INSTITUT FÜR INFORMATIK

DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Entwicklung einer policy-basierten Managementanwendung für das prozeßorientierte Abrechnungsmanagement

Vitalian Danciu

Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Bernhard Kempter Igor Radisic

Abgabetermin: 17. Januar 2003

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Entwicklung einer policy-basierten Managementanwendung für das prozeßorientierte Abrechnungsmanagement

Vitalian Danciu

Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Bernhard Kempter Igor Radisic

Abgabetermin: 17. Januar 2003

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 17. Januar 2003

. . . .
(Unterschrift des Kandidaten)

Zusammenfassung

Gegenwärtige Abrechnungssysteme leiden unter der Heterogenität der Schnittstellen ihrer Komponenten und der Benutzung proprietärer Datenformate zum Informationsaustausch. Die gegenwärtig zur Herstellung der Interoperabilität der Abrechnungskomponenten eingesetzten, auf Gateways basierten Lösungen führen hohe Kosten zur Wartung und Anpassung mit sich und berücksichtigen Managementaspekte nicht ausreichend oder überhaupt nicht. Aus diesen Gründen halten sie den Anforderungen des heutigen dienstorientierten, von Outsourcing-Szenarien bestimmten Umfeld nicht stand.

Im Rahmen der vorliegenden Diplomarbeit wird eine Integrationsschicht zur Vereinheitlichung der Nutzungs- und Managementschnittstellen der Komponenten der Abrechnung beschrieben. Diese ermöglicht eine einheitliche, von der Implementierung des Abrechnungssystems unabhängige, prozeßorientierte Managementsicht, wobei gleichzeitig die durch Gateways implementierte Funktionalität repliziert werden kann.

Die Verwaltung einzelner Komponenten und Gateways wird durch ein einheitliches, mittels policy-basierter Techniken realisiertes Management ersetzt. Zu diesem Zweck wird eine Sprache zur Definition von Policies entworfen und in den Entwurf einer Managementanwendung integriert. Eine prototypische Implementierung dient als Beleg für die praktische Realisierbarkeit des Ansatzes.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

1.1 Einleitung

Die meisten Unternehmen und Organisationen verlassen sich heute auf immer komplexer werdende IT-Infrastrukturen. Die Kosten — sowohl zeitliche als auch finanzielle — von Implementierung und Betrieb solcher Infrastrukturen müssen jedoch kalkulierbar bleiben, um die hauptsächliche Tätigkeit des Unternehmens oder der Organisation (das sogenannte Kerngeschäft) nicht zu beeinträchtigen. Erschwerend kommt die schnelle Entwicklung und Veränderung der IT-Anforderungen, sowie der aktuellen Technologien, hinzu. Oft ist es außerdem der Fall, daß ein Unternehmen nicht über die notwendige Kompetenz oder die erforderlichen Ressourcen verfügt, um selbst seine IT-Infrastruktur zu betreiben und zu pflegen.

Outsourcing Als Folge bedienen sich immer mehr Unternehmen externer Dienstleister, die gegen Entgelt eine IT-Infrastruktur zur Verfügung stellen, die den Bedürfnissen des Unternehmens entspricht (outsourcing). Der Dienstleister ist für den Betrieb, sowie für Wartung und Anpassung dieser Infrastruktur zuständig. Zwischen Dienstleister (Anbieter) und Unternehmen oder Organisation (Kunde) kommt ein Vertrag zustande, der einerseits die Merkmale der Dienste beschreibt, die zur Verfügung gestellt werden, andererseits die Tarife, die für die jeweiligen Dienste gelten [GHK+ 01b].

In zeitlichen Abständen läßt der Dienstleister dem Kunden eine Rechnung über die Nutzung der Dienste zukommen. In dieser sind die erbrachten Leistungen summiert aufgeführt, wobei einzelne Posten mit der Nutzungsmenge dem vertraglich festgelegten Preis versehen wurden. Die Erstellung einer solchen Rechnung ist die Aufgabe des *Abrechnungssystems* (accounting system) des Anbieters.

Die angebotenen Dienstleistungen variieren in ihrer Art. Es kann sich dabei um die Vernetzung unterschiedlicher Standorte zur Bereitstellung eines *Corporate Network* handeln, um Dienste wie Email oder die Erstellung von Backups, oder um den Betrieb von Application Servern oder Datenbanksystemen. Die Vereinbarungen bezüglich der Dienstleistung werden in sogenannten *Service Level Agreements* (SLA) festgehalten. Je nach Dienst, kann ein SLA Angaben über Dienstgüte (z. B. Garantien bezüglich der Datenübertragungsrate), Verfügbarkeit kritischer Systeme, oder anderen Kriterien des Dienstes enthalten. Der Anbieter verpflichtet sich, den Dienst entsprechend des SLAs zur Verfügung zu stellen; er übernimmt Verantwortung im gegenteiligen Fall.

Die Vergütung der Dienstnutzung durch den Kunden geschieht entsprechend den zwischen Anbieter und Kunden ausgehandelten Tarifen. Dabei obliegt die Messung der Dienstnutzung und die Erstellung der Rechnung naturgemäß der Verantwortung des Anbieters. Werden Veränderungen der SLAs und Tarife vereinbart, oder ändert sich die Anzahl und Art der Dienste, folgen daraus Konsequenzen für den Abrechnungsvorgang. Diese Konsequenzen haben meist finanzielle Aspekte für den Anbieter, die jedoch durch die Vergütung der Dienstnutzung nicht explizit gedeckt werden.

Probleme des Abrechnungsvorgangs Die durch eine Änderung des Abrechungsvorgangs verursachten Kosten können unverhältnismäßig hoch werden. Die Gründe dafür liegen einerseits in der Fülle von möglichen Tarif- und Geschäftsmodellen, auf denen die vertraglichen Vereinbarungen zwischen Anbieter und Kunde basieren können. Andererseits benutzen heutige kommerzielle Abrechnungssysteme und -komponenten proprietäre Schnittstellen und herstellerspezifische Datenformate. Erschwerend kommt hinzu, daß die Schnittstellen und Datenformate nicht öffentlich dokumentiert sind; infolgedessen existieren keine Middleware-Produkte zur Verbindung von Abrechnungskomponenten.

Die Interoperabilität zwischen Produkten verschiedener Hersteller, wie auch die Migration des Abrechnungsvorgangs von einem Abrechnungssystem auf das andere, sind daher nur unter großem Aufwand zu realisieren. Veränderungen der Kundenanforderungen können allerdings die Umsetzung eben dieser Maßnahmen erfordern. Um Interoperabilität der Abrechnungskomponenten zu realisieren werden Gateways eingesetzt, die die verschiedenen Schnittstellen der Komponenten aufeinander abbilden, indem sie zwischen den proprietären Protokollen übersetzen und die verschiedenen Datenformate bei Bedarf konvertieren. Wird eine der Abrechnungskomponenten ausgetauscht, müssen alle betroffenen Gateways angepaßt werden — eine zeit- und kostspielige Prozedur.

Managementaspekte Um ein Abrechnungssystem in eine Managementinfrastruktur integerieren zu können, müssen seine Komponenten über entsprechende Schnittstellen verfügen. Diese müssen dem Managementsystem angepaßt sein (z. B. durch den Einsatz üblicher Managementprotokolle, wie etwa SNMP [CFSD 90]) und für das Management von Abrechnungskomponenten spezifische Funktionen zur Verfügung stellen. Es existieren zwar Ansätze zur Standardisierung von Schnittstellen und Architektur. Diese Standards werden allerdings nur unvollständig oder überhaupt nicht in der Implementierung derzeitiger Abrechnungskomponenten berücksichtigt.

1.2 Aufgabenstellung

In der vorliegenden Arbeit wird eine Managementanwendung entwickelt, die einen Rahmen für das unifizierte Management der Komponenten von Abrechnungssystemen darstellt. Durch Benutzung eines prozeßorientierten Ansatzes soll dabei eine möglichst genaue Entsprechung der Aktionen in den Geschäftsprozessen des Betreibers und den konkreten Managementaktionen des Abrechnungsmanagement erlaubt werden. Der Einsatz policy-basierten Managements wird benutzt, um die Wissensbasis der Administratoren von Abrechnungssystemen in einer deklarativen Notation erfassen zu können; zudem wird gezeigt, daß die policy-basierten Techniken eine besonders vorteilhafte Abbildung auf den prozeßorientierten Ansatz erlauben.

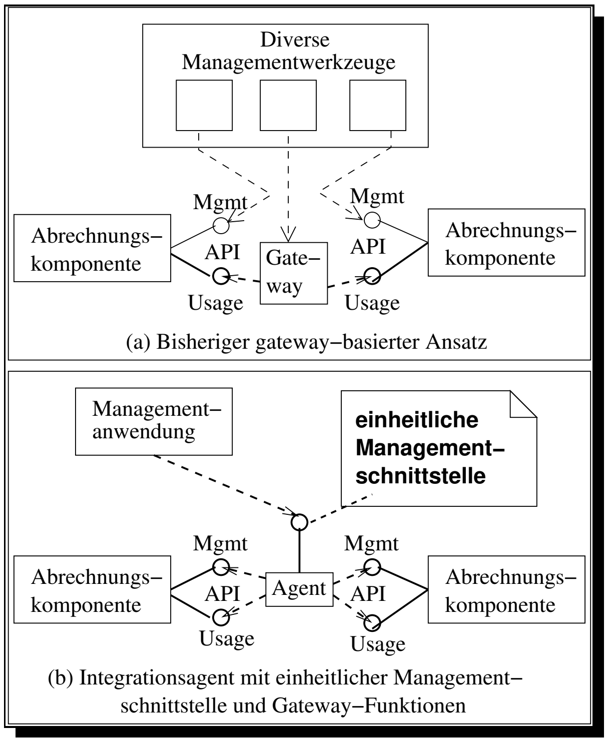

Abbildung 1.1: Unifizierung der Management-schnittstelle

Die (herstellerspezifische) Schnittstelle einer Abrechnungskomponente läßt sich logisch in eine Benutzerschnittstelle (*usage API*) und eine Managementschnittstelle (*management API*) gliedern. Der bisher in der Praxis vorherrschende, gateway-basierte Interoperabilitätsansatz berücksichtigt nur die *usage API* der Abrechnungskomponenten (vgl. Abbildung 1.1a). Die in dieser Arbeit angestrebte einheitliche Managementsicht wird durch eine von Proxy-Agenten gebildete Integrationsschicht erreicht, die der Managementanwendung eine einheitliche Schnittstelle anbietet. Diese kann auf die proprietären Management-APIs der Abrechnungskomponenten abgebildet werden. Sie wird aus den Merkmalen der Teilprozesse der Abrechnung abgeleitet, wobei existierende Standardisierungsdokumente berücksichtigt werden.

Die policy-basierte Steuerung des Abrechnungsmanagements erfordert die Spezifikation einer Sprache, mit Hilfe derer Regeln für das Ausführen von Managementaktionen definiert werden können. In dieser Arbeit wird eine solche Sprache entwickelt und in der Managementanwendung implementiert. Dabei werden auch Aspekte der Erweiterbarkeit und der Anwendung der Sprache in anderen Managementbereichen beachtet. Durch den Einsatz von Domänen, Rollen und Gruppen, sowie durch eine Modularität der Sprache soll eine flexible Konfiguration der Managementanwendung mittels Policies gewährleistet werden. Zusätzlich sollen Mechanismen zur Verfügung gestellt werden, um infolge von Managemententscheidungen Manipulationen der Policies selbst durchführen zu können.

Der Entwurf und die Implementierung der Managementanwendung erfolgt unter Benutzung von CORBA als Verteilungsplattform. Die Kommunikation zwischen der Managementanwendung und den Agenten der Integrationsschicht, sowie die zwischen einzelnen Agenten wird über generische Schnittstellen abgewickelt, die mittels einer exemplarischen Analyse von Teilprozessen der Abrechnung entwickelt werden. Eine Anwendung zum Management heterogener, proprietärer Komponenten in einem Umfeld mit hoher Änderungsdynamik ist auf einfache Veränderbarkeit und Erweiterbarkeit angewiesen. Diese Aspekte werden bei der Entwicklung der Managementanwendung besonders berücksichtigt.

1.3 Vorgehensweise

Anforderungskatalog Die Grundlage für die Modellierung der Managementanwendung bildet ein *Anforderungskatalog*. Ein Teil der darin enthaltenen Anforderungen ergeben sich aus der exemplarischen Analyse der Teilprozesse der Abrechnung. Durch eine generelle Betrachtung der Teilprozesse ergeben sich Entitäten, Ereignisse und Aktionen, die den Abrechnungsvorgang mitbestimmen.

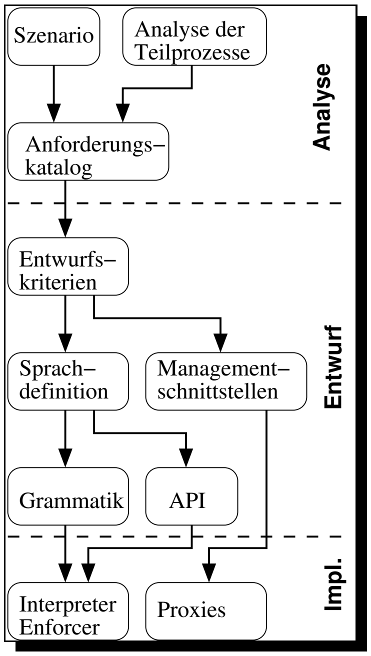

Abbildung 1.2: Die Vorgehensweise der Arbeit

Ein Praxisbezug des Katalogs wird durch die Einführung und Betrachtung eines Szenarios erreicht. Mit Hilfe des Szenarios werden die gegenwärtigen Probleme des Abrechnungsmanagements aufgezeigt, indem die durch Veränderungen entstehenden Komplikationen anhand einer realitätsnah konstruierten Situation analysiert werden.

Weitere Anforderungen ergeben sich aus der Literatur, indem relevante Konzepte auf ihre Anwendbarkeit untersucht werden. Die Akzeptanz einer Technologie oder eines Konzepts steht oft in direktem Zusammenhang mit ihrer Komplexität und Implementierbarkeit. Deshalb werden die in Frage kommenden Arbeiten auch anhand dieser Kriterien bewertet.

Entwurfskriterien Der Anforderungskatalog enthält abstrakte Anforderungen an das Managementsystem als Ganzes. Aus diesen werden konkrete Kriterien für die Definition einer Sprache erstellt, die es erlauben soll, den Abrechnungsvorgang regelbasiert zu steuern. Aus den Anforderungen werden außerdem Kriterien für eine generische *Management-Schnittstelle* der Entitäten in der Abrechnung abgeleitet. Solch eine Schnittstelle ermöglicht eine einheitliche Management-Sicht auf die Abrechnungskomponenten.

Policy Definition Language Anhand der erarbeiteten Kriterien wird eine generische Sprache zur policy-basierten Steuerung definiert. Komplementär zur Definition der Grammatik der Sprache wird in Form einer Programmierschnittstelle (API) ein Kontext spezifiziert, der explizit auf die Anforderungen des Abrechnungsmanagements zugeschnitten ist.

Implementierung Die anhand dieser Überlegungen entstehende Managementanwendung gliedert sich in die folgenden Teile:

Interpreter Die Umsetzung der (generischen) Policy Definition Language wird durch einen Interpreter realisiert. Die Spezialisierung auf den Abrechnungsvorgang wird durch das Bereitstellen einer *Core API* verwirklicht.

Repository Die Verwaltung von Policies wird von dem *Policy Repository* realisiert.

Enforcer Die Ausführung der in einer Policy spezifizierten Aktionen obliegt dem *Enforcer*. Dieser greift einerseits auf die Core API zu, andererseits auf die Schnittstelle der Proxy-Agenten.

Integrationsagenten Die Integrationsagenten repräsentieren eine am Abrechnungsvorgang beteiligte Entität und bieten der Managementanwendung eine konsistente Schnittstelle.

1.4 Gliederung der Arbeit

Die Arbeit gliedert sich in acht Kapitel. In diesem ersten einführenden Kapitel wird die gegenwärtige Problematik um den Abrechnungsvorgang und das Abrechnungsmanagement grob umrissen. Danach werden Aufgabenstellung und Vorgehensweise der Arbeit festgehalten.

Das zweite Kapitel stellt ein Szenario vor, das die Motivation für die Arbeit beispielhaft untermauert. Aus dem Szenario wird ein Katalog von Anforderungen bezüglich der Entwicklung der Managementanwendung abgeleitet.

Im dritten Kapitel werden die Publikationen und Standardisierungswerke, die das Thema betreffen, genannt und hinsichtlich ihrer Relevanz bezüglich dieser Arbeit bewertet. Weiterhin wird die Arbeit gegenüber existierenden Vorarbeiten und Anwendungen positioniert, indem die zur Bewältigung der Aufgaben übernommenen Konzepte identifiziert werden.

Das vierte Kapitel befaßt sich mit wichtigen Aspekten policy-basierter Techniken, sowie der prozeßorientierten Sicht auf das Abrechnungsmanagement und legt die Randbedingungen für den Entwurf von *Policy Definition Language* und Managementanwendung fest. Insbesondere werden Konsequenzen der prozeßorientierten Sicht betrachtet und die erforderliche Ausdrucksmächtigkeit der zu spezifizierenden Sprache untersucht.

Im fünften Kapitel wird die *Policy Definition Language* spezifiziert. Nach der Festlegung der in die Sprache eingehenden Konstrukte wird die EBNF der Policy-Sprache angegeben. Dabei wird die Semantik der Sprachelemente unter Angabe von Beispielen erläutert.

Das sechste Kapitel enthält den Entwurf der Managementanwendung. Dies umfaßt die Architektur des Managementsystems, die Festlegung eines Datenmodells für Policies, sowie den Entwurf des in drei Komponenten gegliederten Laufzeitsystems. Außerdem werden dynamische Aspekte der Managementanwendung, sowie die Auswertung und Ausführung von Policies betrachtet.

Das siebte Kapitel befaßt sich mit der Implementierung der Managementanwendung basierend auf XML, Java und CORBA. Das Kapitel beschreibt die Umsetzung der Policy-Sprache in XML und ausgewählte Implementierungsaspekte der Managementanwendung. Außerdem wird ein experimenteller Integrationsagent und die Organisation des Quellcodes vorgestellt.

Im achten, abschließenden Kapitel werden die Ergebnisse der Arbeit zusammengefaßt. Hier sind auch Hinweise bezüglich weiterer möglicher Arbeiten, sowie gegenwärtiger Desiderate zu finden.

Kapitel 2

Szenario

Aufgrund der ständig wachsenden IT-Infrastruktur beschließen mehr und mehr Unternehmen, deren Betrieb und Wartung spezialisierten Dienstleistern zu überlassen (outsourcing). Die Abrechnung der Nutzung und der Kosten einer unternehmensintern betriebenen Infrastruktur diente lediglich zur Überprüfung ihrer Wirtschaftlichkeit und gegebenenfalls zur Abrechnung zwischen Abteilungen des Unternehmens. Dagegen nimmt die Abrechnung der extern zur Verfügung gestellten Dienste eine wichtige Rolle ein.

Ausgehend von einem grundlegenden Szenario als Ausgangssituation werden zwei Unterszenarien entwickelt. Das erste beschreibt den Status Quo im System des Anbieters. Das zweite zeigt die bei Veränderungen akut auftretenden Probleme auf. Mit Begriffen aus dem Bereich der Softwareentwicklung werden die Unterszenarien als *Primär-* bzw *Sekundärszenario* bezeichnet.

2.1 Grundlegendes ISP-Szenario

In [Radi 02] wird ein ISP (Internet Service Provider, Anbieter) betrachtet, der für eine Versicherungsgesellschaft Intranet/Extranet-Dienste bereitstellt. Die Versicherungsgesellschaft (Kunde) verfügt über zentrale Anwendungen im Intranet (unternehmensspezifisch) sowie eine große Zahl von Außenstellen (Benutzer), denen der Zugang zu diesen Anwendungen ermöglicht werden soll. Zusätzlich werden DNS und Email zur Verfügung gestellt, sowie Zugang zum WWW mittels eines Proxyservers. Das hier betrachtete Szenario lehnt sich an das in [Radi 02] vorgestellte an.

Die Systeme, mittels derer diese Funktionalität umgesetzt wird, können jeweils einem der folgenden Bereiche zugeordnet werden (vgl. Abbildung 2.1):

ISA (Intranet Service Area) ist ein beim Kunden lokalisiertes Netz, das unternehmensspezifische Systeme (application server) verbindet. Es versteht sich als eine Ressource für die Anwender, die von hier beispielsweise aktualisierte Formulare, Verträge, Preislisten etc. beziehen können. Die Anwender können nicht direkt auf das ISA zugreifen; der Zugriff erfolgt über Komponenten im EISA.

IESA (Intranet Extranet Service Area) ist ein Bereich im Netz des Anbieters, der einen Teil der Dienste direkt zur Verfügung stellt (WWW, Email) und außerdem den Zugriff auf im ISA ansässige Dienste ermöglicht (Routing von/zu ISA). In diesem Bereich werden auch zentrale Messungen (*metering*) der Nutzung von Diensten durch Anwender vorgenommen.

Extranet bezeichnet die Menge der Außenstellen bzw. Filialen. Mittels ISDN-Routern und von dem Anbieter bereitgestellten Einwahlknoten kann von den Außenstellen eine Verbindung zum IESA hergestellt werden. Die Angestellten der Außenstellen können über diese Verbindung die angebotenen Dienste nutzen.

ISP-Bereich Als ISP-Bereich wird der 'private' Bereich des ISP bezeichnet. Hier befinden sich ein Teil des Abrechnungssystems sowie die Benutzerdatenbank.

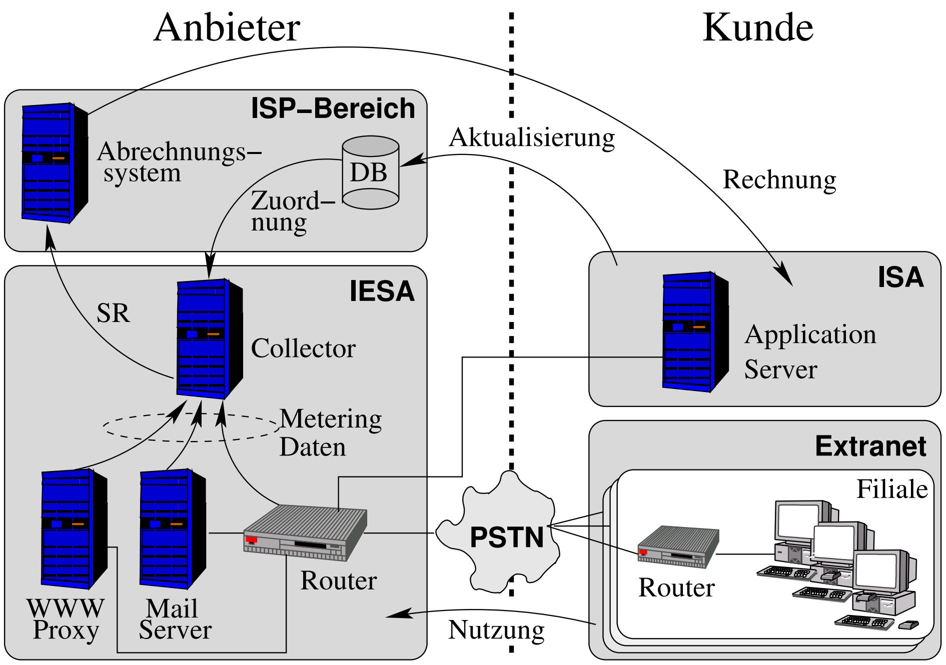

Abbildung 2.1: Szenario für Outsourcing

2.1.1 Abrechnungssystem des Anbieters

Die Messungen der Nutzung erfolgen im IESA-Bereich. Ein *Collector*, ebenfalls im IESA-Bereich, faßt die Meßdaten zu *Subscriber Records (SR)* zusammen, die dann einem Abrechnungssystem im ISP-Bereich zugeführt werden. Dieses wendet die vereinbarten Tarife auf die Nutzungsdaten an und erstellt in zeitlichen Abständen Rechnungen, die an den Kunden gesendet werden. Die in Abb. 2.1 als *Abrechnungssystem* bezeichnete Komponente übernimmt also die Aufgaben eines *Charging-* und *Billingsystems*.

Die Subscriber Records enthalten einzelnen Benutzern zugeordnete, aufsummierte Meßwerte. Um die Zuordnung Benutzer-Kunde zu realisieren, werden die Kennungen der einzelnen Benutzer aus der

Datenbank im ISP-Bereich gelesen. Der Kunde stellt Daten bezüglich aller (potentieller) Benutzer zur Verfügung und aktualisiert die Datenbank bei Veränderungen in der Benutzerpopulation.

2.1.2 Rollen

Die Mitarbeiter der Außenstellen stellen die eigentlichen Benutzer der Systems dar. Sie greifen auf das IESA zu, um aktualisierte Dokumente zu beziehen und mit anderen Außenstellen über Email zu kommunizieren. Die Nutzung des WWW-Dienstes wird in diesem Szenario als genehmigte, private Nutzung durch die Mitarbeiter betrachtet.

Der Anbieter stellt die Dienste zur Verfügung. Zusätzlich muß er die Abrechnung ihrer Nutzung gemäß der vertraglichen Vereinbarung mit der Versicherungsgesellschaft durchführen.

Der Kunde (d. h. die Versicherungsgesellschaft) trifft Vereinbarungen bezüglich der Art, der Güte und des Tarifs der angebotenen Dienste mit dem Anbieter. Um eine detaillierte Abrechnung der Dienstnutzung zu ermöglichen, stellt der Kunde dem Anbieter die Benutzerdaten jedes Mitarbeiters (d. h. potentiellen Benutzers) zur Verfügung.

2.2 Primärszenario für die Abrechnung

Aufgrund der Vereinbarungen zwischen Kunde und Anbieter wird die Nutzung des IESA auf Nutzungs- bzw. Volumenbasis abgerechnet. Dabei werden die Gebühren, die für die ISA-Nutzung anfallen und die Inanspruchnahme von *privat genutzten* Diensten (WWW) getrennt abgerechnet. Dies ermöglicht es dem Kunden, die Kosten privater Nutzung an die jeweiligen Außenstellen bzw. Mitarbeiter weiterzugeben.

Die Trennung der Abrechnungen für geschäftliche und private Nutzung stellen besondere Anforderungen an das Abrechnungssystem. Die Messung der Dienstnutzung (metering) muß sowohl am WWW-Proxy erfolgen (private Nutzung), als auch an den Einwahlknoten und den Routern, die Verbindungen zum ISA herstellen. Nach Erfassung der Dienstnutzung müssen die vereinbarten Tarife auf die Meßdaten angewandt werden (charging) und je nach Art einzelnen Benutzern oder Außenstellen zugeordnet werden.

2.3 Sekundärszenarien

Im Laufe der Zeit ändern sich die Bedürfnisse des Kunden und damit seine Anforderungen an den Anbieter. Der Bedarf des Kunden an der Nutzung bestehender Dienste wächst, während er gleichzeitig die Einführung neuer Dienste wünscht. Dies erhöht zwar den Umsatz für den Anbieter, stellt jedoch auch neue und höhere Anforderungen an seinen Abrechnungsvorgang.

Strukturelle Veränderungen Nach einer Fusion der Versicherungsgesellschaft mit einem anderen Unternehmen ändert sich die Unternehmensstruktur. Das resultierende Unternehmen verfügt über mehr Mitarbeiter in mehr Außenstellen. Infolgedessen werden die Bedingungen für die Dienstleistung neu verhandelt. Für die neu hinzugekommenen Außenstellen wird ein besonderer Tarif vereinbart;

zusätzlich fordert der Kunde eine Abrechnung in Abhängigkeit der Dienstgüte (QoS). Der Kunde paßt die Anzahl, die Belegschaft und den Ort seiner Außenstellen im Zuge der Fusion an, was sich bezüglich der Dienstnutzung in Veränderungen der Benutzerpopulation auswirkt.

Dienste Für seine Mitarbeiter im Außendienst schließt die Versicherungsgesellschaft mit dem Anbieter einen Vertrag über die Nutzung von Mehrwertdiensten (wie etwa die Anzeige von Stadtplänen) ab, die von einer ASP[1]-Sparte des Betreibers angeboten werden. Die dienstliche Nutzung (im Sinne des Kunden) dieser Dienste soll auf die Außendienstmitarbeiter beschränkt sein; alle weiteren Mitarbeiter des Kunden sollen auf die Dienste zum Selbstkostenpreis im Rahmen „privater Nutzung" zugreifen dürfen. Desweiteren stellt der Kunde in Aussicht, zukünftig auch weitere, neue Dienste in Anspruch nehmen zu wollen.

Rechnungen und Berichte Aus Gründen der Übersicht fordert der Kunde die Abrechnung der Gesamtnutzung auf einer einzelnen Rechnung; die Nutzung soll getrennt nach Diensten aufgeführt sein. Außerdem soll ein Abruf des aktuellen Kontostands möglich sein (hot-billing). Zusätzlich zu den Rechnungen sollen dem Kunden periodisch detaillierte Berichte bezüglich privater und geschäftlicher Nutzung des Systems zukommen.

2.4 Managementsicht

Das Abrechnungssystem des ISP besteht aus kommerziellen Softwarekomponenten mit proprietären Schnittstellen und Datenformaten. Deshalb sind Gateways erforderlich, die eine Kommunikation zwischen den einzelnen Komponenten ermöglichen. Das Ersetzen einer Komponente hat Änderungen an den von ihr abhängigen Gateways zur Folge.

Um die Anforderungen des Primärszenarios zu erfüllen muß der Anbieter Meßkomponenten zur Erfassung der Dienstnutzung an Systemen im IESA-Bereich installieren. Abrechnungs- und Meßkomponenten müssen mit Hilfe von Gateways verbunden werden, wie auch die die Abrechnungskomponenten untereinander.

Um den neuen Anforderungen des Sekundärszenarios zu begegnen muß der Anbieter einschneidende Veränderungen am Abrechnungssystem vornehmen. Die Forderung bezüglich QoS-abhängiger Abrechnung kann mittels Installation von Meßkomponenten auf Client-Seite (d. h. am ISDN-Router der einzelnen Außenstellen) erfüllt werden. Dazu muß das Abrechnungssystem so konfiguriert werden, daß es die Meßdaten bei der Erstellung von Subscriber Records berücksichtigt.

Neu hinzukommende Außenstellen und Mitarbeiter ziehen die Neukonfiguration entsprechender Client-seitigen Meßkomponenten nach sich. Dies kaskadiert wiederum in entsprechende Anpassungen der Konfiguration des Abrechnungssystems. Da Werkzeuge zur Automatisierung dieses Prozesses fehlen, erweisen sich solche Änderungen einerseits als ineffizient bezüglich Zeit und Kosten, andererseits als fehleranfällig.

Um ad-hoc-Berichte erstellen zu können, muß das bestehende Billingsystem durch ein anderes ersetzt werden, was dazu führt, das eine neue Gateway zwischen Collector und Billingsystem implementiert

[1]Application Service Provider

werden muß. Generell müssen bei Austausch einer Komponente des Abrechnungssystems Gateways modifiziert oder neu implementiert werden. Das zieht seinerseits Veränderungen am Management der Komponente, sowie an seiner Gateway nach sich.

2.5 Anforderungskatalog

Aus dem Szenario ergeben sich Anforderungen an ein Managementsystem für die Abrechnung. Die weiteren Teile der Arbeit beziehen sich auf die folgende Aufstellung, indem sie versuchen, die hier genannten Anforderungen zu erfüllen. Hintergrund der Aufstellung ist die Forderung nach Flexibilität des Abrechnungssystems bezüglich äußeren (Benutzerpopulation, Dienste, Kundenerwartungen) und inneren (Systemkomponenten, Betriebsweise, Geschäftsmodelle) Veränderungen.

Der Katalog ist entsprechend den Bedürfnissen der am Abrechnungsvorgang beteiligten Akteure organisiert. Diese sind auf Dienstnehmerseite der Kunde (d. h. die Versicherungsgesellschaft) und die Benutzer (die Mitarbeiter der Versicherungsgesellschaft in ihrer Rolle als Endbenutzer); auf Dienstleisterseite sind es der Provider (der ISP als Betreiber von sowohl Diensten als auch des Abrechnungssystems) und die Administratoren (Mitarbeiter des ISP, die die Systeme warten; Endbenutzer des Managementsystems). Aus deren Standpunkte ergeben sich dabei jeweils verschiedene Sichten, anhand derer die Anforderungen gruppiert werden.

2.5.1 Kundensicht

Der Kunde fordert einen für ihn bequemen und transparenten Abrechnungsvorgang. Als Dienstnehmer betrachtet er nur das Kosten/Nutzen-Verhältnis der angebotenen Dienste — die Implementierung der Abrechnung ist aus seiner Betrachtungsweise nicht relevant. Vielmehr liegt es in seinem Interesse, das Angebot den Bedürfnissen seiner eigenen Geschäftsprozesse anpassen zu lassen und dabei die Dienstgüte einzelner Dienste garantieren zu lassen. Organisatorische Veränderungen werden unabhängig von den damit für den Anbieter verbundenen Kosten betrachtet.

Zusätzlich erlaubt der Kunde seinen Mitarbeitern zum Selbstkostenpreis die Nutzung einiger Dienste und unterstützt vertretbare, abrechungsrelevante Forderugen ihrerseits gegenüber dem Anbieter; diese Forderungen sind in Abschnitt 2.5.2 gesondert aufgeführt.

Anforderungen aus Kundensicht

KUN-1 Dienstorientierte Abrechnung: Abrechnung verschiedener Dienste soll getrennt vorgenommen werden.

KUN-2 Abrechnung soll für verschiedene Benutzer getrennt vorgenommen werden.

KUN-3 Der Zugriff verschiedener Benutzergruppen auf die Dienste soll festgelegt werden können.

KUN-4 Preissetzung soll der Dienstgüte folgen.

KUN-5 Neue Dienste sollen hinzugefügt werden können.

KUN-6 Neue Standorte sollen schnell eingebunden (und bedient) werden können.

KUN-7 Neue Benutzer sollen leicht (wenn möglich automatisch) im laufenden Betrieb aufgenommen werden können.

KUN-8 Rechnungen sollen in regelmäßigen Abständen gestellt werden.

KUN-9 Berichte bezüglich der Nutzung sollen bei Bedarf angefordert werden können.

KUN-10 Der aktuelle Stand der Nutzung (Zwischenrechnung) soll angefordert werden können.

2.5.2 Benutzersicht

Der Benutzer ist Mitarbeiter des Kunden und der eigentliche Nutzer der vom Provider angebotenen Dienste. Er nimmt diese Dienste als Teil seiner Beschäftigung in Anspruch; die daraus entstehenden Anforderungen sind also mit den obigen, für den Kunden angegebenen, kongruent. Dabei darf die Abrechnung der Dienste allerdings ihre Inanspruchnahme nicht erschweren oder behindern — sie muß ohne Zutun der Benutzer automatisch abgewickelt werden können.

Aus der Sicht des Benutzers als Mitarbeiter ergeben sich außerdem Anforderungen bezüglich der Abrechnung *privater*, d. h. von der Versicherungsgesellschaft erlaubter, aber kostenpflichtiger Dienstnutzung:

BEN-1 Maßnahmen zur Abrechnung der Nutzung dürfen den Betrieb (d. h. die Inanspruchnahme der Dienste) nicht beeinträchtigen.

BEN-2 Für die „private" Nutzung, die der Benutzer selbst zu vergüten hat, sollen Rechnungen gestellt werden.

BEN-3 Der Benutzer soll jederzeit den aktuellen Stand seiner privaten Nutzung zur Kostenkontrolle einsehen können.

2.5.3 Providersicht

Als sowohl Anbieter der Dienste als auch Betreiber des Abrechnungssystems stellt der Provider Anforderungen an das Abrechnungsmanagement, die sich sowohl aus den Vertragsparametern ergeben, als auch mit Blick auf zukünftige Aufgaben des Abrechnungssystems. In einem dynamischen Umfeld stehen dabei leichte Veränderung und gute Erweiterbarkeit im Vordergrund.

PRO-1 Benutzer verschiedener Konfigurationen sollen diesen entsprechend gruppiert werden können.

PRO-2 Benutzer sollen entsprechend ihrer zugewiesenen Rollen gesondert behandelt werden können.

PRO-3 Komponenten des Abrechnungssystems sollen leicht ausgetauscht werden können.

PRO-4 Erweiterbarkeit: neue Parameter der Abrechnung sollen leicht eingeführt werden können (QoS, dynamic pricing etc.).

PRO-5 Veränderungen der Geschäftsprozesse bzw. Verträge sollen leicht und schnell in der Abrechnung umgesetzt werden können.

PRO-6 Messungen verschiedener Größen (Zeit, Volumen etc.) sollen realisierbar sein.

PRO-7 Die Granularität der Messungen (z. B. byte, kilobyte ...) soll variabel sein.

2.5.4 Administratorsicht

Als Endbenutzer des Managementsystems liegt es im Interesse der Administratoren, eine möglichst einheitliche Sicht auf den Abrechnungsvorgang zu erhalten. Um das Management der Systeme nicht von einzelnen Administratoren abhängig zu machen, soll das Managementwissen im System zugänglich gemacht werden. Etablierte Managementvorgänge sollten außerdem nicht von spezifischen Systemkomponenten abhängig sein. Diese sich indirekt aus dem Szenario ergebenden Überlegungen können ebenfalls als Anforderungen formuliert werden:

ADM-1 Management der Abrechnungskomponenten soll zentral möglich sein.

ADM-2 Die Managementsicht auf die Komponenten soll einheitlich sein.

ADM-3 Management soll invariant, d. h. unabhängig von der Implementierung des Abrechnungssystems sein.

ADM-4 Einarbeitung: Managementwissen soll möglichst im Managementsystem verankert sein, nicht bei einzelnen Administratoren.

ADM-5 Managementschnittstelle soll leicht zu erlernen und zu benutzen sein.

2.6 Zusammenfassung

Das in diesem Kapitel vorgestellte Szenario zeigt einige prinzipielle Schwächen der zur Zeit benutzten Abrechnungswerkzeuge auf, sowie die Probleme, die sich infolgedessen aus Managementsicht ergeben. Verteilte Komponenten verschiedener Hersteller weisen inkompatible Schnittstellen und Datenformate auf und müssen deshalb mittels Gateways betrieben werden. Das Management von Abrechnungskomponenten und Gateways ist nicht einheitlich. Desweiteren sind die einzelnen Komponenten nur schwer auf neue Anforderungen hin konfigurierbar — Änderungen der Kunden- oder Dienstprofile ziehen hohe Kosten zur Anpassung der Abrechnungssysteme und des dazugehörigen Management mit sich. Das Szenario zeigt ein typisches Beispiel gegenwärtigen Abrechnungsmanagements, das einem Umfeld mit hoher Änderungsdynamik nicht gewachsen ist.

Aus der dargestellten Problematik wurde ein Anforderungskatalog entwickelt, der allgemeine Anforderungen an ein Managementsystem für das Abrechnungsmanagement aufzählt und begründet. Der Katalog bildet eine Grundlage für die Erarbeitung konkreter Entwurfskriterien für die zu entwickelnde Managementanwendung.

Kapitel 3

State of the Art

Zu policy-basiertem Management im allgemeinen und policy-basiertem Abrechnungsmanagement im besonderen, sowie zur prozeßorientierten Betrachtungsweise existieren zahlreiche Vorarbeiten. Die Vereinigung dieser Konzepte in einer policy-basierten Managementanwendung für das prozeßorientierte Abrechnungsmanagement ist allerdings ein Novum. Ein direkter Vergleich mit früheren Ansätzen muß also entfallen. Dieses Kapitel gibt einen Überblick relevanter Publikationen und Implementierungen und bewertet ihre Bedeutung für diese Arbeit.

3.1 Abrechnungssysteme und -management

Abrechnungssysteme dienen dazu, die Nutzung von Ressourcen zu erfassen. Aus den resultierenden Daten können einerseits Rechnungen erstellt werden, andererseits können sie zur Einschätzung der Auslastung von Ressourcen oder anderen statistischen Aufgaben benutzt werden.

Um aufgrund der Nutzung eines Dienstes Rechnungen erstellen zu können, sind prinzipiell vier verschiedene Vorgänge notwendig:

- die Nutzung messen
- die Meßdaten einem Verbraucher/Kunden zuordnen
- einzelnen Posten einen Preis zuweisen
- die Posten zu einer Rechnung zusammenfassen

Der Abrechnungsvorgang kann anhand der folgenden Phasen beschrieben werden [TINA 96] [AAH 00]:

Metering An diversen Meßpunkten im System werden Meßdaten bezüglich der Nutzung von Ressourcen gesammelt. Diese können verschiedene Größen beschreiben, z. B. die Dauer der Nutzung eines Dienstes, übertragenes Datenvolumen, Anzahl durchgeführter Operationen etc.

Collecting/Classification Die Meßdaten werden nach Typen, Benutzern etc. sortiert und zusammengefaßt.

Charging/Tariffing Je nach Kontext (Benutzer, Uhrzeit der Nutzung) wird für die Daten ein Tarif gewählt und auf die Meßdaten angewendet.

Billing Die mit Preisen versehenen Daten werden für jeden Kunden zusammengefaßt. Unter Benutzung von Kundendaten (wie z. B. Rechnungsanschrift) kann so eine Rechnung für den Kunden erstellt werden.

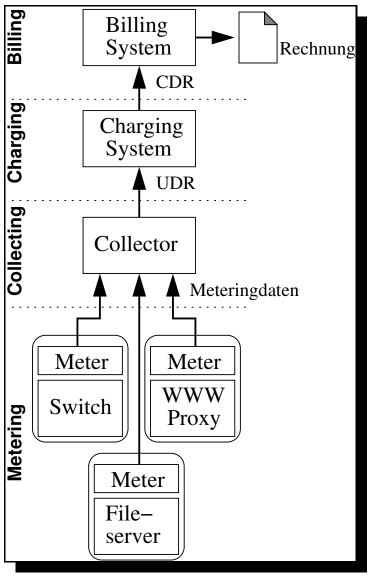

Abbildung 3.1: Architektur und Funktionsweise eines Abrechnungssystems

Ein Abrechnungssystem besteht (zumindest logisch) aus Komponenten, die für jeweils eine der genannten Phasen verantwortlich sind. *Abrechnungsmanagement* beschäftigt sich demzufolge mit der Verwaltung eines verteilten Systems, das aus eben diesen Komponenten besteht. Der Abrechnungsvorgang hat als Eingabedaten Tarife und Vorlagen (z. B. Format der Rechnung) einerseits, und durch Meteringkomponenten erfaßte Meßdaten andererseits. Das Ziel der Abrechnung ist die Erstellung einer korrekten, SLA-konformen Rechnung. Wie in Abbildung 3.1 skizziert, werden durch *Meter* Messungen an Zielkomponenten durchgeführt. Die Meßdaten aller Meter laufen beim *Collector* zusammen und werden dort zu *Usage Detailed Records*[1] (UDR) zusammengefaßt. Diese werden wiederum dem *Charging*-System zugeführt, das nach vorgegebenen Tarifen der Nutzungsposten jedes Endnutzers Preise zuordnet. Das Charging-System übergibt die so erstellten *Customer Detailed Records* an das *Billing*-System, das die Rechnung, also das Endprodukt des Abrechnungsvorgangs, erstellt.

3.1.1 Bemühungen zur Standardisierung

Abrechnung wird überall dort gebraucht, wo erfolgte Nutzung in Rechnung gestellt werden muß. Dies umfaßt einen riesigen Bereich, angefangen mit Telekommunikationsdiensten und Internet Service Providern bis hin zu Anbietern von Mehrwertdiensten und Inhalten. Einige dieser Teilbereiche existieren schon geraumer Zeit, sodaß zu erwarten wäre, daß Standards zur Nutzung und Management von Abrechnungssystemen längst etabliert wurden.

Das *Network Data Management-Usage*-Dokument der **IPDR**[2][IPDR 26] beschreibt ein Informationsmodell, sowie einen XML-basierten Rahmen für den Austausch für Benutzungsdaten. Die *Distributed Accounting Facility* (DAF) [DAF 02] von Fraunhofer **FOKUS** stellt eine Spezifikation für verteilte Nutzungserfassung dar. Sie definiert eine logische Architektur, sowie auf IPDR basierende Nutzungs-Schnittstellen (*usage API*) für den Austausch von Daten zwischen Abrechnungskomponenten. Die Spezifikation wurde von der Deutschen Telekom prototypisch implementiert. Das Dokument beschränkt sich jedoch auf die Spezifikation der Schnittstellen für die Nutzung der Komponenten, ohne Managementaspekte einzubeziehen oder Schnittstellen für das Management der Komponenten zu spezifizieren.

Im Rahmen des **OSI** Management wird in [ITU-T X.742][3] eine Architektur bestehend aus Metering-, Charging- und Billing-Komponenten spezifiziert; die *Recommendation* wurde mit Blick auf die Benutzung im Rahmen von TMN[4] entworfen. Es werden jedoch lediglich Schnittstellen zum Management

[1] Diese werden in der Literatur auch als *Session Detailed Records* (SDR) bezeichnet.
[2] Internet Protocol Detail Record; hier ist jedoch die gleichnamige Organisation gemeint http://www.ipdr.org
[3] Identisch mit dem ISO/IEC International Standard 10164-10.
[4] Telecommunications Management Network

der Metering-Komponente angegeben; die weiteren Komponenten werden als außerhalb des Rahmens der Spezifikation betrachtet, sodaß auf eine detaillierte Beschreibung sowie Schnittstellendefinitionen verzichtet wird.

Das **TINA**-Konsortium (TINA-C) spezifiziert in [TINA 96] eine Managementarchitektur für das Abrechnungsmanagement, Schnittstellen zum Management der Komponenten und Ereignisse, die während des Abrechnungsvorgangs eintreten können. Die Spezifikation erfolgt allerdings für die *TINA Service Architecture* [OMG 96-09-08] und kann nicht ohne weiteres auf eine heterogene Umgebung, wie sie im Szenario in Kapitel 2 beschrieben wird, übertragen werden.

Nach Betrachtung der Standardisierungsbemühungen muß also konstatiert werden, daß die existierenden Standardisierungsdokumente den Abrechnungsvorgang bezüglich Nutzung und Management nicht abdecken. Die meisten beschränken sich auf einen Teilbereich der Abrechnung oder lassen die Managementaspekte des Abrechnungsvorgangs unberücksichtigt.

3.1.2 Managementprozesse

Die Erbringung eines Dienstes und dessen Management kann als ein Modell von *Prozessen* betrachtet werden. Diese Sichtweise erlaubt eine enge Entsprechung zwischen den *Geschäfts*prozessen eines Betreibers und den zu ihrer Umsetzung erforderlichen Managementheuristiken, ohne Details von Diensten und Implementierungsmerkmale einbeziehen zu müssen.

Managementprozesse berücksichtigen zwar die Funktionsweise des Systems, richten sich jedoch ausschließlich auf die Erbringung des Dienstes entsprechend der mit dem Abnehmer vereinbarten Kriterien (Dienstgüte) aus (vgl. [HAN 99]). Die Art und Organisation der Prozesse eines Anbieters in einem Prozeßmodell sind höchst spezifisch für diesen Anbieter, so wie sich auch seine Geschäftskonzepte von denen anderer Anbieter unterscheiden. Der Ausgangspunkt für die Spezialisierung eines Prozeßmodells können allgemein gehaltene Prozeßmodelle sein, die an die Bedürfnisse eines Anbieters angepaßt werden können. Ein Beispiel dafür ist das *Telecom Operations Map* (TOM) [TMF 910] des **TM Forum**, das operationale Prozesse in einem allgemeinen Modell organisiert. Aus Managementsicht entsprechen diese den Managementprozessen zur Verwaltung und Steuerung des den operationale Prozeß implementierenden Systems. Das TM Forum geht bei der Erstellung seines Modells nach einem *top-down*-Ansatz vor, indem zusammen mit Unternehmen der IT-Branche eine Spezifikation vereinbart wird. Umgekehrt werden in der *IT Infrastructure Library* (**ITIL**) die in der Praxis umgesetzten Konzepte untersucht und bewertet, um dann als *deqbest practices* beschrieben zu werden.

Ein Beispiel der Anwendung eines allgemeinen Prozeßmodells stellt die von dem **FORM**-Consortium in [FORM99] beschriebene logische Architektur für ihr *Open Development Framework* dar, die unter Berücksichtigung von Prozessen und ihren Beziehungen zueinander entwickelt wird.

Ein allgemeines Prozeßmodell kann nicht nur bezüglich den Anforderungen eines Anbieters spezialisiert werden, sondern auch in Teilbereiche gegliedert werden. Speziell für das Abrechnungsmanagement kann ein Teilmodell betrachtet werden, das die Prozesse im Rahmen der Abrechnung beschreibt. In [Radi 02] werden die Teilprozesse der Abrechnung identifiziert und entsprechend des Lebenszyklus eines Dienstes organisiert (siehe Abbildung 4.2). Diese Gliederung des Abrechnungsvorgangs in Prozesse wurde der in dieser Arbeit entwickelten einheitlichen Sicht auf das Abrechnungsmanagement zugrunde gelegt.

3.2 Policies

Policies stellen im Grunde Regeln dar, die die Verhaltensweise eines Systems beeinflussen. Ihre Berechtigung ergibt sich aus der Notwendigkeit, immer komplexer werdende (verteilte) Systeme zu beherrschen. In den Arbeiten am Imperial College of London wurde folgende allgemeine (hier sinngemäß übernommene) Definition für Policies formuliert (vgl. [Mari 97], [Dami 02]) :

> *Eine Policy ist eine persistente, deklarative, von Managementzielen abgeleitete Spezifikation einer Regel bezüglich des Verhaltens eines Systems.*

Der deklarative Charakter von Policies erlaubt es, das Verhalten des Systems zu spezifizieren, ohne dabei das konkrete Verfahren zu dessen Durchsetzung Schritt für Schritt, wie etwa in einem prozeduralen Ansatz (z. B. mittels Managementskripten) festzulegen. Wird das gewünschte Verhalten eines Systems als eine Menge Policies formuliert, kann das Managementwissen der Manager bzw. Administratoren eines Systems unabhängig von der konkreten Implementierung der Managementmaßnahmen erfaßt werden. Als Folge kann:

- das Managementwissen anderen (z. B. neuen) Managern verfügbar gemacht werden und
- die Anzahl der notwendigen manuellen Eingriffe mittels *Werkzeugen* verringert werden.

Die tatsächliche Entwicklung solcher Managementwerkzeuge erfordert die Einführung weiterer Konzepte. Zur Formulierung von Policies benötigt man eine dazu geeignete, deklarative Sprache. Es ist ebenfalls erforderlich, die Abbildung einer Policy auf konkrete Managementaktionen zu spezifizieren, wie auch die Art und Weise, in der die Managementaktionen ausgeführt werden.

3.2.1 Policy-Typen

Regeln bezüglich des Verhaltens eines Systems können unterschiedlich abstrakt formuliert werden. An den beiden Extremen stehen einerseits *strategische* Policies, die eine informelle Angabe darstellen, andererseits *operationale* Policies, die eine konkrete Maßnahme bezüglich eines Managementobjekts (MO) spezifizieren.

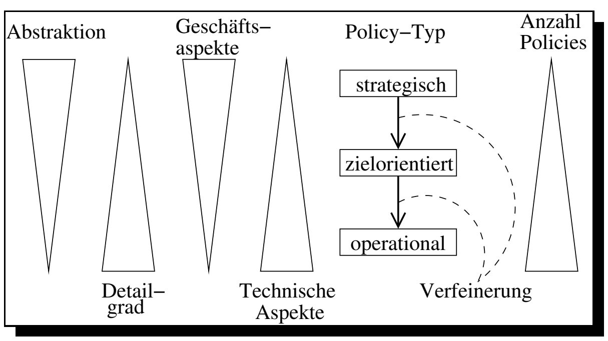

Abbildung 3.2: Policy-Hierarchie nach [Koch 96]

Die unterschiedlichen Abstraktionsebenen für Policies sind in einer *Policy-Hierarchie* organisiert [Wies 95], [Koch 96], [Mari 97], [Radi 98]. Die Anzahl der Policies wächst mit dem Detailgrad, wie anhand der in Abbildung 3.2 dargestellten Hierarchie angedeutet wird. Die Spezialisierung von strategischen zu operationalen Policies geschieht in einem Verfeinerungprozeß. Dieser hat als Ausgangspunkt aus geschäftlichen Aspekten abgeleitete, informelle Beschreibungen niedrigen Detailgrads. Davon ausgehend werden immer detailliertere, konkretere Policies entwickelt. Da in der vorliegenden Arbeit von operationalen Policies ausgegangen wird, wird auf die genannten Verfeinerungsschritte nicht weiter eingegangen.

3.2.2 Sprachen zur Policy-Definition

Zur Spezifikation von Policies kommen eigens zu diesem Zweck konzipierte Sprachen zum Einsatz. Es existieren zahlreiche Spezifikationen solcher Sprachen, die in unterschiedlichem Maße auf bestimmte Managementbereiche spezialisiert sind. Mit Ausnahme von Constraint-basierten Ansätzen (z. B. Miro´ [Heyd 90]) weisen die meisten Sprachen Gemeinsamkeiten bezüglich ihrer grundlegenden Elemente und Ausdrucksmächtigkeit auf.

Ponder

Ein prominentes Beispiel ist die Sprache *Ponder*[DDLS 00]. Ponder unterscheidet zwischen mehreren Typen von Policies und bietet Unterstützung für die Angabe von Rollen, Gruppen und Domänen. Als eines der Grundkonzepte kann die Typisierung der Policies als Mittel zur Angabe von *Rechten* und *Verpflichtungen* gesehen werden. Dabei werden (Zugriffs-) Rechte mittels *Authorization* Policies angegeben, die positive (Erlaubnis) und negative (Verbot) Rechte spezifizieren können. Verpflichtungen werden mittels *Obligation Policies* spezifiziert; ihr negatives Gegenstück findet man in den *Refrain Policies*, die eine Unterlassung angeben, unabhängig von den für die Aktion gewährten Rechte (vgl. [SlLu 99], S. 7f).

Weitere unter dem Begriff der Policy zusammengefaßte, grundlegende Konstrukte sind *Metapolicies* und eine Anzahl Gruppierungsmechanismen. Die Gruppierungsmechanismen *Role*, *Group*, *Relation* und *Management Structure* von Ponder dienen lediglich der Gruppierung der Policies — nicht der von den Policies verwalteten Entitäten. Die Art und Weise, in der dies geschieht (vgl. [DDLS 00], Abschnitt 5) läßt allerdings eine Abbildung der Policy-Gruppen auf (extern verwaltete) Entitätsgruppen zu. Eine *Role* wird beispielsweise als eine Gruppe von Policies mit gemeinsamen Subjekt definiert. In einer Organisationshierarchie hingegen wird eine Rolle Entitäten (z. B. Personen) zugewiesen, die bestimmte Tätigkeiten ausführen oder bestimmte Privilegien besitzen.

Ponder unterstützt für manche Elemente objektorientierte Konzepte, beispielsweise die Spezialisierung von Domänen und Rollen, sowie die Definition von „Templates" (Vorlagen), aus denen wiederum mittels Spezialisierung Policies abgeleitet werden können.

Weitere Policy-Sprachen

Neben *Ponder* existieren weitere allgemeine Sprachen, wie etwa die bei Bell Labs entwickelte *PDL* [LBN 99] oder die in [Krel 97] entwickelte Sprache. Sprachen zur Spezifikation von Policies werden nicht immer für allgemeine Zwecke entwickelt. Für bestimmte Managementbereiche wurden Sprachen zur Policy-Definition entwickelt, deren Anwendbarkeit auf den anvisierten Managementbereich hin geprüft wurde. Trotzdem weisen diese „spezialisierten" Sprachen große Gemeinsamkeiten mit den „allgemeinen" auf, sodaß sie ohne weiteres in anderen als den vorgesehenen Bereichen eingesetzt werden könnten. Als Beispiel können die in [Radi 98] spezifizierte Sprache zum Management nomadischer Systeme, sowie die in der vorliegenden Arbeit entwickelte Sprache (siehe Kapitel 5) genannt werden. Eine umfangreichere Übersicht von Sprachen zur Policy-Spezifikation kann [Dami 02] entnommen werden.

Gemeinsamkeiten

Eine (operationale) Policy in einer dieser Sprachen besteht aus mehreren Teilen, deren Semantik folgendermaßen beschrieben werden kann: Das Objekt, für das die Policy spezifiziert wurde wird in einem *Subject*-Feld ausgedrückt. Die Zielobjekte (MOs) die von der Policy beinflußt werden, sind in einem *Target*-Feld angegeben. Dies schließt natürlich nicht aus, daß der Inhalt von Subject- und Target-Feld identisch sind. Managementoperationen, die auf den Zielobjekten ausgeführt werden, sind in einem *Action*-Feld formuliert. Diese Operationen werden beim Eintreffen bestimmter Ereignisse ausgeführt, die innerhalb der Policy in einem *Event*-Feld angegeben werden. Die Ausführung der Operationen kann außerdem von den in einem *Constraint*-Feld angegebenen Bedingungen abhängig gemacht werden. Aus den genannten Feldern aufgebaut kann eine Policy allgemein notiert werden als:

Policy:	SUBJECT TARGET EVENT ACTION CONSTRAINT

3.2.3 Informationsmodelle

Um eine Policy in einem System zu repräsentieren, ist ein Informationsmodell erforderlich, in dem die oben genannten Elemente dargestellt werden können. Es kann ebenfalls notwendig werden die Subjekte und Zielobjekte der Policies in einem Modell zu repräsentieren. Im folgenden werden Ansätze vorgestellt, die diese Aufgaben zu erfüllen versuchen.

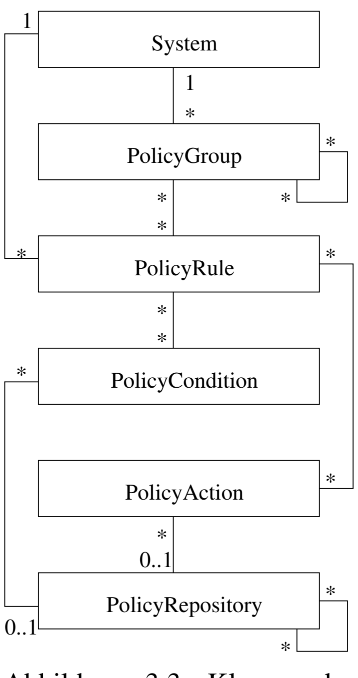

Abbildung 3.3: Klassen des *Policy Core Information Model* der IETF (Auszug)

Die *Distributed Management Task Force* (DMTF) spezifiziert im *Common Information Model* (CIM) [Core 24] ein objektorientiertes, erweiterbares Informationsmodell zur einheitlichen Beschreibung von Systemen. Mit CIM können in UML-Notation (siehe [RJB 99]) Komponenten eines Systems sowie ihre gegenseitigen Beziehungen dokumentiert werden. Durch Spezialisierung einzelner Elemente des CIM wurden Modelle für eine Anzahl Domänen geschaffen. Insbesondere hervorzuheben ist das *CIM Policy Core Information Model* [DSP 0108], das Elemente von Policies und ihre Beziehungen als Objekte und Assoziationen darstellt.

Basierend auf CIM entwickelte die IETF in [MESW 01] das *Policy Core Information Model* (PCIM) zur Beschreibung von Policies und deren Bestandteilen. Im Rahmen der Zusammenarbeit der beiden Gremien DMTF und IETF wurden Erfahrungen mit PCIM in das CIM Policy Core Information Model eingebracht [Radi 03], sodaß seine vorliegende Version 2.7 weitgehend zu PCIM kongruent ist.

3.2.4 Verarbeitung von Policies

Die Festlegung eines Informationsmodells und einer Sprache reichen als Grundlage für die Modellierung einer Managementanwendung nicht aus. Es sind Heuristiken notwendig, die das Verhalten des Systems bezüglich *Auswertung*, *Ausführung* und *Verwaltung* der Policies bestimmen.

Die existierenden Architekturen für Systeme zur Policy-Verarbeitung spezifizieren:

- *Point of Decision* (POD) als Komponente zur Auswertung von Policies
- *Point of Enforcement* (POE) als Komponente zur Ausführung von Aktionen
- *Policy Repository* als Komponente zur Verwaltung der Policies

Die Auswertung einer Policy geschieht im *Point of Decision*. Ereignisgesteuerte Policy-Systeme suchen dabei für ein eingehendes Ereignis die passenden (d. h. diejenigen, deren Event-Feld den Typ des Ereignisses enthält) heraus. Die Bedingungen (Constraints) jeder „gefundenen" Policy werden geprüft. Ist die Bedingung *wahr*, ordnet der *Point of Decision* die Ausführung der in der Policy angegebenen Aktionen an. Die Auswertung der Policies geschieht mittels eines Policy-Interpreters. Dieser kann entweder die textuelle Fassung der Policies bei Bedarf parsen und evaluieren, oder eine kompilierte Form der Policy auswerten.

Die Ausführung der Aktionen einer Policy obliegt dem *Point of Enforcement*. Die tatsächliche Ausführung hängt dabei von der zum Management benutzten Architektur ab. Naheliegend ist eine Umsetzung der Operationen mittels einer Middleware wie CORBA oder Java RMI.

Verteilung von Policies Ein Managementsystem kann zwei Strategien zur Umsetzung der in Policies spezifizierten Regeln verfolgen: zentrale oder verteilte Auswertung der Policies. Der Ansatz mit zentraler Auswertung setzt voraus, daß die Auswertung und Ausführung der Policies in einer einzigen Komponente geschieht. Dies vereinfacht ihre Verwaltung, hat jedoch auch die gängigen Nachteile einer zentralen Komponente zur Folge (schlechte horizontale Skalierbarkeit, *single point of failure* etc.).

Der Ansatz mit verteilter Auswertung der Policies hingegen nutzt die Vorteile der Lokalität aus und skaliert gut. Allerdings impliziert dieser Ansatz mehrere (zur Anzahl der Systeme proportionale Zahl) Komponenten, die Policies auswerten. Will man Lokalitätsvorteile ausnutzen, steht man deshalb vor der Aufgabe, die Policies an die richtigen *Points of Decision* (POD) zu verteilen. Ein Modell dafür wird in [DLSD01] vorgestellt.

Konfliktauflösung Policies können — als die Regeln, die sie darstellen — zueinander widersprüchlich formuliert werden. Deshalb sind Mittel erforderlich, um solche Konflikte zu erkennen und zu beseitigen.

Bezüglich der Sprache Ponder wird in [LuSl 97] das Problem von konfliktären Policies betrachtet und eine Erkennungsstrategie basierend auf der Analyse überlappender Domänenangaben vorgeschlagen. Im CIM ist Konfliktauflösung basierend auf der Zuweisung von Prioritäten an Policies vorgesehen, während für die bei Bell Labs entwickelte PDL in [CLN 00] ein auf Prädikaten basierendes Verfahren entwickelt wurde.

Die Auseinandersetzung mit Konflikten zwischen Policies liegt außerhalb des Rahmens dieser Arbeit. Die im CIM vorgeschlagene, auf Prioritäten basierende Lösung, wurde allerdings in der in Kapitel 5 spezifizierten Sprache aufgegriffen.

Metapolicies Metapolicies stellen einen Verwaltungsmechanismus für Policies dar. Ihre Definition nach [Hosm 92], aufgegriffen in [Avit 98] kann übersetzt werden als:

> *Metapolicies sind Policies bezüglich Policies. Sie formulieren die ausdrückliche Regeln bezüglich Policies und koordinieren die Interaktion mehrerer Policies.*

Das Konzept wurde sowohl mittels operationaler Policies, als auch mit Hilfe logischer Ausdrücke (z. B. in Ponder) umgesetzt. Unter Metapolicies versteht man in Ponder die Angabe von Prädikaten, die beispielsweise Bedingungen für das Auftreten von Konflikten darstellen können. Die in der vorliegenden Arbeit spezifizierte *Policy Definition Language* unterstützt implizit Metapolicies als einen Spezialfall der formulierbaren operationalen Policies.

3.2.5 Policy-basierte Techniken in der Abrechnung

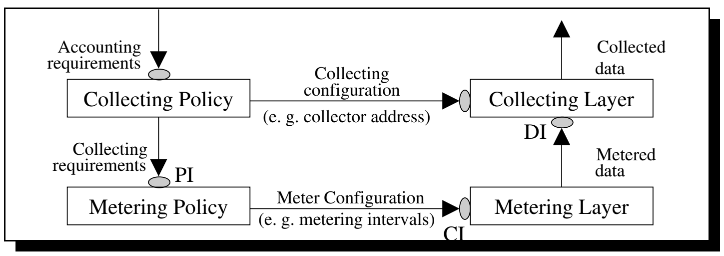

Abbildung 3.4: Das *Billing System Framework* aus [HaCa 99] (Nachgebildeter Ausschnitt der Originalskizze)

Für die policy-basierte Steuerung von Teilen des Abrechnungsvorgangs existieren bereits Konzepte, die sich allerdings jeweils auf bestimmte Teile der Abrechnung beschränken. In [HaCa 99] wird eine policy-basierte Architektur zur Durchsetzung von Tarifen beschrieben, die einen ähnlichen Ansatz aufweist, wie die in Kapitel 4 dieser Arbeit beschriebene Integrationsschicht. Die in Abbildung 3.4 mit „CI" (*configuration interface*) bezeichneten Schnittstellen sind Teil eines *configuration plane*, das eine einheitliche Schnittstelle zur Konfiguration darstellt. Die Anwendbarkeit dieses *configuration plane* wird jedoch nicht für den gesamten Abrechnungsprozeß, sondern lediglich für die Konfiguration verschiedener Tarife untersucht. Messung und Überwachung von Netzkomponenten (metering), die Zusammenfassung der Meßdaten (collecting), sowie die Zuweisung von Preisen (charging) werden nicht betrachtet. Im Ansatz werden Tarife basierend auf der Dienstgüte mittels *dynamic pricing* berücksichtigt. Dabei wird ausschließlich von einem auf DiffServ [BBC^{+} 98] basierenden Dienstangebot ausgegangen.

Basierend auf dieser Architektur beschreibt [ZZC 02] die konfigurierbaren Parameter der betrachteten Teile des Abrechnungsvorgangs (wie etwa Adressierung der von Meßkomponenten generierten Datensätzen und Granularität der Messung) und spezifiziert Schnittstellen für ihre Einstellung. Es wird zwischen *integrated accounting* und *discrete accounting* unterschieden, wobei ersteres als Teil eines Dienstes betrachtet wird und *discrete accounting* als eingenständiger Dienst (der für verschiedene Dienste adaptiert werden kann) modelliert ist.

Die beiden genannten Arbeiten befassen sich mit dem Einsatz von Policies zur Konfiguration von Netzkomponenten (*network elements*). Die dabei beschriebenen Policies sind auf die jeweilige Phase des Abrechnungsvorgangs (metering, collecting ...) spezialisiert und werden lokal (d. h. an jeder

Komponente) ausgewertet. Eine Sprache für die Notation der Policies wird nicht angegeben, lediglich die Beziehungen zwischen den Entitäten eines Abrechnungsvorgangs (aus denen Anforderungen an eine geeigenete PDL abgeleitet werden könnten) sind untersucht worden.

3.3 Positionierung der Arbeit und Zusammenfassung

In diesem Kapitel wurden die Grundlagen von Abrechnungssystemen und -management, sowie policy-basierter Steuerung und Prozeßorientierung skizziert. Dazu wurden Publikationen zu den Teilbereichen der vorliegenden Arbeit untersucht.

Die vorgestellten Policy-Sprachen weisen eine Gemeinsamkeit in der Aufteilung einer Policy in Felder auf. Auch wenn die Felder in den verschiedenen Sprachen dieselben sind, ist ihre Semantik nicht immer die gleiche. Ebenso gibt es Unterschiede bei der Spezifikation von Metapolicies und Laufzeitumgebung des policy-verarbeitenden Systems. Für die in dieser Arbeit entwickelten Sprache werden die sechs in Abschnitt 3.2 genannten Felder übernommen. Ihre Ausdrucksmächtigkeit und Semantik wird in den Kapiteln 4 und 5 festgelegt. Aus der Vielzahl von Ansätzen zu verschienenen Aspekten der Arbeit werden nun einige gewählt, denen besondere Bedeutung zugemessen werden kann:

- Die Unterstützung für Rollen wird auf extern definierte Rollen beschränkt, etwa solche, wie sie im Sicherheitsmanagement vorkommen; die Zuweisung von Rollen an die Policies selbst scheint im Kontext dieser Arbeit weder notwendig noch wünschenswert.
- Metapolicies werden als operationale Policies angenommen. Ein Ansatz mit Prädikaten würde die Spezifikation einer eigens dafür vorgesehenen Sprache und entsprechenden Mehraufwand bei der Implementierung der Laufzeitumgebung erfordern.
- Der in PCIM vorgeschlagener Ansatz zur Konfliktauflösung zwischen Policies wird in die Policy-Sprache mit übernommen. Konflikte zwischen Policies werden zwar als Problem zur Kenntnis genommen, jedoch muß im Rahmen dieser Arbeit auf eine detaillierte Betrachtung verzichtet werden.
- Für die Anforderungen dieser Arbeit reicht die Betrachtung operationaler Policies aus. In der Policy-Hierarchie höher liegende Policies (strategische bzw. funktionale) werden deshalb aufbauenden Arbeiten überlassen.
- Die verteilte Ausführung der Policies wird beim Entwurf und der Implementierung berücksichtigt, repräsentiert jedoch ebenfalls keinen zentralen Baustein der Arbeit.
- Für die Implementierung einer Policy-Sprache sind Werkzeuge notwendig, die in allen vorgestellten Vorarbeiten eng an die Sprache gebunden, also proprietär sind. Sie führen also zusätzliche herstellerspezifische, wenn auch offengelegte, Elemente in das Abrechnungsmanagement ein (z. B. eigene Parser, Editoren etc.). Dieses Problem kann durch die Benutzung einer generischen Sprache wie XML für die Repräsentation der Policy-Sprache gelöst werden. Dadurch wird der Einsatz von XML-Standardwerkzeugen zur Bearbeitung von Policies ermöglicht.

Aus diesen Entscheidungen ergeben sich weitere Charakteristika der Policy-Sprache und der zu entwickelnden Managementanwendung. Er erscheint zweckmäßig, einen auf PCIM basierenden Ansatz anzustreben, wobei die Prozesse des Managementbereichs wann immer möglich in die Entwicklung mit einbezogen werden.

Die Einführung einer Managementsicht unter Verdeckung der herstellerspezifischen Teile des Abrechnungssystems muß existierende Spezifikationen berücksichtigen und sich, wann immer möglich, daran orientieren.

Die Implementierung der Managementanwendung basierend auf offene Standards, wie beispielsweise XML zur Repräsentation der Sprache und CORBA als Managementwerkzeug, verspricht ebenso die Probleme im Abrechnungsmanagement anzusprechen. Eine weitere Anforderung ist eine hochgradige Erweiterbarkeit und Wartbarkeit von sowohl Sprache als auch Managementanwendung. Diese ist bedingt durch den Einsatz in einem unbekannten Software-Umfeld mit hoher Änderungsdynamik.

Zusammen mit den in Kapitel 2 formulierten Anforderungen prägen die hier beschriebenen Charakteristika die weitere Entwicklung der für die Aufgabenstellung notwendigen Konzepte.

Kapitel 4

Konzeption

Der in dieser Arbeit vorgestellte Ansatz kombiniert policy-basierte Steuerung mit einer streng prozeßorientierten Sicht. In diesem Kapitel wird die Anwendung der prozeßorientierten Sicht auf das Abrechnungsmanagement erörtert. Zudem werden die Elemente der regelbasierten Steuerung angesprochen, die im Entwurf der in den folgenden Kapiteln entwickelten Managementanwendung eingehen. Die in Kapitel 2 formulierten Anforderungen, sowie die in Kapitel 3 angegebenen Vorarbeiten bilden dabei eine Grundlage der Vorgehensweise (vgl. auch Abbildung 1.2).

Nach der Diskussion der Prozesse des Abrechnungsvorgangs werden die Rahmenbedingungen des Entwurfs einer Policy Definition Language (PDL) untersucht, mittels derer Policies formuliert werden sollen. Dabei werden Mächtigkeit und Elemente der PDL festgelegt, sowie sich daraus ergebende Anforderungen für den Entwurf des Laufzeitsystems.

4.1 Prozeßorientierte Sicht auf den Abrechnungsvorgang

Der Begriff der *Prozeßorientierung* im Management begründet sich in der Abbildung von Geschäftsprozessen auf Managementprozesse. Das Ziel dieser Zuordnung ist die Entsprechung von Entitäten, Ereignissen und Aktionen aus betriebswirtschaftlicher Perspektive und aus Managementsicht (im Sinne des IT-Systemmanagements). Dabei werden die Prozesse unabhängig von ihrer Implementierung, insbesondere ohne Rücksicht auf eingesetzte Komponenten, betrachtet.

4.1.1 Die Teilprozesse der Abrechnung

Es erscheint sinnvoll, die Teilprozesse in drei Kategorien einzuteilen. Ein Teil der Prozesse können vollständig automatisch umgesetzt werden, während andere ausschließlich von Menschen durchgeführt werden müssen. Zusätzlich gibt es Teilprozesse, die nicht eindeutig einer der zwei genannten Gruppen zugeordnet werden können.

Tabelle 4.1 zeigt die Einteilung der in [Radi 02] angegebenen Teilprozesse der Abrechnung in diese drei Kategorien. Bei den „anwendergesteuerten" Prozessen in der linken Spalte handelt es sich dabei um Teilprozesse der Planungsphase eines Dienstes. In der mittleren Spalte befinden sich die Teilprozesse, die ohne Benutzerinteraktion auskommen, während in der rechten Spalte die „hybriden"

Anwendergesteuert	*Automatisch*	*Hybrid*
customer analysis	configure accounting system	implement/ instantiate meters
service analysis	deploy meters	pricing
cost allocation	deinstall meters	data rollout
	usage accounting	invoice verification
	charging	payment control
	billing	*error*
	reporting	
	change	

Tabelle 4.1: Kategorisierung der Teilprozesse der Abrechnung

Prozesse angegeben sind. Diese können unter Umständen softwaregestützt sein, sind aber nicht in allen Fällen ohne Benutzerinteraktion möglich.

Als Beispiel für einen hybriden Teilprozeß kann *invoice verification* hervorgehoben werden. Es handelt sich dabei um die Überprüfung der Rechnungen durch den Anbieter. Anhand von Heuristiken können auffällige Abweichungen zwischen den Rechnungen eines Kunden festgestellt werden. Diese Abweichungen können von einem veränderten Nutzungsverhalten des Kunden herrühren, oder einen Hinweis auf Fehler im Abrechnungsvorgang darstellen. Um die tatsächliche Ursache zu identifizieren ist die Überprüfung durch das Personal des Anbieters notwendig.

Merkmale eines Teilprozesses

Betrachtet man einen Prozeß allgemein, können folgende offensichtlichen Merkmale festgestellt werden:

Aktionen Jeder Prozeß trägt die Verantwortung für die Lösung einer Aufgabe. Zu diesem Zweck kann ihm eine Aktion oder eine Folge von Aktionen zugeordnet werden, die zur Erfüllung der Aufgabe benötigt wird.

Entitäten An einem Prozeß sind ein oder mehrere Akteure beteiligt, die Aktionen ausführen. Je nach Prozeß können sowohl Menschen als auch Maschinen (Rechner) die Rollen der Akteure innehaben. Die Akteure führen die Aktionen bezüglich *Zielen* aus, die ebenfalls Repräsentationen von Personen oder Maschinen sind. Akteur und Ziel können unter Umständen identisch sein; sie werden unter dem Oberbegriff *Entität* zusammengefaßt.

Ereignisse Durch Aktionen innerhalb oder außerhalb eines Prozesses entstehen *Ereignisse*. Sie markieren die erfolgreiche oder fehlgeschlagene Durchführung von Aktionen, Änderungen am Kontext des Prozesses usw. Das Eintreffen eines Ereignisses kann ein Anlaß zur Ausführung einer Aktion, zum Eintritt oder Verlassen eines Prozesses oder zur Fehlerbehandlung sein.

Aktionen, beteiligte Entitäten und mögliche Ereignisse charakterisieren einen Prozeß; zwei Prozesse, die in diesen drei Merkmalen übereinstimmen sind infolgedessen identisch.

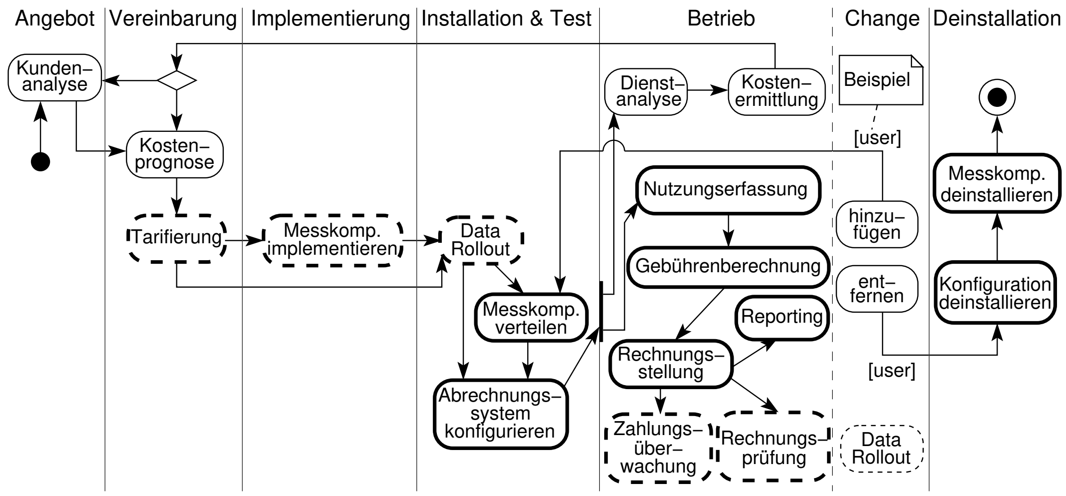

Abbildung 4.2: Die Teilprozesse der Abrechnung im Lebenszyklus eines Dienstes nach [Radi 03]

4.1.2 Teilprozesse als Zustände des Abrechnungsvorgangs

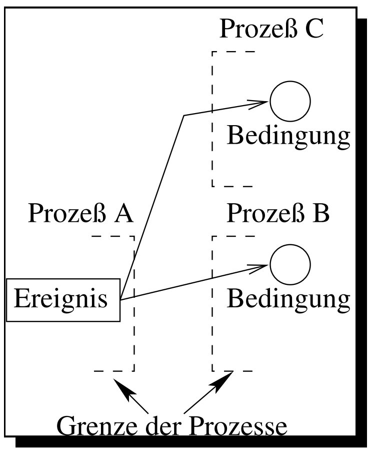

Abbildung 4.1: Ereignisse und Bedingungen eines Petri-Netzes an den Prozeßgrenzen

Ereignisse lösen Übergänge zwischen Teilprozessen aus. Die Verarbeitung eines Vorgangs verläßt den aktuellen Teilprozeß und wird in einem anderen weitergeführt. Die Übergänge sind nicht beliebig, sondern durch den ursprünglichen Teilprozeß und das Ereignis bestimmt. Zusätzliche Bedingungen können das Auslösen eines Übergangs mitbestimmen. Umgekehrt kann das Verweilen in einem Prozeß gewisse Randbedingungen implizieren. Als Mittel zur Darstellung der dynamischen Eigenschaften der Prozesse scheinen deshalb Petri-Netze ein geeignetes Mittel zu sein [HLK+ 98] (vgl. auch [EMS 00], [Cap 93]. Sie erlauben die graphische Notation von Zuständen, zustandsändernden Ereignissen und Aktionen, sowie Nebenbedingungen für einen Zustandsübergang.

Aus einer solchen Prozeßbeschreibung könnten operationale Policies (unter Umständen sogar automatisch) abgeleitet werden, indem die Bedingungen und Ereignisse an den Prozeßgrenzen betrachtet werden (Abbildung 4.1). Die Erstellung eines Petri-Netzes mit vielen Zuständen, Ereignissen und Bedingungen, so wie sie im Fall der Teilprozesse der Abrechnung vorkommen, kann allerdings eine hohe Komplexität besitzen. Da eine Prüfung der Korrektheit sich aus diesem Grund als schwierig erweisen würde, wird hier von diesem Ansatz abgesehen. Statt dessen wird die Darstellung mittels Aktivitätsdiagrammen entsprechend Abbildung 4.2 gewählt.

In der Abbildung sind die Teilprozesse der Abrechnung als eine Folge von Aktivitäten entlang des Lebenszyklus eines Dienstes (siehe [GHK+ 01a], [Radi 02]) angeordnet. Die *automatischen* Prozesse sind dabei mit einer dickeren Umrandung angedeutet, die *hybriden* mit einer gestrichelten. Man kann der Abbildung entnehmen, daß die für ein Managementsystem interessanten Teilprozesse der Abrechnung (nämlich die automatischen) vorwiegend in der Betriebsphase auftreten, sowie den Phasen vor und nach der Betriebsphase.

Der Prozeß *change*

Einer der wichtigsten Teilprozesse aus Managementsicht ist *change*. In diesem Teilprozeß werden Änderungen während des Betriebs eingebracht, bespielsweise das Hinzufügen oder Entfernen von Benutzern oder Diensten, oder Änderungen an Tarifen. Ihm entspricht eine Teilphase im Lebenszyklus eines Dienstes. Tatsächlich wird der Routinebetrieb für Änderungen ausgesetzt und danach wieder aufgenommen; die Rückkehr zu den Teilprozessen der Betriebsphase geschieht allerdings unter Umständen über andere Teilprozesse, wie im Beispiel der Abbildung 4.2 angedeutet wird.

Fehlerbehandlung

In jedem der genannten Prozesse können Fehler auftreten. Die Behandlung von Fehlern wird allerdings an keiner Stelle im Prozeßmodell ausdrücklich berücksichtigt. Es bieten sich drei Strategien an, um diese Lücke zu schließen:

1. Fehler werden in dem Prozeß behandelt, in dem sie auftreten.
2. Fehler werden in einem eigenen Prozeß behandelt.
3. Hybridisierung von 1 und 2: manche Fehler werden im Prozeß behandelt, in dem sie auftreten; andere werden delegiert.

Alternative eins sieht vor, die Fehlerbehandlung in den einzelnen Prozessen zusätzlich zu berücksichtigen. Der Vorteil dieses Ansatzes liegt auf der Hand: die Prozesse sind in sich abgeschlossen. Es muß jedoch bedacht werden, daß bereits einfache Prozesse ein breites Spektrum potentieller Fehlerfälle aufweisen können. Würde also eine umfassende Behandlung von Fehlerfällen in jedem Prozeß eingeführt, führte dies zu einer viel höheren Komplexität der Prozesse. Zusätzlich ist zu erwarten, daß einige Fehlertypen in mehreren Prozessen auftreten können. Auch wenn ihre Behandlung in allen Prozessen gleich wäre, müßte diese gleichwohl in jedem Prozeß vorgesehen sein; dies würde wiederum zu Redundanz in den den Prozessen innewohnenden Verfahren führen.

Der Ansatz von Alternative zwei sieht hingegen vor, die bisherige Spezifikation der Prozesse unverändert zu belassen. Beim Auftreten von Fehlern wird ihre Behandlung an einen eigens dafür vorgesehenen Prozeß delegiert. In diesem Prozeß werden die Fehlerfälle aller Prozesse behandelt; es ist offensichtlich, daß die Komplexität dieses Prozesses mit der Anzahl Fehlertypen und Maßnahmen zur Fehlerbehandlung wächst. Der Ansatz impliziert auch, daß Verfahren anderer Prozesse außerhalb ihres Kontextes durchgeführt werden. Er bietet allerdings den Vorteil, Fehlerbehandlung in die Prozesse einzuführen und gleichzeitig das bereits erstellte und geprüfte Modell intakt zu erhalten. Das bei Alternative eins auftretende Problem der Redundanz definierter Verfahren erübrigt sich außerdem.

Eine Kombination der beiden Ansätze, wie sie in Alternative drei angedeutet wird, erlaubt es, den Ort der Behandlung von Fehlern für jeden Fehlertyp einzeln festzulegen. Auf diese Weise wäre es möglich, die Integrität der Prozesse weitgehend zu erhalten, während eine übermäßige Erhöhung der Komplexität durch Delegierung einzelner Fehlertypen vermieden wird. Indem Verfahren zur Behandlung von Fehlern an zwei Stellen spezifiziert werden, ist die Lokalisierung der einzelnen dazu benutzten Aktionen (und damit ihre Zuordnung zu Prozessen) unter Umständen von Fall zu Fall anders.

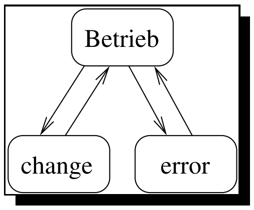

Abbildung 4.3: Übergänge zwischen Routinebetrieb, *change* und Fehlerbehandlung

Dem Teilprozeß *change* wird in der Literatur [HAN 99] ein Teilprozeß zur Fehlerbehandlung zur Seite gestellt. Davon ausgehend und aufgrund der obigen Diskussion wird an dieser Stelle ein weiterer Prozeß namens *error* eingeführt, in dem die Behandlung der in allen Prozessen auftretenden Fehler stattfinden soll. Es handelt sich dabei offensichtlich um einen *hybriden* Prozeß (vgl. Tabelle 4.1), da die Behandlung auftretender Fehler automatisch und/oder durch Personen (Administrator, *Help Desk*-Personal etc.) durchgeführt werden kann.

4.1.3 Policies im prozeßorientierten Management

Wie bereits in Kapitel 3 festgestellt wurde, bestehen Policies aus den Feldern Subject, Target, Event, Action und Constraint. Bildet man diese Felder auf die Merkmale von Prozessen ab, ergibt sich die in Abbildung 4.4 dargestellte Zuordnung.

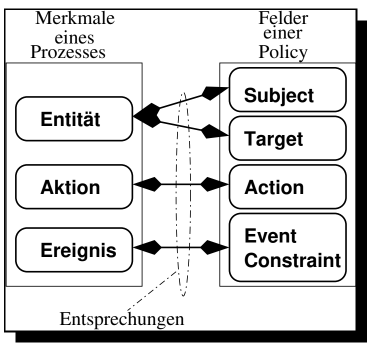

Abbildung 4.4: Abbildung der Prozeßmerkmale auf die Felder einer Policy

Subject und Target repräsentieren Systeme und entsprechen damit den Entitäten der Prozesse. Die Aktionen der Policies finden leicht eine Entsprechung in den Prozeßmerkmalen, während Event und Constraint zusammen den Ereignissen der Prozesse zugeordnet werden können. Ein Ereignis in der prozeßorientierten Sicht gilt somit genau dann als eingetreten, wenn das Event einer Policy eintritt und ihre Constraints *wahr* sind.

Policies eignen sich aufgrund dieser Übereinstimmung gut für das prozeßorientierte Management. Einerseits sind sie in der Lage, die Merkmale der Teilprozesse zu repräsentieren, andererseits können sie durch Ausführung von Aktionen Übergänge zwischen Prozessen realisieren. Ein solcher Ansatz erfüllt die die Anforderung **ADM-4**, indem Managementwissen deklarativ in Form von Policies im System festgehalten wird. Durch den Einsatz einer Policy-Sprache reduziert sich der Aufwand zur Einarbeitung in das Management eines Systems auf das Erlernen dieser Sprache, womit die Anforderung **ADM-5** angesprochen wird.

4.2 Anwendung der prozeßorientierten Sicht

Die Betrachtung des Abrechnungsvorgangs unter Abstraktion von dessen Implementierung erlaubt eine implementierungsunabhängige, einheitliche Managementsicht auf die Abrechnung entsprechend den Anforderungen **ADM 2** und **ADM 3**. Abbildung 4.5 zeigt die prinzipielle Realisierung dieser Sicht mit Hilfe einer aus Agenten bestehenden Integrationsschicht. Ein Agent repräsentiert dabei eine Abrechnungskomponente gegenüber der Managementanwendung, indem er auf die herstellerspezifischen Schnittstellen der Komponente zugreift und der Managementanwendung eine einheitliche Managementschnittstelle bietet.

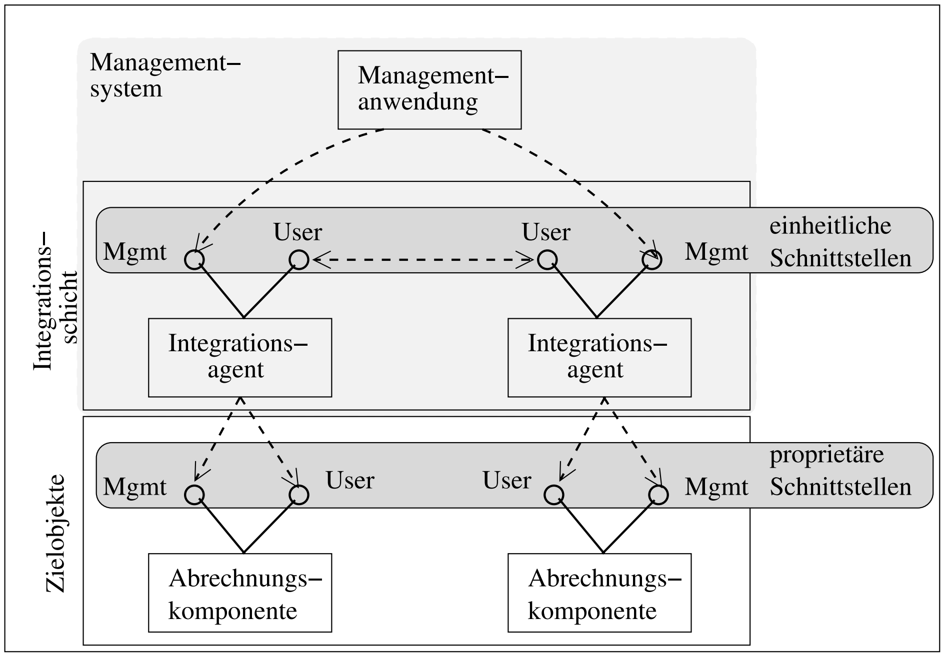

Abbildung 4.5: Einführung einer Integrationsschicht zur Vereinheitlichung der Benutzungs- und Management-Schnittstellen

4.2.1 Integrationsschicht

Indem zwischen Managementanwendung und Abrechnungskomponenten eine Integrationschicht eingeführt wird, verlagert sich die Aufgabe der Vereinheitlichung der Schnittstellen aus der Managementanwendung heraus. Eine einheitliche Sicht auf den Abrechnungsvorgang für einen Manager bzw. Administrator wäre auch möglich, indem die Managementanwendung selbst entsprechend angepaßt wird. Dies hätte allerdings zur Folge, daß solche Anpassungen bei jeder Änderungen der Implementierung des Abrechnungssystems (Austausch von Komponenten) durchgeführt werden müßten [Radi 03]. Die Lösung mittels einer Integrationsschicht umgeht dieses Problem, entspricht damit der Anforderung **PRO 3** und kommt der Anforderung **PRO 4** entgegen. Die Anforderung **ADM-3** betreffend der Invarianz des Management bezüglich der Implementierung des Abrechnungssystems wird ebenfalls angesprochen. Mit Hilfe der Integrationsschicht kombiniert mit einer policy-basierten Steuerung wirken sich Veränderungen in der Implementierung des Abrechnungssystems kaum noch aus, da:

1. sich beim Austauschen von Abrechnungskomponenten die Managementschnittstelle nicht ändert (Integrationsschicht) und
2. die zur Steuerung benutzten Policies sich nur auf diese Schnittstelle beziehen — und damit auch nicht verändert werden müssen.

Die durch verschiedenen Funktionsumfang von ,,alter” und ,,neuer” Komponenten entstehenden Unterschiede können durch entspechende Implementierung der Agenten ausgeglichen werden.

Aufgaben der Agenten

Die Schnittstelle der Abrechnungskomponenten gliedert sich in eine Benutzerschnittstelle für die Kernaufgaben der Komponente und eine Managementschnittstelle. Insgesamt bietet die Komponente jedoch eine einzige API (Application Programming Interface). Die „herkömmlich" eingesetzten Gateways werden zur Verbindung heterogener Komponenten benutzt; ihre wichtigste Aufgabe ist es, die Interoperabilität zu gewährleisten. Managementaspekte werden hierbei meist nicht betrachtet, sodaß die Gateway den Managementteil der API der Komponente außer acht läßt. Als Folge wird das Management der Komponenten unabhängig von den Gateways realisiert. Die Heterogenität der Komponenten widerspiegelt sich dabei auch in ihrem Management: jede Komponente kann offensichtlich nur mittels der von ihr unterstützten Schnittstellen und Managementprotokollen angesprochen werden, sodaß unter Umständen eine Vielzahl von Managementskripten und/oder -werkzeugen von Nöten ist. Erschwerend kommt hinzu, daß auch die Gateways selbst bezüglich Management die beschriebenen Probleme aufweisen können.

Die Aufgaben der Agenten der Integrationsschicht gliedern sich in zwei Bereiche:

- Replizierung der Gateway-Funktionalität
- Management

Ersetzt man eine Gateway durch einen Integrationsagenten, muß die Gateway-Funktionalität im Agenten implementiert werden. Im einfachsten Fall werden dabei Operationsaufrufe weitergeleitet oder in ähnliche Aufrufe basierend auf einer anderen Verteilungsplattform übersetzt. Komplexere Szenarios könnten das Filtern oder Konvertieren von Daten oder die Implementierung von Protokollen erfordern. Um die Funktionalität der ihm entsprechenden Gateway zu realisieren, benutzt ein Agent die Benutzerschnittstelle (in Abbildung 4.5 mit „user" gekennzeichnet) der Abrechnungskomponente.

Die Abbildung der herstellerspezifischen Managementschnittstellen der Abrechnungskomponenten, auf einheitliche, durch die Managementanwendung vorgeschriebene gestaltet sich ähnlich unterschiedlich. Um das Management der ihm zugeordneten Abrechnungskomponente zu ermöglichen, benutzt ein Agent die Managementschnittstelle (in Abbildung 4.5 mit „mgmt" gekennzeichnet) der Abrechnungskomponente. Je verschiedener diese von der einheitlichen Managementschnittstelle ist, die von der Managementanwendung erwartet wird, desto mehr Implementierungsaufwand muß bei der Erstellung der Agenten betrieben werden.

Die Wiederverwendung von Agententeilen kann den Implementierungsaufwand verringern — ein umsichtiger Entwurf der Software vorausgesetzt. Wird die innerhalb der Agenten implementierte Funktionalität (Übersetzung von Schnittstellen, Protokolle usw.) in Module gekapselt, kann bei der Entwicklung neuer Agenten auf diese Module zurückgegriffen werden. Dies ist allerdings eine Aufgabenstellung des Software-Engineering im Rahmen des Entwurfs; an dieser Stelle wird nicht weiter darauf eingegangen.

Probleme bezüglich des Management der Agenten selbst sollten nicht auftreten, da es selbstverständlich ist, daß diese für das Management ihrer selbst auch eine einheitliche Schnittstelle anbieten.

4.2.2 Prozeßorientierung als Werkzeug zur Modularisierung

Die Aufteilung des Abrechnungsvorgangs in Teilprozesse erlaubt es, den Abrechnungsvorgang kontrollierbar zu gestalten. Dies geschieht durch die Einführung einer logischen Struktur auf einer höheren Abstraktionsebene. Dieses Konzept bietet allerdings auch besondere Mittel für den Entwurf einer Sprache zur Spezifikation von Policies, sowie zur Verwaltung der Policies durch das Managementsystem.

Der Einsatz policy-basierter Verwaltung zielt einerseits darauf, Vorgänge, die vielfältige Systeme beeinflussen, als Regeln in einer einheitlichen Notation festzuhalten. Andererseits soll durch den Einsatz von Policy-Datenbanken (sogenannten *policy repositories*) eine Wissensbasis geschaffen werden, die Administratoren zur Verfügung stehen soll.

Betrachtet man solch ein *repository* als eine unstrukturierte Ablage einer großen Menge von Policies, so kann sich die Suche nach einer Policy für eine konkrete Problemstellung als schwierig erweisen. Es wird deshalb erforderlich, die Gesamtmenge der Policies zu strukturieren, indem man die Policies in Kategorien einteilt. Eine Einteilung nach Problemstellung erscheint zweckmäßig, da sie häufigen Anwendungsfällen — wie etwa der Suche nach Policies für eine bestimmte Aufgabe — gerecht wird.

Die Teilprozesse der Abrechnung bieten sich als grundlegende Kategorien für die Einteilung der Policies an, da sie wohldefinierten Abläufen des Abrechnungsvorgangs entsprechen. Zusätzlich besitzt jeder Teilprozeß eine Menge von Entitäten, Aktionen und Ereignissen, die (teilweise sogar exklusiv) dem Teilprozeß zugeordnet werden können.

Modularer Aufbau von Policies

Je nach Mächtigkeit der benutzten PDL werden Mittel zur Gruppierung von Policyteilen (z. B. *targets*), Unterstützung von Rollen und Domänen etc. zur Verfügung gestellt. Das hat zur Folge, daß es nicht nur Sinn macht, ganze Policies zu speichern, sondern auch solche zusammengesetzte Bestandteile. Diese Bestandteile können dann von verschiedenen Policies gleichzeitig genutzt werden (beispielsweise kann eine Gruppe von *targets* in mehreren Policies benutzt, muß jedoch nur einmal definiert werden).

Eine Erweiterung dieses Ansatzes ist die Bereitstellung von Policyteilen in Bibliotheken. Eine Policy kann dann ganz oder teilweise aus schon existierenden Teilen zusammengesetzt werden. Die Vorteile dieser Vorgehensweise liegen auf der Hand: einerseits erleichtert sie die Erstellung neuer Policies, andererseits vermindert sich die Größe der Policy-Datenbank — die Policies selbst werden leichter zu verwalten.

Eine Policy kann in der Regel einem der Teilprozesse des Abrechnungsvorgangs zugeordnet werden. Umgekehrt kann auch ein Teilprozeß einen Rahmen für die Erstellung neuer Policies bieten. Eine Policy für einen bestimmten Teilprozeß, die bei ihrer Erstellung Policyteile aus einer Bibliothek wiederverwendet, wird voraussichtlich Teile benutzen, die eben diesem Teilprozeß zugeordnet werden können. Somit kann in vielen Fällen eine Policy als eine Aggregation von als Policy-Bausteine definierten Merkmalen (Ereignisse, Aktionen, Entitäten) eines Teilprozesses beschrieben werden.

Konsequenzen für die Benutzerschnittstelle der Managementanwendung

Die Umsetzung dieser Konzepte in die Praxis muß insbesondere den Nutzen für Policy-Administratoren (den hauptsächlichen Benutzern dieses Merkmals) berücksichtigen. Das folgende Szenario skizziert einen Ablauf, der für die Managementanwendung als typisch angesehen werden kann.

Szenario Ein Administrator will eine Policy für den Betrieb der Charging-Komponente (Gebührenabrechnung) definieren. Diese Policy kann also dem *charging*-Teilprozess zugeordnet werden. In seiner Managementanwendung wählt er diesen Teilprozeß und bekommt die in der Datenbank vorhandenen, möglicherweise inaktiven, Policies angezeigt. Findet er eine fertige, passende Policy, kann diese aktiviert werden. Ist eine passende Policy noch nicht vorhanden, kann sich der Administrator die Aktionen, Ereignisse, Entitäten, Domänen usw. anzeigen lassen, die für diesen Prozeß relevant sind. Er kann die entsprechenden Bausteine auswählen und zu einer Policy zusammenstellen. Sollte ein benötigter Baustein nicht vorhanden sein, oder nicht ohne Modifikationen übernommen werden können, kann der Administrator ihn manuell modifizieren oder einen neuen definieren. Der neue Baustein wird — falls erwünscht — für zukünftige Anwendung in die Bibliothek aufgenommen.

Anforderungen aus dem Szenario Die Anforderungen, die dem Szenario entnommen werden können, ergeben Anforderungen an die PDL, mit der die Policies notiert werden, an den Funktionsumfang des Managementsystems, sowie an die Art und Weise, in der die Policies gespeichert werden. Die Anforderungen an die PDL beschränken sich im betrachteten Fall auf drei:

- die Angabe von *freien* (d. h. nicht innerhalb einer Policy angegebenen Policybestandteilen) muß möglich sein
- es muß möglich sein, solche freien Bestandteile in einer Policy zu referenzieren, statt sie explizit zu notieren
- jeder Eintrag in der Bibliothek (Policies als auch freie Bestandteile) muß einem oder mehreren Teilprozessen zugeordnet werden können

Zusätzlich werden die folgenden Anforderungen an das Managementsystem und ihre Vorrichtung für persistente Speicherung gestellt:

- eine Suche mit dem Teilprozeß als Suchschlüssel muß unterstützt werden
- vorhandene Bestandteile sollen kopiert und modifiziert werden können

Generische Policy-Bibliothek

Die Integration der in den obigen Abschnitten diskutierten Eigenschaften in einer Implementierung (generische Schnittstellen zur Managementanwendung, Modularität der Policies, Prozeßzugehörigkeit als Attribut von Policies und Policy-Bauteilen) legt den Gedanken nahe, ein Managementsystem von vornherein mit einer Bibliothek generischer Policy-Bestandteile auszustatten. Diese würden Vorlagen für die Erstellung von Policies darstellen, so daß Policies von Anfang an mit Hilfe vorgefertigter Bausteine zusammengestellt werden könnten.

Um geeignete Vorlagen für jeden Teilprozeß zur Verfügung stellen zu können, müssen die Merkmale der Teilprozesse vollständig erarbeitet sein. Es ist also eine Analyse eines jeden Teilprozesses notwendig, um die ihm zugehörigen Entitäten zu identifizieren, sowie die für den laufenden Betrieb notwendigen Aktionen und Ereignisse zu erarbeiten. Eine Analyse dieser Art würde den Umfang dieser Arbeit sprengen; die in Anhang A dargestellten, generischen Merkmale der Teilprozesse können allerdings für die Erarbeitung von Anforderungen an eine Policy-Bibliothek hinzugezogen werden.

Aus der obigen Diskussion ergeben sich die folgenen Anforderungen an die Implementierung der Managementanwendung:

- Die Definition von Policybestandteilen kann außerhalb einer Policy-Definition erfolgen.
- Der Teilprozeß, dem eine Policy oder ein Policy-Baustein zugeordnet ist, kann in der Policy-Sprache als Attribut eines Policy-Bausteins oder einer Policy formuliert werden.
- Die Bestandteile einer Policy können sowohl explizit angegeben werden, als auch Referenzen an frei definierte Bausteine sein.
- Es kann festgelegt werden, ob ein bestimmter Baustein (oder eine Policy) zur Bibliothek des Systems gehört. Sollte die letzte Policy gelöscht werden, die einen Baustein referenziert, so soll dieser nicht gelöscht werden, falls dies der Fall ist. Auf diese Weise ist es möglich, die referentielle Integrität des *Repository* zu wahren, ohne die zur Bibliothek gehörenden Bausteine gesondert zu speichern.

4.3 Analyse der Ausdrucksmächtigkeit der Policy-Sprache

Die minimalen Anforderungen an eine PDL beschränken sich auf die Angabe von Ereignissen und Aktionen. Werden keine Ereignisse angegeben, ist nicht entscheidbar, wann eine Policy ausgewertet werden soll; fehlen die Aktionen, bleibt die Auswertung einer Policy ohne Konsequenzen. Die Auswertung einer Policy und die Ausführung ihrer Aktionen geschieht in einem bestimmten Kontext. Dieser bestimmt die Bedingungen für die Ausführung der Aktionen und die Ziele dieser Aktionen.

Um die Ausführung der Aktionen bestimmten Zielobjekten zuzuordnen, kommen drei Ansätze in Frage:

1. Das Zielobjekt einer Aktion kann statisch als Teil der Aktion angegeben werden.
2. Eine statische Liste mehrerer Ziele kann in der Policy angegeben werden; die Aktionen werden für alle angegebenen Zielobjekte ausgeführt.
3. Das Ziel einer Aktion kann als Attribut eines Ereignisses implizit gegeben sein.

Die im ersten und dritten Ansatz benutzten Semantiken des Zielobjekts stellen Anforderungen an die Spezifikation der Aktionen und Ereignisse, sowie an die Laufzeitumgebung des Managementsystems. Alle drei Ansätze erfordern Unterstützung durch die Policy Definition Language selbst.

Der erste Ansatz erfordert die explizite Angabe des Zielobjekts einer Aktion. In der PDL muß also die Möglichkeit einer derartigen Angabe vorgesehen sein. Zusätzlich ist ein Dienst erforderlich, der den Namen des Objekts in eine Referenz auf das Objekt auflösen kann.

Der zweite Ansatz fordert die explizite Angabe von (möglicherweise multiplen) Zielen innerhalb einer Policy. Der Begriff eines Ziels (*target*) reduziert sich dabei auf den im ersten Ansatz beschriebenen.

Der dritte Ansatz stellt Anforderungen an den Aufbau von Ereignissen. Es ist notwendig, Ereignisse selbst als Objekte zu modellieren, sodaß Attribute transportiert werden können. Das Managementsystem muß dabei das Marshalling und Unmarshalling dieser Objekte unterstützen. Zusätzlich muß die PDL eine Bezugnahme auf die Attribute des Ereignisobjekts erlauben, das die Ausführung der Policy ausgelöst hat.

Weitere grundlegende Sprachkonstrukte

Bedingungen Um eine flexible Zuordnung von Aktionen zu Ereignissen zu gewährleisten, müssen die Vorkommnisse im gemanagten System feingranular kategorisiert werden. Dies kann theoretisch durch eine Spezialisierung der Ereignistypen erreicht werden, derart daß für jedes mögliche Vorkommnis jeweils ein besonderes Ereignis generiert wird. Dabei nimmt man allerdings den Nachteil in Kauf, daß für jede Klasse von Vorkommnissen eine große Menge von Ereignistypen definiert werden muß. Es ist außerdem zu befürchten, daß Änderungen am gemanageten System auch Änderungen der Menge der Ereignistypen nach sich ziehen.

Definiert man statt dessen eine kleinere Anzahl genereller Ereignistypen, so muß zwischen den verschiedenen Untertypen auf anderem Wege unterschieden werden. Da es sich bei den Ereignissen, wie oben angedeutet, um Objekte handelt, bietet es sich an, solche Unterscheidungen anhand von Attributswerten vorzunehmen. Ein bestimmter Ereignisuntertyp ist dann gegeben, wenn die Belegung der Attribute des Ereignisobjekts ein für den Untertyp festgelegtes Muster aufweist. Mit Hilfe von aussagenlogischen Prädikaten kann die Belegung ausgewertet werden. Dieser Ansatz zeigt die Notwendigkeit auf, Bedingungen für die Ausführung der Aktionen einer Policy formulieren zu können. Es wird daher erforderlich, die Mächtigkeit der zu unterstützenden Ausdrücke festzulegen. In [MESW 01] wird vorgeschlagen, konjunktive und disjunktive Normalformen von aussagenlogischen Formeln als Format für Bedingungen anzugeben. Dabei wird davon ausgegangen, daß eine ausreichende Ausdrucksmächtigkeit gegeben ist (da jede aussagenlogische Formel in disjunktiver Normalform angegeben werden kann), und gleichzeitig das in der Sprache unterstützte Konzept für Bedingungen implementierbar bleibt. Es ergibt sich also folgende Sicht auf Ereignisse:

> *Ein Ereignis besteht aus einem Ereignisobjekt und einer Menge von Bedingungen in einer Normalform. Stimmt der Typ des Ereignisobjekts mit einem der in der Policy angegebenen überein und ist der Wahrheitswert der Normalform* wahr*, so gilt das Ereignis als eingetreten und die Aktionen der Policy werden ausgeführt.*

Subjekt Eine Policy *betrifft* eine Entität (oder Menge von Entitäten), wie auch im allgemeinen eine Regel für bestimmte Vorgänge, Sachen, Personen formuliert werden kann. Die Entität, für die eine Policy spezifiziert wird, wird *Subjekt* genannt und als Bestandteil einer Policy aufgenommen.

Nach Hinzunahme dieser Erweiterung kann eine Policy unter Berücksichtigung der oben angegebenen Sicht auf Ereignisse wie folgt definiert werden:

> *Eine Policy besteht aus einem Subjekt, einer Menge von Zielobjekten, einer Menge von Ereignissen und einer Menge von Aktionen. Tritt ein Ereignis ein, werden die Aktionen auf den Zielobjekten ausgeführt.*

Diese Definition berücksichtigt die für eine Policy angegebenen Bedingungen implizit als Teil der Ereignisse. Für die Umsetzung in einer Sprache ist eine Aufspaltung von Ereignissen in Nachricht und

Bedingung zweckmäßig. Die Bedingung wird in den weiteren Teilen dieser Arbeit als eigenständiges Konstrukt betrachtet. Die im weiteren als „Ereignisse" (Event etc.) bezeichneten Konstrukte übernehmen den verbliebenen Teil der Semantik — nämlich den von Nachrichten.

Sprachkonstrukte zur Verwaltung von Policies

Betrachtet man ein umfangreiches zu verwaltendes System, für das eine große Anzahl Policies angegeben wurde, steht die Notwendigkeit von Verwaltungsmechanismen für die Policies selbst außer Frage. Es ist davon auszugehen, daß mehrere Personen gleichzeitig oder zeitlich versetzt Policies erstellen, modifizieren, aktivieren oder deaktivieren. Deshalb ist es erforderlich, die Policies mit Merkmalen zur Dokumentation dieser Veränderungen zu versehen — ähnlich der Attribute der Dateien eines Dateisystems:

- Der Zeitpunkt der Erstellung der Policy
- Die Person, welche die Policy erstellt hat
- Der Zeitpunkt der letzten Änderung der Policy
- Die Person, die die Policy zuletzt modifiziert hat

Gruppierungsmechanismen

Als Gruppierungsmechanismen kommen für die Policy-Sprache Gruppen, Rollen und Domänen in Frage. Die drei genannten Konzepte sind in den meisten Fällen gegeneinander austauschbar, was ihre Mächtigkeit betrifft. Eine Implementierung würde also wohl mit nur einem einzigen auskommen. Jedes von ihnen besitzt allerdings „traditionelle" Anwendungsgebiete, in denen man erwartet, einen bestimmten Gruppierungsmechanismus vorzufinden.

Rollen

Rollen werden dort angewendet, wo von der Identität einer Entität abstrahiert werden soll. Darüberhinaus können sie dazu genutzt werden, um Gruppierungen mit inhärenter Semantik vorzunehmen. In der PDL ergeben sich deshalb für Rollen zwei mögliche Anwendungsbereiche: sie können einerseits als Argumente beziehungsweise Teile einer Policy angegeben werden. Andererseits können Policies selbst Rollen besitzen, anhand derer sie verwaltet werden können.

Die Angabe von Rollen in Teilen einer Policy bietet sich insbesondere an, wenn auf organisatorischer Ebene — beispielsweise aus Gründen der Rechtevergabe — bereits Rollen vergeben wurden. Die Unterstützung von Rollen in der PDL würde unter solchen Umständen die Integration des Managementsystems in eine bestehende Infrastruktur erleichtern. Der hier angesprochene Einsatzbereich bezieht Rollen für Benutzer und Systeme ein, beispielsweise *Benutzer mit Druckquota* oder *Webserver*. In der PDL sollte daher die Substitution realer Subjekte und Zielobjekte durch Rollen ermöglicht werden. So formulierte Policies erfassen alle Inhaber einer Rolle als Subjekt beziehungsweise Zielobjekt. Die Einführung von Rollen für Zielobjekte ist unproblematisch, da von vornherein eine Angabe multipler Ziele vorgesehen ist. Was das Subjekt einer Policy betrifft, muß in der Implementierung der PDL beachtet werden, daß die Angabe einer Rolle eine Multiplizität der Subjekte zur Folge hat.

Ein Einsatz von Rollen wäre auch bezüglich anderer Bestandteile der Policies denkbar — ein Beispiel wäre die Zuweisung von Rollen an Ereignisse oder Aktionen. Es ist allerdings nicht zu erwarten, daß solch eine Rollenvergabe eine Entsprechung in anderen Teilen des Systems hat. Sie würde also lediglich eine Möglichkeit der Gruppierung von Elementen darstellen; solch eine Gruppierung von Policy-Bausteinen sollte allerdings mit dedizierten Konstrukten realisiert werden, die nicht notwendigerweise eine Rollensemantik besitzen müssen.

Die Umsetzung des Rollenkonzepts in der PDL kann ähnlich realisiert werden, wie die Benutzergruppen in Unix und ähnlichen Betriebssystemen. Letztere werden dazu benutzt, um Benutzern den Zugriff auf Ressourcen abhängig von ihrer Gruppenzugehörigkeit zu erlauben oder zu verweigern. Der Systemverwalter richtet unter Unix Gruppen ein und weist jedem Benutzer Mitgliedschaft in null[1] oder mehreren Gruppen zu. Der Benutzer erlangt dadurch Zugriff auf die zur Gruppe gehörenden Ressourcen. Der Namensraum der Rollen ist „flach"; sie werden also nicht hierarchisch organisiert. Die Rollen für *Subject* und *Target* einer Policy können analog spezifiziert werden. Es ergeben sich dadurch folgende Merkmale einer Rolle:

- Eine Rolle besteht aus einem eindeutigen Rollennamen und einer textuellen Beschreibung ihrer Bedeutung.
- Rollen werden unabhängig von Policies definiert.
- Jedem „rollenfähigen" Konstrukt können null oder mehrere Rollen zugewiesen werden
- Wird eine (oder mehrere) Rolle(n) als Policybaustein benutzt, schließt dies die explizite Angabe einer Entität in diesem Baustein aus.
- Wird eine Rolle zugewiesen, müssen nicht notwendigerweise Rolleninhaber existieren. In der Implementierung der PDL muß dieser Fall gesondert behandelt werden.
- Die Angabe einer nicht definierten Rolle gilt als Fehler.

Zusammengefaßt kann die Unterstützung von Rollen für Teile von Policies folgendermaßen ausgedrückt werden:

> Eine Entität kann durch die Angabe einer oder mehrerer vordefinierten Rollen ersetzt werden. Dies entspricht der Angabe der Vereinigungsmenge der Inhaber aller angegebener Rollen.

Die Unterstützung von Rollen wird sowohl aus Kundensicht (**KUN-3**), als auch aus Providersicht gefordert (**PRO-2**).

Gruppen Um die Anzahl der für die Verwaltung eines Systems notwendigen Policies zu reduzieren, kann eine Gruppierung mancher Elemente der Sprache sinnvoll sein. Ebenso kann es von Vorteil sein, Policies anhand ihres Zwecks oder ihrer Eigenschaften in Gruppen einteilen zu können. Gruppen stellen einen generischen Gruppierungsmechanismus dar. Eine Gruppe trägt, anders als eine Rolle, weder einen Namen, noch eine Semantik bezüglich des zu verwaltenden Systems. Sie stellt einen

[1] In gängigen Installationen ist jeder Benutzer Mitglied mindestens einer Gruppe.

generischen Gruppierungsmechanismus dar, der in der PDL auf syntaktischer Ebene verfügbar sein soll und zur Modularisierung von Policies, sowie zur Verwaltung von (in Gruppen zusammengefaßten) Policy-Mengen instrumentalisiert werden kann.

Domänen werden dazu benutzt, um topologisch zusammengehörende Objekte zu gruppieren. Sie werden in Policy-Sprachen, meist in pfadähnlicher Form, zur Angabe von Subject und Target benutzt (z. B. in [DDLS 00]). Domänenausdrücke können ähnlich wie Pfadangaben eines Dateisystems in *Domäne* und *Entität* gegliedert werden. Wird nur der *Pfad*-Teil angegeben, bezieht sich der Domänenausdruck auf die Menge der Entitäten, die in der durch den Pfad angegebenen Domäne vorhanden ist. Die Unterstützung von Domänenausdrücken in der Sprache unterstützt also die Anforderung **KUN-6**, da sie die Angabe von topologisch bestimmten Gruppen in der Sprache erlaubt.

4.4 Zusammenfassung

In diesem Kapitel wurden die erforderlichen Konzepte für den Entwurf der Managementanwendung entwickelt. Die Teilprozesse der Abrechnung wurden entsprechend ihrer Eignung zur automatischen Verarbeitung kategorisiert; außerdem wurde ein weiterer Teilprozeß *error* zur Lokalisierung der Behandlung von Fehlern eingeführt. Zur Erlangung einer prozeßorientierten Sicht auf das Abrechnungsmanagement wurde eine Integrationsschicht betrachtet, die herstellerspezifische Charakteristika der Abrechnungskomponenten auf eine einheitliche Managementschnittstelle abbildet. Mittels dieser, sowie mit Hilfe policy-basierter Techniken kann ein flexibles Management der Abrechnung in Aussicht gestellt werden. Die erforderliche Ausdrucksmächtigkeit einer Policy-Sprache wurde betrachtet, ebenso wie sich aus der Prozeßsicht ergebende Nebenbedingungen für eine solche Sprache. Die in diesem Kapitel betrachteten Konzepte bilden einen Rahmen für die Spezifikation einer Policy-Sprache und für den Entwurf einer Managementanwendung.

Kapitel 5

Die Policy Definition Language

Dieses Kapitel enthält die Spezifikation der *Policy Definition Language*. Mit Hilfe der Analyse der Ausdrucksmächtigkeit der Sprache in Kapitel 4 kann die Struktur der Sprachelemente festgelegt und ihre Semantik erklärt werden. Anschließend wird die Grammatik der PDL in der Erweiterten Backus-Naur-Form (EBNF) angegeben. Dabei wird die Funktion einiger Sprachelemente detaillierter ausgearbeitet.

5.1 Entwurf der Sprachelemente

Die folgende Übersicht der Elemente der PDL zeigt ihre allgemeine Struktur ohne Angabe einer konkreten Grammatik und weist auf ihre vorgesehene Anwendungsweise hin. Ausgehend von den in Kapitel 3 identifizierten Feldern der Policy werden die wichtigsten Sprachelemente grob spezifiziert. Es werden keine Details der Elemente angesprochen, noch werden alle in der EBNF vorkommenden Konstrukte behandelt.

5.1.1 Referenzierbarkeit

Die PDL soll die Modularisierung von Policies mit komplexen Sprachkonstrukten unterstützen und fördern. Gleichzeitig soll für einfache Konstrukte eine „lokale" Definition gestattet werden.

Dieser Ansatz strebt die Entstehung einer Datenbasis (Repository) von Policy-Bausteinen im laufenden System an, die

- wiederverwendbare Policy-Teile enthält,
- einen nur geringen Anteil „trivialer" oder nicht wiederverwendbarer Bausteine enthält
- eine Kategorisierung anhand von Typen, Deskriptordaten und Teilprozessen der Abrechnung erlaubt

Das *Repository* enthält global definierte Policy-Bausteine, sowie Policies, die ihrerseits lokal definierte Bausteine, sowie Referenzen auf global definierte Bausteine enthalten können.

Die Grundlage für den modularen Aufbau von Policies ist die *Referenzierbarkeit ihrer Teile*. Die Bausteine einer Policy (Entitäten, Aktionen etc.) sollen entweder *lokal*, d. h. innerhalb der Policy-

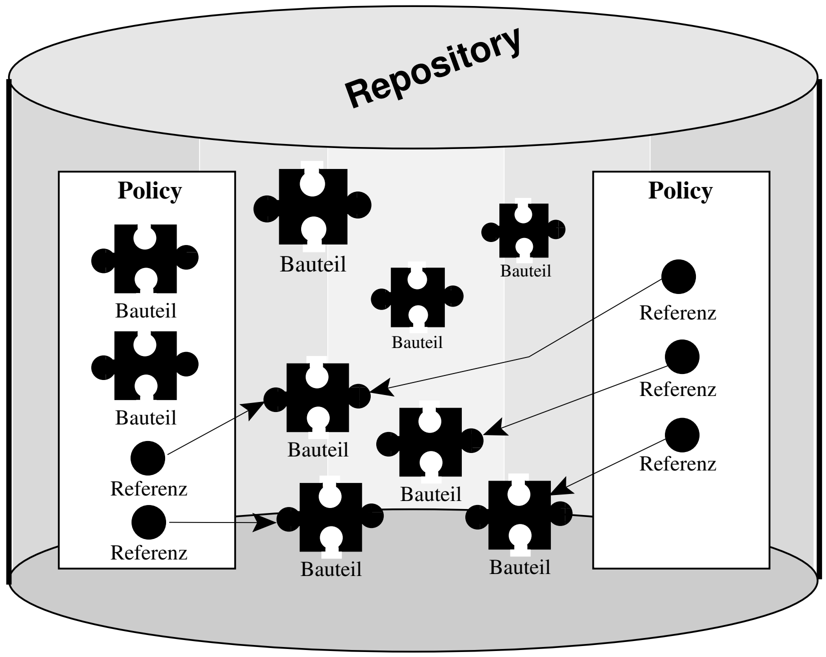

Abbildung 5.1: Repository für Policies und Policy-Bausteine

Definition angegeben werden können, oder *global*, als selbständige Teile. Alternativ zu der lokalen Definition eines Bausteins, sollen in einer Policy *Referenzen auf globale Bausteine* angegeben werden können (vgl. Abbildung 5.1). Dieser Mechanismus soll den Autoren von Policies die Wiederverwendung von Bauteilen nahelegen.

Um Referenzierbarkeit von Policies und Policy-Bausteinen zu realisieren, muß ein Mechanismus zur ihrer eindeutigen Identifizierung in die Sprache eingebettet werden. Dies kann durch die einfache Angabe einer ID der Policy bzw. des Bausteins geschehen. Die ID wird dabei als eindeutig bestimmt vorausgesetzt. Diese eindeutige ID kann zur Dereferenzierung in einer Policy einfach zitiert werden; das Laufzeitsystem ist dann dafür verantwortlich, den Baustein mit dieser ID zu finden und als Teil der Policy einzubinden.

Es mag an dieser Stelle als erstrebenswert erscheinen, den Aufbau der Policies aus global definierten Teilen in jedem Fall zu erzwingen, um die Unterscheidung zwischen lokalen und globalen Teilen zu eliminieren und gleichzeitig sämtliche Policy-Bausteine zur Wiederverwendung freizugeben. Es bestünde jedoch dabei die Gefahr des „Versinkens” brauchbarer Bausteine in der großen Menge der nur formell wiederverwendbaren. Deshalb erscheint es sinnvoller, den Policy-Autor an der Entscheidung bezüglich lokaler oder globaler Definition teilhaben zu lassen, statt diese in der Spezifikation der PDL zu treffen.

Die Zuordnung der Policies und Policy-Bausteine zu Teilprozessen der Abrechnung kann durch die Angabe eines Prozeßnamens innerhalb der Deklaration von Policy oder Baustein geschehen. Die Prozeßnamen sollten — spätestens in der Implementierung — festgelegt sein. Ihre Festlegung bereits im Entwurf der Sprache würde die Anwendbarkeit der Sprache auf andere Managementbereiche als Abrechnung verringern, ohne dabei einen konzeptionellen Vorteil einzubringen.

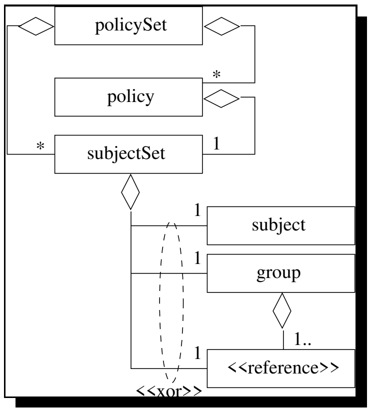

Abbildung 5.2: Alternativen zur Belegung eines Policy-Feldes

Aus dem Szenario zur werkzeuggestützten, prozeßorientierten Erstellung von Policies in diesem Kapitel leitet sich der Bedarf der Zuordnung einer Policy an *alle* oder *keine* Prozesse (als Alternativen zur ausdrücklichen Angabe von Prozeßnamen) ab. Sie sind für Elemente anwendbar, deren Zuordnung zu Prozessen unbekannt oder irrelevant ist (*keine*), sowie für Elemente bezüglich derer keine Einschränkungen durch Prozeßgrenzen gegeben sind (*alle*).

Ein Policy-Feld kann mit einem explizit in der Policy angegebenen Element, mit einer Referenz auf ein global angegebenes Element oder mit einer Gruppe von Referenzen auf solche Elemente belegt werden. Abbildung 5.2 zeigt die Beziehungen zwischen der Policy, dem Policy-Feld (Subject) und den einander ausschließenden, möglichen Belegungen des Feldes. Die Möglichkeit, Subjects global, also außerhalb einer Policy, zu definieren, wird durch das *Policy Set* unterstützt, das als Behälter für sowohl Policies als auch freie Bestandteile fungiert.

5.1.2 Gruppen

Gruppen stellen einen Gruppierungmechanismus für die Elemente der PDL selbst dar. Es können Policies gruppiert werden, wie auch Aktionen, Events usw. Eine Gruppe besteht ihrerseits aus Referenzen auf frei bzw. global definierte Elemente; sie ist also eine Reihung von Referenzen. Eine Gruppe kann in der PDL ein explizit angegebenes Element durch mehrere ersetzen. Sie ist in dieser Hinsicht die einzige Möglichkeit, ein Policy-Feld mit mehreren Elementen des entsprechenden Typs (z. B. Subject) zu belegen. Beispielsweise kann eine Policy eine Gruppe von Events enthalten; sie würde ausgewertet, wenn eines der in der Gruppe spezifizierten Ereignisse eintreten würde.

Die informelle Syntax für eine Gruppe ist also:

Gruppe:	`Referenz, Referenz, ...`

5.1.3 Subject und Target

Sowohl Subject als auch Target repräsentieren Entitäten eines Prozesses. Als solche können sie mittels Rollen und Domänen angegeben werden. Ein Subject oder Target kann entweder als Entität angegeben werden (z. B. mit dem Namen eines repräsentierenden Integrationsagenten) oder als Rolle. Die Rolle ersetzt dabei die Entität, indem sie *alle* ihre Rolleninhaber als Subject bzw. Target spezifiziert.

Entitäten können in Domänen organisiert sein. Aus diesem Grund kann sowohl für eine Entität als auch für eine Rolle eine Domäne angegeben werden. Diese Angabe lokalisiert eine Entität und schränkt im Fall einer Rolle die Rolleninhaber auf die Domäne ein.

Target und Subject können also zwei Formen annehmen:

Subject/Target: `Domäne Entität`

oder

Subject/Target: `Domäne Rolle`

5.1.4 Action

Die für die Angabe von Aktionen erforderlichen Sprachelemente müssen auf die Plattform abgestimmt sein, mittels derer die Aktionen tatsächlich ausgeführt werden. In diesem Fall wird von einer objektorientierten Plattform ausgegangen, die die Abbildung der Aktionen auf Methodenaufrufe unterstützt. Infolgedessen besteht die Aktion aus

- einem Objekt, dessen Methode aufgerufen wird
- dem Methodennamen
- den Parametern der Methode

Das Objekt der Aktion wird als Name angegeben. Dieser kann, ähnlich wie bei einer Entität, der Name eines Integrationsagenten oder eines anderen Objekts sein. Zur Auflösung von Namen kann in der Implementierung ein entsprechender Dienst benutzt werden.

Methodenaufrufe können Rückgabewerte haben, die Werte zur weiteren Verarbeitung liefern. Das Berechnungsmodell, das einem policy-basierten, deklarativen, ereignisgesteuerten System dieser Art zugrundeliegt, macht die Weiterverarbeitung der Rückgabewerte problematisch. Es wären Konzepte notwendig, die anhand von Rückgabewerten eine weitere Verarbeitung im *Policy Enforcer* bestimmen würden. Andererseits kann beobachtet werden, daß der Zweck der Rückgabewerte vieler Methoden sich auf einen Hinweis bezüglich ihrer erfolgreichen oder fehlgeschlagenen Ausführung beschränkt. Aus diesem Grund werden in einer Aktion zwei Aufrufe erlaubt: ein Aufruf, der die für die Policy spezifizierte Aktion ausführt, und einer, der nur dann ausgeführt wird, wenn der erste fehlschlägt. Auf diese Weise ist es möglich, auf fehlgeschlagene Aktionen zu reagieren, ohne die strikt ereignisgesteuerte Vorgehensweise aufzugeben. Demzufolge kann eine Aktion angegeben werden als

Aktion: `Standardaktion Fehlerbehandlungsaktion`

Eine Aktion kann statt der Ausführung eines Methodenaufrufs ein Ereignis generieren und im System propagieren. Dieser Mechanismus kann außer zur Meldung fehlgeschlagener Methodenaufrufe auch zur Verkettung von Policies benutzt werden, indem eine Policy ein Ereignis erzeugt, das in einer anderen Policy spezifiziert wird. Ereignisse können Attribute transportieren, die auf die Herkunft des Ereignisses, seiner Dringlichkeit usw. hinweisen. Zur Erzeugung solcher Ereignisse können — wie bei Methodenaufrufen — Parameter spezifiziert werden.

Aktionen können also jeweils zwei der nachfolgenden Strukturen enthalten :

Methodenaufruf: `Objekt Methode Parameter Parameter ...`

Ereigniserzeugung: `Ereignis Parameter Parameter ...`

Die Parameter eines Methodenaufrufs besitzen Namen, Typen und Werte. Um die Implementierung zu erleichtern, können die Namen der Parameter spezifiziert werden, auch wenn sie implizit durch die

Reihenfolge und den Typ der Parameter gegeben sind. Die Struktur eines Parameters ist also:

Parameter:	`Name Typ Wert`

Die Werte von Parametern können konstant (als Literale) in der Policy angegeben werden. Sie können aber auch Rückgabewerte von Methodenaufrufen, oder Attributswerte eines Objekts darstellen. Rückgabewerte können in derselben Weise wie die Methodenaufrufe beschrieben werden; sie sind schließlich ihr Produkt. Attributswerte, hingegen, bestehen lediglich aus der Angabe eines Objekts und einem Attributsnamen, etwa:

Attribut:	`Objekt Attribut`

Event

Zur Spezifikation von Ereignissen, auf die hin eine Policy ausgewertet wird, genügt ein (eindeutiger) Name für das Event. Anders als bei der Generierung von Ereignissen innerhalb von Aktionen, ist es bei ihrer Erkennung nicht von Bedeutung, ob mit einem Ereignis Daten transportiert werden. Die Form einer Eventdeklaration gestaltet sich also denkbar einfach:

Event:	`Ereignisname`

5.1.5 Constraint

Die in einer Policy angegebenen Bedingungen entscheiden darüber, ob ihre Aktionen ausgeführt werden, oder nicht. Es ist also naheliegend, daß diese Entscheidung aufgrund des *Zustandes des Systems* getroffen wird. Dazu gehören sowohl Angaben zur Zeit, als auch zahlreiche andere Bedingungen, die an beteiligte Komponenten gestellt werden können. Konkret gesehen impliziert dies die Auswertung von Prädikaten, die eine Aussage über den Vergleich zweier Werte machen. Beispielsweise kann die Systemzeit mit einer in der Policy angegebenen Uhrzeit verglichen werden, oder geprüft werden, ob ein bestimmtes Objekt (etwa ein Integrationsagent) bei einem Namensdienst registriert ist. Da die Auswertung von Ausdrücken in einem allgemeinen Prädikatenkalkül aufwendig in der Implementierung ist, beschränken sich die Bedingungen auf die Angabe von Ausdrücken in entweder konjunktiver oder disjunktiver Normalform. Diese Entscheidung begründet sich auch auf einer in [MESW 01] gegebenen Empfehlung. Eine Bedingung besitzt demzufolge die Struktur

Bedingung:	`( a und b) oder (c und d) ...`

oder alternativ

Bedingung:	`( a oder b) und (c oder d) ...`

wobei `a, b, c, d` Prädikate sind. Diese enthalten ihrerseits Vergleiche zwischen Werten, die (wie es bei den Parametern der Fall ist) Literale, Rückgabewerte von Methodenaufrufen oder Attributswerte eines Objekts sein können.

5.1.6 Sprachelemente zur Verwaltung

Wie es in den meisten Sprachen der Fall ist, scheint es auch für die PDL sinnvoll, Elemente zur Annotation der Elemente einzuführen: Kommentare. An den Anforderungen gemessen, daß die anvisierte Managementanwendung leicht zu erlernen sei und Managementwissen transportiert (ADM-3, ADM-4), sind sie ein wichtiger Bestandteil der Sprache. Eine Policy wird nicht zufälligerweise geschrieben. Anders als bei einer prozeduralen Programmiersprache, in der nicht der Bedarf exisitiert, jedes Konstrukt zu dokumentieren, ist für eine Policy der Kommentar *Pflicht* (wie man es etwa bei Klassen oder Methoden einer objektorientierten, prozeduralen Sprache auch erwartet). Der Kommentar soll den Zweck der Policy dokumentieren (ihre Aktionen sind bereits in einer deutlichen Form angegeben) und erlaubt Rückschlüsse auf strategische Policies oder informelle Entscheidungen, die zu ihrer Erstellung und Aktivierung führten.

Um die Historie einer Policy in einem Umfeld mit mehreren Managern erfassen zu können, erscheint es auch sinnvoll, ihren ursprünglichen Autor und die Zeit ihrer Erstellung anzugeben, den Akteur[1], der die Policy zuletzt verändert hat, sowie die Zeit der letzten Veränderung. Auf diese Weise können fehlerhafte Policies leichter ihrem Autor zugeordnet werden.

Die Wirkungzeit von Policies kann durch Service Level Agreements an die Laufzeit eines Vertrages gebunden sein. Um eine automatische Deaktivierung nach Ablauf eines Vertrages zu unterstützen, muß das entsprechende Datum in der Policy notiert werden. Denkbar wären an dieser Stelle erweiterte Konzepte zur automatischen Aktivierung und Deaktivierung, etwa die periodische (De)Aktivierung von Policies entsprechend Werk- bzw. Feiertagen, Haupt- bzw. Nebenzeit etc. Für diese Aufgaben stehen allerdings Bedingungen (eventuell in Kombination mit Metapolicies für Zyklen längerer Periode) zur Verfügung.

Faßt man die genannten Merkmale zu einem *Deskriptor* zusammen, kann dieser folgendermaßen dargestellt werden:

Deskriptor:	`Kommentar Autor1 Zeit1 AutorN ZeitN Ablaufdatum`

5.1.7 Ein Informationsmodell für die PDL

Um die Übersetzung von Policies aus der PDL in ein „maschinenfreundliches" Format realisieren zu können, wird im folgenden ein auf PCIM basierendes Informationsmodell angegeben. Das in [MESW 01] beschriebene *Policy Core Information Model* (PCIM) stellt Mittel zur Verfügung, die als Grundlage für die Zerlegung von Policies in Teile dienen können. Das PCIM beschreibt eine (`PolicyRule`) als Aggregation von Aktionen, Bedingungen oder Gruppen dieser Bestandteile, wie in Abbildung 3.3 angedeutet wurde. Die `PolicyRule` entspricht einer Policy im Sinne der PDL.

Die gemeinsame Oberklasse für sowohl Policies, als auch für ihre Bestandteile, ist die abstrakte Klasse *Policy*. Dies erlaubt eine leichte Gruppierung von Bestandteilen, indem die für die Gruppierung notwendigen Merkmale der Unterklassen (Policies und Policy-Bestandteile) in der Oberklasse spezifiziert werden. Die Gruppierung selbst kann mit der von PCIM bereitgestellten Klasse `PolicyGroup` repräsentiert werden.

[1]Policies können unter Umständen als Folge der Ausführung von Metapolicies erstellt oder modifiziert werden. Es kann daher nicht davon ausgegangen werden, daß eine Policy durch eine *Person* erstellt oder modifiziert wurde.

Das PCIM spezifiziert die Klassen `PolicyCondition` für die Repräsentation von Bedingungen, sowie *PolicyAction* für die Darstellung von Aktionen. Das Modell muß also im Entwurf um Repräsentationen für die Elemente Subject, Target und Event erweitert werden, sodaß sämtliche Sprachelemente dargestellt werden können.

PCIM nimmt keinen Bezug auf die Teilprozesse des Abrechnungsmanagements. Es ist vielmehr ein generischer Rahmen für beliebige policybasierte Systeme. Seine allgemein gehaltenen Strukturen schränken deshalb die Kopplung der Policies an Teilprozesse nicht ein.

5.1.8 Metapolicies

Eine policy-basierte Managementanwendung, die eine große Menge von Policies zur Verwaltung ihrer Zielsysteme beherbergt, wirft das Problem der Automatisierung der Verwaltung der Policies selbst auf. Veränderungen des zu managenden Netzes, der Benutzerpopulation, der Dienste oder Tarife können Auswirkungen auf eine Vielzahl der Policies oder Policy-Bausteinen haben. *Metapolicies* bieten sich als Mittel an, diese kausalen Paare[2] selbst als Policies zu formulieren, um die Administration der Managementanwendung zu automatisieren.

Gegenüberstellung von Policies und Metapolicies

Der Unterschied zwischen „normalen” und Metapolicies liegt dabei in der Belegung ihrer Bestandteile; syntaktisch sind die beiden Policy-Typen nicht zu unterscheiden. Die bisher beschriebenen Policies führen Operationen von Managementobjekten aus, als Reaktion auf Events, die von anderen Managementobjekten erzeugt werden. Ihre Entitäten sind also unter den Managementobjekten zu finden, und ihre Aktionen sind Operationen dieser Objekte. Metapolicies, dagegen führen Operationen aus, die von der Managementanwendung selbst zur Verfügung gestellt werden. Diese Operationen dienen der Manipulation von Policies. Die Typen von Ereignissen, auf die Metapolicies reagieren, gliedern sich in zwei Gruppen: Ereignisse bezüglich Veränderungen an Policies und von Managementobjekten erzeugte Ereignisse. Letztere können also sowohl eine „normale” Policy, als auch eine Meta-Policy feuern.

Operationen für Metapolicies

Eingriffe, die ein Administrator manuell durchführen kann, sind auch Gegenstand der Automatisierung durch Metapolicies. Die in Abbildung 6.9 aufgezeigten Use-Cases, die den Administrator als Akteur aufweisen, können infolgedessen unter die Operationen für Metapolicies übernommen werden. Tabelle 5.1 zeigt eine Übersicht von für Metapolicies plausible Operationen.

Ereignisse für Metapolicies

Metapolicies sind hier als gewöhnliche operationale Policies betrachet, sodaß ihre Ausführung nicht auf eine bestimmte Menge von Ereignissen beschränkt ist. Als Verwaltungswerkzeug bezüglich des

[2] D. h. Ursache-Wirkung-Paare, in denen das Ereignis die Ursache darstellt und die erforderlichen Aktionen die Wirkung darstellen.

Operation	*Parameter*	*Beschreibung*
activatePolicy	ID	Aktiviert eine Policy
deactivatePolicy	ID	Deaktiviert eine Policy
createPolicy	Policy definition	Erzeugt eine neue Policy
createComponent	Component definition	Erzeugt einen neuen Policy-Baustein
deletePolicy	ID	Löscht eine Policy
deleteComponent	ID	Löscht einen global definierten Policy-Baustein
updatePolicy	ID, Policy definition	Verändert eine Policy
updateComponent	ID, Component definition	Verändert einen global definierten Policy-Baustein

Tabelle 5.1: Operationen zur Benutzung durch Metapolicies

Ereignis	*Parameter*	*Beschreibung*
PolicyModified	ID	Eine Policy wurde modifiziert
PolicyActivated	ID	Eine Policy wurde aktiviert
PolicyDectivated	ID	Eine Policy wurde deaktiviert
UserAdded	UserID	Ein Benutzer wurde hinzugefügt
UserRemoved	UserID	Ein Benutzer wurde entfernt
ProfileChanged	UserID	Die Einstellungen für einen Benutzer wurden verändert

Tabelle 5.2: Beispiele für Metapolicy-relvante Ereignisse

Managementsystems selbst sind allerdings diejenigen Ereignisse hervorzuheben, die nicht immer für die allgemeinen Zwecke des Managements der Abrechnungskomponenten relevant sind.

Metapolicies sollen auf Veränderungen an Policies reagieren können. Zu diesem Zweck muß die Managementanwendung dafür vorgesehene Events bei der Durchführung solcher Veränderungen senden. Insbesondere bei der Ausführung der in Tabelle 5.1 angegebenen Operationen müssen Events versendet werden, die auf den Erfolg oder Mißerfolg der Operation hinweisen. Dabei ist es nicht von Belang, ob beispielsweise eine neue Policy durch den Administrator (manuell) oder durch eine Metapolicy (automatisch) erzeugt wurde.

Im Rahmen des Prozesses *change* können Metapolicies benutzt werden, um gewisse Vorgänge zu automatisieren. Beim Hinzufügen eines Benutzers beispielsweise, kann seine Konfiguration durch „normale” Policies vorgenommen werden. Mittels Metapolicies ist es möglich, ebenfalls einen neuen, auf diesen Benutzer ausgerichteten Satz Policies anzulegen. Diese können wiederum durch Metapolicies gelöscht bzw. deaktiviert werden, wenn der Benutzer entfernt wird. Tabelle 5.2 zeigt einige Beispiele für Ereignisse, die besonders für Metapolicies von Bedeutung sind. Die in der mittleren Spalte angegebenen Parameter verstehen sich dabei als Informationen, die mit dem Ereignis mittransportiert werden.

5.2 EBNF der Policy Definition Language

Nach der Festlegung der grundsätzlichen Struktur der wichtigsten Sprachelemente kann nun eine detaillierte Spezifikation der Sprache mittels ihrer EBNF erfolgen. Vorab werden einige Hinweise bezüglich der Voraussetzungen gegeben, entsprechend derer die Sprachdefinition erstellt wird.

5.2.1 Anmerkungen zur Definition

Anwendungsbereiche der PDL

Die Policy Definition Language wird unabhängig von ihrem Einsatzbereich spezifiziert. Sie ist nicht speziell auf Abrechnungsmanagement ausgerichtet, sondern kann ohne Modifikationen in anderen Bereichen des Management benutzt werden.

Die Sprachdefinition setzt ein objektorientiertes Umfeld voraus. So werden etwa Aktionen als Methodenaufrufe spezifiziert. Es wurde auch darauf geachtet, Aufrufe mittels der *CORBA Dynamic Invocation Interface* (siehe [CORBA 2.2]) besonders zu berücksichtigen, sowie der Einsatz eines *Interface Repository*. Aus diesem Grund wird beispielsweise die Angabe von Namen für die Parameter eines Aufrufs unterstützt. Außerhalb einer objektorientierten Umgebung kann die Sprache nur nach Überarbeitung einiger ihrer zentralen Teile eingesetzt werden.

Ausrichtung auf XML

Die in EBNF spezifizierte Syntax wurde bewußt auf die Übertragung der Grammatik in ein XML-Schema ausgelegt. Durch diese Anforderung ergeben sich einige Merkmale der Sprache, die auf den ersten Blick schwerfällig oder redundant erscheinen.

Da mittels einer XML-Anwendung formulierte Policies wohlgeklammert sein müssen, trägt schon die EBNF-Grammatik diese Eigenschaft. Ausdrücke in XML werden entweder als Element oder als Attribut definiert. Die EBNF-Fassung definiert deshalb manche Sprachelemente außerhalb der Konstrukte, in denen sie erwartet werden. Ein weiteres Merkmal, das die Übertragung auf XML-Schema erleichtert ist die Einschließung der Policies und deren Bestandteile in ein `policySet` (vgl. 5.2.3). Die XML-Spezifikation fordert nämlich, daß ein XML-Dokument ein Baum sei — also eine einzige Dokumentwurzel besitzt. Durch das Einführen des Umschließenden `policySet` wird in der EBNF-Fassung der Sprache eben diese Forderung erfüllt. Die Angaben zu den Datentypen der PDL (vgl. 5.2.11) basieren ebenfalls auf den in XML-Schema definierten.

5.2.2 Vereinbarungen und Hilfsproduktionen

Die EBNF-Grammatik der *Policy Definition Language* beschreibt die Syntax der Sprachelemente einer deklarativen, kontextfreien Sprache. Die Terminalsymbole der Sprache verstehen sich als reservierte Schlüsselwörter.Bei der Angabe der Definitionen gelten folgende Vereinbarungen:

Terminalsymbole sind in Anführungszeichen gesetzt und kursiv.
Beispiel: "*Terminalsymbol*"

Nichtterminalsymbole sind in spitzen Klammern gesetzt.
Beispiel: ⟨`Nichtterminalsymbol`⟩

Option In eckigen Klammern eingeschlossene Ausdrücke kommen einmal oder kein Mal vor.
Beispiel: [optional]

Mehrfach	In geschweiften Klammern eingeschlossene Ausdrücke kommen kein Mal oder mehrere Male vor. Beispiel: { mehrfach }
Alternativen	werden durch senkrechte Striche getrennt. Beispiel: alternative1 | alternative2
Gruppierung	geschieht mittels runder Klammern. Beispiel: (⟨a⟩ ⟨b⟩) | ⟨c⟩ .
xsd:	Dieser Präfix wird Datentypen vorangestellt, die in [XMLS-2] spezifiziert werden.

Viele Sprachelemente stützen sich auf die folgenden Hilfsproduktionen. Deshalb werden diese vorab angegeben. Um die Übersichtlichkeit zu wahren, kommt dabei eine verkürzte Form für die Angabe von Folgen von Terminalsymbolen (z. B. Ziffern) zum Einsatz.

⟨**identifier**⟩ =	⟨alphachar⟩ { ⟨alphachar⟩ | ⟨digit⟩ }
⟨**digit**⟩ =	”*0*” | ”*1*” | ... | ”*9*”
⟨**alphachar**⟩ =	”_” | ”*a*” | ”*b*” | ... | ”*z*” | ”*A*” | ... | ”*Z*”
⟨**ref**⟩ =	⟨xsd:IDREF⟩
⟨**refs**⟩ =	⟨xsd:IDREFS⟩
⟨**comment**⟩ =	”*comment*{” ⟨xsd:string⟩ ”}”

In mehreren Produktionen benutzte Nichtterminalsymbole werden im folgenden beschrieben und anschließend zusammengefaßt definiert.

Sowohl Methodenaufrufe als auch die Erzeugung von Ereignissen können die Angabe von Parametern erfordern. Diese bestehen aus einem Namen sowie einem Wert und werden in einem ⟨parameterSet⟩ zusammengefaßt. Die Werte werden in 5.2.11 definiert.

Den Subjekten und Zielen einer Policy liegt das Element ⟨entityContainer⟩ zugrunde. Sie können entweder Entitäten oder Rollen sein (vgl. 5.2.10). Für Entitäten kann optional ihre Domäne angegeben werden.

⟨**parameterSet**⟩ =	⟨namedValue⟩ { ”, ” ⟨namedValue⟩ }
⟨**namedValue**⟩ =	”*namedValue*{” ”*name*=” ⟨identifier⟩ ⟨value⟩ ”}”
⟨**entityContainer**⟩ =	([⟨domain⟩] ⟨entity⟩) | ⟨role⟩
⟨**domain**⟩ =	[”/”] { ⟨identifier⟩ ”/” }
⟨**entity**⟩ =	⟨identifier⟩

5.2.3 Das ⟨**policySet**⟩

Jeder Satz dieser Sprache ist ein sogenanntes *Policy Set*, d. h. eine Menge von Policy-Deklarationen, sowie Deklarationen einzelner Bestandteile von Policies.

⟨**policySet**⟩ =	”*policySet*{” { ⟨policy⟩ | ⟨roleDefinition⟩ | ⟨subject⟩ | ⟨target⟩ | ⟨event⟩ | ⟨action⟩ | ⟨policyGroup⟩ } ”}”
⟨**roleDefinition**⟩ =	”*roleDef*{” ⟨referable⟩ ⟨roleName⟩ ⟨roleDescription⟩ ”}”
⟨**roleName**⟩ =	”*name*{” ⟨xsd:token⟩ ”}”
⟨**roleDescription**⟩ =	”*description*{” ⟨xsd:string⟩ ”}”
⟨**role**⟩ =	”*role*{” ⟨refs⟩ ”}”
⟨**policyGroup**⟩ =	”*policyGroup*{” ⟨group⟩ ⟨comment⟩ ”}”
⟨**group**⟩ =	”*group*{” ⟨refs⟩ ”}”

5.2.4 Referenzierbarkeit und Zuordnung zu Prozessen

Manche der Bestandteile von Policies, sowie die Policies selbst, sind *referenzierbar*. In der Praxis bedeutet dies, daß eine Instanz solcher Konstrukte ein Kennzeichen (ID) enthält, sodaß eine Instanz eines anderen Konstrukts durch Angabe dieser ID darauf verweisen kann. Die Implementierung der Sprache muß für die Eindeutigkeit dieser IDs sorgen. Außerdem soll die Möglichkeit gegeben werden, Sprachkonstrukte als *Bibliotheksbestand* markieren zu können. Das Element `referable` wird in Konstrukte aufgenommen, die diesen Forderungen entsprechen sollen.

⟨**referable**⟩ =	”*id*=” ⟨xsd:ID⟩ [⟨libraryItem⟩] [⟨comment⟩]
⟨**libraryItem**⟩ =	”*libraryItem*=” ⟨xsd:boolean⟩

Deklarationen referenzierbarer Konstrukte können außerhalb von Policy-Definitionen vorkommen. Das Element *comment* bietet dem Anwender der Sprache die Möglichkeit, den Zweck des Eintrags als Teil des Konstrukts zu vermerken.

Im Entwurf der Sprachelemente wurde die Möglichkeit der Zuordnung referenzierbarer Bausteine zu Prozessen der Abrechnung gefordert. Da dies Referenzierbarkeit einschließt, erweitert die folgende Definition *referable*.

⟨**processAssignable**⟩ = ⟨referable⟩ [⟨relatedProcess⟩]

⟨**relatedProcess**⟩ = ”*relatedProcesses*{”
⟨process⟩ { ”,” ⟨process⟩ } ”}”

⟨**process**⟩ = ⟨identifier⟩ | ”*OTHER*” | ”*ALL*”

Ein oder mehrere Namen von Prozessen können angegeben werden; mit *ALL* kann die Zugehörigkeit zu allen Prozessen ausgedrückt werden, *OTHER* bezeichnet einen in der Auswahl nicht vorhandenen Prozeß.

5.2.5 Policy

Die Policy ist das Zielkonstrukt der Sprache. Ihre Funktion läßt sich folgendermaßen grob umschreiben: Eine *policy* enthält in einem *actionSet* Aktionen,

- die auf den im *targetSet* angegebenen Zielobjekten ausgeführt werden,
- wenn ein im *eventSet* spezifiziertes Ereignis eintritt
- und die im *constraintSet* enthaltenen Bedingungen erfüllt sind.

Neben diesen Hauptbestandteilen besitzt eine Policy weitere Attribute, die teils der Verwaltung und Dokumentation des Policy-Systems dienen, teils die Modularität des Konzepts und seine Ausrichtung auf Prozesse unterstützen.

Für eine Policy kann eine *Domäne* angegeben werden, die zwei Funktionen erfüllen kann. Sie

1. beschränkt den Wirkungsradius der Policy, und
2. dient als *default-Domain* für Elemente, deren Domäne nicht explizit angegeben wurde.

Die Domänenangabe in der Policy dient hingegen nicht der Verwaltung der Policies selbst. Eine domänenbasierte Verwaltung der Policies kann allerdings mit Hilfe der *Policy-ID* extern realisiert werden.

Der *Descriptor* der Policy dient der Verwaltung. Er enthält Zeitdaten bezüglich der Erstellung und Modifikation einer Policy, sowie jeweils die Möglichkeit der Angabe des Autors. Optional kann in dem Attribut *expires* ein Zeitpunkt angegeben werden, an dem die Policy automatisch deaktiviert werden soll.

Einige der Verwaltungsattribute wurden außerhalb des Deskriptorkonstrukts definiert:

isEnabled gibt an, ob die Policy gegenwärtig aktiv ist oder nicht.

priority gibt optional die der Policy zugeordnete Priorität an.

version bezieht sich auf die Version der Sprache. Dieses Attribut soll bei Veränderungen der Sprache dazu beitragen, in neuen Versionen der PDL formulierte Policies von den ”Altlasten” zu unterscheiden.

⟨**policy**⟩ =	”*policy*{” ⟨processAssignable⟩ [”*domain*{” ⟨domain⟩ ”}”] [⟨subjectSet⟩] [⟨targetSet⟩] ⟨eventSet⟩ [⟨constraintSet⟩] ⟨actionSet⟩ [⟨descriptor⟩] ⟨attribs⟩ ”}”
⟨**descriptor**⟩ =	”*descriptor*{” [⟨createdBy⟩] [⟨creationDate⟩] [⟨lastModified⟩] [⟨modifiedBy⟩] [⟨expires⟩] ”}”
⟨**attribs**⟩ =	”*attributes*{” ⟨enabled⟩ [⟨priority⟩] [⟨version⟩] ”}”
⟨**createdBy**⟩ =	”*createdBy*=” ⟨xsd:token⟩
⟨**creationDate**⟩ =	”*dateOfCreation*=” ⟨xsd:date⟩
⟨**modifiedBy**⟩ =	”*modifiedBy*=” ⟨xsd:token⟩
⟨**lastModified**⟩ =	”*lastModified*=” ⟨xsd:dateTime⟩
⟨**enabled**⟩ =	”*isEnabled*=” ⟨xsd:boolean⟩
⟨**priority**⟩ =	”*priority*=” ⟨digit⟩ [⟨digit⟩]
⟨**version**⟩ =	”*version*=” ⟨digit⟩ ”.” ⟨digit⟩
⟨**expires**⟩ =	”*expires*=” ⟨xsd:dateTime⟩

Beispiele für Policy-Definitionen Um ein Gefühl für die Syntax der Sprache zu vermitteln, werden im folgenden zwei Beispiel-Policies angegeben. Die in Listing 5.1 angegebene Policy enthält nur lokale Definitionen, während das zweite Beispiel (Listing 5.2) ausschließlich referenzierte Bausteine benutzt. Die in dieser Policy referenzierten Bausteine müssen natürlich (an einer anderen Stelle) definiert werden. Die Aktionen der Beispiel-Policy sind etwa weiter unten in Listing 5.8 angegeben.

Listing 5.1: Policy mit ausschließlich lokalen Definitionen

```
policy{ id=f00ba1
        relatedProcesses{metering}
        domain{/acct/metering}
        subjectSet{
                subject{BWMeter}
        }
        targetSet{
                target{Collector}
        }
        eventSet{
                event{name=Shutdown}
```

```
        }
        actionSet{
                action{
                        default{
                        setUnavailable( namedValue{ name=meter string{BWMeter} } )
                        }
        attributes{isEnabled=true version=1.0}
}
```

Listing 5.2: Policy mit referenzierten Bausteinen.

```
policy{ id=f00ba2
        targetSet{ group{ 0xffaa01 0xffaa02 0xffaa03}}
        eventSet{ group{ 0xeeaa01 0xeeaa02}}
        actionSet{ group{ 0xaaff01 0xaaff02 0xaaff03 }}
        constraintSet{ constraintRef{ 0xccaa05 }}
        attributes{isEnabled=true}
}
```

5.2.6 Hauptbestandteile einer Policy

Die wichtigsten Bestandteile einer Policy sind in Mengen (engl. *set*) organisiert, um die Art der Einbindung eines Bestandteils in eine Policy zu kapseln. Es gibt generell drei Arten, ein Policy-Bestandteil anzugeben:

- lokal, durch Formulierung des Bestandteils in der Policy selbst
- als einzelne Referenz auf ein (anderswo) definiertes Bestandteil
- als Gruppe von Referenzen

So bezeichnet ⟨`event`⟩ ein lokal deklariertes Ereignis, während ⟨`eventRef`⟩ eine Referenz auf ein Ereignis darstellt, das außerhalb der Policy deklariert wurde (vgl. die Definitionen in 5.2.7).

Um die Modularität der Policy-Definitionen, sowie die Wiederverwendung ihrer Bauteile zu fördern, wird die Benutzung von global definierten, referenzierbaren Bauteilen empfohlen. Es sind allerdings Fälle denkbar, in denen diese Strategie zu einer großen Mengen globaler Definitionen führt, die allerdings nur in einer Policy benutzt werden — ein Zustand der die Übersichtlichkeit der Policy-Bibliothek negativ beeinflußt. Deshalb wird jeweils maximal ein lokal formulierter Bestandteil in einem *set* erlaubt.

⟨**subjectSet**⟩ =	”*subjectSet*{” (⟨subject⟩ | ⟨subjectRef⟩ | ⟨subjectGroup⟩) ”}”
⟨**targetSet**⟩ =	”*targetSet*{” (⟨target⟩ | ⟨targetRef⟩ | ⟨targetGroup⟩) ”}”
⟨**eventSet**⟩ =	”*eventSet*{” (⟨event⟩ | ⟨eventRef⟩ | ⟨eventGroup⟩) ”}”
⟨**actionSet**⟩ =	”*actionSet*{” (⟨action⟩ | ⟨actionRef⟩ | ⟨actionGroup⟩) ”} ”
⟨**constraintSet**⟩ =	”*constraintSet*{” (⟨cnf⟩ | ⟨dnf⟩ | ⟨constraint⟩ | ⟨constraintRef⟩) ”}”

5.2.7 Die Elemente im ⟨**eventSet**⟩

Die Deklaration eines Ereignisses besteht im Grunde lediglich aus der Angabe seines Namens. Es ist allerdings zu erwartet, daß die möglichen, von den Systemkomponenten generierten Ereignisse, zum Zeitpunkt der Erstellung von Policies bekannt sind. Infolgedessen kann es zweckmäßig sein, alle Ereignistypen vorab, von den Policies getrennt, zu deklarieren. Aus diesem Grund sind die Ereignisse selbst referenzierbar, während die Ereignismengen statt der expliziten Deklaration eines Ereignisnamens die Angabe einer Referenz auf ein bestehendes Ereignis, oder einer Gruppe solcher Ereignisse, erlauben.

⟨**event**⟩ =	”*event*{” [⟨referable⟩] ”*name*=” ⟨identifier⟩ ”}”
⟨**eventRef**⟩ =	⟨ref⟩
⟨**eventGroup**⟩ =	⟨group⟩

Beispiele für `eventSet` Die folgenden Listings zeigen ein lokal angegebenes Event (5.3), ein einzelnes, referenziertes Event (5.4) und eine Gruppe von Referenzen auf Events (5.5). Die Hauptbestandteile von Policies folgen dem Muster von `eventSet` in ihrer Syntax.

Listing 5.3: Event Set mit lokal definiertem Event

```
eventSet{ event{name=MyEvent} }
```

Listing 5.4: Event Set einer Referenz auf ein Event

```
eventSet{ eventRef{ 0xeeaa01 } }
```

Listing 5.5: Event Set mit Gruppe von Referenzen auf Events

```
eventSet{ group{ 0xeeaa01 0xeeaa01 0xeeaa02 0xeeaa12 }}
```

5.2.8 Die Elemente im ⟨**actionSet**⟩

Eine Aktionsmenge kann — analog zu der Ereignismenge — wahlweise eine lokal deklarierte Aktion enthalten, oder eine Referenz oder Gruppe von Referenzen auf Aktionen.

⟨**action**⟩ =	”*action*{” ⟨processAssignable⟩ ⟨defaultAction⟩ [⟨errorAction⟩] ”}”
⟨**actionRef**⟩ =	⟨ref⟩
⟨**actionGroup**⟩ =	⟨group⟩

Eine Aktion kann aus zwei Teilen bestehen: der auszuführenden Aktion ⟨defaultAction⟩ und der optionalen Aktion ⟨errorAction⟩. Letztere wird — falls angegeben — ausgeführt, wenn die Ausführung der Standardaktion fehlschlägt. Jeder dieser beiden Aktionen kann wahlweise einen Methodenaufruf oder das Erzeugen und Versenden eines Ereignisses bewirken.

⟨**defaultAction**⟩ =	”*default*{” ⟨genericAction⟩ ”}”
⟨**errorAction**⟩ =	”*onError*{” ⟨genericAction⟩ ”}”
⟨**genericAction**⟩ =	⟨methodCall⟩ | ⟨generateEvent⟩
⟨**methodCall**⟩ =	”*invoke*{” [⟨objectName⟩ ”.”] ⟨method⟩ ”}”
⟨**generateEvent**⟩ =	”*generateEvent*{” ⟨eventName⟩ ”(” ⟨parameterSet⟩ ”)}”
⟨**method**⟩ =	⟨identifier⟩ ”(” ⟨parameterSet⟩ ”)”
⟨**eventName**⟩ =	⟨identifier⟩
⟨**objectName**⟩ =	⟨identifier⟩

Beispiele: Die folgenden Beispiele zeigen zwei actionSet. Das erste (Listing 5.6) ist ein global definiertes actionSet mit lokal definierter Aktion. Das zweite (Listing 5.7) enthält eine Gruppe von Referenzen auf Aktionen, die selbst in Listing 5.8 global definiert sind.

Listing 5.6: Action Set mit lokal angegebenen Aktionen

```
actionSet{
        action{
                default{invoke{SomeObject.someMethod() }
                onError{ generateEvent{
                        ActionFailure(name=ident string{"SomeObject.someMethod")
                        }
                }
        }
}
```

Listing 5.7: Action Set mit Gruppe von Referenzen

```
actionSet{
        comment{Anhalten eines Dienstes.}
        group{ 0xaaff01 0xaaff02 0xaaff03 }
}
```

Die in den `actionSet`-Klauseln referenzierten Aktionen sind global definiert. Die erste enthält als Standardaktion den Methodenaufruf `stop()`. Da in der Aktion kein Zielobjekt angegeben wurde, werden die Zielobjekte (`target`) der Policy benutzt. Zur Fehlerbehandlung wird die Erzeugung eines Ereignisses `ActionFailure` mit Parametern `object` und `action` angegeben. Die zweite Aktion gibt Methodenaufrufe an für sowohl die Standardaktion als auch für die Fehlerbehandlung. Die dritte, schließlich, enthält nur eine Standardaktion. Da hier das Objekt `Logger` ausdrücklich angegeben ist, wird die Operation `info` dieses Objekts aufgerufen; bei Mißerfolg des Aufrufs werden keinerlei Maßnahmen durchgeführt.

Listing 5.8: Global definierte Aktionen

```
action{ id=0xaaff01
        comment{Zielobjekt anhalten}
        default{invoke{stop() }
        onError{
                generateEvent{
                        ActionFailure(
                                namedValue{name=object string{Policy.target}}
                                namedValue{name=action string{"stop"}}
                        )
                }
}
action{ id=0xaaff02
        comment{Zielobjekt beenden.}
        default{ invoke{ shutdown() } }
        onError{ invoke{kill() } }
}
action{ id=0xaaff03
        comment{Beendigung eines Zielobjekts in das Protokoll schreiben.}
        default{
                invoke{ Logger.info(name=message string{"Shutdown."}) }
        }
}
```

5.2.9 Die Elemente im ⟨`constraintSet`⟩

Die Menge der Bedingungen einer Policy wird in einer Normalform organisiert. Die PDL unterstützt konjunktive (CNF)[3] und disjunktive Normalform (DNF). Für einfachere Fälle kann die Bedingungsmenge auch eine einzige Bedingung enthalten; auch eine Referenz auf eine außerhalb des `constraintSet` deklarierte Bedingung ist zulässig.

[3]engl: *conjunctive normal form*.

```
⟨cnf⟩ =                  ”and{” ⟨binaryLogOpOR⟩
                         { ”, ” ⟨binaryLogOpOR⟩ } ”}”

⟨dnf⟩ =                  ”or{” ⟨binaryLogOpAND⟩
                         { ”, ” ⟨binaryLogOpAND⟩ } ”}”

⟨constraint⟩ =           (
                             (
                              ”equal” | ”smaller” | ”greater”
                             | ”greaterEqual” | ”smallerEqual”
                             ) ”{” ⟨binaryPredicate⟩ ”}”
                         )
                         | ( [”!”] ⟨value⟩ )

⟨constraintRef⟩ =        ⟨ref⟩

⟨binaryLogOpOR⟩ =        ⟨constraint⟩ { ”||” ⟨constraint⟩ }

⟨binaryLogOpAND⟩ =       ⟨constraint⟩ { ”&&” ⟨constraint⟩ }

⟨binaryPredicate⟩ =      ⟨value⟩ ”, ” ⟨value⟩
```

5.2.10 Die Elemente in ⟨**subjectSet**⟩ und ⟨**targetSet**⟩

Die *subjects* und *targets* der PDL beschreiben Entitäten des zu verwaltenden Systems. Die Definitionen der ihnen entsprechenden Sprachelemente sind deshalb ähnlich.

```
⟨subject⟩ =              ”subject{” [⟨referable⟩]
                         ⟨entityContainer⟩ ”}”

⟨subjectRef⟩ =           ⟨ref⟩

⟨subjectGroup⟩ =         ⟨group⟩

⟨target⟩ =               ”target{” [⟨referable⟩] ⟨entityContainer⟩ ”}”

⟨targetRef⟩ =            ⟨ref⟩

⟨targetGroup⟩ =          ⟨group⟩
```

Beispiele für Target Sets und Subject Sets In Listing 5.9 wird die Definition eines Targets und eines Subjects and derselben Entität demonstriert. Die Angabe von referenzierten `target` und `subject` - Instanzen ist syntaktisch analog zu der von beispielsweise `event` (vgl. 5.2.7).

Listing 5.9: Definitionen derselben Entität als Target und als Subject, jeweils mit Domänenangabe

```
target{ id=0xffaa01 /mgmt/BillingSystem }
subject{ id=0xaa0001 /mgmt/BillingSystem }
```

5.2.11 Werte

Die PDL unterstützt fünf Typen:

1. Ganze Zahlen – *integer*
2. Gleitkommazahlen – *float*
3. Zeichenketten – *string*
4. Wahrheitswerte (*wahr* oder *falsch*)– *boolean*
5. Zeitangaben der Form `yyyyddmmThhmmss` – *dateTime*

Die Wertebereiche der Typen werden in der Spezifikation der PDL bewußt nicht angegeben — ihre Festlegung obliegt dem Anwendungsentwurf. Als Ausgangspunkt werden jedoch die in [XMLS-2] angegebenen Definitionen der Wertebereiche benutzt.

Ein Wert besteht aus der Angabe eines Typs und der eines Inhalts. Es gibt vier Klassen von Elementen, die als Inhalt dienen können:

literal Eine Konstante ist ein textuell in der Policy angegebener Wert. Er ist nur durch Editieren der Policy veränderbar.

array Reihungen haben keine Begrenzung ihrer Länge. Sie dürfen nur Elemente gleichen Typs enthalten.

functionValue ist der Rückgabewert eines Methodenaufrufs.

attributeValue ist ein Attributswert. Bei der Angabe eines *attributeValue* als Wert muß der Name eines Objekts sowie der Name eines Attributs angegeben werden.

⟨**value**⟩ =	⟨type⟩ ”{” ⟨valueContent⟩ ”}”
⟨**type**⟩ =	”*float*” | ”*integer*” | ”*string*” | ”*boolean*” | ”*dateTime*”
⟨**valueContent**⟩ =	⟨literal⟩ | ⟨array⟩ | ⟨functionValue⟩ | ⟨attributeValue⟩
⟨**literal**⟩ =	⟨xsd:float⟩ | ⟨xsd:integer⟩ | ⟨xsd:string⟩ | ⟨xsd:boolean⟩ | ⟨xsd:dateTime⟩
⟨**array**⟩ =	”*array*{” ⟨length⟩ ((⟨literal⟩ { ”, ” ⟨literal⟩ }) | ”*null*”) ”}”
⟨**functionValue**⟩ =	⟨methodCall⟩
⟨**attributeValue**⟩ =	”*attributeValue*{” ⟨objectName⟩ ”.” ⟨attributeName⟩ ”}”
⟨**attributeName**⟩ =	⟨identifier⟩
⟨**length**⟩ =	”*length*=” ⟨xsd:integer⟩

Einige Beispiele zur Notation von Werten:

Konstante: `float{3.14}`

Reihung: `float{array{length=3 3.15, 3.14, 3.13}}`

Rückgabewert: `float{invoke{FooObject.barMethod(name=a integer{2})}`

Attributswert: `float{attributeValue{FooObject.barAttribute}`

5.3 Zusammenfassung

In diesem Kapitel wurde die *Policy Definition Language* spezifiziert, die bei der zu entwickelnden Managementanwendung zum Einsatz kommt. Dem Entwurf der Sprachelemente und Überlegungen zur Modularität folgte die Spezifikation der Sprache in EBNF. Dabei wurden die einzelnen Sprachelemente definiert, erklärt und anhand von Beispielen erläutert.

Die Sprache erlaubt die Formulierung von Policies und von Policy-Bausteinen. Insbesondere bietet sie die Möglichkeit, die für Methodenaufrufe notwendigen Konstrukte sowie Bedingungen auf Basis von Normalformen (konjunktive und disjunktive) zu formulieren. Durch die Wohlklammerung der Elemente weist sie eine gute Repräsentierbarkeit mittels streng baumartiger Strukturen auf.

Kapitel 6

Entwurf der Managementanwendung

Indem in Kapitel 4 Konzepte zur Entwicklung einer Managementanwendung angegeben wurden, sowie die Spezifikation einer Policy Definition Language in Kapitel 5, sind die Vorarbeiten für ihren Entwurf geschaffen. In diesem Kapitel werden die konzeptionellen Bausteine in einem objektorientierten Entwurf zusammengefügt. Nach einer Übersicht der Architektur des Systems wird die Gliederung der Managementanwendung in Komponenten beschrieben, die eine Zusammenfassung verwandter Funktionen der Anwendung erlaubt. Die Darstellung von Policies und Policy-Bausteinen in der Managementanwendung wird erläutert, ebenso wie die Übersetzung von in PDL-Notation vorliegenden Ausdrücken in diese Darstellung. Abschließend werden dynamische Aspekte der Komponenten der Managementanwendung beschrieben, indem auf die Funktionsweise der einzelnen Komponenten eingegangen wird.

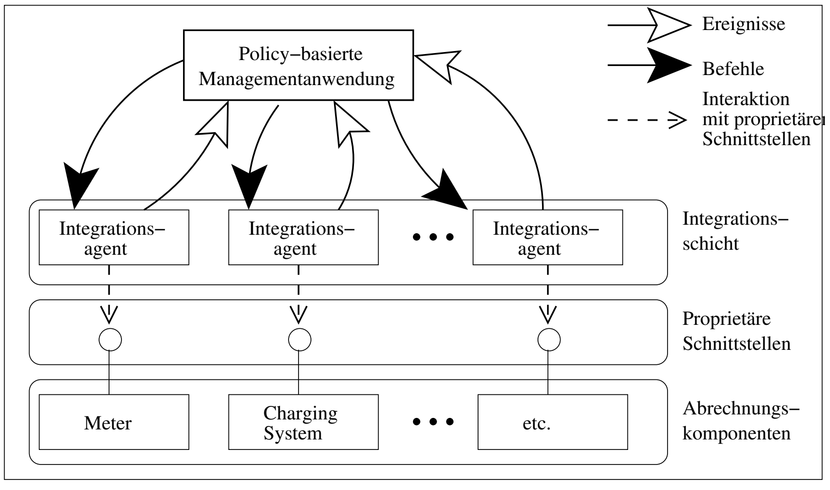

Abbildung 6.1: Architektur des Managementsystems

6.1 Architektur des Managementsystems

Das Managementsystem besteht, wie in Abbildung 6.1 skizziert, aus der Managementanwendung und der Integrationsschicht. Die Integrationsschicht besteht aus Agenten, die einerseits auf die proprietären Schnittstellen der Abrechnungskomponenten zugreifen, andererseits gegenüber der Managementanwendung eine einheitliche Schnittstelle bieten. Außerdem senden sie (entsprechend bei den Abrechnungskomponenten beobachteten Typen von Vorkommnissen) Nachrichten bezüglich auftretender *Ereignisse* an die Managementanwendung. Sind für ein bestimmtes Ereignis Policies spezifiziert worden, können als Folge deren Auswertung *Befehle* an einen Integrationsagenten gesandt werden, die unter Umständen in Zugriffen auf die Schnittstelle der von dem Agenten verwalteten Abrechnungskomponenten resultieren.

6.1.1 Funktionalität der Managementanwendung

Die Managementanwendung muß mehreren, teilweise disjunkten Aufgaben gerecht werden können.

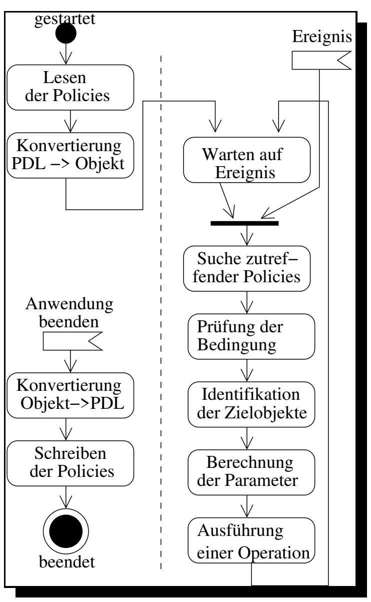

Abbildung 6.2: Folge von Aktivitäten im Betrieb der Managementanwendung

Reaktion Eingehende Ereignisse lösen eine Reaktion aus. Die Managementanwendung prüft, ob für ein aufgetretenes Ereignis Policies spezifiziert wurden. Wenn dies der Fall ist, muß die Auswertung dieser Policies angestoßen werden.

Auswertung von Bedingungen Wurden für eine Policy Bedingungen angegeben, müssen diese ausgewertet werden, um zu entscheiden, ob die Aktionen der Policy ausgeführt werden. Bedingungen sind aussagenlogische Formeln; sie können Prädikate enthalten, die beispielsweise Rückgabewerte von Operationsaufrufen vergleichen. Die Prüfung der Bedingungen einer Policy erfordert daher unter Umständen das Ausführen von Operationen auf Zielobjekten, ähnlich wie in den Aktionen der Policy.

Ausführung der Aktionen Zur Ausführung der Aktionen einer Policy werden Operationen ihrer Zielobjekte aufgerufen. Diese Objekte müssen also zunächst ermittelt werden (z. B. bei Angabe einer Rolle). Ebenso müssen unter Umständen die Werte der Parameter der Operation berechnet werden, da diese selbst Rückgabewerte von Operationen sein können. Anschließend wird im Rahmen der Aktion ein Operationsaufruf pro Zielobjekt ausgeführt.

Erzeugung Zum Umgang mit Policies innerhalb der Managementanwendung wird aus ihrer *textuellen Form* in PDL eine Darstellung mittels Objekten erstellt. Zu diesem Zweck müssen die in PDL angegebenen Policies von einer Speichervorrichtung (z. B. Datenbank, Datei, Webserver etc.) gelesen und in eine interne Repräsentation übersetzt werden.

Speicherung Policies werden als Ausdrücke der PDL persistent gespeichert. Wurden einige während des Betriebs verändert, so muß ihre Objektdarstellung wiederum in Textform übersetzt werden.

Es ist zweckmäßig, eine Gliederung der Anwendung in *Komponenten* vorzunehmen, und die umzusetzende Funktionalität auf diese zu verteilen. Die Gliederung richtet sich einerseits nach der Reihenfolge und dem Ort der Ausführung der Funktionen, andererseits spielen Implementierungsaspekte sowie Annahmen bezüglich der Erweiterbarkeit eine Rolle. Anhand von Abbildung 6.2 können bereits zwei Komponenten identifiziert werden: eine, die für die Speicherung und Übersetzung von Policies verantwortlich ist, und eine, die die Auswertung von Bedingungen sowie die Ausführung von Aktionen übernimmt. Dabei wurden allerdings die dynamischen Aspekte der Policy-Verwaltung noch nicht betrachtet. Während des Betriebs sollen Policies hinzugefügt, gelöscht, modifiziert und aktiviert bzw. deaktiviert werden können. Diese Funktionen werden in der Regel durch einen Benutzer kontrolliert. Aufgrund dieser Überlegungen ist es sinnvoll, die Funktionen der Managementanwendung unter folgende drei Komponenten aufzuteilen:

PolicyRepository: ist für die Erzeugung der Policy-Objekte sowie für die persistente Speicherung der Policies verantwortlich.

PolicyEnforcer: empfängt Ereignisse, wertet die Bedingungen der passenden Policies aus und führt ggf. ihre Aktionen aus.

PolicyManager: registriert aktive Policies beim Enforcer, stellt Verwaltungsoperationen auf Policies zur Verfügung und interagiert mit dem Benutzer.

Die Aufteilung der Funktionen folgt dabei dem von Gamma et al in [GHJV 95] beschriebenen Entwurfsmuster *Mediator* (vgl. Abbildung 6.3), wobei der Policy Manager die Funktionalität von Repository und Enforcer (lose) koppelt.

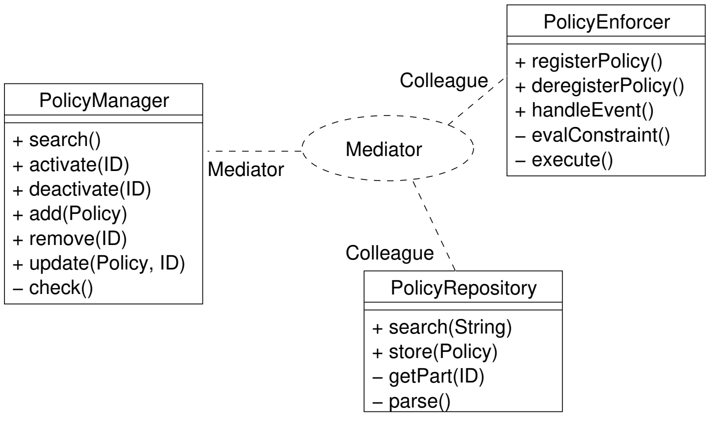

Abbildung 6.3: Aufteilung der Funktionalität der Managementanwendung mit Hilfe des *Mediator Pattern*

Die Funktionalität des eingangs (vgl. Kapitel 1) beschriebenen *Interpreters* wird dabei zwischen Repository und Enforcer aufgeteilt. Dabei werden die Aufgaben mit statischem Charakter (z. B. Parsen von PDL-Ausdrücken) bereits im Repository bewältigt, während Funktionen mit dynamischen Charakter (Auflösung von Objektnamen in (Ziel-) Objekte, Auswertung von Bedingungen) dem Enforcer zufallen.

Repository	*Manager*	*Enforcer*
Policy in XML lesen	Policy(-Baustein) hinzufügen	Ereignis empfangen
Policy in XML schreiben	Policy(-Baustein) löschen	Policy empfangen und registrieren
XML-Dokument parsen	Policy(-Baustein) modifizieren	Policy deregistrieren
Policy-Objekte generieren	Suche nach Policy(-Baustein)	Bedingung auswerten
XML aus Policy-Objekt generieren	Suche nach Zeichenkette	Aktion ausführen
Syntax des XML-Dok. prüfen	Integrität einer Policy prüfen	Ereignis generieren

Tabelle 6.1: Aufgaben der drei Komponenten der Managementanwendung

Der Umgang mit PDL-Ausdrücken in ihrer Textform wird infolgedessen auf das Repository beschränkt; in den weiteren Komponenten wird die Objektrepräsentation von Policies verarbeitet.

6.1.2 Komponenten der Managementanwendung

Die in Tabelle 6.1 angegebenen, den drei Komponenten zugeteilten Funktionen werden an untergeordnete, aber klar abgegrenzte Teilkomponenten delegiert. So besteht das **Repository** beispielsweise aus einer *PolicyStorage*, die Policies persistent in XML-Dokumenten speichert und aus XML-Dokumenten einliest, einem *Parser* der PDL-Ausdrücke in einen Syntaxbaum umwandelt, sowie einer *Policy Factory*, die Objektrepräsentationen von Policies und Policy-Bausteinen erzeugt. Der **Enforcer** besteht aus einem *EventHandler*, der Ereignisse empfängt und nach passenden, aktiven Policies sucht, einem *Evaluator*, der die Bedingungen einer Policy auswertet, sowie einem *Executor*, der Aktionen ausführt bzw. Ereignisse generiert.

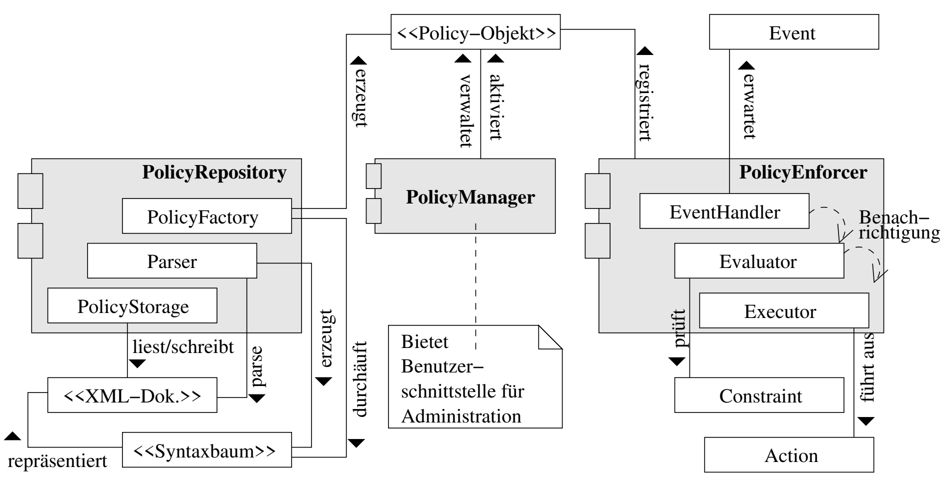

Abbildung 6.4: Beziehungen zwischen (Unter-) Komponenten und Policies bzw. Policy-Bausteinen

Die drei Komponenten bilden dabei jeweils ein *Facade*-Entwurfsmuster [GHJV 95], indem sie die Schnittstellen ihrer Unterkomponenten zum Teil nach außen hin anbieten, trotzdem aber deren realen Schnittstellen und Implementierung kapseln. Die Funktionsweise der Managementanwendung kann daher als Zusammenspiel der drei Hauptkomponenten Repository, Enforcer und Manager betrachtet werden.

6.2 Darstellung und Übersetzung von Policies

Alle Komponenten der Managementanwendung müssen Policies verarbeiten, sei es in ihrer PDL-Notation oder in einer internen, objektorientierten Repräsentation. Dazu muß die Darstellung einer Policy und ihren Bestandteilen mittels Objekten spezifiziert werden, wie auch eine Vorgehensweise für die Übersetzung von Ausdrücken der PDL in entsprechende Objekte.

6.2.1 Repräsentation von Policies durch Objekte

Die Objektrepräsentation von Policies lehnt sich an die in dem *Policy Core Information Model* (PCIM) [MESW 01] an. Dabei soll diese einerseits eine leicht veränderbare Struktur darstellen und andererseits die Umsetzung der Policy Definition Language als XML-Anwendung unterstützen. Durch die in Kapitel 5 angegebene, wohlgeklammerte Sprache bietet sich die Möglichkeit an, eine Policy als Baum ihrer Komponenten darzustellen — ein Charakteristikum, das auch PCIM, wie auch ähnlichen Standardisierungsbemühungen entspricht. Solch eine Darstellung erlaubt ebenfalls die separate Verwaltung eigenständiger, „freier" Policy-Bestandteile.

Eine Policy, also eine *Regel*, wird durch die Klasse `PolicyRule` beschrieben (Abbildung 6.5). Sie besitzt außer ihren Attributen (ID, Angaben zur Prozeßzugehörigkeit etc.) eine feste Anzahl von referenzierten Objekten, die jeweils ein *Feld* (vgl. Kapitel 3) der Policy repräsentieren. Diese Objekte besitzen die Klassen `PolicyAction` für die Repräsentation des Aktionfeldes der Policy, `PolicyCondition` für die des Bedingungsfeldes, `PolicyEvent` für die Repräsentation des Ereignisfeldes, sowie `PolicyTarget` und `PolicySubject`, die Zielobjekte bzw. *Subjekte* repräsentieren.

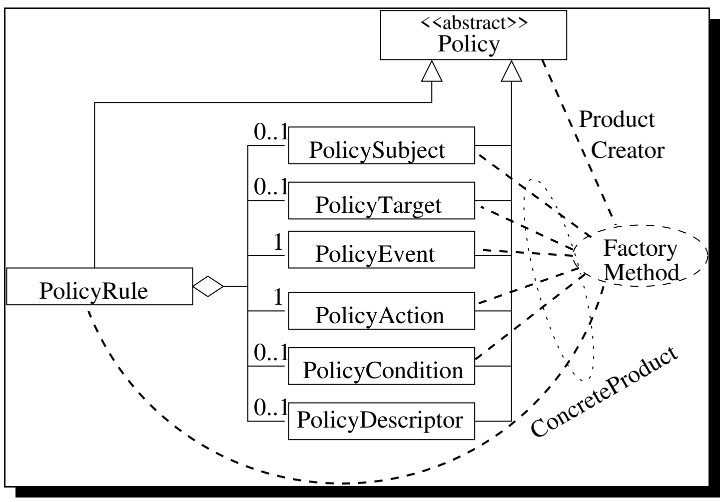

Abbildung 6.5: Objektrepräsentation einer Policy

Entsprechend PCIM werden gemeinsame Merkmale dieser Klassen in einer abstrakten Klasse `Policy` zusammengefaßt, die zusätzlich die *Creator*-Rolle bei der Erstellung von Instanzen von PolicyRule und seinen Bestandteilen im Rahmen eines *Factory*-Entwurfsmusters übernimmt. Der PDL-Spezifikation folgend besitzt eine PolicyRule außerdem einen Deskriptor, der insbesondere eine Zeitangabe zum Ablauf der Gültigkeit der PolicyRule enthalten kann.

Auch wenn diese Klassen verschiedene Felder einer Policy darstellen, besitzen sie die Gemeinsamkeiten, die in der Spezifikation der Policy Definition Language zwischen den *Set*-Ausdrücken (z. B. *ActionSet*) zu finden sind: sie erlauben Belegungen des von ihnen dargestellten Policy-Feldes mittels eines „lokal definierten" Objekts, einer *Referenz* auf ein Objekt, sowie eine *Gruppe* von Referenzen. Die PDL spezifiziert die Referenzierung von Policy-Bestandteilen mittels eindeutigen Identifikatoren. Diese Referenzen müssen bei der Erstellung der Objektrepräsentation einer Policy aufgelöst werden, sodaß die Unterscheidung zwischen lokal definiertem Bestandteil, Referenz und Gruppe entfällt. Die Repräsentation eines Policy-Feldes vereinfacht sich also nach Übersetzung der textuellen Policy und Auflösen der Referenzen in eine Liste von Objekten eines Typs, die ein oder mehrere Einträge enthalten kann. Zur leichten weiteren Verarbeitung der `PolicyRule` bieten die genannten Klassen die in `ComponentSet` angegebene Schnittstelle an; diese erlaubt es, die Länge der Liste, den Typ ihrer Elemente, sowie einzelne Einträge der Liste abzufragen.

Hierarchie der Repräsentationsobjekte

Eine Policy wird zur Laufzeit durch einen Objektbaum repräsentiert, der eine `PolicyRule` als Wurzel hat. Direkte Söhne der Wurzel sind Hauptbestandteile der Policy. Die Objekte auf den tieferen Stufen transportieren die in den Policy-Bestandteilen enthaltenen Informationen.

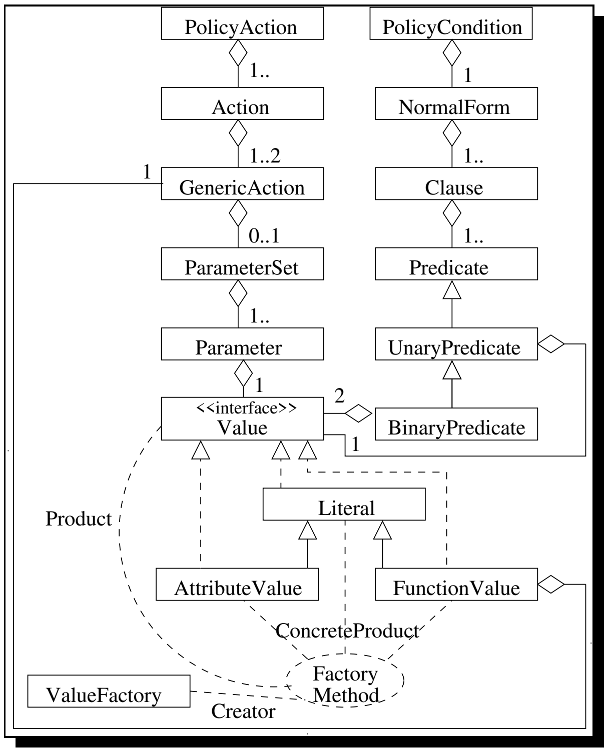

Abbildung 6.6: Repräsentation von Aktionen und Bedingungen

Während sich die genannten Hauptbestandteile lediglich als ein Mittel zur Strukturierung einer Policy in Felder vestehen, stellen ihre Elemente demnach die eigentliche Belegung der Felder dar, beispielsweise den Namen eines Ereignisses, Zielobjekt und Methodenname einer Aktion etc. Ihrerseits können die Elemente eigene, spezifische Bestandteile enthalten, die der Struktur der Sprachelemente der PDL entsprechen. Es entsteht eine Hierarchie von Klassen, die es erlaubt, beliebige Ausdrücke der Policy Definition Language darzustellen.

Aktionen und Bedingungen Dabei bilden die Aktionen und Bedingungen die komplexeren Strukturen der Hierarchie, sodaß sie separat in Abbildung 6.6 dargestellt werden.

Die PDL erlaubt in einer Policy eine beliebige Anzahl von Aktionen (Klasse `Action`), die ihrerseits eine Standardaktion enthält, sowie eine, die beim Fehlschlagen der Standardaktion durchgeführt wird (vgl. Kapitel 5). Die beiden letzteren werden durch Instanzen der Klasse *GenericAction* repräsentiert und können einen Funktionsaufruf oder die Erzeugung eines Ereignisses spezifizieren. Beide Varianten erfordern Parameter, die mit Hilfe der Klasse `ParameterSet` zusammengefaßt werden. Ein Parameter besitzt einen Wert, der entweder konstant oder durch einen Funktionsaufruf bzw. das Auslesen eines Attributswerts zur Laufzeit ermittelt werden muß. Indem Funktionsaufrufe als Parameter von Funktionsaufrufen erlaubt werden, wird Verkettung und *Rekursion* unterstützt.

Die benutzte konjunktive bzw. disjunktive Normalform kann als „Konjunktion von Disjunktionen" respektive „Disjunktion von Konjunktionen" betrachtet werden. Die folgende Darstellung sieht von der Art der Normalform ab:

$$\boxed{(P_1 \text{ op}_1 \text{ P}_2) \text{ op}_2 \text{ (P}_3 \text{ op}_1 \text{ P}_4) \text{ op}_2 \text{ ... op}_2 \text{ (P}_{n-1} \text{ op}_1 \text{ P}_n)}$$

Dabei sind $P_1...P_n$ Prädikate, op_1 und op_2 die logischen Operatoren „und" bzw. „oder". Ist op_2 die Konjunktion handelt es sich um eine konjunktive Normalform, folglich ist op_1 die Disjunktion. Bei umgekehrter Belegung von op_1 und op_2 handelt es sich offensichtlich um die disjunktive Normalform. Zur Repräsentation beider Normalformen werden dieselben Klassen benutzt. Ein Objekt der Klasse `NormalForm` verweist auf eine Anzahl Instanzen der Klasse `Clause`, die die Ausdrücke in den Klammern repräsentieren. Jedes `Clause`-Objekt verwaltet eine Anzahl Prädikate der Typen `UnaryPredicate` oder `BinaryPredicate`. Die obige vereinfachte formale Darstellung zeigt lediglich zwei Prädikate innerhalb der Klammern. Die Anzahl der von `Clause` verwalteten Prädikate ist natürlich nicht auf zwei beschränkt. Sowohl NormalForm als auch Clause besitzen ein privates Attribut `boolean conjunction`, das angibt, ob der verwendete Operator die Konjunktion ist.

Die in der PDL spezifizierten Prädikate sind ein- oder zweistellig. Die einstelligen kapseln einen boolschen Wert, der durch das Prädikat negiert werden kann. Die zweistelligen Prädikate repräsentieren Vergleiche zweier Werte mittels der üblichen Vergleichsoperatoren„gleich"„größer", „kleiner", „kleiner gleich", „größer gleich". Die verglichenen Werte müssen dabei von selben Typ sein. Die Klassen `UnaryPredicate` bzw. `BinaryPredicate` benutzen entsprechend der Stelligkeit der Prädikate, die sie repräsentieren, ein oder zwei Objekte zur Repräsentation von Werten. Diese Objekte müssen die in `Value` implementierte Schnittstelle zur Verfügung stellen und gehören einer der Klassen `Literal`, `AttributeValue` oder `FunctionValue` an. Dabei entspricht eine `Literal`-Instanz einer in der Policy angegebenen Konstante, `AttributeValue` repräsentiert einen Attributswert eines Objekts und `FunctionValue` verweist auf eine `GenericAction`-Instanz, die ihrerseits einen Methodenaufruf repräsentieren muß; der Rückgabewert dieses Aufrufs ist der von der `FunctionValue`-Instanz repräsentierte Wert.

Ereignisse, *Target* und *Subject* Die verbliebenen Bestandteile der Policy benötigen weitaus einfachere Repräsentationsstrukturen, wie man Abbildung 6.7 entnehmen kann.

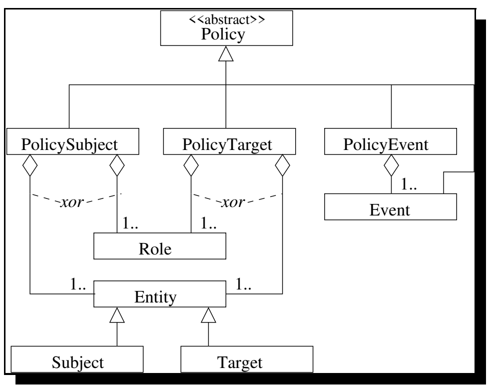

Abbildung 6.7: Repräsentation von Ereignissen, *Targets* und *Subjects*

`PolicyEvent` etwa verwaltet lediglich eine Anzahl von `Event`-Instanzen, die ihrerseits lediglich den Namen eines Ereignisses enthalten. Die Kapselung des Ereignisnamens in einer eigenen Klasse ist gerechtfertigt, da die Policy in PDL-Notation Referenzen auf Ereignisse enthalten kann, die aufgelöst werden müssen. Dieser Vorgang ermittelt `Event`-Instanzen, die an anderer Stelle „frei" definiert wurden. `Event` erbt dazu die in `Policy` definierten Verwaltungsmechanismen (z. B. das ID-Attribut). *Targets* und *Subjects* werden in der PDL mittels des selben Konstrukts ⟨`entityContainer`⟩ definiert, das in der Objektrepräsentation seine Entsprechung in der Klasse `Entity` findet. Die von dieser Klasse abgeleiteten Klassen `Target` und `Subject` dienen lediglich der Untertypisierung der `Entity`-Instanzen.

Die durch `PolicySubject` und `PolicyTarget` repräsentierten Policy-Felder werden wahlweise durch eine Anzahl Instanzen von `Entity`s Unterklassen oder durch Objekte der Klasse `Role` belegt.

Beispiel der Darstellung einer Aktion Klassendiagramme weisen lediglich auf die statischen (möglichen) Beziehungen zwischen den zur Laufzeit existierenden Objekten hin. Das Objektdiagramm in Abbildung 6.8 soll einen Eindruck einer zur Laufzeit typischen Struktur vermitteln. Es zeigt eine Policy `pr` mit einer Aktion; auf die weiteren Felder wurde in der Darstellung zugunsten der Übersichtlichkeit verzichtet.

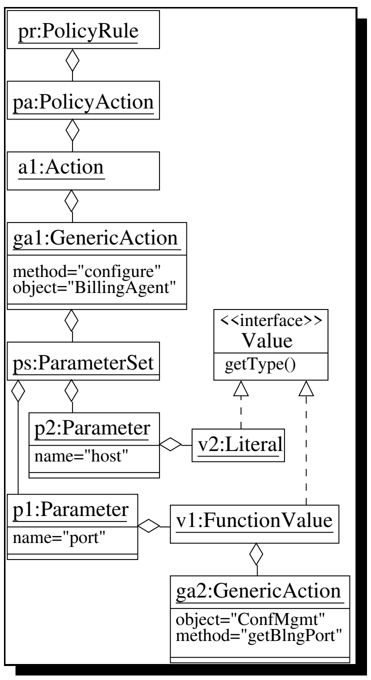

Abbildung 6.8: Beispiel der Objektdarstellung einer Policy mit einer Aktion

Das Objekt `pa` der Klasse `PolicyAction` kapselt eine Aktion `a1` der Klasse *Action*, die wiederum (entsprechend der PDL) eine Operation `ga1` (`GenericAction`) enthält. Diese weist die für die Durchführung der Operation notwendigen Namen von Zielobjekt und Methode auf, sowie die innerhalb eines `ParameterSet` enthaltenen Argumente des Aufrufs. Dabei enthält ein Parameter einen konstanten Wert, der andere beschreibt wiederum einen Operationsaufruf. Der Wert des Parameters `p1` ergibt sich also als Rückgabewert des in `ga2` angegebenen Aufrufs.

Die verhältnismäßig hohe Anzahl von Objekten, die zur Repräsentation von Policy-Teilen notwendig ist, wird durch die Vorteile gerechtfertigt, die sich bezüglich Veränderbarkeit und Erweiterbarkeit von Sprache und Datenstruktur ergeben. Indem diese Hierarchie der Klammerung in der PDL entspricht ist es möglich, die Veränderung eines Sprachelements durch Modifikation nur einer Klasse in der Objektrepräsentation umzusetzen. Ebenso ist es leicht, Elemente in den Baum hinzuzufügen, ohne Veränderungen an unbeteiligten Zweigen notwendig zu machen. Die Aufteilung der policyverarbeitenden Komponenten in die in den folgenden Abschnitten beschriebenen funktionalen Einheiten unterstützt diesen Ansatz, indem Veränderungen am Laufzeitsystem auf wenige Klassen beschränkt sind.

6.2.2 Übersetzung von Ausdrücken der Policy Definition Language

Ein weiterer wichtiger Grund für die Aufteilung der objektorientierten Repräsentation von Policies in viele Klassen ist die Erleichterung der Übersetzung zwischen den zwei Darstellungen. PDL-Ausdrücke werden in zwei Läufen übersetzt. Im ersten Lauf wird der Eingabetext gescannt und geparsed, wodurch ein Syntaxbaum entsteht. Im zweiten Lauf wird durch Analyse des Syntaxbaums die Objektrepräsentation der PDL-Ausdrücke erstellt. Dabei übernimmt jede Klasse der Repräsentationshierarchie die Analyse des ihr entsprechenden Teilbaums im Syntaxbaum. Werden beim Durchlauf eines Teilbaums Zweige gefunden, die einer anderen Klasse der Hierarchie zugeordnet werden können, so wird die Analyse dieses Zweigs an diese Klasse weitergegeben. So ist es möglich, mit einem rekursiven Durchgang des Syntaxbaums den Objektbaum der internen Darstellung einer Policy komplett zu generieren. Dabei muß jede Klasse nur diejenigen anderen kennen, die entsprechend der Grammatik der PDL als Teilbäume „ihres” Syntaxteilbaums vorkommen können. Beispielsweise kann die Angabe einer Aktion keine Bedingungen enthalten; daher können die Klassen, die Aktion bzw. Bedingung repräsentieren bezüglich der Übersetzung als disjunkt betrachtet werden.

Auflösen von Referenzen

Die Policy Definition Language unterstützt und fördert die Wiederverwendung von Policy-Bausteinen mittels Referenzierung. Während der Übersetzung müssen diese Referenzen aufgelöst werden. Die Vorgehensweise zur Auflösung der Referenzen lehnt sich zwar an den „klassischen" Ansatz mit Benutzung einer Symboltabelle an, vereinfacht diesen aber durch Berücksichtigung der Merkmale der PDL.

Bei Betrachtung der Grammatik der PDL kann festgestellt werden, daß Referenzierung nur in bestimmten Elementen der Sprache vorkommen kann. Weiterhin fällt auf, daß manche referenzierbare Bausteine, beispielsweise einzelne, frei definierte Ausdrücke (z. B. von Event, Target etc.) keine Referenzen enthalten dürfen, wohl aber eine ID, damit sie selbst referenziert werden können. Dies führt zu einer einfachen Heuristik zur Auflösung der Referenzen während der Erzeugung des Objektbaums. Zunächst werden die frei definierten Ausdrücke übersetzt, die entsprechend der Grammatik selbst keine Referenzen enthalten dürfen. Die dadurch entstehenden Repräsentationsobjekte werden anhand der ID der Ausdrücke indexiert abgelegt (eine Hashtable eignet sich gut für diesen Zweck, wobei die ID des Ausdrucks als Schlüssel benutzt werden kann). Frei definierte Ausdrücke sind im Syntaxbaum (durch die Wohlgeklammerung der PDL) immer direkte Söhne der Wurzel, sodaß nur eine kleiner Teil des Syntaxbaums durchlaufen werden muß, um sie zu übersetzen. Danach können die verbliebenen Teilbäume der Wurzel durchlaufen werden; sollten diese Referenzen auf die soeben übersetzten Elemente enthalten, können die entsprechenden, schon vorhandenen Objekte anhand ihrer ID abgerufen werden. Werden im ersten Schritt bereits übersetzte Teilbäume aus dem Syntaxbaum entfernt, verkürzt sich der Durchlauf im zweiten Schritt entsprechend der Anzahl der im Eingabetext vorhandenen „freien" Ausdrücke.

Das Ergebnis der Übersetzung ist allgemein betrachtet ein Objektgraph. In nicht allzu komplexen Fällen handelt es sich dabei um einen Wald von Objektbäumen, die teils „ganze" Policies, teils Policy-Bausteine repräsentieren. Die Policies werden zur Steuerung der Managementanwendung benutzt, während die Policy-Bausteine für die Erstellung neuer Policies zur Verfügung stehen.

6.3 Verwaltung der Policies

Die Verwaltung der Policies in der Managementanwendung geschieht im Zusammenspiel zwischen PolicyManager und PolicyRepository. Zu ihren grundlegenden Funktionen gehört die persistente Speicherung von Policies und Policy-Bausteinen, die Suche nach Policies und Policy-Bausteinen, sowie ihre Manipulationsfunktionen, die es erlauben, Policies und Bausteine zu:

- erstellen
- modifizieren
- löschen
- aktivieren
- deaktivieren

Auch wenn diese Funktionen durch Metapolicies angestoßen werden können, steht die Manipulation von Policies durch den Administrator, also dem Benutzer der Managementanwendung im Vordergrund. Es muß beachtet werden, daß eine Policy unter Umständen gleichzeitig in zwei Formen vorliegt: als Text in PDL-Notation, sowie als Objektbaum für ihre interne Darstellung. Um die beiden Darstellungen konsistent zu halten, können Techniken, die für die Verwaltung von Caches benutzt werden (vgl. z. B. [HePa 96]). Wie in Abbildung 6.2 angedeutet wird, besteht eine einfache Lösung darin, die Übersetzung zwischen textueller Form und Objektdarstellung nur beim Start der Anwendung durchzuführen, und die Übersetzung in die andere Richtung bei ihrem Herunterfahren. Zusätzlich müssen neu hinzugefügte, in PDL vorliegende Policies und Policy-Bausteine bei Bedarf in ihre Objektrepräsentation übersetzt werden. Solch eine Vorgehensweise kann zu Datenverlust z. B. beim Auftreten eines Gerätefehlers führen, da die aktuelle Version modifizierter Policies nur im Hauptspeicher als Objektbaum vorliegt. Da aufwendige Konsistenzverfahren über den Zweck einer prototypischen Implementierung hinausgehen, wird dieses Problem an dieser Stelle nicht weiter betrachtet. Statt dessen werden die Verwaltungsoperationen aufgeteilt in jene, die sich ausschließlich mit der Objektrepräsentation der Policies befassen, und solche, die auch die textuelle Form berücksichtigen müssen. Zu den letzteren gehören die Suchfunktionen, das Erstellen und Modifizieren, sowie selbstverständlich die persistente Speicherung.

6.3.1 Manipulation und Suche

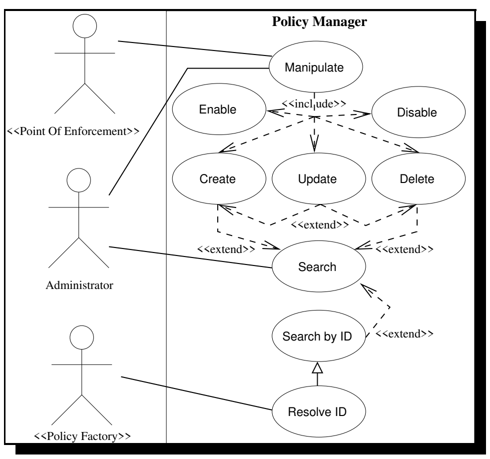

Abbildung 6.9: Use-Cases des PolicyManager

Der PolicyManager stellt die zentrale Schnittstelle für die Verwaltung von Policies und Policy-Teilen dar. Er führt die entsprechenden Aktionen selbst nicht aus, sondern setzt „high-level"-Aktionen, die beispielsweise dem Administrator mittels einer Benutzeroberfläche angeboten werden können, aus Grundfunktionen von Repository und Enforcer zusammen. Abbildung 6.9 zeigt Use-Cases des PolicyManagers, die von drei verschiedenen Akteuren in Anspruch genommen werden. Die in der Abbildung mit *<<include>>* bzw. *<<extend>>* beschrifteten Beziehungen zwischen den Use-Cases geben die Komposition von Funktionen an. Will etwa ein Administrator, oder auch der Enforcer (als Aktion einer Metapolicy) eine Policy aktivieren, so wird diese zunächst im Repository gesucht und dann beim Enforcer registriert. Die Kontrolle über diesen Vorgang liegt allerdings — entprechend dem beschriebenen Mediator-Entwurfsmuster — beim PolicyManager; Repository und Enforcer sind in diesem Fall lediglich für die Ausführung einzelner Teile zuständig.

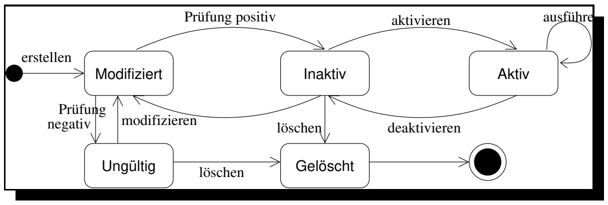

Abbildung 6.10: Der Lebenszyklus eines Policy-Objektbaums

Bei der Modellierung der Verwaltungsoperationen muß der in Abbildung 6.10 skizzierte Lebenszyklus einer Policy in ihrer Objektrepräsentation berücksichtigt werden. Damit eine Policy ordnungsgemäß ausgewertet und ausgeführt werden kann, muß sie eine gültige Objektrepräsentation besitzen. Insbesondere muß sie den minimalen Anforderungen für eine Policy genügen, d. h. eine Belegung ihres Ereignisfeldes und ihres Aktionsfeldes besitzen. Die Prüfung dieser Anforderungen geschieht beim erstmaligen Erstellen der Objektrepräsentation, sowie bei eventuellen Modifikationen der Policy. Ist das Ergebnis der Prüfung negativ, so ist die Policy ungültig und somit unbrauchbar. Dieser Zustand kann beispielsweise eintreffen, wenn ein in der Policy referenzierter Baustein nicht definiert ist, die Referenz einen Baustein des falschen Typs (z. B. eine Bedingung statt ein Ereignis) bezeichnet oder mehrdeutig ist. Eine sorgfältige Implementierung des PolicyManager kann helfen, solche Fälle zu vermeiden, indem sie die Eindeutigkeit der Referenzen sicherstellt, sowie Verfahren zur Wahrung der referentiellen Integrität einbringt. Insbesondere sollte dabei zugesichert werden, daß ein von mindestens einer Policy referenzierter „freier" Baustein nicht gelöscht wird.

Beispiel: Verändern einer Policy Ein typischer Vorgang im laufenden Betrieb der Managementanwendung ist die Veränderung einer Policy durch den Administrator, etwa das Ersetzen eines Bausteins (z. B. eine Aktion) durch einen neuen.

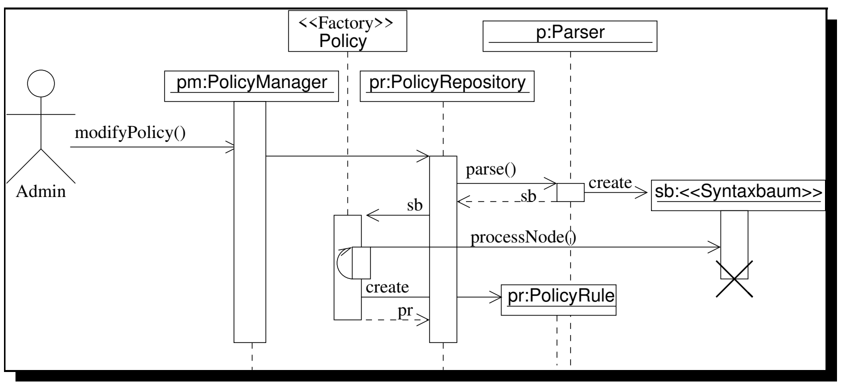

Abbildung 6.11: Verändern einer Policy

Der neue Baustein wird in der Policy Definition Language angegeben. Er muß also durch die Managementanwendung in die intern benutzte Objektrepräsentation übersetzt werden. Eine Veränderung der Policy entspricht dem Erstellen einer neuen Policy, mit der die alte Version ersetzt wird.

Suchparameter und -vorgehensweise Die Suche nach Policies sowie Policy-Bausteinen im Repository ist für die Zusammenstellung neuer Policies aus Bausteinen, sowie für die Auflösung von Referenzen auf solche Bausteine bei der Generierung einer Objektrepräsentation für eine Policy notwendig. Suchanfragen nach der ID einer Policy (bzw. eines Bausteins) werden vor allem durch das Laufzeitsystem (genauer: durch den PolicyManager) automatisch gestellt.

Es wird erwartet, daß für einen Suchvorgang, der von der Interaktion mit einer Person bestimmt wird, andere Suchkriterien erforderlich sind. Neben einer „Volltextsuche" nach in der PDL-Notation vorkommenden Zeichenketten kann nach Attributen wie

- Objektnamen
- Methodennamen
- Ereignisnamen
- Prozeß

gesucht werden, etwa um Bausteine zu finden, die sich auf ein bestimmtes Objekt (oder eine bestimmte Rolle) beziehen, oder die einem bestimmten Teilprozeß der Abrechnung zugeordneten Policies. Da die zu durchsuchenden Policies und Policy-Bausteine zum Teil sowohl in textueller Form als auch als Objekte vorliegen, wird die Reihenfolge der Suche festgelegt: Die Objektrepräsentation wird zuerst durchsucht, da Modifikationen zunächst in dieser Darstellung umgesetzt werden. Danach werden die verbleibenden Policies und Bausteine in ihrer Textform durchsucht. Durch dieses Verfahren werden jeweils die aktuellsten Policies oder Bausteine erfaßt, unabhängig ihrer Darstellung zum Zeitpunkt der Suche.

6.3.2 Das Policy Repository

Das PolicyRepository ist für sowohl die Speicherung der Policies, als auch für die Erzeugung ihrer Objektrepräsentation zuständig. Es bildet die Schnittstelle zwischen den beiden Formen, in denen Policies vorliegen können. Bei Aktivierung einer Policy wird diese durch den PolicyManager angefordert. Bei Bedarf wird ihre textuelle Form eingelesen und daraus die Objektrepräsentation erzeugt. Die Funktionalität zur Übersetzung von Policies und Bausteinen ist dabei, wie schon beschrieben, in den Klassen zur Repräsentation dieser Konstrukte lokalisiert. Mit Hilfe eines *Factory*-Entwurfsmusters wird die Erzeugung von Repräsentationsobjekten aus PDL-Ausdrücken kontrolliert. Dies reduziert die Funktionalität des PolicyRepository auf die Speicherung der als PDL-Ausdrücke vorliegenden Policies und auf die Verwaltung der Referenzen auf frei definierte Bausteine. Die Transformation der beiden Darstellungsformen ineinander wird durch das PolicyRepository lediglich angestoßen.

Policies liegen, wenn sie persistent gespeichert werden, als Text vor. Dabei ist unerheblich, ob es sich beim „Text" um in PDL notierte Policies in Dateien handelt, oder ob diese beispielsweise in einem Datenbanksystem mittels eines geeigneten Schemas abgelegt werden. Um die Portierbarkeit der Policies zu erhalten, muß allerdings von der Speicherung der Objektrepräsentation selbst abgesehen werden. Abgesehen von dieser Einschränkung werden an dieser Stelle bewußt keine weiteren Vorgaben zur Speicherung gemacht; die genaue Art und Weise ihrer Umsetzung soll von der jeweiligen Implementierung bestimmt werden.

6.4 Die Enforcer-Komponente

Die Kernaufgabe der Managementanwendung ist die Steuerung der zu verwaltenden Zielobjekte durch Managementaktionen. Bei Eintreffen eines Ereignisses wird die Auswertung der dafür definierten Policies ausgelöst und gegebenenfalls deren Aktionen ausgeführt. Da diese Vorgänge eng miteinander verbunden sind, werden sie durch den *PolicyEnforcer* realisiert. Dieser besteht aus mehreren Klassen, die jeweils eine der Aufgaben übernehmen. Der Enforcer verfügt über eine *Registry*, bei der aktive

Policies eingetragen werden können; er verarbeitet ausschließlich die Objektrepräsentation der Policies. Bei Bedarf können diese anhand der Ereignisse, auf die sie reagieren, abgerufen werden. Bei Deaktivierung einer Policy wird sie aus der Registry entfernt.

6.4.1 Ereignisse

Ereignisse werden als Ereignisobjekte von einem `EventReceiver` empfangen und an den Enforcer weitergeleitet, der in der Registry dazu passende Policies sucht und die Auswertung ihrer Bedingungen anstößt. Mit Hilfe der in `EventHandler` definierten Schnittstelle werden die ereignisempfangenden Objekte generalisiert, sodaß sie auch in anderen Teilen der Anwendung wiederverwendet werden können (Abbildung 6.12).

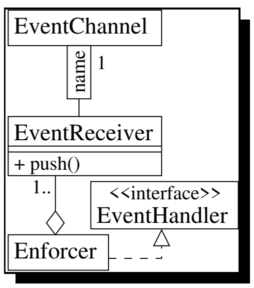

Abbildung 6.12: Propagierung von Ereignissen

Zum Senden und Empfangen von Ereignissen eignet sich der CORBA Event-Service. Dieser unterstützt zwei Modelle zur Propagierung von Ereignissen: das *Push*- und das *Pull*-Modell. Beide beschreiben Erzeuger (supplier) und Verbraucher (consumer) von Ereignissen. Im Pull-Modell erfragt der Verbraucher aufgetretene Ereignisse (polling), während im Push-Modell der Verbraucher über das Eintreten eines Ereignisses benachrichtigt wird. Für das Managementsystem wird das Push-Modell gewählt. Der tatsächliche Transport der Ereignisse obliegt einem *EventChannel*, bei dem sich Verbraucher für Benachrichtigungen registrieren, und an den die Ereigniserzeuger die Ereignisse senden. Der EventReceiver ist somit ein Verbraucher von Ereignissen, die ihm von einem EventChannel gesendet werden; er ist an einen mit Namen identifizierten EventChannel gebunden. Es besteht die Möglichkeit, mehrere EventReceiver zu benutzen, etwa um Ereignisse aus mehreren administrativen Domänen in demselben Enforcer zu bearbeiten.

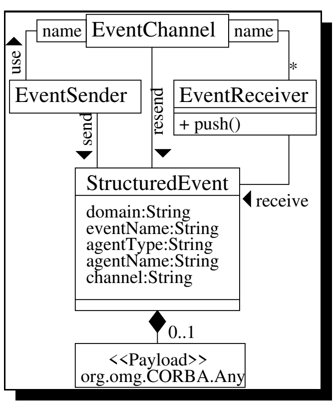

Abbildung 6.13: Struktur der Ereignisse

Ereignisse werden durch *Ereignisobjekte* dargestellt, die entspechend der Spezifikation eines *StructuredEvent* des CORBA NotificationService [OMG 00-06-20] jeweils mehrere Textfelder als Attribute, sowie ein nicht weiter beschriebenes Objekt besitzen. Die Semantik der Textfelder ist in der Spezifikation des StructuredEvent angegeben, während das freibleibende *Payload*-Objekt für benutzerspezifische Erweiterungen zur Verfügung steht (Abbildung 6.13).

Im Rahmen des hier beschriebenen Managementsystems kann dieses Objekt zum Transport von Informationen benutzt werden, die für einen bestimmten Ereignistyp charakteristisch sind. Signalisiert ein Ereignis z. B. einen aufgetretenen Fehler in einem der Integrationsagenten, können Details über die Art und den Grund des Fehlers in einem Objekt gekapselt dem Ereignis beigelegt werden. In Policies kann auf Attribute dieses Objekts zugegriffen werden, um deren Werte z. B. mit in der Policy angegebenen Konstanten zu vergleichen. Eine Spezifikation der Struktur dieser Payload-Objekte erfolgt nicht, da sie von einem Anwendungsfall zum anderen verschiedene Anforderungen erfüllen muß. Indem das Auslesen seiner Attribute dynamisch geschehen kann, ist die Festlegung der Klasse dieser Objekte auch nicht notwendig; die Werte der Attribute — oder auch Methoden — können direkt in einer Policy abgerufen werden.

Das Gegenstück des EventReceiver ist der `EventSender`. Dieser setzt Ereignisobjekte zusammen und sendet sie mittels eines EventChannel an die registrierten EventReceiver. Die Agenten der Integrationsschicht benutzen EventSender-Instanzen, um die Managementanwendung über Ereignisse in einer überwachten Abrechnungskomponente zu informieren. Ereignisse können auch als Aktion einer Policy erzeugt werden. In diesem Fall werden sie vom Enforcer gesendet und — im Fall nur einer Enforcer-Instanz — von ebendiesem wieder empfangen und verarbeitet.

Die Komponenten, die Ereignisse generieren bzw. verarbeiten sind auf mehrere Systeme, womöglich in verschiedenen Domänen, verteilt. In der Konfiguration des Managementsystems muß deshalb auf die Paarung von Sendern und Empfängern mittels eines anhand seines Namens identifizierbaren EventChannel geachtet werden.

6.4.2 Verarbeitung

Beim Empfang eines Ereignisses durch den Enforcer werden die auf dieses Ereignis reagierenden, aktiven Policies von dem Enforcer ermittelt. Diejenigen von ihnen, die keine Bedingungen enthalten, können sofort ausgeführt werden (genauer: ihre Aktionen). Die verbleibenden werden nur dann ausgeführt, wenn ihre Bedingungen jeweils erfüllt sind.

Der Enforcer delegiert die Auswertung der Bedingungen an den `Evaluator` und die Ausführung der Aktionen an den `Executor`. Diese beiden Klassen bilden zusammen mit den bereits genannten (Registry, EventReceiver) die wichtigsten Teile der Enforcer-Komponente. Abbildung 6.14 zeigt ihre Beziehungen und Schnittstellen.

Kontext von Auswertung und Ausführung

Sowohl die Auswertung der Bedingungen, als auch die Ausführung der Aktionen einer Policy, können sich auf Werte beziehen, die in anderen Teilen der Policy zu finden sind. Wird beispielsweise für eine Methode kein Zielobjekt explizit angegeben, so wird das im Target-Feld der Policy angegebenen Objekt (oder Objekte) substituiert. Ein ähnlicher Fall ist der Vergleich eines in einer Bedingung angegebenen (konstanten) Wertes mit einem Attributswert des Payload-Objekts eines Ereignisses. Hierbei handelt es sich nicht um ein in der Policy verfügbares Datum, sondern um einen Wert, der von außen, z. B. von einem Integrationsagenten generiert und mittels eines Ereignisses transportiert wird.

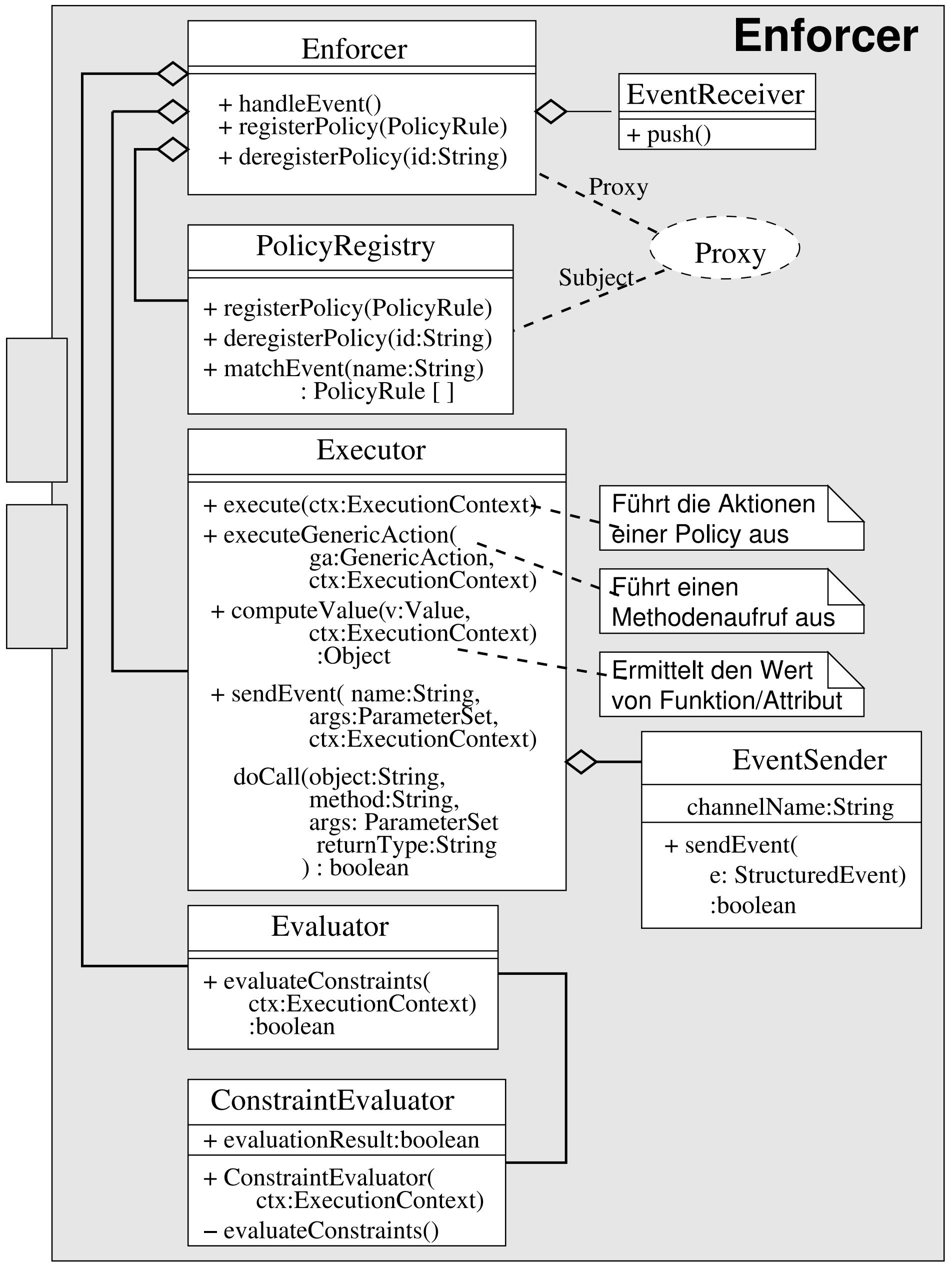

Abbildung 6.14: Übersicht der Enforcer-Komponente

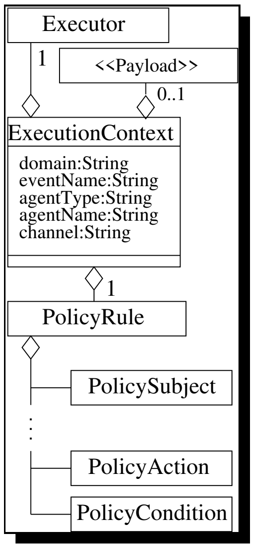

Abbildung 6.15: Kontextobjekt für Auswertung und Ausführung

Um solche Fälle bei der Auswertung und Ausführung einer Policy erfassen zu können, wird ein `ExecutionContext` (Abbildung 6.15) eingeführt, der alle für Bedingungen und Aktionen relevanten Informationen bündelt. Im ExecutionContext sind die Werte der Attribute des empfangenen StructuredEvent vorhanden, das mit ihm transportierte Payload-Objekt (gegebenenfalls), die `PolicyRule` die ausgewertet oder ausgeführt werden soll, sowie eine Referenz auf eine Executor-Instanz zur Ausführung von Aktionen. Da die meisten dieser Informationen als Referenzen bzw. Zeiger auf bestehende Objekte vorliegen, entsteht ein nur geringer Zusatzaufwand durch die Einführung des Kontextes. Es ergeben sich allerdings zwei weitere Vorteile. Indem alle notwendigen Informationen innerhalb eines Kontextobjektes gebündelt werden, können einfachere Schnittstellen für Executor und Evaluator benutzt werden. Statt einer Reihe von Argumenten muß lediglich eine Referenz auf das Kontextobjekt an die öffentlichen Methoden von Executor und Evaluator übergeben werden; diese Schnittstellen sind also unabhängig von der Implementierung der Ereignisobjekte und invariant gegenüber Veränderungen der Objektrepräsentation von Policies und deren Bausteine. Der Evaluator erhält innerhalb eines Kontextes eine Executor-Instanz für die Auswertung von Bedingungen, die Funktions- oder Attributswerte enthalten. Dadurch können mehrere Executor- und Evaluator-Instanzen innerhalb derselben Enforcer-Komponente parallel arbeiten. Bei Benutzung eines Thread-Pools würde der Enforcer jeder Bedingungsauswertung aktiv eine verfügbare Executor-Instanz zuweisen und so auf (synchronisierte) Zugriffsmechanismen für diesen Pool (wie auch auf für Executor und Evaluator sichtbare Warteschlangen) verzichten.

Nach Erstellung einer ExecutionContext-Instanz sind deren Attribute nicht mehr zu verändern. Die Lebensdauer der Instanz reicht bis zur vollständigen Verarbeitung der Bedingungen und Aktionen der Policy als Reaktion auf ein Ereignis.

Auswertung von Bedingungen

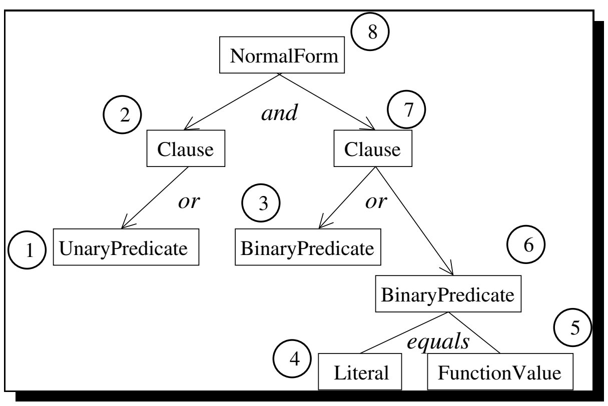

Abbildung 6.16: Auswertung eines Bedingungsbaums

Bedingungen bilden Bäume von logischen Ausdrücken, deren Blätter Prädikate sind, die zu booleschen Werten ausgewertet werden können. Ein solcher Baum wird in einem Tiefendurchlauf (*depth-first*) rekursiv durchlaufen, beginnend bei der durch eine *NormalForm*-Instanz gegebenen Wurzel. Bei jedem Rückschritt von dem letzten Sohnknotens eines Vaterknoten wird der im Vaterknoten angegebene Operator auf die Söhne angewendet. Das Beispiel in Abbildung 6.16 zeigt eine konjunktive Normalform repräsentiert durch eine `NormalForm` mit zwei `Clause`-Söhnen. Die eingekreisten Zahlen zeigen die Reihenfolge der Auswertung. Knoten derselben Ebene werden mit dem kursiv gesetzten Operator logisch verknüpft, entsprechend der Art der Normalform werden etwa die Werte der Söhne der `Clause`-Instanzen ver-odert, während die Werte der `Clause`-Instanzen selbst konjunktiv verknüft werden.

Exemplarisch werden für eine der `BinaryPredicate`-Instanzen die Platzhalter für ihre Werte angegeben. Es handelt sich dabei um eine Konstante und den Rückgabewert eines Methodenaufrufs, die auf Gleichheit hin geprüft werden (durch *equals* in der Abbildung angedeutet). Um den Rückgabewert des in der `FunctionValue`-Instanz angegebenen Methodenaufrufs zu ermitteln, muß offensichtlich die Methode aufgerufen werden. Dies geschieht mit Hilfe des Executors, der im Ausführungskontext angegeben ist.

Ausführung von Aktionen

Eine Policy enthält ein Anzahl Aktionen, die mittels ein oder zwei `GenericAction`-Instanzen Methodenaufrufe oder die Erzeugung von Ereignissen spezifizieren. Aus dem Aktionsbaum, wie er bereits in Abbildung 6.8 angedeutet wurde, werden also die `GenericAction`-Objekte interpretiert. Die Knoten oberhalb dieser Ebene bilden lediglich die Baumstruktur, diejenigen unterhalb dieser Ebene sind Parameter des Methodenaufrufs oder zur Erzeugung eines Ereignisobjektes. Der `Executor` verfügt über Methoden, die jeweils eine Ebene des Baumes verarbeiten, sodaß dieser in einem Tiefendurchlauf verarbeitet wird.

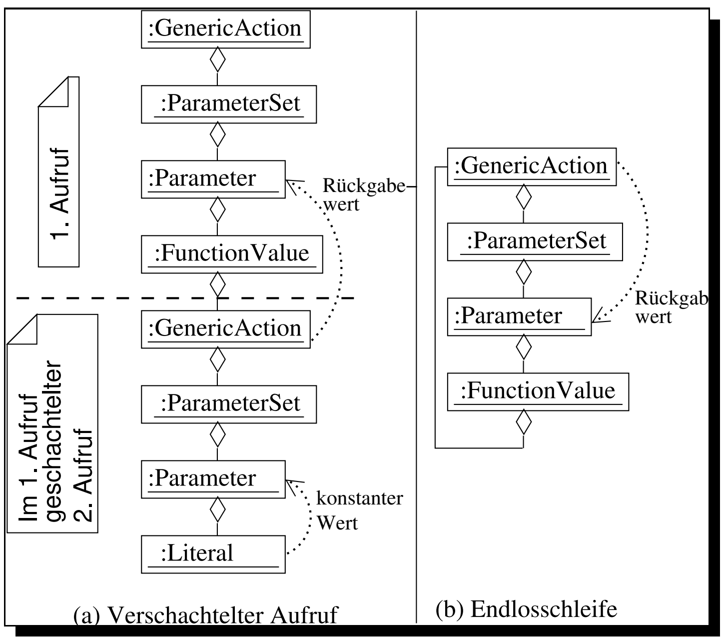

Abbildung 6.17: Ausführung eines Aktionsbaums

Zu einem Aufruf einer Methode eines Managementobjekts, etwa eines Integrationsagenten, gehören der Name des Objekts, der Name der Methode, sowie die Werte der Parameter. Letztere können wiederum Rückgabewerte von Methodenaufrufen sein, die durch die Rekursion durch den Aktionsbaum aufgerufen werden, sodaß der endgültige Wert des Parameters dem im `GenericAction`-Objekt Aufruf zugeführt werden kann (Abbildung 6.17a). Die Repräsentationsklassen erlauben im Prinzip zirkuläre Strukturen. Sollte der Ausführungsbaum zu einem Graphen entarten, der Kreise enthält, kommt es zu einer Endlosschleife, die nur durch das Versagen eines Aufrufs unterbrochen wird (Abbildung 6.17b).

Rollen und Domänen Ist für einen Aufruf kein Objektname angegeben, werden die in der Policy spezifizierten *Targets* als Zielobjekte benutzt, die aus dem Ausführungskontext ermittelt werden können.

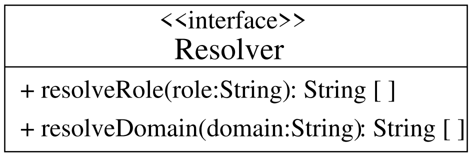

Abbildung 6.18: Schnittstelle zur Auflösung von Rollen und Domänen

Handelt es sich bei einem der Targets um eine Rolle, so wird jeweils ein Aufruf an alle Inhaber dieser Rolle durchgeführt. Die Verwaltung von Rollen und Rolleninhabern ist keine Aufgabe der Managementanwendung; es wird angenommen, daß bei der Rollenvergabe auf bereits extern definierte Rollen zurückgegriffen wird, um eine mehrfache Definition der Rollen und die damit drohende Inkonsistenz der Definitionen zu vermeiden. Zu diesem Zweck könnte beispielsweise eine für eine Sicherheitskomponente getroffene Rollenvergabe übernommen werden. Für die Zwecke des Executor wird lediglich eine Schnittstelle (Abbildung 6.18) zur Auflösung von Rollen oder Domänenausdrücken in Objektnamen definiert, die aus den Executor-Methoden benutzt werden kann. Die beiden Methoden der Schnittstelle liefern eine Reihung von Namen zurück, die z. B. mittels eines Namensdienstes (etwa dem CORBA Naming Service [OMG 97-02-08]) in Objektreferenzen aufgelöst werden können. Die Implementierung dieser Schnittstelle kann beispielsweise auf einen Verzeichnisdienst (LDAP, X.500, NIS/YP etc) abgebildet werden.

Erzeugung von Ereignissen

Ereignisse können als Aktion einer Policy erzeugt werden. Wie bereits erwähnt besitzt ein Ereignis einen Namen als eines seiner Attribute, und kann ein Payload-Objekt transportieren, dessen Struktur nicht festgelegt ist. Ereignisse ohne Payload-Objekt können einfach durch Erzeugung eines `StructuredEvent` erstellt werden. Soll dem Ereignisobjekt zusätzlich ein Paylod-Objekt beigelegt werden, kann dies, ggf. unter der Angabe von Parametern in einem ParameterSet generiert und in das `StructuredEvent` integriert werden. Die Generierung solcher Ereignisse wird von der `EventFactory` übernommen. Diese Klasse muß bei der Hinzunahme neuer Klassen von Payload-Objekten jeweils angepaßt werden.

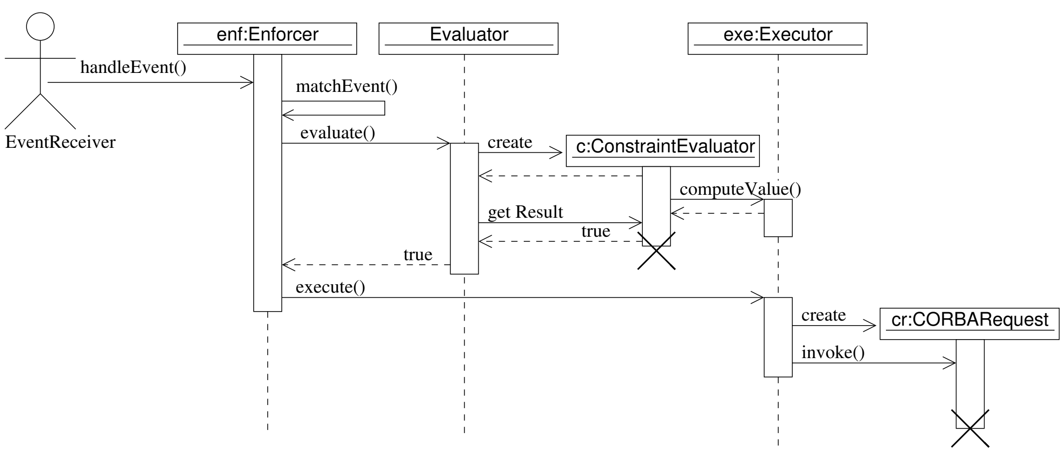

Abbildung 6.19: Reaktion der Managementanwendung auf ein empfangenes Ereignis.

Beispiel für die Verarbeitung einer Policy Das Laufzeitverhalten der Enforcer-Komponente nach Erhalt eines Ereignisses kann mit folgendem Beispiel verdeutlicht werden. Das in Abbildung 6.19 gezeigte Sequenzdiagramm beschreibt die Abläufe in den Klassen des Enforcers nach Empfang eines

Ereignisses. Die Umsetzung eines Methodenaufrufs variiert natürlich mit der zur Implementierung benutzen Verteilungsplattform; im vorliegenden Beispiel wird von einer CORBA-Umgebung ausgegangen. Das Beispiel geht weiterhin davon aus, daß eine Policy für das Ereignis spezifiziert wurde und daß die Bedingungen positiv (zu *wahr*) ausgewertet werden. Der Schwerpunkt liegt auf das Zusammenspiel von Enforcer, Evaluator und Executor; die Benutzung des Ausführungskontexts ist in der Darstellung nicht berücksichtigt.

6.5 Zusammenfassung

In diesem Kapitel wurde der Entwurf der Managementanwendung vorgestellt. Die Anwendung gliedert sich in drei Komponenten: das PolicyRepository, das für die Speicherung und Verwaltung der Policies, als auch für die Erzeugung ihrer Objektrepräsentation zuständig ist; der Enforcer, der als Reaktion auf Ereignisse die zutreffenden Policies ermittelt, ihre Bedingungen auswertet und gegebenenfalls ihre Aktion ausführt; der PolicyManager, der einerseits die Benutzerschnittstelle der Managementanwendung bereithält, andererseits die Interaktion von Repository und Enforcer koordiniert.

Die beschriebene Repräsentation der Sprachelemente der Policy Definition Language lehnt sich an das Policy Core Information Model (PCIM) der IETF an. Beim Entwurf der Objektrepräsentation von Policies wurde auf Erweiterbarkeit und enge Entsprechung zu der wohlgeklammerten PDL geachtet. Nach Beschreibung der Verwaltungsoperationen auf Policies im Repository wurde die Auswertung ihrer Ereignisse und die Ausführung ihrer Aktionen durch den Enforcer beschrieben. Die dabei spezifizierten Verfahren machen sich die Eigenschaften der sich aus der Objektrepräsentation der Policies ergebenden Bäume zunutze.

Der Entwurf legt keine Technologien für die Implementierung fest. Er ist allerdings, wie im nächsten Kapitel deutlich wird, auf eine Implementierung mit Java und XML unter Verwendung von CORBA als Verteilungsplattform ausgerichtet.

Kapitel 7

Implementierung

Dieses Kapitel vermittelt eine Übersicht der Implementierung der Managementanwendung. Nach der Festlegung des Rahmens für die Implementierung wird die Umsetzung der Policy Definition Language als XML-Anwendung beschrieben. Anschließend werden ausgewählte Implementierungsaspekte der Komponenten der Managementanwendung und Integrationsagenten angesprochen.

7.1 Kontext der Implementierung

Für die Entwicklung des Prototypen kommen eine Anzahl Technologien, Bibliotheken und Werzeuge zum Einsatz. In der nachstehenden Übersicht dieser Hilfsmittel wird ihre Notwendigkeit und Eignung für die Implementierung des Managementsystems diskutiert.

7.1.1 Verwendete Technologien

XML Die in dieser Arbeit spezifizierte Policy Definition Language wird für die Implementierung auf eine XML-Anwendung abgebildet; zu diesem Zweck wird aus ihrer EBNF eine XML-Schema-Definition abgeleitet. Auf diese Weise können mit Hilfe von Standardwerkzeugen Ausdrücke der PDL vearbeitet werden.

Flexible Agenten Zur Implementierung der Integrationsagenten wird ein bestehendes Agentensystem benutzt. Die Agenten sind bereists mit einheitlichen Managementschnittstellen ausgestattet. Dies beschränkt den Bedarf, Schnittstellen zu definieren, auf diejenigen der repräsentierten Abrechnungskomponeten. Die Managementanwendung wird ebenfalls mittels flexibler Agenten umgesetzt; einerseits vereinheitlicht sich dadurch die Implementierung der Teile des Managementsystems, andererseits erweist sich dieser Ansatz bei verteilter Verarbeitung der Policies als vorteilhaft.

CORBA Für die Implementierung von Ereignissen und Methodenaufrufen im Rahmen von Aktionen wird die *Common Object Request Broker Architecture* als Verteilungsplattform eingesetzt. Es handelt sich bei diesem Standard um eine ausgereifte Technologie, die auch anderweitig zu Managementzwecken eingesetzt wird [HAN 99]. Der *CORBA EventService* wird für den Transport von Ereignisobjekten benutzt.

Java Als Programmiersprache kommt Java zum Einsatz. Aufgrund der problemlosen Portierbarkeit von Java-Code und der in den für Java spezifizierten APIs eignet sich die Sprache sehr gut für den Einsatz in einem heterogenen Umfeld. Aus diesen Gründen wurde auch das benutzte Agentensystem in Java implementiert — ein weiterer, zwingender Grund für den Einsatz der Sprache. Eine Abbildung von CORBA auf Java existiert ebenfalls [OMG 01-06-07], ebenso wie eine Auswahl in dieser Sprache implementierte, freie XML-Parser.

7.1.2 Externe Software-Komponenten

Für die prototypische Implementierung des Managementsystems kamen die im folgenden beschriebenen Softwarepakete zum Einsatz. Es handelt sich hierbei durchgehend um in Java programmierte Komponenten.

Xerces ist ein XML-Parser-Paket des Apache-Projekts. Es enthält einen *Document-Object-Model*-konformen Parser mit Unterstützung für die DOM Level 1 bis 3, sowie einen Parser, der die *Simple API for XML* (SAX) unterstützt. Aus den in Abschnitt 7.2.2 dargelegten Gründen wird für die Managementanwendung der DOM-basierte Parser zum Einsatz kommen. Dieser unterstützt sowohl DTD-Grammatiken nach der XML-Spezifikation [XML 00], als auch Validierung mittels der XML-Schema Grammatikdefinitionen, die in der vorliegenden Implementierung zum Einsatz kommen. Für die Implementierung kam die Version 2.0.1 zum Einsatz.

MASA Die *Mobile Agent System Architecture* (MASA) ist eine Spezifikation und Implementierung eines Agentensystem für mobile Agenten [GHR 99] [KRRV01]. Die MASA-Implementierung basiert auf Java und CORBA, unterstützt sowohl mobile als auch stationäre Agenten und bietet einen Rahmen für die Entwicklung eigener, MASA-konformer Agenten. Außerdem enthält MASA einen eigenen Namensdienst, der auf die Bedürfnisse mobiler Agenten abgestimmt ist.

Orbacus ist eine CORBA-Implementierung der Firma OOC (Object Oriented Concepts) für Java und C++. Für die Implementierung der Managementanwendung kommt die Version 3.1.2 zum Einsatz. Diese Version von Orbacus bringt einen CORBA NameService und einen CORBA EventService mit. Der Einsatz dieser Version ist aus zwei Gründen zwingend erforderlich: einerseits basiert die benutzte MASA-Version auf dieser Orbacus-Version, andererseits unterstützt die genannte Orbacus-Version nur den *Basic Object Adapter* (BOA). Im Gegensatz zu ORBs, die einen *Portable Object Adapter*(POA) bereithalten, können auf BOA eingeschränkte ORBs nicht problemlos zusammen eingesetzt werden. Der Verzicht auf Orbacus in der Managementanwendung würde deshalb notwendigerweise die Umstellung der MASA-Implementierung auf einen anderen ORB voraussetzen.

7.2 Entwicklung der PDL in XML-Schema

7.2.1 Einführung in XML

Die *Extensible Markup Language* (XML), die von der W3C in [XML 00] spezifiziert wird, basiert auf SGML (*Standard Generalized Markup Language* [ISO 8879]). Das Ziel der Entwicklung von XML war es, das für viele Anwendungen zu komplizierte SGML zu vereinfachen und die Implementierung von Werkzeugen zu erleichtern. Die in SGML gebotenen Möglichkeiten wurden für XML auf eine Untermenge eingeschränkt. Ein XML-Dokument ist deshalb auch ein SGML-Dokument, umgekehrt gilt dies jedoch nur bedingt[1]. Die Einschränkung der Syntax von XML gegenüber der SGML-Syntax erlaubt es, für XML einfachere Parser einzusetzen. Da jedes Werkzeug, das zur Verarbeitung von strukturierten Daten entwickelt wird notwendigerweise einen Parser enthält, ist dies bereits eine Erleichterung bei der Implementierung solcher Werkzeuge.

Ein charakteristisches Merkmal der Sprache sind die von SGML übernommenen, durch spitze Klammern abgegrenzten Marken beziehungsweise *Tags*. Dabei weisen schließende Tags nach der offenen spitzen Klammer ein Divisionszeichen (*slash*) auf, etwa: `<tag> ... </tag>`. Eine wichtige Verschärfung der Syntaxregeln von XML gegenüber SGML stellt die Forderung nach Wohlklammerung dar; damit wird implizit die sogenannte Tagminimierung[2] eliminiert.

Das XML-Dokument

Ein XML-Dokument beginnt mit einem sogenannten *Prolog*, in dem Angaben zu seiner Grammatik gemacht werden können. Der eigentliche Inhalt des Dokuments ist im *Körper* des Dokuments zu finden. Anders als bei SGML ist die Angabe einer Grammatik für das Dokument optional. Ein XML-Dokument, das wohlgeklammert ist und auch sonst der für XML-Anwendungen vorgesehenen Syntax entspricht, heißt *wohlgeformt* (well formed). Entspricht ein wohlgeformtes Dokument zusätzlich der ihm zugeordneten Grammatik, so heißt es *gültig* (valid).

Jedes wohlgeformte XML-Dokument läßt sich als ein einziger Baum darstellen; ein Dokument mit mehreren Wurzelelementen (also ein Wald) gilt nicht als wohlgeformt. Die Knoten des Baumes entsprechen *Elementen*. Elemente sind die Grundbausteine von XML-Dokumenten. Sie besitzen ein Tag-Paar (öffnend und schließend), wobei der Name des Elements dem der Tags entspricht. Insbesondere trägt die Wurzel des Baumes den Namen des *Dokumententyps*.

Elemente bestehen aus dem oben genannten Tag-Paar, einem Inhalt (zwischen den Tags) und Attributen. Attribute werden innerhalb des öffnenden Tags angegeben, etwa:

`<tag attribut=``wert''> ... </tag>`.

Ein Spezialfall für Elemente sind die *leeren Elemente*. Sie bestehen lediglich aus einem einzigen Tag, der einen *slash* vor der schließenden spitzen Klammer aufweist, z. B. `<leererTag/>`. Sie können ebenso wie normale Elemente Attribute enthalten.

[1] Tatsächlich ist ein XML-Dokument genau dann ein gültiges SGML-Dokument, wenn es selbst gültig ist, d. h. wenn es einer explizit angegebenen DTD-Grammatik entspricht.

[2] In SGML muß einem geöffneten Tag nicht unbedingt ein entsprechender schließender Tag zugeordnet werden. Der Parser muß selbst entscheiden, wo ein solcher schließender Tag einzufügen ist.

Grammatik einer XML-Anwendung

Das Markup von Daten zum Erstellen eines XML-Dokuments geschieht mittels einer *XML-Anwendung*, die einer Klasse von XML-Dokumenten entspricht. Das prominenteste Beispiel hierzu ist XHTML [XHTML], eine Redefinition von HTML (Hypertext Markup Language) als XML-Anwendung[3].

Die Grammatik einer XML-Anwendung kann mittels der von SGML übernommenen Sprache *DTD* (Document Type Definition) definiert werden. Die Sprache DTD besitzt eine eigene, von SGML/XML verschiedene Syntax. Produktionen der Grammatik können sowohl im Prolog eines XML-Dokuments als auch extern (in separaten Dateien) vorkommen.

Als Alternative zu DTDs wurde *XML-Schema* in [XMLS-0], [XMLS-1] und [XMLS-2] definiert. XML-Schema ist selbst eine XML-Anwendung und benutzt daher XML-Syntax. Zusätzlich bietet es über die von DTD hinausgehende Möglichkeiten. Auf XML-Schemata wird im Prolog von XML-Dokumenten bezug genommen.

7.2.2 XML-Schema

Die Definition einer Grammatik mit XML-Schema erfolgt durch Angabe der Elemente und Attribute, die in den Dokumenten vorkommen sollen. Dies geschieht mittels der Elemente `<element>` und `<attribute>` von XML-Schema. Zusätzlich zur Angabe des Element- oder Attributnamens muß sein *Typ* angegeben werden.

In einem Schema werden die *Typen* von Elementen definiert, die mit Verbunden (`record`, `struct`) in Programmiersprachen vergleichbar sind. In der Typdefinition sind die möglichen, erlaubten und verbotenen Inhalte eines Dokumentteils beschrieben. Dabei unterscheidet man zwischen einfachen (SimpleType) und zusammengesetzten (ComplexType) Typen. XML-Schema stellt eine Anzahl grundlegender Datentypen als `simpleType` zur Verfügung. Benutzerdefinierte Typen können durch Ableitung von vordefinierten Typen oder durch Komposition erstellt werden. Nachstehend werden Altenativen zur Definition solcher Typen beschrieben. Ein Beispiel ist in Listing 7.1 angegeben: Der Typ `mySimpleType` ist eine ganze Zahl zwischen null und zwölf; seine Definition geschieht durch Einschränkung des Wertebereichs des vordefinierten Typs `xsd:integer`. Der Typ `myComplexType` besteht aus den zwei Elementen `myEl1` und `myEl2`, sowie einem Attribut *myAtt*. Das Element `myEl1` ist vom Typ `mySimpleType`; das bei der Deklaration des Elements `myEl2` angegebene Attribut `minOccurs=''0''` gibt an, das dieses Element mindestens null mal vorkommen soll — es handelt sich also um ein optionales Element.

SimpleType Typen, die selbst keine Elemente enthalten, können als `simpleType` deklariert werden. Der Typ des Elements ist dann aus in [XMLS-2] festgelegten Typen wählbar. Einschränkungen des Wertebereichs des vordefinierten Typs sind möglich, wie auch Aufzählungen möglicher Werte (*enumeration*) und reguläre Ausdrücke.

ComplexType Für einen mittels `complexType` definierten Elementtyp ist die Angabe von Attributen, sowie untergeordneten bzw. enthaltenen Elementen möglich. Die Reihenfolge der enthaltenen

[3] XHTML entspricht HTML in der Version 4.

Elemente, eine Auswahl an möglichen Elementen und die Anzahl der Vorkommen eines Elements können festgelegt werden.

Listing 7.1: Beispiele für die Anwendung von *simpleType* und *complexType*.

```
<xsd:simpleType name=``mySimpleType`` >
    <xsd:restriction base="xsd:integer">
        <xsd:minInclusive value="0"/>
        <xsd:maxInclusive value="12"/>
    </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name=``myComplexType``>
    <xsd:sequence>
        <xsd:element name=``myEl1`` type=``mySimpleType``/>
        <xsd:element name=``myEl2`` type=``xsd:float`` minOccurs=``0``/>
    </xsd:sequence>
    <xsd:attribute name=``myAtt`` type=``xsd:string``/>
</xsd:complexType>
```

Vererbung und Abstrakte Typen Ein Typ kann als *abstrakt* deklariert werden und darf in diesem Fall nicht instantiiert werden: er kann nur als Obertyp benutzt werden. Ein gültiges XML-Dokument darf somit keine Elemente eines abstrakten Obertyps enthalten. Die Ableitung von Typen von einem Obertyp kann in XML-Schema mittels zweier Mechanismen geschehen:

- Erweiterung — der abgeleitete Typ fügt den geerbten Eigenschaften eigene hinzu.
- Einschränkung — der abgeleitete Typ wird auf eine Untermenge der geerbten Eigenschaften eingeschränkt.

Beide Mechanismen sind sowohl für einfache als auch für komplexe Typen vorhanden. Die Erweiterung bzw. Einschränkung bezieht sich je nach Typ entweder auf den Wertebereich oder auf den Inhalt. Einschränkung und Erweiterung können kombiniert werden.

Eine weitere interessante Möglichkeit, um erweiterbare Schemata zu definieren, ist die Ableitung von Typen eines Schemas von Typen, die in einem anderen Schema definiert wurden. So kann eine Hierarchie von Schemadefinitionen erstellt werden, die — ähnlich wie bei Vererbungshierarchien objektorientierter Programmiersprachen — sowohl die Gemeinsamkeiten der Anwendungen an der Spitze der Hierarchie erfaßt, als auch spezialisierte Schemata für einzelne XML-Anwendungen enthält. Die Beschreibung solcher Hierarchien, die eine gute Erweiterbarkeit der definierten XML-Anwendungen gewährleisten, würden allerdings den Rahmen dieses Abschnitts überschreiten. Details finden sich in den Dokumenten des W3C, beispielsweise in [XMLS-1] und [DOM-AS].

Gültige XML-Dokumente

Policies werden in einer XML-basierten Sprache definiert. Dabei wird gefordert, daß die resultierenden XML-Dokumente gültig sind, d. h. daß sie der für die PDL angegebenen Grammatik entsprechen; erst dann werden die in einem Dokument angegebenen Policies als syntaktisch korrekt akzeptiert. Aus dieser Forderung ergibt sich die Möglichkeit, manche Fehler in den Policy-Definitionen schon im Parserlauf zu entdecken. Die Art und die Anzahl der entdeckten Fehlertypen hängt dabei direkt mit der Definition der Grammatik zusammen.

Namespaces Werden in zwei verschiedenen Schemata gleichnamige Elemente deklariert, können Konflikte in Dokumenten auftreten, die beide Schemata benutzen. Ein im Dokument vorkommendes Element kann nicht eindeutig einer Typdefinition zugeordnet werden. Diesem Problem begegnen die *XML-Namespaces* [XMLNS]. Sie erlauben es, in XML-Dokumenten *Prefixe* zu deklarieren, die an eine Schemadefinition gebunden sind. In der Praxis wird ein Name an einen URI (*Uniform Resource Identifier*, siehe [BLFM 98]) gebunden, der auf ein Schema verweist. Die in dieser Grammatik spezifizierten Tags können dann mit dem als Präfix angegebenen Namen eindeutig identifiziert werden. Dabei wird der Name mit einem Semikolon von dem Tag getrennt, wie z. B. `<xsd:string>`.

Verarbeitung von XML-Dokumenten

Eine zentrale Komponente von Anwendungen, die XML-Dokumente verarbeiten, ist der XML *Parser*. Aufgrund der Einschränkungen, die der XML-Syntax auferlegt wurden (siehe oben), sind XML-Parser einfacher zu implementieren als z. B. Parser für SGML. Dies führte in den letzten Jahren zu zahlreichen Implementierungen, die teils kommerziell, teils frei verfügbar sind. Um eine Anwendung nicht auf eine bestimmte Parserimplementierung festlegen zu müssen, wurden Standards entwickelt, die unter anderem Schnittstellen zu den Parsern definieren. Im folgenden werden zwei dieser Standards beschrieben.

Document Object Model

Das DOM [DOM98] definiert für die Objekte, die aus einem XML- oder HTML-Dokument erzeugt werden, ein Objektmodell und eine Reihe von Schnittstellen. Die Schnittstellen sind mittels OMG IDL (siehe [CORBA 2.2]) angegeben und nicht an eine bestimmte Programmiersprache gebunden.

Ein DOM-konformer Parser erstellt aus einem XML-Dokument einen Objektbaum, dessen Knoten den Elementen, Attributen und Inhalten des Dokuments entsprechen. Die Knoten (*Node*) können als die grundlegende Struktur im Objektmodell betrachtet werden. Ihre Schnittstelle erlaubt es, den Baum zu durchlaufen, Kinderknoten zu erstellen, zu löschen und zu modifizieren. Spezialisierte Knoten repräsentieren Elemente und Attribute des XML-Dokuments, Inhalte von Elementen, oder das Dokument selbst (Wurzelknoten). Dieser Sachverhalt wird exemplarisch an der Vererbungshierarchie der Schnittstellen einer Java-Implementierung von DOM in Abbildung 7.1 dargestellt.

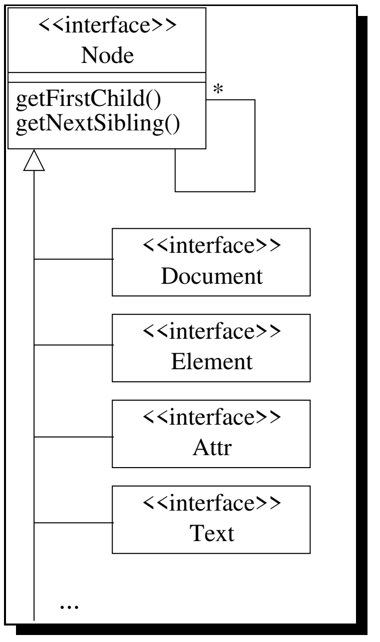

Abbildung 7.1: Hierarchie der Schnittstellen im DOM

Anwendungen, die einen DOM-Parser benutzen operieren somit auf einem von dem Parser erstellten, fertigen Objektbaum. Das Objektmodell ist unabhängig von der zu verarbeitenden XML-Anwendung — sie ist nicht an eine bestimmte Struktur/Grammatik der Dokumente gebunden. Diese Vorgehensweise ist — wenn auch bequem — nicht immer die effizienteste Art, Informationen aus einem XML-Dokument auszuwerten und zu manipulieren. DOM setzt voraus, daß das gesamte Dokument verarbeitet und vollständig in einer Datenstruktur zur Verfügung gestellt wird. Werden aus einem

Dokument nur bestimmte Inhalte gebraucht, verursacht dieser Ansatz unnötigen Verbrauch von Ressourcen.

Simple API for XML

Eine weit einfachere Schnittstelle für XML-Parser stellt die *Simple API for XML* (SAX, vgl. [Brow 02]) dar. Sie gibt kein Objektmodell für das XML-Dokument vor. Vielmehr basiert dieser Ansatz auf einem Durchlauf des Dokuments, bei dem jedes angetroffene Element, Attribut etc. ein *Ereignis* generiert. Der Benutzer ist, anders als bei DOM, selbst dafür verantwortlich, ein Objektmodell für die zu verarbeitende XML-Anwendung zu entwerfen. Anhand der beim Durchlauf des Dokuments gefeuerten Ereignisse, werden von einem (ebenso vom Benutzer zur Verfügung zu stellenden) *Document Handler* die benötigten Inhalte in das Objektmodell eingefügt; für die Anwendung irrelevante Ereignisse können dabei einfach ignoriert werden.

Der Einsatz eines SAX-Parsers für die Verarbeitung tief verschachtelter Strukturen erfordert Sorgfalt bei der Implementierung, da der Parser lediglich Beginn und Ende einer Schachtelung (d. h. das Auftreten von öffnendem bzw. schließendem Tag) signalisiert; die Verfolgung des Kontexts und der jeweiligen Schachtelungstiefe bleibt Aufgabe der Anwendung.

DOM versus SAX

Beide APIs haben Vor- und Nachteile bezüglich ihrer Eignung für die Umsetzung einer XML-basierten Policy Definition Lanugage. Die Erwartungen bezüglich der Verarbeitung von in XML notierten Policies und Policy-Teilen zeigen keine der beiden als „natürliche" Wahl auf. Für DOM sprechen folgende Annahmen:

- Alle Policies in einem Dokument müssen mindestens einmal ausgewertet werden.
- Policies und Policy-Teile sind innerhalb eines Dokuments in beliebiger Reihenfolge angeordnet. Um in einer Policy referenzierte, global definierte Policy-Teile zu ermitteln, können mehrere Läufe durch das Dokument notwendig werden. Dies ist bei Benutzung eines DOM-Objektmodells ohne Wiederholung des Parserlaufs möglich.

Die Gründe, die für SAX sprechen, sind mit der Fokussierung auf eine einzige Grammatik verbunden:

- Für Policies muß ein angepaßtes Objektmodell definiert werden — das durch DOM angegebene Modell ist zur weiteren Verarbeitung nicht geeignet.
- Es kann schon im Parserlauf nach bestimmten Elementen im Dokument gesucht werden.

Anders als bei XML-Anwendungen zur bloßen Repräsentation von Daten, ist die Überprüfung von PDL-Konstrukten noch vor ihrer Auswertung ein ausschlaggebendes Kriterium. Dies ist nur bei der Benutzung eines DOM-Parsers gegeben; dieser meldet eventuelle Fehler im XML-Dokument bereits bevor das erzeugte Objektmodell zur weiteren Verarbeitung zur Verfügung steht.

Die Referenzierbarkeit von Policy-Bestandteilen stellt zudem ein zentrales Merkmal der PDL dar. Auch die Perspektive von späteren Erweiterungen der Sprache zum Einsatz in anderen Managementbereichen legt nahe, einen DOM-basierten Ansatz im Interpreter für die PDL zu wählen.

Die in Kapitel 5 definierte Policy Definition Language wurde zur leichten Abbildung auf eine XML-Anwendung konzipiert. In diesem Abschnitt erfolgt eben diese Umsetzung, unter Benutzung von

XML-Schema als Metasprache für die Definition der Grammatik. Die Abbildung auf eine XML-Anwendung geht „von außen nach innen” vor. Es werden also zunächst Elemente wie `policySet` und `policy` beschrieben; danach werden ihre Bestandteile angegeben. Die Angabe der XML-Schema-Definitionen geschieht aus Gründen der Übersichtlichkeit auszugsweise — die vollständige Schemadefinition befindet sich im Anhang.

Die Grammatik der Policy Definition Language besteht aus Definitionen von (zum Teil abstrakten) Obertypen, Typen der Policyteile und dem Typ für Policies. Eigenschaften, die mehreren Policy-Bestandteilen zugeordnet werden können werden in den Obertypen zusammengefaßt. Die Typdefinitionen für die einzelnen Bestandteile werden von diesen Obertypen abgeleitet. Gemeinsamkeiten zwischen den Typen der PDL werden — ähnlich wie bei der Vererbung in einer objektorientierten Programmiersprache — mittels eines Obertyps ausgedrückt.

7.2.3 Referenzierbarkeit und Prozeßzugehörigkeit

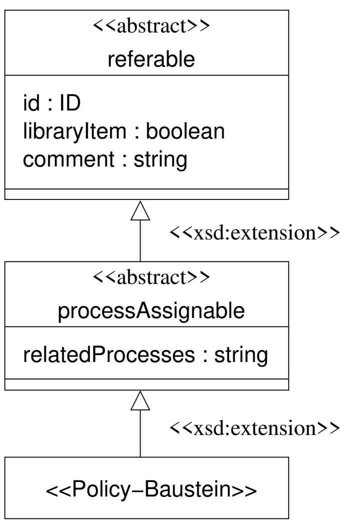

Abbildung 7.2: Abstrakte Obertypen der Policy-Bausteine

Unter den Gemeinsamkeiten der Policy-Bausteine befinden sich ihre *Referenzierbarkeit*, ihre Zuordnung zu Prozessen, sowie die Möglichkeit, einen Baustein als „Bibliothekseintrag” zu kennzeichnen. Die Referenzierbarkeit erfordert eine eindeutige ID für jeden Baustein. Für die optionale Zuordnung zu Prozessen muß innerhalb des Bausteins eine Liste von Prozessen angegeben werden können. Da Prozessen zugeordnete Bausteine implizit referenzierbar sein müssen, werden diese Merkmale mittels der in Abbildung 7.2 skizzierten Vererbungshierarchie eingebracht. Da nur referenzierbare Bestandteile in eine Bibliothek aufgenommen werden können, werden diese Eigenschaften in einem abstrakten Typ zusammengefaßt. Das in der Abbildung dargestellte UML-Diagramm ist wie eines für die Modellierung einer Klassenhierarchie zu lesen; `<<xsd:extension>>` entspricht dabei der Spezialisierung entsprechend einer objektorientierten Programmiersprache wie etwa C++ oder Java.

In Listing 7.2 wird die Typdefinition von `referable` angegeben, die als Basis für die Typen der meisten grundlegenden, referenzierbaren Policy-Bausteine sowie dem für Policies benutzt wird.

Listing 7.2: Obertyp für referenzierbare Policy-Bausteine

```
<xsd:complexType name="referable" abstract="true">
        <xsd:sequence>
               <xsd:element name="id" type="xsd:ID"/>
               <xsd:element ref="comment" />
               <xsd:attribute name="libraryItem"
                       type="xsd:boolean"
                       use="optional"
                       default="false"/>
        </xsd:sequence>
</xsd:complexType>
```

7.2.4 Definition von Policy-Bausteinen in XML

Eine vollständige Beschreibung der XML-Umsetzung der PDL würde den Rahmen dieses Kapitels sprengen. Statt dessen wird exemplarisch die Definition der `Action` vorgestellt, sowie ein Beispiel für eine einfache Policy in XML. Das komplette XML-Schema für die PDL befindet sich im Anhang.

Typdefinition für Action

Eine Aktion der PDL besteht aus einer oder zwei `GenericAction`-Elementen. Das erste, obligatorische, spezifiziert den auszuführenden Methodenaufruf, oder das zu erzeugende Ereignis. Das zweite, optionale, wird dann benötigt, wenn die Ausführung des ersten versagt. Aktionen können global, als „freie” Bausteine definiert werden, sowie Prozessen zugeordnet werden. Aus diesem Grund wird zur Typdefinition `processAssignable` als Basistyp herangezogen. Die Typdefinition in Listing 7.3 spezifiziert die Elemente `default` für die „Standardaktion”, sowie `error` für die optionale Aktion zur Fehlerbehandlung.

Listing 7.3: Typdefinition für `action`

```
<xsd:complexType name="actionType">
 <xsd:complexContent>
  <xsd:extension base="pdl:processAssignable">
        <xsd:sequence>
                <xsd:element name="default" type="genericActionType"/>
                <xsd:element name="error" type="genericActionType"
                        minOccurs="0"/>
        </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

Beispiel für eine Policy in XML

Das Listing 7.4 zeigt ein komplettes XML-Dokument, das eine Policy, sowie global definierte Bausteine enthält. Das Dokument enthält außer der Policy-Definition die Angabe eines `event`-Definition, sowie einer `action`-Definition. Letztere enthält lediglich die obligatorische Standardaktion. Die Angabe eines Objektnamens fehlt, sodaß entsprechend der im Entwurf festgelegten Vorgehensweise für die Ausführung das `target` der Policy substituiert wird. Die Policy ist den Teilprozessen *deploy meters* und *usage accounting* zugeordnet (vgl. Tabelle 4.1); tatsächlich löst sie den Übergang zwischen diesen Teilprozessen aus.

Listing 7.4: Beispiel für eine Policy in XML

```
<?xml version="1.0" encoding="UTF-8"?>

<policySet>
        <comment> Beispiel fuer Policies und Policy-Bauteile </comment>
        <policy id="0000f1" enabled="true" version="0.8">
                <comment>
                        Ein Meter soll sofort nach seiner
                        Installation gestartet werden
```

```
        </comment>
        <relatedProcesses>
            deployMeters
            usageAccounting
        </relatedProcesses>
        <targetSet>
            <target>
                <domain>/mgmt/users/premium</domain>
                <entity>ThroughputMeter</entity>
            </target>
        </targetSet>
        <eventSet> <eventRef>eded05</eventRef> </eventSet>
        <actionSet> <actionRef>ad0001</actionRef></actionSet>
        <policyDescriptor>
            <createdBy>VAD</createdBy>
            <dateOfCreation>20021119</dateOfCreation>
        </policyDescriptor>
    </policy>
    <event id="eded05">
        <comment>
            Ein Metering-Agent wurde
            erfolgreich installiert.
        </comment>
        <eventName>ThroughputMeterReady</eventName>
    </event>
    <action id="ad0001">
        <comment>Starte Metering-Agent.</comment>
        <default>
            <invoke>
                <method>start</method>
            </invoke>
        </default>
    </action>
</policySet>
```

7.2.5 Übersetzung der XML-Policies

Der Übersetzungsvorgang verläuft in zwei Phasen[4]. In der ersten Phase erfolgt der Scanner/Parser-Lauf, in dem ein XML-Parser aus dem PDL-Text einen Syntaxbaum erzeugt, der dem *Document Object Model* entspricht. Die zweite Phase erzeugt aus dem Syntaxbaum, entsprechend des in Kapitel 6 beschriebenen Verfahrens, die Objektrepräsentation der im Dokument spezifizierten Policies und Policy-Bausteine (Abbildung 7.3).

[4] Das Übersetzungsverfahren stellt sicher, daß die zwei Phasen vollständig und nacheinander ausgeführt werden. Es handelt sich also eher um *Schritte*, als um Phasen im üblichen Sprachgebrauch.

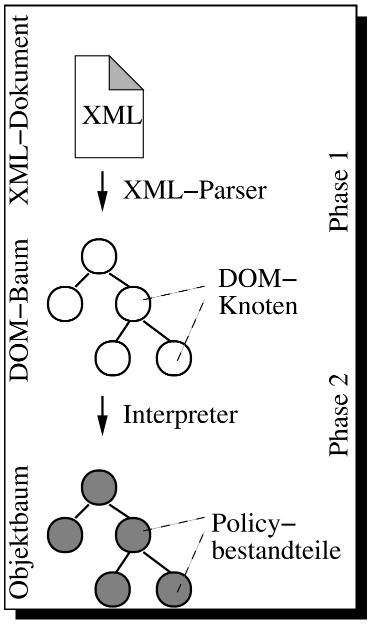

Abbildung 7.3: Übersetzung der in der XML-PDL formulierten Policies in eine interne Repräsentation

Erste Phase Der XML-Parser erzeugt aus dem XML-Dokument einen DOM-Baum. Dieser besteht aus Knoten (der Klasse `org.w3c.dom.Node`), die den Elementen des XML-Dokuments entsprechen. Werden während des Parser-Laufs Fehler entdeckt, beispielsweise Syntaxfehler, so wird das XML-Dokument zur Korrektur zurückgewiesen. Anderenfalls wird als Übergang in die zweite Phase die Wurzel des Baumes an die abstrakte Klasse `Policy` übergeben, die mittels einer *Factory*-Methode die zweite Phase durchführt.

Zweite Phase Aus dem DOM-Baum wird die Objektrepräsentation der Policies und Policy-Bausteine generiert. Dazu werden die Söhne der Wurzel des DOM-Baums betrachtet und zunächst diejenigen verarbeitet, die Policy-Bausteinen entsprechen (*target*, *action* etc); Söhne der Dokumentwurzel repräsentieren diese Knoten global definierte Komponenten. Die freien Policy-Bausteine werden anhand ihrer ID abgelegt, um für die Auflösung von Referenzen in Policies zur Verfügung zu stehen. Eine fehlende ID in einem solchen Baustein gilt als Fehler, ebenso eine nicht eindeutige ID; eine ID-lose, „freie" Komponente kann nicht referenziert werden und ist somit für das Laufzeitsystem unbrauchbar, während für eine doppelte Belegung der ID durch zwei gleichzeitig existierende Bausteine nicht entschieden werden kann, welcher von ihnen mit der Referenz gemeint ist.

7.3 Implementierung der Managementanwendung

Die Managementanwendung wurde auf Basis der MASA als eine Menge von Agenten implementiert. Die Agenten übernehmen jeweils die Funktionalität einer der Komponenten der Managementanwendung . Die in Kapitel 6 beschriebenen Klassen wurden zunächst unabhängig von MASA, jedoch mit Rücksicht auf seine Anforderungen implementiert. Danach wurden für die Komponenten MASA-Agenten erstellt. Zum Managementsystem gehören außer der Managementanwendung auch Integrationsagenten, sowie Hilfsmittel zur Auflösung von Namen und Rollen, sowie zur Propagierung von Ereignisobjekten. Abbildung 7.4 skizziert die Implementierung mit MASA und CORBA im Überblick.

7.3.1 MASA im Überblick

Um die Beschreibung der Implementierung zu vereinfachen, lohnt sich ein Blick auf die wichtigsten in MASA benutzten Konzepte. Die Architektur spezifiziert Agenten, die innerhalb sogenannter *Agentensysteme* ausgeführt werden. Diese stellen den Agenten eine einheitliche Laufzeitumgebung zur Verfügung. Umgekehrt müssen die Agenten auch eine einheitliche Schnittstelle aufweisen, um beispielsweise durch das Agentensystem gestartet und angehalten werden zu können.

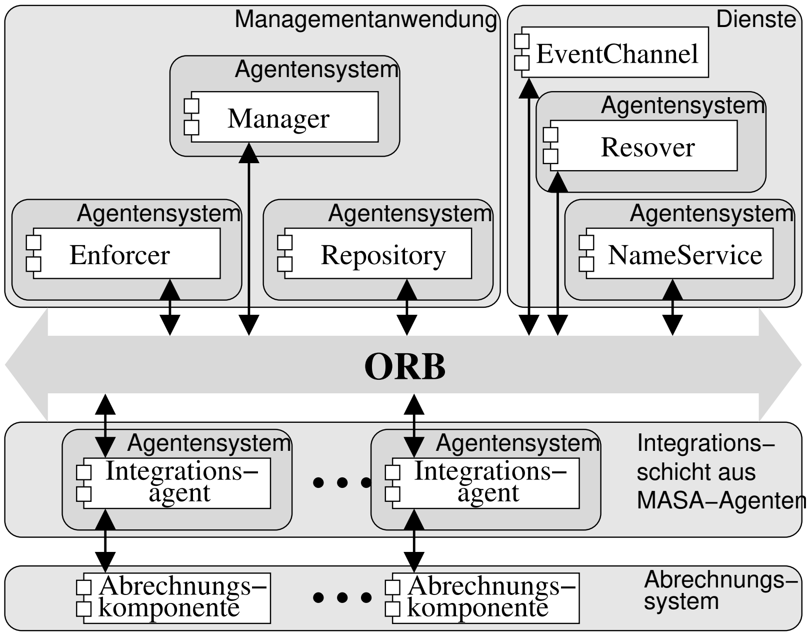

Abbildung 7.4: Implementierungsskizze

MASA unterstützt *mobile* Agenten, die zwischen Agentensystemen transportiert werden können. Sollen zwei Agenten miteinander kommunizieren, müssen sie in der benutzten CORBA-Umgebung die Möglichkeit haben, eine Objektreferenz (z. B. als IOR) anhand eines Namens zu ermitteln. Da der Aufenthaltsort der Agenten nicht konstant ist, und Agenten zu gewissen Zeitpunkten gerade „in Bewegung" sind, stellt MASA einen eigenen Namensdienst zur Verfügung, der — anders als der CORBA NameService — diese Spezifika berücksichtigt. Das ist für die Implementierung der Managementanwendung deshalb von Bedeutung, weil der MASA-Namensdienst zwar IORs als Antwort auf eine Anfrage zurückliefert, allerdings nicht über GIOP/IIOP anzusprechen ist (wie ein CORBA NameService), sondern mittels eines HTTP-Requests. Im weiteren bezeichnet der Begriff *Namensdienst* oder *NameService* den MASA-Namensdienst.

Da der von MASA eigentlich benötigte *CORBA NotificationService* mit dem zur Implementierung benutzten *Orbacus* nicht mitgeliefert wird, wurde eine Klasse **StructuredEventWrapper** erstellt, mit Hilfe derer der Ereignistyp des NotificationService emuliert werden kann. Die in dieser Arbeit entwickelte Managementanwendung erweitert diese Klasse zur Erzeugung von Ereignisobjekten mit *Payload*.

7.3.2 Implementierung der Hauptkomponenten

Der für den Betrieb innerhalb von MASA notwendige Agentencode ist nicht in die Komponenten integriert, sondern wirkt ähnlich wie ein *Stub* einer Verteilungsplattform, wobei er die Schnittstellen der Komponenten repliziert.

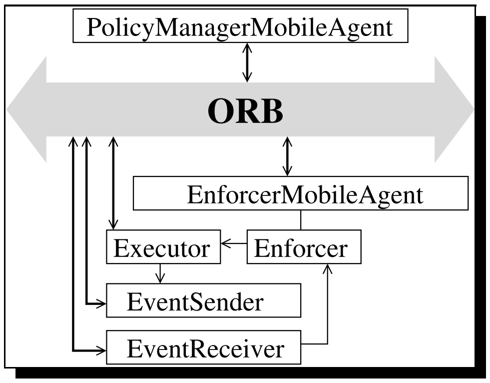

Abbildung 7.5: Kommunikation zwischen Komponenten

Diese Vorgehensweise wirkt sich auch auf die Beziehung der Klassen zu den CORBA-Bibliotheken aus: auf CORBA-Funktionalität wird im Code der Managementanwendung nur dort direkt zugegriffen, wo eine durch MASA nicht zur Verfügung gestellte Funktionalität benötigt wird. Dies sind einerseits das Senden und Empfangen von Ereigsnissen an/von einem CORBA Event-Channel (MASA unterstützt die für die Ereignisobjekte spezifizierten Payload-Objekte nicht), andererseits zum dynamischen Erstellen und Ausführen von Operationaufrufen. CORBA-Kommunikation zwischen den verteilten Komponenten werden von dem Agentencode abgewickelt, wie in Abbildung 7.5.

Der Agentencode ist auch dafür verantwortlich, die Ausführung seiner Komponente zu starten, eine Aufgabe was in einer selbständigen Java *Application* von der `main`-Methode übernommen worden wäre. Dazu werden die für den Start notwendigen Klassen instantiiert und initalisiert.

Manager, Repository und Enforcer wurden als drei verschiedene Agenten umgesetzt. Entsprechend der MASA-Nomenklatur heißen diese Agenten

- `PolicyManagerMobileAgent`,
- `RepositoryMobileAgent` und
- `EnforcerMobileAgent`.

Zur Erzeugung der drei Agenten wurden die Schnittstellen der ihnen jeweils entsprechenden Komponenten der Managementanwendung in IDL formuliert und mit einem IDL-Java-Übersetzer die für die Agenten notwendigen Klassen generiert.

Speicherung der XML-Dateien

Das Repository ist so ausgelegt, das es mit Hilfe der zur Verwaltung der XML-Dokumente benutzten Klasse `PolicyStorage` mehrere Typen von Speicherungsmedien benutzen kann. Mit Hilfe der in `Storage` definierten Schnittstelle wird von der Art der Speicherung abstrahiert. Das Paket `engine.storage` enthält mit `FileStorage` eine Implementierung der Schnittstelle für Speicherung der XML-Dokumente in Dateien. Klassenskelette existieren außerdem in `DBStorage` und `NetworkStorage` für Speicherung in einer Datenbank bzw. auf einem Webserver. Das implementierte Repository liest und schreibt XML-Dokumente als Textdateien mit Hilfe der Klasse `FileStorage`.

Interaktion mit der Managementanwendung

Der PolicyManager bietet mittels einer graphischen Benutzeroberfläche (GUI) die Möglichkeit, Policies zu aktivieren bzw. zu deaktivieren. Außerdem können die sich im System befindlichen Policies eingesehen werden. Da die Erstellung und Modifizierung von Policies und Policy-Bausteinen direkt im XML-Format aufgrund fehlender Werkzeuge gegenwärtig nicht praktikabel ist, wurde die GUI um diese Möglichkeiten erweitert. Abbildung 7.6 zeigt das Hauptfenster der PolicyManager-GUI; in Abbildung 7.7 wurde die *Tabpane* zur Editierung des *Subject*-Felds einer Policy aufgeschlagen.

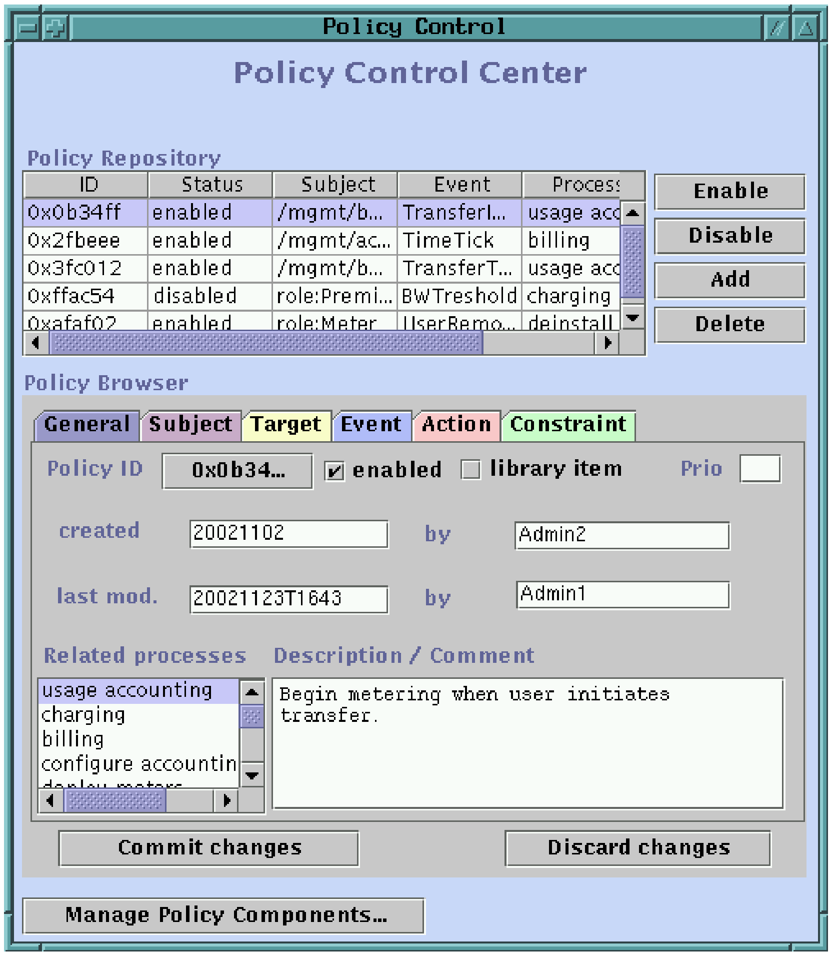

Abbildung 7.6: Graphische Oberfläche der Managementanwendung

7.3.3 Hilfsmittel der Managementanwendung

Die in Abbildung 7.4 als *Dienste* angedeuteten Hilfsmittel wurden zum Teil von MASA zur Verfügung gestellt, zum Teil mußten sie neu implementiert oder modifiziert werden. Der Namensdienst von MASA erfüllt die Anforderungen des Managementsystems und konnte deshalb ohne Veränderungen zum Lokalisieren von Objekten benutzt werden.

Die in MASA verfügbare Implementierung einer EventChannel-*Application* in der Klasse `AgentChannel` konnte mit nur geringen Veränderungen für die Managementanwendung angepaßt werden. Ebenso mußte die Klasse `StructuredEventWrapper` von MASA um die Unterstützung von Payload-Objekten für Ereignisse erweitert werden. Bei den Veränderungen wurde darauf geachtet, daß der resultierende Code zu dem restlichen MASA-Code kompatibel ist, sodaß dieselben Klassen sich zum Betrieb mit den Agenten der Managementanwendung, wie auch mit anderen Agenten eignen.

Ein simplistischer Agent zur Auflösung von Rollen in Namen wurde zur Demonstration erstellt. Er implementiert die in Kapitel 6 spezifizierte Schnittstelle `Resolver`.

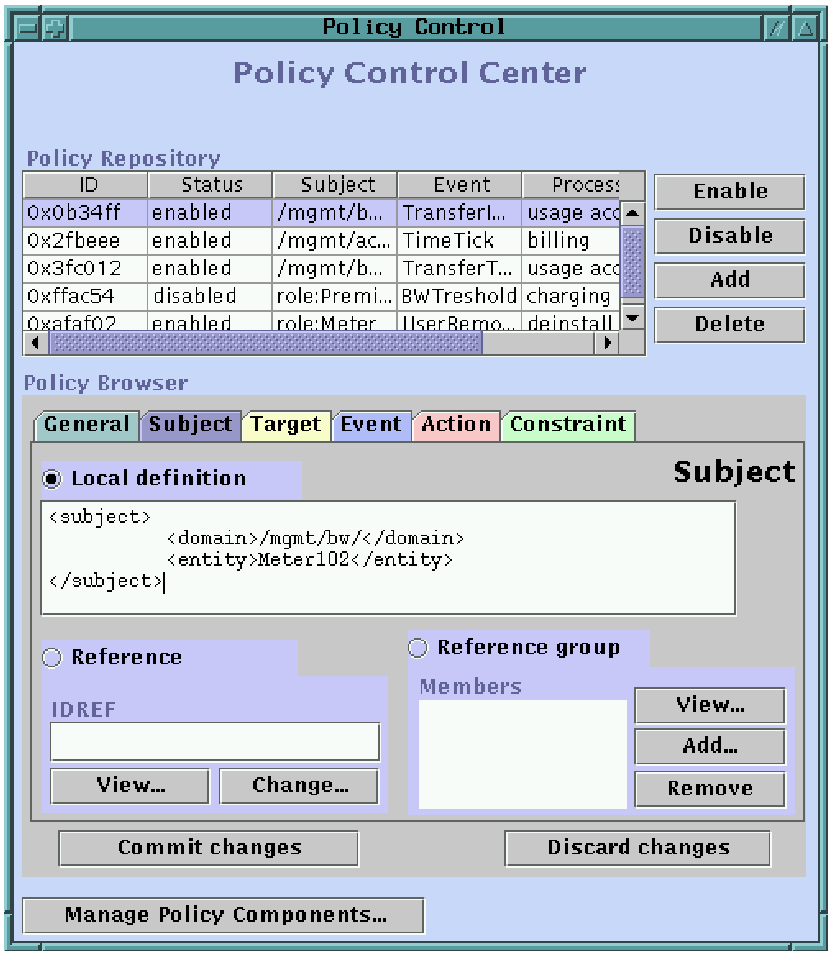

Abbildung 7.7: Editierung eines Policy-Feldes

7.3.4 Integrationsagenten

Die Implementierung von Integrationsagenten kann nur mit Kenntnis der Schnittstellen der Abrechnungskomponenten vorgenommen werden. Auch die Spezifikation der einheitlichen Schnittstellen zur Kommunikation mit der Managementanwendung sind nicht Teil dieser Arbeit. Zu Demonstrationszwecken wurde allerdings ein Versuch unternommen, experimentelle Schnittstellen zu definieren und einen solchen Agenten zu implementieren.

Generische Schnittstellen

Im Managementsystem werden die an der Abrechnung beteiligten Komponenten durch Integrationsagenten repräsentiert. Diese sollen gegenüber der Managementanwendung einheitliche Schnittstellen bieten. Die Schnittstellen gängiger Implementierungen von Abrechnungskomponenten sind nicht hinreichend — oder überhaupt nicht — bekannt. Die folgenden exemplarisch angegebenen Schnittstellen für Integrationsagenten wurden aus der Funktion der jeweiligen Komponenten abgeleitet. Sie verstehen sich als Annahme bezüglich der realen Schnittstellen.

Basisschnittstelle Eine Abrechnungskomponente ist im Grunde ein Programm, das gestartet und angehalten werden kann. Diese grundlegenden Operationen werden für alle Integrationsagenten vorgeschrieben, zusammen mit einer Operation zur Feststellung des Status der durch den Agenten repräsentierten Abrechnungskomponente.

void	**startup();**	startet die Komponente, wenn sie noch nicht läuft.
void	**shutdown();**	hält die Komponente an, wenn sie läuft.
boolean	**isRunning();**	ermittelt den Status der Komponente; liefert *true*, wenn die Komponente läuft, *false* sonst.

Schnittstelle für Metering-Agenten Eine Metering-Komponente sammelt Daten bezüglich der Nutzung von Ressourcen. Der sie repräsentierende Agent wandelt bei Bedarf die Daten in ein erwünschtes Format um und leitet sie weiter. Die Methoden zum Starten und Anhalten der Messung beziehen sich nur auf die Messung selbst. Ihre Aufgabe ist es nicht, die Komponente zu starten oder anzuhalten.

void	**startMeter();**	startet die Messung.
void	**stopMeter();**	hält die Messung an.
void	**resetMeter();**	löscht die bisher ermittelten Daten.
void	**setUnit(in string unit);**	setzt die Einheit der Messung. Je nach Art der Metering-Komponente können verschiedene Einheiten in Frage kommen, z. B. bezüglich Volumen (byte, kb, Mb etc.), Zeit (m, s, ms) oder Durchsatz (byte/sekunde etc.). Diese Methode kann benutzt werden, um implizit die Granularität der Messung zu bestimmen.
string	**getUnit();**	liefert die gegenwärtige Einheit der Messung zurück.
long	**getDataType();**	liefert eine Beschreibung des Datentyps der gemessenen Daten zurück. Dieser wird durch eine ganze Zahl repräsentiert, die einer von mehreren noch festzulegenden Konstanten entsprechen muß. Beispiele dafür wären ganze Zahlen (für Zähler) oder Reihungen (für Meßreihen).
any	**getCollectedData();**	liefert die bisher gemessenen Daten in einem CORBA Any-Objekt gekapselt zurück. Diese Methode wurde zur Benutzung durch einen anderen Integrationsagenten ausgelegt, etwa dem Repräsentanten einer Collector-Komponente. Dieser könnte die erhaltenen Daten in ein für „seine” Komponente geeignetes Format bringen und sie dann dem Collector zuführen.

Implementierung eines Integrationsagenten

Die Integrationsagenten werden als mobile Agenten der MASA implementiert. Da keine realen Abrechnungskomponenten zur Verfügung stehen, wurde als Ersatz ein MASA-Agent zur Messung der von einer Ethernet-Karte übertragenen Datenmenge erstellt. Die Messung ist auf Systemen möglich, die das in *proc(5)* (Linux man-pages) beschriebene *process information pseudo-filesystem* besitzen.

Der Agent weist die experimentielle generische Schnittstelle auf, die für Metering-Agenten spezifiziert wurde. Durch die Implementierung der Tätigkeit einer Metering-Komponente (Messung) und der Managementschnittstelle in einem einzigen Agenten erfüllt er einerseits die Funktion als Abrechnungskomponente. Andererseits kann er als Integrationsagent, der diese Abrechnungskomponenten repräsentiert, betrachtet werden.

Die Funktionen eines Integrationsagenten wurden in einer Oberklasse `GenericMeteringAgent` ausgegliedert. Diese kann für weitere Implementierungen für Metering-Komponenten zugeordenten Integrationsagenten benuzt werden. Die Meßfunktionalität befindet sich in der Klasse `NetworkTrafficMeterAgent`.

7.3.5 Package-Struktur

Die Gruppierung der Implementierungsklassen zu Paketen richtet sich nicht nach der Zugehörigkeit zu Komponenten, sondern stellt eine funktionale Aufteilung dar. Die Zusammenstellung zu Komponenten wird von den Vererbungs- und Assoziationsbeziehunen zwischen den Klassen bestimmt. Nachstehend werden die Pakete der Implementierung anhand von Diagrammen dargestellt, in denen auch diese Beziehungen angedeutet sind. Es handelt sich dabei um keine detaillierte Darstellung; die wichtigsten Beziehungen zwischen den Klassen wurden bereits in Kapitel 6 erläutert.

`policy` Die für die Objektrepräsentation von Policies benötigten Klassen wurden zu dem in Abbildung 7.10 Java-Package `policy` zusammengefaßt. Die Objektrepräsentation wird in allen drei Hauptkomponenten genutzt, sodaß die meisten Klassen dieses Pakets in allen Komponenten verfügbar sein müssen.

`engine` Das `engine`-Paket enthält Klassen, die den Kern der Komponenten der Managementanwendung ausmachen.

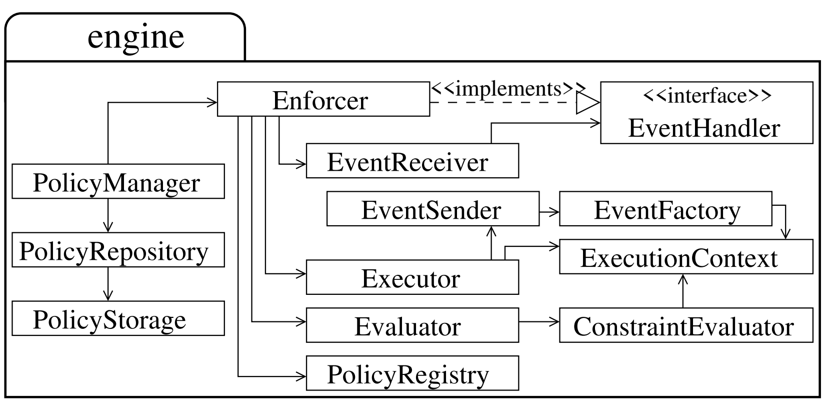

Abbildung 7.8: Das *engine*-Package

`engine.storage` Dieses Paket enthält die zur Speicherung und Verwaltung von XML-Dokumenten erstellten Klassen.

`util` Für die Managementanwendung benötigte Werkzeuge, die nicht eindeutig einem anderen Paket zugeordnet werden konnten, finden sich im Paket `util` (Abbildung 7.9). Es handelt sich dabei meist um Klassen bestehend aus `static`-Methoden und nicht um instantiierbare Typen.

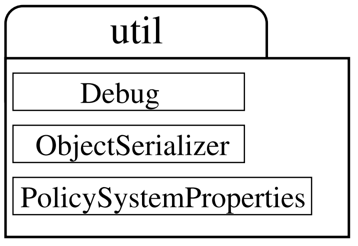

Abbildung 7.9: Das *util*-Package

`integration.meter` Der Code des als Beispiel erstellten Meters befindet sich in diesem Paket. Die Klassen des Pakets entsprechen denen des beschriebenen, experimentellen Metering-Agenten . Zusätzlich kommen die von *jidl* generierten, für die CORBA-Kommunikation benötigten Klassen hinzu.

7.4 Zusammenfassung

In diesem Kapitel wurde die prototypische Implementierung der in der vorliegenden Arbeit entwickelten Konzepte vorgestellt. Nach der Bekanntgabe der zur Implementierung benutzten Technologien und Produkte wurde die Abbildung der in Kapitel 5 spezifizierten Policy Definition Language auf eine XML-Anwendung erläutert. In diesem Rahmen wurde auch ein Überblick von XML und verwandter Technologien gegeben, sowie die Übersetzung von als XML-Dokument vorliegenden Policies in ihre Objektrepräsentation beschrieben. Anschließend wurde die Umsetzung der Managementanwendung mit Java, CORBA und MASA beschrieben, sowie experimentelle Schnittstellen und die Implementierung eines Integrationsagenten. Zuletzt wurde die Paketstruktur der Implementierung angegeben, um einen Einblick in die Organisation des Quellcodes zu gewähren.

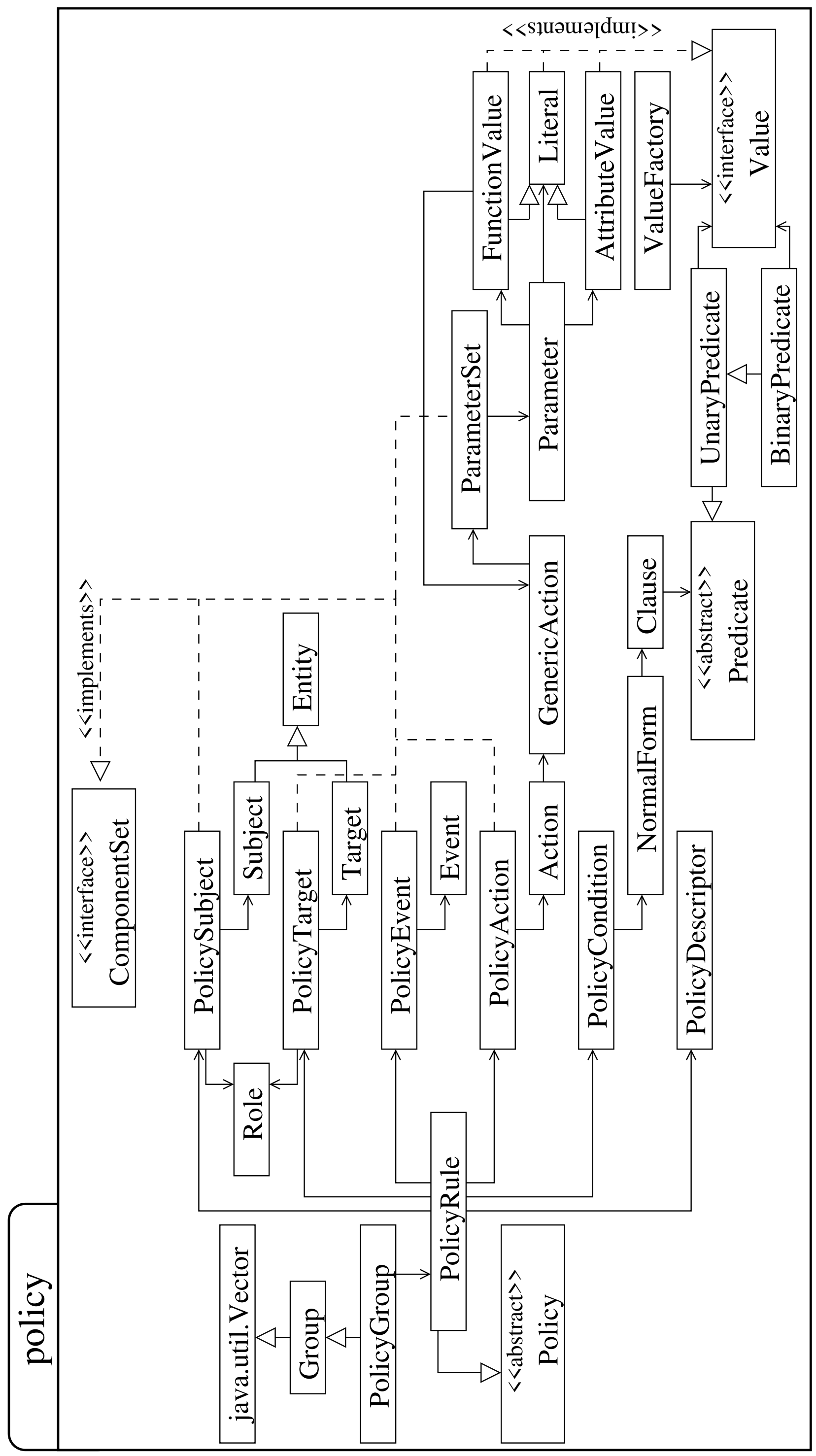

Abbildung 7.10: Das *policy*-Package

Kapitel 8

Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde ein policy-basierter Ansatz für das prozeßorientierte Management der Abrechnung vorgestellt. In ihrem Rahmen wurde eine Sprache zur Notation von Policies spezifiziert, die Konzepte der Prozeßorientierung berücksichtigt. Im Entwurf einer Managementanwendung wurden die in der Arbeit gesammelten und entwickelten Konzepte eingebracht; eine prototypische Umsetzung entspechend dieses Entwurfs belegt seine Implementierbarkeit.

Der Betrieb von Abrechnungskomponenten im allgemeinen und ihr Management im besonderen leiden unter der Heterogenität der Schnittstellen dieser Komponenten. Um ihre Interoperabilität zu ermöglichen, müssen Gateways implementiert werden, deren Wartung und Anpassung in einem Umfeld mit hoher Änderungsdynamik sich als aufwendig und kostspielig erweist. Eine Suche nach bestehenden Ansätzen und Standardisierungbemühungen ergab eine Anzahl wertvoller Arbeiten zu Teilaspekten dieser Arbeit. Es mußte jedoch festgestellt werden, daß ein konsistenter, allgemeiner Ansatz für das Abrechnungsmanagement fehlt. Außerdem zeigt sich der Umsetzunggrad der zur Verfügung stehenden Standards in Software-Produkte für die Abrechnung als unbefriedigend — Managementaspekte werden in den gängigen Implementierungen sogar völlig unberücksichtigt gelassen.

Ziel der Arbeit war es deshalb, die Konzepte der policy-basierten Steuerung mit einer prozeßorientierten Sicht auf das Abrechnungsmanagement zu vereinen, um eine flexible Managementstruktur zu schaffen, die eine einheitliche Managementsicht, konsistente Bedienung sowie eine hochgradige Anpassungsfähigkeit bezüglich Veränderungen am zu verwaltenden System aufweist. Konkrete Anforderungen an ein solches Managementsystems wurden mit Hilfe eines Szenarios erarbeitet und in einem Anforderungskatalog zusammengefaßt. Durch die Betrachtung eines praxisnahen Falls im Szenario ergaben sich Anforderungen aus den Sichten von Provider und Kunde; auch Anforderungen aus den Sichten der Endnutzer des Gesamtsystems (Administratoren des Providers und Mitarbeiter des Kunden) wurden berücksichtigt.

In diesem durch Anforderungen und Vorarbeiten gegebenen Rahmen wurden die Teilprozesse der Abrechnung analysiert und ihre Abbildung auf policy-basiertes Management untersucht. Dabei wurden Entitäten, Aktionen und Ereignisse als Charakteristika der Teilprozesse identifiziert und auf allgemeine Elemente von Sprachen zur Policy-Spezifikation abgebildet. Aufgrund der gefundenen Entsprechungen konnten Kriterien für die Entwicklung einer solchen Sprache erarbeitet werden. Die einheitliche Managementsicht auf die Abrechnungskomponenten wird durch die Einführung einer Integrationsschicht gestaltet, die aus den Abrechnungskomponenten zugeordneten Agenten besteht. Diese agieren als Proxies für die ihnen jeweils zugeordneten Komponenten, wobei die herstellerspezifische Managementschnittstelle jeder Abrechnungskomponente auf eine einheitliche abgebildet wird.

Zur Formulierung von Policies für das Abrechnungsmanagement wurde in dieser Arbeit eine deklarative, kontextfreie, wohlgeklammerte Sprache spezifiziert: die *Policy Definition Language* (PDL). Die Spezifikation stützt sich auf sorgfältig ausgewählte Vorarbeiten aus den Bereichen der Policy-Sprachen, der Informationsmodelle und der Markup-Sprachen. Anders als die im Vorfeld betrachteten Sprachen erlaubt die PDL eine Wiederverwendung von Policy-Bausteinen, indem sie sowohl *lokale* (d. h. innerhalb einer Policy angegebene) Definitionen erlaubt, als auch die Referenzierung *globaler* Bausteine, die außerhalb von Policies „frei" definiert werden können. Die durch Referenzen ermöglichte Komposition von Policies warf das Problem der referentiellen Integrität der so erstellten Konstrukte auf. Seine Lösung liegt in dem Entwurf des Laufzeitsystems und nicht in der Konzeption der Sprache, jedoch stellt die PDL Mittel zur Verfügung, um Bausteine einer Policy-Bibliothek zuordnen zu können. Auf diese Weise können Bausteine „geschützt" werden, sodaß sie nicht zusammen mit einer Policy gelöscht werden, die nicht mehr gebraucht wird. Neu aufgesetzte Managementsysteme könnten außerdem durch eine Policy-Bibliothek mit einem bewährten Satz von Policies und Bausteinen ausgestattet werden. Sowohl ganze Policies als auch Policy-Bausteine können (Teil-) Prozessen zugeordnet werden, um die Entwicklung von Verfahren und Werkzeugen zur ermöglichen, die unter Einbeziehung der Teilprozesse eines Managementbereichs die Erstellung und Modifizierung von Policies unterstützen.

Eines der Ziele Policy-basierten Managements ist die Speicherung des Managementwissens der Administratoren eines Systems in Policy-Notation. Die PDL besitzt zusätzlich Mechanismen zur Dokumentation von Policies und deren Bausteinen (z. B. Kommentare), die für manche Sprachelemente als notwendiger Teil ihrer Syntax vorgeschrieben sind. Damit wird ein Problem angesprochen, das in manchen anderen Sprachdefinitionen nur am Rande betrachtet wird. Zur automatischen Verwaltung der Policies selbst werden im allgemeinen sogenannte *Metapolicies* benutzt. Wie eine Untersuchung bestehender Sprachen ergab, werden diese oft in einer von der Policy-Sprache unterschiedlichen Notation angegeben. Mit der in dieser Arbeit spezifizierten PDL können sowohl Policies als auch Metapolicies in derselben Notation angegeben werden. Metapolicies werden dabei als operationale Policies bezüglich anderer Policies betrachtet. Dazu wurden die grundlegenden Teile einer Schnittstelle (API) entwickelt, mit Hilfe derer Aktionen von Metapolicies formuliert werden können.

Die Policy Definition Language kommt in einer Managementanwendung zum Einsatz, die neben der Integrationsschicht zur Repräsentation der gemanageten Komponenten den wichtigsten Teil des Managementsystems ausmacht. Der Entwurf dieser Anwendung wurde entsprechend den im Anforderungskatalog formulierten Kriterien entwickelt. Dabei wurde auf die Benutzung bewährter objektorientierter Entwurfsmuster Wert gelegt, um ein flexibles, gut erweiterbares Werkzeug zu schaffen. Die Funktionalität der Managementanwendung wurde deshalb unter drei Komponenten aufgeteilt, die Verwaltung von Policies und Policy-Bausteinen, Auswertung und Ausführung von Policies, beziehungsweise Koordination und Interaktion mit dem Benutzer übernehmen: Repository, Enforcer und Manager. Außerdem wurde eine objektbasierte Repräsentation für Policy-Bausteine, sowie für ganze Policies und Policy-Gruppen entworfen, wobei auf Spezifikationen bezug genommen wurde, die von Standardisierungsgremien für die objektorientierte Darstellung von Policies verfaßt wurden. Das *Policy Core Information Model* der IETF, das weitgehend mit entsprechenden Arbeiten der DMTF übereinstimmt, wurde als Ausgangspunkt für den Entwurf der Repräsentation gewählt und mittels geeigneter Erweiterungen an die Repräsentationsbedürfnisse der Policy Definition Language angepaßt.

Die drei genannten Komponenten der Managementanwendung, wie auch Beispiele für Integrationsagenten wurden in dem durch die Mobile Agent System Architecture (MASA) gebotenen Rahmen in Java implementiert. Die Integrationsagenten weisen dabei generische Schnittstellen auf, die aus der Analyse der Teilprozesse abgeleitet wurden. Die von der MASA benutzte CORBA-Umgebung wurde auch für die Interkommunikation der Teile des Managementsystems übernommen. Die Übersetzung der Policies aus der PDL-Notation in das entwickelte Objektmodell wurde in die Klassen eben dieses Modells eingebaut, sodaß durch die Konsistenz von Modell- und Übersetzungscode eine gute Erweiterbarkeit des Repräsentationsmodells für Policies gewährleistet ist. Die Grammatik der Policy Definition Language wurde als XML-Anwendung mit Hilfe von XML-Schema definiert, sodaß die Notwendigkeit der Implementierung eines proprietären Parsers für die PDL vermieden konnte, indem ein den von dem W3C-Konsortium verabschiedeten Standards entsprechender XML-Parser eingesetzt wurde.

Durch die Wahl von XML zur Notation von Policies könnten — so die Visionen im Bereich der Entwicklung von XML — Standardwerkzeuge zur Erstellung und Editierung von PDL-Konstrukten benutzt werden, beispielsweise XML-Editoren. Da solche Werkzeuge in der Praxis meist als Teil umfangreicher (und teurer) Software-Pakete vertrieben werden und gegenwärtig keine als freie Software entwickelten Editoren den von der Verarbeitung der PDL-Ausdrücke gestellten Anforderugen genügen, standen zum Zeitpunkt der Implementierung keine „Standardwerkzeuge" zur Verfügung. Die graphische Oberfläche zur Bedienung der Managementanwendung wurde deshalb um Funktionalität zur Erstellung und Modifikation von Policies erweitert.

Ein ausgereiftes Konzept für prozeßorientiertes Management setzt ein lückenloses Verfahren zur schrittweisen Verfeinerung von Managementregeln voraus, das von den Geschäftsprozessen des Betreibers ausgeht und als Endprodukt operationale Policies erzeugt. In der vorliegenden Arbeit wurde lediglich die Abbildung von Teilprozessen eines Managementbereichs auf operationale Policies betrachtet. Zudem wurden die Teilprozesse der Abrechnung nur summarisch analysiert, da Erfahrungsberichte aus der Praxis und Schnittstellenspezifikation für Komponenten von Abrechnungssystemen nicht zur Verfügung standen. Die daraus abgeleiteten Schnittstellen der Integrationsagenten mußten deshalb generisch und aufgrund von Annahmen spezifiziert werden. Eine detaillierte Analyse der Teilprozesse unter Hinzunahme von Erfahrungswerten mit Abrechnungssystemen könnte zur Spezifikation von realistischen statt nur plausiblen Schnittstellen beitragen.

Die Verfeinerung von Policies innerhalb der Policy-Hierarchie, sowie die Modellierung der Teilprozesse mit Hilfe von Petri-Netzen mußten ebenfalls unbetrachtet bleiben. Zusammengenommen eröffnen diese beiden Konzepte allerdings Möglichkeiten der graphischen Modellierung des Managements. Zielorientierte Policies könnten als Ereignisse, Bedingungen und Aktionen eines Petri-Netzes dargestellt werden. Dieses Netz könnte in Segmente organisiert werden, die den Teilprozessen entsprechen. Das Ergebnis wäre ein Graph von Petri-Netz-Segmenten, der dem Prozeßmodell entspricht. Jeder Knoten dieses Graphen bestünde selbst aus einem Graphen mit höherem Detailgrad bezüglich der beschriebenen Vorgänge. Für den Verfeinerungsvorgang könnte dann ein Werkzeug zur Visualisierung der Graphen erstellt werden, um die Modellierung von Policies verschiedener Detailstufen zu unterstützen. An den Segmentgrenzen (die den Übergängen zwischen Teilprozessen entsprächen) wären die Bestandteile der zu formulierenden operationalen Policies lokalisiert, die solche Übergänge bewirken. Die Kanten innerhalb der Segmente würden Bestandteile operationaler Policies angeben, die Zustandsänderungen innerhalb eines Teilprozesses bewirken. Allerdings sind Untersuchungen, die die Machbarkeit eines solchen Vorhabens untermauern notwendig, besonders mit Blick auf die Komplexität und die Korrektheitsprüfung der dabei entstehenden Petri-Netze.

Die vorgestellte Policy-Sprache und ihre Umsetzung unterstützen die Erstellung von Policy-

Bibliotheken. Die Entwicklung von oft benutzten, also „nützlichen”, Policy-Bausteinen kann allerdings nur als Erfahrungswert aus dem Betrieb des Managementsystems hervorgehen. Ein weiteres noch zu betrachtendes Problem stellt die Auflösung von Konflikten zwischen Policies dar. Entsprechend der in PCIM formulierten Empfehlung unterstützt die PDL Prioritäten für Policies. Die Reihenfolge der Ausführung kann bei Bedarf auf diese Weise festgelegt werden. Es fehlt jedoch weiterhin eine Methode zur Bestimmung dieser Reihenfolge.

Manche Entwurfsentscheidungen müßten ebenfalls für einen realen Einsatz des Managementsystems überprüft werden. Sind etwa die Komponenten der Managementanwendung in verschiedenen Domänen verteilt, könnte beispielsweise das zur Signalisierung von Ereignissen benutzte Push-Modell durch Sicherheitssysteme der verschiedenen Domänen (z. B. Firewalls) behindert werden. Um die Signalisierung weiterhin zu gewährleisten, ohne die Sicherheitsstandards einzelner Domänen zu lockern (durch das Öffnen von Ports in der Firewall), müßte ein hybrides Modell aus den Push- und Pull-Modellen entwickelt werden. Im Rahmen einer CORBA-basierten Implementierung kann unter Umständen der Einsatz eines *Gatekeeper* Abhilfe verschaffen.

Die in dieser Arbeit entwickelte Sprache und Managementarchitektur entspricht den Anforderungen des Abrechnungsmanagement, wurde jedoch nicht auf diesen Managementbereich spezialisiert. In der Tat wurden die meisten Konzepte der vorliegenden Arbeit in der Hoffnung entwickelt, daß sie auch auf andere Managementbereiche übertragbar seien. Ihre Eignung bezüglich der Integration mehrerer Managementbereiche muß natürlich erst geprüft werden.

Anhang A

Analyse der Teilprozesse der Abrechnung

Die folgende Übersicht der Charakteristika einzelner Teilprozesse dient als Grundlage für ihre Abbildung auf eine konkrete regelbasierte Steuerung. Dieses soll je nach Konfiguration (durch Policies) als Antwort auf ein *Trigger-Ereignis* eine oder mehrere *Aktionen* ausführen. Zu den Aktionen gehören auch die Ereignisse, die in einem Teilprozess erzeugt werden, also die *generierten Ereignisse*. Diese ermöglichen Übergänge zwischen Teilprozessen, sowie den wiederholten Eintritt in einen Teilprozess. Aufgrund der der Flexibilität, die ein solcher ereignisgesteurter Ansatz bietet, wird dabei die Gefahr von unkontrollierten Schleifen durch falsch formulierte Policies in Kauf genommen.

Anmerkung: Die hier vorgestellten Merkmale der Teilprozesse sind weder vollständig noch allgemein genug. Die Beschreibungen mancher Ereignisse oder Aktionen beziehen sich auf das in Kapitel 2 beschriebene Szenario. Es ist aber selbstverständlich, daß versucht wurde, diese Ereignisse und Aktionen allgemein zu fassen, so daß sie sich auf beliebige Anwendungen übertragen lassen.

A.1 configure accounting system

Semantik: Komponenten des Abrechnungssystems werden konfiguriert. Dabei werden an die Komponenten Daten verteilt bezüglich:

- Benutzer
- Kunden
- Dienste
- Adressaten für Rechnungen
- Tarife

Trigger-Ereignisse

- `systemStartup` - ein Abrechnungssystem wurde hochgefahren
- `systemShutdown` - ein Abrechnungssystem wird heruntergfahren

- `configurationSetChanged` - die Konfiguration des Gesamtsystems wurde verändert, z. B. durch Editieren einer Konfigurationsdatei durch den Administrator.
- `customerAdded` - ein neuer Kunde wurde hinzugefügt
- `customerProfileChanged` - das Profil eines Kunden wurde verändert, z. B. die abbonierten Dienste, die anzuwendenden Tarife oder die Rechnungsadresse
- `tariffAdded` - ein neuer Tarif wurde hinzugefügt
- `tariffChanged` - ein Tarif wurde verändert
- `accountingUnitChanged` - eine Maßeinheit wurde verändert, z. B. kann die Granularität in der Messung der Übertragungsrate von Kilobyte/Sekunde auf Byte/Sekunde verfeinert werden. Die Meter müssen entsprechend umkonfiguriert werden.
- `userRemoved` - ein Benutzer wurde gelöscht
- `newDataRollout` - die Konfigurations-Datenbank wurde aktualisiert. Dieses Ereignis sollte verhindern, daß bei großen Veränderungen in der Benutzerpopulation eine Unmenge von Ereignissen der Typen `userAdded` und `userRemoved` generiert werden. Statt dessen kann aufgrund dieses Ereignisses eine Aktualisierung der gesamten Konfiguration veranlaßt werden.

Aktionen Die Aktionen, die in diesem Teilprozeß ausgeführt werden, beziehen sich auf die Komponenten des Abrechnungssystems selbst, d. h. Metering-, Collecting/Accounting-, Charging- und Billingkomponenten. Es werden im Folgenden Aktionen spezifiziert, deren Angabe in Policies (als Reaktion auf ein Triggerereignis) plausibel ist. Die Aktionen sind dabei jeweils auf eine der Komponenten bezogen; sie werden als Methodenaufrufe des Proxyobjekts der Komponente formuliert.

- `system.configure` - ein neu hinzugefügtes oder hochgefahrenes System wird neu konfiguriert.
- `reconfigure` - die Konfiguration des Gesamtsystems wird aktualisiert
- `...`

Generierte Ereignisse aus diesem Teilprozess beziehen sich auf den Erfolg oder das Fehlschlagen eines Konfigurationsschritts

- `reconfigurationSuccess`
- `reconfigurationFailure` -
- `accountingUnitUpdated` - die neue Maßeinheit wurde erfolgreich eingestellt
- `accountingUnitUpdateFailed` - z. B. : der Meter unterstützt die neue Maßeinheit nicht
- `meterUnreachable` - auf einen Meter kann nicht zugegriffen werden

Entitäten Kunde, Dienst, Tarif, Komponenten des Abrechnungssystems

A.2 deploy meters

Semantik: Um Nutzung der angebotenen Dienste messen zu können, müssen Meteringagenten installiert werden.

Trigger-Ereignisse

- `systemAdded` - ein Abrechnungssystem wurde (neu) in das Netzwerk eingebunden
- `systemProfileChanged` - der Einsatzbereich eines Rechners wurde verändert, so daß (neue) Messungen notwendig sind
- `serviceAdded` - ein neuer Dienst wurde spezifiziert
- `userAdded` - ein neuer Benutzer wurde registriert
- `userProfileChanged` - das Profil eines Benutzers wurde verändert, z. B. bezüglich Zugriffsrechten oder Dienstgüte-Zusicherungen
- `serviceProfileChanged` - die Spezifikation eines Dienstes wurde verändert

Aktionen

- `system.installMeter` - das Zielsystem wird angewiesen, einen Meter zu installieren
- `system.testMeter` - Überprüfung der Anwesenheit und Funktionsfähigkeit des Meters

Generierte Ereignisse

- `deploySuccess` - Meter erfolgreich installiert
- `deployFailure` - Meter konnte nicht in Betrieb genommen werden

Entitäten Benutzer, Meter, (Ziel-)System

A.3 deinstall meters

Semantik: Nach der Beendung des Vertragsverhältnisses zwischen Anbieter und Kunde sollen die Vorrichtungen zur Messung sowie die dazu gehörenden Daten (Benutzerdaten, Tarife etc.) entfernt werden.

Trigger-Ereignisse

- `customerRemoved` - ein Kunde ist nicht mehr Vertragspartner
- `serviceRemoved` - ein Dienst wurde deaktiviert

Aktionen

- `meter.finalize` - letzte evtl. noch erfaßte Messdaten sollen an das Abrechnungssystem geliefert werden, danach sollen keine weiteren Messungen erfolgen
- `meter.terminate` - der Meter wird angehalten
- `system.removeMeter` - das Zeilsystem wird angewiesen, den Meter zu löschen
- `system.removeConfiguration` - das Zielsystem wird angewiesen, die Konfiguration für die Abrechnung zu löschen

Generierte Ereignisse

- `deinstallSuccess` - der Meter bzw. die Konfiguration wurde erfolgreich deinstalliert
- `deinstallFailure` - der Meter bzw. die Konfiguration konnte nicht entfernt werden
- `meterNotInstalled` - es gibt gar keinen Meter, der deinstalliert werden könnte

Entitäten Kunde, Meter, (Ziel-)System

A.4 usage accounting

Semantik: Messung der Inanspruchnahme der Dienste durch die Endbenutzer

Trigger-Ereignisse

- `usageInitiated` - der Benutzer startet eine zu messende Operation, z. B. einen Datentransfer
- `usageTerminated` - die zu messende Operation wird (ordnungsgemäß) abgeschlossen
- `usageAborted` - die zu messende Operation wird nicht ordnungsgemäß abgeschlossen, sondern aufgrund von Fehlern, Eingriff von außen (Administrator oder anderen Einflüssen außerhalb des Wirkungsbereichs des Benuters) abgebrochen.
- `meterDataReady` - ein Meter signalisiert, daß Daten zum Abruf bereit stehen

Aktionen

- `meter.recordStart` - Messung wird gestartet
- `meter.recordStop` - Messung wird angehalten
- `meter.recordRollback` - Messung wird verworfen
- `meter.getData` - abrufen der vom Meter gesammelten Daten

Generierte Ereignisse

- `usageRecordReady` - ein *Usage Record* steht zum Abruf bereit
- `meterUnreachable` - auf einen Meter kann nicht zugegriffen werden

Entitäten Benutzer, Meter

A.5 charging

Semantik: Aus den im *usage accounting*-Prozeß erfaßten Daten werden Tarife in Abhängigkeit von der Vereinbarung zwischen Anbieter und Kunde angewendet

Trigger-Ereignisse

- `usageRecordReady` - Meßdaten stehen bereit
- `chargingRequest` - Charging wird explizit initiiert
- `tariffMissing` - ein Tarif (für einen Teil der Meßdaten) steht nicht zur Verfügung

Aktionen

- `applyTariff` - auf die Meßdaten wird ein Tarif angewendet
- `applyDefaultTariff` - auf die Meßdaten wird (mangels eines passenden) der Standardtarif angewendet

Generierte Ereignisse

- `cdrReady` - ein *Customer Detailed Record* steht zum Abruf bereit
- `tariffMissing` - ein Tarif (für einen Teil der Meßdaten) steht nicht zur Verfügung
- `defaultTariffMissing` - es steht kein Standardtarif zur Verfügung

Entitäten Benutzer, Tarif, Accounting-Komponente

A.6 billing

Semantik: Aus den mit Preisen versehenen Nutzungsdaten wird für den Kunden eine Rechnung erstellt. Dabei wird die Form der Rechnung (z. B. Einzelabrechnung) berücksichtigt.

Trigger-Ereignisse

- `billingRequest` - es wurde eine Rechnung angefordert; dies kann manuell oder automatisch (in zeitlichen Abständen) geschehen
- `cdrReady` - ein *Customer Detailed Record* steht zum Abruf bereit
- `missingInvoiceProfile` - eine Form für die Rechnung steht nicht zur Verfügung

Aktionen

- `chargingRequest` - die Charging-Komponente wird angewiesen, ausstehende *Customer Detailed Records* zu bearbeiten und bereitzustellen
- `getCDR` - Abruf von bereit stehenden *Customer Detailed Records*
- `createInvoice` - Rechnung erstellen
- `createStandardInvoice` - Rechnung in Standardform erstellen

Generierte Ereignisse

- `invoiceGenerated` - eine Rechnung wurde erstellt
- `invoiceGenerationFailed` - es kann keine Rechnung erstellt werden (generisch)
- `missingInvoiceProfile` - eine Form für die Rechnung steht nicht zur Verfügung
- `missingProfileData` - für die Erstellung der Rechnung notwendige Daten sind nicht Verfügbar (z. B. Rechnungsadresse)

Entitäten Kunde, Charging-Komponente, Profil-DB(Rechnungsprofil)

A.7 reporting

Semantik: Dem Kunden sollen von Zeit zu Zeit (zusätzlich zu den Rechnungen) Berichte über die Nutzung der Dienste zugestellt werden.

Trigger-Ereignisse

- `reportRequest` - es wurde ein Bericht angefordert; dies kann manuell oder automatisch (in zeitlichen Abständen) geschehen
- `usageLimitReached` - eine vereinbarter Schwellwert der Nutzung wurde erreicht, z. B. ein bestimmtes übertragenes Datenvolumen
- `missingReportProfile` - eine Form für den Bericht steht nicht zur Verfügung

Aktionen

- `chargingRequest` - die Charging-Komponente wird angewiesen, ausstehende *Customer Detailed Records* zu bearbeiten und bereitzustellen
- `getCDR` - Abruf von bereit stehenden *Customer Detailed Records*
- `createReport` - ein Bericht wird erstellt
- `createStandardReport` - ein Bericht in Standardform wird erstellt

Generierte Ereignisse

- `reportGenerated` - ein Bericht wurde erstellt
- `reportGenerationFailed` - Bericht kann nicht erstellt werden
- `missingReportProfile` - eine Form für den Bericht ist nicht verfügbar
- `missingReportProfileData` - für die Erstellung des Berichts notwendige Daten sind nicht verfügbar (z. B. Adresse)

Entitäten Kunde, Controller (kundenseitig), Charging-Komponente, Profil-DB(Berichtsprofil)

A.8 change

Semantik: Im laufenden Betrieb werden Veränderungen an der Benutzerpopulation, den Dienstprofilen etc. vorgenommen. In dieser Fassung beschränkt sich der Prozeß auf

- das Hinzufügen/Löschen eines Benutzers
- das Ändern eines Benutzerprofils
- das Ändern eines Dienstprofils

Trigger-Ereignisse

- `userModification` - Änderung in der Benutzerpopulation
- `profileModification` - ein Profile (eines Benutzers, Dienstes etc.) soll verändert werden

Aktionen

- `addUser` - Benutzer wird hinzugefügt
- `deleteUser` - Benutzer wird gelöscht
- `changeUserProfile` - Benutzerprofil wird verändert
- `changeServiceProfile` - Dienstprofil wird verändert

Generierte Ereignisse

- `userAdded`
- `userRemoved`
- `userProfileChanged`
- `serviceProfileChanged`
- `userPresentlyActive` - Benutzer, der gelöscht werden soll nutzt zur Zeit einen Dienst
- `serviceInUse` - Dienst, der geändert werden soll, wird zur Zeit genutzt
- `cannotAddUser` - Benutzer kann nicht hinzugefügt werden
- `cannotChangeUserProfile` - ein Benutzerprofil kann nicht geändert werden

Entitäten Benutzer, Dienst, Benutzer-DB, Dienste-DB

A.9 pricing

Semantik: Die Gebühren für die Nutzung von Diensten werden festgelegt. Bei statischem Pricing, d. h. wenn der Preis unabhänging von der Auslastung der Ressourcen bestimmt wird, *pricing* ein rein anwendergesteuerter Prozeß; der Preis wird also in Verhandlungen zwischen Anbieter und Kunde ermittelt. Bei dynamischem Pricing variiert der Preis mit der aktuellen Auslastung der Ressourcen (verfügbare Übertragungsrate, Rechenleistung etc.); die im Folgenden beschriebenen Merkmale des Prozesses beziehen sich auf eben diesen Fall. Dabei wird angenommen, daß die verfügbaren Kapazitäten überwacht werden und beim Erreichen vordefinierter Schwellwerte entsprechende Ereignisse generiert werden.

Trigger-Ereignisse

- `usageBoundReached`

Aktionen

- `changeTariff` - der Tarif wird verändert
- `chargingRequest` - charging wird angefordert (für die bereits erfolgte Nutzung)

Generierte Ereignisse

- `tariffChanged` - ein Tarif wurde verändert

Entitäten Benutzer, Kunde, Tarif-DB, Systemkomponenten, Charging-Komponente

Anhang B

Sprachdefinition in EBNF

⟨**identifier**⟩ =	⟨alphachar⟩ { ⟨alphachar⟩ | ⟨digit⟩ }
⟨**digit**⟩ =	”*0*” | ”*1*” | ... | ”*9*”
⟨**alphachar**⟩ =	”_” | ”*a*” | ”*b*” | ... | ”*z*” | ”*A*” | ... | ”*Z*”
⟨**ref**⟩ =	⟨xsd:IDREF⟩
⟨**refs**⟩ =	⟨xsd:IDREFS⟩
⟨**comment**⟩ =	”*comment*{” ⟨xsd:string⟩ ”}”
⟨**parameterSet**⟩ =	⟨namedValue⟩ { ”, ” ⟨namedValue⟩ }
⟨**namedValue**⟩ =	”*namedValue*{” ”*name*=” ⟨identifier⟩ ⟨value⟩ ”}”
⟨**entityContainer**⟩ =	([⟨domain⟩] ⟨entity⟩) | ⟨role⟩
⟨**domain**⟩ =	[”/”] { ⟨identifier⟩ ”/” }
⟨**entity**⟩ =	⟨identifier⟩
⟨**policySet**⟩ =	”*policySet*{” { ⟨policy⟩ | ⟨roleDefinition⟩ | ⟨target⟩ | ⟨event⟩ | ⟨action⟩ | ⟨policyGroup⟩ } ”}”
⟨**roleDefinition**⟩ =	”*roleDef*{” ⟨referable⟩ ⟨roleName⟩ ⟨roleDescription⟩ ”}”
⟨**roleName**⟩ =	”*name*{” ⟨xsd:token⟩ ”}”

⟨**roleDescription**⟩ = ”*description*{” ⟨xsd:string⟩ ”}”

⟨**role**⟩ = ”*role*{” ⟨refs⟩ ”}”

⟨**policyGroup**⟩ = ”*policyGroup*{” ⟨group⟩ ⟨comment⟩ ”}”

⟨**group**⟩ = ”*group*{” ⟨refs⟩ ”}”

⟨**referable**⟩ = ”*id*=” ⟨xsd:ID⟩ [⟨libraryItem⟩] [⟨comment⟩]

⟨**libraryItem**⟩ = ”*libraryItem*=” ⟨xsd:boolean⟩

⟨**processAssignable**⟩ = ⟨referable⟩ [⟨relatedProcess⟩]

⟨**relatedProcess**⟩ = ”*relatedProcesses*{” ⟨process⟩ { ”,” ⟨process⟩ } ”}”

⟨**process**⟩ = ⟨identifier⟩ | ”*OTHER*” | ”*ALL*”

⟨**policy**⟩ = ”*policy*{” ⟨processAssignable⟩ [”*domain*{” ⟨domain⟩ ”}”] [
⟨subjectSet⟩] [⟨targetSet⟩] ⟨eventSet⟩ [⟨constraintSet⟩]
⟨actionSet⟩
[⟨descriptor⟩] ⟨attribs⟩ ”}”

⟨**descriptor**⟩ = ”*descriptor*{” [⟨createdBy⟩] [⟨creationDate⟩] [
⟨lastModified⟩] [⟨modifiedBy⟩] [⟨expires⟩] ”}”

⟨**attribs**⟩ = ”*attributes*{” ⟨enabled⟩ [⟨priority⟩] [⟨version⟩] ”}”

⟨**createdBy**⟩ = ”*createdBy*=” ⟨xsd:token⟩

⟨**creationDate**⟩ = ”*dateOfCreation*=” ⟨xsd:date⟩

⟨**modifiedBy**⟩ = ”*modifiedBy*=” ⟨xsd:token⟩

⟨**lastModified**⟩ = ”*lastModified*=” ⟨xsd:dateTime⟩

⟨**enabled**⟩ = ”*isEnabled*=” ⟨xsd:boolean⟩

⟨**priority**⟩ = ”*priority*=” ⟨digit⟩ [⟨digit⟩]

⟨**version**⟩ = ”*version*=” ⟨digit⟩ ”.” ⟨digit⟩

⟨**expires**⟩ = ”*expires*=” ⟨xsd:dateTime⟩

⟨**targetSet**⟩ = ”*targetSet*{” (⟨target⟩ | ⟨targetRef⟩ | ⟨targetGroup⟩) ”}”

⟨**actionSet**⟩ = ”*actionSet*{”
(⟨action⟩ | ⟨actionRef⟩ | ⟨actionGroup⟩) ”} ”

⟨**eventSet**⟩ = ”*eventSet*{”
(⟨event⟩ | ⟨eventRef⟩ | ⟨eventGroup⟩) ”}”

⟨**subjectSet**⟩ = ”*subjectSet*{”
(⟨subject⟩ | ⟨subjectRef⟩ | ⟨subjectGroup⟩) ”}”

⟨**constraintSet**⟩ = ”*constraintSet*{” (⟨cnf⟩ | ⟨dnf⟩ | ⟨constraint⟩ |
⟨constraintRef⟩) ”}”

⟨**event**⟩ = ”*event*{” [⟨referable⟩] ”*name*=” ⟨identifier⟩ ”}”

⟨**eventRef**⟩ = ⟨ref⟩

⟨**eventGroup**⟩ = ⟨group⟩

⟨**action**⟩ = ”*action*{” ⟨processAssignable⟩ ⟨defaultAction⟩ [
⟨errorAction⟩] ”}”

⟨**actionRef**⟩ = ⟨ref⟩

⟨**actionGroup**⟩ = ⟨group⟩

⟨**defaultAction**⟩ = ”*default*{” ⟨genericAction⟩ ”}”

⟨**errorAction**⟩ = ”*onError*{” ⟨genericAction⟩ ”}”

⟨**genericAction**⟩ = ⟨methodCall⟩ | ⟨generateEvent⟩

⟨**methodCall**⟩ = ”*invoke*{” [⟨objectName⟩ ”.”] ⟨method⟩ ”}”

⟨**generateEvent**⟩ = ”*generateEvent*{” ⟨eventName⟩ ”(” ⟨parameterSet⟩ ”)}”

⟨**method**⟩ = ⟨identifier⟩ ”(” ⟨parameterSet⟩ ”)”

⟨**eventName**⟩ = ⟨identifier⟩

⟨**objectName**⟩ = ⟨identifier⟩

⟨**cnf**⟩ = ”*and*{” ⟨binaryLogOpOR⟩ { ”, ” ⟨binaryLogOpOR⟩ } ”}”

⟨**dnf**⟩ = ”*or*{” ⟨binaryLogOpAND⟩ { ”, ” ⟨binaryLogOpAND⟩ } ”}”

⟨**constraint**⟩ = ((”*equal*” | ”*smaller*” | ”*greater*” | ”*greaterEqual*” |
”*smallerEqual*”) ”{” ⟨binaryPredicate⟩ ”}”) |
([”*!*”] ⟨value⟩)

⟨**constraintRef**⟩ =	⟨ref⟩
⟨**binaryLogOpOR**⟩ =	⟨constraint⟩ { "‖" ⟨constraint⟩ }
⟨**binaryLogOpAND**⟩ =	⟨constraint⟩ { "&&" ⟨constraint⟩ }
⟨**binaryPredicate**⟩ =	⟨value⟩ ", " ⟨value⟩
⟨**subject**⟩ =	"*subject*{" [⟨referable⟩] ⟨entityContainer⟩ "}"
⟨**target**⟩ =	"*target*{" [⟨referable⟩] ⟨entityContainer⟩ "}"
⟨**subjectRef**⟩ =	⟨ref⟩
⟨**subjectGroup**⟩ =	⟨group⟩
⟨**targetRef**⟩ =	⟨ref⟩
⟨**targetGroup**⟩ =	⟨group⟩
⟨**value**⟩ =	⟨type⟩ "{" ⟨valueContent⟩ "}"
⟨**type**⟩ =	"*float*" | "*integer*" | "*string*" | "*boolean*" | "*dateTime*"
⟨**valueContent**⟩ =	⟨literal⟩ | ⟨array⟩ | ⟨functionValue⟩ | ⟨attributeValue⟩
⟨**literal**⟩ =	⟨xsd:float⟩ | ⟨xsd:integer⟩ | ⟨xsd:string⟩ | ⟨xsd:boolean⟩ | ⟨xsd:dateTime⟩
⟨**array**⟩ =	"*array*{" ⟨length⟩ ((⟨literal⟩ { ", " ⟨literal⟩ }) | "*null*") "}"
⟨**functionValue**⟩ =	⟨methodCall⟩
⟨**attributeValue**⟩ =	"*attributeValue*{" ⟨objectName⟩ "." ⟨attributeName⟩ "}"
⟨**attributeName**⟩ =	⟨identifier⟩
⟨**length**⟩ =	"*length*=" ⟨xsd:integer⟩

Anhang C

Sprachdefinition in XML

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        targetNamespace="pdl"
        xmlns:pdl="http://www.nm.informatik.uni-muenchen.de/pdl">
<xsd:annotation>
 <xsd:documentation xml:lang="de">
        Sprache zur Definition von Policies.
        Vitalian A. Danciu, 2002
        Version 0.7
 </xsd:documentation>
</xsd:annotation>


<!-- ROOT ELEMENTS AND THEIR TYPES -->
<xsd:element name="policySet" type="policySetType"/>
<xsd:element name="comment" type="xsd:string"/>

<!-- The elements that MAY be defined standalone in document instances -->
<xsd:complexType name="policySetType">
 <xsd:sequence>
  <xsd:element name="policy" type="policyType"
                       minOccurs="1" maxOccurs="unbounded"/>
  <xsd:element name="roleDefinition" type="roleDefinitionType"
                       minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="subject" type="entityContainer"
                       minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="target" type="entityContainer"
                       minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="event" type="eventType"
                       minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="action" type="actionType"
                       minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="constraint" type="constraintType"
                       minOccurs="0" maxOccurs="unbounded"/>
```

```
  <xsd:element name="policyGroup" type="group"
                        minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element ref="comment" minOccurs="0"/>
        </xsd:sequence>
</xsd:complexType>

<!-- END ROOT ELEMENTS AND THEIR TYPES -->


<!-- BASE TYPES -->

<!-- Abstract base type for all nodes that are referable by ID.
                These MAY be library items. -->
<xsd:complexType name="referable" abstract="true">
 <xsd:sequence>
        <xsd:element name="id" type="xsd:ID"/>
        <xsd:element ref="comment" />
        <xsd:attribute name="libraryItem" type="xsd:boolean"
                 use="optional" default="false"/>
 </xsd:sequence>
</xsd:complexType>

<!-- Referable items can be assigned to one or more processes.
A list of valid process names is embedded into the type definition. -->
<xsd:complexType name="processAssignable" abstract="true">
 <xsd:complexContent>
  <xsd:extension base="pdl:referable">
   <xsd:sequence>
        <xsd:element name="relatedProcess">
                <xsd:simpleType>
                 <xsd:union>

                 <!-- A list of processes that apply.
                Process names MUST NOT contain whitespace. -->
                  <xsd:simpleType>
                        <xsd:list itemType="xsd:String">
                                <xsd:restriction base="xsd:string">
                                        <xsd:enumeration value="accounting"/>
                                        <xsd:enumeration value="charging"/>
                                        <xsd:enumeration value="billing"/>
                                        <xsd:enumeration value="change"/>
                                        <xsd:enumeration value="deployMeters"/>
                                        <!-- TODO: complete list -->
                                </xsd:restriction>
                        </xsd:list>
                  </xsd:simpleType>

                  <!-- MEANING: this item is related to all processes -->
                  <xsd:simpleType>
                        <xsd:restriction base="xsd:string">
                                <xsd:enumeration value="ALL"/>
                        </xsd:restriction>
                  </xsd:simpleType>
```

```
                   <xsd:simpleType>
                        <xsd:restriction base="xsd:string">
                                <xsd:enumeration value="OTHER"/>
                        </xsd:restriction>
                  </xsd:simpleType>
                 </xsd:union>
                </xsd:simpleType>
        </xsd:element>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>




<xsd:complexType name="abstractValueType" abstract="true">
        <xsd:attribute name="type">
                <xsd:simpleType>
                        <xsd:restriction base="xsd:token">
                                <xsd:enumeration value="float"/>
                                <xsd:enumeration value="integer"/>
                                <xsd:enumeration value="string"/>
                                <xsd:enumeration value="boolean"/>
                                <xsd:enumeration value="dateTime"/>
                        </xsd:restriction>
                </xsd:simpleType>
                </xsd:attribute>
</xsd:complexType>

<!-- END BASE TYPES -->


<!-- STANDALONE COMPONENT TYPES -->

<!-- A group is a list of references to items of the same type.
Groups themselves CANNOT be referenced.

There is one generic group defined for grouping all
sorts of elements like policies, targets, events etc.
Beware: this implies that the consistency of the group
content CANNOT be assured by the parser, i. e. a targetSet
containig a group consisting of actions and events
DOES NOT violate this schema !!
Such anomalies MUST be checked by the interpreter. -->
<xsd:complexType name="group">
   <xsd:sequence>
        <xsd:element name="items" type="xsd:IDREFS"/>
        <xsd:attribute name="itemsType" use="required">
                <xsd:simpleType>
                        <xsd:restriction base="xsd:string">
                                <xsd:enumeration value="target"/>
                                <xsd:enumeration value="event"/>
```

```
                                <xsd:enumeration value="action"/>
                                <xsd:enumeration value="policy"/>
                        </xsd:restriction>
                </xsd:simpleType>
        </xsd:attribute>
   </xsd:sequence>
</xsd:complexType>


<!-- This complexType is defined for the SUBJECT and
TARGET elements that have many similarities -->
<xsd:complexType name="entityContainer">
        <xsd:sequence>
                <xsd:choice>
                        <xsd:sequence>
                                <xsd:element name "domain" type="domainType"/>
                                <xsd:element name="entity" type="xsd:string"/>
                        </xsd:sequence>
                        <xsd:element name="role" type="xsd:IDREF"/>
                </xsd:choice>
        </xsd:sequence>
</xsd:complexType>


<!-- The domain type is just a string containing an absolute
or relative path e.g. "/a/bxx/cx" or "axxx/bx/cxx".
It is encapsulated in a named type of its own to allow for extensibility. -->
<xsd:simpleType name="domainType">
        <xsd:restriction base="xsd:token">
                <xsd:pattern value="/?([0-9a-zA-Z]/){1,}"/>
<!--
In EBNF:
pattern ::= ['/']{('0'|'1'|..|'9'|'a'|'b'|..|'z'|'A'|'B'|..|'Z')'/'}
-->
        </xsd:restriction>
</xsd:simpleType>

<!-- Roles are to be defined separately using this type.
When actually used, an existing role MUST be referenced
using an element of the type roleType -->
<xsd:complexType name="roleDefinitionType">
<xsd:complexContent>
  <xsd:extension base="pdl:referable">
        <xsd:sequence>
                <xsd:element name="name" type="xsd:token"/>
                <xsd:element name="description" type="xsd:string"/>
        </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<!-- When assigning a role to an item, the role MUST already
be defined so it can be assigned by using its ID-reference.
```

```
One or more roles can be assigned to the same item. -->
<xsd:simpleType name="roleType">
        <xsd:restriction base="xsd:IDREFS"/>
</xsd:simpleType>

<!-- Targets can be single ones or reference to a group of targets. -->
<xsd:complexType name="targetSetType">
        <xsd:sequence>
                <xsd:choice>
                        <xsd:element name="target" type="entityContainer"/>
                        <xsd:element name="targetRef" type="xsd:IDREF"/>
                        <xsd:element name="targetGroup" type="group" />
                </xsd:choice>
        </xsd:sequence>
</xsd:complexType>

<!-- Events can be single ones or a reference to a group of events -->
<xsd:complexType name="eventSetType">
        <xsd:sequence>
        <xsd:choice>
                <xsd:element name="event" type="eventType"/>
                <xsd:element name="eventRef" type="xsd:IDREF"/>
                <xsd:element name="eventGroup" type="group"/>
        </xsd:choice>
        </xsd:sequence>
</xsd:complexType>

<!-- A single event with a name -->
<xsd:complexType name="eventType">
<xsd:complexContent>
  <xsd:extension base="pdl:processAssignable">
        <xsd:sequence>
                <xsd:element name="eventName" type="xsd:Name"/>
        </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<!-- An action set contains one action or a group of actions. -->
<xsd:complexType name="actionSetType">
        <xsd:sequence>
         <xsd:choice>
                <xsd:element name="action" type="actionType"/>
                <xsd:element name="actionRef" type="xsd:IDREF"/>
                <xsd:element name="actionGroup" type="group"/>
         </xsd:choice>
        </xsd:sequence>
</xsd:complexType>

<!-- An action consists of a generic action to be executed
and an optional error action to be executed if the genericAction fails.
Alternatively, an event can be generated and propagated -->
<xsd:complexType name="actionType">
```

```
<xsd:complexContent>
  <xsd:extension base="pdl:processAssignable">
        <xsd:sequence>
         <xsd:choice>
                 <xsd:sequence>
                        <xsd:element name="default" type="genericActionType"/>
                        <xsd:element name="error" type="genericActionType"
                                minOccurs="0"/>
                 </xsd:sequence>
         </xsd:choice>
        </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>


<!-- END STANDALONE COMPONENT TYPES -->

<!-- BUILDING BLOCKS FOR STANDALONE COMPONENTS -->


<xsd:complexType name="binaryLogicalOperator">
        <xsd:sequence>
                <xsd:element name="constraint" type="constraintType"
                        maxOccurs="unbounded"/>
        </xsd:sequence>
</xsd:complexType>

<!-- A constraint set contains either a single condition or
an expression in either conjunctive or disjuctive normal form.
-->
<xsd:complexType name="constraintSetType">
  <xsd:sequence>
   <xsd:choice>

        <xsd:element name="constraintCNF">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element name="or" type="binaryLogicalOperator"
                                        maxOccurs="unbounded"/>
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>

        <xsd:element name="constraintDNF">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element name="and" type="binaryLogicalOperator"
                                        maxOccurs="unbounded"/>
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>
```

```
        <xsd:element name="constraint" type="constraintType" />
        <xsd:element name="constraintRef" type="xsd:IDREF"/>
   </xsd:choice>
  </xsd:sequence>
</xsd:complexType>

<!-- A constraint is an expression that can be a unary or binary predicate. -->
<xsd:complexType name="constraintType">
        <xsd:choice>
                <xsd:element name="equal" type="binaryPredicate"/>
                <xsd:element name="smaller" type="binaryPredicate"/>
                <xsd:element name="greater" type="binaryPredicate"/>
                <xsd:element name="greaterEqual"  type="binaryPredicate"/>
                <xsd:element name="smallerEqual" type="binaryPredicate"/>
                <xsd:element name="predicate" type="valueType"/>
                <xsd:element name="negatedPredicate" type="valueType"/>
        </xsd:choice>
</xsd:complexType>

<xsd:complexType name="binaryPredicate">
        <xsd:sequence>
                <xsd:element name="value" type="valueType"
                        minOccurs="2" maxOccurs="2"/>
        </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="genericActionType">
        <xsd:sequence>
         <xsd:choice>
                <xsd:element name "invoke" type="invocation"/>
                <xsd:element name="generateEvent">
                        <xsd:complexType>
                                <xsd:sequence>
                                        <xsd:element name="eventName" type="xsd:string"/>
                                        <xsd:element name="parameterSet" type="parameterSetType"
                                                        minOccurs="0"/>
                                </xsd:sequence>
                        </xsd:complexType>
                </xsd:element>
          </xsd:sequence>
         </xsd:choice>
        </xsd:sequence>
</xsd:complexType>


<!-- A method invocation consists of an optional object,
a method name and an optional parameterSet.
The object can either be a Name or a <subject/> element;
if it is omitted, the targets of the policy are used as object. -->
<xsd:complexType name="invocation">
        <xsd:sequence>
                <xsd:choice>
                        <xsd:element name="object" type="xsd:Name" minOccurs="0"/>
```

```
                        <xsd:element name="object">
                                <xsd:complexType>
                                        <element name="subject" empty=true minOccurs="0" />
                                </xsd:complexType>
                        </xsd:element>
                <xsd:element name="method" type="xsd:Name"/>
                <xsd:element name="parameterSet" type="parameterSetType" minOccurs="0"/>
        </xsd:sequence>
</xsd:complexType>

<!-- A parameter set is a set of name-value pairs. -->
<xsd:complexType name ="parameterSetType">
 <xsd:element name ="parameter">
  <xsd:complexType>
        <xsd:element name="paramName" type = "xsd:token"/>
        <xsd:element name="value" type="valueType"/>
  </xsd:complexType>
 </xsd:element>
</xsd:complexType>

<!-- A value is one of float, int, string, boolean or ISO dateTime.
It can be either
- a constant/literal, or
- a value retrieved from an attribute of an object, or
- a return value of a method invocation
Arrays of constants are also values.
-->
<xsd:complexType name="valueType">
 <xsd:complexContent>
  <xsd:extension base="abstractValueType">
        <xsd:choice>
                <xsd:element name="literal">
                 <xsd:simpleType>
                  <xsd:union memberTypes="xsd:float␣xsd:integer␣xsd:string␣xsd:boolean␣xsd:dateTime"/>
                 </xsd:simpleType>
                </xsd:element>
                <xsd:element name="array" type="arrayType"/>
                <xsd:element name="functionValue" type="genericActionType"/>

                <xsd:element name="attributeValue">
                        <xsd:complexType>
                                <xsd:sequence>
                                        <xsd:element name="object" type="xsd:Name"/>
                                        <xsd:element name="attribute" type="xsd:Name"/>
                                </xsd:sequence>
                        </xsd:complexType>
                </xsd:element>
        </xsd:choice>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<!-- An array is a list of items of the same type.
```

```
A string array is a whitespace delimited list of strings. -->
<xsd:complexType name="arrayType">
 <xsd:sequence>
  <xsd:element name="arrayContent">
        <xsd:simpleType>
                <xsd:union>
                 <xsd:simpleType><xsd:list itemType="xsd:float"/></xsd:simpleType>
                 <xsd:simpleType><xsd:list itemType="xsd:integer"/></xsd:simpleType>
                 <xsd:simpleType><xsd:list itemType="xsd:string"/></xsd:simpleType>
                 <xsd:simpleType><xsd:list itemType="xsd:boolean"/></xsd:simpleType>
                 <xsd:simpleType><xsd:list itemType="xsd:dateTime"/></xsd:simpleType>
                 <xsd:simpleType>
                        <xsd:restriction base="xsd:string">
                                <xsd:enumeration value="null"/>
                        </xsd:restriction>
                 </xsd:simpleType>
                </xsd:union>
                </xsd:simpleType>
         </xsd:element>
        </xsd:sequence>

</xsd:complexType>

<!-- END BUILDING BLOCKS FOR STANDALONE COMPONENTS -->



<!-- POLICY -->


<xsd:complexType name="policyType">
 <xsd:complexContent>
  <xsd:extension base="pdl:processAssignable">
        <xsd:sequence>
         <xsd:element name="policyDomain" type="domainType" minOccurs="0"/>
         <xsd:element name="subject" type="entityContainer" minOccurs="0"/>
         <xsd:element name="targetSet" type="targetSetType"/>
         <xsd:element name="eventSet" type="eventSetType"/>
         <xsd:element name="constraintSet" type="constraintSetType" minOccurs="0"/>
         <xsd:element name="actionSet" type="actionSetType"/>
         <xsd:element name="policyDescriptor">
          <xsd:complexType>
           <xsd:sequence>
                <xsd:element name="createdBy" type="xsd:token" minOccurs="0"/>
                <xsd:element name="dateOfCreation" type="xsd:date"      minOccurs="0"/>
                <xsd:element name="lastModified" type="xsd:dateTime" minOccurs="0"/>
                <xsd:element name="lastModifiedBy" type="xsd:token" minOccurs="0"/>
                <xsd:element name="expires" type="xsd:dateTime" minOccurs="0"/>
           </xsd:sequence>
          </xsd:complexType>
         </xsd:element>
        </xsd:sequence>

```

```
<xsd:attribute name="isEnabled" type="xsd:boolean"
                use="optional" default="true"/>
 <xsd:attribute name="priority" use="optional" default="0">
                <xsd:simpleType>
                        <xsd:restriction base="xsd:integer">
                                <xsd:minInclusive value="0"/>
                                <xsd:maxInclusive value="99"/>
                        </xsd:restriction>
                </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="version" type="xsd:float"
                use="optional" default="1.0"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

</xsd:schema>
```

Abkürzungsverzeichnis

API Application Programming Interface

ASP Application Service Provider

BOA Basic Object Adapter

CDR Customer Detailed Record

CIM Common Information Model

CNF Conjuctive Normal Form

CORBA Common Object Request Broker Architecture

DAF Distributed Accounting Facility

DII Dynamic Invocation Interface

DMTF Distributed Management Task Force

DNF Disjunctive Normal Form

DOM Document Object Model

DTD Document Type Definition

EBNF Extended Backus-Naur-Form

GIOP General Inter-ORB Protocol

GUI Graphical User Interface

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IDL Interface Definition Language

IESA Intranet Extranet Service Area

IETF Internet Engineering Task Force

IIOP Internet Inter-ORB Protocol

IOR Interoperable Object Reference

IPDR Internet Protocol Data Record

ISA Intranet Service Area

ISDN Integrated Services Digital Network

ISO International Standardization Organization

ISP Internet Service Provider

IT Information Technology

ITIL IT Infrastructure Library

ITU Internation Telecommunication Union

LDAP Lightweight Directory Access Protocol

MASA Mobile Agent System Architecture

MO Management Object

OMG Object Management Group

ORB Object Request Broker

OSI Open System Interconnection

PCIM Policy Core Information Model

PDL Policy Definition Language (auch: Policy Description Language)

POA Portable Object Adapter

POD Point of Decision

POE Point of Enforcement

QoS Quality of Service

RMI Remote Method Invocation

SAX Simple API for XML

SDR Session Detailed Record

SGML Standard Generalized Markup Language

SLA Service Level Agreement

SNMP Simple Network Management Protocol

SR Subscriber Record

TINA Telecommunications Information Networking Architecture

TINA-C Telecommunications Information Networking Architecture Consortium

TMN Telecommunications Management Network

TOM Telecom Operations Map

UDR Usage Detailed Record

W3C WWWW Consortium

WWW World Wide Web

XHTML Extensible Hypertext Markup Language

XML Extensible Markup Language

Literaturverzeichnis

[AAH 00] ABOBA, B., J. ARKKO und D. HARRINGTON: *RFC 2975: Introduction to Accounting Management*. RFC, IETF, Oktober 2000, ftp://ftp.isi.edu/in-notes/rfc2975.txt .

[Avit 98] AVITABILE, M.: *An Examination of Requirements for Meta–Policies in Policy–Based Management*. Diplomarbeit, Technische Universität München, November 1998, http://wwwmnmteam.informatik.uni-muenchen.de/common/Literatur/MNMPub/Diplomarbeiten/avit98/avit98.shtml .

[BBC+ 98] BLAKE, S., D. BLACK, M. CARLSON, E. DAVIES, Z. WANG und W. WEISS: *RFC 2475: An Architecture for Differentiated Service*. RFC, IETF, Dezember 1998, ftp://ftp.isi.edu/in-notes/rfc2475.txt .

[BLFM 98] BERNERS-LEE, T., R. FIELDING und L. MASINTER: *RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax*. RFC, IETF, August 1998, ftp://ftp.isi.edu/in-notes/rfc2396.txt .

[Bosa 97] BOSAK, J.: *XML, Java, and the future of the Web*. White Paper, Sun Microsystems, 1997, http://sunsite.unc.edu/pub/sun-info/standards/xml/why/xmlapps.ps.zip .

[Brow 02] BROWNELL, D.: *SAX2*. O'Reilly, Januar 2002.

[Cap 93] CAP, C. H.: *Theoretische Grundlagen der Informatik*. Springer, Wien, New York, 1993.

[CFSD 90] CASE, J.D., M. FEDOR, M.L. SCHOFFSTALL und C. DAVIN: *RFC 1157: Simple Network Management Protocol (SNMP)*. RFC, IETF, Mai 1990, ftp://ftp.isi.edu/in-notes/rfc1157.txt .

[CLN 00] CHOMICKI, J., J. LOBO und S. NAQVI: *A Logic Programming Approach to Conflict Resolution in Policy Management*. Technischer Bericht, Bell Labs, 2000.

[CORBA 2.2] *The Common Object Request Broker: Architecture and Specification*. OMG Specification Revision 2.2, Object Management Group, Februar 1998.

[Core 24] DMTF SYSTEM AND DEVICES WORKING GROUP: *Common Information Model (CIM) Core Model*. White Paper, Distributed Management Task Force, August 2000, http://www.dmtf.org/spec/Whitepapers/DSP111.pdf .

[DAF 02] *Distributed Accounting Facility (DAF), Version 3.0*. TC Document telecom/2002-03-01, Object Management Group, März 2002, http://cgi.omg.org/cgi-bin/doc?telecom/02-03-01 .

[Dami 02] DAMIANOU, N. C.: *A Policy Framework for Management of Distributed Systems*. Doktorarbeit, 2002.

[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*. Technischer Bericht, Imperial College of Science, Technology and Medicine. Department of Computing, 2000.

[DLSD01] 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): *Integrated Network Management VII (IM 2001)*, Seattle, Washington, USA, Mai 2001. IEEE Publishing.

[DOM-AS] CHANG, B., E. LITANI, J. KESSELMAN und R. RAHMAN: *Abstract Schmeas Object Model*. Technischer Bericht, W3C, 2002.

[DOM98] APARAO, V. ET AL: *Level 1 Document Object Model Specification*. W3C Recommendation, W3C, 1998, http://www.w3.org/TR/1998/WD-DOM-19980720 .

[DSP 0108] DMTF SERVICE LEVEL AGREEMENT WORKING GROUP: *CIM Core Policy Model White Paper*. White Paper, Distributed Management Task Force, März 2002, http://www.dmtf.org/standards/documents/CIM/DSP0108.pdf .

[EMS 00] EIRUNG, H., B. MÜLLER und G. SCHREIBER: *Formale Beschreibungsverfahren der Informatik*. B. G. Teubner. Stuttgart, Leipzig, Wiesbaden., 2000.

[FORM99] LEWIS, D. (ED.): *A Logical Architeture for an Open Development Framework for Component-based Management Systems*. White Paper, FORM Consortium, 2001, http://kdeg.cs.tcd.ie/form/results/whitepapers/ps/w1-v1-0.ps .

[GHJV 95] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES: *Design Pattern, Elements of Reusable Object–Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[GHK+ 01a] GARSCHHAMMER, M., R. HAUCK, B. KEMPTER, I. RADISIC, H. ROELLE und H. SCHMIDT: *The MNM Service Model — Refined Views on Generic Service Management*. Journal of Communications and Networks, 3(4):297–306, August 2001.

[GHK+ 01b] GARSCHHAMMER, M., R. HAUCK, B. KEMPTER, I. RADISIC, H. ROELLE und H. SCHMIDT: *The MNM Service Model — Refined Views on Generic Service Management*. Journal of Communications and Networks, 3(4):297–306, Dezember 2001, http://www.nm.informatik.uni-muenchen.de/Literatur/MNMPub/Publikationen/ghkr01/ghkr01.shtml .

[GHR 99] GRUSCHKE, B., S. HEILBRONNER und H. REISER: *Mobile Agent System Architecture — Eine Plattform für flexibles IT–Management*. Technischer Bericht 9902, Ludwig-Maximilians-Universität München, Institut für Informatik, München, August 1999.

[HaCa 99] HARTANTO, F. und G. CARLE: *Policy-based Billing Architecture for Differentiated Services*. In: *Proceedings of IFIP Fifth International Conference on Broadband Communications (BC'99)*, September 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, 1999. 651 p.

[HePa 96] HENNESSY, J. L. und PATTERSON D. A.: *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2 Auflage, 1996.

[Heyd 90] HEYDON, A., M. W. MAIMONE, J. D. TYGAR, J. M. WING und A. M. ZAREMSKI: *Miro´: Visual specifications of security*. In: *IEEE Transactions on Software Engineering 16*, 1990.

[HLK+ 98] HAUX, R., A. LAGERMANN, P. KNAUP und P. SCHMÜCKER: *Management von Informationssystemen*. B. G. Teubner, Stuttgart, 1998.

[Hosm 92] HOSMER, H.: *Metapolicies II*. In: *Proceeding of the 15th National Computer Security Conference, Baltimore*, 1992.

[IPDR 26] COTTON, S. und OTHERS: *Network Data Management – Usage (NDM–U) For IP-Based Services*. Version 2.6, IPDR, Inc., Juni 2001.

[ISO 8879] *Information processing — Text and Office Systems — Standard Generalized Markup Language (SGML)*. Technischer Bericht 8879:1986(E), ISO (International Organization for Standardization), 1986.

[ITU-T X.742] *Information Technology — Open Systems Interconnection — Systems Management: Usage Metering Function for Accounting Purposes*. Recommendation X.742, ITU-T, April 1995.

[Koch 96] KOCH, T.: *Automated Management of Distributed Systems*. Doktorarbeit, 1996.

[Krel 97] KRELL, C.: *Policy-basiertes Management von verteilten System*. Diplomarbeit, 1995.

[KRRV01] KEMPTER, B., H. REISER, H. ROELLE und G. VOGT: *Implementierung eines MASIF konformen Agentensystems — Die Mobile Agent System Architecture (MASA) —* . PIK – Praxis der Informationsverarbeitung und Kommunikation, 24(3):141–148, September 2001, http://www.nm.informatik.uni-muenchen.de/Literatur/MNMPub/Publikationen/krrv01/krrv01.shtml .

[LBN 99] LOBO, J., R. BHATIA und S. NAQVI: *A Policy Description Language*. In: *Proceedings of the Sixteenth National Conference on Artificial Intelligence Eleventh Innovative Applications of AI Conference, Orlando, Florida, USA*, 1999.

[LuSl 97] LUPU, E. und M. SLOMAN: *Conflict Analysis for Management Policies*. In: LAZAR, A., R. SARACCO und R. STADLER (Herausgeber): *Integrated Network Management V (IM'97)*, San Diego, USA, Mai 1997. Chapman & Hall.

[LuSL 97b] LUPU, E. und M. SLOMAN: *A Policy Based Role Object Model*. In: *Proceedings of EDOC'97*, oct 1997.

[Mari 97] MARRIOTT, DAMIAN A.: *Policy Service for Distributed Systems*. Doktorarbeit, Imperial College London, Juni 1997.

[MESW 01] MOORE, B., E. ELLESSON, J. STRASSNER und A. WESTERINEN: *RFC 3060: Policy Core Information Model – Version 1 Specification*. RFC, IETF, Februar 2001, ftp://ftp.isi.edu/in-notes/rfc3060.txt .

[OMG 00-06-20] *Notification Service stand-alone document*. OMG Specification formal/00-06-20, Object Management Group, Juni 2000, ftp://ftp.omg.org/pub/docs/formal/00-06-20.pdf .

[OMG 01-06-07] *Java Language to IDL Mapping*. OMG Specification formal/01-06-07, Object Management Group, Juni 2001, ftp://ftp.omg.org/pub/docs/formal/01-06-07.pdf .

[OMG 96-09-08] *TINA Services Architecture*. TC Document telecom/96-09-08, Object Management Group, September 1996.

[OMG 97-02-08] *CORBAservices - Naming Service*. OMG Specification formal/97-02-08, Object Management Group, Februar 1997, ftp://ftp.omg.org/pub/docs/formal/97-02-08.pdf .

[Radi 02] 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, http://www.nm.informatik.uni-muenchen.de/Literatur/MNMPub/Publikationen/radi02/radi02.shtml .

[Radi 03] RADISIC, I.: *Ein prozessorientierter, policy–basierter Ansatz für ein integriertes, dienstorientiertes Abrechnungsmanagement*. Doktorarbeit, Ludwig-Maximilians-Universität München, Januar 2003.

[Radi 98] RADISIC, I.: *Konzeption eines policy-basierten Konfigurationsmanagements für nomadische Systeme in Intranets*. Diplomarbeit, Ludwig-Maximilians-Universität München, August 1998, http://www.nm.informatik.uni-muenchen.de/common/Literatur/MNMPub/Diplomarbeiten/radi98/radi98.shtml .

[RJB 99] RUMBAUGH, J., I. JACOBSON und G. BOOCH: *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[SlLu 99] SLOMAN, M. AND LUPU, E.: *Policy Specification for Programmable Networks*. In: *Proceedings of First International Working Conference on Active Networks*, 1999.

[TINA 96] HAMADA, T.: *Accounting Management Architecture*. Version 1.3 (Draft), TINA Consortium, Februar 1996.

[TMF 910] *Telecom Operations Map (GB 910)*. Technischer Bericht, TeleManagement Forum, April 1999, http://www.tmforum.org/pages/publications/gb910.pdf .

[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.

[XHTML] PEMBERTON, S. ET AL: *XHTML 1.0 : The Extensible HyperText Markup Language (Second Edition)*. W3C Recommendation, W3C, 2000, http://www.w3.org/TR/2002/REC-xhtml1-20020801 .

[XLink] DEROSE, S., E. MALER und D. ORCHARD: *XML Linking Language (XLink) Version 1.0*. W3C Recommendation, W3C, 2001, http://www.w3.org/TR/2000/REC-xlink-20010627/ .

[XML 00] BRAY, T., J. PAOLI, C. M. SPERBERG-MCQUEEN und E. (EDS.) MALER: *Extensible Markup Language (XML) 1.0*. W3C Recommendation, W3C, 2000, http://www.w3.org/TR/2000/REC-xml-20001006/ .

[XMLNS] BRAY, T., D. HOLLANDER und A. LAYMAN: *Namespaces in XML*. W3C Recommendation, W3C, 1999, http://www.w3.org/TR/1999/REC-xml-names/19990114/ .

[XMLS-0] FALLSIDE, D. C. (EDITOR): *XML Schema Part 0: Primer*. W3C Recommendation REC-xmlschema-0-20010502, World Wide Web Consortium (W3C), Mai 2001, http://www.w3.org/TR/xmlschema-0/ .

[XMLS-1] THOMPSON, H. S., D. BEECH, M. MALONEY und N. (EDS.) MEHDELSOHN: *XML Schema Part 1: Structures*. W3C Recommendation REC-xmlschema-1-20010502, World Wide Web Consortium (W3C), Mai 2001, http://www.w3.org/TR/xmlschema-1/ .

[XMLS-2] BIRON, P. V. und A. (EDS.) MALHOTRA: *XML Schema Part 2: Datatypes*. W3C Recommendation REC-xmlschema-2-20010502, World Wide Web Consortium (W3C), Mai 2001, http://www.w3.org/TR/xmlschema-2/ .

[XPath] CLARK, J. und S. (EDS.) DEROSE: *XML Path Language (XPath)*. W3C Recommendation, W3C, 2002, http://www.w3.org/TR/1999/REC-xpath-19991116 .

[XQu 02] BOAG, S., D. CHAMBERLIN und M. F. ET AL (EDS.) FERNANDEZ: *XQuery 1.0 : An XML Query Language*. W3C Recommendation, W3C, 2002, http://www.w3.org/TR/2002/WD-xquery-20021115/ .

[XSL] ADLER, S., A. BERGLUND, J. CARUSO, S. DEACH, T. GRAHAM, P. GROSSO, E. GUTENTAG, A. MILOWSKI, S. PARNELL, J. RICHMAN und S. ZILLES: *Extensible Stylesheet Language (XSL) Version 1.0*. W3C Recommendation, W3C, 2001, http://www.w3.org/TR/2001/REC-xsl-20011015/ .

[XSLT] CLARK, J. (ED.): *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation, W3C, 1999, http://www.w3.org/TR/1999/REC-xslt-19991116 .

[ZZC 02] ZSEBY, T., S. ZANDER und C. CARLE: *RFC 3334: Policy-Based Accounting*. RFC, IETF, Oktober 2002, ftp://ftp.isi.edu/in-notes/rfc3334.txt .