

Entwurf und Implementierung eines Übersetzers für eine Zugriffskontrollsprache nach XACML

Roland Illig

5. November 2008

**Diplomarbeit am
Arbeitsbereich „Sicherheit in verteilten Systemen“,
Department Informatik,
Universität Hamburg**

**Erstbetreuer: Prof. Dr. Joachim Posegga
Zweitbetreuer: Prof. Dr. Norbert Ritter
Technische Betreuung: Christopher Alm**

Inhaltsverzeichnis

1	Einleitung	7
1.1	Aufbau der Arbeit	8
1.2	Dank	8
2	Grundlagen	11
2.1	Begriffe	11
2.2	Zugriffskontrollmodelle	12
2.3	Zugriffskontrollmatrix	13
2.4	RBAC	13
2.4.1	Das RBAC-Modell von Ferraiolo und Kuhn	14
2.4.2	Alternative Modelle	14
2.4.3	Static Separation of Duties	15
2.4.4	Dynamic Separation of Duties	15
2.4.5	History-based Separation of Duties	16
2.5	ORKA	16
2.6	XACML	17
2.7	Zusammenfassung	17
3	ORKA und OPL	19
3.1	Grundbegriffe des ORKA-Modells	19
3.2	Aufbau der Modulbeschreibungen	19
3.3	RBAC-Core	21
3.3.1	Datentypen	21
3.3.2	Persistente Daten	22
3.3.3	Schnelllebige Daten	22
3.3.4	Zugriffsentscheidung	22
3.3.5	Zusammenfassung	23
3.4	RBAC mit Rollenhierarchie	23
3.4.1	Datentypen	23
3.4.2	Persistente Daten	23
3.4.3	Schnelllebige Daten	23
3.4.4	Zugriffsentscheidung	23
3.4.5	Zusammenfassung	24
3.5	Context Constraints	24
3.5.1	Datentypen	25
3.5.2	Persistente Daten	26

3.5.3	Schnelllebige Daten	28
3.5.4	Zugriffsentscheidung	28
3.5.5	Zusammenfassung	28
3.6	Separation of Duty	29
3.6.1	Datentypen	29
3.6.2	Persistente Daten	29
3.6.3	Schnelllebige Daten	30
3.6.4	Zugriffsentscheidung	30
3.7	Separation of Duty mit Rollenhierarchie	31
3.7.1	Datentypen	31
3.7.2	Persistente Daten	31
3.7.3	Schnelllebige Daten	31
3.7.4	Zugriffsentscheidung	31
3.8	Weitere Module	31
3.9	Kombination der Zugriffsentscheidungen	31
3.10	Zusammenfassung	32
4	XACML	33
4.1	Übersicht	33
4.1.1	Begriffe	33
4.1.2	Ablauf einer Zugriffsanfrage	34
4.2	Datenstrukturen	36
4.3	Abkürzende Notation für XACML	36
4.3.1	Beispiel für die Notation in sXACML	37
4.4	Anfrage	38
4.4.1	Mehrere Subjekte in einer Anfrage	38
4.4.2	Beispielanfrage für mehrere Subjekte	39
4.5	Aufbau der Attribute	41
4.5.1	Beispiele für Attribute	41
4.6	Datentypen	42
4.6.1	Änderungen in XACML 2.0	44
4.7	Policys	44
4.8	Anwendungsbereich	45
4.8.1	Optimierungsmöglichkeiten	46
4.9	Abgleich (Match)	48
4.9.1	Vordefinierte Funktionen	48
4.10	Arten von Ausdrücken	52
4.11	Regeln	54
4.12	Kombinationsalgorithmen	55
4.13	Auswertung einer Policy	56
4.14	Antwort des PDP	56
4.15	Geschichte und aktuelle Entwicklung	57
4.15.1	In Planung: XACML 3.0	57
4.16	Möglichkeiten und Grenzen	58

4.17 Zusammenfassung	58
5 Übersetzung	61
5.1 Gründe für die Übersetzung	61
5.2 Umfang der Übersetzung	61
5.2.1 Lösung 1: RBAC-Autorisierungsserver	61
5.2.2 Lösung 2: XACML-Autorisierungsdienst	62
5.2.3 Praktische Umsetzung	63
5.3 Umgebung des Übersetzers	63
5.4 Komponenten der ORKA-Policys	63
5.5 Übersetzung der ORKA-Datentypen	64
5.6 Struktur der erzeugten XACML-Policys	65
5.7 RBAC-Core	65
5.7.1 Einfache Implementierung	66
5.7.2 Ausklammern einzelner Teile	67
5.7.3 Bewertung der Varianten	68
5.7.4 Konkrete Übersetzung	68
5.8 RBAC mit Rollenhierarchie	70
5.9 Context Constraints	70
5.9.1 Übersetzung der einzelnen Parameter	71
5.9.2 Übersetzung der Funktionsaufrufe	73
5.9.3 Erzeugen der XACML-Policys	73
5.10 Separation of Duty	78
5.11 Separation of Duty mit Rollenhierarchie	78
5.12 Kombination der Module	78
5.13 Zusammenfassung	79
6 Implementierung der Übersetzung	81
6.1 ORKA-Quelltext	81
6.1.1 Organisation des Quellcodes	81
6.2 sunxacml	82
6.2.1 Implementierungsfehler	82
6.3 Java-Pakete und Klassen des Übersetzers	83
6.4 Eingliederung des Übersetzers in ORKA	83
6.5 Übersetzung der Berechtigungsmodule	84
6.5.1 RBAC-Core	84
6.5.2 RBAC mit Rollenhierarchien	84
6.5.3 Context Constraints	85
6.6 Übersetzung der ORKA-Anfrage in XACML	86
6.7 Evaluation der Anfrage	86
6.8 Auflösen der schnelllebigen Daten	87
6.9 Übersetzung der XACML-Antwort in ORKA	87
7 Zusammenfassung und Ausblick	89

Inhaltsverzeichnis

7.1	Zusammenfassung der Arbeit	89
7.2	Auswirkungen auf die Welt	90
7.3	Ausblick	90
A	Beispiel	91
A.1	OPL-Policy	91
A.2	Übersetzte XACML-Policy	96
B	Implementierter Code	109
B.1	Hilfsklassen	127

1 Einleitung

In den letzten Jahrzehnten sind Computer immer leistungsfähiger und Speicher immer größer geworden. Durch das Internet sind viele Computer vernetzt und können Daten austauschen. Viele dieser Daten sind vertraulich oder ihre Integrität muss gewährleistet bleiben. Dazu können die Daten auf einem abgeschirmten System gespeichert werden, das mittels Zugriffskontrolle sicherstellt, dass nur berechtigte Personen auf diese Daten zugreifen können. Welche Zugriffe erlaubt und welche verboten sind, wird in einer Zugriffsrichtlinie festgelegt.

Derzeit (2008) ist es so, dass viele Systeme, in denen Daten gespeichert sind (Dateisystem, Datenbank, Verzeichnis), eigene Zugriffsregeln haben, die jeweils auf den speziellen Anwendungsbereich zugeschnitten sind. Die dabei verwendeten Zugriffskontrollmodelle sind in vielen Bereichen ähnlich, unterscheiden sich jedoch oft in kleinen Details. Dadurch ist es in umfangreichen Systemen nötig, die Zugriffsregeln in einer allgemeinen, systemunabhängigen Sprache zu formulieren und diese dann in die konkreten Modelle der einzelnen Systeme zu übersetzen. Eine solche unabhängige Sprache hat die „Organization for the Advancement of Structured Information Standards“ (OASIS) im Jahre 2003 unter dem Namen XACML veröffentlicht. XACML stellt ein allgemeines Rahmenwerk zur Verfügung, um die Zugriffskontrolle systemunabhängig zu formulieren. Die Zugriffskontrolle basiert in dieser Sprache auf einzelnen Regeln, die flexibel zu größeren Einheiten kombiniert werden können.

Richtlinien, die in dieser Sprache formuliert sind, sind jedoch sehr schreibaufwendig: Eine Regel, die in natürlicher Sprache gerade einmal zwei Zeilen lang ist, umfasst in der XML-Darstellung von XACML zwei DIN-A4-Seiten.¹ XACML-Policys sind daher nicht geeignet, direkt von Menschen verfasst und bearbeitet zu werden. Stattdessen ist es sinnvoller, in den Bereichen, die von Menschen verwaltet werden, eine spezialisierte und einfacher strukturierte Zugriffskontrollsprache zu benutzen, die sich an den Anforderungen typischer Organisationen orientiert. Zum Durchsetzen von Zugriffsrichtlinien in heterogenen Systemen ist XACML jedoch gut geeignet. Daher ist eine naheliegende Idee, die menschenorientierte Sprache in XACML zu übersetzen, so dass die Vorteile beider Sprachen vereinigt werden.

Die in dieser Arbeit verwendete Quellsprache stammt aus dem ORKA-Projekt und heißt OPL (ORKA Policy Language). Sie basiert auf RBAC („Role Based Access Control“) und ist daher in den Grundzügen leicht verständlich. Insbesondere sind die Datenstrukturen an die menschliche Kognition angepasst und können daher von Menschen leicht interpretiert werden.

¹[XACML-2.0, Abschnitt 4.2.3, Regel 2]

1 Einleitung

Das Kernstück dieser Arbeit ist die Übersetzung von OPL-Policys in äquivalente XACML-Policys. Durch diese Übersetzung werden die Vorteile beider Sprachen vereint: Auf der OPL-Seite die menschenfreundliche und übersichtliche Darstellung, und auf der XACML-Seite die maschinenorientierte Darstellung in einer standardisierten Sprache, um die Zusammenarbeit zwischen unterschiedlichen Systemen zu ermöglichen.

1.1 Aufbau der Arbeit

Im Bereich der Zugriffskontrolle gibt es viele Definitionen für Begriffe, die sich teilweise widersprechen. Die in dieser Arbeit verwendeten Begriffe werden in Kapitel 2 definiert. Zusätzlich werden einige Zugriffskontrollmodelle vorgestellt, um eine grobe Übersicht zu geben.

Letztendlich soll in der Arbeit ein Übersetzungsprogramm entwickelt werden. Eine Übersetzung findet generell von einer Quellsprache in eine Zielsprache statt. In den Kapiteln 3 und 4 werden diese beiden Sprachen detailliert vorgestellt. Für die spätere Übersetzung ist besonders der strukturelle Aufbau der Sprachen interessant, deshalb wird darauf besonders intensiv eingegangen. Sowohl bei der Quellsprache (OPL) als auch bei der Zielsprache (XACML) werden die Designentscheidungen an einigen Stellen nachvollzogen und Alternativen für Designentscheidungen aufgezeigt, die zu einem einfacheren Modell führen.

Der Übersetzer selbst wird in Kapitel 5 entwickelt. Die Quellsprache ist in mehrere Zugriffskontrollmodule unterteilt, die jeweils einzeln übersetzt und schließlich kombiniert werden. Die bei der Übersetzung möglichen Alternativen werden durchdacht und jeweils eine begründete Entscheidung getroffen.

In Kapitel 6 wird die konkrete Implementierung dokumentiert. Zunächst werden dort die Quelltexte des ORKA-Projekts und der XACML-Implementierung kritisch beurteilt. Anschließend wird der von mir entwickelte Code vorgestellt und detailliert dokumentiert. Dieses Kapitel ist dazu gedacht, anhand des Codes nachzuvollziehen, was die besonderen Schwierigkeiten bei der Übersetzung sind.

Schließlich wird in Kapitel 7 die Arbeit zusammengefasst und ein Ausblick gegeben.

1.2 Dank

Zum guten Gelingen dieser Diplomarbeit haben beigetragen:

- Christopher Alm, der mich durchgehend hervorragend unterstützt hat, indem er mir anfangs Starthilfe beim Einarbeiten gegeben hat, zwischendurch die Entwürfe meiner Arbeit probegesehen hat und mich oft motiviert hat, intensiver an der Arbeit weiterzumachen als bisher.
- Tom Bergmann, Mieke Hildebrand und Tobias Müller, die meine Diplomarbeit kritisch durchgesehen haben und durch ihre Kommentare die Lesbarkeit dieser Arbeit stark verbessert haben.

- Meine Familie, durch deren Geduld und finanzielle Unterstützung ich mich voll auf die Diplomarbeit konzentrieren konnte.
- Meine alte Festplatte, die zwar unverhoffterweise den Geist aufgegeben hat, aber auch danach gerade noch so weit funktioniert hat, dass ich alle meine Daten retten konnte.

1 Einleitung

2 Grundlagen

In diesem Abschnitt dieses Kapitels werden die Begriffe definiert, die im Rest der Arbeit verwendet werden. Der Rest des Kapitels stellt einige Zugriffskontrollmodelle vor, beginnend mit der sehr einfachen Zugriffskontrollmatrix, bis hin zu XACML.

2.1 Begriffe

Bei der **Zugriffskontrolle** ist die zentrale Frage: *Wer darf was mit welchen* Objekten machen, und unter welchen *Bedingungen*? Um diese allgemeine Frage zu beantworten, ist es nützlich, ein gemeinsames Vokabular zu haben, damit sich alle Beteiligten auf technischer Ebene einig sind. Insbesondere bei der Zugriffskontrolle gibt es verschiedene Modelle, die unterschiedliche Begriffe für gleiche Konzepte verwenden oder aber auch gleiche Begriffe für unterschiedliche Konzepte. Um Missverständnisse auszuschließen, werden die zentralen Begriffe hier definiert und erläutert.

Ein **Benutzer** (engl.: *User*) ist ein Mensch, der prinzipiell berechtigt ist, sich am System anzumelden. In vielen Zugriffskontrollmodellen kann ein Benutzer zu einem bestimmten Zeitpunkt mehrmals am System angemeldet sein. Jede dieser Anmeldungen entspricht einem Subjekt.

Ein **Subjekt** (engl.: **Subject**), oft auch **Session** genannt, ist ein aktiv handelndes Etwas. (Das „wer“ aus der einleitenden Frage.) Dabei kann es sich um einen Betriebssystem-Prozess handeln, einen am System angemeldeten Benutzer, ein bestimmtes Anwendungsprogramm, etc. Die hauptsächlich verwendete Bedeutung innerhalb dieser Arbeit ist der am System angemeldete Benutzer. In dem Fall wird beim Anmelden ein Subjekt erzeugt, das alle Berechtigungen des Benutzers übernimmt. Diese Berechtigungen können sich zwischenzeitlich ändern, zum Beispiel durch das Hinzufügen oder Entfernen von Rollen. Wenn der Benutzer sich vom System abmeldet, wird das dazugehörige Subjekt zerstört. In vielen Zugriffskontrollmodellen wird nicht zwischen Benutzern und Subjekten unterschieden. Ein Beispiel, in dem der Unterschied deutlich wird, ist die Rechtstrennung in RBAC (siehe Abschnitte 2.4.3 und 2.4.4).

Eine **Rolle** (engl.: *Role*) ist eine benannte Zusammenfassung von Benutzern, die in einem System die gleichen Berechtigungen haben (Beispiel: Manager, Systemadministrator, Sachbearbeiter). Durch die Einführung von Rollen müssen die Berechtigungen nicht mehr jedem Benutzer einzeln zugewiesen werden. Stattdessen bekommt ein Benutzer eine Menge von Rollen zugewiesen, und diese Rollen wiederum bekommen die Berechtigungen. Dadurch wird die Verwaltung vereinfacht und übersichtlicher gehalten.

Eine **Ressource** (engl.: *Resource*) oder **Objekt** (engl.: *Object*) ist ein passives Etwas, das durch die Zugriffskontrolle geschützt wird. Beispiele für Objekte sind Dateien, ganze Datenbanken, aber auch ein einzelnes Datenfeld in einem ganz bestimmten Datensatz.

Eine **Aktion** (engl.: *Action*) oder **Operation** ist das, was ein Subjekt mit einer Ressource machen kann. (Das „was“ aus der einleitenden Frage.) Typische Beispiele sind: auslesen, hinzufügen, löschen, verändern (für Dateien), bestätigen, unterschreiben (für Dokumente).

Eine **Berechtigung** (engl.: *Permission*) ist eine Zusammenfassung von einer Aktion und einer Ressource, da diese Kombination an vielen Stellen innerhalb der Zugriffskontrolle auftaucht und unabhängig von der konkreten Aktion und der Ressource betrachtet werden kann.

2.2 Zugriffskontrollmodelle

Um die Zugriffskontrolle umzusetzen, gibt es viele verschiedene Modelle, die sich vor allem in der Verständlichkeit und der Skalierbarkeit (Anzahl der Benutzer, der zu schützenden Objekte und sonstiger Parameter) unterscheiden. Ein paar davon werden im folgenden beschrieben.

Ein Zugriffskontrollmodell wird in dieser Arbeit als zweigeteiltes Softwaresystem angesehen, das aus *Daten* und *Code* besteht.

1. Die *Daten* können sich im Laufe der Zeit ändern. Beispiele hierfür sind Mengen von Benutzern, Rollen, Operationen, Objekten sowie Relationen zwischen diesen Mengen.
2. Der *Code* ändert sich im Laufe der Zeit nicht. Er benutzt die Daten, um zu einer Zugriffsentscheidung zu kommen.

Je nach Zugriffskontrollmodell sind diese beiden Teile unterschiedlich komplex ausgeprägt. Ob ein Teil eines Zugriffskontrollmodells als Daten oder als Code aufzufassen ist, lässt sich nur abhängig vom jeweiligen Modell entscheiden. So kann das, was in einem einfachen Modell der Code ist, in komplexeren Modellen durchaus innerhalb der Daten dargestellt werden, zum Beispiel wenn es möglich ist, ausführbaren Code in die Daten zu integrieren. Die folgenden Beispiele sollen nur einen ersten Eindruck vermitteln. Die dazugehörigen Modelle werden in den folgenden Abschnitten noch ausführlicher erklärt.

- Bei einer Zugriffskontrollmatrix (Abschnitt 2.3) bestehen die Daten lediglich aus einer dreistelligen Relation (Subjekt, Aktion, Objekt). Der Code zur Zugriffsentscheidung besteht daraus, einen einzelnen Eintrag aus der Relation auszulesen.
- Bei RBAC (Abschnitt 2.4) sind die Daten in zwei Relationen unterteilt (eine, um von Subjekten zu Rollen zu gelangen, und eine, um von Rollen zu Berechtigungen zu gelangen), und der Code verknüpft die Daten der beiden Relationen, um zur Zugriffsentscheidung zu gelangen.
- Bei XACML (Abschnitt 2.6) können die aus typischen Programmiersprachen bekannten Ausdrücke innerhalb der Daten repräsentiert werden. Mit diesen Ausdrücken lassen sich komplexe Regeln formulieren. Der Code legt hier nur das Verfahren fest,

	O1	O2	O3	O4
alice	rwX	rwX	rwX	rwX
bob	--	--	r-	r-
charly	rwX	rwX	r-	r-
dennis	r-	r-	r-	r-

Abbildung 2.1: Eine Zugriffskontrollmatrix mit 4 Subjekten und 4 Objekten

nach dem die in den Daten formulierten Regeln interpretiert werden. Er ist mit einem Interpreter für Programmiersprachen vergleichbar.

2.3 Zugriffskontrollmatrix

Das einfachste Modell, um Zugriffe von bestimmten Subjekten auf bestimmte Objekte zu beschreiben, ist eine Funktion, die als Parameter ein Subjekt, ein Objekt und eine Aktion bekommt und entweder `true` oder `false` zurückgibt, je nachdem, ob der Zugriff erlaubt sein soll oder nicht. Die strukturell einfachste Form, diese Funktion zu definieren, ist über eine dreistellige Relation ($S \times Op \times Ob$). Die traditionelle graphische Darstellung dieser Relation ist eine zweidimensionale Tabelle, in der die erlaubten Operationen pro Subjekt und Objekt zusammengefasst dargestellt werden. Aus dieser Darstellungsform leitet sich auch der Name „Zugriffskontrollmatrix“ ab.

Die Zugriffskontrollmatrix ist in erster Linie ein theoretisches Modell. In Anwendungsbereichen mit nur wenigen Subjekten und Objekten kann die Zugriffskontrolle auch direkt als Matrix implementiert werden. Ein Beispiel für eine Zugriffskontrollmatrix ist in Abbildung 2.1 dargestellt. Die möglichen Operationen für jede Kombination von Subjekt und Objekt sind `read`, `write` und `execute`.

Wenn die Anzahl der Subjekte oder der Ressourcen steigt, wird sehr schnell klar, worin die Grenzen der Matrix liegen: Für jeden Benutzer und für jede Ressource müssen die erlaubten Operationen explizit festgelegt werden. In der Praxis ist es jedoch so, dass die erlaubten Operationen meistens nicht direkt von einer natürlichen Person abhängen, sondern eher von der Rolle, die diese Person in einer Organisation einnimmt. Aus der Zugriffskontrollmatrix ist auch nicht direkt ersichtlich, *warum* bzw. für welchen Zweck ein Subjekt bestimmte Rechte hat. In rollenbasierten Modellen sind dagegen die Berechtigungen an benannte Rollen gebunden, so dass man durch den Namen der Rolle einen Anhaltspunkt für die Berechtigungen hat.

2.4 RBAC

Die Grundidee der *Rollenbasierten Zugriffskontrolle* (engl.: *Role Based Access Control*; RBAC) ist, dass ein Subjekt seine Berechtigungen nicht aufgrund seiner Identität hat, sondern aufgrund der Rollen, die es in einer Organisation ausübt. Daher werden die

Berechtigungen – anders als bei der Zugriffskontrollmatrix – nicht direkt an Subjekte vergeben, sondern über den Zwischenschritt der Rollen.

2.4.1 Das RBAC-Modell von Ferraiolo und Kuhn

Das RBAC-Modell wurde erstmals 1992 in einem Artikel von David Ferraiolo und Richard Kuhn beschrieben [RBAC1992] und 2004 von ANSI standardisiert [ANSI-RBAC]. Das Modell basiert auf drei disjunkten Grundmengen und einigen Relationen zwischen diesen Mengen:

- S , die Menge der Subjekte. Im traditionellen RBAC-Modell wird nicht zwischen (persistenten) Benutzern und (schnelllebigen) Subjekten unterschieden.
- R , die Menge der Rollen.
- T , die Menge der Transaktionen (in dieser Arbeit „Berechtigungen“ genannt).

Ein Subjekt kann zu jedem Zeitpunkt nur eine einzige „aktive Rolle“ haben. Diese Rolle wird von der Funktion $AR: S \rightarrow R$ zurückgegeben. Ein Subjekt kann jedoch mehr als einer Rolle „zugewiesen sein“, was in der Relation $RA \subset S \times R$ (Role Assignment) ausgedrückt wird. Die Verbindung zwischen Rollen und Transaktionen schafft die Relation $TA \subset R \times T$ (Transaction Assignment), die bestimmt, welche Transaktionen eine bestimmte Rolle ausführen darf.

Ob ein Subjekt eine bestimmte Transaktion durchführen kann, entscheidet die Funktion $exec: S \times T \rightarrow \mathbb{B}$. In [RBAC1992] wird diese Funktion nicht explizit angegeben, es werden nur Kriterien aufgestellt, wann diese Funktion garantiert **false** zurückgibt. Es gibt mehrere Funktionen, die alle Kriterien erfüllen. Die Funktion, die den Bedingungen an $exec$ genügt und ansonsten immer **true** zurückgibt, ist:

$$exec(s, t) := (AR(s), t) \in TA.$$

In Worten: Das Subjekt s muss eine gerade aktive Rolle haben, und dieser Rolle muss es erlaubt sein, die gewünschte Transaktion t auszuführen.

2.4.2 Alternative Modelle

Da die Idee, die Berechtigungen an Rollen statt direkt an Subjekte zu vergeben, nicht sonderlich schwierig zu finden ist, gibt es viele RBAC-Modelle, die sich nur in Kleinigkeiten unterscheiden. Einige der möglichen Unterschiede sind:

- Ein Subjekt kann nicht nur eine einzige aktive Rolle haben, sondern beliebig viele. Um eine Transaktion auszuführen, genügt es, wenn eine beliebige der aktiven Rollen berechtigt ist, die Transaktion auszuführen.
- Unter den Rollen kann eine Hierarchie eingeführt werden, um die Rollenzuweisung übersichtlich zu halten. Dadurch muss beispielsweise den Sachbearbeitern einer Firma nicht explizit die Rolle „Mitarbeiter“ zugewiesen werden, da diese Rolle schon implizit in der Rolle „Sachbearbeiter“ enthalten ist.

- Es können zahlreiche weitere Einschränkungen an die Berechtigungen vorgenommen werden, die in den folgenden Abschnitten ausführlicher erklärt werden, insbesondere verschiedene Formen der Rechtstrennung (engl.: Separation of Duty).
- Es kann eine Unterscheidung zwischen Benutzern und Subjekten geben. Die Relation $UA \subset U \times R$ (User Assignment) gibt dann an, welche Rollen ein Subjekt dieses Benutzers überhaupt einnehmen darf, während $SR \subset S \times R$ (Subject Roles) die gerade aktiven Rollen darstellt. Die Zuordnung von Subjekten zu Benutzern übernimmt die Funktion $SU: S \rightarrow U$ (Subject User).

2.4.3 Static Separation of Duties

Bei der rollenbasierten Zugriffskontrolle gibt es oft Kombinationen von Rollen, für die sichergestellt sein soll, dass kein Subjekt allen diesen Rollen angehört. Diese Forderung wird *Statische Rechtstrennung* (engl.: *Static Separation of Duties (SSoD)*) genannt.

Beispiel: In einer Firma soll niemand jemals gleichzeitig Administrator und Sachbearbeiter sein. Im Identitätsmanagementsystem (IMS) der Firma wird daher eingetragen, dass jeder Benutzer aus der Rollenmenge {Administrator, Sachbearbeiter} nur maximal einem Element zugewiesen sein darf.

2.4.4 Dynamic Separation of Duties

Nicht ganz so streng wie SSoD ist die *Dynamische Rechtstrennung* (engl.: *Dynamic Separation of Duties (DSoD)*). Hier wird die Unterscheidung zwischen Benutzer und Subjekt deutlich, denn ein Benutzer darf durchaus mehreren kritischen Rollen zugeordnet sein, die Subjekte, die zu diesem Benutzer gehören, jedoch nicht.

Es gibt zwei leicht unterschiedliche Formen von DSoD. Bei der ersten wird jedes Subjekt einzeln betrachtet und darf für sich die DSoD-Bedingung nicht verletzen. Bei der zweiten Form werden alle Subjekte eines Benutzers gemeinsam betrachtet, und die Rollen aller Subjekte zusammengekommen dürfen die DSoD-Bedingung nicht verletzen.

Beispiel: Ein Benutzer hat zwei mögliche Rollen, Administrator und Sachbearbeiter. Diese beiden Rollen stehen jedoch auch in der DSoD-Relation. Bei der ersten Form darf der Benutzer sich nicht als „Administrator und Sachbearbeiter“ gleichzeitig anmelden. Er kann sich jedoch in zwei verschiedenen Sitzungen anmelden, einmal als Administrator und einmal als Sachbearbeiter.

Bei der zweiten Form ist diese doppelte Anmeldung nicht möglich. Wenn sich der Benutzer zuerst als Administrator anmeldet, wird die zweite Anmeldung als Sachbearbeiter fehlschlagen, da es sonst einen Benutzer geben würde, dessen Gesamtheit der Subjekte die beiden kritischen Rollen Administrator und Sachbearbeiter inne hätte.

2.4.5 History-based Separation of Duties

Bei der verlaufsorientierten Rechtstrennung (engl.: history-based separation of duties) wird eine Folge von Vorgängen an einer Ressource betrachtet. In dieser Folge dürfen bestimmte Vorgänge nicht von ein und demselben Benutzer ausgeführt werden. (Beispiel: Das Ausarbeiten und das Genehmigen einer bestimmten Beschaffung sollten von zwei unterschiedlichen Personen vorgenommen werden.) Der Spezialfall, dass mindestens zwei Benutzer an dem Vorgang beteiligt sein müssen, wird „keyword Vier-Augen-Prinzip“ genannt.

Umsetzung

Bei der Umsetzung dieser Einschränkung sollte man so vorgehen, dass möglichst früh erkannt wird, ob ein Geschäftsprozess überhaupt noch gemäß der SoD-Einschränkung fertiggestellt werden kann. Es soll beispielsweise nie vorkommen, dass in einem Prozess nur noch zwei Schritte zu erledigen sind, aber noch drei weitere Personen beteiligt sein müssen.

Als einfaches Modell wird hier ein Geschäftsprozess aus S Schritten betrachtet, wobei an jedem Schritt genau ein Subjekt beteiligt ist. Weiterhin darf jedes Subjekt jeden Schritt ausführen. Letztendlich sollen an dem Geschäftsprozess N Subjekte mitgewirkt haben. Es seien schon s Schritte getan, und die Liste der mitwirkenden Subjekte umfasse bisher n Einträge.

Falls das aktuelle Subjekt noch nicht in der Liste der Mitwirkenden enthalten ist, kann der Schritt erlaubt werden, da es nur vorteilhaft ist, wenn die Liste der Mitwirkenden schnell größer wird. Falls das Subjekt allerdings schon in der Liste enthalten ist, sollte durch die Zugriffskontrolle sichergestellt werden, dass für die restlichen Schritte noch genug Subjekte verfügbar sind. Andernfalls könnte es passieren, dass der Geschäftsprozess nicht mehr ordnungsgemäß beendet werden kann. Es darf also nie zu einer Situation kommen, in der noch mehr neue Subjekte benötigt werden als Schritte zu erledigen sind, d. h. es muss stets gelten: $S - s \leq N - n$.

Falls die im ersten Absatz gemachten Annahmen nicht zutreffen, müssen ausgeklügeltere Algorithmen benutzt werden, um die Forderung sicherzustellen.

2.5 ORKA

ORKA ist ein Forschungsprojekt, das vom Bundesministerium für Bildung und Forschung unterstützt wird. Ziel des Projektes ist es, ein Zugriffskontrollmodell zu entwickeln, das die Kontrollkonzepte, die typischerweise in Unternehmen vorkommen, einfach umsetzen kann. Solche Konzepte sind zum Beispiel Rechtstrennung oder das Vier-Augen-Prinzip. Der Schwerpunkt des ORKA-Projekts liegt darin, die Zugriffskontrolle bei Geschäftsprozessen zu modellieren. Das Zugriffskontrollmodell soll später in bestehende Enterprise Resource Planning („ERP“)-Systeme integriert werden.

Das ORKA-Modell basiert auf RBAC und ergänzt es durch zahlreiche einschränkende Kontrollkonzepte. Das so erzeugte Modell wird in Java implementiert und mittels dreier verschiedener Techniken durchgesetzt:

- Nativ, also mit einer Policy-Entscheidungs-Komponente, die direkt auf den ORKA-Datenstrukturen arbeitet.
- Mit „Drools“, einem Geschäftsregel-Managementsystem, das auf einer deklarativen Programmiersprache basiert.
- Mit XACML, einer Zugriffskontrollsprache, die nicht auf einer allgemeinen Programmiersprache basiert, sondern nur auf schachtelbaren Ausdrücken, um die Berechnungskomplexität klein zu halten.

Die Daten, die innerhalb des ORKA-Modells für die Zugriffskontrolle relevant sind, werden in der Sprache OPL („ORKA Policy Language“) formuliert. OPL besteht hauptsächlich aus Mengen und Relationen und kann daher relativ abstrakt formuliert werden. Eine konkrete Syntax wurde auf der Basis von XML definiert und heißt OPL/XML.

Eine ausführlichere Einführung in das Zugriffskontrollmodell von ORKA findet sich in Kapitel 3.

2.6 XACML

XACML ist eine Sprache zur Zugriffskontrolle, deren Ziel es ist, ein möglichst allgemeines Zugriffskontrollmodell bereitzustellen und trotzdem nahe an den Anforderungen der Anwendungen zu bleiben. Die Sprache wurde erstmals im Jahr 2003 standardisiert. Version 2.0 folgte zwei Jahre später, Version 3.0 ist in Vorbereitung.

Eine Zugriffsanfrage in XACML besteht aus einer Reihe von Attributen (Name-Wert-Paar). Diese Anfragen werden mit Hilfe von Policys evaluiert, um zu einer Zugriffsentscheidung zu gelangen. Innerhalb der Policys können die Anfragen mit den typischen aus Programmiersprachen bekannten Funktionen (Vergleiche, Arithmetik, einfache Zeichenkettenoperationen, Datums- und Zeitberechnungen) analysiert werden, wobei beliebige komplexe Ausdrücke möglich sind.

XACML wurde als eine allgemeine, abstrakte Norm entworfen und kann daher an vielen Stellen ergänzt werden. So können zum Beispiel anwendungsspezifische Attributnamen, Datentypen und Funktionen hinzugefügt werden.

Da XACML ein zentraler Baustein dieser Arbeit ist, wird die Sprache in Kapitel 4 noch detailliert erklärt.

2.7 Zusammenfassung

In diesem Kapitel wurden die grundlegenden Begriffe für den Bereich Zugriffskontrolle erläutert und einige Zugriffskontrollmodelle eingeführt. Ein sehr einfaches Modell ist die Zugriffskontrollmatrix, bei der jeder Kombination von Subjekt und Objekt jeweils die erlaubten Berechtigungen direkt zugewiesen werden. Danach wurde RBAC vorgestellt,

2 Grundlagen

ein geringfügig komplexeres Modell. Bei RBAC werden die Berechtigungen den Subjekten nicht direkt zugewiesen, sondern über den Zwischenschritt der Rollen. Dieses einfache Modell wird in der Praxis häufig durch zusätzliche Bedingungen eingeschränkt, zum Beispiel Rechtstrennung oder das Vier-Augen-Prinzip. Das Modell des ORKA-Projekts baut auf RBAC auf und wurde speziell für Geschäftsprozesse entwickelt. Das abstrakteste der vorgestellten Modelle ist das regelbasierte XACML, das alle anderen vorgestellten Modelle nachbilden kann.

Die nächsten beiden Kapitel beschäftigen sich intensiv mit zwei dieser Zugriffskontrollmodelle, ORKA und XACML. Im darauffolgenden Kapitel wird dann ein Übersetzer von OPL (der „ORKA Policy Language“) nach XACML entwickelt.

3 ORKA und OPL

Das Projekt ORKA wurde bereits in Abschnitt 2.5 grob dargestellt. Hier werden die Details des ORKA-Zugriffskontrollmodells dargestellt. Dieses Modell wird dann in Kapitel 5 in XACML übersetzt.

Das ORKA-Zugriffsmodell basiert auf einem RBAC-Modell und ergänzt diese Idee durch zusätzliche Berechtigungsmodule, die auf RBAC aufbauen, aber jeweils einige Details verändern. Diese Module treffen jeweils zu einer Zugriffsanfrage die Entscheidung, ob dieser Zugriff erlaubt oder verboten sein soll. Dadurch werden einige Zugriffe, die in reinem RBAC erlaubt sind, verboten. In einigen wenigen Fällen werden aber auch Zugriffe, die in reinem RBAC verboten sind, erlaubt.

Die Gesamtheit der in den folgenden Abschnitten beschriebenen Datenstrukturen wird *ORKA Policy Language (OPL)* genannt. Um Policies zwischen Komponenten auszutauschen, gibt es ein XML-Format, OPL/XML. Für diese Arbeit ist die konkrete XML-Repräsentation nicht relevant und wird daher auch nicht weiter behandelt.

In den folgenden Abschnitten werden die ORKA-Module vorgestellt, und am Ende des Kapitels wird diskutiert, wie die einzelnen Zugriffsentscheidungen der Module zu einer Gesamtentscheidung kombiniert werden.

3.1 Grundbegriffe des ORKA-Modells

Das ORKA-Zugriffskontrollmodell schützt *Objekte* vor unerlaubten *Zugriffen* durch *Subjekte*. Diese Begriffe werden so verwendet, wie sie in Kapitel 2 definiert wurden. Besonders ist allerdings, dass innerhalb von ORKA die Begriffe *Benutzer* und *Subjekt* klar unterschieden werden.

Ein Benutzer ist ein Mensch, der prinzipiell berechtigt ist, das durch ORKA geschützte System zu benutzen. Wenn sich ein Benutzer am System anmeldet, wird dadurch ein neues Subjekt erzeugt, bei dem alle Rollen aktiviert sind, die der Benutzer überhaupt aktivieren darf. Im weiteren Verlauf können diese Rollen einzeln deaktiviert und wieder aktiviert werden. Wenn der Benutzer sich vom System abmeldet, wird das Subjekt zerstört. Für die Zugriffskontrolle sind nur die Subjekte entscheidend, nicht jedoch die Benutzer an sich.

3.2 Aufbau der Modulbeschreibungen

Das Zugriffskontrollmodell von ORKA ist in Module unterteilt, die jeweils thematisch eigenständige Bereiche abdecken. Das grundlegende Modul „RBAC-Core“ definiert die für ein RBAC-Modell typischen Datenstrukturen wie Benutzer, Subjekte, Rollen und

Berechtigungen. Die anderen Module bauen auf diesem Modul auf und ergänzen es um neue Daten und/oder verändern die Zugriffsfunktion.

In den folgenden Abschnitten werden die einzelnen Module in einem einheitlichen Schema beschrieben. Dieses Schema bildet später auch die Grundlage für den Übersetzungsprozess. Die einzelnen Module sind in Object Z definiert, einer streng typisierten, objektorientierten Spezifikationssprache. Die Details der Sprache sind für diese Arbeit irrelevant und werden daher nicht behandelt. Einige Eigenschaften von Object Z, insbesondere die strenge Typisierung, beeinflussen jedoch die Art der Spezifikation. Die folgenden Beschreibungen unterteilen ein Modul in vier Bereiche, die jeweils getrennt behandelt werden. Von dieser Einteilung macht die Übersetzung in Kapitel 5 ausgiebigen Gebrauch.

- Im Einleitungssatz wird beschrieben, worauf das Modul basiert (Basismodul). Zwischen den einzelnen Modularten gibt es eine einfache Vererbungshierarchie. Einige der in ORKA verwendeten Module nutzen auch Mehrfachvererbung, diese Module werden jedoch in dieser Arbeit nicht behandelt. Ein Modul hat Zugriff auf alle Daten seines Basismoduls.
- Nach der Einleitung werden die Datentypen beschrieben, die von dem Modul definiert werden. Diese Datentypen werden für die Beschreibung der Variablen eines Moduls benötigt.
- Dann werden diejenigen Variablen beschrieben, die die persistenten Daten ausmachen. Die persistenten Daten bleiben über längere Zeiträume stabil. Sie werden üblicherweise nur von Administratoren geändert, nicht jedoch im normalen Betrieb des Systems.
- Anschließend werden die Variablen beschrieben, die die schnelllebigen Daten ausmachen. Hauptsächlich sind das die Daten, die mit einem Subjekt assoziiert sind.
- Nach den Variablen wird die Zugriffsfunktion *checkAccess(s, op, ob)* definiert, die einen Zugriff (*op*) eines Subjekts (*s*) auf ein Objekt (*ob*) prüft. Der Rückgabewert dieser Funktion ist entweder `true` oder `false`. Die Funktion ist in Python geschrieben, da der Quelltext in dieser Sprache knapp formuliert werden kann und trotzdem gut lesbar ist.

Die ORKA-Datenstrukturen bestehen hauptsächlich aus Mengen und Relationen. Die Daten, die im ORKA-Modell als Mengen repräsentiert werden, sind in Python Listen, und die ORKA-Relationen werden als Listen von Tupeln repräsentiert. Die Entscheidung, genau diese Repräsentation zu wählen, fiel dadurch, dass es einfach sein sollte, aus einer Relation die gewünschten Daten zu extrahieren.

Dieses Extrahieren wird von einer Funktion *R.lookup(x)* erledigt. Diese Funktion sucht in einer zweistelligen Relation *R* alle Paare (*a, b*), bei denen „*a* zu *x* passt“. Von allen diesen Paaren wird jeweils das *b* zurückgegeben. Ob „*a* zu *x* passt“, hängt vom Typ von *x* ab:

- Wenn x eine Liste ist, passt a zu x , falls a zu einem beliebigen Element von x passt. Dadurch ist es möglich, das Ergebnis eines Aufrufs direkt als Eingabe eines nächsten Aufrufs zu verwenden.
- Wenn x ein Tupel ist, passt a zu x , falls a auch ein Tupel ist und sie komponentenweise passen. Dieser Fall wird insbesondere für Berechtigungen benötigt, da diese aus Aktionen und Objekten zusammengesetzt sind.
- In allen anderen Fällen passt a zu x , wenn sie gleich sind.

Zum besseren Verständnis sind im folgenden ein paar Beispiele für die Verwendung der Funktion angegeben.

Beispiel: R sei eine Relation mit den Elementen $(1, 1)$, $(4, 2)$, $(4, -2)$, $(9, 3)$ und $(9, -3)$. Auf dieser Relation ergeben sich folgende Aufrufe:

$R.lookup(\dots)$	Ergebnis
1	[1]
2	[]
4	[2, -2]
[]	[]
[1, 4]	[1, 2, -2]

3.3 RBAC-Core

Das Modul RBAC-Core ist die Grundlage für alle weiteren Module. Es definiert die für RBAC grundlegenden Daten: Benutzer, Subjekte, Rollen und Berechtigungen, sowie drei Relationen, um die Daten miteinander zu verbinden. Die anderen Module bauen auf diesem Modul auf und können auf dessen Daten uneingeschränkt zugreifen. Das ist insbesondere bei der Zugriffsfunktion wichtig.

3.3.1 Datentypen

Es gibt die folgenden Datentypen, deren Exemplare wohlunterscheidbar sind, die aber ansonsten keine weiteren Eigenschaften haben:

- *USER*, die Menge aller möglichen Benutzer,
- *ROLE*, die Menge aller möglichen Rollen,
- *SUBJECT*, die Menge aller möglichen Subjekte,
- *OBJECT*, die Menge aller möglichen zu schützenden Objekte,
- *OPERATION*, die Menge aller möglichen Aktionen, die ein Subjekt auf den Objekten ausführen könnte.

Der Datentyp $PERMISSION := OBJECT \times OPERATION$ fasst Objekte und die zugehörigen Operationen zusammen, da die Operationen oft an bestimmte Objekte gebunden sind und gerade diese Kombination bei der Zugriffsentscheidung eine große Rolle spielt.

3.3.2 Persistente Daten

Aufbauend auf diesen Datenstrukturen definiert das Modul RBAC-Core die folgenden Relationen:

- $U \subset USER$, die Menge der Benutzer, die dem System bekannt sind,
- $R \subset ROLE$, die Menge der Rollen, die dem System bekannt sind,
- $P \subset PERMISSION$, die Menge der Berechtigungen, die dem System bekannt sind,
- $UA \subset U \times R$ (User Assignment), die Zuweisung von Rollen an Benutzer,
- $PA \subset R \times P$ (Permission Assignment), die Zuweisung von Berechtigungen an Rollen.

3.3.3 Schnelllebige Daten

Die folgenden Variablen sind schnelllebiger und werden daher nicht persistent gespeichert:

- $S \subset SUBJECT$, die Menge der Subjekte, die dem System bekannt sind,
- $SR \subset S \times R$ (Subject Roles), die Zuordnung von Rollen zu Subjekten. Beim Anmelden am System werden alle Rollen des Benutzers übernommen, der sich gerade anmeldet. Danach können Rollen gezielt deaktiviert und wieder aktiviert werden,
- $SU: S \rightarrow U$ (Subject User), die Zuordnung der Subjekte zu dem Benutzer, aus dem sie jeweils entstanden sind.

3.3.4 Zugriffsentscheidung

Um zu prüfen, ob ein Zugriff eines Subjekts auf ein Objekt erlaubt werden soll, wird die folgende Funktion ausgewertet:

```
def checkAccess(s, op, ob):  
    roles = SR.lookup(s)  
    perms = PA.lookup(roles)  
    return (op, ob) in perms
```

Um zu prüfen, ob ein Zugriff erlaubt ist, werden zunächst all diejenigen Rollen bestimmt, die für das anfragende Subjekt gerade aktiviert sind. Zu all diesen Rollen werden dann die Berechtigungen bestimmt. Wenn die angefragte Berechtigung darunter ist, wird der Zugriff gewährt, sonst nicht.

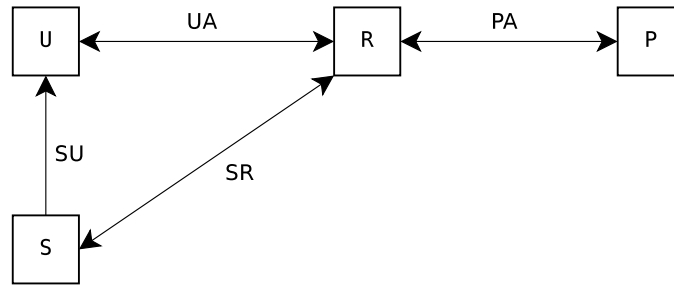


Abbildung 3.1: Die Daten aus RBAC-Core im Zusammenhang

3.3.5 Zusammenfassung

Die in diesem Modul eingeführten Daten sind in Abbildung 3.1 dargestellt. Die eingerahmten Variablen stellen die Entitäten dar, alle anderen Variablen sind Relationen zwischen den Entitäten. Die Entitäten sind die Benutzer (U), die Subjekte (S), die Rollen (R) und die Berechtigungen (P). Die Relationen zwischen ihnen sind User Assignment (UA), Permission Assignment (PA), Subject User (SU) und Subject Roles (SR).

In der Abbildung wird nochmal deutlich, dass Benutzer (U) und Subjekte (S) sehr ähnlich sind (beide sind mit den Rollen verbunden), und dass sie keine direkte Verbindung zu den Berechtigungen P haben.

3.4 RBAC mit Rollenhierarchie

3.4.1 Datentypen

Dieses Modul definiert keine neuen Datentypen.

3.4.2 Persistente Daten

Dieses Modul erweitert RBAC-Core um eine persistente Relation:

- $RH \subset R \times R$ (Role Hierarchy), die Rollenhierarchie; wenn zwei Rollen (r_1, r_2) in dieser Relation stehen, heißt das, dass alle Berechtigungen, die r_2 hat, automatisch an r_1 weitergegeben werden. r_2 ist also die „kleinere“ Rolle mit weniger oder gleich vielen Rechten wie r_1 .

3.4.3 Schnelllebige Daten

Dieses Modul definiert keine schnelllebigen Daten.

3.4.4 Zugriffsentscheidung

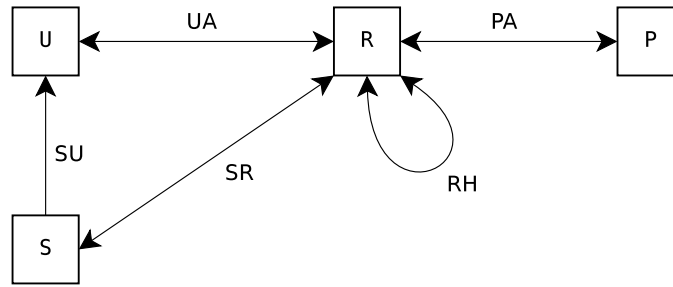


Abbildung 3.2: Die Daten aus RBAC mit Rollenhierarchie im Zusammenhang

```

def checkAccess(s, op, ob):
    roles = SR.lookup(s)
    allRoles = RH.lookupReflexivelyAndTransitively(roles)
    perms = PA.lookup(allRoles)
    return (op, ob) in perms

```

Die Funktion unterscheidet sich von der in RBAC-Core dadurch, dass die Rollen, die dem Subjekt zugewiesen sind, nicht mehr nur direkt aus der Relation *SR* kommen, sondern zusätzlich aus *RH*.

Durch diese Änderung kann ein Subjekt unter Umständen Operationen durchführen, die es unter reinem RBAC nicht kann. Eine derartige Berechtigungserweiterung sollte daher sehr kritisch betrachtet werden. Eine mögliche Alternative, die diese Änderung umgeht, wäre dem Subjekt bei der Anmeldung all seine Rollen und die untergeordneten Rollen zuzuweisen und die Funktion zur Zugriffsentscheidung unverändert zu lassen. Zusätzlich müssten auch die Verfahren zum Aktivieren und Deaktivieren einzelner Rollen geändert werden. Das ORKA-Projekt hat sich jedoch für die hier dargestellte Variante entschieden.

3.4.5 Zusammenfassung

Abbildung 3.2 zeigt, an welcher Stelle die Relation *RH* ergänzt wurde. Der restliche Aufbau bleibt gegenüber RBAC-Core unverändert.

3.5 Context Constraints

In vielen Fällen hängt eine Zugriffsentscheidung nicht nur von dem anfragenden Subjekt, der Operation und dem Objekt ab, sondern auch von „komplizierteren Umständen im Umfeld“, wie beispielsweise der aktuellen Tageszeit. Aus diesem Grund gibt es in ORKA sogenannte „Context Constraints“ (CC), die genau solche Einschränkungen an die Relationen *P*, *PA* und *R* im Modul RBAC-Core knüpfen. Dadurch kann beispielsweise

eine Rolle bestimmte Operationen nur zu bestimmten Uhrzeiten oder an bestimmten Wochentagen ausführen.

Ein CC ist eine boolesche Funktion und eine Reihe von Argumenten (in ORKA „Parameter“ genannt). Wenn die Funktion `false` zurückgibt, ist der CC nicht erfüllt, und der Zugriff wird verweigert. Jeder der Parameter kann entweder ein Literal sein (346, "September", 08:00) oder von einem sogenannten Context Provider zur Auswertungszeit bereitgestellt werden.

Im Gegensatz zu vielen Programmiersprachen können Funktionsaufrufe in ORKA nicht geschachtelt werden; es sind also nur Einschränkungen möglich, die sich auf genau einen Funktionsaufruf abbilden lassen.

3.5.1 Datentypen

Dieses Modul definiert drei neue Datentypen ($\mathbb{P} R$ bezeichnet die Potenzmenge von R):

- *ContextFunctionId*, ein Namensraum für Funktionsnamen,
- *ContextFunctionParameter*, ein Parameter eines Funktionsaufrufs,
- *ContextConstraint* $\equiv$ (*ContextFunctionId*, $\mathbb{P}$ *ContextFunctionParameter*), ein vollständiger CC, bestehend aus Funktions-ID und beliebig vielen Parametern.

Vordefinierte Kontext-Funktionen

Die von ORKA vordefinierten¹ Funktionen sind:

- *equal(left, right)*, um zwei Werte des gleichen Datentyps auf Gleichheit zu prüfen.
- *not-equal(left, right)*, um zwei Werte des gleichen Datentyps auf Ungleichheit zu prüfen.
- *more-than(left, right)*, um zu testen, ob die Zahl *left* größer ist als *right*.
- *less-than(left, right)*, um zu testen, ob die Zahl *left* kleiner ist als *right*.
- *equal-or-more-than(left, right)*, um zu testen, ob die Zahl *left* mindestens so groß ist wie *right*.
- *equal-or-less-than(left, right)*, um zu testen, ob die Zahl *left* höchstens so groß ist wie *right*.
- *in-between(a, b, c)*², um zu prüfen, ob *a* in dem Intervall (*b*; *c*) liegt. Insbesondere darf *a* nicht den gleichen Wert haben wie *b* oder *c*.

¹Es gibt im ORKA-Projekt keine definitive Quelle für die gültigen Funktionsnamen, aber die folgenden sind mir beim Verfassen der Arbeit begegnet.

²Die Namen der Parameter waren zu dem Zeitpunkt, als diese Arbeit fertiggestellt wurde, noch nicht festgelegt.

3 ORKA und OPL

- *time-between*(*time*, *begin*, *end*), um zu prüfen, ob der Zeitpunkt *time* innerhalb der Zeitpunkte *begin* und *end* (einschließlich) liegt.

Der ORKA-Datentyp *ContextFunctionParameter* (abgekürzt: *CFParam*) ist ein Tripel (*key*, *type*, *value*). *key* ist der Name des Parameters, um ihn von den anderen Parametern zu unterscheiden, *type* ist der Datentyp des Parameters, und *value* ist der Wert des Parameters, in der Praxis als String codiert.

Der Datentyp eines Parameters kann einer der folgenden vier elementaren Datentypen sein:

- *string*, eine Folge von Unicode-Zeichen.
- *int*, eine ganze Zahl. Es ist jedoch nicht spezifiziert, ob es sich dabei um einen beschränkten Typ (etwa XMLSchema:int oder XMLSchema:long) handelt oder um einen unbeschränkten (wie XMLSchema:integer)). In der aktuellen Implementierung ist der Typ auf 32 Bit beschränkt.
- *date*, ein Datum, das gemäß [ISO-8601] codiert ist.
- *time*, ein Zeitpunkt, der gemäß [ISO-8601] codiert ist.

Zusätzlich gibt es die Möglichkeit, einem Parameter den „Datentyp“ *context* zuzuweisen, obwohl das nicht wirklich ein Datentyp ist, sondern eher eine Verarbeitungsanweisung. In diesem Fall ist der Wert des Parameters nicht direkt (als Literal) gespeichert, sondern wird vor der Verarbeitung noch von einem sogenannten „Context Provider“ ausgewertet. Ein typischer Anwendungsfall ist, dass der String `"wms.logon.subject"` dem Context Provider „wms“ (für Workflow Management System) übergeben wird. Dieser wertet den verbleibenden Ausdruck (`"logon.subject"`) aus und gibt die ID des gerade aktiven Subjekts zurück.

3.5.2 Persistente Daten

Passend zu den Datentypen beinhaltet dieses Modul eine Menge von CCs (in der Relation *CC*) und verbindet sie über drei weitere Relationen (*RCC*, *PACC*, *PCC*) mit Elementen aus RBAC-Core.

- $CC \subset \text{ContextConstraint}$, die Menge der CCs, die dem System bekannt sind. Über die drei folgenden Relationen werden die CCs dann mit den einzelnen Rollen bzw. Berechtigungen verknüpft.
- $RCC \subset R \times CC$ (Role Context Constraint), die Zuordnung von CCs an bestimmte Rollen. Eine Rolle darf nur aktiviert werden, wenn alle zugehörigen CCs erfüllt sind. Zusätzlich wird bei jedem Zugriff noch überprüft, ob der CC erfüllt ist. Ansonsten könnte ein CC „nur zwischen 10 und 12 Uhr“ nicht seine volle Wirkung entfalten, da er nur beim Aktivieren einer Rolle, nicht aber direkt beim Zugriff geprüft würde.

- $PACC \subset PA \times CC$ (Permission Assignment Context Constraint), die Zuordnung von CCs an Rollen-Berechtigungen. Eine Rolle hat die entsprechende Berechtigung nur, wenn alle zugehörigen CCs erfüllt sind.
- $PCC \subset P \times CC$ (Permission Context Constraint), die Zuordnung von CCs an Berechtigungen. Eine Berechtigung kann (unabhängig von der Rolle) nur dann genutzt werden, wenn alle zugehörigen CCs erfüllt sind.

Die Relation RCC kann dazu benutzt werden, eine bestimmte Rolle mit all ihren Berechtigungen einzuschränken, die Relation PCC schränkt die Ausführung einer bestimmten Berechtigung ein, unabhängig von der Rolle. Die Relation $PACC$ dagegen beschränkt eine Berechtigung nur dann, wenn sie von einer bestimmten Rolle durchgeführt wird.

Beispiele

In der folgenden Tabelle sind einige CC-Parameter beispielhaft aufgelistet.

Name	Datentyp	Wert
left	string	"Hallo"
right	int	34783734
begin	time	08:00
left	context	logon.subject
right	context	wfms_context_provider1.getTask(parameters.taskinstanceid)

Beispiel für einen vollständigen CC

Ein vollständiger CC, um bestimmte Zugriffe auf die Arbeitszeit einzuschränken, sieht in OPL/XML so aus:

```
<context_constraint cc_id="Arbeitszeit">
  <context_function_id id="time-between"/>
  <context_function_parameters>
    <parameter key="begin" type="time" value="08:00"/>
    <parameter key="time" type="context" value="env.current_time"/>
    <parameter key="end" type="time" value="19:00"/>
  </context_function_parameters>
</context_constraint>
```

In Worten: Der CC wird ausgewertet, indem die Funktion *time-between* mit drei Parametern aufgerufen wird: Zwei davon (*begin* und *end*) sind Literale, deren Wert und Datentyp direkt angegeben ist. Der Wert des dritten Parameters (*time*) wird erst zur Auswertungszeit ermittelt. Für die Auswertung ist der Context Provider „env“ zuständig. Der CC ist erfüllt, wenn der Context Provider „env“ die aktuelle Zeit liefern kann und diese zwischen 08:00 und 19:00 (einschließlich) liegt.

3.5.3 Schnelllebige Daten

Dieses Modul definiert keine schnelllebigen Daten.

3.5.4 Zugriffsentscheidung

Die Funktion zur Zugriffsentscheidung basiert auf der von RBAC-Core, verlangt aber zusätzlich, dass alle anwendbaren CCs erfüllt sind. Welche der CCs anwendbar sind, wird weiter unten erläutert.

```
def checkAccess(s, op, ob):  
    roles = SR.lookup(s)  
    perms = PA.lookup(roles)  
    return ((op, ob) in perms) \  
        and checkCCs(RCC.lookup(roles)) \  
        and checkCCs(PACC.lookup((roles, (op, ob)))) \  
        and checkCCs(PCC.lookup(perms))
```

Der erste Teil dieser Funktion entspricht der Funktion aus dem Modul RBAC-Core. Im zweiten Teil werden die Beschränkungen der anwendbaren CCs ausgewertet, und wenn einer dieser CCs nicht erfüllt ist, wird der Zugriff ebenfalls verweigert. Die anwendbaren CCs sind:

- Alle CCs aus der Relation *RCC*, deren Gegenstück eine gerade aktive Rolle ist,
- alle CCs aus der Relation *PACC*, deren Gegenstück (r, p) ist, wobei r eine gerade aktive Rolle ist und p die angeforderte Berechtigung,
- alle CCs aus der Relation *PCC*, deren Gegenstück p ist.

Durch die Entscheidung, dass relativ viele der CCs für eine bestimmte Anfrage anwendbar sind, entsteht ein Paradoxon: Es sind nämlich alle CCs anwendbar, die zu einer gerade aktiven Rolle gehören, auch wenn diese Rolle gar nicht an der Zugriffsentscheidung beteiligt ist. In diesem Fall kann ein Subjekt durch das gezielte Deaktivieren und Reaktivieren von Rollen zusätzliche Berechtigungen bekommen. Wenn ein Zugriff verweigert wird, kann es also sinnvoll sein, den gleichen Zugriff mit weniger Rollen und damit (intuitiv) auch weniger Rechten nochmal zu probieren.

Dieses Paradoxon kann aufgelöst werden, indem bei mehreren möglichen Wegen, zu einer Berechtigung zu kommen, nicht mehr alle CCs für alle Wege erfüllt sein müssen, sondern nur alle CCs für mindestens einen Weg. Dadurch ergeben sich vordergründig mehr Berechtigungen für die Subjekte, aber in Wirklichkeit sind diese Berechtigungen auch im restriktiveren Modell da, dort sind sie nur nicht so offensichtlich.

3.5.5 Zusammenfassung

Abbildung 3.3 stellt die Neuerungen des Moduls „Context Constraints“ im Zusammenhang dar: Die Menge der Context Constraints (*CC*) wurde neu eingeführt. Sie beeinflusst die

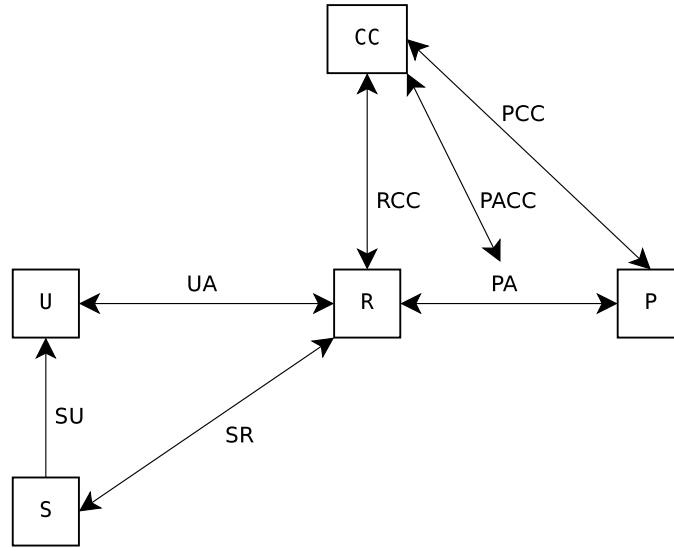


Abbildung 3.3: Die Daten aus Context Constraints im Zusammenhang

Zugriffsentscheidung an drei Stellen: Mit *RCC* werden den Rollen zusätzliche Bedingungen auferlegt, mit *PACC* wird die Relation *PA* eingeschränkt, und mit *PCC* bekommen auch die Berechtigungen zusätzliche Bedingungen. In der Abbildung nicht enthalten sind die einzelnen Teile eines Context Constraints (die Funktion und die Parameter), da sie für die Zugriffsentscheidung keine direkte Rolle spielen, sondern nur im Zusammenhang mit einem bestimmten CC.

3.6 Separation of Duty

Das ORKA-Modul für Separation of Duty fasst vier verschiedene Arten der Rechte-trennung in einem Modul zusammen. Die Statische Rechtetrennung wurde bereits in Abschnitt 2.4.3 beschrieben, die Dynamische Rechtetrennung in 2.4.4. Die anderen beiden in diesem Modul definierten Arten sind noch nicht beschrieben worden.

3.6.1 Datentypen

Dieses Modul definiert keine neuen Datentypen.

3.6.2 Persistente Daten

Das Modul RBAC-Core wird um vier Relationen erweitert:

- **SSoD** $\subset \mathbb{P}R \times \mathbb{N}$ (Static Separation of Duty) ordnet einigen Rollenmengen (*rs*) jeweils eine natürliche Zahl (*n*) zu. Ein Benutzer darf aus jeder dieser Rollenmengen

nur maximal n Rollen zugewiesen sein. In der Praxis ist n oft 2, um zwei Rollen zu definieren, die sich gegenseitig ausschließen.

- **DSoD** $\subset \mathbb{P}R \times \mathbb{N}$ (Dynamic Separation of Duty) ist nicht ganz so streng wie *SSoD*. Ein Benutzer kann mehr als n kritischen Rollen zugewiesen sein, aber alle Subjekte, die diesem Benutzer zugeordnet sind, dürfen insgesamt nicht mehr als n Rollen aus rs annehmen oder bereits angenommen haben. Um das zu kontrollieren, werden alle Rollen, die ein Subjekt bisher angenommen hat, gespeichert.
- So wie *SSoD* die Relation *UA* (Benutzer $\leftrightarrow$ Rolle) beschränkt, tut **SSoDP** $\subset \mathbb{P}P \times \mathbb{N}$ (Static Separation of Duty for Permissions) das für die Relation *PA* (Rolle $\leftrightarrow$ Berechtigung). Jedes Element (ps, n) dieser Relation legt fest, dass eine Rolle nur maximal n der Berechtigungen aus ps haben darf. Es kann jedoch immer noch passieren, dass ein Subjekt mehrere dieser Rollen hat und dadurch doch mehr kritische Berechtigungen hat als beabsichtigt.

*Beispiel: Die drei Berechtigungen p_1, p_2 und p_3 werden als kritisch angesehen. Jede Rolle darf deshalb nur mit maximal einer dieser Berechtigungen verbunden sein. Es gibt drei Rollen r_1, r_2 und r_3 , wobei r_i jeweils mit p_i verbunden ist. Die Bedingungen von *SSoDP* sind damit alle erfüllt. Es kann aber trotzdem passieren, dass ein Subjekt alle drei Berechtigungen gleichzeitig hat. Das ist der Fall, wenn es allen drei Rollen zugeordnet ist. *SSoDP* verhindert also nur, dass eine einzelne Rolle zu viele der kritischen Berechtigungen hat; um zu verhindern, dass ein Subjekt zu viele der kritischen Berechtigungen hat, muss *SSoDP* mit *SSoD* kombiniert werden.*

- Die Relation **SSSoD** $\subset \mathbb{P}R \times \mathbb{N}$ (Strict Static Separation of Duty) entspricht *SSoD*, fordert aber, dass die Berechtigungen aller Rollen in einem rs disjunkt sind.

Durch die explizite Trennung von *DSoD* und *SSoD* wird insbesondere der Unterschied zwischen Benutzern und Subjekten deutlich gemacht, der in einfacheren Zugriffskontrollmodellen nur teilweise von Bedeutung ist.

3.6.3 Schnelllebige Daten

Für die Beschränkung *DSoD* muss festgehalten werden, welche Rollen ein Subjekt im Laufe seines Lebens eingenommen hat. Das wird in der Relation $History_{SR} \subset S \times R$ getan.

3.6.4 Zugriffsentscheidung

Die Zugriffsentscheidung ist gegenüber „RBAC-Core“ unverändert. Die zusätzlichen Einschränkungen bezüglich der Rollen und Berechtigungen haben keinen Einfluss auf die Zugriffsfunktion, sondern werden an anderen Stellen des ORKA-Systems geprüft. Diese Stellen sind das Anmelden am System (wenn die maximal möglichen Rollen eines

Benutzers an ein Subjekt weitergegeben werden) und das Aktivieren einer Rolle für ein Subjekt.

3.7 Separation of Duty mit Rollenhierarchie

Dieses Modul basiert auf dem Modul „RBAC mit Rollenhierarchie“ und ergänzt es um zwei der Relationen, die bereits im Modul „Separation of Duty“ beschrieben wurden.

3.7.1 Datentypen

Dieses Modul definiert keine neuen Datentypen.

3.7.2 Persistente Daten

- **SSoD** $\subset \mathbb{P} R \times \mathbb{N}$ entspricht der Relation *SSoD* im Modul „Separation of Duty“, nur dass hier auch die vererbten Rollen miteinbezogen werden.
- **DSoD** $\subset \mathbb{P} R \times \mathbb{N}$ entspricht der Relation *DSoD* im Modul „Separation of Duty“, nur dass hier auch die vererbten Rollen miteinbezogen werden und zusätzlich nur die gerade aktiven Rollen geprüft werden. Es gibt hier also im Gegensatz zu „Separation of Duty“ keine History.

3.7.3 Schnelllebige Daten

Dieses Modul definiert keine schnelllebigen Daten.

3.7.4 Zugriffentscheidung

Die Zugriffentscheidung ist gegenüber „RBAC mit Rollenhierarchie“ unverändert.

3.8 Weitere Module

Im ORKA-Projekt sind noch weitere Module definiert, die in dieser Arbeit aber nicht behandelt werden. Einen großen Teil davon machen Module für Geschäftsprozesse aus.

3.9 Kombination der Zugriffentscheidungen

Die in den vorigen Abschnitten vorgestellten Module kommen jeweils für sich zu einer Zugriffentscheidung. Diese einzelnen Entscheidungen müssen jetzt noch zu einer Gesamtentscheidung zusammengefasst werden. In ORKA wird dazu der Algorithmus **deny-overrides** verwendet. Dieser Algorithmus verweigert den Zugriff, sobald eines der Module den Zugriff verweigert. Nur, wenn alle Module den Zugriff erlauben, wird er auch insgesamt erlaubt. Dass diese einfache Festlegung zu Problemen führt, wird in Abschnitt 5.12 erläutert.

3.10 Zusammenfassung

In diesem Kapitel wurde das Zugriffskontrollmodell von ORKA vorgestellt. Das Modell basiert auf einzelnen Modulen, die jeweils unabhängig voneinander zu einer Zugriffsentscheidung kommen. Die einzelnen Entscheidungen werden dann zu einer Gesamtentscheidung kombiniert.

Bei der Beschreibung der einzelnen Module wurden besonders diejenigen Details, die für die Implementierung der Zugriffskontrolle wichtig sind, erwähnt. Details, die die administrative Seite betreffen, beispielsweise das Aktivieren einer Rolle für ein Subjekt oder das Löschen eines Benutzers, wurden bewusst ausgelassen, da sie für die Übersetzung der ORKA-Policy in XACML keine Rolle spielen.

In diesem Kapitel wurden die folgenden ORKA-Module im Detail vorgestellt: Das Modul „RBAC-Core“ definiert die grundlegenden Daten, die für ein RBAC-System notwendig sind: Benutzer, Subjekte, Rollen und Berechtigungen. Darauf aufbauend führt das Modul „RBAC mit Rollenhierarchie“ eine Hierarchie unter den Rollen ein, so dass rollenübergreifende Berechtigungen zusammengefasst werden können und nicht jeder Rolle einzeln zugewiesen werden müssen. Das Modul „Context Constraints“ schränkt Zugriffe abhängig von externen Bedingungen ein. Die Module „Separation of Duty“ und „Separation of Duty mit Rollenhierarchie“ schränken die möglichen Rollen eines Subjekts bzw. die möglichen Berechtigungen einer Rolle ein. All diese Module kommen jeweils unabhängig voneinander zu einer Zugriffsentscheidung. Die Kombination der einzelnen Entscheidungen zu einer Gesamtentscheidung wird angerissen und in einem späteren Teil der Arbeit ausführlicher behandelt.

Ein umfangreiches Beispiel für eine in OPL/XML formulierte ORKA-Policy befindet sich in Anhang A.1.

Das nächste Kapitel beschäftigt sich mit XACML, der Zielsprache der Übersetzung. In Kapitel 5 wird dann beschrieben, wie die in diesem Kapitel vorgestellte ORKA-Policy in eine äquivalente XACML-Policy übersetzt wird.

4 XACML

Nachdem im letzten Kapitel die Quellsprache der Übersetzung (OPL) vorgestellt wurde, folgt nun die Zielsprache (XACML). Die „eXtensible Access Control Markup Language“ (XACML) ist eine XML-basierte Sprache zur Zugriffskontrolle. Im Jahre 2003 wurde XACML 1.0 von der „Organization for the Advancement of Structured Information Standards“ (OASIS) normiert, Version 2.0 folgte zwei Jahre später, Version 3.0 ist in Vorbereitung. Da das ORKA-Projekt sich für eine Implementierung von XACML 1.0 entschieden hat, wird hauptsächlich diese Version vorgestellt. Auf die Änderungen hin zu 2.0 und 3.0 wird teilweise eingegangen, um die Weiterentwicklung von XACML aufzuzeigen.

4.1 Übersicht

XACML definiert ein System zur Zugriffskontrolle, das so flexibel ist, dass es sich gut in bestehende IT-Systeme zur Zugriffskontrolle integrieren lässt. Ein XACML-System besteht aus fünf zentralen Komponenten (PAP, PEP, Context Handler, PDP, PIP), die zusammengenommen die Zugriffskontrolle realisieren. Die einzelnen Komponenten werden im nächsten Abschnitt vorgestellt und in Verbindung gebracht.

4.1.1 Begriffe

Ein **PDP** (engl. **Policy Decision Point**) ist dasjenige Teilsystem der Zugriffskontrolle, das eine Entscheidung trifft, ob ein bestimmter Zugriff erlaubt werden soll oder nicht. Der PDP hat nicht die Aufgabe, Ressourcen vor unberechtigtem Zugriff zu schützen, sondern ist alleine dafür zuständig, Entscheidungen zu treffen. Er bekommt Anfragen vom (vertrauenswürdigen) PEP und wertet sie aus. Wenn er dafür zusätzliche Informationen braucht, fragt er den PIP danach. Das Ergebnis der Auswertung („erlaubt“, „verboten“, „nicht anwendbar“, „Fehler“) wird dann an den PEP zurückgeschickt. Zusätzlich kann der PDP dem PEP bestimmte Verpflichtungen (Obligations) auferlegen, die der PEP einhalten muss. Typische Beispiele sind das Mitloggen von erlaubten und unerlaubten Zugriffen.

Ein **PEP** (engl. **Policy Enforcement Point**) ist dasjenige Teilsystem der Zugriffskontrolle, das schützend zwischen den Subjekten und den Ressourcen steht. Wenn ein Zugriffswunsch bei ihm eingeht, fragt er den PDP, ob der Zugriff gewährt werden soll oder nicht, und handelt dementsprechend. Für den PEP ist es irrelevant, *warum* ein Zugriff erlaubt ist oder nicht, denn die Entscheidung darüber wird ganz alleine vom PDP gefällt. Der PEP sorgt nur dafür, dass die Entscheidung des PDP auch durchgesetzt wird.

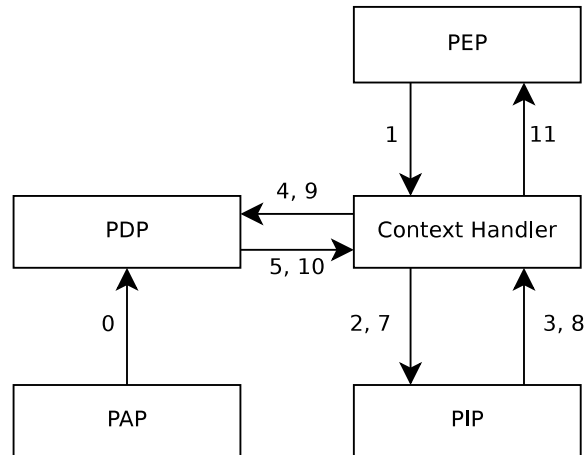


Abbildung 4.1: Kontrollfluss in XACML (vereinfacht)

Ein **PAP** (engl. **Policy Administration Point**) ist das Teilsystem der Zugriffskontrolle, in dem die Regeln, nach denen Zugriffe erlaubt oder verboten sind, festgelegt werden. Damit die Regeln später angewendet werden, müssen sie erst zum PDP übertragen werden. Der PAP kann zum Beispiel eine Datenbank sein, in der die Zugriffsregeln gespeichert werden, oder auch eine graphische Benutzeroberfläche zum Bearbeiten der Regeln.

Wenn der PDP über eine Anfrage entscheidet, aber nicht genug Informationen hat, um die Anfrage zu beantworten, fragt er den **Policy Information Point (PIP)** danach.

Beispiel: In einer Anfrage an den PDP sind üblicherweise nur Subjekt, Aktion und Objekt angegeben, nicht jedoch die Rollen des Subjekts. Die Policies verwenden jedoch üblicherweise Rollen, um zur Zugriffsentscheidung zu kommen. Daher muss der PDP irgendwie an die Information kommen, welche Rollen dem Subjekt zugewiesen sind. Dazu fragt er den PIP (vereinfacht): „rollen(Anfrage)?“, der PIP sucht aus der Anfrage das Subjekt heraus, schaut in einer Datenbank (oder einer anderen Quelle) nach, welche Rollen zu diesem Subjekt gehören und antwortet dann: „(Rolle1, Rolle2, Rolle3)“.

4.1.2 Ablauf einer Zugriffsanfrage

XACML legt fest, welche Aufgaben jede der einzelnen Komponenten hat und durch welche Arten von Nachrichten sie kommunizieren. Die Nachrichten (Anfragen, Antworten) haben ein festes Format, das auch spezifiziert wird. Schließlich wird auch der Ablauf, wie die Anfragen vom PEP ausgehend in das System gelangen, dort evaluiert werden und wie schließlich die Antwort wieder zum PEP gelangt, festgelegt. Dieser Ablauf ist in Abbildung 4.1 dargestellt.

Die Policies werden im Policy Administration Point (PAP) von menschlichen Administratoren oder auch automatisch erzeugt. Von dort werden sie dem Policy Decision Point

(PDP) verfügbar gemacht (Schritt 0). Dieser Vorgang geschieht unabhängig von einer konkreten Zugriffsanfrage.

Sobald die relevanten Policies dem PDP bekannt sind, kann das XACML-System Zugriffsanfragen beantworten. Das geschieht folgendermaßen:

- Schritt 1: Der PEP ist in die jeweilige Anwendung integriert und bekommt dort eine Zugriffsanfrage. Diese Anfrage schickt er im anwendungsspezifischen Format an den Context Handler.
- Schritt 2: Der Context Handler wandelt die Anfrage aus dem anwendungsspezifischen Format ins XACML-Format um. Er kann dabei fehlende Attribute vom Policy Information Point (PIP) anfragen und in die XACML-Anfrage einbinden. Zu diesem Zeitpunkt können jedoch noch nicht alle fehlenden Attribute bestimmt werden. Der PDP bekommt daher später nochmal die Möglichkeit, fehlende Attribute anzufordern.
- Schritt 3: Der PIP wertet die Attribute aus und gibt deren Werte zurück.
- Schritt 4: Nachdem die XACML-Anfrage um die fehlenden Attribute ergänzt wurde, wird sie an den PDP geschickt, um die Zugriffsentscheidung zu erhalten.
- Schritt 5: Der PDP sucht die Policy heraus, die für diese Anfrage zuständig ist. Er interpretiert die Policy und versucht, die Regeln darin mit der Anfrage abzugleichen.
- Schritt 6: Falls der PDP beim Interpretieren auf ein unbekanntes Attribut trifft, schickt er dieses Attribut an den Context Handler.
- Schritt 7: Der Context Handler leitet das fehlende Attribut an den PIP weiter. Der Zwischenschritt über den Context Handler ist sinnvoll, da es mehrere PIPs geben kann und der PDP so trotzdem immer den gleichen Ansprechpartner hat.
- Schritt 8: Der PIP schickt das aufgelöste Attribut an den Context Handler.
- Schritt 9: Der Context Handler schickt das Attribut an den PDP weiter.
- Schritt 10: Wenn die Policy vollständig ausgewertet ist, schickt der PDP das Ergebnis an den Context Handler zurück.
- Schritt 11: Der Context Handler wandelt das Ergebnis in das anwendungsspezifische Format um und schickt es an den PEP. Abhängig von der Entscheidung gewährt der PEP dann den Zugriff, oder er verweigert ihn.

Dieses Modell trennt ganz klar die verschiedenen Aufgaben innerhalb der Zugriffskontrolle: Der PEP schützt die Ressourcen vor unberechtigtem Zugriff, kann aber die Entscheidung, ob ein Zugriff erlaubt werden soll, nicht selbst treffen. Der Context Handler ist der Mittelpunkt des Geschehens und vermittelt zwischen den restlichen Komponenten.

Der PDP trifft die Zugriffsentscheidung, hat aber eventuell nicht alle Informationen, die für die Entscheidung notwendig sind. Der PIP liefert dem PDP die zur Entscheidung notwendigen Informationen aus einer anwendungsspezifischen Datenquelle.

Durch diesen Aufbau sind der PDP und der PAP vollständig unabhängig von der Anwendung. Die Komponenten, die jeweils angepasst werden müssen, sind der PEP, der Context Handler und der PIP.

4.2 Datenstrukturen

XACML definiert nicht nur die oben angesprochenen Komponenten, sondern auch die Nachrichtenformate, mit denen die Komponenten kommunizieren sowie den Aufbau der Policies. Es gibt zwei Nachrichtenformate: Die Anfragen werden in Abschnitt 4.4 behandelt, die Antworten in Abschnitt 4.14. Den größten Teil der Spezifikation nimmt jedoch die Beschreibung der Policies ein, die hier in Abschnitt 4.7 erläutert werden.

4.3 Abkürzende Notation für XACML

Auf den ersten Blick sehen Datenstrukturen, die in XACML formuliert sind, sehr kompliziert und unübersichtlich aus. Daher werden in dieser Arbeit die XACML-Listings nicht wirklich in XACML ausgeschrieben, sondern in einer abkürzenden Variante (sXACML). Die abgekürzten Listings werden trotzdem als XACML bezeichnet, da die Semantik identisch ist.

Das genaue Verfahren, nach dem die Abkürzung funktioniert, benötigt zur Beschreibung viele Details, die erst später erläutert werden. Es ist nicht nötig, die Details der Abkürzung zu verstehen, um die Listings in dieser Arbeit zu lesen.

Die Gründe für die ausführliche Schreibweise von XACML-Dokumenten sind:

- XACML definiert Datenstrukturen, die sich sehr nahe an der Implementierung orientieren und weniger an höheren Konzepten, die näher an der Anwendungsebene sind. So müssen beim Erstellen einer Policy stets alle strukturellen Ebenen aufgeschrieben werden, auch wenn sie jeweils nur ein Element enthalten. Im Grunde entsprechen die XACML-Datenstrukturen einem abstrakten Syntaxbaum (engl.: Abstract Syntax Tree (AST)), und die sind in fast allen Programmier- und sonstigen Sprachen aufwendiger zu lesen als der Quelltext.
- Um die Implementierung zu vereinfachen, enthält eine XACML-Policy relativ viel Redundanz, insbesondere bei den Datentypen von verschachtelten Ausdrücken. Wäre das nicht so, müssten die Implementierungen die „fehlenden“ Informationen entweder per Typinferenz berechnen oder zur Laufzeit eine Fehlermeldung ausgeben. Die Redundanz dient bei XACML dazu, eventuelle Typfehler in den Policies möglichst früh zu erkennen.
- Durch die exzessive Nutzung von URIs (als Bezeichner für Attribute, Datentypen, Regeln, Policies) wirken XACML-Dokumente stark aufgebläht. Für das menschliche

Verständnis reicht jedoch meistens schon der letzte Teil eines URI aus, da der Rest nur einen Namensraum angibt und in einem bestimmten Anwendungsbereich viele URIs allein durch die Angabe dieses letzten Teils eindeutig bestimmbar sind. Ein typisches Beispiel ist `urn:oasis:names:tc:xacml:1.0:subject:subject-id`, das durch die Abkürzung zu `subject-id` wird. Die abgekürzten URIs sind syntaktisch keine URIs mehr (das „Schema“ fehlt), für menschliche Leser stellt das jedoch kein Problem dar.

- Die Spezifikation schreibt vor, dass die `<Target>`-Elemente in jedem Fall ausgeschrieben werden müssen, auch wenn sie an sich leer sind. In XACML 2.0 wurde diese Forderung dadurch etwas gelockert, dass die (semantisch unnötigen) Elemente `<anySubject>` auch syntaktisch verworfen wurden. Noch effizienter wird die Schriftform, wenn ein `<*Match>`-Element, das einzeln in einem `<Target>`-Abschnitt auftaucht, direkt in das `<Target>`-Element geschrieben werden kann und nicht die beiden zusätzlichen Ebenen (etwa `<Subjects>` und `<Subject>`) benötigt.

Um Papier und die Zeit des Lesers zu sparen, werden die meisten XACML-Fragmente in dieser Arbeit in einer verkürzten Schreibweise (sXACML) dargestellt. Gegenüber reinem XACML ergeben sich folgende Abkürzungen:

- Bei URIs aus den „üblichen“ Namensräumen (XACML-1.0, XACML-2.0) werden alle Komponenten bis auf die letzte weggelassen, da die Eindeutigkeit auch so gewährleistet ist. Bezeichner aus anderen Namensräumen werden vereinfacht in der Form „*namensraum:bezeichner*“ dargestellt, d. h. der Namensraum wird nicht komplett ausgeschrieben.
- Bei Attributen und beim Aufruf von Funktionen müssen eigentlich die Datentypen aller Parameter angegeben werden. In den Fällen, in denen der Datentyp durch die Benennung der Funktion oder durch sonstige einfache Typinferenz hergeleitet werden kann, wird dieses Attribut weggelassen.
- Befindet sich ein `<*Match>`-Element alleine in einem Abschnitt eines `<Target>`s, werden die Zwischenelemente (beispielsweise `<Subjects>` und `<Subject>`) weggelassen.

4.3.1 Beispiel für die Notation in sXACML

Ein XACML-Fragment, um ein Attribut aus einer Anfrage mit einem literalen Wert zu vergleichen, kann so aussehen:

```
<ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
  <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string"
    >read</AttributeValue>
  <ActionAttributeDesignator
    AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
    DataType="http://www.w3.org/2001/XMLSchema#string"/>
</ActionMatch>
```

(Anmerkung: Aufgrund der langen Zeilen in dieser Darstellung, und weil in XML Leerzeichen innerhalb von Elementen relevant sind, muss die schließende Klammer ()) direkt vor dem `read` stehen. Stünde sie in der Zeile darüber, würde der Attributwert die Leerzeichen beinhalten, wäre also "`\nUUUUread`".)

Das gleiche Fragment sieht in sXACML so aus:

```
<ActionMatch MatchId="string-equal">
  <AttributeValue>read</AttributeValue>
  <ActionAttributeDesignator AttributId="action-id"/>
</ActionMatch>
```

Durch diese Abkürzung geht keine Information verloren, da das sXACML-Fragment problemlos wieder zurückgewandelt werden kann. Der Vorteil von sXACML ist, dass die für die Betrachtung wesentlichen Elemente stärker hervorgehoben werden als das, was in der Spezifikation vorgeschrieben ist, aber für das menschliche Verständnis nicht notwendig ist.

4.4 Anfrage

Eine XACML-Anfrage (`<Request>`) besteht aus einer Reihe von Attributen, die in vier, manchmal auch mehr, Abschnitten gruppiert sind. Die vier Abschnitte heißen `<Subject>`, `<Action>`, `<Resource>` und `<Environment>`. Die ersten drei entsprechen den typischen Elementen einer Zugriffsanfrage: „*Wer darf was womit machen?*“ Alles, was keinem dieser drei Abschnitte zugeordnet werden kann, wird als „Umgebung“ der Anfrage bezeichnet.

Ein Beispiel für eine einfache XACML-Anfrage ist in Listing 4.1 dargestellt. Diese Anfrage enthält drei Attribute: ein Subjekt (Karl Schmidt), eine Aktion (`read`) und eine Ressource (346713662). Insgesamt stellt diese Beispielanfrage einen einfachen Lesezugriff auf eine medizinische Akte dar.

4.4.1 Mehrere Subjekte in einer Anfrage

Im Normalfall gehört zu einer Anfrage jeweils ein Subjekt, eine Aktion und eine Ressource. Es kann jedoch auch vorkommen, dass mehrere Subjekte an einer Anfrage beteiligt sind. Jedem dieser Subjekte wird dazu eine Subjekt-Kategorie zugeordnet, wobei es in jeder Kategorie nur ein Subjekt geben kann. Sollte es (syntaktisch) doch mehr als ein Subjekt pro Kategorie geben, werden die Attribute all dieser Subjekte zu einem gemeinsamen Subjekt (semantisch) zusammengemischt. Vordefinierte Subjekt-Kategorien sind:

access-subject: Das Subjekt, das Zugriff auf eine Ressource verlangt. Da dies der Normalfall ist, muss die Subjekt-Kategorie hier nicht explizit angegeben werden.

recipient-subject: Das Subjekt, das die Daten letztendlich empfängt.

intermediary-subject: Das Subjekt, das die Daten nur weiterleitet.

```

<Request>
  <Subject>
    <Attribute Attributeld="subject-id">
      <AttributeValue>Karl Schmidt</AttributeValue>
    </Attribute>
  </Subject>
  <Action>
    <Attribute Attributeld="action-id">
      <AttributeValue>read</AttributeValue>
    </Attribute>
  </Action>
  <Resource>
    <Attribute Attributeld="resource-id">
      <AttributeValue>file:///usr/local/medtool/records/346713662</AttributeValue>
    </Attribute>
  </Resource>
</Request>

```

Listing 4.1: Beispiel für eine XACML-Anfrage

codebase: Die Codebasis, die die Anfrage generiert. Was das genau bedeutet, ist in XACML nicht definiert, es sind nur Beispiele angegeben. So kann es sich hierbei um eine URL handeln, die ein Programmpaket identifiziert, oder auch einen Identifikator für den Herausgeber des Programms.

requesting-machine: Der Rechner, von dem aus der Zugriff angefordert wird. Auch hier gibt es mehrere Möglichkeiten, zum Beispiel den DNS-Namen, die IP-Adresse, den NetBIOS-Namen, eine IPsec-Identität, die MAC-Adresse. In XACML selbst ist nur IPsec erwähnt.

XACML sieht es vor, dass weitere Subjekt-Kategorien je nach Bedarf definiert werden können.

4.4.2 Beispielanfrage für mehrere Subjekte

Die Anfrage aus Listing 4.1 kann relativ einfach um ein zweites Subjekt erweitert werden. In Listing 4.2 ist das der Rechner, von dem aus der Zugriff erfolgt.

Die beiden Subjekte dieser Anfrage werden durch ihre Subjektkategorie unterschieden. Daher werden ihre Attribute nicht vermischt, sondern können vom PDP unterschieden werden.

```
<Request>
  <Subject SubjectCategory="access-subject">
    <Attribute Attributeld="subject-id">
      <AttributeValue>Karl Schmidt</AttributeValue>
    </Attribute>
  </Subject>
  <Subject SubjectCategory="requesting-machine">
    <Attribute Attributeld="subject-id">
      <AttributeValue>192.168.23.7</AttributeValue>
    </Attribute>
  </Subject>
  <Action>
    <Attribute Attributeld="action-id">
      <AttributeValue>read</AttributeValue>
    </Attribute>
  </Action>
  <Resource>
    <Attribute Attributeld="resource-id">
      <AttributeValue>file:///usr/local/medtool/records/346713662</AttributeValue>
    </Attribute>
  </Resource>
</Request>
```

Listing 4.2: Beispiel für eine XACML-Anfrage mit mehreren Subjekten

4.5 Aufbau der Attribute

Jedes Attribut (`<Attribute>`), das in einer XACML-Anfrage vorkommt, besteht aus drei Komponenten: Name, Datentyp und Wert(e).

- Der Name des Attributs (`AttributeId`) ist ein URI. XACML definiert einige oft benötigte Namen, zum Beispiel die Subjekt-ID, Aktions-ID, Ressourcen-ID. XACML ist jedoch nicht auf die vordefinierten Namen beschränkt; neue Namen können einfach verwendet werden, ohne dass sie vorher irgendwo deklariert werden müssen. Der Name allein sagt noch nichts über den Datentyp des Attributs aus, deshalb muss der explizit angegeben werden.
- Der Datentyp (`DataType`) ist auch ein URI. Die üblichen Werte hierfür sind die XMLSchema-Datentypen (`string`, `integer`, `anyURI`) sowie einige weitere. Diese Datentypen werden im nächsten Abschnitt beschrieben. XACML sieht es vor, dass eigene Datentypen definiert werden können, aber das erfordert einige Arbeit: So müssen zusätzlich zu dem Datentyp auch Funktionen (für die Auswertung innerhalb der Policies) definiert werden, die mit diesem Datentyp umgehen können, etwa Vergleichsfunktionen oder Umwandlungsfunktionen in andere Datentypen.
- Der Wert des Attributs ist eine Folge von `<AttributeValue>`-Tags (meistens aber nur eins), das jeweils beliebige Struktur haben kann. Für die vordefinierten Datentypen wird der Attributwert jeweils direkt als Text in das `<AttributeValue>`-Tag geschrieben, ohne weitere XML-Struktur.

4.5.1 Beispiele für Attribute

Beispiele für Attribute sind:

Name	Datentyp	Wert
subject-id	string	Julius
subject-id	integer	507
role	string	doctor
action-id	string	read
resource-id	anyURI	http://med.example.com/records/bart.xml
resource-id	string	/home/jhibbert/secrets.txt
current-time	time	09:42:00+02:00
current-time	time	07:42:00Z
current-date	date	2008-08-05

Diese Attribute kann man in vier Gruppen einordnen, je nachdem, welchen Teil der Anfrage sie beschreiben:

- Zum Subjekt gehören `subject-id` und `role`.
- Zur Aktion gehört `action-id`.

- Zur Ressource gehört `resource-id`.
- Zur Umgebung gehören `current-date` und `current-time`.

Subjekt, Aktion und Ressource sind die üblichen drei Dinge, aus denen eine Anfrage besteht. Alle Attribute, die sich nicht in eine dieser drei Gruppen einordnen lassen, werden zur Umgebung gerechnet.

4.6 Datentypen

XACML 1.0 definiert 14 Datentypen und erlaubt dazu noch benutzerdefinierte Datentypen, falls die vordefinierten nicht ausreichen. Die Datentypen werden durch URIs benannt, aber in der Praxis stellt sich das schnell als unhandlich heraus. Deshalb werden die Datentypen in dieser Arbeit nur mit ihrem Kurznamen benannt und oft sogar ganz weggelassen (vgl. Abschnitt 4.3). Die vordefinierten Datentypen kommen aus drei verschiedenen Namensräumen:

- `http://www.w3.org/2001/XMLSchema` (XMLSchema)
- `urn:oasis:names:tc:xacml:1.0:data-type` (XACML 1.0)
- `http://www.w3.org/TR/2002/WD-xquery-operators-20020816` (XQuery 1.0)

Die einzelnen Datentypen sind (Reihenfolge wie in [XACML-1.0, Appendix B.4]):

x500Name: Ein Bezeichner, wie er in *ITU-T Rec. X.520 Distinguished Name* definiert ist und auch in LDAP verwendet wird, in UTF-8 codiert. Ein Beispiel ist „CN=Julius Hibbert, OU=Medical, O=Springfield Hospital“.

rfc822Name: Ein Bezeichner, wie er für E-Mail-Adressen verwendet wird, zum Beispiel „jhibbert@med.example.com“.

string: Eine Kette von Unicode-Zeichen. Dieser Datentyp kann laut Definition in XML-Schema nicht für Binärdaten verwendet werden, da ein `string` zum Beispiel keine ASCII-Sonderzeichen (U+0000 bis U+001f, außer U+000a und U+000d) enthalten darf. Er ist also wirklich nur für Textdaten geeignet.

boolean: Ein Wahrheitswert, der durch die Stringlitterale `true` und `false` bzw. die Zahlen 1 und 0 dargestellt wird.

integer: Alle ganzen Zahlen, ohne Größenbeschränkung.

double: Der „double-precision 64-bit floating point type“ aus IEEE-754, jedoch mit der Eigenschaft, dass `NaN = NaN` ist.¹ Das ist in IEEE-754 nicht der Fall.²

¹[XMLSchema, 3.5.2 double]

²[IEEE754, 5.7 Comparison]

time: Ein Zeitpunkt innerhalb eines Tages, optional inclusive Angabe der Zeitzone. Dadurch, dass es Zeitpunkte *mit* und *ohne* Angabe einer Zeitzone geben kann, können Zeitpunkte nicht total geordnet werden und sind daher für direkte Vergleiche ungeeignet. In XACML 1.0 ist der Datentyp daher schlecht verwendbar, in XACML 2.0 wurde als Abhilfe die Funktion `time-in-range` eingeführt, die dieses Problem explizit löst.

date: Ein oben halboffenes Zeitintervall von genau einem Tag Länge. Durch die verschiedenen Zeitzonen ist bei der Verwendung dieses Typs Vorsicht angesagt, da das Datum in verschiedenen Zeitzonen unterschiedlich sein kann.

dateTime: Ein Zeitpunkt an einem bestimmten Datum, optional mit Zeitzone. Dadurch, dass hier Datum und Tageszeit zusammenkommen, kann der genaue Zeitpunkt in die UTC umgerechnet werden, so dass eine vollständige Ordnung möglich ist.

anyURI: Ein URI, wie in RFC 2396 und RFC 2732 definiert. Viele Dinge in XACML haben diesen Datentyp, um auf die Notwendigkeit von Namensräumen hinzuweisen, gerade zur Identifikation von Subjekten und Ressourcen.

hexBinary: Eine Folge von Bytes, die jeweils in zwei Hex-Ziffern codiert sind.

base64Binary: Eine Folge von Bytes, die mit base64 codiert sind. Obwohl die repräsentierten Werte mit *hexBinary* identisch sind, werden sie doch wie verschiedene Datentypen behandelt und nicht etwa wie verschiedene Codierungen des selben Datentyps. Diese Unterscheidung ist schon in XMLSchema vorhanden und wurde von dort übernommen. Es gibt in XACML 2.0 auch keine Möglichkeit, diese beiden Datentypen zu vergleichen oder ineinander umzuwandeln. Auch in XACML 3.0 ist diese Möglichkeit bisher nicht vorgesehen.

dayTimeDuration: Eine Zeitdauer, die in Tagen und Tagesbruchteilen gemessen wird.

yearMonthDuration: Eine Zeitdauer, die in Jahren und Monaten gemessen wird.

Die Datentypen aus XMLSchema (*string*, *boolean*, *integer*, *double*, *time*, *date*, *dateTime*, *anyURI*, *hexBinary*, *base64Binary*) stellen die allgemein üblichen Datentypen bereit, wie sie auch in anderen Programmiersprachen vorkommen. Die Datentypen aus XACML 1.0 (*x500Name*, *rfc822Name*) ergänzen diese, um die Identifikation von Subjekten und Objekten im Bereich der Zugriffskontrolle einfach abzubilden.

XMLSchema definiert noch den Datentyp *duration*, der aber nicht in XACML übernommen wurde, aus gutem Grund: Aufgrund der Unregelmäßigkeiten des Gregorianischen Kalenders (Monate mit unterschiedlich vielen Tagen) ist es nicht möglich, auf diesem Datentyp eine vollständige Ordnung zu definieren. (Beispiel: sind 30 Tage mehr als, weniger als oder gleich viel wie ein Monat?) Um dieses Problem zu umgehen, hat XACML statt dem allgemeinen Datentyp *duration* die spezielleren Datentypen *dayTimeDuration* und *yearMonthDuration* aus XQuery 1.0 übernommen.

4.6.1 Änderungen in XACML 2.0

In XACML 2.0 wurden zwei neue Datentypen hinzugefügt: *ipAddress* und *dnsName*. Durch diese Datentypen wird die Identifikation von Rechnern und IP-Netzwerken vereinfacht.

ipAddress: Ein DNS-Name (sic!) oder eine IP-Adresse (IPv4 oder IPv6), optional mit Netzmaske und optional mit Portbereich. Beispiele: 4.1.1.1:53, 10.0.0.0/24:80, 127.0.0.1:1-1023, www.google.com.

dnsName: Ein DNS-Name (aber keine IP-Adresse), mit der zusätzlichen Möglichkeit, ein Sternchen als Wildcard-Operator statt der linken Subdomain einzusetzen. Wie schon bei *ipAddress* kann auch hier ein optionaler Portbereich angegeben werden. Beispiele: *.google.com, med.example.com:80.

4.7 Policys

Der Großteil der XACML-Norm beschäftigt sich mit der Definition und Verarbeitung der Policys. Es gibt zwei Arten von Policys: eine einfache (*<Policy>*), in der einzelne Regeln kombiniert werden, und eine zusammengesetzte (*<PolicySet>*), in der Policys zu größeren Einheiten kombiniert werden. Gemeinsam ist diesen beiden Arten, dass sie einen Anwendungsbereich (*<Target>*) haben, der festlegt, auf welche Anfragen die Policy überhaupt Antworten geben kann. Die Kindelemente einer *<Policy>* sind Regeln, die Kindelemente einer *<PolicySet>* sind *<Policy>*s, *<PolicySet>*s oder Referenzen darauf. Gemeinsam ist diesen Kindelementen, dass man sie bezüglich einer Anfrage auswerten kann. Das Ergebnis so einer Auswertung ist eine Entscheidung, die die folgenden Werte annehmen kann:

Permit, wenn der Zugriff erlaubt sein soll,

Deny, wenn der Zugriff verboten sein soll,

NotApplicable, wenn die Policy nicht auf die Anfrage anwendbar ist, und

Indeterminate, wenn bei der Auswertung ein Fehler aufgetreten ist.

Eine weitere Eigenschaft einer Policy ist der Kombinationsalgorithmus, der als Parameter die Kindelemente (Regeln oder Policys) bekommt, diese dann nach Bedarf auswertet und die Ergebnisse geeignet kombiniert.

Listing 4.3 zeigt, wie eine typische Policy aussieht. Die Details der Kindelemente sind ausgelassen, um die Darstellung einfach zu halten.

In den folgenden drei Abschnitten wird zunächst beschrieben, wie der Anwendungsbereich einer Policy festgelegt wird (Abschnitte 4.8 bis 4.9.1), dann wie die Regeln aussehen (Abschnitt 4.11) und schließlich, wie diese Regeln zu einer Policy kombiniert werden (Abschnitt 4.12).

```

<Policy RuleCombiningAlgId="first-applicable">
  <Target>
    ...
  </Target>
  <Rule RuleId="Admins" Effect="Permit">
    <Description>Administrators may do everything.</Description>
    ...
  </Rule>
  <Rule RuleId="scratch" Effect="Permit">
    <Description>Everyone may do everything with scratch objects</Description>
    ...
  </Rule>
  ...
  <Rule RuleId="fallback" Effect="Deny"/>
</Policy>

```

Listing 4.3: Fragment einer XACML-Policy

4.8 Anwendungsbereich

Der Anwendungsbereich (`<Target>`) ist ein Bestandteil der Elemente `<Policy>`, `<PolicySet>` und `<Rule>` (Elternelement). Er definiert, auf welche Anfragen das Elternelement anwendbar ist. Es ist auf eine Anfrage anwendbar, wenn der Anwendungsbereich zu der Anfrage *passt*.

Das `<Target>`-Element besteht aus drei Abschnitten (`<Subjects>`, `<Actions>`, `<Resources>`; ab XACML 2.0 zusätzlich noch `<Environments>`). Es *passt* zu einer Anfrage, wenn alle Abschnitte passen.

In jedem dieser Abschnitte kann es jeweils Unterabschnitte, in der Singularform benannt, geben. Die heißen dann `<Subject>`, `<Action>`, `<Resource>` und (ab XACML 2.0) `<Environment>`. Ein Abschnitt *passt* zu einer Anfrage, wenn mindestens ein Unterabschnitt passt.

Ein Unterabschnitt wiederum enthält eine Reihe von `<*Match>`-Elementen, zum Beispiel `<SubjectMatch>`, `<ResourceMatch>`, `<ActionMatch>`. Ein Unterabschnitt *passt* zu einer Anfrage, wenn alle `<*Match>`es passen. Wie ein `<*Match>` aussieht, und ob er passt, wird im nächsten Abschnitt beschrieben.

Abbildung 4.2 stellt die Zusammenhänge übersichtlich dar: Ein `<Target>` stellt eine Konjunktion von Elementen in Zeile 2 dar, diese sind wiederum Disjunktionen der Elemente aus Zeile 3, und diese schließlich Konjunktionen von `<*Match>`es. In der Spezifikation werden die Elemente aus der Abbildung jeweils einzeln und ausführlich definiert, obwohl sie strukturell eigentlich gleich sind. Der einzige Unterschied zwischen den verschiedenen `<*Match>`es ist, dass die Namen ihrer jeweiligen Kindelemente (z. B. `<SubjectAttributeDesignator>`) darauf hinweisen, aus welcher Kategorie innerhalb der Anfrage das Attribut kommt. Die Unterscheidung von strukturell gleichen Elementen macht

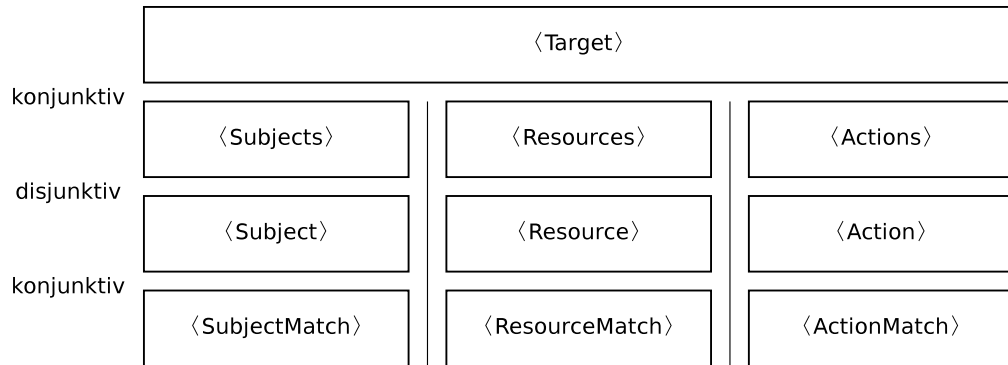


Abbildung 4.2: Aufbau des <Target>-Elements in XACML 1.0

den XACML-Standard aber unnötig kompliziert und erschwert so die Lesbarkeit. Aus diesem Grund wurden die Elemente in XACML 3.0 vereinheitlicht (vgl. Abschnitt 4.8.1).

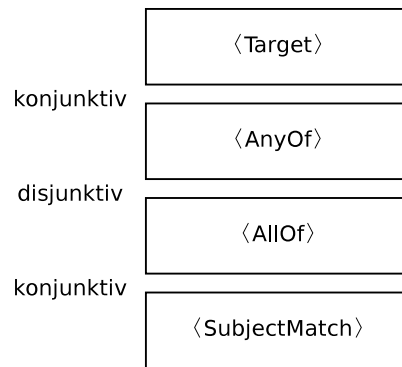
Wenn einer der Abschnitte (<Subjects>, <Resources> oder <Actions>) leer ist, muss ein spezielles XML-Element eingefügt werden. So ist es zum Beispiel nicht erlaubt, einfach <Subjects/> zu schreiben, wenn das Subjekt der Anfrage irrelevant für die Anwendbarkeit ist. Die korrekte Schreibweise für so einen Fall ist <Subjects><anySubject/></Subjects>.

4.8.1 Optimierungsmöglichkeiten

Ein Ziel bei der Entwicklung des <Target>-Elements war, dass ein PDP gerade bei vielen gleichberechtigten Policies bzw. Regeln schnell entscheiden kann, welche davon anwendbar sind und welche nicht. Zu diesem Zweck sollten die Funktionen, die in den <*Match>-Elementen verwendet werden, „leicht indizierbar“ sein.[XACML-1.0, A.12] Eine mögliche Optimierung wäre, alle Stringvergleiche zu einem bestimmten Attribut aus den <Target>-Elementen herauszusuchen und die Stringliterals in einer Hashtabelle zu speichern, die zu einem Stringliteral die Liste aller Policies speichert, die anwendbar sind, wenn das Attribut genau diesen Wert hat. Durch diese Optimierung sinkt der Suchaufwand für eine Policy, der vorher $O(n)$ ist, auf einen Wert zwischen $O(n)$ und $O(1)$, jeweils abhängig davon, wie viele Elternelemente genau diese Kombination erfordern.

Beispiel: Gegeben sei ein System mit 100 Rollen und 100 Ressourcen. In einer Policy existiert für jede dieser Kombination eine Regel, die den Zugriff entweder erlaubt oder verbietet. Ein unoptimierter PDP würde nun nacheinander (oder auch parallel) alle Regeln durchgehen, bis er eine findet, die passt. Das erfordert durchschnittlich 5000,5 Regelauswertungen.

Ein optimierter PDP dagegen würde alle Regeln nach den benötigten Werten eines bestimmten Attributs (zum Beispiel der Rolle) in eine Tabelle einsortieren. In diesem Beispiel sind das pro Rolle noch 100 Regeln. Bei einer Anfrage muss er dann nur noch die Liste dieser 100 Regeln durchgehen. Wenn

Abbildung 4.3: Aufbau des `<Target>`-Elements in XACML 3.0

er das mit linearer Suche durchführt, bleiben durchschnittlich 50,5 Regelauswertungen, eine Verbesserung fast um den Faktor 100. Die Verbesserung wird jedoch dadurch geschmälert, dass am Anfang jeder Policyauswertung die passende Liste der verbleibenden Regeln aus einer Tabelle herausgesucht werden muss. Zudem muss die Tabelle vor der ersten Anfrage einmalig erzeugt werden und danach dauerhaft im Speicher verfügbar sein.

Änderungen in XACML 2.0 und 3.0

Ab XACML 2.0 ist der zusätzliche Aufwand, leere Abschnitte mit `<anySubject>` o. ä. zu markieren, entfallen, da diese Elemente keinen praktischen Nutzen haben. Ohne das explizite Markieren der Abschnitte ist die Semantik genau die gleiche wie mit dem Markieren, da der Wahrheitswert einer leeren Konjunktion stets `true` ist.

Statt der Vielzahl von XML-Elementen, die in Abbildung 4.2 auftauchen, wurde in XACML 3.0 eine einheitliche, einfache Struktur eingeführt, die in Abbildung 4.3 dargestellt ist. An die Stelle der Abschnitte tritt das Element `<AnyOf>`, dessen Name jetzt nicht mehr die Kategorie beinhaltet, sondern nur noch die logische Funktion. Dementsprechend tritt in der Ebene darunter das Element `<AllOf>` auf, und die verschiedenen `<*Match>`-Elemente wurden zu einem einzigen `<Match>` zusammengefasst. Bei dieser Neugestaltung ist jedoch eine wichtige Information nicht mehr vorhanden: In XACML 1.0 kann zum Beispiel innerhalb von einem `<SubjectMatch>` ein `<SubjectAttributeDesignator>` benutzt werden, um die Werte eines Attributs aus der Anfrage zu ermitteln. Die Kategorie innerhalb der Anfrage, aus der das Attribut kommen soll, ist hier im Elementnamen (Subject) codiert, und genau der fällt bei der Umstrukturierung weg. In XACML 3.0 wird die gewünschte Kategorie stattdessen direkt und explizit im `<AttributeDesignator>`-Element als Attribut `Category` angegeben. Durch diese Änderung werden die Policies in ihrer XML-Form deutlich kürzer und damit lesbarer.

```

<SubjectMatch MatchId="string-equal">
  <AttributeValue>admin</AttributeValue>
  <SubjectAttributeDesignator AttributId="orka:role"/>
</SubjectMatch>

```

Listing 4.4: Beispiel für ein `<SubjectMatch>`-Element

4.9 Abgleich (Match)

Ein `<*Match>`-Element prüft, ob ein Attribut aus einer Anfrage zu einem Attribut in einer Policy oder Regel *passt*. Es enthält eine Funktions-ID, einen Attributwert und einen Verweis auf ein Attribut aus der Anfrage. Die Funktions-ID bezeichnet eine der im folgenden Abschnitt definierten Funktionen oder auch eine benutzerdefinierte Funktion. Diese Funktion bekommt als erstes Argument den Attributwert aus dem `<*Match>`-Element und als zweites Argument nacheinander alle Attributwerte aus der Anfrage. Wenn die Funktion für eine dieser Kombinationen `true` zurückgibt, *passt* das `<*Match>`-Element, ansonsten *passt* es nicht.

Die Einschränkungen, die sich für die Funktion ergeben, sind also:

- Sie muss als Rückgabotyp *boolean* haben.
- Sie muss genau zwei Argumente akzeptieren.
- Das erste Argument muss aus der Policy stammen.
- Das zweite Argument muss aus der Anfrage stammen.
- Beide Argumente müssen von elementarem Datentyp sein, also keine Multimengen.

Typische Funktionen, die all diese Bedingungen erfüllen, sind die Vergleichsfunktionen *type-equal*, *type-greater-than* sowie *regexp-string-match*.

Listing 4.4 demonstriert, wie ein `<*Match>`-Element aussieht, um ein Attribut eines Subjekts (`orka:role`) mit einem festen Wert ("`admin`") zu vergleichen.

Auswertung

Wenn ein `<*Match>`-Element bezüglich einer Anfrage ausgewertet wird, ist das Ergebnis entweder `true`, `false` oder `Indeterminate`. Wenn bei der Auswertung kein Fehler auftritt, wird der Rückgabewert der (booleschen) Funktion übernommen, ansonsten ist das Ergebnis `Indeterminate`.

4.9.1 Vordefinierte Funktionen

XACML 1.0 definiert insgesamt 205 Funktionen, die beim Evaluieren von Policies verwendet werden können. Die Anzahl der Funktionen ist hauptsächlich deshalb so groß, weil XACML Multimengen (Bags) von elementaren Datentypen unterstützt. Die Definition

eines Datentyps beinhaltet in XACML praktisch nur den Namen des Datentyps und die Repräsentation von Literalen als Zeichenketten, nicht jedoch eine Funktion, die zwei Werte des Datentyps auf Gleichheit testet. Gerade diese Funktion ist jedoch eine Voraussetzung für viele Mengenoperationen. Da diese Funktion nicht verfügbar ist, bleibt nur die Möglichkeit, für jeden der Datentypen die 9 Mengenfunktionen einzeln zu definieren. Bei den Funktionen *type-bag*, *type-bag-size* und *type-one-and-only* wäre das eigentlich nicht nötig gewesen, da diese Funktionen keine Vergleiche durchführen, aus Gründen der Einheitlichkeit gibt es sie trotzdem.

Funktionen für Multimengen und Mengen

Die Funktionen zum Arbeiten mit Multimengen machen die größte Anzahl aus. Für jeden der 14 in XACML 1.0 vordefinierten Datentypen gibt es die folgenden Funktionen:

- *type-at-least-one-member-of*(X, Y): Prüft, ob mindestens ein Element aus der Multimenge X in der Multimenge Y enthalten ist.
- *type-bag*($xs \dots$): Kombiniert eine Liste von elementaren Werten xs zu einer Multimenge.
- *type-bag-size*(x): Gibt die Anzahl der Elemente in der Multimenge zurück.
- *type-intersection*(X, Y): Interpretiert X und Y als einfache Mengen und gibt deren Durchschnittsmenge zurück.
- *type-is-in*(x, X): Prüft, ob x in der Multimenge X enthalten ist.
- *type-one-and-only*(X): Extrahiert aus einer Multimenge mit genau einem Element dieses Element. Falls die Multimenge nicht genau ein Element enthält, evaluiert die Funktion zu **Indeterminate**.
- *type-set-equals*(X, Y): Interpretiert X und Y als einfache Mengen und prüft, ob sie gleich sind.
- *type-subset*(X, Y): Interpretiert X und Y als einfache Mengen und prüft, ob X eine Teilmenge von Y ist.
- *type-union*(X, Y): Interpretiert X und Y als einfache Mengen und gibt deren Vereinigungsmenge zurück.

Um Multimengen effizient zu verarbeiten, gibt es Funktionen höherer Ordnung, die jeweils eine Funktion als Argument haben und diese Funktion auf die restlichen Argumente anwenden. In der folgenden Darstellung ist f eine boolesche Funktion mit zwei Parametern, x ein elementarer Wert und M und N Multimengen; die Datentypen von M und N müssen nicht gleich sein, sind es aber meistens.

- *all-of*(f, x, M): $\forall m \in M: f(x, m)$.

- $\text{any-of}(f, x, M): \exists m \in M: f(x, m).$
- $\text{all-of-all}(f, M, N): \forall m \in M: \forall n \in N: f(m, n).$
- $\text{all-of-any}(f, M, N): \forall m \in M: \exists n \in N: f(m, n).$
- $\text{any-of-all}(f, M, N): \exists m \in M: \forall n \in N: f(m, n).$
- $\text{any-of-any}(f, M, N): \exists m \in M: \exists n \in N: f(m, n).$

In vielen Fällen ist f eine Vergleichsfunktion der Art *type-equal* oder *type-greater-than*. Durch die explizite Angabe der Vergleichsfunktion sind diese Funktionen unabhängig von dem verwendeten Datentyp, im Gegensatz zu den typspezifischen Funktionen weiter oben.

Eine weitere Funktion höherer Ordnung ist $\text{map}(f, M)$, die die Funktion f auf alle Elemente der Multimenge M anwendet und diese Ergebnisse zu einer neuen Multimenge kombiniert. f muss dazu eine Funktion sein, die einen einzigen Parameter hat, der selbst keine Multimenge ist. Sie darf auch keine Multimenge zurückgeben. Diese Einschränkungen führen dazu, dass map nur auf wenige der vordefinierten Funktionen sinnvoll anwendbar ist.

Vergleichsfunktionen

Jeder der 14 vordefinierten Datentypen hat eine *type-equal*-Funktion, und die Datentypen *string*, *int*, *double*, *date*, *dateTime* und *time* haben auch die übrigen Vergleichsfunktionen *type-less-than*, *type-less-than-or-equal*, *type-greater-than* und *type-greater-than-or-equal*.

Arithmetische Funktionen

XACML unterstützt arithmetische Operationen für beliebig große Ganzzahlen sowie für 64-Bit-Gleitkommazahlen.

Die Funktionen für Ganzzahlarithmetik sind: *integer-add*, *integer-subtract*, *integer-multiply*, *integer-divide*, *integer-mod*³ und *integer-abs*.

Die Funktionen für Gleitpunktarithmetik sind: *double-add*, *double-subtract*, *double-multiply*, *double-divide*⁴, *double-abs*, *floor* und *round*.

Um zwischen Ganzzahlen und Gleitpunktzahlen zu konvertieren, gibt es die Funktionen *integer-to-double* und *double-to-integer*.

Die Funktionen für Datums- und Zeitarithmetik sind in zwei Kategorien eingeteilt: Bei Datumsangaben vom Typ *date* sind nur Berechnungen möglich, die in ganzen Monaten erfolgen. Diese Berechnungen werden von den Funktionen *date-add-yearMonthDuration* und *date-subtract-yearMonthDuration* bereitgestellt. Es ist beim Datentyp *date* also insbesondere nicht möglich, einzelne Tage zu addieren oder zu subtrahieren.

³Es ist nicht festgelegt, wie sich diese Funktion bei negativen Argumenten verhält. Vergleiche <http://de.wikipedia.org/wiki/Modulo>.

⁴Der Divisor darf im Unterschied zu [IEEE754] nicht 0 sein.

Flexiblere Möglichkeiten zur Zeitberechnung bietet der Datentyp *dateTime*, bei dem mit den Funktionen *dateTime-add-dayTimeDuration*, *dateTime-add-yearMonthDuration*, *dateTime-subtract-dayTimeDuration* und *dateTime-subtract-yearMonthDuration* Zeitabschnitte addiert oder subtrahiert werden können, deren Dauer entweder in Jahren und Monaten oder aber in Tagen, Stunden, Minuten und Sekunden angegeben ist.

Für den Datentyp *time* stehen gar keine arithmetischen Funktionen zur Verfügung. Die einzigen Operationen auf Werten dieses Typs sind damit die Vergleichsfunktionen.

Boolesche Funktionen

Die Funktionen *and*, *or* und *n-of* werten ihre Argumente der Reihe nach aus (und nicht wie die anderen Funktionen in beliebiger Reihenfolge). Die Funktion *n-of(n, cond...)* gibt genau dann **true** zurück, wenn mindestens *n* der weiteren Argumente **true** sind. Die Funktion *not* negiert ihr einziges Argument.

Sonstige Funktionen

Um mit Zeichenketten zu arbeiten, gibt es die Funktion *regex-string-match*, die prüft, ob ein regulärer Ausdruck (gemäß XQuery 1.0) zu einem String passt. Weiterhin kann man mit *string-normalize-space* führende und abschließende Leerzeichen einer Zeichenkette entfernen. Leerzeichen in der Mitte bleiben von der Normalisierung unberührt. Mit *string-normalize-to-lower-case* kann man alle Großbuchstaben einer Zeichenkette in Kleinbuchstaben umwandeln. Es gibt jedoch keine Funktion, um Kleinbuchstaben in Großbuchstaben umzuwandeln, vermutlich mit Rücksicht auf das deutsche „ß“.

Die Funktionen *rfc822Name-match* und *x500Name-match* prüfen, ob ein Wert dieses Datentyps zu dem angegebenen Muster passt. Die Muster ähneln regulären Ausdrücken, sind aber speziell auf die jeweiligen Datentypen und deren Anwendungsbereiche zugeschnitten.

Es gibt noch drei Funktionen, um mit XQuery auf Daten in der Anfrage zuzugreifen (*xpath-node-count*, *xpath-node-equal*, *xpath-node-match*), aber die sind optional.

Schließlich gibt es noch die Funktion *present*, deren Verhalten nirgends spezifiziert ist, die aber trotzdem in der Liste der Funktionen auftaucht, die auf jeden Fall zu implementieren sind. [XACML-1.0, Abschnitt 10.2.8] Wahrscheinlich sollte sie überprüfen, ob ein bestimmtes Attribut in einer Anfrage definiert ist oder nicht.

Änderungen in XACML 2.0

In XACML 2.0 wurde die Funktion *present* entfernt. Die Funktion *regex-string-match* wurde in *string-regex-match* umbenannt, damit der Name besser zu den anderen *string-**-Funktionen passt, [Anderson2005] die Reihenfolge der Parameter (RegExp, String) wurde aber beibehalten. Neu hinzugekommen sind:

anyURI-regex-match: Prüft, ob der URI zu dem regulären Ausdruck passt.

ipAddress-regex-match: Prüft, ob die IP-Adresse zu dem regulären Ausdruck passt.

Obwohl der Datentyp *ipAddress* eigentlich auch DNS-Namen zulässt, steht in der

4 XACML

Funktionsbeschreibung explizit, dass hier nur IPv4- und IPv6-Adressen akzeptiert werden.

dnsName-regexp-match: Prüft, ob der DNS-Name zu dem regulären Ausdruck passt.

rfc822Name-regexp-match: Prüft, ob die Emailadresse zu dem regulären Ausdruck passt.

string-concatenate: Hängt zwei Zeichenketten aneinander.

time-in-range: Prüft, ob ein Zeitpunkt zwischen zwei anderen liegt. Diese Funktion berücksichtigt insbesondere die Zeitzonen, so dass zwei Zeitpunkte verglichen werden können, selbst wenn einer eine Zeitzone beinhaltet (z. B. 20:00+02:00) und der andere nicht (z. B. 18:30). In XMLSchema ist definiert,⁵ dass diese beiden Zeitpunkte „ungeordnet“ sind, da der Zeitpunkt ohne Zeitzone eine Abweichung von ± 13 Stunden haben kann.⁶

uri-string-concatenate: Hängt den String an den URI an und gibt das Ergebnis als URI zurück.

x500Name-regexp-match: Prüft, ob der X.500-Name zu dem regulären Ausdruck passt.

Die beiden in XACML 2.0 neu eingeführten Datentypen können im Unterschied zu den bestehenden Datentypen nicht auf Gleichheit geprüft werden, da die Funktionen *dnsName-equal* und *ipAddress-equal* nicht existieren. Auch andere naheliegende Fragen wie etwa „Kommt eine IP-Adresse aus einem bestimmten Netzwerk?“, „Ist der Port aus der Anfrage innerhalb des erlaubten Bereichs?“, „Gehört ein DNS-Name zu einer bestimmten Domain?“ können nicht allgemein ausgedrückt werden. Einige Spezialfälle können durch die Funktionen **-regexp-match* abgedeckt werden, aber bei weitem nicht alle.

Änderungen in XACML 3.0

In XACML 3.0 wurden Umwandlungsfunktionen für alle elementaren Datentypen außer den beiden Binärtypen (*hexBinary* und *base64Binary*) eingeführt. Sie lassen sich jeweils in Strings (*string-from-type*) und wieder zurück (*type-from-string*) umwandeln.

Die Funktion *uri-string-concatenate* wurde verworfen, da sie durch die neuen Umwandlungsfunktionen ausgedrückt werden kann. Die *type-regexp-match*-Funktionen, die genauso redundant geworden sind, bleiben jedoch weiterhin.

4.10 Arten von Ausdrücken

Mit Hilfe der eben eingeführten Funktionen ist es möglich, komplexe Ausdrücke zu bilden. Ein Grundbaustein hierfür ist die Anwendung einer Funktion auf eine Liste von Argumenten. (Partielle Anwendung wird jedoch nicht unterstützt.) Diese Argumente

⁵[XMLSchema, Abschnitte 3.2.8 sowie 3.2.7.4]

⁶Siehe <http://sunxacml.sourceforge.net/guide.html#xacml-time>

müssen wiederum Ausdrücke sein. Es gibt folgende Arten von Ausdrücken, die hier zusammen mit ihrem XACML-Namen beschrieben werden:

- Ein Ausdruck der Art `<AttributeValue>` stellt einen Ausdruck von einem bestimmten elementaren Datentyp dar, dessen Wert durch ein Literal angegeben ist. Beispiele dafür sind:

```
<AttributeValue DataType="string">Hallo, Welt.</AttributeValue>
<AttributeValue DataType="integer">12345</AttributeValue>
<AttributeValue DataType="dayTimeDuration">P180D</AttributeValue>
```

- Ein Ausdruck der Art `<*AttributeDesignator>` dient dazu, aus einer Anfrage die Werte eines benannten Attributes zu ermitteln. Da es einige Standards gibt, die mehrere Werte pro Attribut erlauben (zum Beispiel LDAP, XML-Query), und XACML leicht mit diesen Standards kombiniert werden können soll, kann ein Attribut auch in XACML mehrwertig sein. Je nachdem, zu welcher Kategorie das Attribut gehört, werden verschiedene Varianten des `<*AttributeDesignator>`s benutzt. Beispiele sind:

```
<SubjectAttributeDesignator Attributeld="subject-id" DataType="string"/>
<SubjectAttributeDesignator Attributeld="subject-id" DataType="integer"/>
<ActionAttributeDesignator Attributeld="action-id" DataType="string"/>
<ResourceAttributeDesignator Attributeld="resource-id" DataType="anyURI"/>
<EnvironmentAttributeDesignator Attributeld="current-time" DataType="time"/>
```

Durch die explizite Trennung von Attributname und Datentyp ist es möglich, dass ein Attribut in einer Anfrage mehrere Werte von unterschiedlichen Datentypen bekommt. So kann zum Beispiel für `subject-id` sowohl eine numerische ID als auch der Benutzername angegeben werden. Welcher Wert letztendlich verwendet wird, kann die Policy frei entscheiden.

- Ein Ausdruck der Art `<*AttributeSelector>` dient wie schon `<*AttributeDesignator>` dazu, Daten aus einer Anfrage zu extrahieren, allerdings werden die Daten hier anders identifiziert: statt eines Namens wird ein XPath-Ausdruck angegeben, mit dem auf fast beliebige XML-Knoten der Anfrage zugegriffen werden kann. Diese Art wird insbesondere verwendet, wenn es sich bei der Ressource um ein XML-Dokument handelt und der Zugriff vom Inhalt dieses Dokuments abhängt. In so einem Fall wird das Dokument vollständig oder teilweise in die Anfrage eingebettet. Die Unterstützung für `<*AttributeSelector>` ist allerdings nicht verpflichtend. Ein Beispiel, um die Patienten-ID aus einer medizinischen Akte zu extrahieren, ist:

```
<AttributeSelector DataType="string"
  RequestContextPath="//xacml-context:Resource/xacml-
  context:ResourceContent/med:record/med:patient@id"/>
```

- Ein Ausdruck der Art `<Function>` ist nur als Argument für Funktionen höherer Ordnung sinnvoll. Es gibt daher auch in XACML keinen speziellen Datentyp (im Sinne von Abschnitt 4.6) dafür.
- (Ab XACML 2.0) Ein Ausdruck der Art `<VariableReference>` gibt den Wert einer Variablen zurück, der vorher mittels `<VariableDefinition>` definiert wurde. Variablendefinitionen gelten im Bereich einer einzelnen Policy und können verwendet werden, um häufig benötigte Ausdrücke mit einem Namen zu versehen und eventuell Rechenzeit zu sparen.
- Ein Ausdruck der Art `<Apply>` ist das Ergebnis eines Funktionsaufrufs. Während die anderen Arten nur Endknoten im Ausdrucksbaum darstellen, lassen sich mit diesem Konstruktor Ausdrücke beliebig komplex schachteln.

4.11 Regeln

Eine Regel ist nach einer Policy das nächstkleinere Element, das für die Zugriffsentscheidung wichtig ist. Sie wird definiert durch ihren Anwendungsbereich (`<Target>`), optional eine Zusatzbedingung (`<Condition>`) sowie die Auswirkung der Regel (`Effect`).

Das `<Target>` ist das gleiche wie schon bei der Policy. Während eine Policy immer einen Anwendungsbereich definieren muss, kann der bei einer Regel fehlen. In so einem Fall wird der Anwendungsbereich der Policy übernommen.

Die `<Condition>` ist ein Ausdruck, der syntaktisch mit `<Apply>` identisch ist (vgl. die Ausdrucksarten im vorherigen Abschnitt) und hier bloß einen anderen Namen trägt, um die zusätzliche Bedeutung gegenüber einer allgemeinen Funktionsanwendung deutlicher zu betonen. Durch den Namen wird auch deutlich, dass es sich bei der hier angegebenen Funktion um eine boolesche Funktion handeln muss.

Die Auswirkung der Regel ist entweder **Permit** oder **Deny**.

Ein Beispiel für eine Regel ist in Listing 4.5 dargestellt. Im `<Target>` wird überprüft, ob der Benutzer ein Administrator ist. Die zusätzliche Bedingung ist, dass das System im Debugmodus läuft. Diese Bedingung kann nicht im `<Target>` codiert werden, da das nicht in der Syntax von XACML 1.0 vorgesehen ist. (Ab XACML 2.0 geht es.) Da das Attribut, das vom `<EnvironmentAttributeDesignator>` zurückgegeben wird, mehrwertig sein kann, muss es zunächst auf einen einzigen Wert reduziert werden (*boolean-one-and-only*), bevor es von anderen Funktionen (*boolean-equal*) weiterverarbeitet werden kann.

Auswertung

Die Auswertung einer Regel ergibt einen der Werte **Permit**, **Deny**, **NotApplicable** oder **Indeterminate**.

Zuerst wird geprüft, ob die Regel überhaupt anwendbar ist, ihr `<Target>` also zu der Anfrage passt. Falls das nicht der Fall ist, ist das Ergebnis **NotApplicable**. Danach wird die `<Condition>` ausgewertet, und wenn sie **true** ergibt, ist das Ergebnis der Wert des `Effect`-Attributs. Wenn sie **false** ergibt, ist das Ergebnis **NotApplicable**. Falls in irgendeinem Schritt dieses Ablaufes ein Fehler auftritt, ist das Ergebnis **Indeterminate**.

```

<Rule Effect="Permit">
  <Description>In the debugging mode, administrators may do everything.</Description>
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>admin</AttributeValue>
      <SubjectAttributeDesignator Attributeld="role"/>
    </SubjectMatch>
  </Target>
  <Condition FunctionId="boolean-equal">
    <Apply FunctionId="boolean-one-and-only">
      <EnvironmentAttributeDesignator Attributeld="debuggingmode"/>
    </Apply>
    <AttributeValue>true</AttributeValue>
  </Condition>
</Rule>

```

Listing 4.5: Beispiel für eine XACML-Regel

4.12 Kombinationsalgorithmen

In XACML 1.0 sind die folgenden Kombinationsalgorithmen vordefiniert, jeweils in Paaren: einer, um Regeln zu einer Policy zu kombinieren, der andere, um Policies zu noch größeren Policies (`<PolicySet>`) zu kombinieren.

Die vordefinierten Algorithmen evaluieren ihre Kindelemente (entweder Regeln oder weitere Policies) und berechnen ihr Ergebnis dann aus den Ergebnissen der Evaluierungen.

Bei der Evaluierung eines Kindelements können vier verschiedene Ergebnisse auftreten:

- **Permit**: Das Element erlaubt den Zugriff.
- **Deny**: Das Element verbietet den Zugriff.
- **NotApplicable**: Das Element ist nicht auf die Anfrage anwendbar.
- **Indeterminate**: Das Element kann aus irgendeinem Grund nicht ausgewertet werden.

Wenn das Ergebnis **Indeterminate** ist, kann man noch danach fragen, zu welchem Ergebnis die Evaluation kommen *könnte*, wenn kein Fehler aufgetreten wäre. Eine Regel hat fest encodiert, ob sie zu **Permit** oder zu **Deny** evaluiert. Daher ist in so einem Fall klar, dass die Regel niemals zum gegenteiligen Ergebnis evaluieren kann. Bei Policies ist das Ergebnis nicht fest encodiert, so dass diese in so einem Fall sowohl zu **Permit** als auch zu **Deny** evaluieren könnte. Sowohl Regeln als auch Policies könnten auch zu **NotApplicable** evaluieren. Viele der kombinierenden Algorithmen nutzen diese Unterscheidung der verschiedenen **Indeterminate**-Ergebnisse.

deny-overrides: Kombiniert die Evaluierungsergebnisse mit Tendenz zu **Deny**. Die genaue Funktionsweise ist:

1. Wenn (mindestens) ein Kindelement zu **Deny** evaluiert, ist das Ergebnis **Deny**.
2. Wenn ein Kindelement zu **Deny** evaluieren könnte, ist das Ergebnis **Indeterminate**.
3. Wenn ein Kindelement zu **Permit** evaluiert, ist das Ergebnis **Permit**.
4. Wenn ein Kindelement zu **Permit** evaluieren könnte, ist das Ergebnis **Indeterminate**.
5. Andernfalls gibt es nur noch Kindelemente, die zu **NotApplicable** evaluieren, und das Ergebnis ist ebenfalls **NotApplicable**.

permit-overrides: kombiniert die Evaluierungsergebnisse mit Tendenz zu **Permit**. Der Algorithmus funktioniert so wie **deny-overrides**, nur dass **Permit** und **Deny** vertauscht sind.

first-applicable: Wertet die Kindelemente der Reihe nach aus. Das erste Ergebnis, das nicht **NotApplicable** ist, wird übernommen.

only-one-applicable: Wertet alle Kindelemente aus. Wenn alle zu **NotApplicable** evaluieren, ist das auch das Ergebnis. Wenn genau eins anwendbar ist, wird dessen Ergebnis übernommen. Ansonsten ist das Ergebnis **Indeterminate**, da die Grundannahme des Algorithmus, dass nur genau ein Kindelement anwendbar ist, verletzt ist.

In der Spezifikation ist die Funktionsweise der Algorithmen deutlich verwirrender beschrieben, kommt aber in fast allen Fällen zu den gleichen Ergebnissen. Dort, wo Unterschiede auftreten, ist XACML unterspezifiziert, beabsichtigt war ein Verhalten gemäß dieser Beschreibung.⁷

4.13 Auswertung einer Policy

Der Kombinationsalgorithmus einer $\langle \text{Policy} \rangle$ wird in dem XML-Attribut **RuleCombiningAlgId** angegeben, bei einer $\langle \text{PolicySet} \rangle$ heißt das Attribut **PolicyCombiningAlgId**. Der Algorithmus, der in diesem Attribut benannt ist, wird auf die Kindelemente angewandt. Die einzelnen Algorithmen sind in Abschnitt 4.12 beschrieben. In welcher Reihenfolge die Algorithmen die Kindelemente auswerten, und ob sie überhaupt ausgewertet werden, hängt vom jeweiligen Algorithmus ab.

4.14 Antwort des PDP

Nachdem der PDP die Policy ausgewertet hat, schickt er die Antwort mit dem Ergebnis an den Context Handler, der sie dann wiederum für den PEP aufbereitet. Auch in der

⁷Siehe <http://lists.oasis-open.org/archives/xacml-comment/200808/msg00024.html> sowie <http://lists.oasis-open.org/archives/xacml-comment/200809/msg00002.html>, „Issue S“.

Struktur der Antwort zeigt sich, dass XACML mit einem Schwerpunkt auf Flexibilität entworfen wurde.

Eine Antwort wird als XML-Element `<Response>` dargestellt, in dem sich ein oder mehr `<Result>`-Elemente befinden. Bei den meisten Anfragen enthält die Antwort genau ein Resultat, es gibt jedoch eine XACML-Erweiterung, die mehrere gebündelte Anfragen erlaubt.

Jedes `<Result>`-Element enthält die Entscheidung des PDP (`Permit`, `Deny`, `NotApplicable`, `Indeterminate`) sowie einen Status. Der Status hat einen Statuscode (`ok`, `missing-attribute`, `syntax-error`, `processing-error`) sowie optional eine ausführlichere Beschreibung.

4.15 Geschichte und aktuelle Entwicklung

Die erste offizielle Version von XACML wurde im Februar 2003 veröffentlicht. Dies ist auch die Version, die in dieser Arbeit vornehmlich behandelt wird. Version 2.0 erschien im Februar 2005. Die Änderungen zwischen den Versionen 1.0 und 2.0 sind in [XACMLdiff-1.x-2.0] beschrieben. Sie beziehen sich hauptsächlich auf die Erweiterungen von XACML („Profile“ genannt). Syntaktisch hat sich an den Policies, Anfragen und Antworten nur wenig geändert:

- Die Elemente `<anySubject>`, `<anyResource>` und `<anyAction>` wurden verworfen, weil sie überflüssig sind. Sie dienten dazu, in einem `<Target>` die einzelnen Abschnitte *nicht* einzuschränken.
- Verweise auf Policies und PolicySets haben ein neues Attribut, um die gewünschte Versionsnummer der referenzierten Policy festzulegen.
- In Antworten ist es nun möglich, eine explizite Fehlermeldung für fehlende Attribute zurückzugeben (`<MissingAttributeDetail>`).
- Zusätzlich zu den Gruppen `<Subject>`, `<Resource>` und `<Action>` gibt es innerhalb eines `<Target>`s nun die Möglichkeit, auch `<Environment>`-Attribute mit dem neu eingeführten Element `<EnvironmentMatch>` abzufragen. Vorher musste die entsprechende Abfrage in einer `<Condition>` formuliert werden.
- In Policies ist es möglich, Variablen zu definieren, um einfacher auf oft benötigte Ausdrücke zurückzugreifen.
- Es gibt zwei neue Datentypen (`dnsName` und `ipAddress`) und ein paar Funktionen dazu.

4.15.1 In Planung: XACML 3.0

In XACML 3.0 wird die syntaktische Struktur gründlich überarbeitet, indem Elemente, die syntaktisch gleich aufgebaut sind, auch die gleichen Namen tragen. So hat das `<Target>`-Element nicht mehr verschiedene Bereiche wie `<Subjects>`, `<Actions>`, sondern die

Elemente heißen jetzt entsprechend ihrer Funktionsweise nur noch $\langle \text{AnyOf} \rangle$ und $\langle \text{AllOf} \rangle$. Abgesehen davon bleibt die Struktur aber gleich, so dass ein $\langle \text{Target} \rangle$ immer noch die Form „Konjunktion von Disjunktion von Konjunktion von $\langle \text{Match} \rangle$ -Elementen“ hat.

Insgesamt wirkt XACML 3.0 sehr viel durchdachter und aufgeräumter als XACML 1.0. Dadurch, dass die URIs innerhalb des Standards stets ausgeschrieben werden, bleibt das Lesen des Standards anstrengend, trotz der vielen Verbesserungen.

4.16 Möglichkeiten und Grenzen

XACML stellt einen anwendungsunabhängigen Rahmen zur Verfügung, innerhalb dessen die Zugriffskontrolle implementiert werden kann. Zur Identifizierung von Subjekten, Ressourcen und Aktionen werden Attribute mit frei wählbaren Datentypen eingesetzt, die dadurch an jede Anwendung angepasst werden können. Die Policies bestehen aus Funktionsaufrufen, wie sie auch aus anderen Programmiersprachen bekannt sind. Alle in XACML vordefinierten Funktionen sind frei von Nebenwirkungen (referentiell transparent). Was jedoch fehlt, sind Ausdrücke, die den Kontrollfluss ändern, zum Beispiel Fallunterscheidungen (if) oder gar Schleifen (for, repeat, while). Dass keine Schleifen erlaubt sind, ist Absicht, um die Berechnungskomplexität der Policies niedrig zu halten. Eine Aufgabe, für die häufig for-Schleifen eingesetzt werden, ist das Iterieren über Mengen. Dieser Spezialfall ist in XACML über die Funktionen höherer Ordnung bereitgestellt, daher ist das Fehlen von Schleifen nicht ganz so kritisch.

XACML kennt keinen Funktionsaufruf, der einer Fallunterscheidung der Form „if a then b else c “ entspricht. Es ist jedoch teilweise möglich, solche Fallunterscheidungen mit Hilfe der anderen XACML-Konstrukte nachzubilden. Der Ausdruck $(a \wedge b) \vee (\neg a \wedge c)$ ist äquivalent zu der Fallunterscheidung, aber nur, wenn b und c vom Typ Boolean sind. Für andere Datentypen funktioniert das Verfahren nicht.

Innerhalb der Policies ist es dadurch schwierig, Zugriffsregeln, die stark auf Fallunterscheidungen basieren, zu implementieren. Abhilfe könnte eine benutzerdefinierte Funktion *ifelse* schaffen, die genau den fehlenden Bereich der Fallunterscheidungen abbildet.

Durch die Verwendung von XML zur Codierung von Policies, Anfragen und Antworten entsteht ein relativ großer Overhead, da die ansonsten einfachen Datenstrukturen zum Transport in XML umgewandelt und nach dem Transport zurückgewandelt werden müssen. Der Vorteil dieser Darstellung ist jedoch die Systemunabhängigkeit.

4.17 Zusammenfassung

In diesem Kapitel wurde das Zugriffskontrollmodell von XACML vorgestellt. Es wurde beschrieben, aus welchen Komponenten das Gesamtsystem zusammengesetzt ist und wie diese Komponenten zusammenarbeiten. Anschließend wurde die Struktur der Zugriffsanfragen detailliert vorgestellt. Eine XACML-Anfrage besteht aus einer Reihe von Attributen, die jeweils einer bestimmten Kategorie (Subjekt, Ressource, Aktion, Umgebung) zugeordnet sind.

Im Anschluss an die Anfragen wurde der Aufbau der XACML-Policys dargestellt. Die Hauptaufgabe einer Policy ist, zu einer bestimmten Anfrage eine Zugriffsentscheidung („erlaubt“, „verboten“, „nicht anwendbar“, „Fehler“) zu treffen. Es wurde detailliert beschrieben, wie diese Entscheidung getroffen wird und welche Datenstrukturen hierbei eine Rolle spielen. An dieser Stelle wurde speziell auf die Veränderungen zwischen den derzeitigen XACML-Versionen eingegangen, um die positive Entwicklung von XACML zu veranschaulichen.

Ein besonderes Merkmal von XACML ist die Flexibilität und Erweiterbarkeit, mit der die Policys definiert werden können. So ist es möglich, die Attributwerte aus den Anfragen mittels benutzerdefinierter Funktionen zu verarbeiten oder einzelne Regeln mit eigenen Algorithmen zu kombinieren, falls die vordefinierten nicht ausreichen.

Nun, da sowohl die Quell- als auch die Zielsprache des Übersetzers erläutert sind, kann die eigentliche Übersetzung in Kapitel 5 beginnen.

5 Übersetzung

In den vorherigen zwei Kapiteln wurden die Quell- und die Zielsprache des Übersetzers ausführlich vorgestellt. In diesem Kapitel wird ein Übersetzer entwickelt, der eine in OPL formulierte ORKA-Policy in eine äquivalente XACML-Policy umwandelt.

Dazu werden zunächst die Gründe für die Übersetzung erläutert und die Umgebung, in der die Übersetzung stattfindet, vorgestellt. Anschließend werden die einzelnen Module des ORKA-Modells jeweils für sich nach XACML übersetzt. Für jedes Modul entsteht beim Übersetzen eine XACML-Policy. Diese Policies werden anschließend zu einer Gesamtpolicy zusammengefügt, die die in Abschnitt 3.9 erwähnte Gesamtentscheidung trifft.

5.1 Gründe für die Übersetzung

Eins der Ziele von ORKA ist, die entwickelte Autorisierungslösung in ein ERP-System zu integrieren. Dafür ist es hilfreich, wenn das ORKA-System seine Dienste in einer wohldefinierten Schnittstelle nach außen präsentiert. Die meisten ERP-Systeme sollten so flexibel sein, dass sie auch direkt mit einem ORKA-System kommunizieren könnten, aber die Chance, dass solche Systeme schon Schnittstellen für die etablierten Standards anbieten, ist sehr viel höher. Aus diesem Grund ist es praktisch, aus dem ORKA-System heraus schon XACML zu unterstützen, damit der Anpassungsaufwand später minimiert wird.

5.2 Umfang der Übersetzung

Die einfachste Idee, den ORKA-PDP für einen XACML-PDP nutzbar zu machen, besteht darin, einen minimalen Wrapper in XACML zu formulieren, der direkt auf den ORKA-PDP zugreift und die Anfragen quasi nur weiterleitet. Eine andere Möglichkeit ist, den persistenten Datenbestand des ORKA-PDP vollständig in eine XACML-Policy zu übersetzen, so dass der ORKA-PDP im Autorisierungsprozess nur noch für die schnelllebigen Daten benutzt wird. Zwischen diesen beiden Extrema gibt es viele Möglichkeiten, die versuchen, die Vorteile beider Systeme zu vereinen. All diese Möglichkeiten haben ihre Vor- und Nachteile, die im folgenden gegenübergestellt werden.

5.2.1 Lösung 1: RBAC-Autorisierungsserver

Der XACML-PDP enthält eine minimale Policy, die Anfragen lediglich an den ORKA-PDP weiterleitet.

Vorteile:

- Der ORKA-PDP implementiert das zugrundeliegende RBAC-Modell in einer höheren Programmiersprache, daher kann man den Programmcode gut mit der Spezifikation vergleichen. In XACML ist das schwierig, da dort nur sehr grundlegende Datenstrukturen zur Verfügung stehen, nicht jedoch Kontrollstrukturen und mächtige Funktionsbibliotheken wie in typischen Programmiersprachen. Dadurch wird der Code, wenn er überhaupt möglich ist, länger als zum Verständnis erforderlich. Gerade in sicherheitskritischen Bereichen sollte der Code jedoch einfach zu lesen und verständlich sein.
- Wenn der ORKA-PDP mittels einer relationalen Datenbank implementiert ist, kann ein Administrator im Falle von Fehlern oder Merkwürdigkeiten direkt auf die Datenstrukturen zugreifen und sie analysieren.
- Dadurch, dass die Autorisierungsentscheidungen direkt aus der Datenbank ermittelt werden, gibt es keine Verzögerungen oder Cache-Probleme. So wirken sich beispielsweise Rollenänderungen sofort auf den Autorisierungsprozess aus und müssen dem XACML-PDP nicht erst mitgeteilt werden.
- Der Autorisierungsdienst kann Anfragen optimieren, so dass die zu verarbeitenden Datenmengen möglichst klein gehalten werden.

Nachteile:

- Der ORKA-PDP definiert eigene Schnittstellen und kann daher nicht direkt an andere Systeme angeschlossen werden. Um die Systeme zu verbinden, muss erst ein Wrapper programmiert werden, der die Anfragen und Antworten der Systeme ineinander übersetzt.

5.2.2 Lösung 2: XACML-Autorisierungsdienst

Immer, wenn an den persistenten Daten der ORKA-Policy etwas geändert wird, wird die geänderte Policy in XACML übersetzt und dem PDP zur Verfügung gestellt. Der PDP benutzt dann diese Policy zur Autorisierung.

Vorteile:

- Der ORKA-PDP wird entlastet, da alle persistenten Daten direkt im XACML-PDP zur Verfügung stehen und dort ausgewertet werden.
- Es existieren einige zuverlässige Implementierungen eines XACML-PDPs, der die Hauptarbeit übernimmt. Dadurch wird eine gewisse Zuverlässigkeit erreicht.

Nachteile:

- Die Policy ist doppelt vorhanden: einmal als ORKA-Policy und zusätzlich als XACML-Policy. Diese beiden Policys müssen synchron gehalten werden. Nach jeder Änderung der ORKA-Policy muss diese übersetzt und dem PDP zugänglich gemacht werden.

- Die Algorithmen des ORKA-PDP müssen komplett in XACML nachprogrammiert werden. Da XACML nicht so mächtig wie die üblichen Programmiersprachen ist, kann das dazu führen, dass einige der ORKA-Funktionen nicht in XACML abgebildet werden können. Für die in dieser Arbeit behandelten ORKA-Module war die Übersetzung möglich, sobald die Zugriffsfunktionen (vgl. Kapitel 3) komplexer werden, wird eine Umsetzung in XACML schwieriger bis unmöglich.

5.2.3 Praktische Umsetzung

Jede praktische Umsetzung wird sich irgendwo zwischen diesen beiden Extrema bewegen. Dabei ist für jede funktionale Komponente abzuwägen, ob sie sinnvoller in ORKA oder in XACML darstellbar ist. Nimmt man an, dass die Zugriffsanfragen an den XACML-PDP gestellt werden, so ist es sinnvoll, alle Funktionen, die effizient in XACML implementiert werden können, auch dort zu implementieren. Dadurch wird das ORKA-System entlastet, und auch der Kommunikationsaufwand zwischen dem XACML-PDP und dem ORKA-System sinkt. Die Funktionen jedoch, die in XACML schwer umzusetzen sind, sollten an das ORKA-System delegiert werden. Was das für die einzelnen ORKA-Module bedeutet, wird in den folgenden Abschnitten diskutiert.

5.3 Umgebung des Übersetzers

Für die Konzeption des Übersetzers ist es wichtig zu wissen, welche der benötigten Daten dem Übersetzer zur Verfügung stehen und welche nicht. Wenn die Daten verfügbar sind, ist es für die Implementierung wichtig, auf welche Weise sie verfügbar sind. Für die konzeptionelle Übersetzung reicht die Information, *ob* sie verfügbar sind.

Auf der Quellseite (ORKA-System) stehen dem Übersetzer die persistenten Daten aus allen ORKA-Modulen zur Verfügung. Aus diesen Daten sollen die XACML-Policys generiert werden. Die schnelllebigen Daten sind zur Übersetzungszeit nicht bekannt, da sie nicht langlebig genug sind, dass es sich lohnen würde, sie in die Policys einzubinden.

Der Übersetzer ist in Java geschrieben und kann daher direkt mit den ORKA-Daten arbeiten, die auch als Java-Objekte repräsentiert werden. Alternativ wäre es auch möglich, aus einer XML-codierten Policy die Daten zu extrahieren und dann zu übersetzen, ohne von den ORKA-Datenstrukturen abhängig zu sein. Es ist jedoch praktischer, diese Datenstrukturen direkt zu benutzen, da die „selbstgemachten“ Datenstrukturen sehr ähnlich, wenn nicht sogar identisch, wären.

5.4 Komponenten der ORKA-Policys

In Kapitel 3 wurden die zu übersetzenden ORKA-Module in ihrem Aufbau vorgestellt. Jetzt geht es darum, wie diese einzelnen Module in äquivalente XACML-Policys übersetzt werden können. Die einzelnen Module bestehen aus den folgenden Komponenten, die alle übersetzt werden müssen:

- Die **Datentypen** müssen in XACML-Datentypen umgewandelt werden. Falls mehrere ORKA-Datentypen in einen gemeinsamen XACML-Datentyp umgewandelt werden, dürfen dadurch keine Mehrdeutigkeiten entstehen; etwa wenn eine Rolle und ein Subjekt gleich benannt sind.
- Die **persistenten Daten** jedes Moduls werden fest in die Policy encodiert. Wie das genau abläuft, hängt von dem einzelnen Modul ab.
- Die **schnellebigen Daten** jedes Moduls können nicht in die Policy encodiert werden, da sie zur Übersetzungszeit nicht zur Verfügung stehen. Es muss daher ein Weg gefunden werden, diese Daten zur Auswertungszeit der Policy bereitzustellen. Diese Aufgabe übernimmt in XACML der Policy Information Point (PIP).
- Die **Zugriffsentscheidung** wird mit Hilfe der XACML-Standardfunktionen nachprogrammiert. In allen ORKA-Modulen, die in dieser Arbeit übersetzt wurden, war das ohne weitere Probleme möglich.

5.5 Übersetzung der ORKA-Datentypen

In der aktuellen Implementierung sind die ORKA-Datentypen aus den einzelnen Modulen folgenden Java-Klassen und XACML-Datentypen zugeordnet:

ORKA-Datentyp	Java-Klasse	XACML
RBAC-Core:USER	UserRef	string
RBAC-Core:ROLE	Role	string
RBAC-Core:SUBJECT	ActiveSubject	string
RBAC-Core:OBJECT	String	string
RBAC-Core:OPERATION	String	string
CC:ContextFunctionId	(entfällt)	(entfällt)
CC:ContextFunctionParameter	(entfällt)	(entfällt)
CC:ContextConstraint	(entfällt)	(entfällt)

Die elementaren Datentypen aus RBAC-Core werden in XACML alle durch den Datentyp *string* repräsentiert. Dabei muss sichergestellt sein, dass es nicht zu Verwechslungen kommt, falls Objekte von verschiedenen Typen mit dem gleichen String repräsentiert werden.

Die Datentypen aus CC können nicht so einfach übernommen werden, aber das ist auch gar nicht nötig. Da CCs im Grunde genommen Funktionsaufrufe sind, werden sie direkt auf die in XACML vordefinierten Funktionen übersetzt. Nähere Details finden sich im Abschnitt 5.9.

5.6 Struktur der erzeugten XACML-Policys

In den folgenden Abschnitten wird jeweils ein ORKA-Modul in eine XACML- $\langle$ Policy $\rangle$ übersetzt. So eine Policy besteht hauptsächlich aus zwei Teilen: dem Anwendungsbereich ($\langle$ Target $\rangle$) und einer Liste von Regeln. Jede der Regeln besteht aus dem Anwendungsbereich (wie schon bei der Policy), einem Effekt („Permit“ oder „Deny“) und weiteren Bedingungen ($\langle$ Conditions $\rangle$), die für diesen Effekt erfüllt sein müssen. Für jede der erzeugten Policys müssen diese Eigenschaften festgelegt werden. Bei Stellen, an denen es mehrere Möglichkeiten der Implementierung gibt, wird auf die verschiedenen Möglichkeiten und deren Auswirkungen eingegangen.

Das Hauptziel der Übersetzung ist Korrektheit und Vollständigkeit. Eins der Nebenziele ist, dass ein PDP die Policys effizient auswerten kann. Für die Effizienz sind einige Designentscheidungen hilfreich:

- Der Anwendungsbereich einer Policy ($\langle$ Target $\rangle$) sollte möglichst einfach zu berechnen sein, damit die Entscheidung, ob eine Policy überhaupt anwendbar ist, schnell durchgeführt werden kann.
- Der Anwendungsbereich einer Policy sollte möglichst eng definiert sein, damit der Rest der Policy nur ausgewertet wird, wenn eine gute Chance besteht, dass sie überhaupt anwendbar ist.
- Um den Arbeitsspeicher zu schonen, sollte die Policy möglichst ohne Redundanz codiert werden. Das heißt insbesondere, dass mehrfache gleiche Ausdrücke innerhalb der Policys vermieden werden.

Um diese Anforderungen umzusetzen, muss man beachten, welchen syntaktischen Einschränkungen eine XACML-Policy unterliegt. Hierbei spielt insbesondere das $\langle$ Target $\rangle$ -Element eine Rolle. Dessen Struktur ist, wie schon in Abschnitt 4.8 erläutert, „Konjunktion von Disjunktionen von Konjunktionen von $\langle$ *Match $\rangle$ -Elementen“.

Für die erzeugte Policy müssen daher folgende Fragen beantwortet werden:

- Wie sind die einzelnen Regeln für die Policy aufgebaut?
- Mit welchem Algorithmus werden die Regeln kombiniert?

5.7 RBAC-Core

Um die Policy zu übersetzen, muss die schon in Abschnitt 3.3 erwähnte Funktion in XACML implementiert werden:

```
def checkAccess(s, op, ob):
    roles = SR.lookup(s)
    perms = PA.lookup(roles)
    return perms.contains((op, ob))
```

Aus dieser Darstellung geht hervor, dass die Relationen U , R , P , UA , S und SU für die Übersetzung von RBAC-Core-Policys irrelevant sind. Sie könnten höchstens dazu eingesetzt werden, die Invarianten der Policy¹ bei jedem Zugriff noch einmal zu kontrollieren, aber das ist nicht sinnvoll, da die Policy sich nicht mehr ändert, sobald sie einmal übersetzt wurde, und daher die Prüfung immer zum gleichen Ergebnis kommen müsste. Es ist effizienter, die Prüfung einmal vor oder während der Übersetzung durchzuführen.

Die eigentliche XACML-Policy benötigt also nur SR und PA als Eingabe. Da die Relation SR schnelllebig ist, kann sie nicht in der Policy selbst codiert werden. Die darin enthaltenen Informationen müssen daher während der Zugriffsentscheidung vom PIP abgefragt werden.

Als Eingabe für die Übersetzung bleibt somit nur noch die dreistellige Relation PA , die aus Rolle, Operation und Objekt besteht. Für die rollenbasierte Zugriffskontrolle bietet XACML mit der Erweiterung [XACML-RBAC] das nötige Gerüst, um beliebige Zugriffsregeln zu implementieren. Diese Erweiterung wird jedoch in dieser Arbeit nicht verwendet, da darin eine bestimmte Implementierungsmethode vorgeschrieben wird, die für diese Arbeit unpassend ist. Stattdessen wird ein eigenes XACML-Attribut (`orka:role`) für Rollen definiert.

Es gibt viele Möglichkeiten, die Relation PA in einer XACML-Policy zu codieren. Einige werden im folgenden vorgestellt.

5.7.1 Einfache Implementierung

Die einfachste Struktur für eine Policy, die die Relation PA abbildet, ist die folgende, aufgrund der Klarheit des Codes in Scheme ausgedrückt²:

```
(define (check-access role op ob)
  (or
    (and (string-equal role "role1") (string-equal op "op1") (string-equal ob "ob1"))
    (and (string-equal role "role2") (string-equal op "op2") (string-equal ob "ob2"))
    (and (string-equal role ... ) (string-equal op ... ) (string-equal ob ... ))
    (...)))
```

Die Struktur dieser Funktion lässt sich gut in XACML umwandeln. Die einzelnen Aufrufe von *string-equal* können jeweils in ein `<*Match>`-Element (Abschnitt 4.9) übersetzt werden. Matches können als Kinder der Elemente `<Subject>`, `<Action>` und `<Resource>` auftauchen und werden dort konjunktiv verknüpft, was zu dem **and** passt. Allerdings hat XACML die Beschränkung, dass alle Kindelemente an dieser Stelle aus der gleichen Kategorie (Subjekt, Ressource, Aktion) kommen müssen, und die drei `<*Match>`es kommen aus jeweils unterschiedlichen Kategorien.

Die **and**-Knoten müssen also in ein anderen XACML-Element übersetzt werden. Die nächsthöhere Ebene ist `<Subjects>`, `<Resources>` und `<Action>`, aber die verknüpfen ihre Kinder disjunktiv. Noch eine Ebene höher liegt das `<Target>`, und das ist eine Konjunktion,

¹ORKA spezifiziert für jedes Policy-Modul Invarianten bezüglich der Datenstrukturen. Darauf wird in dieser Arbeit jedoch nicht weiter eingegangen.

²Aus ästhetischen Gründen heißt die Scheme-Funktion *string=?* hier *string-equal*.

deren Elemente aus unterschiedlichen Kategorien kommen dürfen. Ein **and**-Knoten kann also in ein `<Target>` übersetzt werden.

Die **or**-Knoten müssen in ein XACML-Element übersetzt werden, das ein `<Target>` als Kind hat und seine Kinder disjunktiv verknüpft. Das einfachste dieser Elemente ist `<Rule>`, und durch die Flexibilität der Kombinationsalgorithmen (vgl. Abschnitt 4.12) sind an dieser Stelle sowohl Disjunktionen als auch Konjunktionen möglich. Eine Disjunktion kann umgesetzt werden, indem jeder **or**-Knoten in eine Regel mit dem Effekt **Permit** übersetzt wird, eine Fallback-Regel mit dem Effekt **Deny** hinzugefügt wird und diese Regeln mit dem Algorithmus **permit-overrides** oder **first-applicable** kombiniert werden. Die beiden Algorithmen führen zum gleichen Ergebnis. In einer solchen Situation sollte jedoch **permit-overrides** vorgezogen werden, da hier die Auswertungsreihenfolge nicht vorgeschrieben ist und dadurch die Auswertung optimiert werden kann (vgl. Abschnitt 4.8.1).

Der Vorteil dieses Verfahrens ist die einfache Struktur der erzeugten Policy. Die folgenden Verfahren sind nur kleine Modifikationen gegenüber diesem einfachen Verfahren.

5.7.2 Ausklammern einzelner Teile

Um den Rechenaufwand etwas zu verringern, kann man sich auch dazu entscheiden, jeweils die Rolle eines Tupels „vor die Klammer zu ziehen“ und so die mehrfachen Vergleiche zu sparen. Das ist besonders nützlich, wenn die Rollen jeweils viele Berechtigungen haben. Die analoge Funktion sieht dann so aus:

```
(define (check-access role op ob)
  (or
    (and (string-equal role "role1")
      (or
        (and (string-equal op "op1a") (string-equal ob "ob1a"))
        (and (string-equal op "op1b") (string-equal ob "ob1b"))
        (and (string-equal op ... ) (string-equal ob ... ))))
    (and (string-equal role "role2")
      (or
        (and (string-equal op "op2a") (string-equal ob "ob2a"))
        (and (string-equal op "op2b") (string-equal ob "ob2b"))
        (and (string-equal op ... ) (string-equal ob ... ))))
    (and ... )
    ... ))
```

Alternativ kann man auch die Operation und/oder das Objekt vor die Klammer ziehen. Das ist besonders dann günstig, wenn die Berechtigungen für viele Rollen gleich sind.

Die Darstellung dieser optimierten Form wird in XACML relativ komplex gegenüber der einfachen Variante. Dort wurde der äußere **or**-Knoten schon in eine Regel übersetzt, und der optimierte Scheme-Ausdruck hat noch eine Ebene **or** und **and** mehr. Die äußeren **and**-Knoten müssen also jeweils in eine `<Policy>` übersetzt werden und die äußeren **or**-Knoten in eine `<PolicySet>`. Dadurch wird die Lesbarkeit der Policies schnell eingeschränkt.

5.7.3 Bewertung der Varianten

Die „Einfache Implementierung“ hat den Vorteil, dass sie einfach umzusetzen ist und man jeden Eintrag in der *PA*-Relation schnell im XACML-Code wiederfinden kann. Der Nachteil dagegen ist, dass die gleichen Attribute mehrfach verglichen werden. Andererseits entspricht die erzeugte XACML-Policy genau den Empfehlungen, die im XACML-Standard bezüglich möglicher Optimierungen angegeben sind, [XACML-1.0, A.12] so dass die begründete Hoffnung besteht, dass eine gute PDP-Implementierung die doppelten Vergleiche wegoptimiert.

Das „Ausklammern einzelner Teile“ hat den Vorteil, dass die Verarbeitungsgeschwindigkeit selbst in nicht-optimierten PDPs steigt, da viele der Ausdrücke nur noch einmal ausgewertet werden müssen. Dafür stehen im erzeugten XACML-Code die einzelnen Elemente der *PA*-Relation nicht mehr so dicht beisammen, so dass die Lesbarkeit und Übersichtlichkeit der Policies darunter leidet. Außerdem passt sich die erzeugte XACML-Policy nicht mehr so nahtlos in das Schema ein, das die Entwickler von XACML sich ausgedacht haben. Dadurch, dass die einzelnen Attribute auf mehrere Ebenen verteilt sind, wird es dem PDP unnötig erschwert, die Policies zu optimieren.

Die Vorteile der einfachen Implementierung überwiegen bei weitem die Nachteile. Insbesondere für optimierende PDPs ist sie sehr gut geeignet, da mit relativ einfachen Mitteln (Zusammenfassung gleicher Teilausdrücke, um den Speicherverbrauch zu verringern, sowie Hashtabellen, um direkte Vergleiche der Attribute zu sparen) schon beachtliche Verbesserungen möglich sind.

5.7.4 Konkrete Übersetzung

Für jedes Vergleichstripel (r, op, ob) wird eine Regel der folgenden Form generiert, wobei die Bezeichner mit „\$“ durch die entsprechenden Elemente des Tripels ersetzt werden.

```

<Rule Effect="Permit">
  <Target>
    <Subjects>
      <Subject>
        <SubjectMatch MatchId="string-equal">
          <AttributeValue>$r</AttributeValue>
          <SubjectAttributeDesignator AttributId="orka:role"/>
        </SubjectMatch>
      </Subject>
    </Subjects>
    <Resources>
      <Resource>
        <ResourceMatch MatchId="string-equal">
          <AttributeValue>$ob</AttributeValue>
          <ResourceAttributeDesignator AttributId="resource-id"/>
        </ResourceMatch>
      </Resource>
    </Resources>
  </Target>
</Rule>

```

```

</Resources>
<Actions>
  <Action>
    <ActionMatch MatchId="string-equal">
      <ActionAttributeDesignator Attributeld="action-id"/>
      <AttributeValue>$op</AttributeValue>
    </ActionMatch>
  </Action>
</Actions>
</Target>
</Rule>

```

Listing 5.1: XACML-Fragment für ein RBAC-Core-Tripel

Hier zeigt sich, wie schreibaufwendig XACML gegenüber der Darstellung in Scheme ist. Statt einer einzigen Zeile werden hier 28 Zeilen benötigt.

Alle diese so erzeugten Regeln müssen nun noch zu einer Policy kombiniert werden. Die Policy besteht aus dem Anwendungsbereich und den Regeln. Der Anwendungsbereich der Policy ist nochmal etwas komplizierter, der Regelteil ist dagegen sehr einfach.

Der Anwendungsbereich der Policy kann für RBAC-Core sinnvoll eingeschränkt werden, und zwar mit den folgenden Bedingungen:

- Die Rolle der Anfrage muss einer der Rollen aus *PA* entsprechen.
- Die Aktion der Anfrage muss einer der Operationen aus *PA* entsprechen.
- Die Ressource der Anfrage muss einem der Objekte aus *PA* entsprechen.

Dadurch ergibt sich der folgende Code für das `<Target>`-Element der Policy:

```

<Target>
  <Subjects>
    <Subject>
      <SubjectMatch MatchId="string-equal">
        <SubjectAttributeDesignator Attributeld="orka:role">
          <AttributeValue>$r</AttributeValue>
        </SubjectMatch>
      </Subject>
    <Subject>
      ...
    </Subject>
  </Subjects>
  <Resources>
    <Resource>
      <ResourceMatch .../>
    </Resource>
  </Resources>

```

```

...
</Resource>
</Resources>
<Actions>
...
</Actions>
</Target>

```

Listing 5.2: Anwendungsbereich einer RBAC-Core-Policy

Die einzelnen `<SubjectMatch>`-Elemente müssen disjunktiv verknüpft werden. Dazu werden sie jeweils einzeln in ein `<Subject>`-Element gesteckt, da diese disjunktiv verknüpft werden. Das gleiche Verfahren wird auch auf `<ResourceMatch>` und `<ActionMatch>` angewendet.

Die gesamte Policy sieht dann wie folgt aus, wobei $r \dots$ die Liste aller Regeln ist, die oben erzeugt wurden:

```

<Policy RuleCombiningAlgorithmId="permit-overrides">
  $Target
  $r...
</Policy>

```

Listing 5.3: XACML-Policy für RBAC-Core

5.8 RBAC mit Rollenhierarchie

Die Übersetzung von RBAC mit Rollenhierarchie ist trivial. Da bei der Einführung der Rollenhierarchie nur die dynamische Relation SR geändert wird, muss auch nur der dynamische Teil der Zugriffskontrolle angepasst werden. Die Policy wird daher genau wie bei RBAC-Core übersetzt, mit einer kleinen Änderung: Statt des Attributs `orka:role` wird das Attribut `orka:transitive-role` abgefragt. Die Hauptaufgabe, nämlich das Bereitstellen der Relation RH , wird vom PIP erledigt.

5.9 Context Constraints

Die Übersetzung der Context Constraints (CC) ist komplizierter als RBAC-Core. Während bei RBAC-Core nur eine einfache Relation in XACML codiert wurde, kommen bei den CCs externe Bedingungen dazu, die in ORKA durch Funktionsaufrufe umgesetzt sind. Jeder dieser Funktionsaufrufe ist in einem benannten CC gekapselt, und dadurch gibt es zwei Möglichkeiten der Umsetzung.

1. Von den CCs bekommt der XACML-PDP lediglich die IDs zu sehen. Diese ID gibt er dann an eine benutzerdefinierte Funktion (etwa `orka:eval-cc`) weiter, die die Hauptarbeit erledigt.

2. Die Funktionsaufrufe innerhalb der CCs werden in äquivalente XACML-Funktionsaufrufe übersetzt, soweit das die Sprachdefinition von XACML zulässt.

Die 1. Möglichkeit führt dazu, dass pro CC, der in einer Anfrage enthalten ist, der XACML-PDP genau eine Rückfrage an den ORKA-PDP stellen muss. Der ORKA-PDP wertet dann den CC aus, und gibt `true`, `false` oder `Indeterminate` zurück.

Die 2. Möglichkeit führt dazu, dass pro Variable, die in einem CC vorkommt, der XACML-PDP eine Rückfrage an den ORKA-PDP stellen muss. In vielen Fällen enthält ein CC mehr als eine Variable, so dass der Kommunikationsaufwand steigt.

Für eine Umsetzung in einem realen System ist die 1. Möglichkeit vorzuziehen, da sie effizienter ist. Der XACML-PDP hat dann nur die Aufgabe, ein standardisiertes Bindeglied zwischen dem ORKA-PDP und dem Rest des Systems zu sein. Wichtige Berechnungen finden weiterhin im ORKA-PDP statt.

In dieser Arbeit wird die 2. Möglichkeit umgesetzt, da das Hauptziel von ORKA nicht eine effiziente Implementierung ist, sondern zu untersuchen, wie gut sich der ORKA-PDP in bestehende Zugriffskontrollsysteme integrieren lässt, und wie weit XACML dafür verwendet werden kann. Hierbei wird möglichst viel von den anfallenden Aufgaben innerhalb des XACML-PDP gelöst, der Rest wird dem ORKA-PDP überlassen.

5.9.1 Übersetzung der einzelnen Parameter

Bei der Übersetzung der Funktionsparameter müssen zwei Fälle unterschieden werden:

- Der Parameter repräsentiert einen direkten Wert (Literal).
- Der Parameter ist „dynamisch“ und muss daher erst von einem sog. Context-Provider ausgewertet werden.

Parameter mit literalen Werten

Der erste Fall ist trivial, da es für jeden ORKA-Datentyp (*int*, *string*, *date*, *time*) einen entsprechenden XACML-Datentyp gibt und die Literale weitgehend gleich codiert werden. Bei der Repräsentation der Literale ist XACML etwas strenger als ORKA, deshalb müssen einige Literale erst normalisiert werden, bevor sie von XACML akzeptiert werden. Ein Beispiel dafür sind Zeitpunkte, die in ORKA üblicherweise nur auf die Minute genau formuliert werden (etwa 11:22), in XACML jedoch immer auf Sekunden genau angegeben werden (also 11:22:00).

Dynamische Parameter

Der zweite Fall ist schon anspruchsvoller, da die Ausdrücke, die an die Context Provider gegeben werden, in der Mächtigkeit und der Syntax etwa denen aus Java entsprechen. Beispiele dafür sind in Abschnitt 3.5.2 aufgelistet. Es stellt sich die Frage, ob diese Ausdrücke von XACML-PDP ausgewertet werden können, oder ob diese Aufgabe an das ORKA-System ausgelagert wird. Bei einigen dieser Ausdrücke ist es möglich, sie direkt

in entsprechende XACML-Funktionen zu übersetzen, bei vielen anderen bleibt nur die Möglichkeit, die Ausdrücke so wie sie sind in der XACML-Policy zu codieren und dann an eine spezielle ORKA-Auswertungsfunktion zu übergeben.

Alle Parameter, die sich auf irgendeine Weise auf ORKA-Daten beziehen, müssen sowie so vom ORKA-PDP evaluiert werden, da eine (zusätzliche) Implementierung in XACML aufwendig und schwerfällig ist. Schon das Parsen der Ausdrücke würde Funktionen zur Stringmanipulation erfordern, die es in XACML nicht gibt. Die wenigen Ausnahmen, die direkt in XACML ausgewertet werden können, sind Parameter, die keinen Bezug zu den ORKA-Daten haben, zum Beispiel die aktuelle Uhrzeit. Aber auch in diesem Fall ist es sinnvoll, den Parameter vom ORKA-PDP interpretieren zu lassen, damit die Uhrzeit innerhalb aller Komponenten gleich interpretiert wird und nicht etwa dadurch verfälscht wird, dass ORKA und XACML die Uhrzeit in bezug auf verschiedene Zeitzonen interpretieren.

Letztendlich bringt es also keinen Vorteil, einen Teil der dynamischen Parameter nach XACML zu übersetzen. Deshalb werden sie einfach nur codiert in der XACML-Policy gespeichert und dann mittels einer ORKA-Komponente evaluiert.

Jeder Parameter hat einen zur Übersetzungszeit feststehenden Datentyp, der bei den dynamischen Parametern jedoch immer *context* ist, also keinerlei Informationen über den repräsentierten Wert preisgibt. Der „echte“ Datentyp eines Parameters kann zwar während der Policy-Auswertung bestimmt werden, nicht jedoch zur Übersetzungszeit. Das liegt daran, dass der Context-Provider (rein theoretisch) bei jeder Evaluierung des Parameters einen anderen Wert oder gar einen Wert mit anderem Datentyp zurückgeben könnte. Da so ein Verhalten im allgemeinen jedoch dazu führt, dass das Autorisierungssystem an sich schwieriger zu verstehen ist, kann man davon ausgehen, dass so ein Fall selten auftritt. Unter der Annahme, dass sich der Datentyp nicht ändert, könnte man also schon zur Übersetzungszeit den Parameter „probeweise“ evaluieren und einfach den Datentyp des zurückgegebenen Werts übernehmen.

Der Übersetzer hat zur Übersetzungszeit jedoch keinen Zugriff auf den Context-Provider. Um den Datentyp eines Parameters zu bestimmen, müssen daher andere Quellen benutzt werden. Die Übersetzung eines Funktionsaufrufs erfolgt jeweils durch spezialisierte Übersetzer, die die Eigenarten der zu übersetzenden Funktionen kennen. So ist zum Beispiel klar, dass die Funktion *time-between* drei Parameter vom Typ *time* hat, und dass die Vergleichsfunktionen wie *equal* bzw. *more-than* nur für Parameter mit jeweils gleichem Typ sinnvoll sind. Das heißt, der fehlende Datentyp eines Parameters kann in einigen Fällen durch die Datentypen der anderen Parameter bestimmt werden. Es gibt aber auch Fälle, in denen das nicht möglich ist. Ein Aufruf der Funktion *equal* mit zwei Parametern vom Typ *context* kann nicht übersetzt werden, da zum Zeitpunkt der Übersetzung nicht bestimmbar ist, ob die ORKA-Funktion *equal* in die XACML-Funktion *string-equal* oder *int-equal* übersetzt werden soll. Eine mögliche Abhilfe (abgesehen von der Behebung des oben schon erwähnten Designfehlers) wäre, in XACML benutzerdefinierte Funktionen *orka-equal*, *orka-greater-than* u. s. w. einzuführen, aber das behebt nur die Symptome, nicht jedoch das eigentliche Problem. Übersetzungen in andere Sprachen hätten dann immer noch mit dem gleichen Problem zu kämpfen, so dass dies keine langfristige Lösung darstellt.

```

<context_constraint cc_id="cc:cc1">
  <context_function_id id="equal-or-less-than"/>
  <context_function_parameters>
    <parameter value="cst.getCustomerExternalRating(sCtx.getProcessVariable(parameters.
      processContextId,str.customerId))" type="context" />
    <parameter value="100000" type="int"/>
  </context_function_parameters>
</context_constraint>

```

Listing 5.4: Beispiel für einen zu übersetzenden CC

Letztendlich können also alle Parameter, sowohl Literale als auch dynamische, übersetzt werden, mit der Ausnahme, dass ein Doppel-*context* nicht bei den Vergleichsfunktionen auftauchen darf. In so einem Fall wird die Übersetzung mit einer aussagekräftigen Fehlermeldung abgebrochen.

5.9.2 Übersetzung der Funktionsaufrufe

Bei der Übersetzung der Funktionsaufrufe ist folgendes zu beachten:

- In ORKA gibt es generische Funktionen (z. B. *equals*, *not-equals*), die auf beliebige Datentypen anwendbar sind. In XACML dagegen gibt es spezialisierte Funktionen für jeden Datentyp. Daher muss schon zur Übersetzungszeit feststellbar sein, von welchem Datentyp die Parameter eines Funktionsaufrufs sind. Wenn der Datentyp des Parameters direkt angegeben ist, ist das einfach, wenn als Datentyp jedoch „context“ angegeben ist, muss zuerst der „echte“ Datentyp bestimmt werden, damit die richtige XACML-Funktion ausgewählt werden kann.
- Es gibt keine definitive Liste der ORKA-Funktionen. Daher können nur einige wenige Funktionen, die schon im restlichen ORKA-Quelltext erwähnt werden, direkt übersetzt werden. Alle anderen führen zu einer Fehlermeldung bei der Übersetzung.

Aber abgesehen davon ist die Übersetzung von Funktionsaufrufen relativ einfach: Zuerst werden die Parameter einzeln übersetzt, dann wird die zur ORKA-Funktion passende XACML-Funktion herausgesucht, und aus diesen Ergebnissen dann ein XACML-Funktionsaufruf zusammengesetzt. Die Fallstricke, die bei der konkreten Übersetzung aufgetaucht sind, werden im nächsten Kapitel, Abschnitt 6.5.3 beschrieben.

Ein Beispiel für die Übersetzung ist in den Listings 5.4 und 5.5 angegeben.

5.9.3 Erzeugen der XACML-Policys

Nun, da jeder CC für sich in ein XACML-*<Apply>*-Element übersetzt ist, müssen diese Elemente noch zu einer Policy kombiniert werden. Für die Funktion zur Zugriffsentscheidung müssen folgende Fragen beantwortet werden:

```

<Apply FunctionId="integer-greater-than">
  <Apply FunctionId="integer-one-and-only">
    <EnvironmentAttributeDesignator Attributeld="orka:cfparam#cst.
      getCustomerExternalRating%28sCtx.getProcessVariable%28parameters.
        processContextId%2Cstr.customerId%29%29"/>
  </Apply>
  <AttributeValue>100000</AttributeValue>
</Apply>

```

Listing 5.5: Der übersetzte CC in XACML

- Welche CCs gehören zu den gerade aktiven Rollen?
- Welche CCs gehören zu einer Kombination aus gerade aktiven Rollen und der gewünschten Berechtigung?
- Welche CCs gehören zu der gewünschten Berechtigung?
- Sind alle oben genannten CCs erfüllt?

Alternativ kann auch direkt geprüft werden, ob alle diese CCs erfüllt sind.

Um den generierten Code kurz zu halten, wäre es ganz praktisch gewesen, für jeden der CCs eine Variable zu definieren, um den Code für die einzelnen CCs nicht mehrfach ausschreiben zu müssen. Aber Variablen innerhalb von Policys wurden erst in XACML 2.0 eingeführt, und ORKA verwendet XACML 1.0. Daher bleibt nur die Möglichkeit, jeden CC bei jeder Verwendung vollständig auszuschreiben.

An dieser Stelle ist es wichtig, sich die *genaue* Definition der Entscheidungsfunktion in Erinnerung zu rufen, daher ist sie hier nochmal angeführt:

```

def checkAccess(s, op, ob):
    roles = SR.lookup(s)
    perms = PA.lookup(roles)
    return ((op, ob) in perms) \
        and checkCCs(RCC.lookup(roles)) \
        and checkCCs(PACC.lookup((roles, (op, ob)))) \
        and checkCCs(PCC.lookup(perms))

```

Bei oberflächlichem Betrachten des Codes könnte man auf die Idee kommen, das Übersetzungsschema von RBAC-Core zu übernehmen (vgl. Abschnitt 5.7) und in jede *<Rule>* eine zusätzliche Bedingung einzubauen, die die jeweils anwendbaren CCs prüft. Diese Vorgehensweise ist jedoch falsch, da in der Definition des CC-Moduls explizit gefordert wurde, dass auch die CCs von eigentlich unbeteiligten Rollen anwendbar sind (vgl. Abschnitt 3.5). Genau diese Forderung würde jedoch mit dieser Implementationsidee nicht erfüllt.

Die korrekte Lösung muss so aussehen, dass einerseits die Bedingung aus RBAC-Core, so wie sie in Abschnitt 5.7 definiert ist, geprüft wird, und andererseits jede Verletzung eines anwendbaren CC zum Ergebnis **Deny** führt.

Die Policy, die letztendlich erzeugt wird, hat damit die folgende Form:

```
<PolicySet PolicyCombiningAlgId="first-applicable">
  <Target>
    ...
  </Target>
  <Policy PolicyId="check-applicable-ccs" RuleCombiningAlgId="deny-overrides">
    ...
  </Policy>
  <PolicyIdReference>orka:rbaccore</PolicyIdReference>
</PolicySet>
```

Die Policy, die die CCs auswertet, gibt entweder **Deny** oder **NotApplicable** zurück. Die RBAC-Core-Policy wird nur in letzterem Fall ausgewertet, da das Ergebnis ansonsten schon feststeht. Die Wahl der unterschiedlichen Kombinationsalgorithmen wurde mit Blick auf die Optimierbarkeit der Anfragen vorgenommen: Wann immer möglich, werden Algorithmen eingesetzt, bei denen die Auswertungsreihenfolge irrelevant ist.

Die Policy aus dem Modul RBAC-Core wird über eine Verknüpfung eingebunden (`<PolicyIdReference>`), um Arbeitsspeicher und Leseaufwand zu sparen. Für die Übersetzung der CCs muss daher nur noch die Policy **check-applicable-ccs** entwickelt werden. Dazu werden die Relationen *RCC*, *PACC* und *PCC* jeweils einzeln in XACML-`<Rule>`s übersetzt und dann zu der gewünschten Policy kombiniert.

Übersetzung der Relation *RCC*

Die Relation *RCC* besteht aus Elementen der Form (r, cc) . Ein Zugriff wird verweigert, wenn das Subjekt der Rolle *r* zugeordnet ist, der dazugehörige CC *cc* aber nicht erfüllt ist. In Scheme sieht diese Formulierung so aus:

```
(define (check-rcc role)
  (cond
    ((and (string-equal role "role1") (not (eval-cc rcc1))) 'Deny)
    ((and (string-equal role "role2") (not (eval-cc rcc2))) 'Deny)
    ((and (string-equal role ... ) (not (eval-cc ... ))) 'Deny)
    (else 'NotApplicable)))
```

Jede Zeile dieses Codes wird in eine XACML-`<Rule>` der folgenden Form übersetzt, wobei `$r` die Rolle ist und `$cc` der CC in Form eines `<Condition>`-Elements.

```
<Rule Effect="Deny">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>$r</AttributeValue>
```

```

    <SubjectAttributeDesignator Attributeld="orka:role"/>
  </SubjectMatch>
</Target>
<Condition FunctionId="not">
  $cc
</Condition>
</Rule>

```

Übersetzung der Relation *PACC*

Die Übersetzung von *PACC* verläuft ähnlich wie die von *RCC*, nur dass bei *PACC* zusätzlich die Attribute **resource-id** und **action-id** abgefragt werden. In Scheme sieht das so aus:

```

(define (check-pacc role op ob)
  (cond
    ((and (string-equal role "role1") (string-equal op "op1") (string-equal ob "ob1")
          (not (eval-cc rcc1)))) 'Deny)
    ((and (string-equal role "role2") (string-equal op "op2") (string-equal ob "ob2")
          (not (eval-cc rcc2)))) 'Deny)
    ((and (string-equal role ... ) (string-equal op ... ) (string-equal ob ... )
          (not (eval-cc ... )))) 'Deny)
    (else 'NotApplicable)))

```

Die Übersetzung nach XACML verläuft genau so geradlinig wie schon bei *RCC*:

```

<Rule Effect="Deny">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>$r</AttributeValue>
      <SubjectAttributeDesignator Attributeld="orka:role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>$ob</AttributeValue>
      <ResourceAttributeDesignator Attributeld="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>$op</AttributeValue>
      <ActionAttributeDesignator Attributeld="action-id"/>
    </ActionMatch>
  </Target>
  <Condition FunctionId="not">
    $cc
  </Condition>
</Rule>

```

Übersetzung der Relation *PCC*

Die Übersetzung von *PCC* entspricht *PACC*, nur dass das Subjekt nicht am Vergleich teilnimmt. In Scheme:

```
(define (check-pcc op ob)
  (cond
    ((and (string-equal op "op1") (string-equal ob "ob1") (not (eval-cc rcc1))) 'Deny)
    ((and (string-equal op "op2") (string-equal ob "ob2") (not (eval-cc rcc2))) 'Deny)
    ((and (string-equal op ... ) (string-equal ob ... ) (not (eval-cc ... ))) 'Deny)
    (else 'NotApplicable)))
```

Die entsprechende XACML-*<Rule>* sieht so aus:

```
<Rule Effect="Deny">
  <Target>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>$ob</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>$op</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
  <Condition FunctionId="not">
    $cc
  </Condition>
</Rule>
```

Kombination der drei Relationen

Die drei Relationen *RCC*, *PACC* und *PCC* wurden jeweils in Listen von Regeln übersetzt. Diese Regeln müssen nun noch zu einer Policy kombiniert werden. Die Auswertungsreihenfolge ist dabei egal, daher kann der Algorithmus **deny-overrides** verwendet werden. Die XACML-Policy sieht dann so aus, wobei *\$r* die Liste der Regeln ist:

```
<Policy PolicyId="check-applicable-ccs" RuleCombiningAlgId="deny-overrides">
  <Target/>
  $r...
</Policy>
```

Diese Policy kann jetzt in die oben erwähnte Gesamtpolicy eingefügt werden und erfüllt genau die Spezifikation des ORKA-Moduls „Context Constraints“.

5.10 Separation of Duty

Die Übersetzung des in Abschnitt 3.6 vorgestellten Moduls ist trivial. Da sich an der Zugriffsfunktion und allen daran beteiligten Daten gegenüber RBAC-Core nichts ändert, ist die generierte Policy mit der aus RBAC-Core (Abschnitt 5.7) identisch.

5.11 Separation of Duty mit Rollenhierarchie

Für dieses Modul gilt das gleiche wie für „Separation of Duty“, nur dass hier die generierte Policy aus dem Modul „RBAC mit Rollenhierarchie“ übernommen wird.

5.12 Kombination der Module

Letztendlich soll bei der Übersetzung als Ergebnis eine einzige Policy entstehen. Bisher wurden jedoch nur für die einzelnen Module Policies erzeugt. Das Endergebnis der Zugriffsentscheidung ist abhängig von den Ergebnissen der einzelnen Module (und nur davon). Im letzten Schritt muss daher eine Policy erzeugt werden, die die Einzelergebnisse der anderen Policies kombiniert.

Für diesen Zweck bietet XACML einige vordefinierte Algorithmen an, zum Beispiel **deny-overrides**, **permit-overrides**, **first-applicable**. Eine naheliegende Idee ist daher, die Teilergebnisse aller Module mit dem Algorithmus **deny-overrides** zu verknüpfen, um nicht aus Versehen Zugriffe zu erlauben, die eigentlich verboten sein sollen. So gut das auch klingt, in der Praxis führt es zu falschen Ergebnissen. Denn durch das Modul „RBAC mit Rollenhierarchie“ werden einige Zugriffe absichtlich erlaubt, die das Modul „RBAC-Core“ verbietet, nämlich all diejenigen, die sich auf indirekte Rollen beziehen. Das zusammengefasste Ergebnis muss daher wie folgt aussehen:

- Das Teilergebnis von RBAC-Core wird ignoriert, falls das Modul „RBAC mit Rollenhierarchie“ in der zu übersetzenden Policy vorkommt.
- Wenn „RBAC mit Rollenhierarchie“ das Ergebnis „Deny“ liefert, ist das Gesamtergebnis „Deny“.
- Wenn „Context Constraints“ das Ergebnis „Deny“ liefert, ist das Gesamtergebnis „Deny“.
- Das Teilergebnis von „Separation of Duty“ ist mit RBAC-Core identisch und wird daher ignoriert.
- Das Teilergebnis von „Separation of Duty mit Rollenhierarchie“ ist mit „RBAC mit Rollenhierarchie“ identisch und wird daher ignoriert.

Wenn man aus der obigen Liste alle Module entfernt, die ignoriert werden, bleiben nur noch zwei übrig, und für die passt dann der Algorithmus **deny-overrides**. Es bleibt nur noch ein Problem: Das Gesamtergebnis ist auch in diesem Fall nicht das erwartete,

denn das Modul „Context Constraints“ ist in keiner Weise mit dem Modul „RBAC mit Rollenhierarchie“ verknüpft, so dass die indirekten Rollen für die Context Constraints keine Rolle spielen. Letztendlich kann kein noch so ausgeklügelter Algorithmus die Einzelergebnisse zum gewünschten Gesamtergebnis kombinieren. Dies ist jedoch nicht einem fehlerhaften Übersetzungsprozess anzulasten, sondern der Tatsache, dass in der ORKA-Spezifikation keine Kombination der Module „RBAC mit Rollenhierarchie“ und „Context Constraints“ existiert. Dadurch kann es passieren, dass Zugriffe erlaubt sind, die eigentlich durch die Context Constraints verhindert werden sollen. Diese Sicherheitslücke kann nur dadurch vermieden werden, dass alle Rollen, die in den Context Constraints verwendet werden (Relationen *RCC* und *PACC*), direkte Rollen der Subjekte sind und nicht etwa indirekte.

Da es nicht die Aufgabe dieser Diplomarbeit ist, die bereits vorhandene Spezifikation zu ändern, muss hier eine andere Lösung gefunden werden. Ich habe mich dafür entschieden, je nachdem, welche Module in der konkreten Policy überhaupt benutzt werden, diese mit dem Algorithmus **first-applicable** zu kombinieren, um so zuerst die unerfüllten CCs herauszufiltern und die Entscheidung dann gemäß RBAC-Core oder „RBAC mit Rollenhierarchie“ zu treffen. Dieses Verfahren unterscheidet sich von den restlichen im ORKA-Projekt implementierten, scheint mir aber am vernünftigsten zu sein. In den bisherigen Testfällen fällt der Unterschied auch gar nicht auf, da dort die Module „Context Constraints“ und „RBAC mit Rollenhierarchie“ nicht zusammen getestet werden.

5.13 Zusammenfassung

In diesem Kapitel wurde ausführlich beschrieben, wie die ORKA-Datenstrukturen in äquivalente XACML-Datenstrukturen übersetzt werden können. Die einzelnen Module, die in Kapitel 3 vorgestellt wurden, wurden einzeln in XACML- $\langle$ Policy $\rangle$ s übersetzt, deren Ergebnisse dann zu einem Gesamtergebnis kombiniert wurden.

6 Implementierung der Übersetzung

Die Implementierung des Übersetzers findet im Rahmen des ORKA-Projekts statt und benutzt daher dessen Quellcode. Dieser Code befindet sich noch in der Entwicklung. Der Code der Zielsprache (sunxacml) ist dagegen seit dem Release im Jahre 2004 stabil und hat sich seitdem nicht wesentlich verändert.

6.1 ORKA-Quelltext

Der ORKA-Quellcode liegt in Java vor. Er ist sauber in verschiedene Java-Pakete unterteilt, je nach Funktion des Codes und beabsichtigtem Einsatzzweck. Die zur Übersetzung benötigten Klassen repräsentieren hauptsächlich Datenstrukturen ohne weitere Funktionalität, daher sind sie recht einfach gehalten und auch gut benutzbar. Die Namen der Klassen und Methoden entsprechen weitgehend der dazugehörigen Spezifikation. Dadurch sind große Quelltextkommentare in vielen Fällen unnötig und fehlen auch häufig.

6.1.1 Organisation des Quellcodes

Die Java-Pakete, die für die Übersetzung in XACML relevant sind, sind:

org.orka.enforce: Allgemeine Datenstrukturen wie eine Zugriffsanfrage, die dazugehörige Antwort und die Schnittstelle für eine *PolicyDecisionComponent*.

org.orka.enforce.context: Alles, was für dynamische Parameter von Context Constraints notwendig ist.

org.orka.opl: Hier sind in der Klasse *PolicyObjectTree* alle ORKA-Module zusammengefasst und können über eine einheitliche Schnittstelle angesprochen werden.

org.orka.opl.modules: Die ORKA-Module, wie in Kapitel 3 beschrieben. Die einzelnen Module haben keine gemeinsame Schnittstelle, da sie nur zum Speichern der jeweils relevanten persistenten Daten dienen und es dort keine Überschneidungen gibt. Der Code, um eine Anfrage gegenüber einer Policy auszuwerten, ist in einem anderen Java-Paket, nämlich *org.orka.enforce.lang.opl.modules* enthalten, da diese Funktionalitäten (vernünftigerweise) streng getrennt sind.

org.orka.opl.types: Die Datentypen innerhalb der einzelnen OPL-Policys, zum Beispiel *RolePermission*, *ContextConstraint*, *CFParam*.

6.2 sunxacml

Auf der Webseite des XACML-Komitees¹ befindet sich eine Liste mit XACML-Implementierungen, unter denen auch eine von Sun Microsystems ist. Diese Implementierung (*sunxacml*, für Java 1.4) wurde vom ORKA-Projekt ausgewählt, um die ORKA-Policys in XACML zu repräsentieren. Der aktuelle Stand von *sunxacml* (September 2008) ist, dass XACML 1.0 und XACML 2.0 unterstützt werden, jedoch fanden in den 19 Monaten davor keine Weiterentwicklungen an dem Projekt statt. Die einzigen beiden offiziellen Releases stammen noch aus der Zeit vor XACML 2.0, und ORKA verwendet Release 1.2, das neuere der beiden.

Es werden einige der Design Pattern von Gamma et al. [DesignPatterns] aufgegriffen, unter anderem Proxy und Factory. Dadurch wirkt die Implementierung auf den ersten Blick aufgebläht. Bei genauerem Hinsehen bewirkt diese Komplexität jedoch eine Flexibilität und Anpassbarkeit, die gerade für die Implementierung einer so abstrakten Norm wie XACML notwendig ist.

Für die in XACML definierten Datenstrukturen gibt es jeweils eigene Klassen, die in Form von Werttypen, wie sie in [Werttypen] beschrieben werden, implementiert sind. Das heißt, dass die Objekte dieser Klassen mit den Konstruktoren erzeugt werden und danach nicht mehr verändert werden können. (Der Begriff des Werttyps wird jedoch auch von anderen Stellen für andere, verwandte Konzepte verwendet.) In vielen der Klassen sind nur die grundlegenden, elementaren Operationen implementiert, nicht jedoch darauf aufbauende, „handlichere“ Methoden, um häufig verwendete Benutzungsmuster einfach abzubilden. So fehlen an einigen Stellen Konstruktoren, bei denen die optionalen Elemente nicht angegeben werden müssen. Dadurch wird der Code, der *sunxacml* benutzt, länger, und die vielen Vorkommen von `null` im Quelltext lenken auch vom eigentlichen Zweck des Codes ab.

Die einzelnen Klassen und Methoden sind ausführlich dokumentiert,² nur an wenigen Stellen fehlen ein paar kleine Details. Insgesamt lässt sich mit dem Code aber gut arbeiten.

6.2.1 Implementierungsfehler

sunxacml besteht die kompletten Konformitätstests von XACML 1.0. Trotzdem gibt es einige Implementierungsfehler, die sich auf die praktische Arbeit auswirken:

- Die Klassen, die den XACML-Datenstrukturen entsprechen, haben eine Methode, um ihre XML-Repräsentation auszugeben. Diese Methode kann jedoch nicht damit umgehen, wenn in der Eigenschaft `<Description>` oder ähnlichen Sonderzeichen (`<>&`) vorkommen. Zur Abhilfe müsste man die komplette XML-Erzeugung selbst nachschreiben oder aber sicherstellen, dass keine Sonderzeichen in den entsprechenden Eigenschaften auftauchen. (Sourceforge-Bugs #1979167, #2009693)
- Der Code, um XML-Code zu parsen, kann nicht mit Namensräumen umgehen. Wenn man selbst alle Policys erzeugt, kann man dieses Problem umgehen, problematisch

¹<http://www.oasis-open.org/committees/xacml/>

²<http://sunxacml.sourceforge.net/javadoc>

wird es jedoch, wenn man Policys einlesen will, die aus fremden Quellen stammen. (# 944592, # 1725177, # 1725867, # 2008487)

- Beim Einlesen der XML-Datenstrukturen können Teile verlorengehen oder unerwartet Exceptions auftreten. (# 2031179, # 2036819)
- Beim Auswerten von Attributen wird in einigen Fällen unnötigerweise nach einem Attribut gesucht (und keins gefunden), obwohl es schon in der Anfrage selbst vorhanden ist. (# 2014107)
- In der XACML-Spezifikation heißt es, dass ein `<Attribute>` mehrere `<AttributeValues>` haben kann. In sunxacml sind die Datenstrukturen fast alle 1 : 1 abgebildet, ein `<Attribute>` kann dort jedoch nur genau einen Wert haben. Dieser Wert jedoch kann ein Multiwert sein (Java-Klasse *BagAttribute*), so dass auch an anderen Stellen, an denen eigentlich nur ein einziger Wert vorkommen kann, auf einmal Multimengen möglich sind.

All diese Dinge muss man bei der Arbeit mit sunxacml im Hinterkopf behalten, um sicherzustellen, dass es sich wie erwartet verhält.

6.3 Java-Pakete und Klassen des Übersetzers

Innerhalb des ORKA-Codes befindet sich der XACML-Übersetzer im Java-Paket *org.orka.enforce.lang.xacml* und einigen Unterpaketen. Die einzelnen Übersetzer innerhalb dieses Paketes werden in Abschnitt 6.5 beschrieben.

Im Paket *org.orka.enforce.lang.xacml.util* befinden sich einige weitere Klassen, die bei der Implementierung des Übersetzers nützlich waren. In diesen Klassen werden all die Dinge implementiert, die von Java 1.5, ORKA und sunxacml nicht ausreichend benutzerfreundlich zur Verfügung gestellt wurden, um den eigentlichen Code kurz zuhalten, sowie Methoden, die nicht an eine bestimmte Klasse gebunden sind.

Zu der Entwicklung des Übersetzers gehört auch, einige grundlegende Operationen zu testen. Das erledigen die Klassen im Paket *org.orka.test.xacmlpdc*. Derzeit gibt es dort nur zwei Klassen: *RbacCoreTranslatorTest* und *CCTranslatorTest*. Diese beiden Klassen decken jedoch den Großteil des implementierten Codes ab, so dass diese Menge durchaus als ausreichend gelten kann.

6.4 Eingliederung des Übersetzers in ORKA

In ORKA gibt es die Schnittstelle *PolicyDecisionComponent*, die einen PDP spezifiziert. Der Ablauf ist, dass zuerst die für den PDP relevanten Policys mittels einer Methode *registerPolicyObject* registriert werden, anschließend können beliebig viele Anfragen mittels *performAccessDecision* gegen die bereits geladenen Policys evaluiert werden, und schließlich können geladene Policys mit *unregisterPolicyObject* wieder entfernt werden.

6 Implementierung der Übersetzung

Die einzige Klasse des Übersetzers, auf die vom ORKA-Code aus zugegriffen werden muss, ist *XacmlPdc*, die auch das oben genannte Interface implementiert. Alle anderen Klassen sind Implementierungsdetails.

Die Übersetzung der ORKA-Policys in XACML findet in der Methode *registerPolicyObject* statt. Das Ergebnis der Übersetzung wird dann lokal gespeichert, so dass es bei Aufrufen von *performAccessDecision* schnell verfügbar ist. Dadurch muss die Policy nur einmal übersetzt werden und nicht bei jeder Anfrage.

Innerhalb der *performAccessDecision*-Methode wird die ORKA-Anfrage in eine äquivalente XACML-Anfrage übersetzt (Abschnitt 6.6). Dann wird für jede Anfrage jeweils ein neuer XACML-PDP erzeugt, da einige der Komponenten von der konkreten Anfrage abhängen. Diese Aktion ist nicht so rechenaufwendig, wie das vielleicht klingen mag, da im Grunde nur ein paar existierende Objekte zu neuen Objekten kombiniert werden. Der neu erzeugte PDP evaluiert nun die XACML-Anfrage (Abschnitt 6.7). Das Ergebnis der Evaluation ist eine XACML-Antwort, und die muss zu guter Letzt noch in eine äquivalente ORKA-Antwort umgewandelt werden (Abschnitt 6.9). Wie das im Detail abläuft, wird in den folgenden Abschnitten erläutert.

6.5 Übersetzung der Berechtigungsmodule

In den vorangegangenen Abschnitten wurde der allgemeine Teil der Übersetzung und Policy-Auswertung beschrieben, in den folgenden Abschnitten geht es um die Übersetzung der einzelnen ORKA-Policys in XACML-Policys. Dazu werden bei jedem der Module die Besonderheiten der Übersetzung dargestellt.

6.5.1 RBAC-Core

Die Übersetzung des Moduls „RBAC-Core“ findet in der Klasse *RbacCoreTranslator* statt. Die Methode *translate* stellt die Schnittstelle nach außen dar. Die anderen Methoden konstruieren die gewünschten XACML-Datenstrukturen aus der ORKA-Policy. Die Methode *makeRule* ist dabei so allgemein gehalten, dass sie auch von den Übersetzern für die anderen Module benutzt wird.

Die Implementierung entspricht derzeit nicht der Beschreibung in Abschnitt 5.7.4, da das *<Target>* der Policy nicht wie dort angegeben generiert wird, sondern die Policy einfach auf alles anwendbar ist. Dadurch ist der Code kürzer. Sollte sich jedoch später herausstellen, dass es an dieser Stelle Performanceprobleme gibt, kann die Methode immer noch implementiert werden.

6.5.2 RBAC mit Rollenhierarchien

Der Übersetzer für das Modul „RBAC mit Rollenhierarchie“ baut stark auf dem Übersetzer für RBAC-Core auf. Er unterscheidet sich von letzterem nur durch den Namen des Attributs (*orka:role-transitive* statt *orka:role*) und die IDs innerhalb der generierten Policys. Dementsprechend ist der Code auch nur eine einzige Zeile lang.

6.5.3 Context Constraints

Der Übersetzer für die Context Constraints ist da schon interessanter, da hier gegenüber RBAC-Core einige neue Aufgaben zu bewältigen sind. Die Übersetzung eines CCs erfolgt in der Reihenfolge „Top-Down-Up“. Zuerst wird anhand der zu übersetzenden Funktion ein Funktionsübersetzer gesucht, der diesen Funktionsaufruf übersetzen kann. Wenn der gefunden ist, wird ihm die gesamte Arbeit übertragen.

Der Funktionsübersetzer ermittelt abhängig von der konkreten Funktion die Datentypen der einzelnen Parameter und übersetzt die einzelnen Parameter. Dann konstruiert er zu der ORKA-Funktion die passende XACML-Funktion und gibt schließlich einen XACML-Ausdruck zurück, der die Funktion auf die Parameter anwendet.

Die Übersetzung der einzelnen Parameter ist direkt in der Klasse *CCTranslator* implementiert, da es sich nicht gelohnt hat, für die zwei unterschiedlichen Fälle extra eine neue Klasse zu erzeugen. Bei der Übersetzung wird unterschieden, ob es sich um Literalparameter oder Kontextparameter handelt, da sich der Code zur Übersetzung dieser Parameterarten grundsätzlich unterscheidet. Die Funktion *translateLiteral(CFParam)* ist für die Übersetzung eines literalen Parameter in ein XACML-Element `<AttributeValue>` zuständig, und die Funktion *translateContext(CFParam, Type)* übersetzt einen dynamischen Parameter in einen `<*AttributeDesignator>`. Bei den dynamischen Parametern ist als Datentyp immer `context` angegeben, XACML benötigt aber die Angabe des „echten“ Datentyps. Dieser wird daher als zusätzlicher Parameter angegeben.

Die Übersetzung der einzelnen Funktionen geschieht durch die spezialisierten Klassen *ComparisonTranslator* und *TimeBetweenFunctionTranslator*. Diese Klassen implementieren jeweils das Interface *FunctionTranslator*, das einerseits prüfen kann, ob ein CC von diesem Funktionsübersetzer auch übersetzt werden kann (*isTranslatable*), und auch die Übersetzung durchführt (*translate*).

Bei der Übersetzung der Vergleichsfunktionen (Klasse *ComparisonTranslator*) ist es notwendig, dass beide Parameter den gleichen Datentyp haben. Diese Überprüfung erledigt die Methode *commonType*. Dann muss der Name der ORKA-Vergleichsfunktion noch in den entsprechenden XACML-Funktionsnamen umgewandelt werden. ORKA und XACML unterscheiden sich an dieser Stelle etwas in der Wortwahl (zum Beispiel ORKA: *equal-or-more-than*, XACML: *greater-than-or-equal*). Aber abgesehen von solchen Kleinigkeiten verläuft die Übersetzung recht intuitiv.

Eine kleine Komplikation bietet da noch die ORKA-Funktion *not-equal*, die es so in XACML nicht gibt. Diese Funktion muss daher in eine Verkettung der XACML-Funktionen *not* und *type-equal* umgesetzt werden.

In Abschnitt 5.9.3 wurde beschrieben, wie die Gesamtpolicy aussieht. Die Implementierung weicht absichtlich von dieser Beschreibung ab und erzeugt nur die Policy, die dort mit `check-applicable-ccs` bezeichnet ist. Der restliche Teil des dortigen Listings wird bei der Kombination der Policies (siehe Abschnitt 5.12) erledigt.

6.6 Übersetzung der ORKA-Anfrage in XACML

Eine XACML-Anfrage besteht aus einem Subjekt, einer Aktion und einer Ressource, jeweils mit beliebig vielen Attributen. Diese drei Komponenten müssen aus der ORKA-Anfrage (*AccessDecisionRequest*) extrahiert werden. Das Subjekt ist nicht direkt in der Anfrage enthalten, sondern wird vom *ContextManager* als Attribut `logon.subject` bereitgestellt. Die Aktion und die Ressource sind direkt in der ORKA-Anfrage enthalten.

Damit sind die drei Attribute `subject-id`, `resource-id` und `action-id` vorhanden und können zu einer XACML-Anfrage kombiniert werden. Dazu werden sie jeweils in Listen gesteckt, da XACML-Anfragen auch mehrere Attribute pro Komponente unterstützt. Beim Subjekt ist sogar noch eine weitere Listenebene nötig, da XACML auch Anfragen mit mehreren Subjekten kennt.

6.7 Evaluation der Anfrage

Um eine ORKA-Anfrage zu beantworten, wird zunächst die anwendbare Policy aus den bereits übersetzten Policys herausgesucht. Welche der Policys anwendbar ist, wird vom Aufrufer der Methode explizit angegeben, so dass der XACML-PDP diese Entscheidung nicht selbst treffen muss. Dadurch wird das Erzeugen der XACML-Policys einfacher, da jede der erzeugten Policys auf alle Anfragen anwendbar sein darf. Normalerweise verlangt XACML von allen Policys, die dem PDP zur Verfügung stehen, dass nur eine einzige anwendbar sein darf (daher auch der Algorithmus `only-one-applicable` in Abschnitt 4.12).

Anschließend wird ein XACML-PDP erzeugt, der die Anfrage bearbeiten kann. In `sunxacml` besteht ein PDP aus einem *AttributeFinder*, der für das Auflösen von benannten Attributen zuständig ist, einem *PolicyFinder*, der anhand der Anfrage die (eine) passende Policy ermittelt, und einem *ResourceFinder*, der hier nicht benötigt wird.

Der *AttributeFinder* ist speziell an die ORKA-Umgebung angepasst, da er die ORKA-Attribute `logon.subject` und insbesondere die Parameter der Context Constraints auflösen muss. Dazu benötigt er den *ContextManager* aus der ORKA-Anfrage (für die Context Constraints) sowie die ORKA-Policy, um die Rollenzugehörigkeit des Subjekts aufzulösen. (Letzteres wäre eigentlich die Aufgabe des *ContextManagers*, aber ich habe keine Möglichkeit gefunden, an diese Informationen heranzukommen.) In einem Produktivsystem ist es unwahrscheinlich, dass diese Informationen so direkt verfügbar sind, daher wird eine Portierung auf bestehende Systeme an dieser Stelle noch Probleme aufwerfen.

Auch der *PolicyFinder* ist speziell auf die ORKA-Testumgebung zugeschnitten, da er nur eine einzige Policy kennt und diese bei jeder Anfrage zurückgibt. Wie am Anfang dieses Abschnittes bereits erwähnt, ist bei einer Umsetzung in ein Produktivsystem an dieser Stelle mit Problemen zu rechnen.

Die ORKA-Testumgebung stellt jedoch genau die Informationen zur Verfügung, die der XACML-PDP benötigt, um mit relativ geringem Aufwand implementiert zu werden, daher sind die Probleme der Datenverfügbarkeit hier nicht aufgetreten.

6.8 Auflösen der schnelllebigen Daten

Während der Auswertung kann es vorkommen, dass der PDP auf ein Attribut trifft, das er nicht auflösen kann. Dann fragt er seinen *AttributeFinder*, in diesem Fall den *OrkaAttributeFinder*. Dieser stellt dem PDP all die „schnelllebigen Daten“ aus den Modulbeschreibungen in Kapitel 3 zur Verfügung, so dass sie in den Policies benutzt werden können. Das betrifft die folgenden Daten:

- Das aktive Subjekt, das die Anfrage stellt. In den Context Constraints von ORKA wird es `logon.subject` genannt, in XACML heißt das entsprechende Attribut `subject-id`.
- Die Rollen des aktiven Subjekts, ohne Vererbung. Diese Daten sind im ORKA-Modul RBAC-Core in der Relation *RH* festgelegt. In den XACML-Policies tauchen sie unter dem Attributnamen `orka:role` auf.
- Die Rollen des aktiven Subjekts, mit Vererbung. In ORKA ist dies eine Kombination der Relationen *SR* und *RH**. Das dazugehörige XACML-Attribut heißt `orka:role-transitive`
- Die Parameter der Context Constraints, die dynamisch evaluiert werden. Ein Parameter wird als XACML-Attribut mit dem Namen `orka:cfparam#...` codiert, wobei der jeweilige Wert des Parameters statt der ... eingesetzt wird.

6.9 Übersetzung der XACML-Antwort in ORKA

Das Ergebnis des XACML-PDPs ist eine XACML-Antwort, deren Datenstruktur relativ komplex werden kann. Diese Antwort muss auf eine ORKA-Antwort abgebildet werden. Letztere besteht aus einem Status, der einen der Werte `NotLoggedOn`, `InvalidOrIncompleteRequest`, `AccessDenied`, `AccessGranted`, `TemporaryFailure` oder `PermanentFailure` annimmt, sowie einer optionalen Menge an Verpflichtungen (engl.: obligations), die jedoch im Rahmen dieser Arbeit nicht benötigt werden.

Die Antwortmöglichkeiten auf der Seite von XACML sind `Permit`, `Deny`, `NotApplicable` und `Indeterminate`. Diese Antworten müssen nun auf die ORKA-Antworten abgebildet werden. Dabei wird `Permit` naheliegenderweise auf `AccessGranted` abgebildet, `Deny` auf `AccessDenied`, und alle anderen auf `InvalidOrIncompleteRequest`. Dies ist einerseits die einfachste Form, und obwohl es wünschenswert wäre, zwischen temporären und permanenten Fehlern zu unterscheiden, so gibt die XACML-Antwort im allgemeinen keine Auskunft darüber. Daher ist eine detailliertere Einordnung der Fehler nur mit unverhältnismäßig großem Aufwand und abhängig von den genauen XACML-Fehlermeldungen möglich.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung der Arbeit

In dieser Arbeit wurden einige der Zugriffskontrollmodule aus dem ORKA-Projekt nach XACML übersetzt, so dass sie in einem XACML-PDP zur Zugriffskontrolle benutzt werden können.

Zuerst wurden in Kapitel 2 die begrifflichen Grundlagen der Zugriffskontrolle gelegt. Es wurden verschiedene Zugriffskontrollmodelle beschrieben, die sich in ihrer Komplexität grundlegend unterscheiden, von einfach (Zugriffskontrollmatrix) bis komplex (XACML).

Die Quellsprache des Übersetzers (OPL/XML) wurde in Kapitel 3 detailliert beschrieben. Ein besonderes Augenmerk wurde dabei auf die einzelnen Zugriffskontrollmodule gelegt, insbesondere auf die Eigenschaften, die für die spätere Übersetzung relevant sind (Datentypen, persistente und schnelllebige Daten, Zugriffsentscheidung). Es wurde angesprochen, wie die Entscheidungen der einzelnen Module zu einer Gesamtentscheidung kombiniert werden. Die dabei auftretenden Probleme und deren Lösung wurden jedoch auf ein späteres Kapitel verschoben.

Die Zielsprache des Übersetzers (XACML) wurde in Kapitel 4 ausführlich beschrieben. Dabei wurde besonderer Wert darauf gelegt, die Gründe des XACML-Komitees zu verstehen, die dazu geführt haben, dass die Sprache genau so entwickelt wurde, wie sie in [XACML-1.0] festgeschrieben wurde. Diese Gründe bleiben bei solchen Entwicklungen häufig verborgen, und gerade das macht es Neueinsteigern schwer, die Struktur der Sprache nachzuvollziehen und als „unter den gegebenen Umständen bestmöglich“ zu akzeptieren. Zudem ist diese Beschreibung meines Wissens die erste ausführliche, deutschsprachige Einführung in XACML. Um die Ausdrucksstärke und Flexibilität von XACML zu demonstrieren, wurden auch einige Möglichkeiten von XACML vorgestellt, die im Rahmen dieser Arbeit nicht weiter benötigt werden.

In Kapitel 5 wurde auf konzeptioneller Ebene das Übersetzungsverfahren dargelegt. Die einzelnen ORKA-Module wurden in XACML-Policys übersetzt und schließlich zu einer Gesamt-Policy kombiniert. Eine besondere Herausforderung stellte das bereits in Kapitel 3 beschriebene Problem der Kombination mehrerer Policys dar. Dieses Problem wurde aufgegriffen und ausführlich diskutiert. Anschließend wurde eine Lösung entwickelt, die das Problem zwar nicht vollständig beseitigt, die aber unter den gegebenen Umständen bestmöglich ist.

Die konkrete Implementierung des von mir entwickelten Übersetzers wurde in Kapitel 6 beschrieben und detailliert dokumentiert. Außerdem wurden die im Zuge der Implementierung getroffenen Annahmen dargelegt und ausführlich begründet. Um Designentscheidungen verdeutlichen und vorhandene Schwierigkeiten besser beschreiben zu können, wurde der Beschreibung meiner Implementierung eine kritische Analyse

derjenigen Quelltexte (ORKA, sunxacml) vorangestellt, die vom Übersetzer benutzt werden.

7.2 Auswirkungen auf die Welt

Im Rahmen dieser Arbeit habe ich mich gründlich in die Systematik von XACML eingearbeitet. Das wurde insbesondere dadurch erschwert, dass es praktisch keine deutschsprachige Literatur zu XACML gibt. Auch im Englischen sieht die Situation nicht viel besser aus. Das XACML-Komitee stellt zwar die Spezifikation¹ sowie einen einführenden Text² kostenfrei zur Verfügung, weitergehende Erklärungen, wie etwa die Begründung der Designentscheidungen, sind nicht öffentlich verfügbar.

Da es für den Bau eines Übersetzungsprogrammes unerlässlich ist, die Quell- und Zielsprache genau zu kennen, musste ich mir die Details von XACML, die über die bloße Beschreibung im Standard hinausgehen, selbst erarbeiten. In dieser Phase fielen mir etliche Stellen auf, an denen der XACML-Standard undeutlich, unvollständig oder fehlerhaft war. Diese Beobachtungen habe ich dem XACML-Komitee mitgeteilt, und meine Verbesserungsvorschläge wurden auch fast alle angenommen.³ Bei den Vorschlägen handelte es sich um einfache Tippfehler, kürzere und prägnantere Formulierungen, falsche Verwendung von Schlüsselwörtern wie „MAY“, Hinweise auf undefiniertes Verhalten (gegenseitig abhängige Variablen, Umwandlung von großen Ganzzahlen in Fließkommazahlen) und einige mehr.

Die in dieser Arbeit verwendete XACML-Implementierung (sunxacml) stammt von Sun Microsystems. Auch dort habe ich während der Einarbeitungsphase mehrere Fehler entdeckt und den Code an etlichen Stellen verbessert. Da der Hauptentwickler von sunxacml jedoch mittlerweile an anderen Projekten arbeitet und ansonsten wenig Interesse an der Fortführung des Projekts zu bestehen scheint, sind bisher nur wenige meiner Vorschläge in den offiziellen Code eingeflossen. Der Hauptgrund dafür ist, dass die Projektrichtlinien vorschreiben, dass Code nur nach einem Review eingepflegt wird, ich aber niemanden für diese Aufgabe gefunden habe.

7.3 Ausblick

Mit Vollendung dieser Diplomarbeit liegt ein Übersetzungsprogramm vor, das einige Zugriffskontrollmodule der Sprache OPL nach XACML übersetzt. ORKA definiert noch weitere Module, insbesondere für Workflows. Die Übersetzung dieser Module war aus zeitlichen Gründen nicht möglich, wird jedoch außerhalb dieser Arbeit noch von mir implementiert.

¹[XACML-1.0]

²[XACML-1.0-intro]

³Siehe <http://lists.oasis-open.org/archives/xacml-comment/>, Abschnitte 200808 und 200809.

A Beispiel

Um zu demonstrieren, dass der Übersetzer auch wirklich funktionsfähig ist, folgt nun ein Beispiel, das innerhalb des ORKA-Projekts für die Performance-Messungen verwendet wird, und das, was der Übersetzer daraus gemacht hat. Da diese Beispiele relativ lang sind, werden sie in einer kleineren Schrift dargestellt, um Platz zu sparen.

A.1 OPL-Policy

Die folgende Policy stellt dar, in welcher Form die Daten innerhalb des ORKA-Projekts organisiert sind. Eine Policy besteht aus ein paar Informationen, die die Policy an sich beschreiben (`<policy_object_attributes>`) sowie den einzelnen Modulen (vgl. Kapitel 3). Innerhalb eines Moduls werden die in der jeweiligen Modulbeschreibung vorgestellten persistenten Daten aufgezählt.

In der Originalfassung enthielt diese Policy noch weitere Module, die im Rahmen dieser Diplomarbeit jedoch nicht implementiert wurden und daher aus Platzgründen weggelassen wurden.

```
<?DOCTYPE policy_object SYSTEM "policy-object.dtd">
<policy_object>
  <policy_object_attributes>
    <attribute key="name" value="Reference Scenario"/>
    <attribute key="version" value="1.0"/>
    <attribute key="date" value="2008-02-23"/>
    <attribute key="creator" value="rwolf"/>
  </policy_object_attributes>
  <policy_object_modules>
    <!-- ===== Module RBACCore ===== -->
    <module_rbac_core_policy>
      <users>
        <user user_id="user:mkohler"/>
        <user user_id="user:mmueller"/>
        <user user_id="user:amaier"/>
        <user user_id="user:sschaefer"/>
        <user user_id="user:performance-tester"/>
      </users>
      <roles>
        <role role_id="role:clerk_preprocessor"/>
        <role role_id="role:clerk_postprocessor"/>
        <role role_id="role:supervisor"/>
        <role role_id="role:customer"/>
        <role role_id="role:manager"/>
      </roles>
      <permissions>
```

A Beispiel

```
<permission permission_id="permission:query_customer_data">
  <operation operation_id="query"/>
  <object object_id="CustomerData"/>
</permission>
<permission permission_id="permission:update_customer_data">
  <operation operation_id="update"/>
  <object object_id="CustomerData"/>
</permission>
<permission permission_id="permission:prepare_ratingreport">
  <operation operation_id="prepare"/>
  <object object_id="RatingReport"/>
</permission>
<permission permission_id="permission:release_ratingreport">
  <operation operation_id="release"/>
  <object object_id="RatingReport"/>
</permission>
<permission permission_id="permission:post_ratingreport">
  <operation operation_id="post"/>
  <object object_id="RatingReport"/>
</permission>
<permission permission_id="permission:query_ratingreport">
  <operation operation_id="query"/>
  <object object_id="RatingReport"/>
</permission>
<permission permission_id="permission:update_ratingreport">
  <operation operation_id="update"/>
  <object object_id="RatingReport"/>
</permission>
<permission permission_id="permission:query_productbundle">
  <operation operation_id="query_avail_prod"/>
  <object object_id="ProductBundle"/>
</permission>
<permission permission_id="permission:modify_productbundle">
  <operation operation_id="modify"/>
  <object object_id="ProductBundle"/>
</permission>
<permission permission_id="permission:commit_productbundle">
  <operation operation_id="commit"/>
  <object object_id="ProductBundle"/>
</permission>
<permission permission_id="permission:print_contract">
  <operation operation_id="print"/>
  <object object_id="Contract"/>
</permission>
<permission permission_id="permission:sign_contract">
  <operation operation_id="sign"/>
  <object object_id="Contract"/>
</permission>
<permission permission_id="permission:update_contract">
  <operation operation_id="update"/>
  <object object_id="Contract"/>
</permission>
<permission permission_id="permission:open_account">
  <operation operation_id="open"/>
```

```

    <object object_id="Account"/>
  </permission>
  <permission permission_id="permission:createBankingProcess">
    <operation operation_id="createProcess"/>
    <object object_id="process:BankingProcess"/>
  </permission>
  <permission permission_id="permission:endInputCustomerData">
    <operation operation_id="end"/>
    <object object_id="task:inputCustomerData"/>
  </permission>
  <permission permission_id="permission:endVerifyCustomerData">
    <operation operation_id="end"/>
    <object object_id="task:customerIdentification"/>
  </permission>
  <permission permission_id="permission:endCheckCreditWothiness">
    <operation operation_id="end"/>
    <object object_id="task:checkcreditwothiness"/>
  </permission>
  <permission permission_id="permission:endCheckInternalRating">
    <operation operation_id="end"/>
    <object object_id="task:internalRating"/>
  </permission>
  <permission permission_id="permission:endChooseProduct">
    <operation operation_id="end"/>
    <object object_id="task:chooseProduct"/>
  </permission>
  <permission permission_id="permission:endPrintContract">
    <operation operation_id="end"/>
    <object object_id="task:printContract"/>
  </permission>
  <permission permission_id="permission:endQueryCreditBureau">
    <operation operation_id="end"/>
    <object object_id="task:QueryCreditBureau"/>
  </permission>
</permissions>
<user_assignments>
  <user_assignment user_id="user:mkohler"
    role_id="role:clerk_preprocessor"/>
  <user_assignment user_id="user:mkohler"
    role_id="role:clerk_postprocessor"/>
  <user_assignment user_id="user:mmueller"
    role_id="role:clerk_preprocessor"/>
  <user_assignment user_id="user:mmueller"
    role_id="role:clerk_postprocessor"/>
  <user_assignment user_id="user:mkohler"
    role_id="role:supervisor"/>
  <user_assignment user_id="user:mmueller"
    role_id="role:supervisor"/>
  <user_assignment user_id="user:amaier"
    role_id="role:manager"/>
  <user_assignment user_id="user:sschaefer"
    role_id="role:customer"/>
  <user_assignment user_id="user:performance-tester"
    role_id="role:clerk_preprocessor"/>

```

A Beispiel

```
<user_assignment user_id="user:performance-tester"
  role_id="role:clerk_postprocessor"/>
<user_assignment user_id="user:performance-tester"
  role_id="role:supervisor"/>
<user_assignment user_id="user:performance-tester"
  role_id="role:manager"/>
<user_assignment user_id="user:performance-tester"
  role_id="role:customer"/>
</user_assignments>
<permission_assignments>
  <permission_assignment permission_id="permission:query_customer_data"
    role_id="role:clerk_preprocessor"/>
  <permission_assignment permission_id="permission:update_customer_data"
    role_id="role:clerk_preprocessor"/>
  <permission_assignment permission_id="permission:prepare_ratingreport"
    role_id="role:clerk_postprocessor"/>
  <permission_assignment permission_id="permission:post_ratingreport"
    role_id="role:clerk_postprocessor"/>
  <permission_assignment permission_id="permission:query_ratingreport"
    role_id="role:clerk_postprocessor"/>
  <permission_assignment permission_id="permission:update_ratingreport"
    role_id="role:supervisor"/>
  <permission_assignment permission_id="permission:query_productbundle"
    role_id="role:clerk_postprocessor"/>
  <permission_assignment permission_id="permission:modify_productbundle"
    role_id="role:clerk_postprocessor"/>
  <permission_assignment permission_id="permission:print_contract"
    role_id="role:clerk_postprocessor"/>
  <permission_assignment permission_id="permission:sign_contract"
    role_id="role:customer"/>
  <permission_assignment permission_id="permission:sign_contract"
    role_id="role:manager"/>
  <permission_assignment permission_id="permission:update_contract"
    role_id="role:manager"/>
  <permission_assignment permission_id="permission:open_account"
    role_id="role:clerk_postprocessor"/>
  <permission_assignment permission_id="permission:release_ratingreport"
    role_id="role:clerk_postprocessor"/>
  <permission_assignment permission_id="permission:release_ratingreport"
    role_id="role:supervisor"/>
  <permission_assignment permission_id="permission:commit_productbundle"
    role_id="role:clerk_postprocessor"/>
  <permission_assignment permission_id="permission:commit_productbundle"
    role_id="role:supervisor"/>
  <permission_assignment permission_id="permission:endInputCustomerData"
    role_id="role:clerk_preprocessor"/>
  <permission_assignment permission_id="permission:endVerifyCustomerData"
    role_id="role:clerk_preprocessor"/>
  <permission_assignment permission_id="permission:createBankingProcess"
    role_id="role:clerk_preprocessor"/>
  <permission_assignment permission_id="permission:createBankingProcess"
    role_id="role:clerk_postprocessor"/>
  <permission_assignment permission_id="permission:endCheckCreditWothiness"
    role_id="role:clerk_postprocessor"/>
```

```

    <permission_assignment permission_id="permission:endCheckInternalRating"
                          role_id="role:clerk_postprocessor"/>
    <permission_assignment permission_id="permission:endChooseProduct"
                          role_id="role:clerk_postprocessor"/>
    <permission_assignment permission_id="permission:endPrintContract"
                          role_id="role:clerk_postprocessor"/>
    <permission_assignment permission_id="permission:endQueryCreditBureau"
                          role_id="role:clerk_postprocessor"/>
    <permission_assignment permission_id="permission:createBankingProcess"
                          role_id="role:supervisor"/>
    <permission_assignment permission_id="permission:endCheckCreditWothiness"
                          role_id="role:supervisor"/>
    <permission_assignment permission_id="permission:endQueryCreditBureau"
                          role_id="role:supervisor"/>
  </permission_assignments>
</module_rbac_core_policy>
<!-- ===== Module ExoContext ===== -->
<module_exo_context_policy>
  <context_constraints>
    <context_constraint cc_id="cc:cc1">
      <context_function_id id="equal-or-less-than"/>
      <context_function_parameters>
        <parameter value="cst.getCustomerExternalRating(sCtx.getProcessVariable(parameters.
          processContextId,str.customerId))" type="context" />
        <parameter value="100000" type="int"/>
      </context_function_parameters>
    </context_constraint>
    <context_constraint cc_id="cc:cc2">
      <context_function_id id="more-than"/>
      <context_function_parameters>
        <parameter value="cst.getCustomerExternalRating(sCtx.getProcessVariable(parameters.
          processContextId,str.customerId))" type="context" />
        <parameter value="100000" type="int"/>
      </context_function_parameters>
    </context_constraint>
    <!-- For Requirement 3 (Part 1) -->
    <context_constraint cc_id="cc:cc3">
      <context_function_id id="not-equal"/>
      <context_function_parameters>
        <parameter value="cst.getCustomerType(sCtx.getProcessVariable(parameters.processContextId
          ,str.customerId))" type="context" />
        <parameter value="industrial" type="string" />
      </context_function_parameters>
    </context_constraint>
    <!-- For Requirement 4 (Part 1) -->
    <context_constraint cc_id="cc:cc4">
      <context_function_id id="equal-or-more-than"/>
      <context_function_parameters>
        <parameter value="cst.getCustomerInternalRating(sCtx.getProcessVariable(parameters.
          processContextId,str.customerId))" type="context" />
        <parameter value="0" type="int"/>
      </context_function_parameters>
    </context_constraint>
  </context_constraints>

```

```

<context_constraint_assignments>
  <pacc cc_id="cc:cc1" permission_id="permission:release_ratingreport"
    role_id="role:clerk_postprocessor"/>
  <pacc cc_id="cc:cc2" permission_id="permission:release_ratingreport"
    role_id="role:supervisor"/>
  <pacc cc_id="cc:cc1" permission_id="permission:commit_productbundle"
    role_id="role:clerk_postprocessor"/>
  <pacc cc_id="cc:cc2" permission_id="permission:commit_productbundle"
    role_id="role:supervisor"/>
</context_constraint_assignments>
</module_exo_context_policy>
</policy_object_modules>
</policy_object>

```

A.2 Übersetzte XACML-Policy

Die folgende XACML-Policy ist das Ergebnis der Übersetzung der obigen ORKA-Policy. In der syntaktisch vollständigen Fassung würde diese Policy die folgenden 30 Seiten einnehmen, deshalb ist sie (wie in dieser Arbeit üblich) verkürzt dargestellt (vgl. Abschnitt 4.3). Dadurch sind es „nur noch“ etwa 12 Seiten.

Die Gesamtpolicy besteht aus zwei Teilen, die so kombiniert werden, dass die erste anwendbare Policy über den Zugriff entscheidet (Algorithmus: **first-applicable**).

Die erste Teilpolicy betrifft das Modul „Context Constraints“, erkennbar an den IDs, die mit **orka:id/cc** beginnen. Diese Policy ist das Ergebnis der Übersetzung in Abschnitt 5.9. Jedes **<Rule>**-Element enthält den Anwendungsbereich **<Target>**, in dem festgelegt wird, für welche Subjekte, Ressourcen und Aktionen der CC erfüllt sein muss, sowie in der **<Condition>** die negierte Bedingung des CC. Sobald einer der CCs nicht erfüllt ist, wird der Zugriff insgesamt verweigert (Algorithmus: **deny-overrides**).

Die zweite Teilpolicy wurde aus dem ORKA-Modul „RBAC-Core“ übersetzt (vgl. Abschnitt 5.7), erkennbar an den IDs, die mit **orka:id/rbaccore** beginnen. Ein Zugriff wird hier erlaubt, wenn die drei Attribute **subject-id**, **resource-id** und **action-id** aus der Anfrage mit den in einer beliebigen Regel festgelegten Attributwerten übereinstimmen (Algorithmus: **permit-overrides**).

```

(PolicySet PolicySetId="orka:id/all0" PolicyCombiningAlgId="first-applicable")
  <Target>
    <Subjects>
      <AnySubject/>
    </Subjects>
    <Resources>
      <AnyResource/>
    </Resources>
    <Actions>
      <AnyAction/>
    </Actions>
  </Target>
  <Policy PolicyId="orka:id/cc4" RuleCombiningAlgId="deny-overrides">
    <Target>

```

```

<Subjects>
  <AnySubject/>
</Subjects>
<Resources>
  <AnyResource/>
</Resources>
<Actions>
  <AnyAction/>
</Actions>
</Target>
<Rule RuleId="orka:id/cc0" Effect="Deny">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:clerk_postprocessor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>RatingReport</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>release</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
  <Condition FunctionId="not">
    <Apply FunctionId="integer-less-than-or-equal">
      <Apply FunctionId="integer-one-and-only">
        <EnvironmentAttributeDesignator AttributId="orka:cparam#cst.getCustomerExternalRating%28
          sCtx.getProcessVariable%28parameters.processContextId%2Cstr.customerId%29%29"/>
      </Apply>
      <AttributeValue>100000</AttributeValue>
    </Apply>
  </Condition>
</Rule>
<Rule RuleId="orka:id/cc1" Effect="Deny">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:supervisor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>RatingReport</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>release</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
  <Condition FunctionId="not">
    <Apply FunctionId="integer-greater-than">
      <Apply FunctionId="integer-one-and-only">

```

A Beispiel

```
<EnvironmentAttributeDesignator Attributeld="orka:cfparam#cst.getCustomerExternalRating%28
  sCtx.getProcessVariable%28parameters.processContextId%2Cstr.customerId%29%29"/>
</Apply>
<AttributeValue>100000</AttributeValue>
</Apply>
</Condition>
</Rule>
<Rule RuleId="orka:id/cc2" Effect="Deny">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:clerk_postprocessor</AttributeValue>
      <SubjectAttributeDesignator Attributeld="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>ProductBundle</AttributeValue>
      <ResourceAttributeDesignator Attributeld="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>commit</AttributeValue>
      <ActionAttributeDesignator Attributeld="action-id"/>
    </ActionMatch>
  </Target>
  <Condition FunctionId="not">
    <Apply FunctionId="integer-less-than-or-equal">
      <Apply FunctionId="integer-one-and-only">
        <EnvironmentAttributeDesignator Attributeld="orka:cfparam#cst.getCustomerExternalRating%28
          sCtx.getProcessVariable%28parameters.processContextId%2Cstr.customerId%29%29"/>
        </Apply>
        <AttributeValue>100000</AttributeValue>
      </Apply>
    </Condition>
  </Rule>
<Rule RuleId="orka:id/cc3" Effect="Deny">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:supervisor</AttributeValue>
      <SubjectAttributeDesignator Attributeld="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>ProductBundle</AttributeValue>
      <ResourceAttributeDesignator Attributeld="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>commit</AttributeValue>
      <ActionAttributeDesignator Attributeld="action-id"/>
    </ActionMatch>
  </Target>
  <Condition FunctionId="not">
    <Apply FunctionId="integer-greater-than">
      <Apply FunctionId="integer-one-and-only">
        <EnvironmentAttributeDesignator Attributeld="orka:cfparam#cst.getCustomerExternalRating%28
          sCtx.getProcessVariable%28parameters.processContextId%2Cstr.customerId%29%29"/>
        </Apply>
        <AttributeValue>100000</AttributeValue>
      </Apply>
    </Condition>
  </Rule>
```

```

    </Apply>
  </Condition>
</Rule>
</Policy>
<Policy PolicyId="orka:id/rbaccore30" RuleCombiningAlgId="permit-overrides">
  <Description>rbac-core</Description>
  <Target>
    <Subjects>
      <AnySubject/>
    </Subjects>
    <Resources>
      <AnyResource/>
    </Resources>
    <Actions>
      <AnyAction/>
    </Actions>
  </Target>
  <Rule RuleId="orka:id/rbaccore0" Effect="Permit">
    <Target>
      <SubjectMatch MatchId="string-equal">
        <AttributeValue>role:clerk_preprocessor</AttributeValue>
        <SubjectAttributeDesignator AttributId="orka:attr/role"/>
      </SubjectMatch>
      <ResourceMatch MatchId="string-equal">
        <AttributeValue>CustomerData</AttributeValue>
        <ResourceAttributeDesignator AttributId="resource-id"/>
      </ResourceMatch>
      <ActionMatch MatchId="string-equal">
        <AttributeValue>query</AttributeValue>
        <ActionAttributeDesignator AttributId="action-id"/>
      </ActionMatch>
    </Target>
  </Rule>
  <Rule RuleId="orka:id/rbaccore1" Effect="Permit">
    <Target>
      <SubjectMatch MatchId="string-equal">
        <AttributeValue>role:clerk_preprocessor</AttributeValue>
        <SubjectAttributeDesignator AttributId="orka:attr/role"/>
      </SubjectMatch>
      <ResourceMatch MatchId="string-equal">
        <AttributeValue>CustomerData</AttributeValue>
        <ResourceAttributeDesignator AttributId="resource-id"/>
      </ResourceMatch>
      <ActionMatch MatchId="string-equal">
        <AttributeValue>update</AttributeValue>
        <ActionAttributeDesignator AttributId="action-id"/>
      </ActionMatch>
    </Target>
  </Rule>
  <Rule RuleId="orka:id/rbaccore2" Effect="Permit">
    <Target>
      <SubjectMatch MatchId="string-equal">
        <AttributeValue>role:clerk_postprocessor</AttributeValue>
        <SubjectAttributeDesignator AttributId="orka:attr/role"/>
      </SubjectMatch>
    </Target>
  </Rule>
</Policy>

```

A Beispiel

```
</SubjectMatch>
<ResourceMatch MatchId="string-equal">
  <AttributeValue>RatingReport</AttributeValue>
  <ResourceAttributeDesignator AttributId="resource-id"/>
</ResourceMatch>
<ActionMatch MatchId="string-equal">
  <AttributeValue>prepare</AttributeValue>
  <ActionAttributeDesignator AttributId="action-id"/>
</ActionMatch>
</Target>
</Rule>
<Rule RuleId="orka:id/rbaccore3" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:clerk_postprocessor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>RatingReport</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>post</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore4" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:clerk_postprocessor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>RatingReport</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>query</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore5" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:supervisor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>RatingReport</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
```

```

    <AttributeValue>update</AttributeValue>
    <ActionAttributeDesignator AttributId="action-id"/>
  </ActionMatch>
</Target>
</Rule>
<Rule RuleId="orka:id/rbaccore6" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:clerk_postprocessor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>ProductBundle</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>query_avail_prod</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore7" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:clerk_postprocessor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>ProductBundle</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>modify</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore8" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:clerk_postprocessor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>Contract</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>print</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore9" Effect="Permit">

```

A Beispiel

```
<Target>
  <SubjectMatch MatchId="string-equal">
    <AttributeValue>role:customer</AttributeValue>
    <SubjectAttributeDesignator AttributId="orka:attr/role"/>
  </SubjectMatch>
  <ResourceMatch MatchId="string-equal">
    <AttributeValue>Contract</AttributeValue>
    <ResourceAttributeDesignator AttributId="resource-id"/>
  </ResourceMatch>
  <ActionMatch MatchId="string-equal">
    <AttributeValue>sign</AttributeValue>
    <ActionAttributeDesignator AttributId="action-id"/>
  </ActionMatch>
</Target>
</Rule>
<Rule RuleId="orka:id/rbaccore10" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:manager</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>Contract</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>sign</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore11" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:manager</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>Contract</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>update</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore12" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:clerk_postprocessor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
```

```

    <AttributeValue>Account</AttributeValue>
    <ResourceAttributeDesignator AttributId="resource-id"/>
  </ResourceMatch>
  <ActionMatch MatchId="string-equal">
    <AttributeValue>open</AttributeValue>
    <ActionAttributeDesignator AttributId="action-id"/>
  </ActionMatch>
</Target>
</Rule>
<Rule RuleId="orka:id/rbaccore13" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:clerk_postprocessor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>RatingReport</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>release</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore14" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:supervisor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>RatingReport</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>release</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore15" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:clerk_postprocessor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>ProductBundle</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>commit</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>

```

A Beispiel

```
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore16" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:supervisor</AttributeValue>
      <SubjectAttributeDesignator AttributeId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>ProductBundle</AttributeValue>
      <ResourceAttributeDesignator AttributeId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>commit</AttributeValue>
      <ActionAttributeDesignator AttributeId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore17" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:clerk_preprocessor</AttributeValue>
      <SubjectAttributeDesignator AttributeId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>task:inputCustomerData</AttributeValue>
      <ResourceAttributeDesignator AttributeId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>end</AttributeValue>
      <ActionAttributeDesignator AttributeId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore18" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:clerk_preprocessor</AttributeValue>
      <SubjectAttributeDesignator AttributeId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>task:customerIdentification</AttributeValue>
      <ResourceAttributeDesignator AttributeId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>end</AttributeValue>
      <ActionAttributeDesignator AttributeId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore19" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
```

```

    <AttributeValue>role:clerk_preprocessor</AttributeValue>
    <SubjectAttributeDesignator AttributId="orka:attr/role"/>
  </SubjectMatch>
  <ResourceMatch MatchId="string-equal">
    <AttributeValue>process:BankingProcess</AttributeValue>
    <ResourceAttributeDesignator AttributId="resource-id"/>
  </ResourceMatch>
  <ActionMatch MatchId="string-equal">
    <AttributeValue>createProcess</AttributeValue>
    <ActionAttributeDesignator AttributId="action-id"/>
  </ActionMatch>
</Target>
</Rule>
<Rule RuleId="orka:id/rbaccore20" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:clerk_postprocessor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>process:BankingProcess</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>createProcess</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore21" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:clerk_postprocessor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>task:checkcreditworthiness</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>end</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore22" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:clerk_postprocessor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>task:internalRating</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
  </Target>
</Rule>

```

A Beispiel

```
</ResourceMatch>
<ActionMatch MatchId="string-equal">
  <AttributeValue>end</AttributeValue>
  <ActionAttributeDesignator AttributId="action-id"/>
</ActionMatch>
</Target>
</Rule>
<Rule RuleId="orka:id/rbaccore23" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:clerk_postprocessor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>task:chooseProduct</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>end</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore24" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:clerk_postprocessor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>task:printContract</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>end</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore25" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:clerk_postprocessor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>task:QueryCreditBureau</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>end</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
```

```

</Rule>
<Rule RuleId="orka:id/rbaccore26" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:supervisor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>process:BankingProcess</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>createProcess</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore27" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:supervisor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>task:checkcreditworthiness</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>end</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore28" Effect="Permit">
  <Target>
    <SubjectMatch MatchId="string-equal">
      <AttributeValue>role:supervisor</AttributeValue>
      <SubjectAttributeDesignator AttributId="orka:attr/role"/>
    </SubjectMatch>
    <ResourceMatch MatchId="string-equal">
      <AttributeValue>task:QueryCreditBureau</AttributeValue>
      <ResourceAttributeDesignator AttributId="resource-id"/>
    </ResourceMatch>
    <ActionMatch MatchId="string-equal">
      <AttributeValue>end</AttributeValue>
      <ActionAttributeDesignator AttributId="action-id"/>
    </ActionMatch>
  </Target>
</Rule>
<Rule RuleId="orka:id/rbaccore29" Effect="Deny"/>
</Policy>
</PolicySet>

```

A Beispiel

B Implementierter Code

```
package org.orka.enforce.lang.xacml;

import static com.sun.xacml.ctx.Result.DECISION_DENY;
import static com.sun.xacml.ctx.Result.DECISION_PERMIT;
import static org.orka.enforce.lang.xacml.util.Util.makeSet;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

import org.orka.enforce.AccessDecisionRequest;
import org.orka.enforce.AccessDecisionResponse;
import org.orka.enforce.PolicyDecisionComponent;
import org.orka.enforce.context.ContextManager;
import org.orka.enforce.db.AuthorizationModule;
import org.orka.enforce.db.PolicyObject;
import org.orka.enforce.lang.xacml.modules.RbacCoreTranslator;
import org.orka.enforce.lang.xacml.modules.RbacRoleHierarchyTranslator;
import org.orka.enforce.lang.xacml.transform.CCTranslator;
import org.orka.enforce.lang.xacml.util.ArgCheck;
import org.orka.enforce.lang.xacml.util.IdGenerator;
import org.orka.enforce.lang.xacml.util.Identifiers;
import org.orka.enforce.lang.xacml.util.Util;
import org.orka.enforce.lang.xacml.util.XacmlUtil;
import org.orka.opl.PolicyObjectTree;
import org.orka.opl.interfaces.IModuleNames;

import com.sun.xacml.AbstractPolicy;
import com.sun.xacml.PDP;
import com.sun.xacml.PDPConfig;
import com.sun.xacml.PolicySet;
import com.sun.xacml.attr.StringAttribute;
import com.sun.xacml.combine.DenyOverridesPolicyAlg;
import com.sun.xacml.combine.PolicyCombiningAlgorithm;
import com.sun.xacml.ctx.Attribute;
import com.sun.xacml.ctx.RequestCtx;
```

```
import com.sun.xacml.ctx.ResponseCtx;
import com.sun.xacml.ctx.Result;
import com.sun.xacml.ctx.Subject;
import com.sun.xacml.finder.AttributeFinder;
import com.sun.xacml.finder.PolicyFinder;

/**
 * The XACML PDC transforms OPL policies into equivalent XACML policies
 * and later matches AccessRequests to them.
 *
 * @author Roland Illig (University of Passau)
 */
public class XacmlPdc extends PolicyDecisionComponent
{
    private static final AccessDecisionResponse DENY = createResponse(
        AccessDecisionResponse.Result.AccessDenied);

    private static final AccessDecisionResponse GRANT = createResponse(
        AccessDecisionResponse.Result.AccessGranted);

    private static final AccessDecisionResponse INVALID = createResponse(
        AccessDecisionResponse.Result.InvalidOrIncompleteRequest);

    /**
     * Creates an immutable AccessDecisionResponse with a specified result.
     */
    private static AccessDecisionResponse createResponse(AccessDecisionResponse.
        Result r)
    {
        return new AccessDecisionResponse(r)
        {
            public void addObligation(Obligation obligation)
            {
                throw new UnsupportedOperationException();
            }
        };
    }

    private final HashMap<Integer, AbstractPolicy> policyMap = new HashMap<
        Integer, AbstractPolicy>();

    @Override
    public boolean initialize()
    {

```

```

        return true;
    }

    private static RequestCtx makeRequest(String s, String op, String ob) {
        Attribute sa = new Attribute(Identifiers.SUBJECT_ID, null, null, new
            StringAttribute(s));
        Attribute opa = new Attribute(Identifiers.ACTION_ID, null, null, new
            StringAttribute(op));
        Attribute oba = new Attribute(Identifiers.RESOURCE_ID, null, null, new
            StringAttribute(ob));
        return new RequestCtx(makeSet(new Subject(makeSet(sa))), Util.makeSet(oba),
            Util.makeSet(opa), Util.makeSet());
    }

    private static AccessDecisionResponse print(AccessDecisionResponse adr, String msg)
    {
        System.out.println("XacmlPdc:␣" + msg);
        return adr;
    }

    /**
     * Checks whether access is granted using an unordered-deny-overrides algorithm.
     */
    @Override
    public AccessDecisionResponse performAccessDecision(
        ContextManager contextManager, AccessDecisionRequest request,
        PolicyObject po)
    {
        if (po == null || contextManager == null || request == null)
            return print(INVALID, "NullPointerException");

        if (!policyMap.containsKey(po.getId()))
            return print(INVALID, "Policy␣not␣found␣in␣map:␣" + po.getId());
        final AbstractPolicy pol = policyMap.get(po.getId());

        AttributeFinder af = new OrkaAttributeFinder(contextManager, new
            PolicyObjectTree(po));
        PolicyFinder pf = new PolicyFinder();
        pf.setModules(Util.makeSet(new OrkaPolicyFinderModule(pol)));
        PDPCConfig pdpc = new PDPCConfig(af, pf, null);
        PDP pdp = new PDP(pdpc);
        RequestCtx req = makeRequest(Util.getSubject(contextManager), request.
            getAction(), request.getTarget());
    }

```

```
// System.out.println("Request: ");
// req.encode(System.out, new Indenter());
// System.out.println("Policy: ");
// pol.encode(System.out, new Indenter());

    ResponseCtx res = pdp.evaluate(req);
    Result result = (Result)res.getResults().iterator().next();
    switch (result.getDecision())
    {
        case DECISION_PERMIT:
            return GRANT;
        case DECISION_DENY:
            return DENY;
    }
    return print(INVALID, "Fallthrough.");
}

@Override
public boolean registerPolicyObject(PolicyObject po)
{
    ArgCheck.requireNonNull(po, "po");
    ArgCheck.requireNonNull(po.getId(), "po.id");
    ArgCheck.requireNonNull(po.getPolicyClass(), "po.policyClass");

    PolicyObjectTree pot = new PolicyObjectTree(po);

    List<AbstractPolicy> policies = new ArrayList<AbstractPolicy>();
    for (AuthorizationModule am : po.getPolicyClass().getAuthorizationModules())
    {
        final String name = am.getName().intern();

        if (name == IModuleNames.RBACCORE)
            policies.add(new RbacCoreTranslator().translate(pot));
        else if (name == IModuleNames.ROLEHIERARCHY)
            policies.add(new RbacRoleHierarchyTranslator().translate(pot));
        else if (name == IModuleNames.EXOCONTEXT)
            policies.add(new CCTranslator().translate(pot));
        else
        {
            throw new IllegalArgumentException("Unknown_policy_name: " + name
                );
        }
    }
}
```

```

        PolicyCombiningAlgorithm deny_overrides = new DenyOverridesPolicyAlg();
        IdGenerator gen = new IdGenerator("all");
        policyMap.put(po.getId(), new PolicySet(gen.next(), deny_overrides, XacmlUtil.
            ANY_TARGET, policies));
        return true;
    }

    @Override
    public boolean unregisterPolicyObject(PolicyObject po)
    {
        ArgCheck.requireNonNull(po, "po");

        if (!policyMap.containsKey(po.getId()))
            return false;

        policyMap.remove(po.getId());
        return true;
    }
}

```

```

package org.orka.enforce.lang.xacml.modules;

import java.net.URI;
import java.util.ArrayList;
import java.util.Vector;

import org.orka.enforce.lang.xacml.util.IdGenerator;
import org.orka.enforce.lang.xacml.util.Identifiers;
import org.orka.enforce.lang.xacml.util.XacmlUtil;
import static org.orka.enforce.lang.xacml.util.Util.makeList;
import org.orka.opl.PolicyObjectTree;
import org.orka.opl.types.RolePermission;

import com.sun.xacml.AbstractPolicy;
import com.sun.xacml.Policy;
import com.sun.xacml.Rule;
import com.sun.xacml.Target;
import com.sun.xacml.TargetMatch;
import com.sun.xacml.attr.AttributeDesignator;
import com.sun.xacml.attr.StringAttribute;
import com.sun.xacml.combine.PermitOverridesRuleAlg;
import com.sun.xacml.cond.Apply;
import com.sun.xacml.ctx.Result;

```

```

public class RbacCoreTranslator {
    private final URI roleId;
    private final IdGenerator gen;

    public RbacCoreTranslator() {
        this(XacmlUtil.ORKA_ROLE, new IdGenerator("rbaccore"));
    }

    /**
     * This constructor is used for the role hierarchy module, since these two
     * modules are translated in very similar ways.
     *
     * @param roleId
     * The name of the role attribute. The attribute differs among
     * RBAC-Core and RBAC-RH.
     * @param gen
     * The generator for unique identifiers. Since this class is also
     * used by the RBAC-RH module, it is configurable.
     */
    protected RbacCoreTranslator(URI roleId, IdGenerator gen) {
        this.roleId = roleId;
        this.gen = gen;
    }

    private static TargetMatch stringEqual(int type, URI attr, String value)
    {
        assert TargetMatch.SUBJECT == AttributeDesignator.SUBJECT_TARGET;
        assert TargetMatch.RESOURCE == AttributeDesignator.RESOURCE_TARGET;
        assert TargetMatch.ACTION == AttributeDesignator.ACTION_TARGET;
        return new TargetMatch(type, XacmlUtil.STRING_EQUAL, new
            AttributeDesignator(type, Identifiers.STRING, attr, true), new
            StringAttribute(value));
    }

    public static Rule makeRule(String role, String op, String ob, Apply cond, URI roleId,
        IdGenerator gen)
    {
        TargetMatch roleMatch = stringEqual(TargetMatch.SUBJECT, roleId, role);
        TargetMatch obMatch = stringEqual(TargetMatch.RESOURCE, Identifiers.
            RESOURCE_ID, ob);
        TargetMatch opMatch = stringEqual(TargetMatch.ACTION, Identifiers.
            ACTION_ID, op);
        Target t = new Target(makeList(makeList(roleMatch)), makeList(makeList(
            opMatch)), makeList(makeList(obMatch)));
    }
}

```

```

        Rule r = new Rule(gen.next(), Result.DECISION_PERMIT, null, t, cond);
        return r;
    }

    private Rule makeRule(String role, String op, String ob, Apply cond) {
        return makeRule(role, op, ob, cond, roleId, gen);
    }

    private Target makePolicyTarget(Vector<RolePermission> rp) {
        // TODO: implementieren, wenn der Rest zu langsam ist.
        return XacmlUtil.ANY_TARGET;
    }

    public AbstractPolicy translate(PolicyObjectTree pot) {
        ArrayList<Rule> rules = new ArrayList<Rule>();
        Vector<RolePermission> pav = pot.getPA();

        for (RolePermission rp : pot.getPA())
            rules.add(makeRule(rp.getRole().getId(), rp.getPermission().getOperationId(),
                               rp.getPermission().getObjectId(), null));

        rules.add(new Rule(gen.next(), Result.DECISION_DENY, null, XacmlUtil.
                        ANY_TARGET, (Apply)null));

        Policy pol = new Policy(gen.next(), new PermitOverridesRuleAlg(), "rbac-core",
                                makePolicyTarget(pav), rules);
        return pol;
    }
}

```

```

package org.orka.enforce.lang.xacml.modules;

import org.orka.enforce.lang.xacml.util.IdGenerator;
import org.orka.enforce.lang.xacml.util.XacmlUtil;

public class RbacRoleHierarchyTranslator extends RbacCoreTranslator {
    public RbacRoleHierarchyTranslator() {
        super(XacmlUtil.ORKA_ROLE_RH, new IdGenerator("rbac-rh"));
    }
}

```

```

package org.orka.enforce.lang.xacml.transform;

```

```
import static org.orka.enforce.lang.xacml.util.Util.makeList;

import java.net.URI;
import java.util.ArrayList;
import java.util.Vector;

import org.orka.enforce.context.Type;
import org.orka.enforce.lang.xacml.modules.RbacCoreTranslator;
import org.orka.enforce.lang.xacml.transform.func.ComparisonTranslator;
import org.orka.enforce.lang.xacml.transform.func.FunctionTranslator;
import org.orka.enforce.lang.xacml.transform.func.TimeBetweenFunctionTranslator;
import org.orka.enforce.lang.xacml.util.ArgCheck;
import org.orka.enforce.lang.xacml.util.IdGenerator;
import org.orka.enforce.lang.xacml.util.XacmlUtil;
import org.orka.opl.PolicyObjectTree;
import org.orka.opl.types.CFParam;
import org.orka.opl.types.ContextConstraint;
import org.orka.opl.types.PermCC;
import org.orka.opl.types.PermRoleCC;
import org.orka.opl.types.Permission;
import org.orka.opl.types.Role;
import org.orka.opl.types.RoleCC;
import org.orka.opl.types.RolePermission;

import com.sun.xacml.AbstractPolicy;
import com.sun.xacml.ParsingException;
import com.sun.xacml.Policy;
import com.sun.xacml.Rule;
import com.sun.xacml.Target;
import com.sun.xacml.UnknownIdentifierException;
import com.sun.xacml.attr.AttributeDesignator;
import com.sun.xacml.attr.AttributeFactory;
import com.sun.xacml.combine.DenyOverridesRuleAlg;
import com.sun.xacml.cond.Apply;
import com.sun.xacml.cond.Evaluatable;
import com.sun.xacml.cond.Function;
import com.sun.xacml.cond.FunctionBase;
import com.sun.xacml.cond.FunctionFactory;
import com.sun.xacml.cond.FunctionTypeException;
import com.sun.xacml.ctx.Result;

public class CCTranslator {

    private static final AttributeFactory af = AttributeFactory.getInstance();
```

```

private static final FunctionFactory ff = FunctionFactory.getGeneralInstance();

private final IdGenerator gen = new IdGenerator("cc");

private Evaluatable translateLiteral(CFParam p) {
    TypeInfo ti = TypeInfo.typeInfoForOrkaType(p.getType());
    try {
        String value = p.getValue();
        // XACML needs the seconds in all time values, ORKA doesn't.
        if (ti.orkaType == Type.TIME && p.getValue().matches("^\\d\\d:\\d\\d\\d$"))
            value += ":00";

        return af.createValue(ti.xacmlName, value);
    } catch (UnknownIdentifierException e) {
        throw new IllegalStateException("Internal_error.");
    } catch (ParsingException e) {
        throw new IllegalArgumentException("Invalid_value.", e);
    }
}

private Evaluatable translateContext(CFParam p, Type guessedType)
{
    TypeInfo ti = TypeInfo.typeInfoForOrkaType(guessedType);
    URI attrName = XacmlUtil.makeAttributeUri(p.getValue());
    AttributeDesignator ad = new AttributeDesignator(AttributeDesignator.
        ENVIRONMENT_TARGET, ti.xacmlName, attrName, true);
    try
    {
        Function one = ff.createFunction(FunctionBase.FUNCTION_NS + ti.
            xacmlShortName + "-one-and-only");
        return new Apply(one, makeList(ad), false);
    } catch (UnknownIdentifierException e)
    {
        throw new IllegalArgumentException(e);
    } catch (FunctionTypeException e)
    {
        throw new IllegalArgumentException(e);
    }
}

private static final ArrayList<FunctionTranslator> templates = new ArrayList<
    FunctionTranslator>();
static {

```

```
        templates.add(new ComparisonTranslator());
        templates.add(new TimeBetweenFunctionTranslator());
    }

    public Evaluatable translate(CFParam p, Type guessedType) {
        if (p.isLiteralValue())
            return translateLiteral(p);
        return translateContext(p, guessedType);
    }

    public boolean isTranslatable(ContextConstraint cc) {
        for (FunctionTranslator ft : templates) {
            if (ft.isTranslatable(cc, this))
                return true;
        }
        return false;
    }

    public Evaluatable translate(ContextConstraint cc) {
        ArgCheck.requireNonNull(cc, "cc");

        for (FunctionTranslator ft : templates) {
            if (ft.isTranslatable(cc, this))
                return ft.translate(cc, this);
        }
        throw new IllegalArgumentException("Cannot translate cc: " + cc);
    }

    private ArrayList<ContextConstraint> getApplicableRCCs(RolePermission pa,
        Vector<RoleCC> rccs)

    {
        Role r = pa.getRole();

        ArrayList<ContextConstraint> result = new ArrayList<ContextConstraint>();
        for (RoleCC rcc : rccs)
        {
            if (r.equals(rcc.getRole()))
                result.add(rcc.getContextConstraint());
        }
        return result;
    }

    private ArrayList<ContextConstraint> getApplicablePACCs(
```

```

        RolePermission pa, Vector<PermRoleCC> paccs)
    {
        Role r = pa.getRole();
        Permission p = pa.getPermission();

        ArrayList<ContextConstraint> result = new ArrayList<ContextConstraint>();
        for (PermRoleCC prcc : paccs)
        {
            if (r.equals(prcc.getRole()) && p.equals(prcc.getPermission()))
                result.add(prcc.getContextConstraint());
        }
        return result;
    }

    private ArrayList<ContextConstraint> getApplicablePCCs(
        RolePermission pa, Vector<PermCC> pccs)
    {
        Permission p = pa.getPermission();

        ArrayList<ContextConstraint> result = new ArrayList<ContextConstraint>();
        for (PermCC pcc : pccs)
        {
            if (p.equals(pcc.getPermission()))
                result.add(pcc.getContextConstraint());
        }
        return result;
    }

    private Rule translate(RolePermission pa, PolicyObjectTree pot,
        ArrayList<ContextConstraint> applicable)
    {
        String role = pa.getRole().getId();
        String op = pa.getPermission().getOperationId();
        String ob = pa.getPermission().getObjectId();

        // XXX: duplicates are currently ignored.
        ArrayList<Evaluable> args = new ArrayList<Evaluable>();
        for (ContextConstraint cc : applicable) {
            // XXX: the CCs in the various relations are merely references to
            // CCs, yet they have the "full" datatype ContextConstraint. Before
            // they can be of any use, they need to be resolved properly.
            args.add(translate(pot.findCC(cc.getId())));
        }
        Apply cond = new Apply(XacmlUtil.AND, args, true);
    }

```

```
        return RbacCoreTranslator.makeRule(role, op, ob, cond, XacmlUtil.ORKA_ROLE,
            gen);
    }

    public AbstractPolicy translate(PolicyObjectTree pot)
    {
        Vector<RolePermission> pas = pot.getPA();
        Vector<RoleCC> rccs = pot.getRCC();
        Vector<PermRoleCC> paccs = pot.getPACC();
        Vector<PermCC> pccs = pot.getPCC();

        ArrayList<Rule> rules = new ArrayList<Rule>();
        for (RolePermission pa : pas) {
            ArrayList<ContextConstraint> applicable = new ArrayList<
                ContextConstraint>();
            applicable.addAll(getApplicableRCCs(pa, rccs));
            applicable.addAll(getApplicablePACCs(pa, paccs));
            applicable.addAll(getApplicablePCCs(pa, pccs));
            rules.add(translate(pa, pot, applicable));
        }
        rules.add(new Rule(gen.next(), Result.DECISION_DENY, null, null, null));

        return new Policy(gen.next(), new DenyOverridesRuleAlg(), new Target(null,
            null, null), rules);
    }
}
```

```
package org.orka.enforce.lang.xacml.transform;

import java.net.URI;
import java.util.ArrayList;
import java.util.HashMap;

import org.orka.enforce.context.Type;

import com.sun.xacml.attr.DateAttribute;
import com.sun.xacml.attr.IntegerAttribute;
import com.sun.xacml.attr.StringAttribute;
import com.sun.xacml.attr.TimeAttribute;

/**
 * Informationen zur Konvertierung zwischen ORKA-Datentypen und
 * XACML-Datentypen.
```

```

*
* @author Roland Illig
*/
public class TypenInfo {

    /**
     * Der Name des Typs in ORKA.
     */
    public final Type orkaType;

    /**
     * Der kurze XACML-Typname. Er wird in Funktionsnamen, wie zum Beispiel
     * string-equal benutzt.
     */
    public final String xacmlShortName;

    /**
     * Der lange XACML-Typname.
     */
    public final URI xacmlName;

    /**
     * Erzeugt eine neue Zuordnung zwischen einem ORKA- und einem
     * XACML-Datentyp.
     *
     * @param orkaType
     * Der ORKA-Typname.
     * @param xacmlShortName
     * Der XACML-Kurzname.
     * @param xacmlName
     * Der XACML-Name.
     */
    public TypenInfo(Type orkaType, String xacmlShortName, URI xacmlName) {
        this.orkaType = orkaType;
        this.xacmlShortName = xacmlShortName;
        this.xacmlName = xacmlName;
    }

    private static final ArrayList<TypenInfo> types = new ArrayList<TypenInfo>();
    static {
        types.add(new TypenInfo(Type.STRING, "string", URI
            .create(StringAttribute.identifier)));
        types.add(new TypenInfo(Type.INT, "integer", URI
            .create(IntegerAttribute.identifier)));
    }
}

```

```

        types.add(new TypeInfo(Type.DATE, "date", URI
            .create(DateAttribute.identifier)));
        types.add(new TypeInfo(Type.TIME, "time", URI
            .create(TimeAttribute.identifier)));
    }
    private static final HashMap<Type, TypeInfo> typesByOrkaName = new HashMap
        <Type, TypeInfo>();
    private static final HashMap<URI, TypeInfo> typesByXacmlName = new HashMap
        <URI, TypeInfo>();
    static {
        for (TypeInfo ti : types) {
            typesByOrkaName.put(ti.orkaType, ti);
            typesByXacmlName.put(ti.xacmlName, ti);
        }
    }

    /**
     * Returns the type information for an ORKA type.
     *
     * @param type
     * The type to be looked up. The name is case-insensitive, so
     * "String" and "string" are equivalent.
     * @return The type information.
     * @throws IllegalArgumentException
     * if there is no type information for the given type.
     */
    public static final TypeInfo typeInfoForOrkaType(Type type) {
        TypeInfo rv = typesByOrkaName.get(type);
        if (rv == null)
            throw new IllegalArgumentException("Unknown type: " + type);
        return rv;
    }

    /**
     * Looks up the type information for an XACML type.
     *
     * @param type
     * The type to be looked up.
     * @return The type information, or <code>null</code>, if no such type
     * exists.
     */
    public static final TypeInfo typeInfoForXacmlType(URI type) {
        return typesByXacmlName.get(type);
    }
}

```

```
}
```

```
package org.orka.enforce.lang.xacml.transform.func;

import org.orka.enforce.lang.xacml.transform.CCTranslator;
import org.orka.opl.types.ContextConstraint;

import com.sun.xacml.cond.Apply;

/**
 * Eine Übersetzungsschablone, um ORKA-Context-Constraints in
 * XACML-Funktionsaufrufe zu übersetzen.
 */
public interface FunctionTranslator {

    /**
     * Prüft, ob der {@link ContextConstraint} übersetzt werden kann.
     *
     * @param cc
     * Der zu übersetzende {@link ContextConstraint}.
     * @param cct
     * Der Übersetzer, mit dessen Hilfe die Parameter übersetzt
     * werden können.
     * @return <code>true</code>, wenn der {@link ContextConstraint}
     * übersetzt werden kann, sonst <code>false</code>.
     */
    public boolean isTranslatable(ContextConstraint cc, CCTranslator cct);

    /**
     * Übersetzt den {@link ContextConstraint} in einen XACML-Funktionsaufruf.
     *
     * @param cc
     * Der zu übersetzende {@link ContextConstraint}.
     * @param cct
     * Der Übersetzer, mit dessen Hilfe die Parameter übersetzt
     * werden können.
     * @return Der äquivalente XACML-Funktionsaufruf.
     * @throws IllegalArgumentException
     * Wenn <i>cc</i> nicht übersetzt werden kann.
     */
    public Apply translate(ContextConstraint cc, CCTranslator cct);
}
```

```
package org.orka.enforce.lang.xacml.transform.func;

import static org.orka.enforce.lang.xacml.util.Util.makeList;

import java.util.Arrays;
import java.util.Vector;

import org.orka.enforce.context.Type;
import org.orka.enforce.lang.xacml.transform.CCTranslator;
import org.orka.enforce.lang.xacml.transform.TypeInfo;
import org.orka.enforce.lang.xacml.util.XacmlUtil;
import org.orka.opl.types.CFParam;
import org.orka.opl.types.ContextConstraint;
import org.orka.opl.types.ContextFunction;

import com.sun.xacml.cond.Apply;
import com.sun.xacml.cond.Evaluatable;
import com.sun.xacml.cond.Function;
import com.sun.xacml.cond.FunctionBase;
import com.sun.xacml.cond.NotFunction;

public class ComparisonTranslator implements FunctionTranslator
{
    /**
     * Determines the common type of a list of types. {@link Type#CONTEXT} is
     * ignored, and all other types in the list must be the same.
     *
     * @return the common type, or null if there is none.
     */
    private static Type commonType(Type... types)
    {
        Type ct = null;
        for (Type t : types) {
            if (t == Type.CONTEXT)
                continue;

            if (ct == null)
                ct = t;
            else if (t != ct) {
                return null;
            }
        }
        return ct;
    }
}
```

```

}

public boolean isTranslatable(ContextConstraint cc, CCTranslator cct)
{
    String fnname = cc.getContextFunctionID();
    ContextFunction fn;
    try
    {
        fn = ContextFunction.getByName(fnname);
    } catch (IllegalArgumentException e)
    {
        return false;
    }
    String[] paramNames = fn.getParamNames();
    if (!Arrays.equals(paramNames, new String[] { "left", "right" }))
        return false;

    // only parameters of the same type can be compared.
    if (commonType(cc.getContextFunctionParameters().get(0).getType(), cc.
        getContextFunctionParameters().get(1).getType()) == null)
        return false;
    return true;
}

private Function makeXacmlFunction(TypeInfo ti, String operation)
{
    String op = operation.intern();

    String xop = (op == "less-than") ? "less-than"
        : (op == "equal-or-less-than") ? "less-than-or-equal"
        : (op == "equal") ? "equal"
        : (op == "equal-or-more-than") ? "greater-than-or-equal"
        : (op == "more-than") ? "greater-than"
        : null;
    if (xop == null)
        throw new IllegalArgumentException("Invalid function: " + op);
    String fnname = FunctionBase.FUNCTION_NS + ti.xacmlShortName + "-"
        + xop;
    return XacmlUtil.getFunction(fnname);
}

public Apply translate(ContextConstraint cc, CCTranslator cct)
{
    String fnname = cc.getContextFunctionID().intern();

```

```

Vector<CFParam> params = cc.getContextFunctionParameters();
Type t0 = params.get(0).getType();
Type t1 = params.get(1).getType();
Type t = commonType(t0, t1);
if (t0 == Type.CONTEXT && t1 == Type.CONTEXT)
    throw new IllegalArgumentException("Cannot compare two values of context type.");
if (t == null)
    throw new IllegalArgumentException("Cannot compare two values of different types: " + t0 + " vs. " + t1 + ".");
TypeInfo ti = TypeInfo.typeInfoForOrkaType(t);

Evaluable xp0 = cct.translate(params.get(0), t);
Evaluable xp1 = cct.translate(params.get(1), t);

if (fnname == "not-equal") {
    Apply eq = new Apply(makeXacmlFunction(ti, "equal"), makeList(xp0, xp1), false);
    Apply not = new Apply(XacmlUtil.getFunction(NotFunction.NAME_NOT), makeList(eq), false);
    return not;
}
return new Apply(makeXacmlFunction(ti, fnname), makeList(xp0, xp1), false);
}
}

```

```

package org.orka.enforce.lang.xacml.transform.func;

import java.util.ArrayList;

import org.orka.enforce.context.Type;
import org.orka.enforce.lang.xacml.transform.CCTranslator;
import org.orka.opl.types.CFParam;
import org.orka.opl.types.ContextConstraint;

import com.sun.xacml.cond.Apply;
import com.sun.xacml.cond.Evaluatable;
import com.sun.xacml.cond.Function;

```

```

/**
 * This class translates a call to the TimeBetween function from ORKA to XACML.
 *
 * @author Roland Illig

```

```

*/
public class TimeBetweenFunctionTranslator implements FunctionTranslator {

    private boolean print(boolean result, String msg) {
        System.err.println(getClass().getName() + ":\u0020" + msg);
        return result;
    }

    public boolean isTranslatable(ContextConstraint cc, CCTranslator cct) {
        if (!cc.getContextFunctionID().equals("time-between"))
            return print(false, "invalid\u0020function");
        for (CFParam p : cc.getContextFunctionParameters()) {
            if (p.getType() != Type.TIME && p.getType() != Type.CONTEXT)
                return print(false, "type\u0020of\u0020a\u0020parameter\u0020is\u0020not\u0020time");
        }
        return true;
    }

    public Apply translate(ContextConstraint cc, CCTranslator cct) {
        if (!isTranslatable(cc, cct))
            throw new IllegalArgumentException("Not\u0020translatable.");
        Function fn = new TimeInRangeFunction();
        ArrayList<Evaluable> args = new ArrayList<Evaluable>();
        // ORKA: time-between time begin end
        // XACML: time-between time begin end
        args.add(cct.translate(cc.getContextFunctionParameters().get(0), Type.TIME));
        args.add(cct.translate(cc.getContextFunctionParameters().get(1), Type.TIME));
        args.add(cct.translate(cc.getContextFunctionParameters().get(2), Type.TIME));
        return new Apply(fn, args, false);
    }
}

```

B.1 Hilfsklassen

```

package org.orka.enforce.lang.xacml.util;

import java.net.URI;

public class IdGenerator
{
    private final String prefix;
    private int id;
}

```

```
/**
 * Creates a new generator for unique IDs.
 *
 * @param prefix
 * A short name indicating the intended use of the IDs, for
 * example "rbacore" or "cc".
 */
public IdGenerator(String prefix)
{
    this.prefix = prefix;
    this.id = 0;
}

public URI next()
{
    return URI.create(XacmlUtil.ORKA_NS + "/id/" + prefix + (id++));
}
}
```

```
package org.orka.enforce.lang.xacml.util;

import java.io.UnsupportedEncodingException;
import java.net.URI;
import java.net.URLDecoder;
import java.net.URLEncoder;
import java.util.ArrayList;

import org.orka.enforce.context.Type;
import org.orka.opl.types.CFParam;

import com.sun.xacml.Target;
import com.sun.xacml.UnknownIdentifierException;
import com.sun.xacml.attr.AttributeDesignator;
import com.sun.xacml.attr.StringAttribute;
import com.sun.xacml.cond.Apply;
import com.sun.xacml.cond.EqualFunction;
import com.sun.xacml.cond.Evaluatable;
import com.sun.xacml.cond.Function;
import com.sun.xacml.cond.FunctionFactory;
import com.sun.xacml.cond.FunctionTypeException;
import com.sun.xacml.cond.LogicalFunction;

public class XacmlUtil {
```

```

/**
 * Der gemeinsame Namensraum für alles, was mit ORKA zu tun hat.
 */
public static final String ORKA_NS = "http://www.orka-projekt.de/xacml";

/**
 * Das Präfix für alle {@link CFParm}s mit Typ {@link Type#CONTEXT}.
 */
public static final String ORKA_CFPARAM = ORKA_NS + "/cfparam#";

/**
 * Die nicht-transitiven Rollen eines Subjekts.
 */
public static final URI ORKA_ROLE = URI.create(ORKA_NS + "/attr/role");

/**
 * Die transitiven Rollen eines Subjekts.
 */
public static final URI ORKA_ROLE_RH = URI.create(ORKA_NS + "/attr/role-
    transitive");

/*
 * Einige oft benötigte XACML-Funktionen.
 */

public static final Function STRING_EQUAL = new EqualFunction(EqualFunction.
    NAME_STRING_EQUAL);
public static final Function AND = new LogicalFunction(LogicalFunction.
    NAME_AND);
public static final Function OR = new LogicalFunction(LogicalFunction.NAME_OR);

public static final Target ANY_TARGET = new Target(null, null, null);

public static Apply stringEqualMatch(String literal, URI attrname, int target) {
    ArrayList args = new ArrayList(2);
    args.add(new StringAttribute(literal));
    args.add(new AttributeDesignator(target, Identifiers.STRING, attrname, true));
    return new Apply(STRING_EQUAL, args, false);
}

public static Apply apply(Function fn, Evaluatable... args) {
    ArrayList<Evaluatable> al = new ArrayList<Evaluatable>(args.length);
    for (Evaluatable arg : args)

```

```
        al.add(arg);
    }
    return new Apply(fn, al, false);
}

public static URI makeAttributeUri(String attr) {
    try {
        return URI.create(ORKA_CFPARAM + URLEncoder.encode(attr, "UTF-8"))
            ;
    } catch (UnsupportedEncodingException e) {
        throw new RuntimeException(e);
    }
}

public static String extractAttributeFromUri(URI uri) {
    String attrURI = uri.toString();
    if (!attrURI.startsWith(ORKA_CFPARAM))
        throw new IllegalArgumentException("Illegal attribute ID: " + attrURI);
    try {
        return URLDecoder.decode(attrURI.substring(ORKA_CFPARAM.length()), "UTF-8");
    } catch (UnsupportedEncodingException e) {
        throw new RuntimeException(e);
    }
}

public static Function getFunction(String name) {
    try {
        return FunctionFactory.getGeneralInstance().createFunction(name);
    } catch (UnknownIdentifierException e) {
        throw new IllegalArgumentException(e);
    } catch (FunctionTypeException e) {
        throw new IllegalArgumentException(e);
    }
}
}
```

Literaturverzeichnis

- [ANSI-RBAC] INCITS. American National Standard for Information Technology – Role Based Access Control, 2004.
- [Anderson2005] Anne Anderson. A Comparison of Two Privacy Languages: EPAL and XACML. <http://research.sun.com/>, 2005.
- [DesignPatterns] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.
- [IEEE754] IEEE Standard for Binary Floating-Point Arithmetic. Technical report, IEEE, 1985.
- [ISO-8601] Data elements and interchange formats – Information interchange – Representation of dates and times. Technical report, International Standards Organization, 2004.
- [RBAC1992] David F. Ferraiolo and D. Richard Kuhn. *Role-Based Access Control*. Artech House, second edition, 2007.
- [Werttypen] Jörg Rathlev. *Ein Werttyp-Konstruktor für Java*. Universität Hamburg, Fachbereich Informatik, 2006.
- [XACML-1.0] Simon Godik (editor) and Tim Moses (editor). eXtensible Access Control Markup Language (XACML) Version 1.0. <http://www.oasis-open.org/committees/xacml/repository/oasis-xacml-1.0.pdf>, 2003.
- [XACML-1.0-intro] N. N. A Brief Introduction to XACML. http://www.oasis-open.org/committees/download.php/2713/Brief_Introduction_to_XACML.html, 2003.
- [XACML-2.0] Tim Moses (editor). eXtensible Access Control Markup Language (XACML) Version 2.0. http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf, 2005.
- [XACML-RBAC] Anne Anderson (editor). XACML Profile for Role Based Access Control (RBAC). <http://docs.oasis-open.org/xacml/cd-xacml-rbac-profile-01.pdf>, 2004.
- [XACMLdiff-1.x-2.0] N. N. Changes Since XACML 1.0. http://www.oasis-open.org/committees/download.php/12821/xacml_1.x_2.0_diffs_draft.doc, 2005.

- [XMLSchema] XML Schema Part 2: Datatypes Second Edition. Technical report, W3C, 2004.

Index

- Action, 12
- ⟨Action⟩, 38, 45, 57, 66
- ⟨ActionAttributeDesignator⟩, 53
- ⟨ActionMatch⟩, 45, 70
- ⟨Actions⟩, 45, 46, 57
- Aktion, 12
- ⟨AllOf⟩, 47, 58
- Anwendungsbereich, 45
- ⟨anyAction⟩, 57
- ⟨AnyOf⟩, 47, 58
- ⟨anyResource⟩, 57
- ⟨anySubject/⟩, 46
- ⟨anySubject⟩, 37, 47, 57
- ⟨Apply⟩, 54, 73
- ⟨Attribute⟩, 41, 83
- ⟨*AttributeDesignator⟩, 53, 85
- ⟨AttributeDesignator⟩, 47
- ⟨*AttributeSelector⟩, 53
- ⟨AttributeValue⟩, 41, 53, 85
- ⟨AttributeValues⟩, 83
- Benutzer, 11
- Berechtigung, 12
- CC* (Context Constraint), 26
- Codebasis, 39
- ⟨Condition⟩, 54, 57, 75, 96
- ⟨Conditions⟩, 65
- Context Constraint (ORKA-Entität), 26
- ⟨Description⟩, 82
- DSoD, 15
- DSoD* (Dynamic Separation of Duty), 30
- Dynamic Separation of Duties, 15
- Dynamic Separation of Duty (ORKA-Relation), 30
- Dynamische Rechtstrennung, 15
- ⟨Environment⟩, 38, 45, 57
- ⟨EnvironmentAttributeDesignator⟩, 53, 54
- ⟨EnvironmentMatch⟩, 57
- ⟨Environments⟩, 45
- ⟨Function⟩, 54
- ⟨*Match⟩, 37, 45–48, 65, 66
- ⟨Match⟩, 47, 58
- ⟨MissingAttributeDetail⟩, 57
- Modul, 19
- OASIS, 7, 33
- Object, 11
- Objekt, 11
- Obligations, 33
- Operation, 12
- OPL, 7, 19
- OPL/XML, 17, 19, 89
- ORKA Policy Language, 19
- P* (Permission), 22
- PA* (Permission Assignment), 22
- PACC* (Permission Assignment Context Constraint), 27
- PAP, 34
- PCC* (Permission Context Constraint), 27
- PDP, 33
- PEP, 33
- Permission, 12
- Permission (ORKA-Entität), 22
- Permission Assignment (ORKA-Relation), 22
- Permission Assignment Context Constraint (ORKA-Relation), 27

- Permission Context Constraint (ORKA-Relation), 27
- PIP, 34, 64
- $\langle \text{Policy} \rangle$, 44, 45, 56, 65, 67, 79
- Policy Administration Point, 34
- Policy Decision Point, 33
- Policy Enforcement Point, 33
- Policy Information Point, 34, 64
- $\langle \text{PolicyIdReference} \rangle$, 75
- $\langle \text{PolicySet} \rangle$, 44, 45, 55, 56, 67

- R* (Role), 22
- RBAC, 13
- RCC* (Role Context Constraint), 26
- $\langle \text{Request} \rangle$, 38
- Resource, 11
- $\langle \text{Resource} \rangle$, 38, 45, 57, 66
- $\langle \text{ResourceAttributeDesignator} \rangle$, 53
- $\langle \text{ResourceMatch} \rangle$, 45, 70
- $\langle \text{Resources} \rangle$, 45, 46, 66
- $\langle \text{Response} \rangle$, 57
- Ressource, 11
- $\langle \text{Result} \rangle$, 57
- RH* (Role Hierarchy), 23
- Role, 11
- Role (ORKA-Entität), 22
- Role Based Access Control, 13
- Role Context Constraint (ORKA-Relation), 26
- Role Hierarchy (ORKA-Relation), 23
- Rolle, 11
- Rollenbasierten Zugriffskontrolle, 13
- $\langle \text{Rule} \rangle$, 45, 67, 74, 75, 77, 96

- S* (Subject), 22
- Session, 11
- SR, 15
- SR* (Subject Roles), 22
- SSoD, 15
- SSoD* (Static Separation of Duty), 29
- SSoDP* (Static Separation of Duty for Permissions), 30
- SSSoD* (Strict Static Separation of Duty), 30

- Static Separation of Duties, 15
- Static Separation of Duty (ORKA-Relation), 29
- Static Separation of Duty for Permissions (ORKA-Relation), 30
- Statische Rechtstrennung, 15
- Strict Static Separation of Duty (ORKA-Relation), 30
- SU*, 15
- SU* (Subject User), 22
- Subject, 11
- $\langle \text{Subject} \rangle$, 37, 38, 45, 57, 66, 70
- Subject (ORKA-Entität), 22
- Subject Roles, 15
- Subject Roles (ORKA-Relation), 22
- Subject User, 15
- Subject User (ORKA-Relation), 22
- $\langle \text{SubjectAttributeDesignator} \rangle$, 45, 47, 53
- $\langle \text{SubjectMatch} \rangle$, 45, 47, 48, 70
- $\langle \text{Subjects} \rangle$, 37, 45, 46, 57, 66
- Subjekt, 11
- sunxacml, 82

- $\langle \text{Target} \rangle$, 37, 44–47, 54, 57, 58, 65–67, 69, 84, 96

- U* (User), 22
- UA, 15
- UA* (User Assignment), 22
- User, 11
- User (ORKA-Entität), 22
- User Assignment, 15
- User Assignment (ORKA-Relation), 22

- $\langle \text{VariableDefinition} \rangle$, 54
- $\langle \text{VariableReference} \rangle$, 54
- Verpflichtungen, 33

- XACML, 7, 89
- XACML 1.0, 33

- Zugriffskontrolle, 11

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Unterschrift

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Departments Informatik einverstanden.

Unterschrift