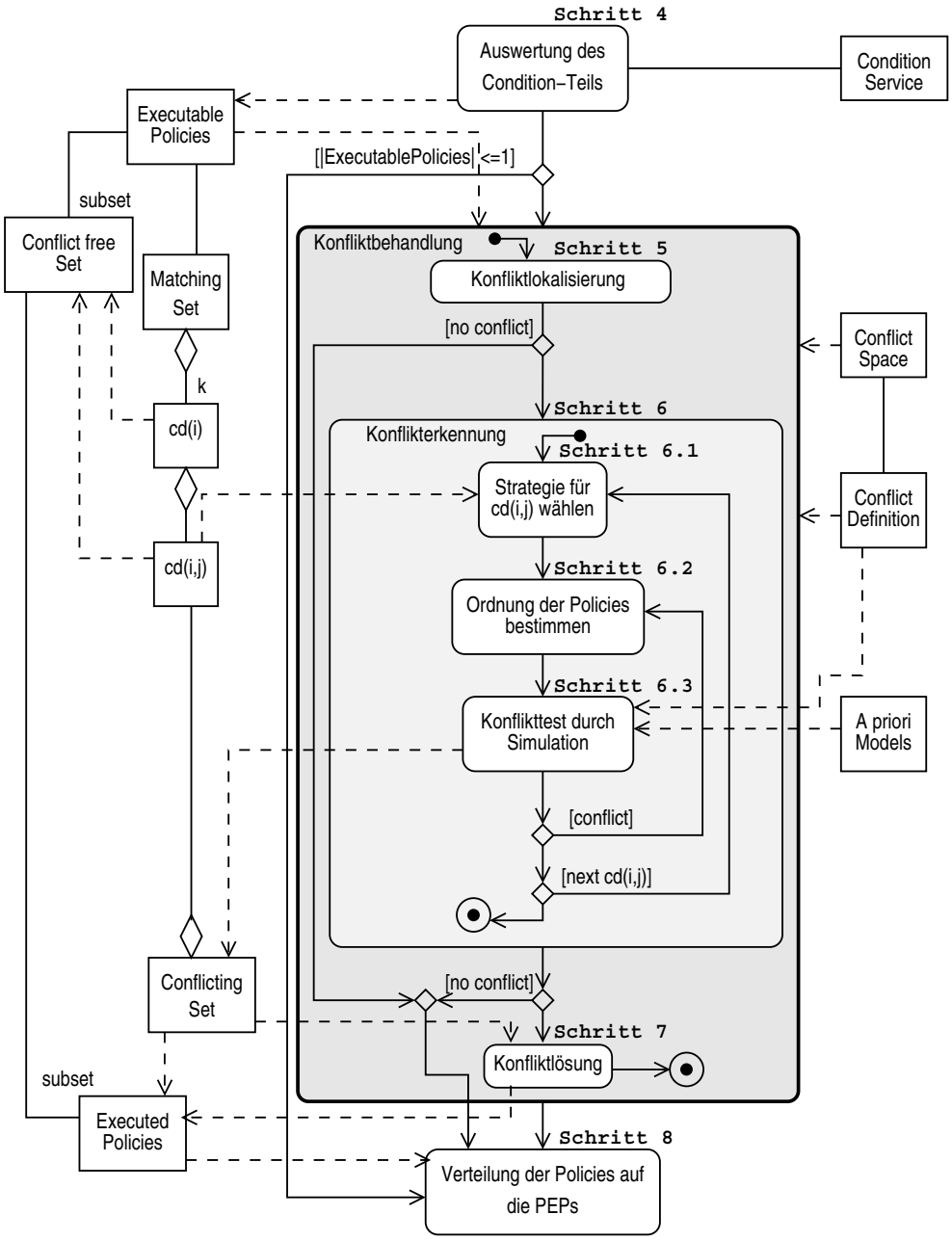

Abbildung 5.4: Der Teilprozess der Konflikterkennung

C: Genau eine Policy wird ausgewählt. Diese Strategie ist die in der Literatur am häufigsten vorgeschlagene Strategie [LuSl 99, Dami 02, KKK 96, Verm 00]. Es wird genau eine Policies aus der Menge ausgewählt und ausgeführt. Zur Auswahl werden meist Prioritäten verwendet, die bereits in der Spezifikationsphase einer Policy vergeben werden (in [LuSl 99], siehe Abschnitt 3.2.2, werden weitere Auswahlkriterien vorgeschlagen).

D: Eine Untermenge an Policies wird ausgewählt. Sind mehrere Policies zueinander in Konflikt, so besteht die Möglichkeit, dass eine Teilmenge der in Konflikt befindlichen Policies ausführen kann, ohne dass ein Konflikt eintritt. Damit wäre das Kriterium erfüllt, möglichst viele Policies auszuführen.

Schritt 7.2: **Strategie ausführen**
Für die beiden ersten Strategien (A und B) ist dieser Schritt trivial. Entweder wird die Konfliktmenge $cd(i,j)$ gelöscht oder komplett in die Menge der konfliktfreien Policies übernommen. Bei der Strategie C muss die am höchsten priorisierte Policy ausgewählt werden, die restlichen Policies werden aus der Menge $cd(i,j)$ gelöscht. In Strategie D wird eine Policy gelöscht. Die Auswahl der zu löschenden Policy kann durch Prioritäten unterstützt werden. Daraufhin muss überprüft werden, ob diese Teilmenge tatsächlich konfliktfrei ist. Dies wird durch den `Schritt 6.2 Test durch Simulation` überprüft (siehe Abbildung 5.5). Dieser Algorithmus terminiert spätestens, wenn nur noch eine Policy in der Menge vorhanden ist.

Komplexitätsbetrachtung Sei m die Anzahl der zu betrachtenden Teilmengen $cd(i,j)$ (mit maximal n Policies).

`Schritt 7.1` Die Auswahl einer Strategie für eine Teilmenge ist von konstantem Faktor, damit gilt: $m * O(1) = O(m)$

`Schritt 7.2` Strategie A und B sind in konstanter Zeit ausführbar: $m*O(1) = O(m)$. Für Strategie C muss in jeder Menge die Policy mit der höchsten Priorität ermittelt werden: $m * O(n) = O(m * n)$. Bei der Strategie D werden Policies gelöscht und dem Schritt 6.2 zugeführt: $(n-1) * O(n^n) = O(n^n)$

5.4 Präventive Konfliktbehandlung

Die präventive Konfliktbehandlung wird zur Zeit der Policy–Spezifikation durchgeführt. In dieser Phase sollen Policy–Konflikte behandelt werden, ohne dass Po-

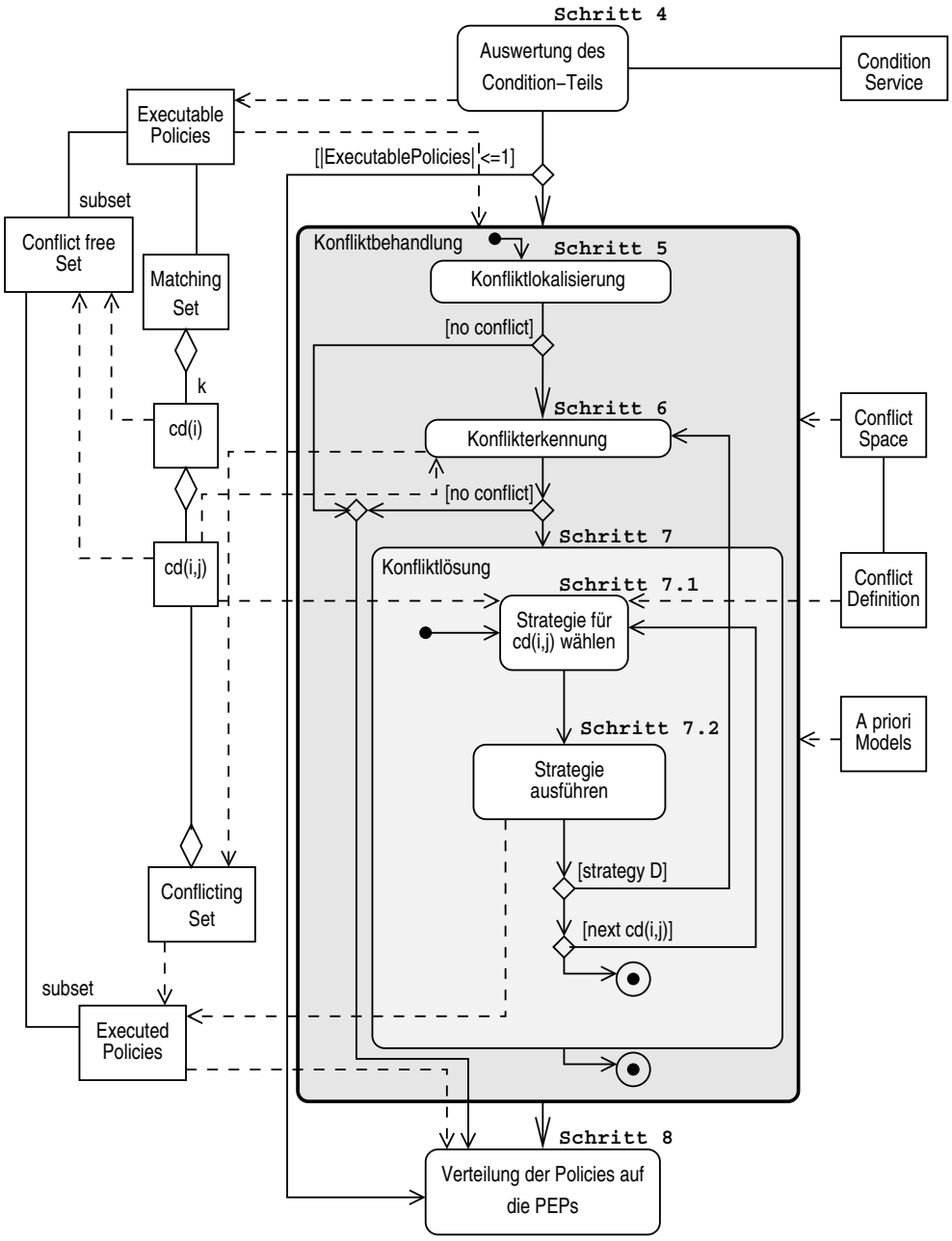

Abbildung 5.5: Der Teilprozess der Konfliktlösung

licies tatsächlich ausgeführt werden.

Hierfür ist zu untersuchen, in wie weit der Prozess des PDP, wie er in Abschnitt 5.2 dargestellt wurde, präventiv angewandt werden kann. Die einzelnen Teilschritt werden nun analysiert:

`Schritt 1:` **Warten auf Ereignis**
Dieser Schritt existiert bei der präventiven Konfliktbehandlung nicht.

`Schritt 2:` **Auswahl der Policies die auf Ereignis triggern**
Die Menge aller aktiven Policies wird bezüglich der spezifizierten Ereignisse in disjunkte Teilmengen aufgeteilt.

`Schritt 3:` **Auflösung der Domänen und Rollen**
Die Domänen und Rollen müssen bezüglich der aktuellen Instanzmodelle ausgewertet werden. Wie in Abschnitt 2.5 und Kapitel 4 dargestellt, sind die in dieser Arbeit untersuchten Konflikte modellabhängig. Die Domänen und Rollen im Target–Teil werden mit Hilfe des Domain–Service aufgelöst, das bedeutet, dass die Domänen und Rollen in Managementobjekt–Klassen aufgelöst werden.

`Schritt 4:` **Auswerten des Condition–Teils**
Der Condition–Teil kann im Allgemeinen nicht präventiv ausgewertet werden, da der Zustand des Instanzmodells bei der präventiven Konfliktbehandlung nicht bekannt ist. Daraus folgt, dass die nachfolgende Konfliktbehandlung nur potentielle Konflikte analysieren kann.

`Schritt 5:` **Konfliktlokalisierung**
Dieser Teilschritt ist der erste Schritt der Konfliktbehandlung und betreibt die Konfliktlokalisierung. Sie stellt fest, welche Policies an einem Konflikt potentiell beteiligt sind und welche nicht. Die Konfliktlokalisierung besteht aus folgenden Teilschritten (siehe Abschnitt 5.3.1):

`Schritt 5.1` **Abgleichen der Policies mit Konfliktraum** Dieser Schritt kann präventiv durchgeführt werden, da der Konfliktraum bereits spezifiziert ist.

`Schritt 5.2` **Gruppieren nach Konfliktdefinitionen** Dieser Schritt kann ebenfalls präventiv durchgeführt werden, denn die Konfliktdefinition sind bereits spezifiziert.

`Schritt 5.3` **Gruppieren nach Managementobjekten** Dieser Schritt wird bei der präventiven Konfliktlokalisierung für die aktuellen Instanzmodelle durchgeführt.

`Schritt 6:` **Konflikterkennung**
Die Konflikterkennung prüft, ob sich in der Menge der `MatchingSet` Policies tatsächlich in Konflikt zueinander befinden und legt diese in der Menge `ConflictingSet` ab.

`Schritt 6.1:` **Strategie für cd(i) festlegen** Wie in Abschnitt 5.5 beschrieben kann nicht für jede Konfliktdefinition der Konflikterkennungsschritt präventiv durchgeführt werden, da Konfliktdefinition existieren, die Instanzmodellabhängig sind. Diese Teilmengen $cd(i)$ werden in diesem Schritt aussortiert.

`Schritt 6.2:` **Ordnung der Policies bestimmen** Dieser Teilschritt wird präventiv analog zum reaktiven Fall durchgeführt.

`Schritt 6.3:` **Konflikttest durch Simulation** Dieser Teilschritt wird präventiv analog zum reaktiven Fall durchgeführt.

`Schritt 7:` **Konfliktlösung**
Die Konfliktlösung wählt gemäß einer vorgegebenen Strategie einen Konfliktlösungsansatz aus. Im Fall der präventiven Konfliktbehandlung soll der Policy–Spezifizierer auf den Konfliktfall informiert werden. Durch die Angabe der verletzten Konfliktdefinitionen und der Menge der Policies $cd(i)$ kann dem Policy–Spezifizierer eine detailierte Konfliktanalyse präsentiert werden.

Die präventive Konfliktbehandlung endet mit diesem Schritt.

`Schritt 8:` **Verteilung der Policies auf die PEPs**
Dieser Schritt findet bei der präventiven Konfliktbehandlung nicht statt.

Ergebnis Es besteht die Einschränkung bei der präventiver Konfliktbehandlung bezogen auf die reaktiver Konfliktbehandlung: der Condition–Teil kann im Allgemeinen nicht ausgewertet werden, somit können nur potentielle Konflikte betrachtet werden. Im Spezialfall, dass kein Condition–Teil für eine zu untersuchende Menge $cd(i)$ spezifiziert wurde, gilt diese Einschränkung nicht.

5.5 Konfliktkategorisierung

Die Analyse der reaktiven und präventiven Konfliktbehandlung hat bezüglich der Konfliktdefinition aus Kapitel 4 neue Erkenntnisse gebracht, die nun analysiert werden. Diese Analyse mündet in eine Konfliktkategorisierung.

Neben den bereits getroffenen Kriterien für Konfliktdefinitionen (beispielsweise die Invariante, auf die sich eine Konfliktdefinition bezieht) konnten bei der Konfliktbehandlung weitere wichtige Kriterien identifiziert werden:

Reihenfolgeabhängigkeit Die Konflikterkennung untersucht verschiedene Reihenfolgen von Policies in einer Menge $cd(i)$ (siehe Abschnitt 5.3.2). Manche der Konfliktdefinition sind von der Reihenfolge der Aktionen unabhängig, das heißt, es genügt genau eine Reihenfolge zu testen.

Strategie zur Konfliktlösung Für Konfliktdefinition kann bereits in der Spezifikationsphase eine optimale Strategie zur Konfliktlösung festgelegt werden.

Zusammen mit den Konfliktkriterien aus Abschnitt 2.5.4 ergibt sich folgende Konfliktklassifikation:

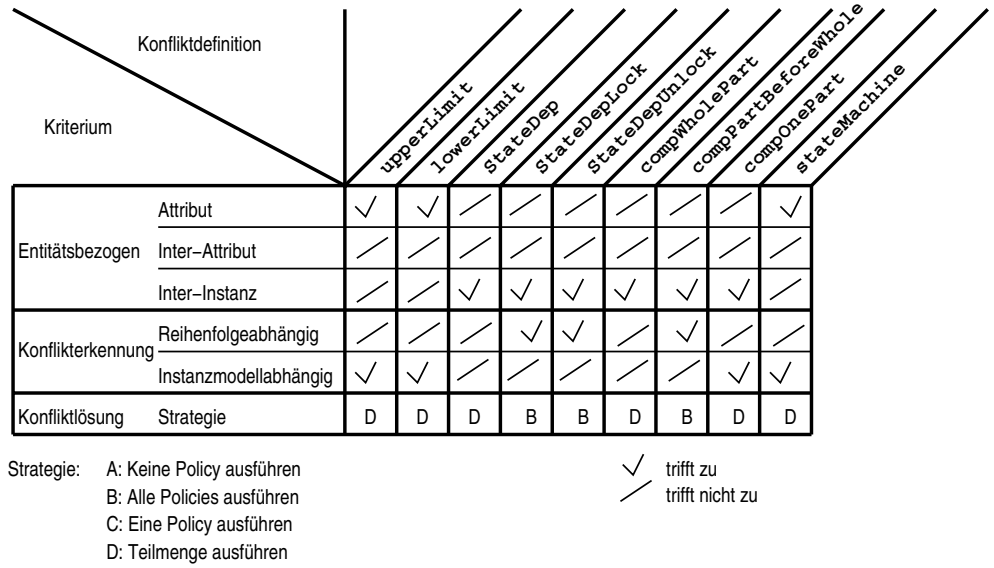

Abbildung 5.6: Kategorisierung der Konflikte

Die Konflikte sind mit ihren Namen in den Spalten eingetragen. Alle Kriterien, die einen Konflikt kennzeichnen, sind in den Zeilen geordnet. Das erste Kriterium ist durch die Entitätsbezogenheit gekennzeichnet. Jeder Konflikt gehört einer der drei Möglichkeiten. Das nächste Kriterium bezieht sich auf die Erkennung der einzelnen Konflikte. Reihenfolgeabhängig bedeutet, dass das Eintreten eines Konflikts davon abhängt, in welcher Reihenfolge die Policies ausgeführt werden.

Das Kriterium Konfliktlösung gibt die optimale Strategie für die Konfliktdefinition an. Für alle Konflikte bis auf einen ist die Strategie „Teilmenge ausführen“ optimal, wenn möglichst viele Policies ausgeführt werden soll. Bei dem Konflikt `compPartBeforeWhole` können alle Policies ausgeführt werden, lediglich eine korrekte Reihenfolge muss gewährleistet werden.

5.6 Zusammenfassung

In diesem Kapitel wurde eine Konfliktbehandlung für den reaktiven und präventiven Fall entwickelt. In der Literatur existieren bis jetzt keine ausführlichen Algorithmen zur Policy–Konfliktbehandlung. Es wurde der Prozess des PDPs um die Komponente der Konfliktbehandlung erweitert. Die Konfliktbehandlung setzt sich aus den drei Phasen Konfliktlokalisierung, Konflikterkennung sowie Konfliktlösung zusammen. Für jede Phase konnten effiziente Verfahren angegeben werden, um Policy–Konflikte zu behandeln. Die reaktive und präventive Konfliktbehandlung können großteils mit den gleichen zeiteffizienten Algorithmus betrieben werden. Alle bis auf einen Teilschritt besitzen eine worst-case Komplexität von $O(n)$. Lediglich ein Teilschritt hat die worst-case Komplexität $O(n^n)$. Für diesen Teilschritt konnten aber für viele Konfliktarten effiziente Heuristiken angegeben werden. Bei der präventiven Konfliktbehandlung kann der Condition–Teil nicht ausgewertet werden. Somit können nur potentielle Konflikte erkannt werden. Durch die Analyse der Konfliktbehandlung konnten weitere Kriterien zur Unterscheidung von Konflikten abgeleitet werden. Diese mündeten in eine an den Phasen der Konfliktbehandlung orientierte Konfliktkategorisierung.

Die Konfliktbehandlung prüft bei der Verletzung von Invarianten, ob ein Konflikt ursächlich für die Verletzung verantwortlich ist. Ist dies nicht der Fall, so liegt zumindest ein Fehler vor. Das bedeutet, dass die Definition von Invarianten auch zum Erkennen von Fehlern beiträgt.

Kapitel 6

Integration des Lösungsansatzes in ein Informationsmodell am Beispiel des Common Information Models

Kapitelüberblick

6.1 Einleitung

In Kapitel 4 wurde eine generische Methodik zur Konfliktbehandlung entwickelt. Dabei wurde mittels eines allgemeinen Beziehungsmodells die Anwendbarkeit

der Methodik gezeigt. Da die Methodik sowie das Beziehungsmodell bewußt generisch gehalten wurden, wird in diesem Kapitel exemplarisch die Abbildung in ein existierendes Management–Informationsmodell beschrieben. Dieses Kapitel beschreibt nun anhand des Common Information Models, einem Management–Informationsmodell, wie das Beziehungsmodell und die Methodik in diesem Fall abzubilden bzw. anzuwenden sind.

Der folgende Abschnitt gibt eine Einführung in das Common Information Model. Abschnitt 6.3 beschreibt die Abbildung des Beziehungsmodells und der in OCL spezifizierten Bedingungen nach CIM. Abschnitt 6.4 zeigt, wie die Schritte der Methodik im Falle des Common Information Models anzuwenden sind.

6.2 Das Common Information Model

Das Common Information Model (CIM) [CIM 2.2] der Distributed Management Task Force (DMTF) ist ein objektorientiertes Managementinformationsmodell. Ziel von CIM ist es einen plattform– und technologieunabhängigen Austausch von Management–Informationen zu gewährleisten und damit die Interoperabilität von Managementsystemen zu unterstützen. Die DMTF wurde 1992 gegründet und viele namhafte Vertreter aus dem IT–Sektor sind Mitglied: Microsoft, IBM, HP, Sun Microsystems, Cisco Systems, Intel uvm.

Es folgt ein kurzer Überblick über den CIM Standard, anschließend werden die für die Fragestellung der Abbildung in Frage kommenden Modelle detailliert vorgestellt.

6.2.1 CIM im Überblick

Der CIM Standard definiert Modelle, die unterschiedlich abstrakt sind und unterschiedliche Anwendungsbereiche adressieren:

Core Model legt die allgemeingültigen Oberklassen für alle Managementbereiche fest und stellt meist abstrakte Klassen dar. Das Modell ist in acht Bereiche wie beispielsweise `LogicalElement`, `Product` und `SettingData` unterteilt.

Common Model erweitert das Core Model und fokussiert auf unterschiedliche Managementbereiche. Es existieren in der Version 2.8 dreizehn verschiedene Common Models wie zum Beispiel `Application`, `Network`, `Metrics` oder `Physical`. Diese Modelle sind so allgemein gehalten, dass keine Herstellerspezifika aufgewiesen werden.

Extension Schema stellen schließlich eine technologiespezifische Erweiterung der Common Models dar.

Das Core Model und das Common Model zusammen bezeichnet CIM als *CIM Schema*. Sie sind derzeit in Version 2.8.1 verfügbar. Die Extension Schema sind nicht Teil des CIM Standards.

Der CIM Standard spezifiziert ein Metamodel, das sogenannte Meta Schema. Das Meta Schema legt formal Form und Inhalt von CIM Modellen fest. Zur Definition des Meta Schemas wird UML [uml 03] verwendet.

Als textuelle Beschreibungssprache setzt der CIM Standard das Managed Object Format (MOF) ein, eine Sprache, die auf der Interface Definition Language (IDL) der Object Management Group (OMG) basiert [CORBA-302]. In MOF können Klassen, Assoziationen und Instanzen beschrieben werden.

Das Meta Schema und das Core Model werden nun detailliert beschrieben, da beide Modelle Kandidaten für die Einbindung von OCL–Konstrukten, wie sie in Kapitel 4 zur Beschreibung von Invarianten und Konfliktdefinitionen verwendet wurden, sind.

6.2.2 CIM Meta Model

Abbildung 6.1 zeigt das Metamodell von CIM. Die Oberklasse des Metamodells ist das Element `NamedElement`. Alle Elemente des Metamodells erben direkt oder über Subklassen von dieser Oberklasse. Eine Klasse `Class` besteht aus Methoden `Method` und Eigenschaften `property` (in der Objektorientierten Welt meist als Attribut bezeichnet). Eine Beziehung `Association` ist eine Klasse und besitzt Referenzen `Reference`. Ein Trigger `Trigger` zeigt eine Zustandsänderung an. `Indikation` ist ein Objekt, das als Ergebnis eines Triggers erzeugt wurde. Ein Schema `schema` ist eine Ansammlung von Klassen. Ein `Qualifier` charakterisiert ein beliebiges Element. Beispielsweise stellt eine Kardinalitätsgrenze einen `Qualifier` dar.

6.2.3 CIM Core Model

Abbildung 6.2 zeigt das Übersichtsdiagramm des Core Models. Die abstrakte Klasse `ManagedElement` ist die Oberklasse aller weiteren Klassen in CIM. Die weiteren Klassen wie `Capabilities` oder `Setting` werden in gesonderten Modellen detailliert dargestellt.

Als weiteres Beispiel des Core Models zeigt Abbildung 6.3 das Model der abstrakten Klasse `LogicalElement` und seiner Subklassen.

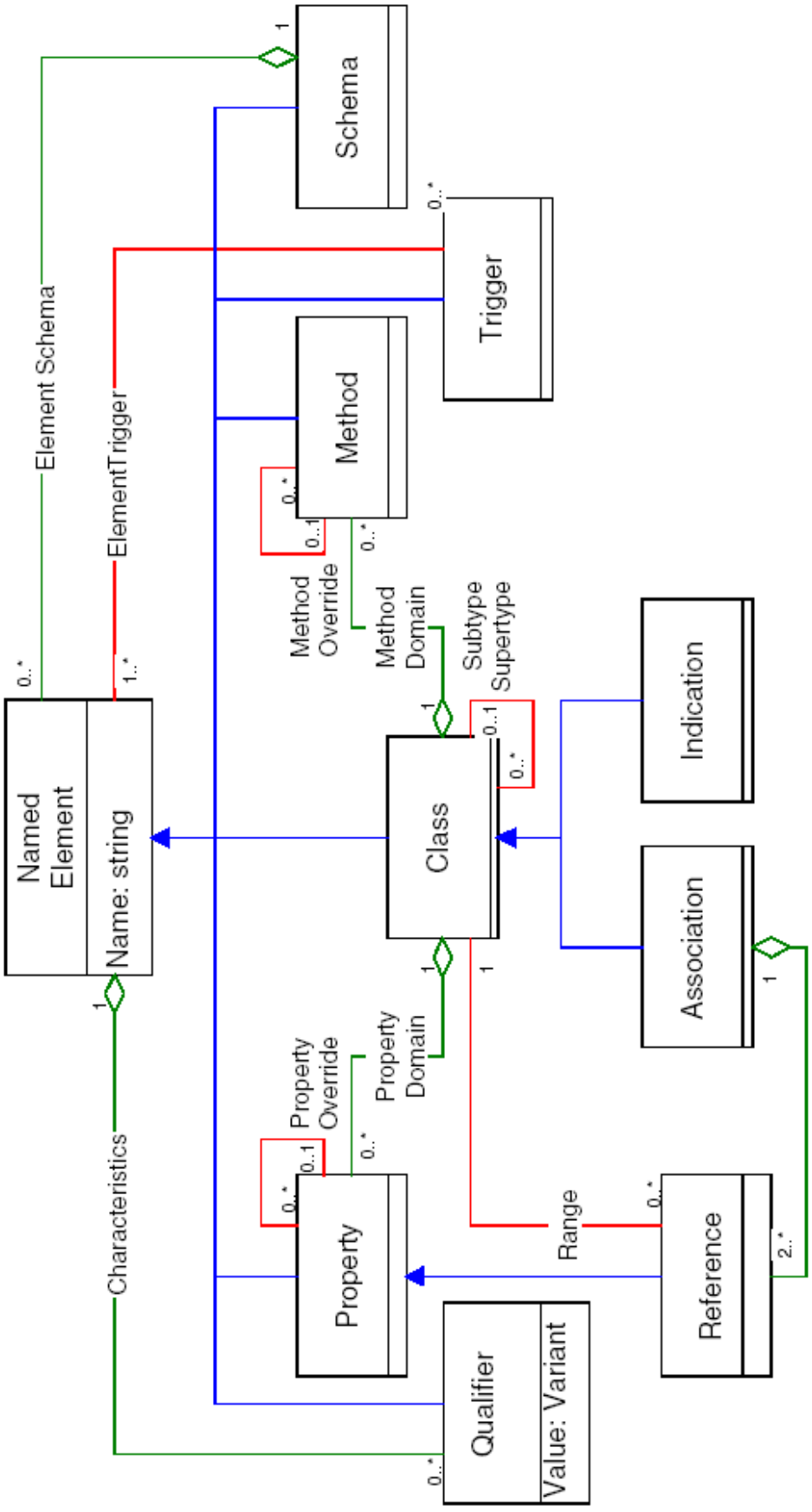

Abbildung 6.1: CIM Meta Schema Structure aus [CIM 2.2]

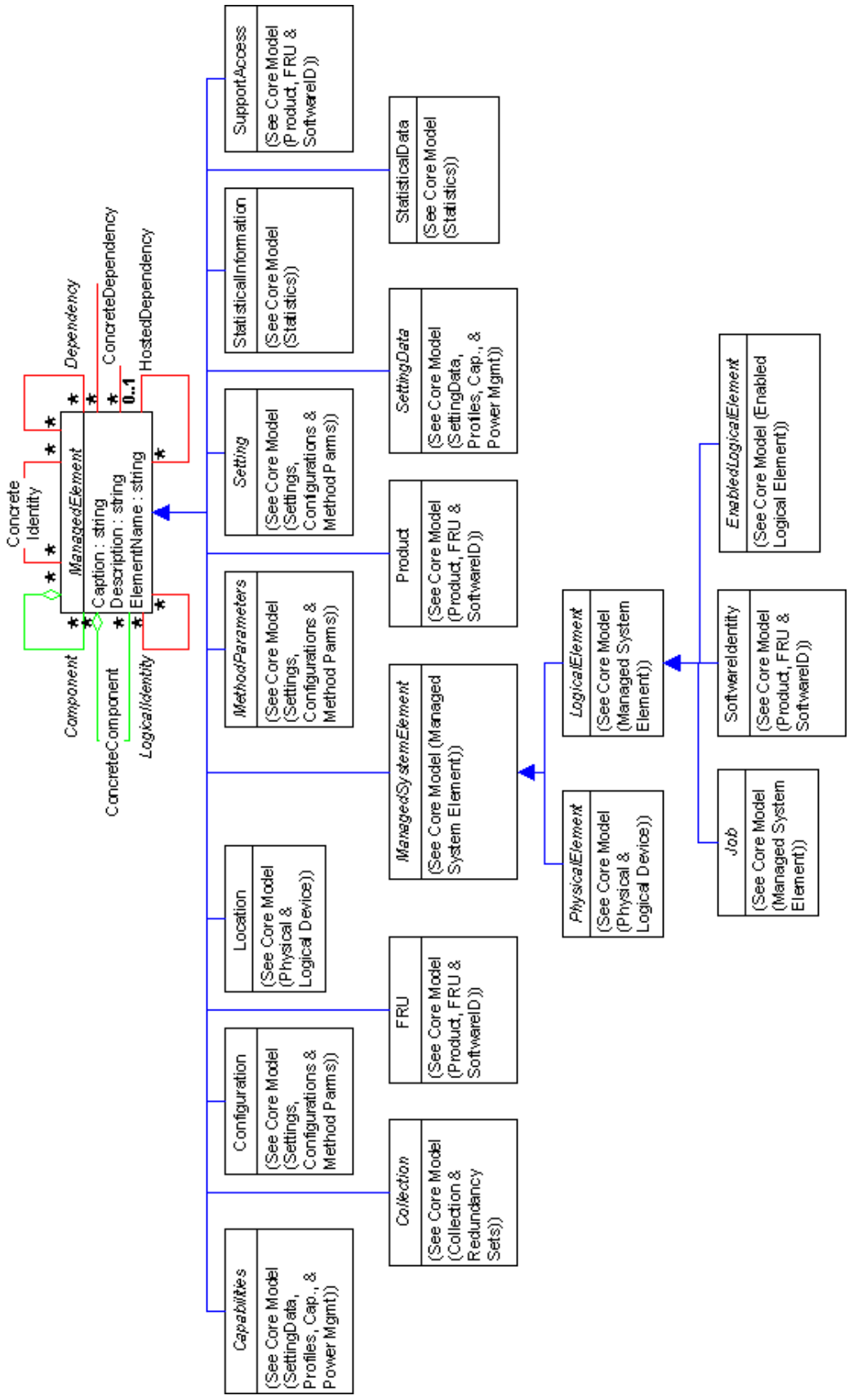

Abbildung 6.2: Übersichtsdiagramm des CIM Core Model [CIM 2.2]

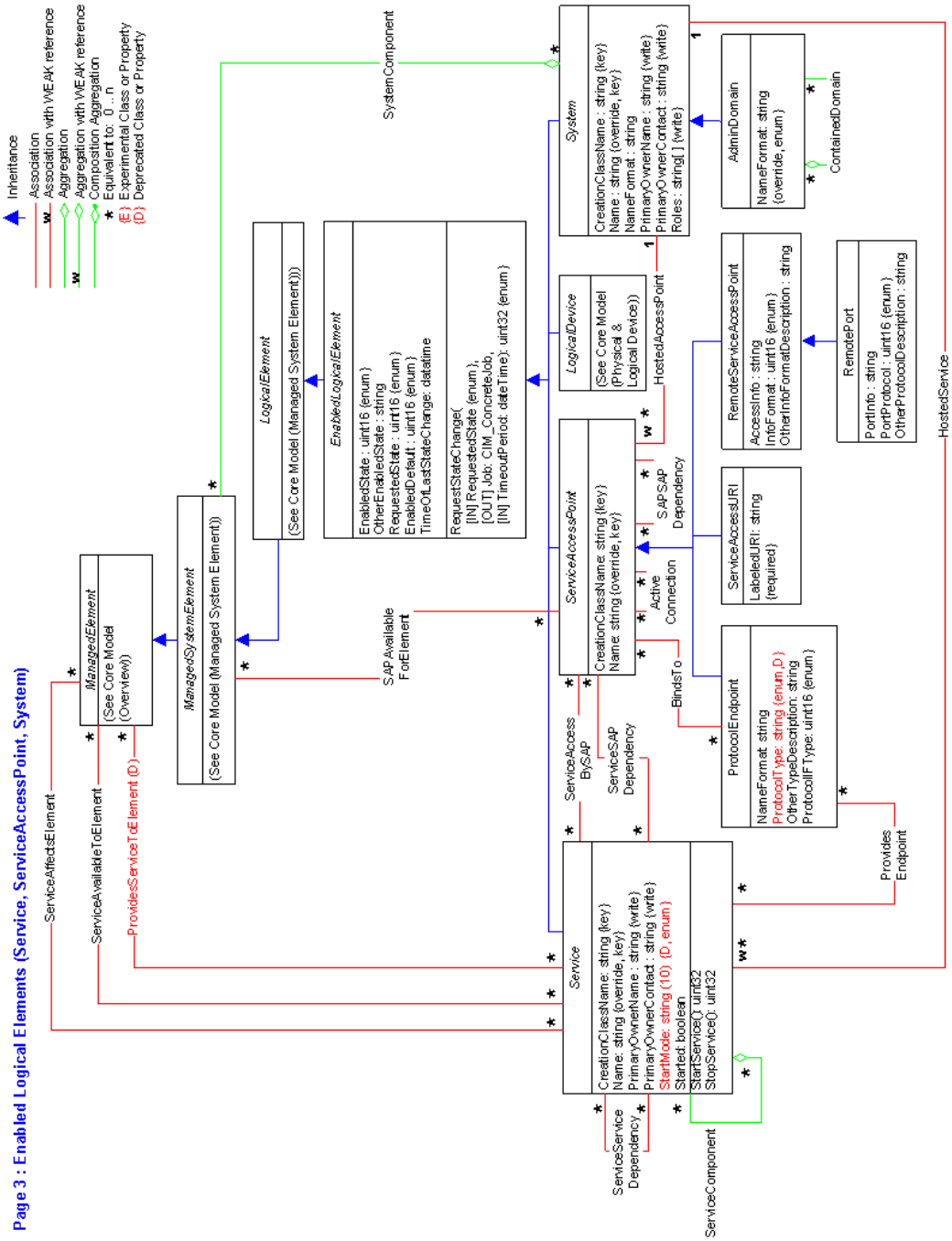

Abbildung 6.3: CIM Core Model der abstrakten Klasse `LogicalElement` [CIM 2.2]

Von der abstrakten Klasse `EnabledLogicalElement`, welche zahlreiche Zustände modelliert, werden alle weiteren Klassen für dieses Modell abgeleitet. Wichtigste Vertreter sind die Klasse `Service`, welche als Oberklasse zur Modellierung von Diensten jeder Art dienen soll und die abstrakte Klasse `System`, welche aus Komponenten besteht und ein funktionales Ganzes repräsentiert (beispielsweise ein Computer System).

CIM betont die Modellierung von Beziehungen, die über die Vererbungsbeziehung hinaus gehen, wie an diesem Modell zu sehen ist. Beispielsweise ist jeder Service einem System zugeordnet. Ebenso ist ein Service über die Assoziation `ServiceAccessBySAP` mit der Klasse `ServiceAccessPoint` verbunden.

CIM unterscheidet drei Haupttypen von Beziehungen:

Dependency Abhängigkeiten werden durch Dependency–Beziehungen dargestellt. Jede Dependency hat zwei Assoziationsenden, einen `Antecedent` und einen `Dependent`. Die Dependencies werden in einer flachen Dependency–Hierarchie geordnet.

Aggregation stellt die Enthaltenseinsrelationen dar. Alleine das Core Model unterscheidet über zwanzig verschiedenen Aggregationsklassen, die aber nur in Ausnahmenfällen eine Vererbungsbeziehung zueinander haben.

Association ist die allgemeinste Form von Beziehungen. In ihr werden alle Beziehungen modelliert, die keine Abhängigkeit oder Aggregation darstellen. Ebenfalls stehen die meisten Assoziationsklassen nicht mit anderen Assoziationsklassen in Beziehung.

Abbildung 6.4 zeigt die Dependency–Hierarchie des Core Models. Alle Abhängigkeitsklassen erben von der abstrakten Oberklasse `Dependency`.

6.3 Abbildung der Modelle nach CIM

Nachdem im letzten Abschnitt die Modelle des CIM Standards vorgestellt wurden, wird in diesem Abschnitt die Abbildung der OCL–Konstrukte sowie des generischen Beziehungsmodells aus Kapitel 4 diskutiert.

6.3.1 Abbildung von OCL-Konstrukten nach CIM

Damit die Konfliktdefinition, Invarianten, Vor– und Nachbedingungen (siehe Kapitel 4), die in OCL textuell gegeben sind, auf die graphisch orientierten CIM

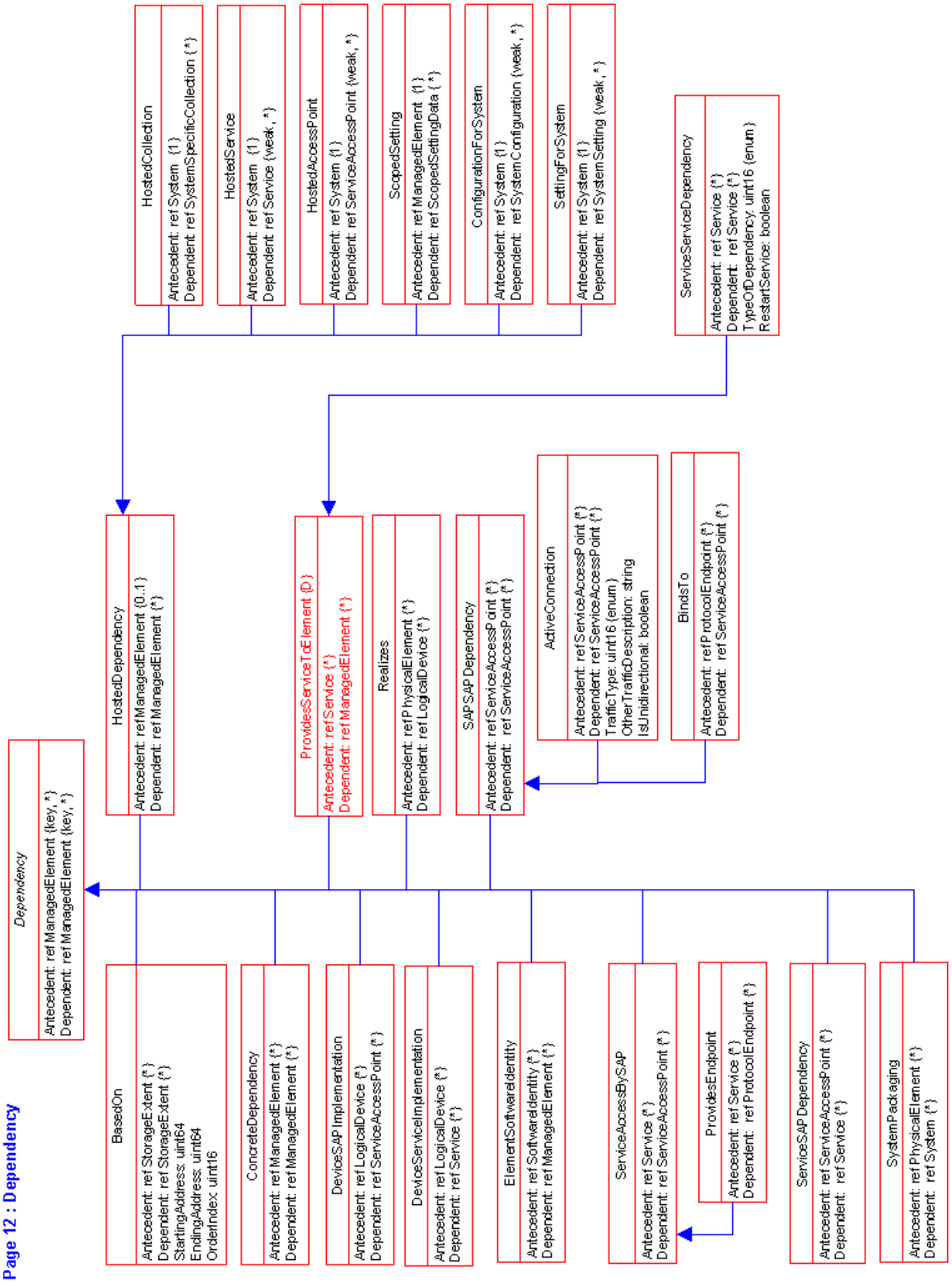

Abbildung 6.4: CIM Core Model Dependency–Hierarchie [CIM 2.2]

Modelle abgebildet werden können, ist es notwendig, eine graphische Notation der OCL–Konstrukte vorzunehmen.

Abbildung 6.5 zeigt die graphische Darstellung in UML Notation der Konfliktdefinition aus Kapitel 4. Es werden die einzelnen OCL-Blöcke auf Klassen abgebildet und zueinander in Beziehung gesetzt. Jede Konfliktdefinition bezieht sich auf eine Invariante und besteht aus dem Konfliktraum und einer Vorbedingung. Jeder Konfliktraum wiederum besteht aus einer Menge von Methoden.

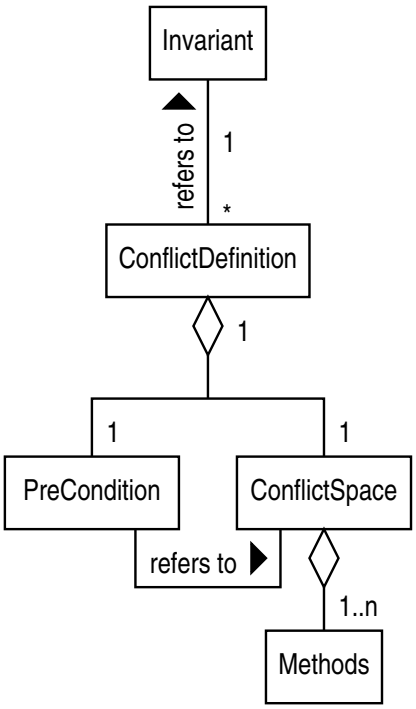

Abbildung 6.5: Modellierung der Konfliktdefinition

Prinzipiell existieren zwei mögliche Ebenen, in der die Konfliktdefinition eingehängt werden kann: die Konfliktdefinition kann in das Core Model oder im Metamodell eingehängt werden, beide Möglichkeiten werden im CIM Standard als Erweiterungsstrategien vorgeschlagen und nun diskutiert. Der CIM Standard zur Erweiterung der Modelle sieht vor, Klassen zu den Modellen hinzuzufügen oder sogenannte `Qualifier` zu erweitern.

Einbindung im Core Model

Da CIM wie das generische Beziehungsmodell ebenfalls auf UML basiert, kann das Modell der Konfliktdefinition ohne zusätzlichen Aufwand in CIM an die ent-

sprechende Stelle eingehängt werden.

Es soll möglich sein, dass prinzipiell für jedes Managementobjekt Invarianten und somit Konfliktdefinitionen spezifiziert werden können. Deswegen wird das Modell der Konfliktdefinition an die Oberklasse aller Managementobjekte, die Klasse `ManagedElement` gehängt.

Abbildung 6.6 zeigt die Einbindung der Konfliktdefinition in das CIM Modell. Jedem Managementobjekt kann können Invarianten assoziiert werden.

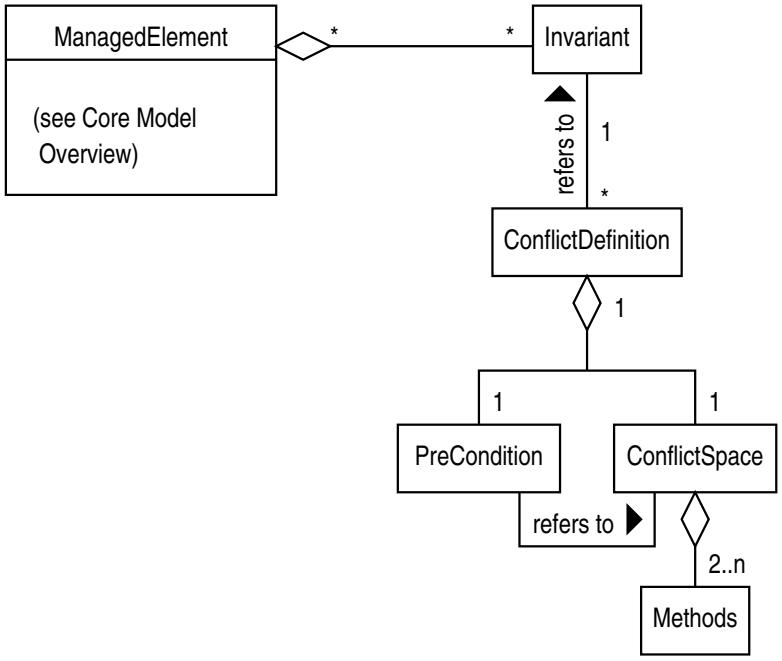

Abbildung 6.6: Einbindung der Konfliktdefinition in das CIM Core Model für die Oberklasse `ManagedElement`

Da CIM Beziehungen (bzw. Beziehungsklassen) selbst nicht als Managementobjekte modelliert, aber die Definition von Invarianten für Beziehungen, wie in Kapitel 4 gezeigt, ein wichtiges Element der Konfliktbehandlung ist, müssen die Beziehungsklassen von CIM ebenfalls (siehe Abschnitt 6.2.1) mit der Klasse `Invariant` in Beziehung gesetzt werden. Hier zeigt sich ein Nachteil des CIM Models: da Beziehungen nicht strikt in einer Beziehungshierarchie gebunden sind bzw. keine abstrakte Oberklasse für Beziehungen existiert, müßte für jede Beziehungsklasse des Core, Common und Extension Schema gefordert werden, dass an jede Klasse (nachträglich) das Model der Konfliktdefinition gebunden wird. Dies ist nicht besonders praktikabel, da eine große Anzahl von isolierten Beziehungsklassen modifiziert werden müssen.

Einbindung in das Meta Model

Qualifier werden im CIM Standard als Möglichkeit explizit genannt, CIM Modelle zu erweitern.

Jeder Qualifier hat die Form:

```
Qualifier Name :
            Type = defaultValue,
            Scope(AppliesToElement),
            Flavor(propagationRules);
```

Die einzelnen Felder eines Qualifiers bedeuten folgendes:

`Name` ist der Name des Qualifiers.

`Type` ist der Datentyp des Qualifiers mit einem vorgegebenen Wert.

`Scope` gibt an, auf welche Elementarten des Metamodels es sich bezieht.

`Flavor` gibt an, ob und wie diese Eigenschaft an Subklassen und Instanzen weitergereicht wird.

Beispielsweise wird die Eigenschaft einer Klasse, dass sie abstrakt ist, durch folgenden Qualifier ausgedrückt [CIM 2.2]:

```
Qualifier Abstract :
              boolean = false,
              Scope(class, association, indication),
              Flavor(DisableOverride, Restricted);
```

Der Qualifier kann auf Klassen, Assoziationen und Indikationen angewandt werden (`Scope`). Er wird nicht an Subklassen vererbt (`Restricted`) und kann nicht überschrieben werden.

OCL Konstrukte können als Qualifier spezifiziert werden. Jeweils für Invarianten, Vor– und Nachbedingungen wird ein Qualifier definiert:

```
Qualifier OCLInvariant :
            string= null,
            Scope(class, association),
            Flavor(ToSubclass);

Qualifier OCLPostCondition :
```

```
            string= null,
            Scope(method, class),
            Flavor(enableOverride, ToSubclass);

Qualifier OCLPreCondition :
            string= null,
            Scope(method, class),
            Flavor(enableOveride, ToSubclass);
```

Der Qualifier `OCLInvariant`, bezieht sich auf Klassen und Assoziationen und wird an Subklassen vererbt. Die Vor– und Nachbedingungen beziehen sich auf Methoden, werden an Subklassen vererbt und können überschrieben werden (falls die Methode in der Subklasse überladen wird). Alle drei Qualifier sind nicht vorbelegt.

Vergleich der beiden Einbindungsmöglichkeiten Die Einbindung der Bedingungen in das Core Model mittels aggregierter Klassen zieht die Modifikation zahlreicher Klassen nach sich. Die Einbindung über die Einführung neuer Qualifier ist hingegen einfach und wird von CIM gut unterstützt. Deshalb wird diese zweite Einbindung angenommen.

6.3.2 Abbildung der Beziehungshierarchie nach CIM

Abbildung 4.2 in Abschnitt 4.3.1 zeigt die generische Beziehungshierarchie, die nach CIM abgebildet werden muss.

Als erstes muss festgestellt werden, ob Klassen mit identischer Semantik, aber eventuell unterschiedlicher Bezeichnung existieren. Die abstrakten Klassen der Beziehungshierarchie `Association`, `Symmetric Association` und `Directed Association` existieren im CIM Core Model nicht.

Folgende Tabelle fasst die Abbildung der generischen Beziehungsklassen in CMI Klassen zusammen:

generische Beziehungsklasse		CIM Core Model Klasse
`Association`	→	existiert keine Klasse
`Symmetric Association`	→	existiert keine Klasse
`Directed Association`	→	existiert keine Klasse
`Containment`	→	`Component`
`Aggregation`	→	über Qualifier `Aggregation`
`Composition`	→	über Qualifier `Composition`
`Dependency`	→	`Dependency`
`FunctionalDependency`	→	existiert keine Klasse
`StateDependency`	→	existiert keine Klasse

Für die generischen Klassen `Aggregation` und `Composition` existieren keine analogen CIM Klassen. CIM spezifiziert allerdings Qualifier, die für beliebige Assoziationsklassen spezifiziert werden können. Die Aggregation und Komposition ist damit ausdrückbar. Die Klassen, die keine Entsprechung in CIM haben, können entsprechend hinzugefügt werden, wie es der CIM Standard vorschlägt.

6.4 Anwendung der Methodik in CIM

Dieser Abschnitt untersucht, wie die Methodik aus Kapitel 4 im Fall von CIM angewandt wird.

Dazu werden die einzelnen Schritte der Methodik untersucht:

Schritt 2 **Invarianten aus dem Modell extrahieren**
Die Definition von Invarianten geschieht analog zum angegebenen Vorgehen in Kapitel 4. Beispielsweise können die aufgestellten Invarianten des generischen Beziehungsmodells übernommen werden (lediglich die veränderte Namensgebung der Klassen muss beachtet werden).

Schritt 3 **Bezug der Aktionen zu den Invarianten spezifizieren**
CIM spezifiziert im Core und Common Model (fast) keine Methoden. Die Anreicherung der Klassen mit Methoden soll laut CIM anwendungsspezifisch erfolgen. Die Methoden, die bereits im generischen Modell definiert wurden, können somit direkt übernommen werden. Das Hinzufügen von Methoden zu Klassen wird in CIM Standard explizit als erlaubte Modifikation der Schemas betrachtet.

Schritt 4 **Kritische Aktionen identifizieren**
Für diesen Schritt kann die Methodik direkt aus Kapitel 4 übernommen werden.

Schritt 5 **Konfliktdefinition festlegen**
Dieser Schritt kann ebenfalls direkt übernommen werden.

Schritt 6 **Konfliktlösung**
In diesem Schritt erfolgt die Spezifikation von Vorbedingungen und die Abbildung auf Policies. Die Spezifikation der Vorbedingungen erfolgt analog dem der allgemeinen Methodik. Die Abbildung auf Policies muss ebenfalls im Informationsmodell (CIM) erfolgen. CIM spezifiziert das *Policy* Schema. Darin werden die wesentlichen Bestandteile von Policies als Managementobjekte modelliert. Abbildung 6.7 zeigt einen Teil des Modells. Eine Policy besteht aus einem Condition–Teil (`PolicyCondition`) und Policy–Aktionen (`PolicyAction`). Da die neu definierten Qualifier für OCL–Konstrukte (siehe Abschnitt 6.3.1) für beliebige Klassen anwendbar sind, können die OCL–Konstrukte in die Klasse `PolicyAction` übernommen werden.

Durch die Definition der zusätzlichen Qualifier lässt sich die Methodik für CIM ohne Modifikation anwenden.

6.5 Anwendung in der Praxis

Das Common Information Model wird beispielsweise in den Betriebssystemen Windows XP, Windows 2000 und Windows 98 der Firma Microsoft als Management–Informationsmodell eingesetzt. Dort wird es unter dem Namen Windows Management Instrumentation (WMI) verwendet. WMI ist Nachfolger von Web–Based Enterprise Management (WBEM). Es wird für die Bereiche des System– und Anwendungsmanagement eingesetzt.

Wie in diesem Kapitel gezeigt, kann die Konfliktdefinition, die Nachbedingungen der Methoden und Policies in CIM ohne großen Aufwand integriert werden. Ist ein policy–basiertes Managementsystem unter diesen Voraussetzungen implementiert und der Konfliktbehandlungsalgorithmus in den PDP integriert, so kann die Konfliktbehandlung wie in Kapitel 5 beschrieben eingesetzt werden. Eine Werkzeugunterstützung für den Einsatz eines policy–basierte Managementsystem existiert bis jetzt noch nicht.

Die Policy–Sprache kann (fast) frei gewählt werden. Notwendig ist ein dedizierter Action–Teil und die Unterstützung von Ereignissen. Alle gängigen Policy–Sprachen im Bereich des IT–Managements besitzen diese Eigenschaft (beispielsweise Ponder und ProPolis).

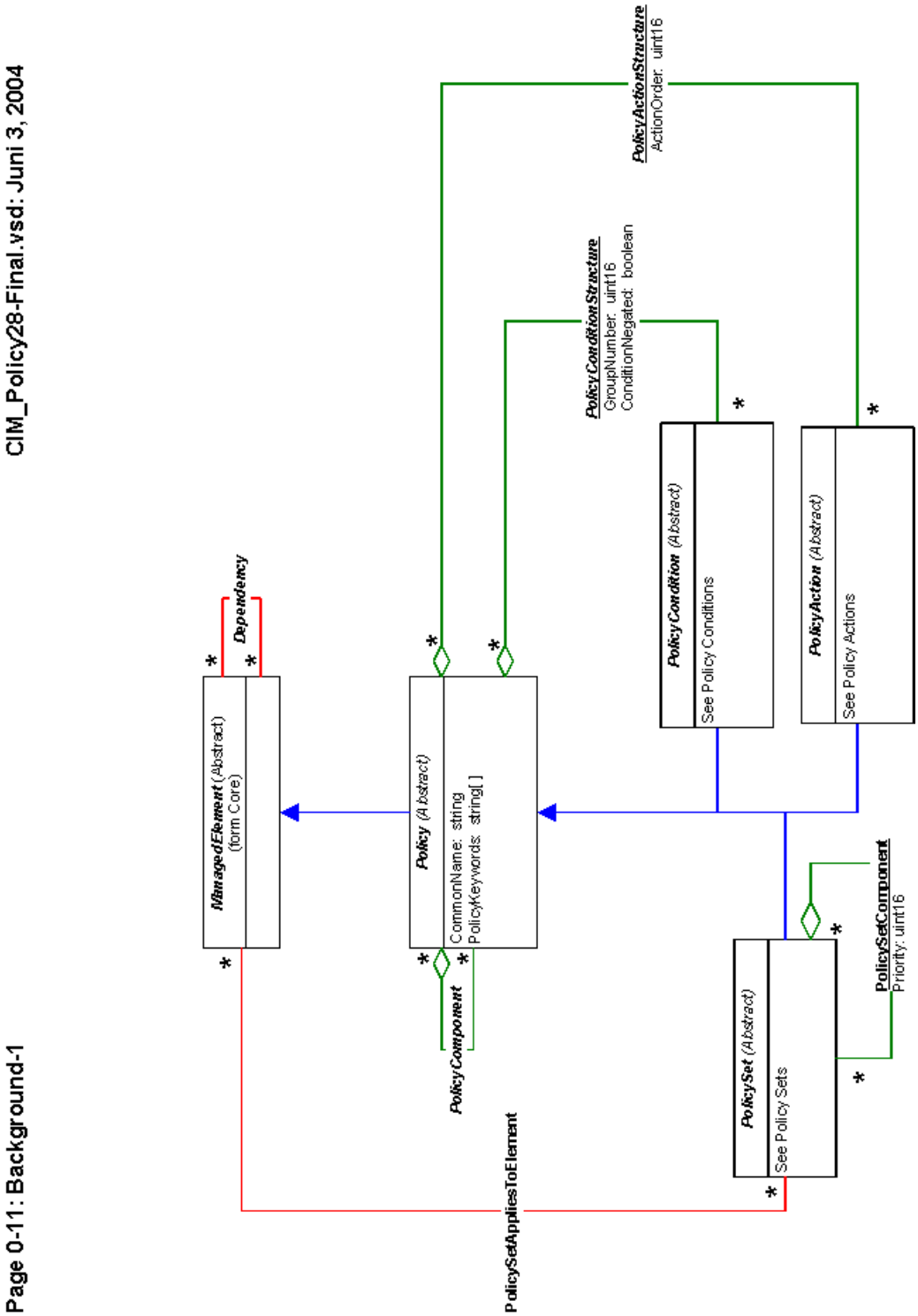

Abbildung 6.7: Ausschnitt aus dem CIM Policy Schema [CIM 2.2]

6.6 Zusammenfassung

Dieses Kapitel diente dazu, die Integration des generischen Beziehungsmodells und der Methodik aus Kapitel 4 exemplarisch in ein Management–Informationsmodell zu demonstrieren.

Die Einbindung des generischen Beziehungsmodells in das objektorientierte CIM kann ohne großen Aufwand erfolgen, da CIM bereits Beziehungen als wichtigen Teil eines Informationsmodells identifiziert hat. Die OCL–Konstrukte zur Spezifikation von Invarianten, Vor– und Nachbedingungen lassen sich elegant durch die Definition von sogenannten Qualifiern im CIM Meta Model erreichen. Die Methodik ist durch diese einfache Integration praktisch ohne Modifikationen direkt einsetzbar.

Existiert ein policy–basierte Managementsystem, dass sich auf CIM abstützt, so können die Lösungskonzepte aus Kapitel 4 und Kapitel 5 ohne großen Aufwand eingesetzt werden.

Kapitel 7

Zusammenfassung und Ausblick

Dieses Kapitel fasst die Arbeit zusammen, beschreibt die wichtigsten Ergebnisse und diskutiert offene und weiterführende Forschungsfragestellungen bezüglich Policy–Konflikte.

Der Bereich des policy–basierten Managements nimmt in der Industrie einen immer wichtigeren Stellenwert ein. Da die Spezifikation von Policies verteilt stattfindet, jedoch divergierende Ziele verfolgt und Policies parallel ausgeführt werden, ist die Gefahr, dass Policy–Konflikte auftreten gegeben. Die in der Literatur existierenden Arbeiten im Bereich der Policy–Konflikte besitzen drei wesentliche Defizite: die möglichen behandelbaren Konfliktarten sind durch die gewählten Konfliktbehandlungsansätze bereits im vorhinein stark eingeschränkt, die Konfliktbehandlungsansätze sind inhärent mit einer dedizierten Policy–Sprache verwoben und es fehlt eine Methodik, wie neuartige Konflikte in die Konfliktbehandlung eingebunden werden sollen. Die Folgen der genannten Defizite bestehen darin, dass auftretende Konflikte nicht erkannt werden, dass der Aufwand einen Konfliktbehandlungsansatz für eine andere Policy–Sprache zu etablieren hoch ist und dies zu einer langwierigen Anpassung der Konfliktbehandlung bei der Entdeckung neuer Konfliktarten führt.

Die Problemanalyse dieser Arbeit hat gezeigt, dass unter der Berücksichtigung von Managementmodellen eine Reihe von Konflikten existieren, die bis jetzt in der Literatur nicht behandelt wurden. Policies manipulieren im policy–basierten Management letztendlich Managementobjekte, die wiederum in Modellen eingebunden sind. Diese Arbeit begreift Managementmodelle als a priori Modelle, also eine Sammlung von einzuhaltenden Bedingungen. Damit konnte unter dieser Sichtweise die Existenz von neuartigen Konflikten gezeigt werden. Als wesentliche, bisher unbehandelte Konfliktart wurden Konflikte bei Beziehungen zwischen Managementobjekten nachgewiesen. Da Beziehungen in Managementmodellen eine essentielle Rolle spielen, diese jedoch bis jetzt nicht systematisch bezüglich

Policy–Konflikten analysiert wurden, ist ein generisches Beziehungsmodell entwickelt worden, das auf beliebige Management–Informationsmodelle abbildbar ist.

Da in der Literatur keine Vorgehensweise existiert, um methodisch Policy–Konflikte zu behandeln, ist in dieser Arbeit eine Methodik dazu entwickelt worden. Ziel der Methodik ist es, ausgehend von einem Modellaspekt über die Definition von Invarianten (mittels der Object Constraint Language) zu einer Konfliktdefinition zu gelangen, die sich auf Policy–Aktionen bezieht. Den letzten Schritt der Methodik bildet die Spezifikation von Vorbedingungen für Policy–Aktionen, deren Einhaltung einen Konflikt verhindert. Die allgemeine Anwendbarkeit der Methodik wurde anhand des Beziehungsmodells für die Beziehungen der funktionalen Abhängigkeit sowie Enthaltenseinsrelationen und für einen Vertreter von dynamischen Modellen, den endlichen Automaten, demonstriert. Für jede dieser Anwendungsfälle konnten Konfliktdefinitionen und Vorbedingungen gefunden werden. Dabei sind weder Methodik noch die Konfliktdefinitionen von einer dedizierten Policy–Sprache abhängig, oder die behandelbaren Konfliktarten beschränkt. Somit ist eine breite Anwendbarkeit der erarbeiteten Konzepte gewährleistet.

Der eigentliche Konfliktbehandlungsalgorithmus, also das Erkennen und Lösen von Policy–Konflikten, kann präventiv oder reaktive geschehen und wurde (ebenfalls) nicht allgemein in der Literatur behandelt. Die reaktive Konfliktbehandlung beschäftigt sich mit Policy–Konflikten zum Zeitpunkt der Ausführung von Policies. Es wurde ein effizienter, in weiten Teilen parallelisierbarer Algorithmus entwickelt, der durch Teilmengenbildung die Anzahl der zu betrachteten Policies schnell reduziert. Es wurden Strategien zur Konflikterkennung und Konfliktlösung entwickelt, die individuell auf die betrachteten Konfliktarten abgestimmt sind. Die präventive Konfliktbehandlung dagegen greift bereits zum Zeitpunkt der Policy–Spezifikation, sie kann den Algorithmus der reaktiven Konfliktbehandlung im Wesentlichen übernehmen und ist in der Lage, potentielle Konflikte zu erkennen. Schließlich mündete die Analyse der Strategien und der Methodik in eine neuartige Klassifizierung von Policy–Konflikten. Die Konfliktbehandlung prüft bei einer Verletzung von Invarianten auf das Auftreten einen Konflikts. Ist kein Konflikt die Ursache der Verletzung so liegt kein Konflikt vor, aber die Verletzung kann als Fehler behandelt werden. So ist der Einsatz des Konfliktbehandlungsalgorithmus nicht nur auf Konflikte beschränkt sondern kann auch einen Beitrag zum Fehlermanagement liefern.

Am Beispiel des Common Information Model wurde gezeigt, dass sich das generische Beziehungsmodell und die Konfliktdefinitionen in ein existierendes Management–Informationsmodell integrieren lassen und die Methodik dort ohne Modifikation anwendbar ist.

Weiterführende Forschungsfragestellungen

Konflikte sind kein alleiniges Phänomen des policy–basierten Management, sondern jedes Managementsystem, welches verteilt über eine Infrastruktur abläuft, besitzt prinzipiell diese Problematik. Management–Anweisungen werden in Management–Architekturen verteilt ausgeführt. Da die zu managenden Ressourcen, wie in der vorliegenden Arbeit gezeigt, voneinander abhängen, können potentiell Konflikte in jedem Managementsystem auftreten, die diesem Umstand nicht direkt Rechnung tragen. So unterliegen beispielsweise verteilt ablaufende Management–Skripte prinzipiell der gleichen Problematik wie das policy–basierte Management. Es wäre zu untersuchen inwieweit die Ansätze dieser Arbeit auf andere Managementparadigmen anwendbar sind.

Eine weitere Fragestellung betrifft die Definition und Formalisierung von Zielen für das IT–Management. Die Analyse des State of the Art hat gezeigt, dass im Bereich des Goal–Oriented Requirements Engineering bereits interessante Ansätze vorhanden sind. Es existieren jedoch noch keine generischen Zielebeschreibungen oder gar Zielmodelle für den gesamten Bereich des Managements. Diese Zielmodelle würden über den gesamten Management–Komplex, angefangen mit dem Business Management bis hin zum Element Management, die Ziele erfassen und zueinander in Beziehung setzen. Würden diese Zielmodelle existieren, so wäre eine top–down Ableitung von Policies möglich. Damit könnte auch das Problem der Policy–Verfeinerung entscheidend vorangetrieben werden, da man die Korrektheit einer Verfeinerung durch die assoziierten Ziele besser analysieren könnte. Weiterhin könnten Ziele zusammen mit Konfliktdefinitionen in ein Modell integriert werden. Das hätte den Vorteil, dass die Identifikation von divergierenden Zielen teilweise automatisiert werden könnte.

Die in dieser Arbeit entwickelte Konfliktbehandlung geht von einem zentralen Policy Decision Point (PDP) aus. Um die Skalierbarkeit von policy–basierten Management–Systemen zu steigern, existieren Architekturen, die eine Verteilung der PDPs vorsehen. Es wäre zu untersuchen, wie die Konfliktbehandlung bei verteilten PDPs angepasst werden müsste. Insbesondere müsste untersucht werden, ob und wie die benötigten Informationen zur Konfliktbehandlung verteilbar sind. Ebenso wäre zu untersuchen wie verteilte PDPs miteinander kooperieren.

In dieser Arbeit wurde als Beispiel eines dynamisches Modells endliche Automaten untersucht. Es existieren weitere für das IT–Management interessante dynamische Modellarten, die noch zu analysieren wären. Beispielsweise spielt die Prozessorientierung eine immer größere Rolle sowohl bei den Anwendungen als auch für das Management selbst. Die Anwendung der Methodik für Prozessmodelle wäre deshalb ein wichtiger Untersuchungsgegenstand.

Abbildungsverzeichnis

Literaturverzeichnis

[BLR 03] BANDARA, AROSHA K., EMIL C. LUPU und ALESSANDRA RUSSO: *Using Event Calculus to Formalise Policy Specification and Analysis*. In: *Proceedings of HPOVUA 2003*, 2003.

[CIM 2.2] *Common Information Model (CIM) Specification Version 2.2*. Specification, Juni 1999.

[CLN 00] CHOMICKI, J., J. LOBO und S. NAQVI: *A Logic Programming Approach to Conflict Resolution in Policy Management*. In: *7th International Conference on Principles of Knowledge Representation and Reasoning (KR'2000)*, Seiten 121–132, Breckenridge, Colorado, Morgan Kaufman, 2000. .

[CLN 03] CHOMICKI, J., J. LOBO und S. NAQVI: *Conflict Resolution Using Logic Programming*. Transaction on Knowledge and Data Engineering, 15(1):244–249, 2003.

[CORBA-302] *The Common Object Request Broker: Architecture and Specification*. OMG Specification Revision 3.0.2, Object Management Group, Dezember 2002.

[DaKe 04] DANCIU, V. und B. KEMPTER: *From Processes to Policies – Concepts for Large Scale Policy Generation*. In: *Managing Next Generation Convergence Networks and Services: Proceedings of the 2004 IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Band 2004, Seoul, Korea, April 2004. IEEE/IFIP.

[Dami 02] DAMIANOU, N. C.: *A Policy Framework for Management of Distributed Systems*. Doktorarbeit, Imperial College of Science, Technology and Medicine, University of London, Department of Computing, Februar 2002.

[Danc 03] DANCIU, V.: *Entwicklung einer policy–basierten Managementanwendung für das prozessorientierte Abrechnungsmanagement*. Diplomarbeit, Ludwig–Maximilians–Universität München, Januar 2003.

[DDGH 02] DIAZ, G., S. DUFLOS, V. GAY und E. HORLAIT: *A Comparative Study of Policy Specification Languages for Secure Distributed Applications*. In: FERIDUN, M. und P. KROPF (Herausgeber): *Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 2002)*, Lecture Notes in Computer Science (LNCS), Montreal, Canada, Oktober 2002. IFIP/IEEE, Springer.

[DDLS 00] DAMIANOU, N., N. DULAY, E. LUPU und M. SLOMAN: *Ponder: A language for Specifying Security and Management Policies for Distributed Systems. The Language Specification Version 2.3*. Imperial College Research Report DoC 2000/1, Imperial College of Science, Technology and Medicine, University of London, Department of Computing, Oktober 2000.

[Dira 02] DUNLOP, N., J. INDULSKA und K. RAYMOND: *Dynamic Conflict Detection in Policy-Based Management Systems*. In: *6th International Enterprise Distributed Object Computing Conference (EDOC '02)*. IEEE Publishing, 2002.

[DLSD 01] DULAY, N., E. LUPU, M. SLOMAN und N. DAMIANOU: *A Policy Deployment Model for the Ponder Language*. In: PAVLOU, G., N. ANEROUSIS und A. LIOTTA (Herausgeber): *Proceedings of the 7th IEEE/IFIP International Symposium on Integrated Network Management (IM 2001)*, Seattle, Washington, USA, Mai 2001. IFIP/IEEE, IEEE Publishing.

[DSP 0108b] DMTF POLICY WORKING GROUP: *CIM Core Policy Model White Paper*. White Paper, Juni 2003.

[EnKe 01a] ENSEL, C. und A. KELLER: *Managing Application Service Dependencies with XML and the Resource Description Frame work*. In: PAVLOU, G., N. ANEROUSIS und A. LIOTTA (Herausgeber): *Proceedings of the 7th International IFIP/IEEE Symposium on Integrated Management (IM 2001)*, Seattle, Washington, USA, Mai 2001. IEEE Publishing.

[Ense 02] ENSEL, C.: *Abhängigkeitsmodellierung im IT Management: Erstellung eines neuen, auf Neuronalen Netzen basierenden Ansatzes*. Dissertation, Ludwig–Maximilians–Universität München, Juni 2002.

[FrKo 98] FROLUND, SVEND und JARI KOISTINEN: *QML: A Language for Quality of Service Specification*. Technischer Bericht HPL-98-10, Hewlett-Packard Laboratories, Palo Alto, 1998.

[GB 921] *enhanced Telecom Operations Map (eTOM), The Business Process Framework For The Information and Communications Services Industry*. Technischer Bericht GB 921 Approved Version 3.0, TeleManagement Forum, Juni 2002.

[GNY+ 01] GU, NAHRSTEDT, YUAN, WICHADAKUL und XU: *An XML-based Quality of Service Enabling Language for the Web*. Technischer Bericht, Department of Computer Science University of Illinois at Urbana-Champaign, Urbana, UIUCDCS-R-2001-2212, April 2001., 2001.

[Grus 99] GRUSCHKE, B.: *Entwurf eines Eventkorrelators mit Abhängigkeitsgraphen*. Dissertation, Ludwig–Maximilians–Universität München, November 1999.

[HAN 99] HEGERING, H.-G., S. ABECK und B. NEUMAIR: *Integrated Management of Networked Systems — Concepts, Architectures and their Operational Application*. Morgan Kaufmann Publishers, ISBN 1–55860–571–1, Januar 1999. 651 p.

[HAN 99a] HEGERING, H.-G., S. ABECK und B. NEUMAIR: *Integriertes Management vernetzter Systeme — Konzepte, Architekturen und deren betrieblicher Einsatz*. dpunkt–Verlag, ISBN 3–932588–16–9, Januar 1999. 607 S.

[Heil 00] HEILBRONNER, S.: *Konzeption einer Architektur für das integrierte Management der Ressourcennutzung nomadischer Systeme in Datennetzen*. Dissertation, Ludwig–Maximilians–Universität München, Januar 2000.

[Hilp 71] HILPINEN, R. (Herausgeber): *Deontic Logic: Systematic Readings*. D. Reidel Publishing Company, 1971.

[ISO 10164-2] *Information Technology – Open Systems Interconnection – Systems Management – Part 2: State Management Function*. IS

10164-2, International Organization for Standardization and International Electrotechnical Committee, 1993.

[ISO 10164-x] *Information Technology – Open Systems Interconnection – Systems Management – Management Functions*. IS 10164-x, International Organization for Standardization and International Electrotechnical Committee, 1991-97.

[ISO 10165-1] *Information Technology – Open Systems Interconnection – Structure of Management Information – Part 1: Management Information Model*. IS 10165-1, International Organization for Standardization and International Electrotechnical Committee, 1993.

[ISO 10165-7] *Information Technology – Open Systems Interconnection – Structure of Management Information – Part 7: General Relationship Model*. IS 10165-7, International Organization for Standardization and International Electrotechnical Committee, 1997.

[KKK 96] KOCH, T., C. KRELL und B KRÄMER: *Policy Definition Language for Automated Management of Distributed Systems*. In: *IEEE Workshop on Systems Management*. IEEE, 1996.

[Koch 96] KOCHER, P. C.: *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*. In: KOBLITZ, N. (Herausgeber): *Advances in Cryptology - CRYPTO '96*, Band 1109 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 104–113. August, Springer, August 1996.

[KoSe 86] KOWALSKI, R.A. und M.J. SERGOT: *A logic–based calculus of events*. New Generation Computing, 4:67+, 1986.

[LAMS 01] VAN LAMSWEERDE, A.: *Goal–Oriented Requirements Engineering: A Guided Tour*. In: *5th IEEE International Symposium on Requirements Engineering*, Seiten 249–263, 2001.

[LBN 99] LOBO, JORGE, RANDEEP BHATIA und SHAMIM A. NAQVI: *A Policy Description Language*. In: *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, Seiten 291–298, 1999.

[LiPa 99] LILIUS, JOHAN und IVAN PORRES PALTOR: *The Semantics of UML State Machines*. Technischer Bericht TUKS Technical Report No 273, Turku Centre for Computer Science, 1999.

[LuSl 99] LUPU, EMIL C. und MORRIS SLOMAN: *Conflicts in Policy-Based Distributed Systems Management*. IEEE Transactions on Software Engineering, 25(6):852–869, November 1999.

[MoSl 93] MOFFETT, JONATHAN D. und MORRIS S. SLOMAN: *Policy Conflict Analysis in Distributed System Management*. Journal of Organizational Computing, 1993.

[OMG 02-08-04] *Notification Service Specification*. OMG Specification Revision 1.0.1, Object Management Group, August 2002.

[OMG 98-12-09] *CORBA Services complete book*. OMG Specification formal/98-12-09, Object Management Group, Dezember 1998.

[Radi 02d] RADISIC, I.: *Using Policy–Based Concepts to Provide Service Oriented Accounting Management*. In: STADLER, R. und M. ULEMA (Herausgeber): *Proceedings of the 8th International IFIP/IEEE Network Operations and Management Symposium (NOMS 2002)*, Seiten 313–326, Florence, Italy, April 2002. IFIP/IEEE, IEEE Publishing.

[Radi 03] RADISIC, I.: *Ein prozessorientierter, policy–basierter Ansatz für ein integriertes, dienstorientiertes Abrechnungsmanagement*. Dissertation, Ludwig–Maximilians–Universität München, Februar 2003.

[RFC 2748] DURHAM, D., ED., J. BOYLE, R. COHEN, S. HERZOG, R. RAJAN und A. SASTRY: *RFC 2748: The COPS (Common Open Policy Service) Protocol*. RFC, IETF, Januar 2000.

[RFC 3060] MOORE, B., E. ELLESSON, J. STRASSNER und A. WESTERINEN: *RFC 3060: Policy Core Information Model – Version 1 Specification*. RFC, IETF, Februar 2001.

[RFC 3198] WESTERINEN, A., J. SCHNIZLEIN, J. STRASSNER, M. SCHERLING, B. QUINN, S. HERZOG, A. HUYNH, M. CARLSON, J. PERRY und S. WALDBUSSER: *RFC 3198: Terminology for Policy-Based Management*. RFC, IETF, November 2001.

[Sail 02] SAILER, M.: *Klassifizierung und Bewertung von VPN–Lösungen für die Neuausrichtung der europaweiten Extranetstrategie der BMW AG*. Diplomarbeit, Technische Universität München, August 2002.

[SlTw 94] SLOMAN, MORRIS S. und KEVIN TWIDLE: *Domains: A Framework for Structuring Management Policy*, Kapitel 16. Addison–Wesley, 1994.

[Stal 97] STALLINGS, W.: *Operating Systems: Internals and Design Principals*. Prentice Hall, 3 Auflage, 1997.

[uml 03] *OMG Unified Modeling Language Specification, Version 1.5*. Technischer Bericht, Object Management Group, März 2003.

[Verm 00] VERMA, DINESH C.: *Policy-Based Networking*. Technology Series. New Riders Publishing, Indianapolis, Indiana, 2000.

[vLD 98] VAN LAMSWEERDE, A., E. LETIER und R. DARIMONT: *Managing Conflicts in Goal-Driven Requirements Engineering*. IEEE Transaction on Software Engineering, Special Issue on Managing Inconsistency in Software Development, 24(11):908–926, 1998.

[Wies 95] WIES, R.: *Policies in Integrated Network and Systems Management: Methodologies for the Definition, Transformation, and Application of Management Policies*. Dissertation, Ludwig–Maximilians–Universität München, Juni 1995.

[WiMe 93] WIERINGA, R.J. und J.-J.CH MEYER: *Applications of deontic logic in computer science: A concise overview*, Seiten 17–40. Deontic Logic in Computer Science: Normative System Specification. Wieringa, R.J. and Meyer, J.-J.Ch, 1993.

Lebenslauf

Name		Bernhard Kempter
Geburtsdatum		10.11.1969
Geburtsort		München
Ausbildung	09.82 - 08.86	Staatliche Realschule Vaterstetten Abschluss: mittlere Reife
	09.86 - 02.90	Lehre zum Energiegeräteelektroniker bei den Stadtwerken München
	03.90 - 08.90	Energiegeräteelektroniker bei den Stadtwerken München
	09.90 - 07.92	Berufsoberschule in München Abschluss: fachgebundene Hochschulreife
Zivildienst	07.92 - 09.93	Fahr- und allgemeiner Sozialdienst beim Roten Kreuz in Ebersberg
Studium	10.93 - 06.99	Informatikstudium (Nebenfach Elektrotechnik) an der Technischen Universität München Abschluss: Diplom-Informatiker Univ.
Beruf	07.99 - 02.05	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Kommunikationssysteme und Systemprogrammierung des Instituts für Informatik der Ludwig-Maximilians-Universität München