Effiziente Gestaltung und Anwendung von attributbasierter Zugriffskontrolle für RESTful Services

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Marc Hüffmeyer, M.Sc.

aus Wolfenbüttel

Hauptberichter: Prof. Dr. Bernhard Mitschang
Mitberichter: Prof. Dr. Ulf Schreier
Tag der mündlichen Prüfung: 12.09.2019

Institut für Parallele und Verteilte Systeme der Universität Stuttgart

2019

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, ohne deren Mithilfe die Anfertigung dieser Promotionsschrift niemals zustande gekommen wäre:

Mein besonderer Dank gilt Prof. Dr. Ulf Schreier für die umfangreiche Betreuung dieser Arbeit, die vielfältige und geduldige Unterstützung während der Ausarbeitung und der Vollendung, die konstruktiven Anregungen und insbesondere für die vielen fachlichen und persönlichen Diskussionen, die mir immer in freudiger Erinnerung bleiben werden.

Einen herzlichen Dank möchte ich an Prof. Dr. Berhard Mitschang richten, für die Bereitschaft diese Arbeit zu begleiten, mich während der Entstehung der Arbeit in vielen Forschungsfragen zu beraten und für die wertvollen Hinweise bei der Anfertigung der wissenschaftlichen Beiträge dieser Arbeit.

Des Weiteren möchte ich vielen Kollegen und Mitarbeitern aus der Fakultät Wirtschaftsinformatik und dem Rechenzentrum der Hochschule Furtwangen, dem Institut für Parallele und Verteilte Systeme der Universität Stuttgart und dem Herman-Hollerith-Zentrum danken.

Ihre Anmerkungen, differenzierten Betrachtungen und Ideen haben mir bei der Anfertigung der Dissertation sehr geholfen.

Schließlich möchte ich mich bei meiner Familie, meinen Freunden und meiner Freundin bedanken, die mich stets uneingeschränkt unterstützt haben, zu jeder Zeit vollstes Verständnis für mich aufgebracht haben und sowohl in schönen aber auch in schwierigen Momenten an meiner Seite waren. Ohne ihre immerwährende Unterstützung wäre diese Dissertation nicht möglich gewesen.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ACL Access Control List

ABAC Attribute Based Access Control

ALFA Abbreviated Language for Authorization

CoAP Constrained Application Protocol

DAC Discretionary Access Control

HATEOAS Hypermedia As The Engine Of Application State

HTTP Hypermedia Text Transfer Protocol

IAM Identity & Access Managements

IETF Internet Engineering Task Force

IoT Internet of Things

JSON Java Script Object Notation

MAC Mandatory Access Control

OAuth Open Authorization

OASIS Organization for the Advancement of Structured Information Standards

RBAC Role Based Access Control

RFC Request for Comments

REST Representational State Transfer

RestACL REST Access Control Language

SOAP Simple Object Access Protocol

SoC System on a Chip

UMA User Managed Access

URI Uniform Resource Identifier

URL Uniform Resource Locator

URN Uniform Resource Name

WSDL Web Service Description Language

W3C World Wide Web Consortium

XACML eXtensible Access Control Markup Language

XML eXtensible Markup Language

Zusammenfassung

Diese Arbeit beschäftigt sich mit der effizienten Gestaltung von attributbasierter Zugriffskontrolle in verteilten Systemen. Attributbasierte Zugriffskontrolle ermöglicht die Implementierung von flexiblen und variantenreichen Regelwerken, weshalb dieses Zugriffskontrollmodell ein vielversprechender Kandidat ist, um das dominante Modell der Zukunft zu werden. Der aktuelle Stand der Technik basiert überwiegend auf generischen Mechanismen, die das attributbasierte Modell umsetzen, indem sie kompositionelle Evaluationsmechanismen verwenden.

In der Arbeit wird insbesondere die Eignung von indexbasierten Evaluationsmechanismen gegenüber kompositionellen Mechanismen untersucht. Indexbasierte Evaluationsmechanismen können in Umgebungen eingesetzt werden, die eindeutige Schlüssel innerhalb einer Interaktion verwenden. Die Interaktion mit RESTful Services beispielsweise basiert unter anderem auf der Verwendung von eindeutigen Adressen (URIs). Mit Hilfe dieser Schlüssel lassen sich indexbasierte Verfahren abbilden, die eine effizientere Evaluation ermöglichen. Dies ist insbesondere für große Datenmengen interessant. Der Schwerpunkt der Arbeit liegt auf der Untersuchung geeigneter Verfahren für den

Einsatz bei RESTful Services, was die Entwicklung einer domänenspezifischen Sprache und eines zugehörigen Interpreters beinhaltet. Darüber hinaus werden verschiedene Einsatzmöglichkeiten vorgestellt, bei denen diese Verfahren Anwendung finden.

Abstract

This work investigates the efficient structuring of attribute based access control for distributed systems. Attribute based access control enables flexible and fine-grained access control and is therefore a major candidate to become the most dominant access control model in the near future. The state of the art relies on generic mechanisms which implement the model in a general fashion. These mechanisms employ compositional evaluation algorithm because of their generic design.

This work examines the suitability of index based evaluation algorithms and compares them with compositional approaches. Index based evaluation algorithms can be used in environments which use unique keys, such as RESTful Services. RESTful Services are based on unique addresses (URIs). These addresses can be used in an index based approach to increase the efficiency of the evaluation. This is especially interesting when large amounts of data need to be protected. Besides the investigation on evaluation algorithms also the development of a domain specific language and related interpreters is covered in this work. Also, different application areas for such a system are discussed.

Kapitel 1

Einleitung

In diesem Kapitel wird eine Einführung in diese Arbeit gegeben. Dabei werden zuerst die Motivation sowie die Forschungsfragen und Ziele beschrieben. Danach folgen die Herausforderungen, der Stand der Technik und der Aufbau der Arbeit.

1.1 Motivation

Die Verwaltung von Zugriffsberechtigungen ist ein zentrales Thema verteilter Anwendungen [19]. Heutzutage verwalten Systeme oft große Mengen an Daten und Benutzern, welche komplexe Operationen ausführen. Die Potentiale von diesen Systemen wachsen stetig weiter und mit ihnen der Bedarf nach flexibler und feingranularer Zugriffskontrolle.

Klassische Zugriffskontrollverfahren wurden entwickelt, um in klassischen Umgebungen einfache Sicherheitsmechanismen zu implementieren. Beispielsweise wurde Discretionary Access Control (DAC [83]) entwickelt, um die Frage zu beantworten, welche Subjekte auf einzelne Ressourcen (wie etwa eine Netzwerkschnittstelle oder eine Datei in einem lokalen Betriebssystem) zugreifen dürfen. Rollenbasierte Zugriffskontrolle (Role Based Access Control - RBAC [43]) hingegen ordnet eine Menge von Subjekten einer Rolle zu, um so die benötigte Menge von Einträgen zu verkleinern, welche beschreiben, wer auf eine Ressource zugreifen darf. Bei klassischen Zugriffskontrollverfahren wird also die Frage beantwortet *wer* zugreifen darf [60].

Anwendungen, die komplexe Prozesse abbilden, benötigen in verteilten Umgebungen bei einer Vielzahl an Akteuren jedoch die Möglichkeit, über die Frage hinaus *wer* zugreifen darf, weitere Bedingungen zu formulieren [26]. In heutigen Anwendungen stellt sich die Frage nach dem *was*, *wie*, *warum*, *wo* oder *wann*. Das Treffen von Zugriffsentscheidungen hängt zunehmend vom Kontext ab [106]. Ebenfalls müssen diese Systeme in der Lage sein, sich rasch an neue Gegebenheiten in Form von neuen Ressourcen oder Benutzern anpassen zu können.

Attributbasierte Zugriffskontrolle (Attribute Based Access Control - ABAC) ist ein Modell, das die Formulierung flexibler Regelwerke ermöglicht. Das Modell beschreibt vermutlich den nächsten großen Trend im Bereich der Zugriffskontrolle [124]. Gartner sagt in einer Studie aus dem Jahr 2013 voraus, dass im Jahr 2020 70% der Unternehmen ABAC als Zugriffskontrollmodell einsetzen werden, um schützenswerte Inhalte vor unbefugtem Zugriff abzusichern [145]. Regelwerke, welche mit ABAC umgesetzt werden, sind lediglich durch den Reichtum an Attributen und die implementierenden Mechanismen

beschränkt. Die große Herausforderung, die beim Einsatz von ABAC entsteht, ist das Finden von effizienten Lösungen, um große Mengen verschiedenartiger Zugriffsbedingungen auszuwerten [42].

Representational State Transfer (REST [44]) ist ein bekannter Architekturstil für verteilte Systeme. Er dient dem Entwurf von verteilten Systemen, die gleichzeitig hoch performant und skalierend sein müssen. Um diese Ziele zu erreichen, beschreibt REST einige Merkmale, die ein solches System aufweisen muss. Diese Merkmale sind vom Internet abstrahiert und können als Referenz für den Entwurf verteilter Systeme dienen. REST ist bereits ein weit verbreiteter und etablierter Architekturstil, erfreut sich aber dennoch wachsender Popularität. Ein Grund für die weite Akzeptanz ist die Einfachheit von REST-Schnittstellen. Ein weiterer Grund ist die Tatsache, dass die benötigte Infrastruktur für die Erstellung und den Betrieb von RESTful Services als Teil des Internets bereits weit verbreitet ist. RESTful Services bieten Kommunikation und Interaktion mit Informationssystemen durch weit verbreitete Technologien in einem großen Rahmen.

Da die Popularität von REST weiter zunimmt, wächst auch das Bedürfnis nach feingranularer Zugriffskontrolle in gleichem Maße. Deshalb müssen geeignete Zugriffskontrollverfahren gefunden werden, welche den unbefugten Zugriff auf die Ressourcen eines solchen Systems unterbinden können. Bei REST ist eine große Diversität von Inhalten und zugreifenden Systemen gegeben [10], so dass ein Zugriffskontrollverfahren auch eine breite Variation von Zugriffsberechtigungen abbilden können muss. Dies bedeutet, dass ein flexibles Zugriffskontrollverfahren eingesetzt werden muss, welches den Umgang mit den verschiedenartigen Zugriffsberechtigungen unterstützt.

ABAC ist ein geeigneter Kandidat, um die Anforderungen an ein Zugriffskontrollmodell in einem verteilten System zu erfüllen und

gleichzeitig die Vorteile von REST, wie Flexibilität und Skalierbarkeit, zu unterstützen. Dies bedeutet, dass geeignete Mechanismen gefunden werden müssen, welche ABAC für REST ermöglichen. Beispielsweise ist eine Anforderung an einen solchen Mechanismus, dass die Auswertung der Zugriffsberechtigungen für jede Anfrage an einen Server unabhängig durchgeführt werden kann, damit die Vorteile von REST nicht durch das Zugriffskontrollsystem ausgehebelt werden. Dafür müssen die implementierenden Systeme möglichst effizient bei der Verarbeitung von Zugriffsanfragen sein. Andernfalls kann das Zugriffskontrollsystem zum Flaschenhals in der Kommunikation zwischen Client und Server werden.

1.2 Forschungsfragen und Ziele

Durch die beschriebene Problemstellung ergibt sich eine Reihe von Forschungsfragen, auf die diese Arbeit eine Antwort geben soll:

- Lässt sich eine flexible und feingranulare Autorisierung von Zugriffsanfragen für eine große Menge von Ressourcen in RESTful Services umsetzen?
- Wie lassen sich ein allgemein gültiges Modell und eine Sprache konzipieren, so dass keine von der Anwendung abhängigen Elemente abgebildet werden müssen? Kann Externalisierung der Zugriffskontrolle für RESTful Services umgesetzt werden?
- Welche Architekturen und Algorithmen ermöglichen ein effizientes Abbilden, Bearbeiten und Auswerten dieses Modells/dieser Sprache?

- Welche Merkmale bekannter Autorisierungsmechanismen und -standards lassen sich abbilden? Sind positive und negative Zugriffsberechtigungen, Generalisierungen und Ausnahmen, das Überschreiben von Zugriffsberechtigungen, Gruppierungen von Berechtigungen oder auch Vererbung möglich?

- Wie wird bei der Auswertung von Anfragen, insbesondere bei lesenden Zugriffen, verfahren? Kann eine Entscheidung unabhängig vom Inhalt der angefragten Ressourcen getätigt werden? Ist es sinnvoll und vertretbar, die Inhalte zur Entscheidung heranzuziehen?

Ziel dieser Arbeit ist die Integration von ABAC mit den Merkmalen von REST, um ein effizientes Zugriffskontrollsystem für REST-Umgebungen zu gestalten.

1.3 Klassische Zugriffskontrolle

Klassische Zugriffskontrollmodelle [122, 142, 25] sind nicht für den Einsatz in verteilten Umgebungen konzipiert worden und eignen sich deshalb auch nur bedingt für den Einsatz in ebendiesen Umgebungen, so dass sie oft den dortigen Anforderungen nicht gerecht werden können.

Beim Einsatz eines klassischen Modells außerhalb seines ursprünglich angedachten Zwecks kommt es daher häufig zu Problemen. Zugriffskontrolllisten (Access Control Lists - ACL) wurden dafür konzipiert, Dateien eines Betriebssystems vor unbefugten Zugriffen zu schützen oder den Zugriff auf bestimmte Hardwarekomponenten wie

etwa Netzwerkschnittstellen abzusichern. Typischerweise ist beim Einsatz von Zugriffskontrolllisten die Anzahl der zugreifenden Subjekte sehr gering. Ist dies nicht der Fall, so können Zugriffskontrolllisten äußerst groß werden. Liegen große Mengen von zugreifenden Subjekten und zu schützenden Objekten vor, so muss für jedes Paar <*Subjekt, Objekt*> ein Listeneintrag erstellt werden. Muss zudem noch berücksichtigt werden wie zugegriffen wird oder unter welchen Bedingungen zugegriffen werden darf, werden weitere Dimensionen in der Zugriffskontrollliste benötigt. Eckert beschreibt dies als problematisch, da die Auswertung solcher Listen ineffizient ist. Insbesondere ist der Einsatz in verteilten System problematisch, da Zugriffskontrolllisten nicht skalieren [36]. Dies gilt insbesondere dann, wenn feingranulare Zugriffskontrolle stattfinden soll.

Um das Problem der großen Anzahl von Einträgen zu lösen, wurde ein rollenbasierter Ansatz entwickelt. RBAC kategorisiert die zugreifenden Subjekte unter verschiedenen Rollen und reduziert damit die Anzahl der benötigten Einträge. Wenn jedoch keine einfache Kategorisierung möglich ist, kann es zu einem Missbrauch des Modells kommen. RBAC benötigt ein sorgfältiges Role-Engineering. Wenn eine große Quantität und Diversität von Zugriffsberechtigungen gegeben ist, kann ein Over-Engineering der Rollen passieren. Unter einem Over-Engineering von Rollen versteht man den ausufernden Einsatz verschiedener Rollen. Dabei werden neue Rollen eingeführt, um Zugriffsberechtigungen abzubilden, obwohl es für die Rollen nicht unbedingt eine fachliche Existenzbegründung gibt. Sie werden lediglich eingeführt, weil unterschiedliche Zugriffsberechtigungen abgebildet werden müssen. Dies führt in Konsequenz zu einer Rollenexplosion, welche dauerhaft nur schwer beherrschbar ist [152, 80]. Rollenexplosion hat die folgenden, wesentlichen Ursachen:

- **Kartesisches Produkt:** Sobald keine einfache Kategorisierung mehr vorgenommen werden kann, steigt die Anzahl der benötigten Rollen, da für jede Rolle verschiedene Ausprägungen benötigt werden. Beispielsweise ist es denkbar, dass eine Bank die Rollen *Angestellter*, *Abteilungsleiter* und *Filialleiter* verwendet. Betreibt die Bank Filialen in Köln, München und Stuttgart, so werden die Rollen *Angestellter in Köln*, *Angestellter in München* und *Angestellter in Stuttgart* benötigt. Die fachlichen Rollen werden in einem kartesischen Produkt mit den Standorten verknüpft, so dass eine deutlich größere Anzahl an Rollen für die Zugriffskontrolle entsteht.

- **Feingranularität:** Hierarchische Rollen sind ein Konzept, um die Anzahl der benötigten Einträge zum Abbilden von Zugriffsberechtigungen weiter zu reduzieren [125]. Beispielsweise ist es denkbar, dass *Abteilungsleiter* alle Berechtigungen besitzen, die auch *Angestellte* besitzen und die Rolle *Abteilungsleiter* deshalb eine Erweiterung der Rolle *Angestellter* ist. In der Praxis kann es nun passieren, dass Rollen sehr feingranular gestaltet werden. Beispielsweise sind die Basisrollen *Zutrittsberechtigte Tresorraum*, *Zugriffsberechtigte IT-Systeme* und *Kreditberater* denkbar, aus denen sich die Rollen *Angestellter* und *Abteilungsleiter* zusammensetzen. Diese Feingranularität lässt sich beliebig weit fortsetzen bis auf einzelne Dokumente, so dass eine schwer überschaubare Menge von hierarchischen Rollen entsteht.

- **Einzigartigkeit:** Viele Subjekte benötigen eine einzigartige Menge an Zugriffsberechtigungen und lassen sich nicht einer existierenden Rolle zuordnen. Die Verwendung einer eigenen

Rolle für ein einzelnes Subjekt widerspricht dem Grundgedanken von rollenbasierter Zugriffskontrolle.

Das Unternehmen Axiomatics beschäftigt sich mit dem Thema der dynamischen Autorisierung und nennt in einem Whitepaper [13] verschiedene Vorteile, die sich durch den Einsatz von attributbasierter Zugriffskontrolle ergeben. Externalisierung ist ein entscheidender Vorteil und beschreibt die Trennung vom Management der Zugriffsberechtigungen und dem Softwareentwicklungsprozess. Da klassische Zugriffskontrollverfahren oft nicht flexibel genug sind, wird die Zugriffskontrolllogik heutzutage häufig als Teil des Quellcodes einer Anwendung implementiert. Dies bedeutet, dass die Praxis häufig aus individuell entwickelten Lösungen besteht. Individuelle Lösungen sind anfälliger für Fehler, da sie nicht mit der gleichen Intensität getestet wurden, wie es bei weit verbreiteten und etablierten Lösungen der Fall ist.

Zudem ist die Einbindung in den Softwareentwicklungsprozess nur möglich, wenn die Zugriffsberechtigungen eine gewisse Statik aufweisen. Ändern sich die Anforderungen an das Zugriffskontrollsystem dynamisch, muss bei jeder Änderung in den Softwareentwicklungsprozess eingegriffen werden. Dies bedeutet, dass Zugriffsberechtigungen nur in einem sehr starren Rahmen formuliert werden können oder sehr hohe Aufwände einkalkuliert werden müssen, um den Softwareentwicklungsprozess entsprechend aufwendig zu gestalten.

ABAC bietet die Möglichkeit deklarative Autorisierung durchzuführen und flexibel formulierbare Zugriffsregeln in externe Regelwerke auszulagern. Mit einem solchen Ansatz ist es nicht mehr notwendig, die Zugriffsberechtigungen im Quellcode einer Anwendung zu verankern und bei Änderungen von Zugriffsberechtigungen in den

Softwareentwicklungsprozess eingreifen zu müssen. Anwendungslogik und Zugriffskontrolllogik werden hierdurch sauber voneinander abgegrenzt. Über klar definierte Schnittstellen können von der Anwendungslogik Zugriffsentscheidungen beim Zugriffskontrollsystem erfragt werden.

1.4 Aufbau der Arbeit

Nachdem im ersten Kapitel die Motivation, die Forschungsfragen und Ziele sowie der Stand der Technik behandelt wurden, folgen im zweiten Kapitel die Grundlagen der Arbeit. Diese umfassen allgemeine Zusammenhänge und Architekturen des Identity & Access Managements (IAM), die eXtensible Access Control Markup Language (XACML) als bekannten Vertreter eines attributbasierten Zugriffskontrollmechanismus und die Prinzipien von Representational State Transfer (REST). Außerdem werden OAuth und User Managed Access als Vertreter für Zugriffskontrolle bei RESTful Services vorgestellt.

Die wissenschaftlichen Beiträge dieser Arbeit werden in den Kapiteln 3 bis 6 vorgestellt. In Kapitel 3 wird XACML untersucht und Optimierungsstrategien für den Einsatz bei RESTful Services beschrieben. Aus den Ergebnissen dieser Untersuchung ist die REST Access Control Language (RestACL) entstanden, welche in Kapitel 4 detailliert beschrieben wird. Ein formeller Vergleich von XACML und RestACL in Kapitel 5 dient dazu, die Korrektheit und Vollständigkeit von RestACL zu beweisen und liefert gleichzeitig Beiträge, um die Semantik von XACML zu verdeutlichen. Schließlich werden in Kapitel 6 drei Anwendungsgebiete für RestACL vorgestellt, die Beiträge zur Untersuchung von REST und zur Modellierung von Zugriffskontrollsystemen

liefern. In Kapitel 7 werden die Antworten auf die Forschungsfragen gegeben und die wissenschaftlichen Beiträge zusammengefasst. Außerdem wird ein Ausblick auf zukünftige Forschungsarbeiten geliefert.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen der Arbeit beschrieben. Dabei werden zunächst allgemeine Zusammenhänge des Identity & Access Managements erläutert. Danach folgen detaillierte Beschreibungen der eXtensible Access Control Markup Language und des Architekturstils REST.

2.1 Identity & Access Management

Viele der heutigen Informationssysteme und Anwendungen verwalten große Mengen von Benutzern und deren Daten. Hierbei wird den Benutzern die Möglichkeit gegeben, eigene Inhalte (beispielsweise in Form von Dateien oder Dokumenten) für andere Benutzer innerhalb einer Anwendung oder über deren Grenzen hinaus bereitzustellen. Ein

substanzieller Bedarf, den Zugriff auf diese Inhalte zu kontrollieren und einzuschränken, ist die natürliche Konsequenz. Da die Daten der Benutzer nicht mehr lokal gespeichert werden, muss ein Identitäts- und Zugriffsmanagement (Identity & Access Management - IAM) durchgeführt werden, um die Daten vor unbefugten Zugriffen zu schützen.

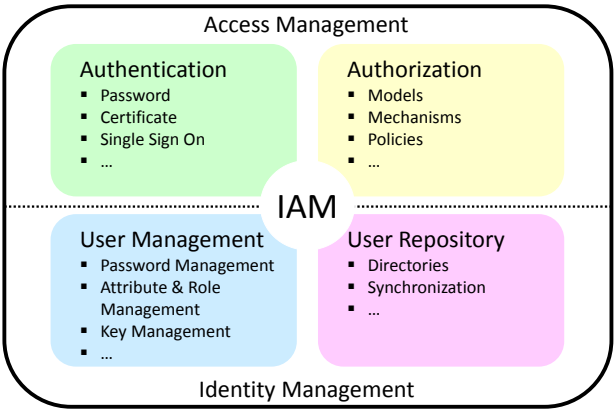

Abbildung 2.1: Aufgaben des Identity & Access Management

Abbildung 2.1 zeigt die wesentlichen Aufgaben des Identity & Access Management. Access Management beschäftigt sich mit den beiden großen Themengebieten Authentifizierung (Authentication) und Autorisierung (Authorization). Authentifizierung beschreibt ganz allgemein den Nachweis einer bestimmten Eigenschaft einer Entität. Eine Entität kann dabei ein beliebiges Subjekt oder Objekt sein wie etwa eine Person, eine Maschine oder eine Organisation. In der Infor-

matik versteht man unter Authentifizierung in der Regel den Nachweis, dass das Gegenüber auch tatsächlich dasjenige ist, als das es sich ausgibt. Das Gegenüber muss also einen Identitätsnachweis erbringen. Hierfür kommen beispielsweise Passwörter oder Zertifikate zum Einsatz. Unter Autorisierung versteht man das Einräumen und/oder Verweigern von Rechten. Ein Autorisierungssystem besteht aus einem Modell, einem Mechanismus, welcher das Modell implementiert und einem Regelwerk zur Abbildung von Berechtigungen.

Identity Management beschäftigt sich mit der Verwaltung von Identitäten. Hierzu gehört die Verwaltung von Identitätsnachweisen wie Passwörtern oder Zertifikaten genauso wie die Verwaltung aller übrigen Eigenschaften der Entitäten. Ebenfalls gehört die sichere Aufbewahrung, also der Schutz der Identitäten, zum Identity Management. Diese Identitäten können beispielsweise in einem User Repository aufbewahrt werden.

2.1.1 Komponenten und Architekturen

Möchte ein Client auf die Dienste eines Servers zugreifen, so ergibt sich die Frage, ob der Client hierzu berechtigt ist. Um diese Frage zu beantworten, ist auf Seiten des Servers ein Autorisierungsprozess nötig.

Einige Technologien und Verfahren kombinieren Teilaufgaben des Identity & Access Management in Diensten, welche sowohl vom Client als auch vom Server genutzt werden. Ein Beispiel für ein solches Verfahren ist der Einsatz von Zugriffstoken. Token werden von einer Client-Anwendung angefordert und dienen beim Server als Nachweis einer Identität oder sogar als Nachweis einer Zugriffsberechtigung. Zum Ausstellen eines Token muss der Dienst, der das Token ausstellt,

Zugriff auf die Identität des Client und gegebenenfalls ein Regelwerk, welches Zugriffsberechtigungen abbildet, haben.

Details zum Nachrichtenaustausch und Informationsfluss sind abhängig vom Anwendungsfall und den eingesetzten Technologien. Um die Abhängigkeiten zwischen den Komponenten zu verdeutlichen, sollen exemplarisch die Informationsflüsse in drei der wichtigsten Architekturen beschrieben werden.

Serverseitiges Identity & Access Management Viele Server- Anwendungen führen sowohl Identity Management als auch Access Management selbst durch. In solchen Systemen registriert sich ein Benutzer typischerweise oder wird durch Dritte angelegt. Hierdurch entsteht eine Identität des Benutzers innerhalb des Identity Managements der Server-Anwendung. Benantar spricht von Local Identities [18]. Diese Identitäten können vom Access Management für die Authentifizierung und Autorisierung ausgewertet werden. Abbildung 2.2 zeigt die Kontrollflüsse in dieser Architektur.

In einem solchen System legt der Benutzer selbst ein Passwort fest oder es wird dem Benutzer ein Passwort zugewiesen, welches im Identity Management der Anwendung sicher verwahrt werden muss. Bei allen folgenden Serviceaufrufen kann der Benutzer sich mit seinem Passwort authentifizieren [138]. Eckert beschreibt das Erbringen eines Identitätsnachweises durch ein Passwort als Authentifizierung durch Wissen [36].

Die Verwaltung von Identitäten kann in dieser Architektur innerhalb von Datenbanken als Teil der Server-Anwendung geschehen. In Webanwendungen kann für die Übermittlung von Passwörtern beispielsweise HTTP Basic oder Digest Authentication [72] verwendet

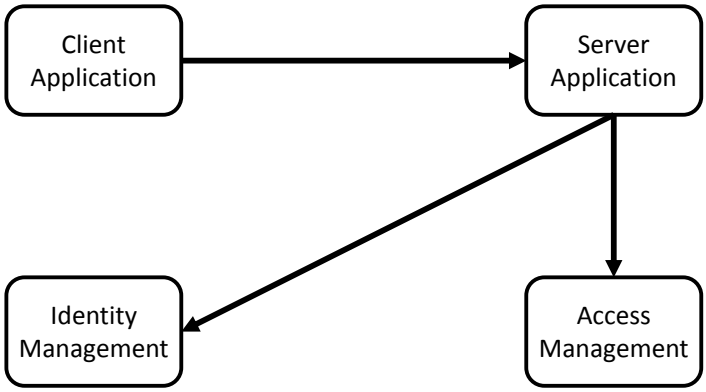

Abbildung 2.2: Serverseitiges Identity & Access Management

werden. Für die Autorisierung können verschiedene Modelle und Mechanismen eingesetzt werden. Rollenbasierte Zugriffskontrolle ist ein häufig eingesetztes Modell. Beispielsweise bietet die Java EE Plattform die Möglichkeit, zu schützende Ressourcen sogenannten Realms zuzuordnen. Authentifizierte Benutzer werden Rollen zugewiesen, die wiederum den Zugriff auf bestimmte Realms ermöglichen.

Ausgelagertes Identity Management Die Verwendung von (Client-)Zertifikaten ist eine Möglichkeit des ausgelagerten Identity Management. Hierbei lässt sich ein Subjekt von einer vertrauenswürdigen dritten Partei zertifizieren. Die dritte Partei übernimmt demnach das Identity Management. Ein Zertifikat dient bei Anfragen eines

Benutzers als Identitätsnachweis. Die Identität kann von der Server-Anwendung bei der dritten Partei überprüft werden. Zertifikate werden häufig eingesetzt, um die Kommunikation zwischen Client und Server zu verschlüsseln [76]. Die Verwaltung von öffentlichen und privaten Schlüsseln ist Teil des Identity Management. Durch ein valides Client-Zertifikat ist die Identität des Clients dem Server bekannt und es kann eine Autorisierung stattfinden. Auch in dieser Architektur können verschiedene Modelle und Mechanismen für die Zugriffskontrolle verwendet werden. Abbildung 2.3 zeigt die Zusammenhänge in dieser Architektur.

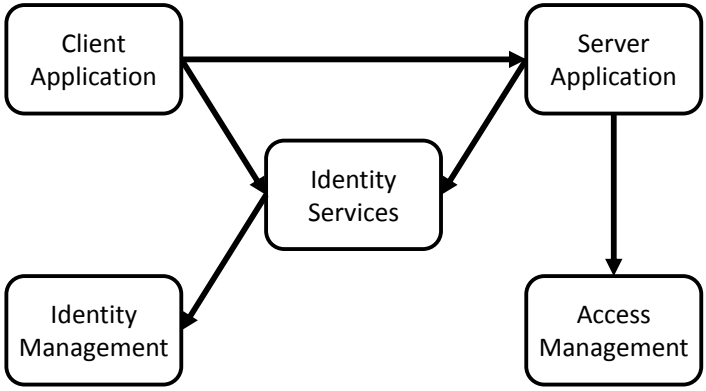

Abbildung 2.3: Ausgelagertes Identity Management

Ausgelagertes Identity & Access Management Durch den Einsatz von Token oder Tickets kann auch das Access Management teilweise oder vollständig ausgelagert werden. Bei diesem Ansatz gewährt die Server-Anwendung nur dann Zugriff, wenn der Client ein gültiges Token beziehungsweise Ticket vorlegt. Um dieses zu bekommen, authentifiziert sich der Client bei einem externen Dienst, welcher in Abhängigkeit der Identität des Client und des Regelwerks entscheidet, ob das Token beziehungsweise Ticket ausgestellt wird.

Verschiedene Protokolle ermöglichen das Auslagern der Authentifizierung. Eckert nennt RADIUS [73] und Kerberos [74] als wichtige Vertreter [36]. Durch das Auslagern der Authentifizierung ist es möglich, dass sich ein Benutzer einmal authentifiziert und mehrere Server-Anwendungen nutzen kann (Single Sign On). Der Benutzer muss sich also nicht bei jeder Server-Anwendung authentifizieren, sondern nur bei einer vertrauenswürdigen dritten Partei, welche Zugriff auf das Identity Management hat. Häufig werden Identitäten in dieser Architektur in Verzeichnissen hinterlegt, welche etwa über das LDAP-Protokoll [75] verwaltet werden. Durch die externe Authentifizierung wird das Access Management teilweise ausgelagert, während die Autorisierung noch immer in der Server-Anwendung statt findet. Es ist jedoch auch möglich, die Autorisierung auszulagern. Verfahren und Technologien wie Shibboleth [130] und Active Directory [96] ermöglichen auch das Hinterlegen von Zugriffsberechtigungen in Verzeichnissen. Abbildung 2.4 zeigt die Abhängigkeiten in dieser Architektur. Die Autorisierung ist dabei entweder auf Seite der Server-Anwendung angesiedelt oder ebenfalls ausgelagert. In ersten Fall besteht eine Verbindung zwischen Server-Anwendung und Authorization Management. Die Dienste für das Identity Management und die Authentifizierung haben dann keine Anbindung zum Authorization Management. Ist das Authorization

Management ebenfalls ausgelagert, so hat hingegen der Server keine Verbindung zum Authorization Management.

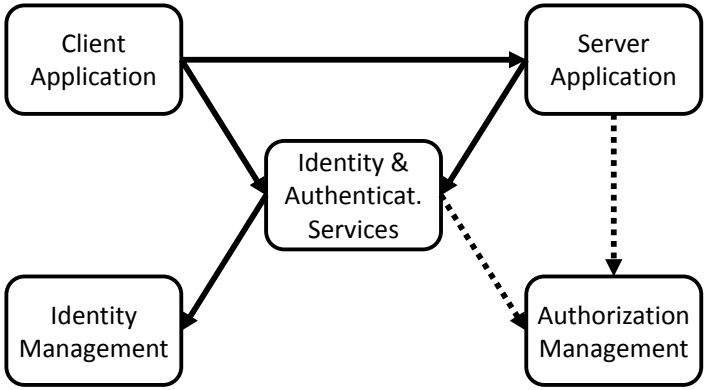

Abbildung 2.4: Ausgelagertes Identity & Access Management

Ein weiteres Beispiel für ein Token-Verfahren im Web ist OAuth [3]. OAuth ist ein Protokoll, welches dem Benutzer das Teilen von Daten über mehrere Anwendungen hinweg erlaubt, ohne dass der Benutzer dabei Passwörter offen legen muss. Bei OAuth übernimmt der Benutzer selbst Teile des Identity & Access Management. Hierfür lässt der Benutzer sich ein Token von einem Dienst der Server-Anwendung ausstellen. Dieses Token kann er an eine Client-Anwendung weiterreichen, welche dann Zugriff auf die Daten des Benutzers in der Server-Anwendung erhält. Details zum OAuth-Protokoll werden in Kapitel 2.4 beschrieben.

2.1.2 Modelle, Mechanismen und Regelwerke

Die Aufgabe eines Zugriffskontrollsystems ist das Anwenden von Regeln [59]. Nun muss zunächst definiert werden, was ein Zugriffskontrollsystem ist. In [122] wird eine klare Abgrenzung zwischen drei fundamentalen Blöcken eines Zugriffskontrollsystems vorgenommen: Ein Zugriffskontrollsystem setzt sich aus Modell, Mechanismus und Regelwerk zusammen.

- **Modell:** ein Zugriffskontrollmodell ist eine formale Beschreibung einer Vorgehensweise zur Umsetzung von Zugriffskontrolle. Ein Modell beschreibt wie Zugriffsberechtigungen und -anfragen formuliert und ausgewertet werden müssen. Das Modell gibt das Verfahren vor, nach denen ein Mechanismus ein Regelwerk auszuwerten hat. Beispielsweise beschreibt Sandhu Modelle für rollenbasierte Zugriffskontrolle [125].

- **Mechanismus:** ein Zugriffskontrollmechanismus ist eine Hardware - oder Softwarekomponente, welche die praktische Anwendung des Modells ermöglicht. Das bedeutet, ein Modell ist die formale Beschreibung, während ein Mechanismus eine praktische Implementierung eines Modells ist. Beispielsweise setzt die Java EE Plattform ein rollenbasiertes Modell um. Dieser Mechanismus kann in vielen Anwendungen wiederverwendet werden.

- **Regelwerk:** ein Regelwerk ist eine Sammlung von Vorschriften (Regeln) zur Umsetzung von Zugriffsberechtigungen. Das heißt, das Regelwerk ist eine Menge von Zugriffsberechtigungen, die zusammen mit einem Mechanismus ein Zugriffskontrollsystem formen. Im Gegensatz zu Modell und Mechanismus

ist das Regelwerk nicht generisch. Das Regelwerk ist also anwendungsbezogen und gilt nur für eine begrenzte Menge von Anwendungen.

2.1.3 Attributbasierte Zugriffskontrolle

Zugriffskontrolle beantwortet üblicherweise die Frage, welche Subjekte welche Aktionen auf welchen Objekten durchführen können [42]. In einer Umgebung, in der eine große Menge von Benutzern eine Vielzahl von Daten teilt, entsteht der Bedarf nach einem flexiblen und hoch performanten Zugriffskontrollsystem, wenn die Benutzer in der Lage sein sollen für ihre Daten detaillierte und spezifische Zugriffsregeln zu formulieren.

Da klassische Zugriffskontrollmodelle wie etwa rollenbasierte Zugriffskontrolle (RBAC) oder Zugriffskontrolllisten (ACL) ursprünglich nicht für diese Einsatzgebiete konzipiert wurden, werden sie dem Anspruch nach Flexibilität bei gleichzeitiger Performanz nicht gerecht. Beispielsweise werden in ACLs eine Menge von Subjekten explizit genannt und gegebenenfalls mit Aktionen kombiniert. Zudem wird eine ACL üblicherweise einem Objekt zugewiesen. Bei steigender Komplexität innerhalb eines Regelwerks, etwa durch eine große Anzahl von Benutzern, ist dieser Mechanismus jedoch nur noch schwer verwaltbar. ACL können zwar für eine beliebige Menge von Objekten definiert werden, bieten jedoch kaum Flexibilität bei der Formulierung von Bedingungen unter denen verschiedene Subjekte zugreifen können, so dass sehr viele Einträge erstellt werden müssen.

RBAC gruppiert deshalb zur Reduktion der Anzahl der Subjekte (und der daraus resultierenden Anzahl von Zugriffsberechtigungen) anhand einer gemeinsamen Eigenschaft der Subjekte: der Rolle [41].

Jedoch führt eine starke Divergenz beim Inhalt der Zugriffsberechtigungen und bei einer gleichzeitig hohen Anzahl an verschiedenen zu schützenden Ressourcen bei RBAC zu einem Over-Engineering der Rollen. Dies bedeutet, dass eine zu große Anzahl an Rollen eingeführt wird, die kaum mehr zu überblicken und zu verwalten sind.

ABAC generalisiert die Idee von RBAC, eine Eigenschaft von Subjekten zur Reduktion der Menge von notwendigen Einträgen zu nutzen. Direkt referenzierte Entitäten werden durch Eigenschaften von Entitäten ersetzt. Die wesentliche Idee hinter ABAC ist die Verwendung beliebiger Eigenschaften von Entitäten zur Formulierung von Zugriffsberechtigungen. Beispielsweise könnte anstelle expliziter Benutzer eine Menge von Benutzern anhand einer Eigenschaft in einer Zugriffsbedingung adressiert werden. Etwa könnte zwischen verschiedenen *Typen* von Benutzern unterschieden werden oder der *Standort* eines Benutzers als Eigenschaft ausgewertet werden. Die Kernidee ist also, dass jede Eigenschaft einer Entität innerhalb von Zugriffsbedingungen ausgewertet werden kann, um eine Zugriffsentscheidung zu fällen.

ABAC bietet die Möglichkeit, sehr komplexe und variantenreiche Zugriffsberechtigungen abzubilden [124]. Klassische Zugriffskontrollmodelle können diese Flexibilität in der Ausdrucksstärke nicht bieten [119]. Dies kann damit begründet werden, dass die klassischen Modelle für bestimmte Einsatzzwecke unter vorgegebenen Bedingungen entworfen wurden. Beispielsweise verwenden Zugriffskontrolllisten als DAC-Ausprägung [83] üblicherweise feste Tupel (etwa das Tripel <*Subject, Action, Object*>). Bei rollenbasierter Zugriffskontrolle existiert eine Bindung an die Rolle, welche die Zugriffsberechtigungen abbildet. Heutzutage reichen diese Modelle jedoch nicht mehr aus, um die Anforderungen der realen Welt abzubilden [40].

Da ABAC ein sehr generisches Modell ist, ist es möglich, den Anwendungszweck von anderen Modellen ebenfalls mit ABAC abzubilden. Beispielsweise können Discretionary Access Control, Mandatory Access Control oder auch Role Based Access Control mit Hilfe von ABAC abgebildet werden. In [80] wird eine formelle Beschreibung des ABAC-Modells vorgenommen. Der Autor zeigt, dass ABAC in der Lage ist, verschiedene andere Zugriffskontrollmodelle abzubilden.

Darüber hinaus können mit ABAC Regelwerke formuliert werden, welche mit klassischen Modellen gar nicht oder nur mit sehr großem Aufwand formuliert werden können. Als Konsequenz lässt sich schließen, dass ABAC ein Modell ist, das ein besonders aussichtsreicher Kandidat für den Einsatz in Umgebungen ist, die eine große Diversität im Regelwerk aufweisen. Das Potential von ABAC wird grundsätzlich nur durch die eingesetzte Sprache und die Mechanismen der implementierenden Systeme beschränkt [42].

Die eXtensible Access Control Markup Language (XACML[5]) ist ein populärer Standard, welcher oftmals verwendet wird, um ein attributbasiertes Zugriffskontrollmodell zu implementieren. XACML ist ein ABAC-Mechanismus und bietet die Flexibilität des unterliegenden ABAC-Modells zur Formulierung von Zugriffskontrollregeln. Details zum XACML werden im folgenden Kapitel beschrieben. Ein propietärer ABAC-Mechanismus für die Zugriffskontrolle auf Dateien und Dateisysteme ist Microsoft Dynamic Access Control [30].

2.2 eXtensible Access Control Markup Language

Die eXtensible Access Control Markup Language (XACML) ist ein Standard [5], der einen möglichen Weg zur Implementierung von ABAC beschreibt. XACML ist ein generischer ABAC-Mechanismus, welcher das ABAC-Modell umsetzt und welcher sich als Standardlösung etabliert hat. Die Flexibilität des Standards ermöglicht es, sehr spezifische Regelwerke zu formulieren. Auf der anderen Seite sind XACML-Regelwerke nur noch sehr schwer beherrschbar und wartbar, sobald eine gewisse Komplexität durch die Menge der Regeln erreicht wird. Zudem wird Expertenwissen benötigt, da die Organisation eines XACML-Regelwerks erheblichen Einfluss auf die Verarbeitungsgeschwindigkeit von Zugriffsanfragen hat. Der XACML-Standard besteht aus drei Teilen:

- **Architekturmuster:** der erste Teil beschreibt ein Architekturmuster, welches verschiedene Komponenten und ihre Verantwortlichkeiten im Autorisierungskontext definiert.
- **Deklarationssprache:** der zweite Teil ist eine Deklarationssprache zur Formulierung von Regelwerken. Das Regelwerk wird in der orginalen Fassung des Standards in XML notiert.
- **Abfragesprache:** der dritte Teil ist eine Abfragesprache zur Formulierung von Zugriffsanfragen und -entscheidungen. Diese werden ebenfalls in XML formuliert.

2.2.1 Architekturmuster

Abbildung 2.5 zeigt die Komponenten der musterhaften Architektur für den Einsatz von XACML. Die Architektur hat keine Kopplung zu der vom Standard beschriebenen Deklarationssprache. Deshalb kann die Architektur auch als Vorlage für die Integration von attributbasierter Zugriffskontrolle in ein System im Allgemeinen verwendet werden.

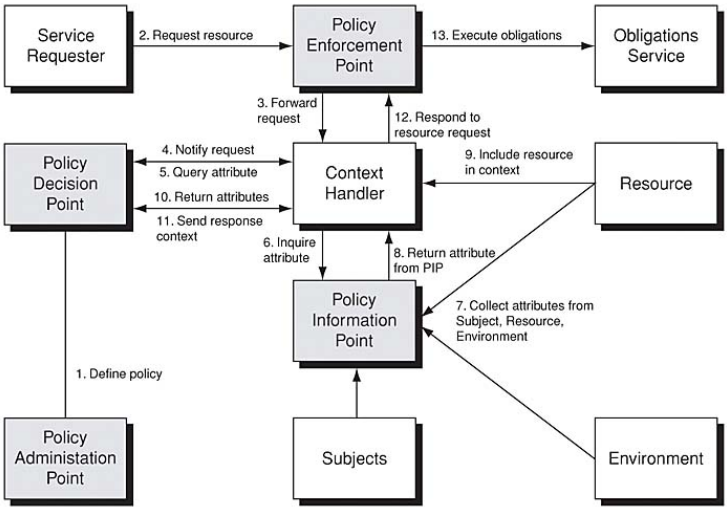

Abbildung 2.5: XACML-Architekturmuster (Quelle: [5])

Die grau unterlegten Komponenten bilden die Basis für das Zugriffskontrollsystem und sind in jedem Fall unverzichtbar.

Der Service Requester repräsentiert das anfragende System, also eine Anwendung, die Zugriffsentscheidungen benötigt. Der Service Requester kontaktiert den Policy Enforcement Point (PEP) und sendet ihm eine Zugriffsanfrage. An dieser Stelle sieht der XACML-Standard keine Definition über das Format der Anfrage vor.

Der PEP ist dafür verantwortlich, die Zugriffsanfrage des Service Requester in das XACML-Format zu überführen. Das Format der Zugriffsanfrage wird in den folgenden Abschnitten beschrieben. Die XACML-konforme Anfrage sendet der PEP an den Context Handler und erhält nach der Auswertung eine Zugriffsentscheidung zurück. Der PEP ist dafür verantwortlich diese Entscheidung durchzusetzen. Das heißt, er muss in Abhängigkeit der Entscheidung den Zugriff ermöglichen oder unterbinden. Hierfür muss der PEP üblicherweise eng mit dem Service Requester verzahnt werden.

Der Context Handler dient dazu, zwischen den verschiedenen Komponenten der Architektur zu vermitteln. Deshalb leitet der Context Handler die vom PEP gestellte Zugriffsanfrage in einem ersten Schritt an einen Policy Information Point weiter.

Die Aufgabe des Policy Information Point (PIP) ist es, die vom PEP gestellte Zugriffsanfrage um zusätzliche Attribute zu erweitern. So ist es beispielsweise denkbar, dass zusätzliche Attribute aus externen Attributquellen geladen werden. Die möglicherweise erweiterte Zugriffsanfrage wird dann zurück zum Context Handler gereicht, welcher die Zugriffsanfrage an die eigentliche Entscheidungskomponente weiterreicht.

Der Policy Decision Point (PDP) verarbeitet die eingehende Anfrage zu einer Zugriffsentscheidung. Dafür evaluiert der PDP die Anfrage nach dem im folgenden Abschnitt beschriebenen Verfahren gegen ein Regelwerk. Das Regelwerk kann vom Policy Administration Point

(PAP) verwaltet werden. Die XACML-Architektur sieht für die Verwaltung des Regelwerks also eine eigene Komponente in der Architektur vor. Nachdem der PDP die Entscheidung gefunden hat, wird diese an den Context Handler zurück geleitet. Der Context Handler wiederum reicht die Entscheidung an den PEP zurück, welcher schlussendlich für die Umsetzung der Entscheidung verantwortlich ist.

Eine optionale Komponente ist der Obligation Service. XACML erlaubt es innerhalb des Regelwerks so genannte Obligations aufzuführen. Kommt es nach der Evaluation zur Anwendung einer Regel mit Obligations, so müssen diese Obligations berücksichtigt werden. Ein einfaches Beispiel für eine Obligation ist das Protokollieren von Zugriffen auf eine Ressource. Beispielsweise ist es denkbar, dass sobald einem Benutzer Zugriff auf eine Ressource gewährt wird, dieser Zugriff protokolliert werden muss. Dazu wird in einer Regel, die den Zugriff auf die Ressource gewährt, eine entsprechende Obligation hinzugefügt. Eine Obligation ist im Wesentlichen ein XML-Element, welches anwendungsspezifisch verarbeitet werden kann. Das heißt, der Obligation-Service muss kontextabhängig in der eigentlichen Anwendung implementiert werden.

2.2.2 Deklarationssprache

Ein in XACML geschriebenes Regelwerk ist in einer Baumstruktur organisiert. In jedem Knoten des Baums können Bedingungen für die Anwendbarkeit der untergeordneten Teilbäume formuliert werden. Die Blätter des Baums können ebenfalls Bedingungen enthalten und müssen einen Effekt (Permit oder Deny) festlegen.

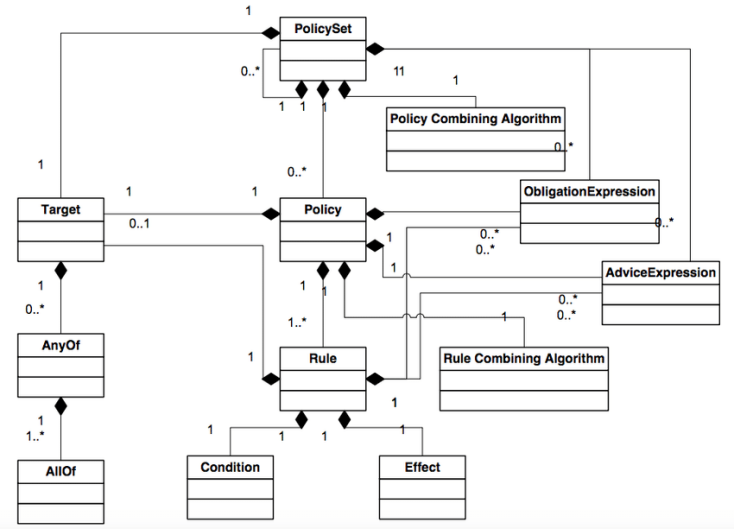

Abbildung 2.6: XACML-Policy-Modell (Quelle: [5])

Policy Sets, Policies und Rules Abbildung 2.6 zeigt das Datenmodell zur Formulierung von Regelwerken mit XACML. Die Wurzel eines Regelwerks kann ein *Policy Set* oder eine *Policy* sein. Policy Sets können Policies oder andere Policy Sets enthalten. Policies wiederum enthalten *Rules*. Diese drei Elemente spannen die Baumstruktur eines XACML-Regelwerks auf. In einem Policy Set, einer Policy und einer Rule muss eine Bedingung in einem sogenannten *Target* definiert werden. Das Target entscheidet über die Anwendbarkeit des Policy Set

beziehungsweise der Policy. Ist die Bedingung im Target durch die Attribute einer Zugriffsanfrage erfüllt, so werden die untergeordneten Policy Sets, Policies und Rules ausgewertet. Wird die Bedingung im Target nicht erfüllt, so werden die untergeordneten Elemente nicht ausgewertet und das Ergebnis der Auswertung dieses Teilbaums ist „nicht anwendbar“. Innerhalb eines Targets werden zum Formulieren von Bedingungen *AnyOf*- und *AllOf*-Elemente verwendet. AnyOf- und AllOf-Elemente sind logische Verknüpfungen der Form *UND* und *ODER*. Ein Target kann beliebig viele AnyOf-Elemente besitzen, welche eine logische Konjunktion bilden. AnyOf-Elemente wiederum müssen mindestens ein AllOf-Element besitzen. Sie können jedoch beliebig viele AllOf-Elemente besitzen, welche eine logische Disjunktion bilden. Das bedeutet, dass in einem Target logische Ausdrücke der Form *UND/ODER/UND* formuliert werden können. In einem AllOf-Element werden *Match*-Elemente definiert. Das Match-Element gibt eine Vergleichsfunktion vor. Außerdem kann es *AttributeValue*-Elemente enthalten, welche feste Werte beschreiben, sowie *AttributeDesignator*-Elemente, welche auf Werte aus der Zugriffsanfrage verweisen.

Rule-Elemente müssen neben einem Target einen *Effect* beinhalten, welcher die Zugriffsentscheidung festlegt. Der Effect kann entweder *Permit* oder *Deny* sein. Außerdem besitzen Sie ein *Condition*-Element, welches neben dem Target weitere Bedingungen beschreiben kann. Conditions sind ausdrucksstärker als Targets, da diese keine festgelegte UND/ODER-Struktur aufweisen müssen. Es kann eine beliebige Struktur zur Verknüpfung von logischen Bedingungen eingesetzt werden. Außerdem erlauben es Condition-Elemente, Attributwerte dynamisch zu vergleichen. Es können also aus der Zugriffsanfrage mehrere Attribute untereinander verglichen werden. In einem Target

kann ein Attributwert aus der Zugriffsanfrage nur mit einem statischen Wert verglichen werden. Es darf also innerhalb eines Match-Elements maximal ein AttributeDesignator-Element vorhanden sein. Dies ist bei Condition-Elementen nicht der Fall. Hier können beliebig viele AttributeDesignator-Elemente eingesetzt werden. Schließlich können in Condition-Elementen beliebige Funktionen, etwa mathematische, auf Attributwerte angewendet werden. Die einzige Bedingung ist, dass die Berechnung schlussendlich einen boolschen Wert zurückliefert.

Combining Algorithm Die Entscheidungsfindung zu einer XACML-Anfrage ist durch zwei wesentliche Schritte geprägt. Zunächst wird das Regelwerk wie zuvor beschrieben von der Wurzel bis hin zu den Blättern ausgewertet. Das heißt, es wird in jedem Knoten geprüft, ob die Attributbedingungen durch die Attribute der Zugriffsanfrage erfüllt sind. Ist dies der Fall, wird der unter dem Knoten liegende Teilbaum weiter ausgewertet. Nach diesem Verfahren gelangt der Auswertungsalgorithmus zu einer Menge von anwendbaren Rules. Jede Rule hat einen Effect, so dass ein Zwischenresultat der Auswertung eine Menge von Effects ist. Diese Menge von Effects muss zu einer einzelnen, finalen Zugriffsentscheidung reduziert werden [5].

Um diese Reduktion durchzuführen, werden sogenannte *Combining Algorithm* eingesetzt. Nachdem die Anwendbarkeit von Rules festgestellt wurde, indem in einem Top-Down-Verfahren der Baum von der Wurzel zu den Blättern durchlaufen wurde, wird die finale Zugriffsentscheidung in einem Bottom-Up-Verfahren bestimmt. Dort, wo anwendbare Knoten einen gemeinsamen, übergeordneten Knoten haben, wird ein Combining Algorithm eingesetzt. Combining Algorithm kommen also in Policy Sets und Policies zum Einsatz. Ein Combi-

ning Algorithm legt fest, welcher Effect der dominantere ist und die Zugriffsentscheidung bestimmt. Beispielsweise könnte eine Policy zwei untergeordnete Rules besitzen von denen jeweils eine den Effect *Permit* und eine den Effect *Deny* hat. Sind beide Rules auf eine Zugriffsanfrage anwendbar, so wird mittels eines Combining Algorithm in der Policy entschieden, wie die Menge der Zugriffsentscheidungen reduziert wird. XACML ist deshalb ein kompositioneller Ansatz. Die finale Zugriffsentscheidung ist eine Komposition von Teilentscheidungen.

Permit	Positive Zugriffsentscheidung
Deny	Negative Zugriffsentscheidung
NotApplicable	Nicht anwendbar
Indeterminate	Unbestimmt - Fehler bei der Verarbeitung

Tabelle 2.1: Mögliche Zugriffsentscheidungen

Rules können die Effects Permit oder Deny haben. Bei der Auswertung eines Targets kann es zu den in Tabelle 2.1 gelisteten Ergebnissen kommen. Neben Permit und Deny ist *NotApplicable* ein weiteres reguläres Ergebnis. *Indeterminate* ist das Ergebnis einer Ausnahmesituation. In diesem Fall ist ein Fehler bei der Auswertung des Targets passiert.

Der Combining Algorithm *PermitUnlessDeny* reduziert eine Menge von Zugriffsentscheidungen nach der in Algorithmus 1 gezeigten Vorschrift. In einem ersten Schritt wird geprüft, ob einer der der Effects aus den Eingangsparametern den Wert Deny hat. In diesem Fall ist die Zugriffsentscheidung Deny. In jedem anderen Fall ist die Zugriffsentscheidung Permit.

Input: Menge von Effects
Output: Einzelner Effect

- Wenn eine Zugriffsentscheidung Deny ist, ist das Resultat Deny.
- In jedem anderen Fall ist das Resultat Permit.

Algorithmus 1: Combining Algorithm PermitUnlessDeny

Für den Combining Algorithm *DenyOverrides* wird eine erweiterte Menge von Zugriffsentscheidungen verwendet. Hier ist es möglich, Fehler bei der Verarbeitung genauer zu bestimmen. Es werden drei Fehlerfälle unterschieden. Tabelle 2.2 zeigt die erweiterte Menge von Zugriffsentscheidungen. Der Combining Algorithm reduziert eine Menge von Zugriffsentscheidungen nach der in Algorithmus 2 gezeigten Vorschrift.

Eine weitere Ausprägung des *DenyOverrides* Algorithmus ist der *OrderedDenyOverrides* Algorithmus, bei dem die Reihenfolge der untergeordneten Elemente bei der Evaluation beachtet werden muss. Analog zu den beschriebenen Algorithmen verhalten sich die Algorithmen *DenyUnlessPermit*, *PermitOverrides* und *OrderedPermitOverrides*. Darüber hinaus definiert XACML in Version 3.0 noch zwei weitere Algorithmen: *FirstApplicable* und *OnlyOneApplicable*. Beim Algorithmus *FirstApplicable* bestimmt das erste anwendbare, untergeordnete Element die Entscheidung. Beim Algorithmus *OnlyOneApplicable* darf nur ein einziges untergeordnetes Element anwendbar sein. Sind mehrere Elemente anwendbar, ist das Ergebnis *Indeterminate*. Die Combing Algorithm werden in [5] - Anhang C definiert.

Input: Menge von Effects
Output: Einzelner Effect

- Wenn eine Zugriffsentscheidung Deny ist, ist das Resultat Deny.
- Andernfalls, wenn eine Zugriffsentscheidung Indeterminate (Permit or Deny) ist, ist das Resultat Indeterminate (Permit or Deny).
- Andernfalls, wenn eine Zugriffsentscheidung Indeterminate (Deny) ist und eine andere Zugriffsentscheidung Indeterminate (Permit) oder Permit ist, ist das Resultat Indeterminate (Permit or Deny).
- Andernfalls, wenn eine Zugriffsentscheidung Indeterminate (Deny) ist, ist das Resultat Indeterminate (Deny).
- Andernfalls, wenn eine Zugriffsentscheidung Permit ist, ist das Resultat Permit.
- Andernfalls, wenn eine Zugriffsentscheidung Indeterminate (Permit) ist, ist das Resultat Indeterminate (Permit).
- Andernfalls, ist das Resultat NotApplicable.

Algorithmus 2: Combining Algorithm DenyOverrides

Permit	Positive Zugriffsentscheidung
Deny	Negative Zugriffsentscheidung
NotApplicable	Nicht anwendbar
Indeterminate (Permit)	Unbestimmt - Fehler bei der Verarbeitung, das Ergebnis hätte Permit sein können, aber nicht Deny
Indeterminate (Deny)	Unbestimmt - Fehler bei der Verarbeitung, das Ergebnis hätte Deny sein können, aber nicht Permit
Indeterminate (Permit or Deny)	Unbestimmt - Fehler bei der Verarbeitung, das Ergebnis hätte Permit oder Deny sein können

Tabelle 2.2: Erweitertes Indeterminate

Beispiel Da die XML-Repräsentation eines XACML-Regelwerks sehr schwer überschaubar ist, hat die Firma Axiomatics eine Notation zur Formulierung von XACML-Regelwerken entwickelt. Diese trägt den Namen Abbreviated Language for Authorization (ALFA) und ist als Working Draft bei der Organization for the Advancement of Structured Information Standards (OASIS) veröffentlicht [9].

Auflistung 2.1 zeigt ein Beispiel eines XACML-Regelwerks in ALFA-Notation. Es beschreibt die Zugriffsbedingungen für den Zugriff auf ein Konto. Das Wurzelelement des Regelwerks ist ein Policy Set, welches anwendbar ist, wenn die Bedingung des zugehörigen Targets erfüllt wird. In diesem Fall muss die Zugriffsanfrage ein Attribut mit dem Namen *resourcePath*, der Kategorie *resource* und dem Wert */account/123* beinhalten. Der *Combining Algorithm* legt fest, dass der

Zugriff verweigert wird, wenn eines der untergeordneten Elemente sich zu *Deny* berechnet. Das einzige untergeordnete Element ist eine Policy, welche anwendbar ist, wenn die Zugriffsanfrage ein Attribut mit dem Namen *method*, der Kategorie *action* und dem Wert *GET* beinhaltet. Die Policy beinhaltet drei Rules und wenn eine der ersten beiden Rules anwendbar ist, so wird der Zugriff gestattet, auch wenn die dritte Rule den Zugriff grundsätzlich verbietet. Bei der ALFA-Notation werden allen grundlegenden XACML-Elementen (PolicySet, Policy und Rule) Namen beziehungsweise Ids zugewiesen, welche auf die optionalen XML-Attribute *policySetId*, *policyId* und *ruleId* übersetzt werden.

```
1  namespace bank {
2    policyset accountAccess {
3      apply denyOverrides
4      target clause Attributes.resourcePath
5        == "/accounts/123"
6
7      policy requestMethod {
8        apply permitOverrides
9        target clause Attributes.actionName == "GET"
10
11       rule grantCustomer456 {
12         permit
13         target clause Attributes.subjectId == 456
14       }
15       rule grantCustomer789 {
16         permit
17         target clause Attributes.subjectId == 789
18       }
```

```
19        rule refuseAnyone {
20          deny
21        }
22    }
23
24    attribute resourcePath {
25      category = resource
26      id = "path"
27      type = string
28    }
29    attribute actionName {
30      category = subject
31      id = "method"
32      type = string
33    }
34    attribute subjectId {
35      category = subject
36      id = "id"
37      type = integer
38    }
39  }
```

Auflistung 2.1: XACML-Regelwerk

Abbildung 2.7 führt eine graphische Notation der Regelwerke ein, welche dabei helfen soll, die Struktur von XACML zu verdeutlichen. In jeder Box wird der Typ des XACML-Elements genannt (Policy Set, Policy oder Rule). Innerhalb der Elemente sind untergeordnete Elemente als eigene Boxen dargestellt. Targets werden als Quadrupel bestehend aus Kategorie, Name, Vergleichsfunktion und erwartetem Wert dargestellt.

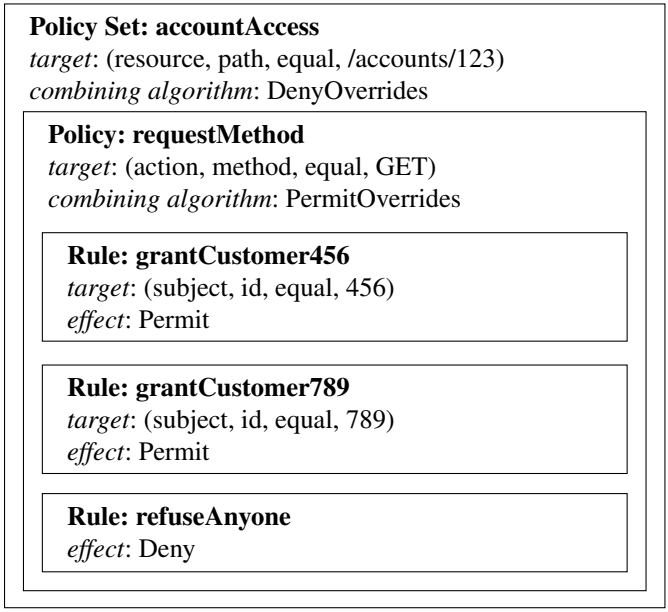

Abbildung 2.7: Graphische Notation für XACML-Regelwerke

2.2.3 Anfragesprache

Eine XACML-Zugriffsanfrage ist eine Menge von Attributen. Die Attribute werden dabei gruppiert nach Kategorie. Auflistung 2.2 zeigt ein Beispiel.

```
{
  "Request" : {
    "Subject" : {
      "Attribute" : {
        "AttributeId" : "type",
        "Value" : "customer"
      },{
        "AttributeId" : "id",
        "Value" : "678"
      }
    },{
      "Action" : {
        "Attribute" : {
          "AttributeId" : "method",
          "Value" : "GET"
        }
      }
    },{
      "Resource" : {
        "Attribute" : {
          "AttributeId" : "path",
          "Value" : "/account/123"
        }
      }
    }
  }
}
```

Auflistung 2.2: XACML-Anfrage

Um die Lesbarkeit zu erhöhen wurde das JSON Profile von XACML verwendet [6]. Das JSON Profile soll eine leichtgewichtige Alternative

zum Formulieren von XACML-Anfragen und Antworten sein und hat den Candidate-Status für einen OASIS-Standard. Wird das JSON Profile nicht verwendet, so nutzt XACML XML zum Formulieren von Anfragen und Antworten.

Die Zugriffsanfrage enthält Attribute der Kategorien *subject*, *action* und *resource*. Attribute besitzen eine *AttributeId* und einen *AttributeValue*. Eine Zugriffsanfrage kann eine beliebige Menge von Attributen enthalten. Diese Attribute sind nicht auf die im Regelwerk existierenden Attribute beschränkt. Im Beispiel ist etwa ein *type*-Attribut für das *subject* angegeben, welches im Regelwerk in Abbildung 2.7 nicht ausgewertet wird.

Eine XACML-Antwort kann neben der Zugriffsentscheidung noch weitere Informationen wie etwa den Status und auferlegte Obligations oder Advices enthalten. Der Status kann beispielsweise Informationen darüber geben, ob alle notwendigen Attribute in der Anfrage enthalten waren oder ob Syntaxfehler vorliegen. Obligations können auferlegt werden, wenn der Zugriff an Verpflichtungen geknüpft ist. Obligations können in Policy Sets, Policies und Rules formuliert werden. Advices sind eine nicht bindende Alternative zu Obligations. Die Syntax ist analog. Im Unterschied zu Obligations können Advices jedoch vom PEP ignoriert werden [12].

Auflistung 2.3 zeigt ein Beispiel. Die Zugriffsentscheidung ist *Deny*, der Status *ok* und eine Obligation verlangt das Loggen des Zugriffsversuchs. Dabei wird ein Attributwert zurück geliefert, welcher geloggt werden muss. Im JSON Profile sind Obligations und Advices in einem Objekt zusammen gefasst. In der auf XML basierenden Version des Standards existieren unterschiedliche Elemente für Obligation und Advice.

```
{
  "Response": [
    {
      "Decision" : "Deny",
      "Status" : {
        "StatusCode" : {
          "Value" : "status:ok"
        },
      "ObligationOrAdvice" : [
        {
          "Id" : "logAccessAttempt"
          "AttributeAssignment" : {
            "Category" : "subject",
            "AttributeId" : "subjectId",
            "Value" : "456"
          }
        }
      ]
    }
  ]
}
```

Auflistung 2.3: XACML-Antwort

2.2.4 REST Profile

Externalisierung ist ein entscheidender Vorteil beim Einsatz von ABAC. Dabei wird die Zugriffskontrolllogik vollständig von der Anwendungslogik getrennt. Ist dies geschehen, kann der PDP in einem Cloud Computing Szenario als Service von der eigentlichen Anwendung ausgelagert werden. Um dies zu ermöglichen, ist das REST Profile

von XACML als Draft zur Standardisierung veröffentlicht worden [7]. Das REST Profile beschreibt, wie XACML als RESTful Service implementiert werden kann. Im Draft wird von *RESTful Authorization as a Service* gesprochen. Das REST Profile definiert hingegen nicht, wie XACML eingesetzt werden kann, um RESTful Services zu schützen. Details zu REST werden im nachfolgenden Abschnitt beschrieben.

Der REST Profile Draft beschreibt das Format, in dem XACML-Anfragen und -Antworten transportiert werden können. In Auflistung 2.4 wird gezeigt, wie eine XACML-Anfrage im Body einer HTTP-POST-Anfrage transportiert werden kann. Es handelt sich um eine übliche POST-Anfrage. Lediglich die *Accept*- und *Content-Type*-Header weisen dedizierte Werte auf.

```
1 POST /authorization/pdp/ HTTP/1.0
2 Accept: application/xacml+xml; version=3.0
3 Content-Type: application/xacml+xml; version=3.0
4
5 <?xml version="1.0"?>
6 <Request>
7   <!-- XACML request -->
8 </Request>
```

Auflistung 2.4: RESTful Authorization as a Service - Anfrage

In Auflistung 2.5 wird die zugehörige Antwort dargestellt. Auch hier handelt es sich um eine normale HTTP-Antwort, die XACML transportiert.

```
1 HTTP/1.0 200 OK
2 Content-Type: application/xacml+xml; version=3.0
3
4 <?xml version="1.0"?>
5 <Response>
6   <!-- XACML response -->
7 </Response>
```

Auflistung 2.5: RESTful Authorization as a Service - Antwort

2.2.5 Stand der Forschung

XACML berechnet Zugriffsentscheidungen zur Laufzeit und muss dabei eine Vielzahl von Attributen auswerten. Deshalb wächst die durchschnittliche Berechnungszeit für eine Zugriffsanfrage mit der Komplexität des Regelwerks. Das Problem der Berechnung der Zugriffsentscheidung zur Laufzeit ist ein generelles Problem, welches der attributbasierte Ansatz mit sich bringt. Sowohl in Attributbedingungen als auch in Zugriffsanfragen können beliebige Attributkategorien, -namen und -werte auftreten. Die Attribute der Zugriffsanfrage mit denen des Regelwerks abzugleichen, ist keine triviale Aufgabe. Insbesondere wenn die Anzahl der Attributbedingungen im Regelwerk sehr hoch ist, können sehr viele Vergleiche notwendig werden, welche sich negativ auf die Verarbeitungsgeschwindigkeit von Zugriffsanfragen auswirken. Hier sind andere Ansätze im Vorteil. Beispielsweise verwenden ACLs feste Tupel (etwa <*Subject, Action, Object*>). Diese können bei der Entscheidungsfindung sehr viel einfacher ausgewertet werden. Ein weiteres Problem ist die Komplexität bei der Auswertung von Zugriffsanfragen, die durch verschiedene Arten von Combining

Algorithm auftritt. In jedem Knoten eines XACML-Regelwerks kann ein unterschiedlicher Combining Algorithm verwendet werden, was die Berechnung einer Zugriffsentscheidung in einem Baum mit vielen Knoten erschwert.

Performanzoptimierungen für XACML-Regelwerke Es existieren eine Reihe von Ansätzen, die sich mit der Optimierung von XACML-Regelwerken beschäftigen. Viele dieser Ansätze haben gemeinsam, dass ein Transformationsprozess stattfindet. Dies bedeutet, dass ein existierendes Regelwerk in verschiedene Strukturen überführt wird, welche mit unterschiedlichen Strategien ausgewertet werden, wenn eine Zugriffsanfrage eintrifft. Der Vorteil dieser Verfahren ist die Verbesserung der Verarbeitungsgeschwindigkeit von Zugriffsanfragen. Der grundsätzliche Nachteil eines Transformationsprozesses ist die Notwendigkeit, den Prozess nach einer Änderung im Regelwerk erneut ausführen zu müssen. In Umgebungen in denen dynamisch Zugriffsberechtigungen vergeben oder entzogen werden, ist dies ein entscheidender Nachteil, weil das Ergebnis des Transformationsprozesses gegebenenfalls bereits wieder obsolet ist, wenn der Transformationsprozess abgeschlossen ist und eine weitere Änderung am Regelwerk notwendig ist. Ein Verfahren, welches auf einem Transformationsprozess beruht, benötigt daher dringend die Angabe, wie inkrementelle Änderungen am Regelwerk umgesetzt werden können.

In [94, 93] wird XEngine beschrieben. XEngine ist ein Ansatz, der Numericalization und Normalization verwendet, um die Performanz zu verbessern. Numericalization beschreibt einen Prozess, der jeden möglichen Attributwert zu einem Integerwert konvertiert. Integerwerte lassen sich sehr viel schneller vergleichen als Zeichenketten,

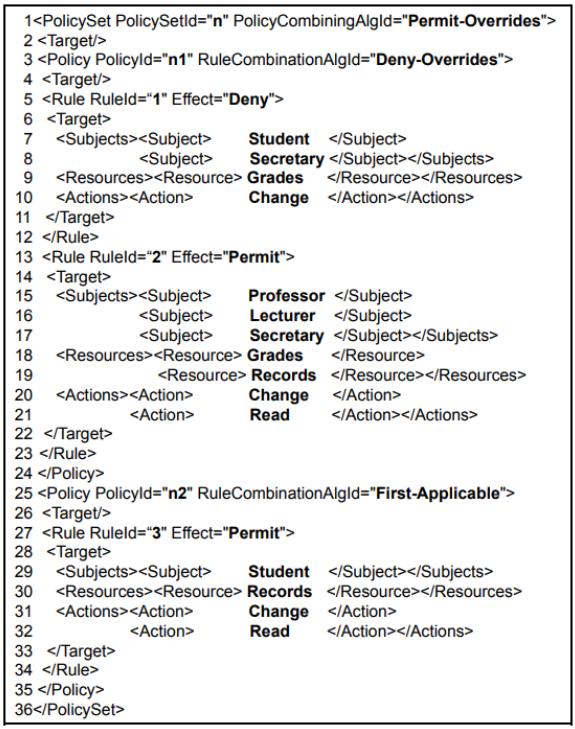

Abbildung 2.8: XACML-Regelwerk (Quelle: [94])

so dass Attributbedingungen erheblich schneller ausgewertet werden können. Kommt in einem Regelwerk beispielsweise ein Attribut mit

dem Namen *email* vor und wird dies in Attributbedingungen auf die Werte *marc@example.com* und *ulf@example.com* geprüft, so findet eine Übersetzung von der Zeichenkette *marc@example.com* auf den Integerwert *1* und von *ulf@example.com* auf *2* statt. So entsteht aus einem Vergleich von Zeichenketten ein Vergleich von Integerwerten. Mit dieser Methode lassen sich verbesserte Antwortzeiten erzielen. Allerdings führt die Übersetzung der Werte dazu, dass lediglich auf Gleichheit geprüft werden kann. So kann beispielsweise nicht mehr geprüft werden, ob eine Zeichenkette die Zeichenfolge *@example.com* enthält. Numericalization kann also nur dann eingesetzt werden, wenn der Wertebereich fest eingegrenzt werden kann. Abbildung 2.8 zeigt ein vereinfachtes Beispiel eines XACML-Regelwerks, welches verwendet wird, um Numericalization zu erläutern. Um die Lesbarkeit zu erhöhen wurden die Angaben zu Attributwerten gekürzt. So fehlen beispielweise die Angabe der Category und des Datentyps der Attribute.

Abbildung 2.9 zeigt das Ergebnis von Numericalization. Die im Regelwerk vorkommenden Werte werden auf Integerwerte übersetzt. Die Attributbedingungen der Regeln werden als Intervalle von Integerwerten ausgedrückt. So sagt beispielsweise R_1 aus, dass der Integerwert für das Subjekt 0 oder 1 betragen muss, der Integerwert für die Ressource 0 betragen muss und der Integerwert für die Aktion ebenfalls 0 betragen muss. Die Entscheidung ist in diesem Fall *deny*. Dies entspricht exakt der Formulierung von der XACML-Rule mit der RuleId *1* aus Abbildung 2.8. Der einzige Unterschied ist, dass Integerwerte verwendet werden anstelle von Zeichenketten. Analog zu R_1 werden auch die Rules R_2 und R_3 abgebildet. Zusätzlich wird exakt eine neue Rule R_{-1} eingeführt, welche den gesamten Wertebereich aller Attribute abdeckt und deren Ergebnis *not applicable (na)* ist.

Subject		Resource		Action	
Student:	0				
Secretary:	1	Grades:	0	Change:	0
Professor:	2	Records:	1	Read:	1
Lecturer:	3				

$$
\begin{aligned}
R_1 &: S \in [0,1] \wedge R \in [0,0] \wedge A \in [0,0] \rightarrow d \\
R_2 &: S \in [1,3] \wedge R \in [0,1] \wedge A \in [0,1] \rightarrow p \\
R_3 &: S \in [0,0] \wedge R \in [1,1] \wedge A \in [0,1] \rightarrow p \\
R_{-1} &: S \in [0,3] \wedge R \in [0,1] \wedge A \in [0,1] \rightarrow na
\end{aligned}
$$

Abbildung 2.9: Numericalization (Quelle: [94])

Das Resultat von Numericalization ist eine Menge von Regeln, welche Integerintervalle beschreiben. Aus dieser Darstellung geht noch nicht hervor, in welcher Beziehung die Regeln zueinander stehen. Das heißt, die Struktur des XACML-Regelwerks wird noch nicht abgebildet. Sind mehrere Regeln auf eine Zugriffsanfrage anwendbar, so kann noch nicht entschieden werden, welche Regel das Ergebnis bestimmt.

Die Struktur des Regelwerks wird erst durch den Einsatz von Normalization berücksichtigt. Normalization konvertiert ein hierarchisches Regelwerk in ein äquivalentes Regelwerk mit einer flachen Struktur. In dieser flachen Struktur kann immer nur genau eine Regel anwendbar sein. Das ermöglicht den Einsatz von *FirstApplicable* als Combining Algorithm, um die Verarbeitungsgeschwindigkeit zu erhöhen, indem nach dem Finden der ersten und einzig anwendbaren Regel

die Auswertung beendet werden kann. Um ein solches Regelwerk aufzustellen, werden in einem *policy decision diagram* (PDD) alle Attributwertkombinationen ermittelt, welche zu einer anwendbaren Regel führen. Abbildung 2.10 zeigt das PDD für das Regelwerk aus Abbildung 2.8.

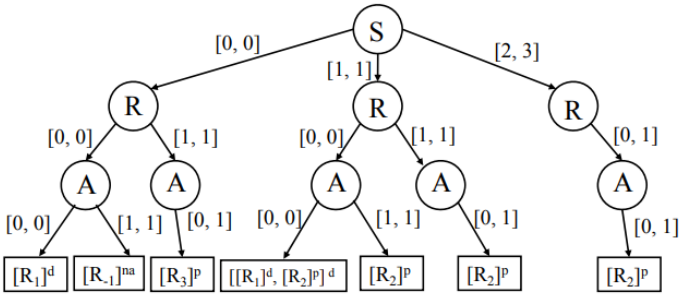

Abbildung 2.10: Policy Decision Diagram (Quelle: [94])

Im Wurzelknoten werden Integerwerte für Subjektbedingungen unterschieden. Es existieren drei Verzweigungen für die Intervalle $[0,0]$, $[1,1]$ und $[2,3]$. Dies entspricht Verzweigungen für die Attributwerte $[Student]$, $[Secretary]$ und $[Professor, Lecturer]$. Für jede dieser Verzweigungen werden die Ressourcenbedingungen unterschieden. In den ersten beiden Verzweigungen ist eine Unterscheidung für die Ressourcenwertintervalle $[0,0]$ und $[1,1]$ notwendig, während im dritten Zweig die Ressourcenwerte in einem Intervall abgebildet werden können: $[0,1]$. Analog wird für die Aktionswerte verfahren. In den Blättern des PDD werden schließlich die anwendbaren Regeln und die Zugriffsent-

scheidung gespeichert. Etwa wird im ersten Blatt gespeichert, dass die Regel R_1 mit dem Effekt *Deny* anwendbar ist. Im zweiten Blatt wird hinterlegt, dass keine Regel anwendbar ist. Im dritten Blatt wird gespeichert, dass die Regel R_3 mit dem Effekt *Permit* anwendbar ist. Im vierten Blatt wird gespeichert, dass die Regeln R_1 und R_2 anwendbar sind. Zusätzlich wird die Zugriffsentscheidung *Deny* gespeichert, da der Combining Algorithm der Policy, in der sich R_1 und R_2 befinden, *DenyOverrides* ist und somit R_1 die Zugriffsentscheidung bestimmt. Analog werden im fünften, sechsten und siebten Blatt gespeichert, dass Regel R_2 anwendbar ist und den Effekt *Permit* liefert.

Abbildung 2.11 zeigt das Ergebnis von Normalization. Das Resultat ist ein Regelwerk mit einer flachen Struktur, bei der genau eine Regel auf eine Zugriffsanfrage anwendbar sein kann, weshalb nach dem Finden der ersten anwendbaren Regel die Auswertung beendet werden und weshalb auch die Reihenfolge, in der die Regeln angeordnet sind, unbedeutend ist. Es lässt sich erkennen, dass die Anzahl an benötigten Regeln vervielfacht wird. Aus den Regeln R_1 und R_2 entstehen fünf neue Regeln $r_1, ..., r_5$. Die Anzahl der neuen Regeln hängt dabei von der Anzahl der möglichen Werte eines Attributs und der Struktur des Regelwerks ab. So wird die ursprüngliche Regel R_3 auf genau eine neue Regel r_6 übersetzt. Außerdem wird die in Abbildung 2.9 eingeführte Regel R_{-1} als neue Regel r_7 integriert. Diese findet überall dort Anwendung, wo eine Wertekombination nicht durch das ursprüngliche Regelwerk abgedeckt ist. Im Beispiel ist dies bei der Wertekombination $(S = 0, R = 0, A = 1)$ der Fall. Das ursprüngliche Regelwerk liefert für die Auswertung dieser Wertkombination das Ergebnis *not applicable (na)*.

Bei der Auswertung einer Zugriffsanfrage gegen die Menge der transformierten Regeln kann der PDD eingesetzt werden. Da die neuen

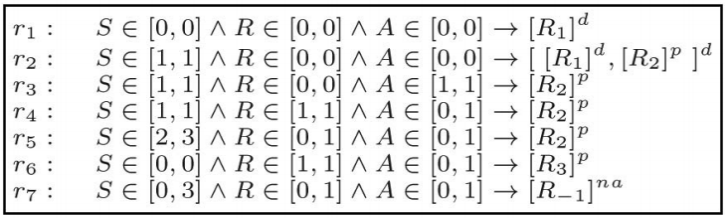

Abbildung 2.11: XEngine Regelwerk (Quelle: [94])

Regeln keine überlappenden Wertebereiche haben, müssen nicht alle Verzweigungen im PDD betrachtet werden. Nach dem Finden eines anwendbaren Zweigs in jeder Ebene des Baums, kann auf die Auswertung der übrigen Verzweigungen in der selben Ebene verzichtet werden, da jeweils nur ein Zweig anwendbar sein kann. Dieses Verfahren zeigt im Vergleich mit einer Open-Source-Implementierung von XACML namens SunPDP [134] deutlich verbesserte Auswertungszeiten.

Neben der Auswertung über den PDD beschreibt die Arbeit ein weiteres Verfahren, um aus der Menge der transformierten Regeln und einer Zugriffsanfrage eine Zugriffsentscheidung zu berechnen. Das zweite Verfahren wird *Forwarding Tables* genannt. Abbildung 2.12 zeigt die Idee von Forwarding Tables. Eingangsparameter ist eine durch Numericalization bearbeitete Zugriffsanfrage, beispielsweise $(1,1,0)$. In einem ersten Schritt wird der erste Wert der Zugriffsanfrage ausgewertet. Der Wert 1 wird über die erste Tabelle auf den Wert 1 übersetzt. Dieser übersetzte Wert dient zusammen mit dem zweiten Wert der Zugriffsanfrage als Eingangsparameter für die zweite Tabelle.

Der Tabelleneintrag liefert den Wert 3, welcher zusammen mit dem dritten Wert der Zugriffsanfrage als Eingangsparameter für die dritte Tabelle dient. So lässt sich berechnen, dass für die Zugriffsanfrage $(1,1,0)$ die Rule R_2 mit dem Effect *permit* das Ergebnis bestimmt.

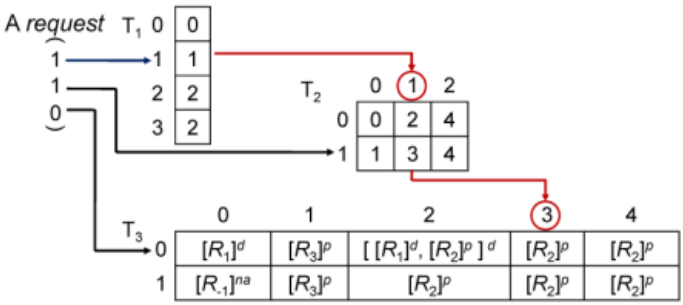

Abbildung 2.12: Forwarding Tables (Quelle: [94])

Der Aufbau der Forwarding Tables geschieht aus dem PDD heraus. Die erste Tabelle ist eindimensional, da hier der Wurzelknoten des PDD betrachtet wird. Für jeden möglichen Attributwert $(0-3)$, wird ein Ausgangswert festgelegt. Der Wurzelknoten des PDD hat drei Verzweigungen, denen die Werte $0,1,2$ zugeordnet werden. Die zweite Tabelle ist zweidimensional. Jede Spalte der Tabelle entspricht einer Verzweigung des Wurzelknotens des PDD. Der erste Knoten in der zweiten Ebene des PDD hat wiederum zwei Verzweigungen, welche mit den Werten 0 und 1 abgebildet werden. Der zweite Knoten der zweiten Ebene des PDD hat ebenfalls zwei Verzweigungen, welche

mit den Werten 2 und 3 abgebildet werden. Der dritte Knoten der zweiten Ebene des PDD hat nur eine Verzweigung, welche mit dem Wert 4 abgebildet wird. Die Werte $0-4$ repräsentieren also die Verzweigungen unterhalb der zweiten Ebene des PDD. Analog bilden die Spalten der dritten Tabelle die Verzweigungen in der dritten Ebene des PDD ab. Der erste Knoten in der dritten Ebene hat zwei Verzweigungen zu den Rules R_1 und R_{-1}. Der zweite Knoten der dritten Ebene eine Verzweigung zu R_3. Analog werden für die übrigen Tabelleneinträge aus dem PDD ermittelt.

Vereinfacht lässt sich sagen, dass XEngine die möglichen Attributwerte auf feste Intervalle begrenzt, die Ergebnisse von allen potentiell möglichen Zugriffsanfragen in einem PDD vorberechnet und in Intervallen von Integerwerten speichert. Beim Forwarding Tables-Ansatz werden zusätzlich die Intervalle aufgelöst und in Tabellenform gespeichert. Beim Eintreffen einer Zugriffsanfrage können diese Intervalle beziehungsweise diese Tabellen sehr schnell ausgewertet werden. Bei beiden Verfahren wird die Auswertungszeit entkoppelt von der Komplexität der Regelwerks. Für jedes Attribut ist eine Auswertungsoperation notwendig, um im PDD die zutreffende Verzweigung beziehungsweise in den Forwarding Tables den zutreffenden Eintrag zu finden. Damit hängt die Komplexität der Auswertung nicht mehr von der Anzahl der Rules im Regelwerk ab, sondern von der Anzahl der verschiedenen Attribute. Diese Verbesserung der Auswertungszeiten ist möglich, solange ein begrenztes Intervall von Werten existiert. Existieren beispielweise sehr viele unterschiedliche Subjekte und sehr viele unterschiedliche Ressourcen, so kann das Aufstellen des PDD sehr aufwendig werden. Dies ist insbesondere dann kritisch, wenn häufige Änderungen am Regelwerk auftreten. Hier bedarf es einem Ansatz, der die inkrementelle Erweiterung des PDD beschreibt.

Ein weiterer Ansatz zur effizienten Auswertung von XACML-Regelwerken wird in [121] beschrieben. Die Autoren beschreiben, dass die Auswertungszeiten von XEngine unabhängig sind von der Komplexität des Regelwerks und stattdessen von der Anzahl unterschiedlicher Attributwerte abhängen. Darüber hinaus nennen die Autoren extremen Speicherverbrauch und die Beschränkung auf Prüfung auf Gleichheit als wesentliche Nachteile von XEngine und beschreiben ihre Arbeit als Weiterentwicklung von XEngine. In ihrer Arbeit zeigen die Autoren einen graphbasierten Ansatz, welcher zwei unterschiedliche Bäume verwendet, um die Performanz der Auswertung einer XACML-Anfrage zu verbessern.

Der erste Baum, der sogenannte Matching-Tree, identifiziert die anwendbaren XACML-Rules und besteht aus einer Knotenebene pro Attribut. Für das zuvor verwendete Bankbeispiel bedeutet dies etwa, dass der Matching-Tree in der ersten Verzweigungsebene die Kontonummer, in der zweiten Verzweigungsebene die Kennung des anfragenden Subjekts und in der dritten Ebene die Zugriffsmethode als Attributbedingungen auswertet. Die Heterogenität von Entscheidungspfaden wird in der Arbeit nicht betrachtet. Dies bedeutet, dass davon ausgegangen wird, dass in jeder Verzweigungsebene eine Attributbedingung existiert, also bei jeder Anfrage alle Attribute ausgewertet werden. So ist es beispielsweise nicht möglich, dass im Regelwerk gleichzeitig solche Entscheidungspfade existieren, welche die Nutzerkennung auswerten und solche, die die Nutzerkennung nicht auswerten.

Der Matching Tree ist ein Entscheidungsbaum und funkioniert ähnlich dem PDD von XEngine. Das bedeutet, dass der Baum nach Attributen strukturiert wird, um so die anwendbaren Regeln zu identifizieren. Anders jedoch als bei XEngine wertet der Matching Tree zur Laufzeit Vergleichsfunktionen aus anstatt einen einfachen Entschei-

dungsbaum darzustellen. Dies bietet den Vorteil, dass auch komplexe Funktionen berechnet werden können und so auch andere Vergleichsoperationen als die Prüfung auf Gleichheit möglich sind. So können etwa reguläre Ausdrücke genutzt werden.

Der zweite Baum, der Combining Tree, enthält die Ausgangsstruktur und kombiniert die Effekte der anwendbaren Rules zu einem finalen Ergebnis. Der Combining Tree ist identisch mit dem XACML-Regelwerk mit der Ausnahme, dass keine Targets enthalten sind. Der Baum beinhaltet also die Combining Algorithm und die Effects des Regelwerks, nicht jedoch die Targets mit Attributbedingungen. Diese sind in den Matching Tree ausgelagert. Da der Ansatz die ursprüngliche Struktur des Regelwerks bewahrt, können auch XACML-Obligations unterstützt werden. Dies ist ein weiterer Unterschied zum XEngine-Ansatz, welcher lediglich die finalen Zugriffsentscheidungen vorberechnet, ohne die dabei geltenden Obligations zu berücksichtigen.

Der Ansatz stellt Transformationsalgorithmen vor, die verwendet werden können, um aus einem XACML-Regelwerk sowohl Matching-Tree als auch Combining-Tree zu berechnen. Das Resultat des Transformationsprozesses sind verbesserte Antwortzeiten für Zugriffsanfragen im Vergleich mit einer Referenzimplementierung von XACML. Da die Tiefe des ersten Baums der Anzahl der unterschiedlichen Attribute entspricht, ist dieser Ansatz jedoch nur bedingt geeignet in Umgebungen, in denen eine große Vielzahl unterschiedlicher Attribute existiert. Da der Ansatz zudem eine Transformation des Regelwerks in zwei Bäume erfordert, muss bei jeder Änderung im Regelwerk eine erneute Transformation stattfinden. Wie bereits zuvor erwähnt, ist dies in Anwendungen mit häufigen Änderungen des Regelwerks kritisch. Dies gilt insbesondere dann, wenn auch inkrementelle Änderungen einen aufwendigen Transformationsprozess erfordern.

In [103] wird ein weiterer Ansatz vorgestellt, der auf Entscheidungsbäumen beruht. Die Autoren beschreiben, dass existierende Ansätze zur Verbesserung der Antwortzeiten von XACML nicht die vollständige Semantik des Standards berücksichtigen, wie etwa die Fehlerbehandlung in Falle von Indeterminate-Entscheidungen oder die Verwendung von Obligations und Advices. Die Autoren analysieren XACML im Detail und vergleichen ihren Ansatz in Messungen mit der Open-Source-Implementierung von XACML namens SunPDP [134]. Auch dieser Ansatz liefert deutlich verbesserte Antwortzeiten gegenüber der Open-Source-Implementierung. Da der Ansatz jedoch ebenfalls auf einem Transformationsprozess basiert, ist auch hier die Eignung in Umgebung mit sich dynamisch ändernden Zugriffbedingungen kritisch.

In [97] werden zwei alternative Methoden angewendet, um die Verarbeitungsgeschwindigkeit zu verbessern. Die erste Methode ist das Clustering eines XACML-Regelwerks. Dabei werden Cluster erzeugt, die das Regelwerk in Abhängigkeit des anfragenden Subject aufteilen. Existieren beispielsweise fünf verschiedene Attributbedingungen für das Subject, so wird die Policy auch in fünf Cluster aufgeteilt. Bei der Auswertung einer Zugriffsanfrage wird dann ausschließlich der entsprechende Cluster ausgewertet und so nicht anwendbare Teile des Regelwerks gemieden. Die zweite Methode zur Verbesserung der Verarbeitungsgeschwindigkeit ist die Umsortierung von Policy Sets, Policies und Rules. Diese Umsortierung basiert auf Wahrscheinlichkeiten. Dazu werden Statistiken aus früheren Zugriffsanfragen erstellt. Elemente, die mit einer hohen Wahrscheinlichkeit anwendbar sind, werden zuerst ausgewertet.

In [133] wird das Regelwerk mittels *Vertical Reordering* umgeschrieben. *Vertical Reordering* eliminiert Redundanzen durch den Ein-

satz von boolschen Ausdrücken innerhalb von XACML-Targets. Dieser Ansatz reduziert die durchschnittliche Anzahl an Target-Vergleichen, erhöht aber auf der anderen Seite die Target-Komplexität.

Ein in [100] beschriebener Ansatz sortiert Policies anhand einer Kostenfunktion. Es wird davon ausgegangen, dass die Kombination von Attributbedingungen aus Policy Sets, Policies und Rules immer aus je einer Subject-, Action- und Resource-Entität besteht. Der Ansatz definiert eine Zugriffsregel als Quadruple bestehend aus diesen drei Entitäten und einem Effect. Die Optimierung der Kosten basiert auf dieser Annahme.

In [14] wird ein Ansatz beschrieben, welcher sich mit Zugriffskontrollbäumen im Cloud Computing beschäftigt. Die Autoren nennen die Fähigkeiten mit großen Mengen von Benutzern und Ressourcen umgehen und gleichzeitig große Mengen von Zugriffsanfragen schnell verarbeiten zu können, eine essenzielle Anforderung für Zugriffskontrollsysteme in Cloud Computing. Der Ansatz geht von festen Tripeln der Form <*Subject, Action, Resource*> aus und beschreibt instanzbasierte Zugriffskontrollbäume.

Ein instanzbasiertes Verfahren nutzt anstelle einer großen Komposition von Zugriffsberechtigungen kleine Teilmengen des Regelwerks, welche bestimmten Subjekten, Ressourcen oder Umgebungen zugeordnet werden. Zur Laufzeit werden dann die jeweiligen Teilmengen ausgewertet. Hierdurch wird der Anteil des gesamten Regelwerks, der beim Eintreffen einer Zugriffsanfrage ausgewertet werden muss, erheblich reduziert. In [14] wird ein instanzbasiertes System mit XACML umgesetzt. Dazu gehen die Autoren von einer festen Struktur im Regelwerk aus. Auf der ersten Ebene sind lediglich Subjektbedingungen erlaubt, in der zweiten Ebene Aktionsbedingungen, in der dritten Ebene Bedingungen zum Ressourcentyp und auf der vierten Ebene

Bedingungen zur Ressource. Die Bedingungen der ersten vier Ebenen sind jeweils disjunkt, so dass jeweils nur ein Zweig des Regelwerks ausgewertet werden muss.

Allgemeiner ließe sich ein instanzbasiertes System umsetzen, indem das gesamte Regelwerk in viele einzelne Regelwerke aufgeteilt wird und das jeweilige Regelwerk zur Laufzeit instanziiert wird. Etwa könnten die Regelwerke Ressourcen zugeordnet werden und beim Zugriff auf eine Ressource in das Zugriffskontrollsystem geladen werden. Diese Weiterentwicklung beschreiben die Autoren in [137]. Diese Arbeit nennt das Abbilden von Zugriffsberechtigungen in Bäumen Caching und beschäftigt sich mit der Optimierung von Zugriffskontrolle, um die Entscheidungsfindung beim Einsatz von XACML im Cloud Computing zu beschleunigen. Die Autoren gehen von einer ähnlichen Struktur für Zugriffsberechtigungen aus. Paare aus Subjekt und Aktion werden einer Liste von Ressourcen (Data Objects) zugeordnet. Subjekte, Aktionen und Ressourcen werden in einem sogenannten *Permit Tree* organisiert. Das bedeutet, alle erlaubten Zugriffsentscheidungen werden durch das System abgebildet. Negative Zugriffsentscheidungen werden nicht modelliert. In der obersten Ebene des *Permit Tree* werden Subjekte abgebildet, in der darunterliegenden Ebene die Aktionen und wiederum darunter liegend die Listen von Ressourcen.

Der Baum wird in zwei verschiendenen Varianten implementiert und beide Varianten werden verglichen. Die erste Variante sichert die Zugriffsberechtigungen in einer relationalen Datenbank, welche in den Hauptspeicher geladen wird. Die zweite und effizientere Variante ist die Sicherung der Zugriffsberechtigung in einer Hash Table. Beide Varianten sind möglich, da die Autoren von einer festen Struktur der Zugriffsberechtigung ausgehen, so dass eindeutige Einträge in Datenbank und Hash Table leicht zu berechnen sind.

Weitere Themen der Forschung zu XACML Ein Ansatz zur Formalisierung von XACML wird in [98] beschrieben. Der Ansatz beschreibt eine alternative Variante zur Formulierung von Regelwerken, welche es Entwicklern vereinfachen soll, XACML-Regelwerke zu erstellen. In diesem Ansatz wird lediglich eine alternative und vereinfachte Syntax beschrieben, die ähnlich wie die bereits zuvor genannte ALFA-Notation strukturiert ist. Verbesserungen der Antwortzeiten sind nicht Gegenstand dieser Arbeit.

In [102] wird ein Ansatz vorgestellt, der die Reduktion von Zugriffsentscheidungen für kompositionelle Regelwerke behandelt. Der Ansatz analysiert im Detail die Combining Algorithm und bietet eine Lösung, wie unterschiedliche Zugriffsentscheidungen sicher zu einer finalen Entscheidung zusammengefasst werden können. Die Reduktion von Zugriffsentscheidungen mit den in XACML beschriebenen Combining Algorithm ist nicht sicher, da bei der Reduktion von *Indeterminate (Permit)*, *Indeterminate (Deny)* und *Indeterminate (Permit or Deny)* zu *Indeterminate* Präzision verloren geht.

Kompositionelle Verfahren werden ebenfalls in [29] behandelt. Die Arbeit beschreibt einen Ansatz, welcher die Definition von Attributbedingungen und die Zusammensetzung des Regelwerks in separate Teilgebiete aufspaltet. Dazu führt der Ansatz zwei Sprachen ein: die Policy Target Language (PTL) und die Policy Composition Language (PCL). Die PTL dient der Formulierung von Attributbedingungen und somit der Definition, wann eine Policy ausgewertet und wann eine Zugriffsentscheidung getroffen wird. Die PCL dient hingegen der Komposition der Elemente der PTL. Durch die PCL wird die Formulierung von komplexen Regelwerken möglich. Die Trennung von Attributbedingungen und Aufbau des Regelwerks ist ein Optimierungsansatz, welchen einige der zuvor beschriebenen Arbeiten ebenfalls vornehmen.

In dieser Arbeit wird diese Trennung als Ausgangsbasis verwendet. Wird eine solche Trennung vom ersten Moment berücksichtigt, können Attributbedingungen einfacher wiederverwendet werden, so dass die Verwaltung der Regelwerke sich vereinfacht. Die Herausforderungen, die beim Ändern des Aufbaus eines Regelwerks entstehen, werden auch in dieser Arbeit nicht berücksichtigt.

2.3 Representational State Transfer

Representational State Transfer (REST) beschreibt einen Architekturstil für verteilte Systeme. REST ist also weder ein Standard noch eine Technologie. Der Architekturstil erfreut sich wachsender Beliebtheit. Der Grund hierfür ist die Einfachheit von REST-Schnittstellen sowie die Tatsache, dass die benötigte Infrastruktur bereits weit verbreitet ist. Ziel beim Einsatz von REST ist es, ein gleichzeitig verteiltes, skalierendes und hoch performantes System zu ermöglichen. Hierfür werden verschiedene Rahmenbedingungen vorgegeben, deren Einhaltung die Existenz eines solchen Systems ermöglicht. Der Architekturstil wurde von Fielding im Rahmen seiner Dissertation zum ersten Mal beschrieben [44]. Kerngedanke des Architekturstils ist eine Ressourcenorientierung gepaart mit einer einheitlichen Schnittstelle und zustandsloser Kommunikation mit beliebigen Ressourcen. Ziel von Fieldings Arbeit war es, die Merkmale eines hoch performanten, verteilten Systems zu identifizieren. Zu diesem Zweck hat er das größte dieser Systeme untersucht: das Internet. Der Architekturstil wird deshalb oft auch als die Architektur des Internets beschrieben. Anwendungen, die dem Architekturstil folgen, profitieren von Skalierbarkeit, Aggregierbarkeit, Nutzbarkeit und Verfügbarkeit [4, 107].

Richardson Maturity Model Aufbauend auf Fieldings Arbeit entwickelte Richardson das sogenannte Richardson Maturity Model [117, 147]. Das Modell definiert vier Level, die es erlauben, den Unterstützungsgrad eines Systems hinsichtlich der wichtigsten Merkmale von REST zu ermitteln. Ausgehend von Level 0 (keine Unterstützung) folgen drei weitere Level bis hin zur vollen Unterstützung der entscheidenden Merkmale von REST. Abbildung 2.13 zeigt das Richardson Maturity Model. Die Level bauen aufeinander auf. Das heißt, ein übergeordneter Level setzt die Eigenschaften eines untergeordneten Levels voraus. Eine Anwendung, welche Level 3 des Richardson Maturity Models unterstützt, wird als *RESTful* bezeichnet.

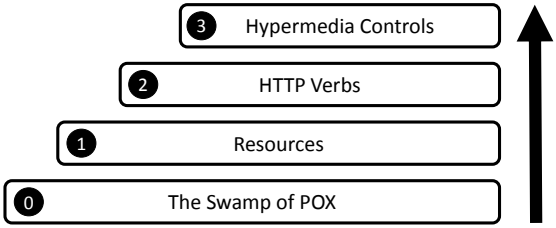

Abbildung 2.13: Richardson Maturity Model

Das Richardson Maturity Model bezieht sich explizit auf den Einsatz von HTTP. Dies ist bei der Umsetzung von RESTful Services keine Voraussetzung, auch wenn HTTP ein sehr geläufiges Protokoll

hierfür ist. Auch andere Protokolle können für die Umsetzung eingesetzt werden. Da sich das Richardson Maturity Model jedoch explizit auf HTTP bezieht soll ein einfaches Beispiel dem Verständnis dienen. Auflistung 2.6 zeigt eine einfache Anfrage. Erstes Element der Anfrage ist ein HTTP Verb, welches die Anfragemthode beschreibt. Es folgt der Pfad der angefragten Ressource und die Angabe der Protokollversion. Danach können eine Reihe von optionalen Headern folgen. Typischerweise ist mindestens der *Host*-Header vorhanden, welcher angibt wo sich die angefragte Ressource befindet.

```
1 GET /path HTTP/1.1
2 Host: www.example.org
```

Auflistung 2.6: HTTP-Anfrage

Auflistung 2.7 zeigt eine mögliche Antwort. Erstes Element ist die Protokollversion. Danach folgt ein Status als Integerwert und in Textform. Auch die Antwort kann wieder optionale Header beinhalten. Schließlich folgen die Daten der angefragten Ressource.

```
1 HTTP/1.1 200 OK
2 Content-Type: text/html
3
4 ... data ...
```

Auflistung 2.7: HTTP-Antwort

Level 0 Es handelt sich um ein verteiltes System, welches entfernte Prozeduraufrufe beinhaltet. Diese Prozeduren kann man sich als wiederverwendbare Methoden vorstellen, welche bestimmte Aufgaben übernehmen. Fowler spricht an dieser Stelle vom Sumpf von altherge-

brachtem XML (Swamp of Plain Old XML), welcher oft eingesetzt wird, um solche Prozeduraufrufe durchzuführen [45].

Auflistung 2.8 zeigt ein Beispiel für einen solchen Aufruf. An einen Konto Service kann eine Anfrage zum Kontostand gesendet werden.

```
1 POST /bankService HTTP/1.1
2
3 <balanceRequest>
4   <account>123</account>
5 </balanceRequest>
```

Auflistung 2.8: Level 0 - Kontostand abfragen

Auflistung 2.9 zeigt die Antwort auf eine solche Anfrage. Der Kontostand wird im Body der Antwort zurückgesendet

```
1 HTTP/1.1 200 OK
2
3 <balance>100</balance>
```

Auflistung 2.9: Level 0 - Kontostand empfangen

Auflistung 2.10 zeigt eine zweite Anfrage an den gleichen Konto Service. Hier wird eine Überweisungsübersicht abgefragt. Der Unterschied zur ersten Anfrage wird lediglich über die im Body enthaltenen Daten festgelegt. Die Adresse des Konto Service bleibt unverändert.

```
1  POST /bankService HTTP/1.1
2
3  <transferOverviewRequest>
4    <account>123</account>
5    <start>01.06.2018</start>
6    <end>30.06.2018</end>
7  </transferOverviewRequest>
```

Auflistung 2.10: Level 0 - Überweisungsübersicht abfragen

Auflistung 2.11 zeigt schließlich die Antwort auf diese Anfrage. Es wird eine Liste von Überweisungen zurückgeliefert.

```
1  HTTP/1.1 200 OK
2
3  <transfers>
4    <transfer id="1" amount="50" recipient="124"
5                                    sender="123" />
6    <transfer id="2" amount="20" recipient="125"
7                                    sender="123" />
8    <transfer id="3" amount="10" recipient="123"
9                                    sender="126" />
10 </transfers>
```

Auflistung 2.11: Level 0 - Überweisungsübersicht empfangen

Level 1 Ressourcenorientierung ist der erste wesentliche Schritt in Richtung RESTful Service [118]. Anstelle des Aufrufs von wiederverwendbaren Diensten oder Methoden, werden Ressourcen direkt angesprochen. Jede Ressource ist dabei eindeutig adressierbar über eine URI. Die Zugriffsmethode ist entweder als Teil der Adresse der

Ressource gekapselt oder wird durch die im Body enthaltenen Daten festgelegt.

Auflistung 2.12 zeigt ein Beispiel für einen solchen Aufruf. Hier wird nicht mehr ein allgemeiner Bankdienst angesprochen, sondern es wird eine einzelne Ressource - das Bankkonto - direkt angesprochen.

```
1 POST /accounts/123 HTTP/1.1
2
3 <balanceRequest />
```

Auflistung 2.12: Level 1 - Ressourcenorientierung

Auflistung 2.13 zeigt die zweite Anfrage zum Abfragen der Überweisungsübersicht. Auch hier wird eine individuelle Ressource angesprochen. Diesmal ist die angefragte Ressource eine Liste von Überweisungen.

```
1 POST /accounts/123/transfers HTTP/1.1
2
3 <transferOverviewRequest>
4   <start>01.06.2018</start>
5   <end>30.06.2018</end>
6 </transferOverviewRequest>
```

Auflistung 2.13: Level 1 - Listenressource

Die Antworten auf die beiden Anfragen sind gegenüber Level 0 des Richardson Maturity Models unverändert. Jedoch sind nun alle Überweisungen auch individuell adressierbar. So könnten beispielsweise weitere Details der ersten Überweisung aus Auflistung 2.11 über den Pfad */accounts/123/transfers/1* abgefragt werden.

Level 2 Auf dem nächsten Level werden die Zugriffsmethoden nicht mehr in der Adresse oder den Daten der Anfrage gekapselt, sondern über das verwendete Protokoll bestimmt. Das Modell bezieht sich an dieser Stelle auf die HTTP Verben wie *GET*, *POST*, *PUT* oder *DELETE* [71]. Allgemeiner lässt sich formulieren, dass die Methoden einer einheitlichen Schnittstelle verwendet werden.

Auflistung 2.14 zeigt ein Beispiel für eine Anfrage, die konform mit Level 2 des Modells ist. In den vorherigen Anfragen wurde die POST-Methode verwendet, da hier im Body der Anfrage Daten übertragen werden können. Auf Level 2 wird die Zugriffsmethode über das Protokoll bestimmt. Da es sich um einen lesenden Zugriff handelt, ist eine GET-Anfrage angemessen.

```
1 GET /accounts/123/balance HTTP/1.1
```

Auflistung 2.14: Level 2 - Zugriffsmethode

Auflistung 2.15 zeigt die zweite Anfrage zum Abfragen der Überweisungsübersicht. Auch hier wird die GET-Methode verwendet. Die Daten zum Filtern des Zeitraums werden als Parameter der Ressourcenadresse übermittelt.

```
1 GET /accounts/123/transfers?start=01062018&end=30062018
2 HTTP/1.1
```

Auflistung 2.15: Level 2 - GET-Anfrage

Auflistung 2.16 macht den Unterschied bei der Verwendung der Verben des Protokolls noch deutlicher. Eine POST-Anfrage dient zum Anlegen einer Ressource. Mit dieser Anfrage wird demnach eine neue Überweisung in Auftrag gegeben.

```
1 POST /accounts/123/transfers HTTP/1.1
2
3 <transfer>
4   <transfer amount="50" recipient="124" />
5 </transfer>
```

Auflistung 2.16: Level 2 - POST-Anfrage

Level 3 Hypermedia Controls beschreiben den Einsatz von Hypermedia As The Engine Of Application State (HATEOAS) und ist im Vergleich mit den unteren Leveln des Modells eine schwerer verständliche Richtlinie. Hier geht es darum, die Verwaltung des Anwendungszustands vom Server zu lösen und sie dem Client zu übertragen. Client und Server können so entkoppelt werden und die Skalierbarkeit des Systems wird verbessert, indem Servern die Aufgabe der Verwaltung von Zuständen abgenommen wird. Dazu werden in den Antworten auf Anfragen eines Client mögliche nächste Zustände als Verweise mitgesendet.

Auflistung 2.17 zeigt die Antwort auf die Abfrage zum Kontostand mit Unterstützung von HATEOAS. Die Antwort enthält nun zusätzlich einen Verweis auf die Überweisungsübersicht.

```
1 HTTP/1.1 200 OK
2
3 <p>Balance: 100</p>
4 <a href="/accounts/123/transfers">Overview</a>
```

Auflistung 2.17: Level 3

2.3.1 Verteiltes System

Level 0 des Richardson Maturity Models kann mit einem einfachen verteilten System gleichgesetzt werden. Das System muss keinen weiteren Vorgaben des REST-Architekturstils folgen. Vielmehr kann ein eigener Mechanismus zum Aufruf entfernter Prozeduren angewendet werden. Ein Beispiel für ein solches System ist der Transport von Serviceaufrufen über das HTTP-Protokoll. Hierbei wird HTTP nur als Tunnel zum Transport verwendet. Beispielsweise ist das Simple Object Access Protocol (SOAP [149]) ein Protokoll, welches eine solche Tunnelung durchführt. SOAP dient dazu entfernte Methoden aufzurufen und erlaubt es dem Entwickler eines verteilten Systems die entfernten Methodenaufrufe in XML zu codieren. Diese Daten werden zwischen zwei Komponenten des verteilten Systems über eine HTTP POST-Anfrage verschickt. HTTP dient dabei lediglich dem Transport der Daten.

2.3.2 Ressourcenorientierung

Ressourcenorientierung ist die Grundlage von RESTful Services. Die Idee dabei ist, dass jede Ressource innerhalb eines verteilten Systems eindeutig adressierbar ist. Ressourcenorientierung setzt also nicht auf die Wiederverwendung verschiedenartiger Dienste zur Erledigung bestimmter Aufgaben, sondern spricht einzelne Ressourcen individuell an. Eine Ressource kann dabei jede Art von Entität innerhalb der verteilten Anwendung sein. Voraussetzung für die Umsetzung einer ressourcenorientierten Architektur ist der Einsatz eines einheitlichen Adressierungsschemas. Hierfür haben sich Uniform Resource Identifier (URI) als Standard etabliert.

Ressourcen sind der Gegenstand der Kommunikation in RESTful Services. Das bedeutet, ein Client fragt bei einem Server eine Ressource an und bekommt im Erfolgsfall eine Repräsentation der Ressource zurück geliefert. Ein Beispiel für eine Ressource ist etwa eine Person. Jede Person kann eindeutig identifiziert werden und eine Repräsentation der Person ist beispielsweise ihr Ausweis.

Ressourcendesign Wichtigste Grundlage in einem ressourcenorientierten System ist die Adressierbarkeit individueller Ressourcen. Jede Ressource muss eindeutig identifizierbar und adressierbar sein, wofür URIs verwendet werden. Eine URI verweist auf genau eine Ressource. Umgekehrt kann jedoch durch mehrere URIs auf ein und dieselbe Ressource verwiesen werden. Das Ressourcendesign (und damit auch das URI-Design) innerhalb einer Anwendung muss sorgfältig durchgeführt werden und der Anwendung angemessen sein. Eine URI sollte idealerweise aus einem beschreibenden Nomen bestehen und nicht aus Verben. Die URI dient der Identifikation der Ressource und nicht der Identifikation der gewünschten Zugriffsmethode. Eine passende URI für eine Konto-Ressource ist beispielsweise *http://www.example.org/account*. Hierbei wird durch das Nomen in der URI deutlich, dass sich hinter der URI ein Konto verbirgt.

Eine URI kann ein Uniform Resource Locator (URL), ein Uniform Resource Name (URN) oder beides sein. Eine URI besteht aus einem *scheme*, einer *authority* und optional aus einem *path*, einem *query string* und einem *fragment* [1]. Ein Beispiel für eine URI ist *http://example.org/resources?id=1#part1* mit *http* als scheme, *example.org* als authority, */resources* als path, *id=1* als query string und *part1* als fragment.

- **scheme:** ein Schema ist eine Referenz, die auf eine Spezifikation zur Identifikationszuweisung innerhalb des Schemas hinweist. Bekannte Beispiele für ein scheme sind *http*, *https* oder *ftp* welche bei URLs häufig verwendet werden.
- **authority:** eine Authority ist ein registrierter Name oder eine Serveradresse, die als Delegierter für die Identifikation der Ressource dient. Die Authority beschreibt also den Host, der die Ressource verwaltet. In einer HTTP-Anfrage steht die Authority üblicherweise in *Host*-Header.
- **path:** der Pfad ist ein hierarchisch organisiertes Element, welches der Identifikation der Ressource bei der Authority dient.
- **query:** ein Query-Parameter dient dem Filtern bestimmter Informationen einer Ressource. Etwa können Listenressourcen mit einem Query-Parameter gefiltert werden.
- **fragment:** ein Fragment dient dem direkten Zugriff auf einem bestimmten Teil einer Ressource.

Listenressourcen Häufig verwalten Anwendungen verschiedene Ressourcen des gleichen Typs. Es ist leicht vorstellbar, dass eine Bankanwendung nicht nur ein Konto verwaltet sondern mehrere. In einem solchen Fall können Listenressourcen eingesetzt werden. Das Abfragen einer Listenressource liefert eine Liste von Ressourcen zurück. Eine Liste von Ressourcen ist also selbst eine Ressource. Beispielsweise sollte das Abfragen der Ressource *http://www.example.org/accounts* eine Liste von Konten zurückliefern. Einzelne Konten können dann über spezifischere Pfade erfragt werden. So ist es denkbar, dass ein

bestimmtes Konto eine eindeutige ID mit dem Wert 123 hat und über die URI *http://www.example.org/accounts/123* abgefragt werden kann.

Filter Filter (also Query-Parameter) dienen dazu bestimmte Informationen einer Ressource abzufragen. Ein Filter besteht aus einem Schlüssel und einem Wert. Beides wird innerhalb der URI nach dem Pfad der Ressource angehangen. Als Trennzeichen dient ein Fragezeichen (*?*). Häufig kommen Filter im Zusammenspiel mit Listenressourcen zum Einsatz. Anstelle einer vollständigen Liste aller Ressourcen, können so Untermengen abgefragt werden. Beispielsweise kann der Filter *lastname=hueffmeyer* dazu verwendet werden, alle Konten von Kontoinhabern mit dem Nachnamen *Hueffmeyer* zu filtern: *http://www.example.org/accounts?lastname=hueffmeyer*. Es ist möglich, eine beliebige Anzahl von Filtern in einer URI anzugeben. Zwischen den Filtern wird ein kaufmännisches Und (*&*) als Trennzeichen eingesetzt.

Ressourcenschablonen Bei der Beschreibung von Ressourcen können URI Templates [2] verwendet werden, um Ressourcen zu gruppieren. URI Templates sind Schablonen, die eine Menge von Ressourcen beschreiben. Beispielsweise ist es denkbar, dass ein Konto eine untergeordnete Ressource hat, welche Überweisungsinformationen zum Konto bereithält. Das Konto mit der Kennung *123*, könnte als Ressource mit der URI *http://www.example.org/accounts/123/transfers* modelliert werden. Um nun die allgemeine Adresse der Überweisungsübersicht zu einem Konto anzugeben, können URI Templates verwendet werden. Anstelle einer konkreten Kennung eines Konto wird in einem Template in geschwungenen Klammern beschrieben, welche Informationen in

der Schablone ergänzt werden müssen, damit eine konkrete URI entsteht. Im Beispiel der Konto-Ressource könnte das folgende Template verwendet werden: *http://www.example.org/accounts/{id}/transfers*.

Repräsentationen Ein weiteres Konzept von REST ist die Unterscheidung zwischen Ressourcen und deren Repräsentationen. Ressourcen können als Domänenobjekte verstanden werden, welche vom Client angefragt werden. Ein Client wird jedoch nur eine Abbildung - eine Repräsentation - der Ressource geliefert bekommen anstatt der Ressource selbst.

Fragt ein Client eine Ressource an, so antwortet der Server im Erfolgsfall mit einer Repräsentation der Ressource. Eine Repräsentation einer Person ist zum Beispiel eine elektronische Visitenkarte [77] oder ein Bild. Der Client hat dabei (in Abhängigkeit vom Protokoll) die Möglichkeit bevorzugte Repräsentationen der Ressource anzugeben. Im Falle von HTTP kann der Client beispielsweise den *Accept*-Header in seiner Anfrage an den Server mitsenden. In diesem Header hat der Client die Möglichkeit bevorzugte Datenformate wie *text/html* oder *application/xml;q=0.9* anzugeben. Der Wert 0.9 zeigt dabei auf, mit welcher Gewichtung sich der Client dieses Datenformat wünscht.

Die Übergänge zwischen Ressourcen und Repräsentation sind fließend und müssen dem Anwendungsfall gemäß berücksichtigt werden [136]. Innerhalb der Bankanwendung kann es beispielsweise sinnvoll sein, einen elektronischen Ausweis als eigenständige Ressource zu modellieren anstatt ihn als eine mögliche Repräsentation der Ressource zuzulassen. In diesem Fall könnte der Pfad */employees/1* auf Kontaktinformationen eines Bankangestellten verweisen, während der Pfad */employees/1/vcard* auf eine elektronische Visitenkarte verweist.

2.3.3 Einheitliche Schnittstelle

Für alle Ressourcen existiert eine einheitliche Schnittstelle. Über diese Schnittstelle kann mit Ressourcen interagiert werden. Der Vorteil beim Einsatz einer einheitlichen Schnittstelle ist der Einsatz von standardisierten Clients. Ist ein Client in der Lage mit einer einzigen Ressource zu interagieren, so kann er auch mit jeder beliebigen anderen Ressource interagieren. Die Schnittstelle beschreibt alle potentiell möglichen Zugriffsmethoden für die Menge der Ressourcen. Jede Ressource kann mit einer Teilmenge dieser Methoden angesprochen werden. Jegliche andere Interaktion mit den Ressourcen ist unmöglich.

Fielding hat die wesentlichen Merkmale des Internet hinsichtlich Skalierung und Performanz untersucht [44]. Das dem Internet zugrunde liegende Protokoll ist HTTP. Deswegen wird REST häufig mit HTTP assoziiert. Dennoch gibt es eine Reihe weiterer Protokolle, mit welchen sich RESTful Services umsetzen lassen. Beispielsweise ist das Constrained Application Protocol (CoAP [78]) eine leichtgewichtige Alternative zu HTTP beim Einsatz im Internet of Things (IoT) [90]. Auch über CoAP können RESTful Services erstellt werden [21].

Da HTTP jedoch das am weitesten gebräuchlichste dieser Protokolle ist, soll die Schnittstelle des HTTP-Protokolls als Beispiel dienen. Das Protokoll definiert eine Reihe von Methoden, die der Client innerhalb einer Anfrage verwenden kann: die so genannten HTTP-Verben. Beispiele für HTTP-Verben sind GET zum Abfragen einer Ressource, POST zum Erstellen einer neuen Ressource, PUT zum Aktualisieren einer Ressource und DELETE zum Löschen einer Ressource. Die einheitliche Schnittstelle zur Interaktion mit Ressourcen beschränkt sich in diesem Beispiel dann auf die Methodenmenge {GET, POST, PUT, DELETE}.

Idempotenz und Gefahrlosigkeit Das wichtigste Argument bei der Verwendung von Nomen anstatt von Verben in URIs ist die Idempotenz bestimmter Methoden. Singh [131] definiert idempotente Methoden als diejenigen Methoden, welche bei mehrmaliger Ausführung keine Seiteneffekte über die erste Ausführung hinaus haben.

Mathematisch beschreibt Idempotenz die Eigenschaft, eine Verknüpfung mehrfach auf ein Element anzuwenden ohne das Ergebnis nach der ersten Ausführung zu verändern. In der Informatik lässt sich diese Eigenschaft so interpretieren, dass sich eine Operation gefahrlos beliebig oft wiederholen lässt, ohne in einen anderen Zustand zu gelangen. Für die Ressourcen eines RESTful Service bedeutet dies also, dass eine idempotente Methode gefahrlos mehrfach auf eine Ressource angewendet werden kann. So ist das Abfragen einer Ressource mittels eines GET-Kommandos beispielsweise idempotent. Die Abfragen können beliebig oft ausgeführt werden ohne den Zustand der Ressource zu verändern. Die HTTP-Methoden GET, PUT und DELETE sind idempotent. Die Methode POST ist es nicht.

Methoden können als gefahrlos eingestuft werden, wenn ihre Verwendung die Ressource nicht verändert. Im Gegensatz zur Idempotenz, wird bei gefahrlosen Methoden auch beim ersten Aufruf die Ressource nicht manipuliert. Die HTTP-Methode GET ist gefahrlos. Die Methoden POST, PUT und DELETE sind es nicht.

Gefahren falschen URI-Designs Werden in URIs nun Verben verwendet, die eine Methode beschreiben, so kann dies im Zusammenspiel mit der Idempotenz der HTTP-Methoden zu unerwünschten Effekten führen. Denkbar wäre beispielsweise, dass ein Server eine Funktion zum Löschen der elektronischen Visitenkarte eines Bankangestellten

anbietet. REST-konform würde diese abgebildet, indem eine DELETE-Anfrage an die URI *http://www.example.org/employees/1/vcard* geschickt wird. Eine alternative Möglichkeit die Funktion umzusetzen ohne die HTTP-Verben zu berücksichtigen wäre über den Pfad *http://www.example.org/deleteVCard?id=1* möglich. Anstatt die Verben in der URI anzugeben, werden sie in der Adresse gekapselt. Die Funktion könnte also mit Hilfe eines einfachen Links umgesetzt werden. Die große Gefahr eines solchen Designs ist der Bruch mit der Idempotenz [136]. Ein Mensch erkennt eventuell aus dem Kontext, dass es sich hierbei um eine Funktion zum Löschen eines Bildes handelt. Eine Maschine geht jedoch davon aus, dass eine gefahrlose GET-Anfrage möglich ist, da die GET-Methode idempotent ist. Tatsächlich ist das Bild jedoch nach der ersten Anfrage gelöscht.

2.3.4 Hypermedia As The Engine Of Application State

Hypermedia As The Engine Of Application State beschreibt ein Konzept, das die Skalierbarkeit für das verteilte System sicherstellt und gleichzeitig hilft, Client und Server weiter zu entkoppeln. Die Idee von HATEOAS ist der Einsatz zustandsloser Kommunikation zwischen Client und Server [109]. Hierdurch werden auf Serverseite Aufwände reduziert, da die Verwaltung von Zuständen entfällt. Als Konsequenz können die Kapazitäten dazu eingesetzt werden, weitere Anfragen zu bedienen, was die Skalierbarkeit des Systems erhöht. Zustandslose Kommunikation ermöglicht es dem Server, zwischen verschiedenen Anfragen eines Client auf die Haltung eines Zustands zu verzichten und somit Rechen- und Speicherkapazitäten frei zu halten [117, 147].

Um eine zustandslose Kommunikation zu ermöglichen, muss jede Anfrage eines Client unabhängig von vorherigen Anfragen auf der Seite des Servers verarbeitet werden können. Damit ein Client dennoch in einen möglichen nächsten Anwendungszustand gelangen kann, muss der Server alle möglichen Zustandsübergänge in seiner Antwort mitsenden. Dies kann entweder als Teil der Repräsentation einer Ressource passieren oder abhängig vom eingesetztem Protokoll über Header-Felder. Für das Beispiel der Konto-Ressource könnte dies bedeuteten, dass die Ressource einen Verweis auf die Überweisungsübersicht enthält. Auflistung 2.18 zeigt eine mögliche Antwort auf eine Anfrage eines Client an eine Konto-Ressource mit der URI *http://www.example.org/accounts/123*. Neben Informationen zum Konto ist der Verweis zur Überweisungsübersicht enthalten. Der Client kann nun diesem Verweis folgen und somit in einen neuen Anwendungszustand gelangen. Der Übergang in den neuen Zustand ist wieder eine eigenständige Anfrage des Clients an den Server. Der Server braucht zur Bearbeitung dieser Anfrage keinerlei Informationen über vorherige Anfragen des Clients.

```
1 HTTP/1.1 200 OK
2
3 <html>
4   <header></header>
5   <body>
6     <p>balance: 100</p>
7     <a href="/accounts/123/transfers">Overview</a>
8   </body>
9 </html>
```

Auflistung 2.18: HATEOAS

Eine wichtige Unterscheidung muss zwischen Anwendungszustand und Ressourcenzustand getroffen werden. Eine einzelne Ressource kann einen Zustand haben. Etwa kann eine Überweisung einen Zustand wie *offen* oder *verbucht* haben. Welche Zustände Ressourcen annehmen können, ist abhängig vom Anwendungsumfeld und spielt nur eine untergeordnete Rolle bei HATEOAS. Bei HATEOAS geht es vielmehr um den Anwendungszustand. Der Server muss derart entlastet werden, so dass er sich nicht merken muss, in welchem Teil der Anwendung sich ein Client befindet und welche Möglichkeiten der Client in diesem Zustand hat.

Eine Umsetzungsmöglichkeit für HATEOAS ist die Verwendung von Link-Headern wie in [92] beschrieben. Hierbei werden alle möglichen Zustandsübergänge als Header der Antwort des Servers hinzugefügt. Dies ermöglicht insbesondere für die maschinelle Verarbeitung der Zustandsübergänge eine Vereinfachung, da das anfragende System nicht die Inhalte der Antwort auswerten muss, um die Zustandsübergänge zu erfahren.

HATEOAS ist in der Theorie schwierig zu verstehen für unerfahrene Entwickler und erfordert Erfahrung [144]. Dabei wendet jeder Webnutzer HATEOAS in der Praxis an [116]. Browser basieren auf HATEOAS. Verwendet der Benutzer einen Browser, so durchläuft er einen Algorithmus:

1) Empfange die Hypermedia-Repräsentation einer Ressource.

2) Interpretiere die Repräsentation, um den aktuellen Ressourcenzustand zu bekommen.

3) Entscheide welcher Hypermedia-Verweis zum Ziel (zum nächsten Anwendungszustand) führt und folge ihm.

4) Wiederhole die Schritte bis die gewünschte Ressource/Repräsentation gefunden ist.

In Level 3 des Richardson Maturity Models wird HATEOAS eingesetzt. Ein System welches die Bedingungen dieses Levels erfüllt, muss also mögliche Zustandsübergänge zusammen mit der Repräsentation der Ressource an den Client schicken. Die Zustandsübergänge werden dabei mittels Hypermedia transferiert, deshalb der Ausdruck Hypermedia As The Engine Of Application State.

Die beiden entscheidenden Vorteile des HATEOAS-Paradigmas sind die Verbesserung der Skalierbarkeit des Systems sowie die Entkopplung von Client und Server. Die Verbesserung der Skalierbarkeit wird durch eine zustandslose Kommunikation zwischen Client und Server erreicht. Da der Server die Verwaltung des Anwendungszustands an den Client überträgt, braucht er selbst keine Informationen zur Kommunikation mit dem Client verwalten. Dies bedeutet, dass er jede Anfrage individuell verarbeiten kann und nach der Bearbeitung keinerlei Kapazitäten für die Kommunikation mit dem Client mehr bereithalten muss.

HATEOAS hilft dabei Client und Server zu entkoppeln, da es möglich wird, standardisierte Hypermedia-Clients zu verwenden. Durch HATEOAS ist es möglich, Änderungen auf Seite des Servers vorzunehmen ohne Adaptionen auf Seite des Clients vornehmen zu müssen. Clients können sich automatisch an Änderungen anpassen [117], da sie die Kommunikation über die einheitliche Schnittstelle beherrschen und nach jeder Anfrage entscheiden, was der nächste Anwendungszustand sein soll.

2.3.5 Stand der Technik

Pautasso beschreibt in [108] die Herausforderungen bei der Erstellung von Kompositionen von RESTful Services. Zunächst erklärt der Autor die Aufgaben verschiedener Komponenten wie *Clients*, *Proxies*, *Gateways* und *Origin Servern*. Proxies und Gateways werden üblicherweise zwischen Client und Server platziert, um den Server von bestimmten Aufgaben zu entlasten. Beispielsweise können Proxies und Gateways dazu dienen Caching oder Zugriffskontrolle durchzuführen. Bei einem kompositionellen RESTful Service existieren nun neue Herausforderungen, da nicht mehr nur ein einziger Server als Ursprung dient, sondern eine Menge von Servern. Beispielsweise ist Idempotenz eine Eigenschaft bei der es fraglich ist, ob sie bei einem kompositionellen Service noch Gültigkeit besitzt. Weitere Herausforderungen sind unter anderem Caching und Fehlerbehandlung. Da Fehlermeldungen ihren Ausgang auch in der Verweigerung von Zugriffsberechtigungen haben können, ist demnach auch Zugriffskontrolle eine Herausforderung bei Kompositionen von RESTful Services.

In [110] vergleichen die Autoren RESTful Web Services mit sogenannten „Big“ Web Services. Big Web Services setzen sich aus einer Klasse von Spezifikationen des World Wide Web Consortium (W3C) zusammen. Diese beinhalten etwa das Simple Object Access Protocol (SOAP [149]) und die Web Service Description Language (WSDL [150]). Die Autoren verwenden einen quantitativen Ansatz, um die beiden Stile hinsichtlich architektonischer Prinzipien zu untersuchen. Eine leichtgewichtige Infrastruktur wird als eine entscheidende Stärke von REST ausgemacht, da hierdurch mit geringen Aufwänden und wenigen Werkzeugen Web Services erstellt werden können. Dies ist ein Grund für die breite Akzeptanz von REST. Die Einfachheit von

RESTful Web Services im Vergleich mit Big Web Services wird dadurch erklärt, dass weniger architektonische Entscheidungen getroffen werden müssen. Als Schwächen machen die Autoren Schwierigkeiten beim Einsatz der einheitlichen Schnittstelle aus. Es wird zwischen Hi-REST-Empfehlungen und Lo-REST-Implementierungen unterschieden. Bei Hi-REST werden alle Methoden der Schnittstelle eingesetzt (beispielsweise GET, POST, PUT, DELETE), während Lo-REST aufgrund technischer Limitierungen lediglich auf eine Teilmenge der Methoden setzt (etwa GET, POST).

Ein interessanter Ansatz, um die Bedeutung der Merkmale von RESTful Services zu beschreiben, wird in [34] vorgestellt. Die Autorin nennt die Konsequenzen, die auftreten, wenn die Paradigmen von REST nicht umgesetzt werden. Ressourcenorientierung als fundamentalste Eigenschaft eines RESTful Service hat eine zentrale Bedeutung. Einige Technologien brechen mit der Ressourcenorientierung und nutzen HTTP lediglich als Tunnel. Innerhalb von JavaScript-Anwendungen beispielsweise können Benutzer agieren, ohne dass sich dabei die Ressourcenadresse ändert. Der Zurück-Knopf eines Browsers funktioniert in solchen Anwendungen nicht, auch können keine Lesezeichen gesetzt werden und auch das Teilen des aktuellen Inhalts ist nicht möglich, da andere Benutzer zum Startpunkt der JavaScript-Anwendung gelangen. Besonders schwerwiegend ist jedoch, dass Caching durch solche Anwendungen ausgehebelt wird und damit die Infrastruktur des Web belastet wird. Die einheitliche Schnittstelle wird als existentiell für das heutige Web betrachtet. Ohne eine einheitliche Schnittstelle wäre es nicht möglich, Suchen im Web durchzuführen. Schließlich wären ohne Hypermedia-Unterstützung die Benutzer selbst dafür verantwortlich, jede einzelne Ressourcenadresse einzugeben und die gewünschte HTTP-Methode auszuwählen.

2.4 OAuth

Autorisierung im Kontext von REST ist standardisiert, wenn es darum geht Ressourcen im Internet zu teilen. OAuth ist ein offenes Protokoll, welches entworfen wurde, um Benutzerdaten in Form von Ressourcen zwischen Client- und Server-Anwendungen zu teilen, ohne dabei Benutzernamen und Passwörter den jeweiligen Anwendungen offen legen zu müssen. Das im Protokoll definierte Verfahren basiert auf einer Reihe von Zugriffstokens. Version 2.0 des Protokolls wurde durch die Internet Engineering Task Force in RFC 6749 [3] beschrieben. Schnittstellensicherheit, Serviceintegration und Autorisierungsdelegation sowie vereinigte Identitäten sind die Stärken von OAuth [132].

Der OAuth-Standard definiert vier Rollen, welche in verschiedenen Nachrichtensequenzen (Flows) zum Einsatz kommen.

- **Resource Owner/User:** der Resource Owner ist das Subjekt, welchem die Daten gehören und welches dazu in der Lage ist, den Zugriff auf die Daten zu steuern. Die Rolle des Resource Owner wird auch als User bezeichnet, wenn der Resource Owner menschlich ist.

- **Resource Server:** der Resource Server ist der Netzwerkteilnehmer auf dem Ressourcen hinterlegt sind. Der Server gewährt nur Zugriff auf die Ressourcen eines Resource Owner, wenn das anfragende Subjekt (der Client) ein gültiges Token für den Zugriff besitzt.

- **Client:** ein Client ist eine Anwendung, welche auf die Daten eines Resource Owner/User in einer anderen Anwendung zugreifen möchte.

- **Authorization Server:** der Authorization Server ist ein Netzwerkteilnehmer, welcher Tokens für den Zugriff auf Daten an Clients ausstellt. Dies geschieht in Abhängigkeit der Entscheidung des Resource Owners.

Der am häufigsten eingesetzte Flow ist der *Authorization Code Flow*, auch bekannt als *Web Server Flow*. Der Flow ist in Abbildung 2.14 dargestellt und besteht aus sieben Schritten von denen sechs standardisiert sind. Der letzte Schritt ist anwendungsabhängig durchzuführen.

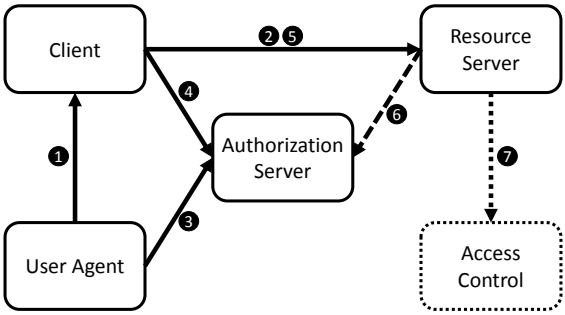

Abbildung 2.14: OAuth - Authorization Code Flow

1. Ein User (Resource Owner) verwendet einen User Agent (beispielsweise einen Browser), um auf eine Client-Anwendung zuzugreifen. Dabei möchte der User der Client-Anwendung kontrolliert Zugriff auf seine Daten auf dem Resource Server gewähren.

2. Die Client-Anwendung möchte auf geschützte Ressourcen zugreifen und startet hierfür den OAuth-Prozess, indem der Client eine Anfrage an den Resource Server sendet. Der Resource Server verweigert zunächst direkten Zugriff auf die Ressourcen und sendet eine Umleitungsnachricht zum Authorization Server an den Client. Diese Nachricht leitet der Client an den User Agent weiter.

3. Sobald der User Agent die Umleitungsnachricht empfangen hat, folgt er der Umleitung zum Authorization Server, welcher den User nach einer Zugriffsentscheidung fragt. Der User gibt Usernamen und Passwort ein und trifft die Entscheidung über das Gewähren der Zugriffsberechtigungen. Gewährt der User den Zugriff, so erstellt der Authorization Server einen Access Code und übermittelt diesen an den User Agent.

4. Der User Agent wiederum leitet den Access Code an den Client weiter, welcher über den Access Code nun ein Access Token beim Authorization Server erfragen kann.

5. Mit dem gültigen Access Token kann die Client-Anwendung erneut beim Resource Server die geschützten Ressourcen anfragen.

6. Optional prüft der Resource Server die Validität des Tokens über den Authorization Server. Alternativ kann eine Signatur im Token verwendet werden, um die Validität des Tokens zu gewährleisten.

7. Schließlich muss der Resource Server noch überprüfen, ob die vom Client gesendete Anfrage zu den im Token enthaltenen

Daten passt. Ein Token gilt für sogenannte Scopes. Scopes sind anwendungsabhängig. Welche Aktionen auf welchen Ressourcen im Scope enthalten sind, ist eine Verwaltungsaufgabe des Resource Server. Der OAuth-Standard macht hierzu keine Vorgaben. Es muss also noch eine Zugriffskontrolle vom Scope zur tatsächlichen Anfrage gemacht werden.

Ein typischer Anwendungsfall für diesen Flow ist die Zugriffsgewährung für eine Client-Anwendung auf Daten eines Users, welche auf einem Resource Server gespeichert werden. Dabei muss der User seine explizite Zustimmung geben. Durch das Protokoll wird es möglich, dass die Client-Anwendung ohne Kenntnisse des Usernamens und des Passworts auf die Daten des Benutzers auf dem Resource Server zugreifen kann.

Ein Beispiel für den Einsatz dieses Flows ist der Zugriff einer Client-Anwendung auf die Daten des Users etwa bei Facebook oder Twitter. Die Client-Anwendung leitet den User zu Prozessbeginn an den Authorization Server von Facebook beziehungsweise Twitter weiter. Die Umleitungsnachricht enthält dabei Informationen über die gewünschten Berechtigungen in Form von Scopes. Beispielsweise existieren bei Facebook die Scopes *email* und *user location* [38], um Leseberechtigungen für die Emailadresse und den Wohnort des Users abzubilden. Gewährt der User der Client-Anwendung diesen Zugriff, so erhält der User einen Access Code, welchen er an den Client weiterleitet. Dieser tauscht den Code beim Authorization Server in ein Token und bekommt damit Zugriff auf die Emailadresse und den Wohnort des Users ohne dessen Passwort bei Facebook zu kennen.

Der OAuth *Client Credential Grant Flow* (vgl. Abbildung 2.15) ist eine vereinfachte Version des *Authorization Code Flow*. Dabei werden

die Client-Anwendungen per se als vertrauenswürdig eingestuft und müssen nicht nach Berechtigungen fragen. Dieser Flow kann verwendet werden, wenn die Client-Anwendung selbst die Daten verwaltet oder wenn zuvor ein Arrangement mit dem Authorization Server getroffen wurde.

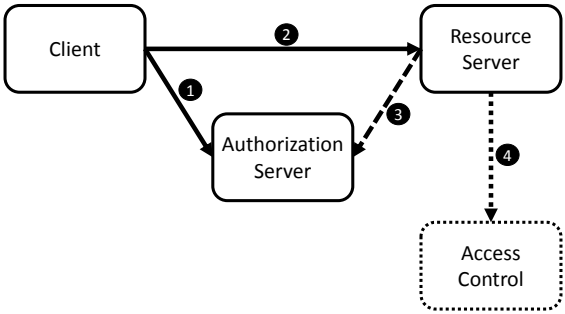

Abbildung 2.15: OAuth - Client Credential Grant Flow

1. Die Client-Anwendung ist generell vertrauenswürdig und möchte auf geschützte Ressourcen zugreifen. Im ersten Schritt authentifiziert sich der Client beim Authorization Server, welcher ihm ein Access Token ausstellt.
2. Mit dem gültigen Access Token, kann die Client-Anwendung beim Resource Server die gewünschten Ressourcen anfragen.
3. Der Resource Server prüft die Validität des Tokens über den Authorization Server oder eine Signatur im Token.

4. Abschließend ist es wie beim *Authorization Code* Flow die Aufgabe des Resource Server zu kontrollieren, ob Scope und tatsächliche Anfrage übereinstimmen.

Zusammenfassend lässt sich sagen, dass im *Authorization Code Flow* der User Zugriffsberechtigungen auf seine Daten von einer Anwendung an eine zweite Anwendung delegiert. Dabei werden interaktive Entscheidungen durch den User getroffen. Im *Client Credential Grant Flow* sind diese nicht notwendig, da den Clients grundsätzlich vertraut wird. Zugriffsrechte werden in Access Tokens über Scopes abgebildet. Es ist die Aufgabe des Resource Server, den Scope in Zugriffberechtigungen zu übersetzen. An dieser Stelle spezifiziert OAuth weder ein Modell noch einen Mechanismus, welche festlegen wie ein Scope in Berechtigungen übersetzt wird [115].

Ein Ansatz zum Zentralisieren von Zugriffsrechten mit Hilfe von OAuth wird in [99] beschrieben. Die Autoren präsentieren einen Ansatz, in welchem ein zentraler Koordinator OAuth-Token ausstellt, welche Clients ermöglichen auf verschiedene Dienste zuzugreifen.

2.5 User Managed Access

User Managed Access (UMA [70]) ist ein Protokoll, welches auf OAuth aufbaut und im Jahr 2015 als Profil des OAuth-Protokolls in Version 1.0 von der Kantara Initiative veröffentlicht wurde. Das Protokoll gibt dem Benutzer die Möglichkeit, den Zugriff auf seine Daten zu kontrollieren ohne dabei explizite Zugriffsberechtigungen zur Laufzeit erteilen zu müssen, so wie es beim Authorization Code Flow der Fall ist. Vielmehr hinterlegt der Benutzer die Zugriffsberechtigungen

bei einem Autorisierungsserver im Vorfeld. Genau wie bei OAuth werden auch bei UMA Scopes verwendet. Scopes werden Zugriffsberechtigungen zugeordnet und gleichzeitig werden auch Ressourcen bestimmten Scopes zugewiesen. Möchte eine dritte Partei auf eine Ressource zugreifen, werden die Zugriffsbedingungen für diesen Scope ausgewertet. Die dritte Partei bekommt bei einer positiven Zugriffsentscheidung Zugriff auf alle Ressourcen mit diesem Scope. Da UMA auf OAuth basiert, werden auch hier Tokens eingesetzt. Allerdings kommen im Vergleich mit OAuth zusätzliche Tokens zum Einsatz. UMA beschreibt keine konkreten Zugriffsmodelle, die für die Bestimmung der Zugriffsentscheidung eingesetzt werden müssen. Dies wird Implementierungen des Standards überlassen. Sowohl OAuth als auch UMA sind benutzerzentrierte Ansätze. Typischerweise werden damit Applikationen adressiert, in denen sich Zugriffsrechte nicht häufig ändern.

Ein wesentlicher Unterschied zum OAuth-Standard ist die Einführung der Requesting Party. Die Requesting Party beschreibt den Anfragesteller und wird in UMA - anders als in OAuth - unabhängig zum Resource Owner behandelt. Der Anfragesteller muss also nicht gleichbedeutend mit dem Resource Owner sein. Zudem ist die Zustimmung durch den Resource Owner beim Ausstellen von Zugriffstoken asynchron. Das heißt, der Resource Owner muss nicht zum Zeitpunkt der Anfrage einwilligen, sondern kann Zugriffsberechtigungen hinterlegen, welche bei einer Anfrage ausgewertet werden. Hierdurch wird es möglich, Zugriffsberechtigungen für beliebige Requesting Parties zu erstellen. Ein Resource Owner kann so Zugriffsberechtigungen für Dritte definieren und muss Zugriffsversuchen durch Dritte darüber hinaus nicht mehr explizit zustimmen. Abbildung 2.16 zeigt die UMA-Akteure und ihre Beziehungen.

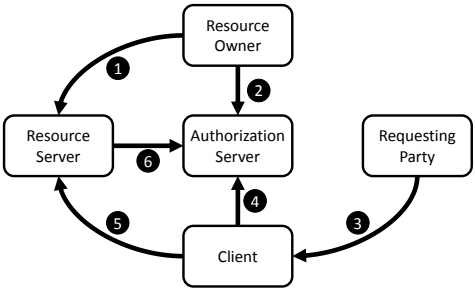

Abbildung 2.16: User Managed Access

1. Der Resource Owner besitzt Ressourcen, welche er auf einem Resource Server hinterlegt.
2. Für diese Ressourcen kann der Resource Owner Zugriffsberechtigungen beim Authorization Server hinterlegen.
3. Die Requesting Party kann über eine Client-Anwendung versuchen auf die Ressourcen des Resource Owner zuzugreifen.
4. Der Client stellt eine Anfrage beim Authorization Server, ob der Requesting Party Zugriff gewährt werden kann.
5. Wird der Zugriff gewährt, erhält der Client ein Token, mit dem er sich an den Resource Server wendet.
6. Der Resource Server prüft die Gültigkeit des Tokens beim Authorization Server und gewährt gegebenenfalls Zugriff.

UMA stellt keinerlei Vorgaben, wie Zugriffsberechtigungen beim Authorization Server hinterlegt und ausgewertet werden. Es wird also kein Zugriffskontrollmodell vorgeschrieben. Analog zu OAuth werden auch bei UMA Scopes verwendet, welche einen Rahmen von Zugriffsberechtigungen abbilden.

Kapitel 3

Zugriffskontrolle für RESTful Services mit XACML

Dieses Kapitel stellt Untersuchungen zum Einsatz von XACML bei RESTful Services vor. Es werden Optimierungsansätze erläutert, welche zeigen wie XACML für REST eingesetzt werden kann. Die Beiträge dieser Arbeit zum Optimieren von XACML für REST wurden in [66] und [67] veröffentlicht. Zudem beinhaltet das Kapitel eine Analyse von XACML, welche in [65] publiziert wurde.

3.1 Optimierung von XACML

XACML ist ein kompositioneller Ansatz zur Abbildung von Zugriffsberechtigungen [114]. Das bedeutet, dass die finale Zugriffsentscheidung durch die Kombination mehrerer Teilentscheidungen entsteht. Die finale Entscheidung hängt also wesentlich von den Kombinationsvorschriften für die Teilentscheidungen ab. Darüber hinaus ist XACML ein generischer Ansatz. Dies bedeutet, dass die Attributbedingungen, die zum Finden von Teilentscheidungen erfüllt werden müssen, beliebige Bedingungen sein können. Beispielsweise kann eine Teilentscheidung erfordern, dass das anfragende Subjekt einen bestimmten Namen hat und eine bestimmte Anfragemethode verwendet wird, während eine andere Teilentscheidung es erfordert, dass eine bestimmte Ressource angefragt wird und das Subjekt von einem bestimmten Standort aus anfragt. Da die erforderlichen Attributbedingungen sehr heterogen sein können, können ohne zusätzliche Einschränkungen keine indexbasierten Verfahren bei der Entscheidungsfindung genutzt werden. Diese beiden Eigenschaften von XACML (kompositioneller Ansatz und heterogene Attributbedingungen) sind ein Nachteil beim Einsatz von XACML zum Schutz von RESTful Services, da so Merkmale von RESTful Services nicht ausgenutzt werden können, um die Effizienz des Zugriffskontrollmechanismus auch für große Datenmengen zu gewährleisten. Etwa kann der Zugriff innerhalb von großen Mengen von Ressourcen nicht mehr auf einfache Weise effizient gesteuert werden, obwohl die Ressourcen alle ein Merkmal des gleichen Typs besitzen: die Ressourcenadresse in Form einer URI. Dennoch lassen sich mit einer Reihe von Optimierungen verbesserte Ergebnisse beim Einsatz von XACML für RESTful Services erzielen. Die Maßnahmen zur Optimierung werden im Folgenden beschrieben.

Das effiziente Design eines Regelwerks sollte eine schnelle Verarbeitung einer Zugriffsanfrage ermöglichen, damit das Zugriffskontrollsystem nicht zum Flaschenhals einer gesamten Anwendung wird. Gleichzeitig sollte das Regelwerk einfach zu pflegen sein. Dies gilt insbesondere dann, wenn häufige Änderungen am Regelwerk zu erwarten sind, beispielsweise wenn neue Ressourcen erzeugt werden, für die dann auch neue Zugriffsberechtigungen vergeben werden müssen. In solch einem Fall müssen neue Einträge im Regelwerk erstellt werden oder existierende Einträge angepasst werden. Die Wartbarkeit des Regelwerks ist daher von großer Bedeutung.

3.2 Evaluationskosten von Zugriffsanfragen

Ein in XACML geschriebenes Regelwerk stellt einen unidirektionalen Graphen dar. Innerhalb dieses Graphen sind keinerlei zirkulare Pfade möglich. Die Knoten des Graphen sind die XACML Targets, in denen Zugriffsbedingungen beschrieben werden. Kanten zwischen den Knoten stellen logische Verundungen der Zugriffsbedingungen dar.

Um den Aufwand der Verarbeitung einer Zugriffsanfrage innerhalb eines einzelnen Knotens dieses Graphen zu bestimmen, wird eine Kostenfunktion c definiert als:

$$c : T \times Q \to \mathbb{Q} \tag{3.1}$$

mit T als Menge der Targets eines XACML-Regelwerks, Q als Menge der möglichen Zugriffsanfragen und $\mathbb{Q}$ als Menge der rationalen Zahlen.

Um die konkrete Kostenfunktion herzuleiten, wird der Pfad zwischen zwei Targets betrachtet. Dieser beschreibt die Vereinigung aller

Zugriffsbedingungen zwischen diesen beiden Targets. Dies bedeutet, dass die Beschreibung eines Pfads in die Potenzmenge der Targets abbildet:

$$p : T \times T \to \mathcal{P}(T) \tag{3.2}$$

mit $\mathcal{P}(T)$ als Potenzmenge der Menge T. Wird der Pfad zwischen dem Target der ersten Regel und dem Target des Policy Sets aus Abbildung 2.7 betrachtet, so entspricht der Pfad der logischen Verundung der Targets des Policy Sets (der Wurzel), der Policy und der Rule. Diese Verundung kann ausgedrückt werden als: *Resource.path = /accounts/123* $\wedge$ *Action.method = GET* $\wedge$ *Subject.id = 456*.

Die Länge eines Pfads kann dann definiert werden als die Anzahl an Targets, welche zwischen den Start- und End-Targets liegen. Da es sich um einen unidirektionalen Pfad ohne Zyklen handelt, ist der Pfad zwischen zwei Targets eindeutig und besitzt eine feste Länge:

$$|p(t_1, t_2)| \tag{3.3}$$

Der zuvor als Beispiel genannte Pfad besitzt eine Länge von 3, da inklusive dem Target des Policy Sets und dem Target der Rule drei Targets logisch entlang des Pfads verknüpft werden.

Die Menge der untergeordneten Targets eines Targets t kann mit Hilfe der Definition der Länge eines Pfads ausgedrückt werden als:

$$T'_t := \{t' \in T \mid |p(t, t')| = 2\} \tag{3.4}$$

Sei α_t der Combining Algorithm eines Policy Sets oder einer Policy mit dem Target t und sei A die Menge der Combining Algorithmen.

Weiterhin sei ε ein Effect aus der Menge der Effects E, so gilt die folgende Aussage für XACML:

$$\forall \alpha_t \in A \; \exists \varepsilon \in E : evaluate(t_i, q) = \varepsilon \Rightarrow \alpha_t \; stops; t_i \in T_t', q \in Q \tag{3.5}$$

Dies bedeutet, dass es für jeden gegebenen Combining Algorithm einen oder mehrere Effects gibt, die dafür sorgen, dass der Algorithmus stoppt, wenn einer der untergeordneten Knoten sich zu diesem Effect berechnet. Berechnet sich keiner der untergeordneten Knoten zu einem dieser Effects, so stoppt der Algorithmus nach der Verarbeitung des letzten untergeordneten Knoten. Beispielsweise beinhaltet das in Abbildung 2.7 gezeigte Regelwerk drei Rules, welche in der übergeordneten Policy durch einen Combining Algorithm verknüpft sind. Ist der Combining Algorithm beispielsweise *PermitOverrides* und berechnet sich die Auswertung der ersten Regel zu *Permit*, so wird das Resultat der Evaluation der Policy ebenso *Permit* sein, egal was die Auswertung der zweiten und dritten Rule ergeben würde. Deshalb kann der Combining Algorithm stoppen und auf die Berechnung der Ergebnisse der zweiten und dritten Rule verzichten. Die Funktion γ für ein Target t beschreibt dieses Verhalten und wird definiert als:

$$\gamma_t(t_i, q) = \begin{cases} 1 & \text{if } \forall t' \in \{t_1, ..., t_{i-1}\} : \alpha_t \; does \; not \; stop \\ 0 & \text{if } \exists t' \in \{t_1, ..., t_{i-1}\} : \alpha_t \; does \; stop \end{cases} \tag{3.6}$$

mit $t_i \in T_t'$ und $q \in Q$. Für das zuvor genannte Beispiel berechnet sich die γ-Funktion des mit der Policy assoziierten Targets zu 1 für das untergeordnete Target der ersten Rule und zu 0 für das untergeordnete

Target der zweiten und dritten Rule. Die Kostenfunktion c kann mit diesen Definitionen beschrieben werden als:

$$c(t,q) = \tau(t,q) + \sum_{i=1}^{|T_t'|} \gamma_t(t_i,q) * c(t_i,q) \tag{3.7}$$

Hierbei beschreibt die Funktion $\tau(t,q)$ den Aufwand, um Attributbedingungen eines Targets t gegen die Attribute einer Zugriffsanfrage q auszuwerten. Dies bedeutet, der Aufwand der Auswertung ist auch davon abhängig, wie viele Attribute in der Zugriffsanfrage q enthalten sind. Zusammenfassend kann gesagt werden, dass die Aufwände zur Berechnung einer Zugriffsentscheidung abhängig sind von der Anzahl der Attributbedingungen innerhalb eines Targets, der Summe der untergeordneten Targets und des Combining Algorithm beziehungsweise der Reihenfolge der untergeordneten Knoten.

Beispiel In Abbildung 2.7 wurde ein Beispiel für ein XACML-Regelwerk eingeführt und in Auflistung 2.2 eine beispielhafte Anfrage mit XACML formuliert. Anhand dieser Beispiele sollen die Kosten für eine Zugriffsanfrage q betrachtet werden.

Betrachtet man zunächst die Wurzel (das Policy Set - PS), so gilt:

$$\gamma_{t_{PS}}(t_P,q) = 1;$$

da das Policy Set lediglich ein untergeordnetes Element besitzt und der Combining Algorithm nicht vor der Bearbeitung dieses Elements stoppen kann. Somit ergibt sich für die Kosten zur Auswertung des Policy Sets:

$$c(t_{PS},q) = \tau(t_{PS},q) + \sum_{i=1}^{|T'_{t_{PS}}|} 1 * c(t_P,q)$$

Dies lässt sich vereinfacht schreiben als:

$$c(t_{PS},q) = \tau(t_{PS},q) + c(t_P,q)$$

Die Policy besitzt drei untergeordnete Elemente. Bei der Zugriffsanfrage aus dem Beispiel, stoppt der Combining Algorithm erst bei der dritten Rule, so dass für alle Regeln R_i gilt:

$$\gamma_{t_P}(t_{r_i},q) = 1;$$

Somit ergibt sich für die Auswertung der Policy:

$$c(t_P,q) = \tau(t_P,q) + \sum_{i=1}^{|T'_{t_P}|} 1 * c(t_{R_i},q)$$

Dies lässt sich für die drei Regeln in diesem Beispiel formulieren als:

$$c(t_P,q) = \tau(t_P,q) + c(t_{R_1},q) + c(t_{R_2},q) + c(t_{R_3},q)$$

Für die Kosten der Auswertung des Policy Sets lässt sich damit schreiben:

$$c(t_{PS},q) = \tau(t_{PS},q) + \tau(t_P,q) + c(t_{R_1},q) + c(t_{R_2},q) + c(t_{R_3},q)$$

Da die Rules keine untergeordneten Elemente besitzen, ergeben sich die Kosten zu:

$$c(t_{PS}, q) = \tau(t_{PS}, q) + \tau(t_P, q) + \tau(t_{R_1}, q) + \tau(t_{R_2}, q) + \tau(t_{R_3}, q)$$

Im Wesentlichen ist dies die formelle Beschreibung, dass die Attribute der Zugriffsanfrage q mit den Attributbedingungen innerhalb der Targets des Regelwerks abgeglichen werden müssen, um die Kosten für die Auswertung einer Zugriffsanfrage zu bestimmen.

Der folgende Abschnitt beschreibt, wie für jeden genannten Faktor der Kostenfunktion eine Minimierung stattfinden kann, so dass der Aufwand zur Auswertung einer Zugriffsanfrage insgesamt minimiert werden kann. Danach wird ein für REST optimiertes Verfahren beschrieben, welches die Optimierungen berücksichtigt und zusätzlich die Wartungsaufwände reduziert indem eine ressourcenorientierte Struktur eingeführt wird. Dies ist insbesondere bei der Erweiterung von Regelwerken und dem Auffinden von Fehlerquellen hilfreich.

3.3 Optimierungsstrategien für XACML

Target-Entwurf (Minimieren von $\tau(t,q)$) Wie man anhand von Gleichung (3.7) sehen kann, sollten Attribute mit Bedacht zu Targets hinzugefügt werden. Hierdurch werden die Bedingungen innerhalb des Targets einfach gehalten und damit die Anzahl an Vergleichen, die im ungünstigsten Fall benötigt werden, reduziert. Beispielsweise könnte ein Regelwerk zwei verschiedene Bedingungen für die Anwendbarkeit erfordern. Dies könnte zum einen eine Subjektbedingung sein, welche die Präsenz eines Attributs *id* mit dem Wert *{subjectId}* erfordert und zum anderen eine Ressourcenbedingung, welche die Präsenz eines Attributs *URI* mit dem Wert */accounts/{accountid}* erfordert. Intuitiv

könnten beide Bedingungen in ein gemeinsames Target geschrieben werden, so wie in Abbildung 3.1 dargestellt. Beide Bedingungen sind innerhalb eines Targets zusammengefasst.

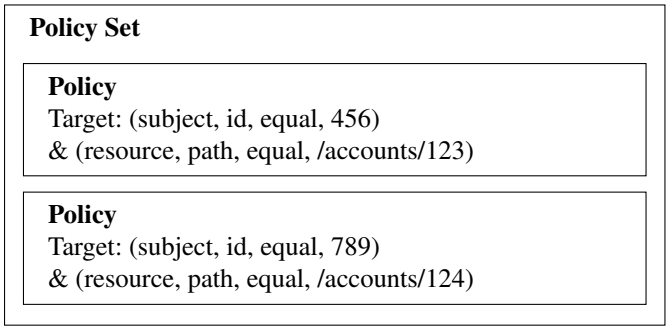

Abbildung 3.1: Target-Entwurf: Max. 4 Vergleiche

Die Verarbeitung einer Anfrage mit einem Subjektattribut *id* und dem Wert *789* sowie einem Ressourcenattribut *path* und dem Wert */accounts/123* könnte im schlimmsten Fall vier Attributvergleiche benötigen, da es keine festgelegte Reihenfolge gibt in der Attribute innerhalb eines Targets ausgewertet werden müssen. Werden die Bedingungen jedoch auf mehrere Targets aufgeteilt, so wie in Abbildung 3.2 dargestellt, wird eine maximale Anzahl von drei Vergleichen benötigt.

Diese Optimierung hilft dabei die Anzahl an Vergleichen - und damit die Bearbeitungskosten einer Anfrage - zu reduzieren, welche für eine nicht anwendbare Zugriffsanfrage auftreten können. Die benötigte

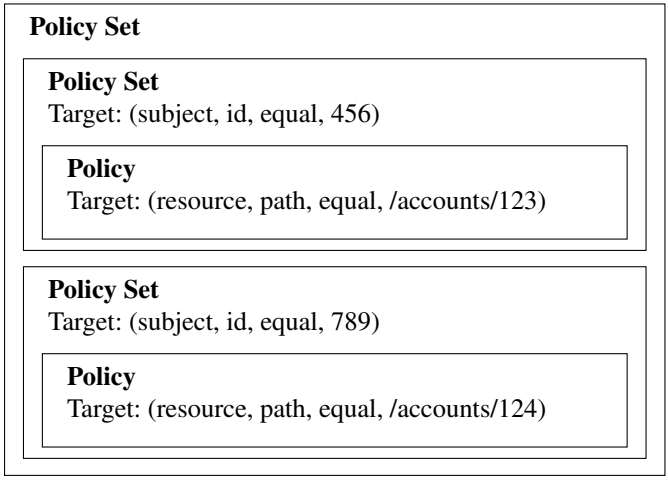

Abbildung 3.2: Target-Entwurf: Max. 3 Vergleiche

Anzahl an Vergleichen für eine anwendbare Zugriffsanfrage bleibt dabei unverändert. Es werden insbesondere Schwankungen in der Bearbeitungszeit einer Anfrage verringert, da eine feste Sequenz bei der Auswertung von Attributen vorgegeben wird.

Zusätzlich hilft die Optimierung auch dabei, die Wartbarkeit eines Regelwerks zu erhöhen. Es wird einfacher neue Bedingungen hinzuzufügen, welche lediglich einen Teil der bereits existierenden Bedingungen wiederverwenden. Beispielsweise könnte relativ einfach eine Bedingung hinzugefügt werden, welche eine Subjektbedingung

mit dem Attribut *id* und dem Wert *456* erfordert, jedoch nicht die Ressourcenbedingung mit dem Attribut *path* und dem Wert */accounts/123*. Wird das Regelwerk gemäß der Struktur in Abbildung 3.1 verwendet, so muss für die Integration der neuen Bedingung zunächst eine Aufspaltung des ersten Targets erfolgen. In der Struktur aus Abbildung 3.2 kann die neue Bedingung einfach im ersten Policy Set integriert werden.

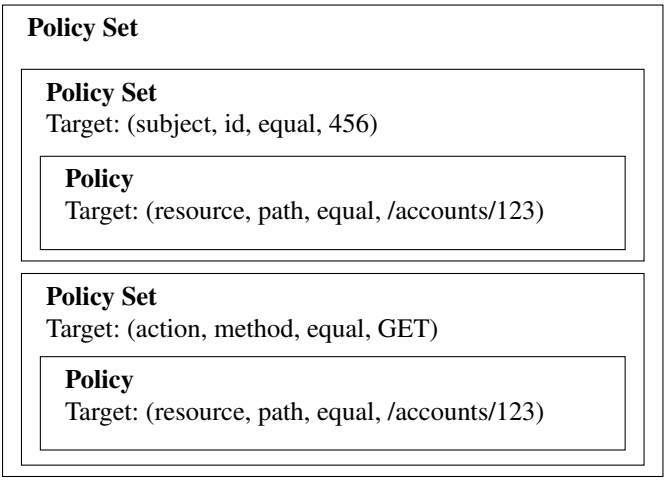

Abbildung 3.3: Anzahl untergeordneter Knoten: Höhere Verzweigungsknoten

Anzahl der Kinderknoten (Minimieren von $\sum_{i=1}^{|T_t'|}$) Aus Gleichung (3.7) kann abgelesen werden, dass die Kosten für die Verarbeitung einer Zugriffsanfrage unter anderem von der Anzahl der untergeordneten Elemente abhängt. Deshalb ist es sinnvoll die Anzahl der untergeordneten Knoten zu minimieren. In Zweigen, in denen gleiche Bedingungen auftreten, sollten diese Bedingungen zusammengefasst werden, wann immer dies möglich ist. Dies bedeutet, dass Verzweigungspunkte innerhalb des Baums so weit unten wie möglich positioniert werden sollten.

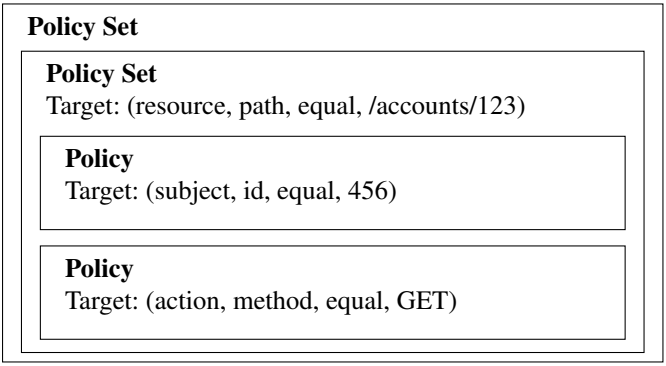

Abbildung 3.4: Anzahl untergeordneter Knoten: Niedrigere Verzweigungsknoten

Dies hilft ebenfalls dabei, die Wartungsaufwände zu minimieren, da das Auftreten der exakt gleichen Bedingungen vermindert wird. Aus Abbildung 3.3 und 3.4 erkennt man, dass die Ressourcenbedin-

gungen, welche den *path* betrifft, in der unteren Struktur lediglich noch einmal vorkommt, während die Bedingung in der ersten Struktur noch zweimal vorhanden ist und somit auch potentiell zweimal ausgewertet werden muss.

Die Optimierung beschreibt ein Verfahren, welches auch beim Compilerbau zum Einsatz kommt. Common Subexpression Elimination fasst gemeinsame Teilausdrücke zusammen, so dass diese nicht mehrfach berechnet werden müssen [27].

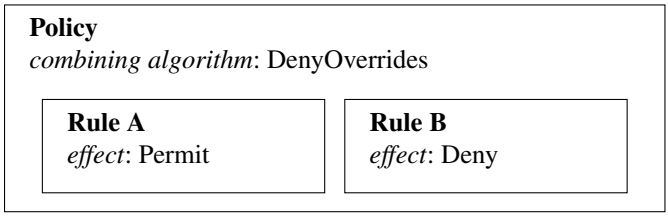

Abbildung 3.5: Normalisierung: Nicht normalisierte Policy

Combining Algorithm und Ordnung der untergeordneten Knoten (Minimieren von γ_k) Die Auswahl des Combining Algorithm und die Anordnung der untergeordneten Kinderknoten hat ebenfalls einen starken Einfluss auf die Performanz. Werden zuerst die Knoten ausgewertet, welche zu einem überschreibenden Effekt führen können, so kann die Verarbeitung der Zugriffsanfrage beendet werden, sobald eine anwendbare Rule gefunden wurde. Dies ist der Grundgedanke von Normalisierung, so wie sie in [93] beschrieben wird.

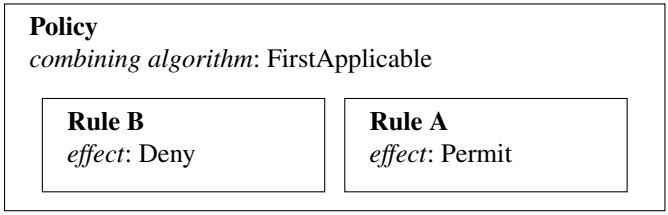

Abbildung 3.6: Normalisierung: Normalisierte Policy

Die Abbildungen 3.5 und 3.6 zeigen den Effekt von Normalisierung. Eine gegebene XACML-Policy mit dem Combining Algorithm *DenyOverrides* und zwei XACML-Rules mit den Effekten *Deny* und *Permit* kann in verschiedenen Anordnungen geschrieben werden. Abbildung 3.5 zeigt eine Variante, bei der die überschreibende Rule nicht zuerst ausgewertet wird. Bei der Auswertung einer Zugriffsanfrage kann es passieren, dass sowohl Rule A als auch Rule B ausgewertet werden. Wird die XACML-Policy jedoch normalisiert, so wird zuerst die Anwendbarkeit von Rule B geprüft. Ist diese anwendbar, so braucht Rule A nicht geprüft zu werden, da Rule B in jedem Fall das Ergebnis bestimmt, wenn Rule B anwendbar ist. Neben der Reihenfolge der Rules wird bei dieser Optimierung auch der Combining Algorithm verändert. Während in Abbildung 3.5 Rule A vor Rule B aufgeführt wird, ist in Abbildung 3.6 die Reihenfolge vertauscht und Rule B steht an erster Stelle. Gleichzeitig wird der Combining Algorithm zu *FirstApplicable* geändert, so dass Rule B zwar weiterhin den dominierenden Effekt bestimmt, wenn beide Rules anwendbar sind, jedoch die

Auswertung von Rule A aufgrund des neuen Combining Algorithm nicht mehr notwendig ist.

3.4 Regelwerke für RESTful Services

Neben den grundlegenden Optimierungsansätzen aus den vorherigen Abschnitten gilt es bei der Erstellung eines Regelwerks für RESTful Services auch die Umgebungseigenschaften der Services mit einzubeziehen. Wie in den vorherigen Abschnitten beschrieben, ist Ressourcenorientierung eines der wichtigsten und bedeutendsten Konzepte von REST, ohne die sich diese Art von Services praktisch nicht effizient gestalten lassen. Beim Entwurf eines XACML-Regelwerks für REST ist es deshalb sehr sinnvoll, das Regelwerk analog zu den RESTful Services ressourcenorientiert aufzubauen. Ressourcenorientierung ermöglicht die schnelle Identifikation derjenigen Regeln innerhalb des Regelwerks, welche zur Bestimmung der Zugriffsentscheidung tatsächlich ausgewertet werden müssen. Der Target-Entwurf von Policy Sets kann deshalb sehr einfach gehalten werden. Es sollte lediglich genau ein Attribut im Target ausgewertet werden: die Ressourcenadresse. Durch die ressourcenorientierte Gestaltung der Policy Sets wird gleichzeitig eine Normalisierung erreicht. Ressourcenadressen sind eindeutig, so dass nach dem ersten anwendbaren Target kein zweites Target mehr anwendbar sein kann und die Auswertung des Policy Sets abgeschlossen werden kann.

Ein zweiter wesentlicher Aspekt eines RESTful Service ist der Einsatz einer einheitlichen Schnittstelle, weshalb die Menge der durchführbaren Aktionen begrenzt ist. Gleichzeitig ist mit jedem Service-Aufruf exakt eine Methode verbunden. Deshalb ist es beim Entwurf

eines XACML-Regelwerks für REST sehr sinnvoll, dass die Zugriffsmethoden als Targets von XACML-Policies abgefragt werden.

Alle übrigen Zugriffsbedingungen für einzelne Ressourcen werden in XACML-Rules beschrieben, welche den Policies zur Abbildung der einheitlichen Schnittstelle untergeordnet werden. Abbildung 3.7 zeigt ein Beispiel für die Strukturierung eines XACML-Regelwerks für drei Ressourcen. Die Ressourcen werden über die Pfade */accounts*, */accounts/123* und */accounts/123/transfers* adressiert. Für jede Ressource wird ein XACML-Policy Set erstellt. Für verschiedene Zugriffsmethoden wie *GET* und *POST* werden XACML-Policies eingesetzt. Alle übrigen Bedingungen, wie beispielsweise die Überprüfung der *id* eines Subjekts, werden in XACML-Rules durchgeführt.

Für Architekten, Entwickler und auch Anwender ist die Struktur eines solchen Regelwerks verhältnismäßig einfach zu verstehen und schnell navigierbar, da es sich um die gleiche Struktur handelt, die auch der RESTful Services selbst aufweist. Dies ist ein bedeutender Vorteil, da die Identifikation von Fehlern deutlich einfacher möglich wird. Ebenfalls wird die Integration neuer Ressourcen und neuer Zugriffsberechtigungen erheblich vereinfacht, wenn in der Anwendung und im Zugriffskontrollsystem identische Strukturen verwendet werden.

Zusammenfassen von Kinderknoten Erweiterbarkeit ist eines der wesentlichen Merkmale von RESTful Services. Mit einer wachsenden Anzahl von Ressourcen, steigt jedoch auch die Anzahl der Einträge im XACML-Regelwerk. Für Listenressourcen bedeutet dies, dass ein Policy Set im zugehörigen XACML-Regelwerk sehr viele Kinderknoten bekommen kann. Dies wiederum widerspricht der Forderung nach einer möglichst geringen Anzahl an untergeordneten Knoten, wie sie

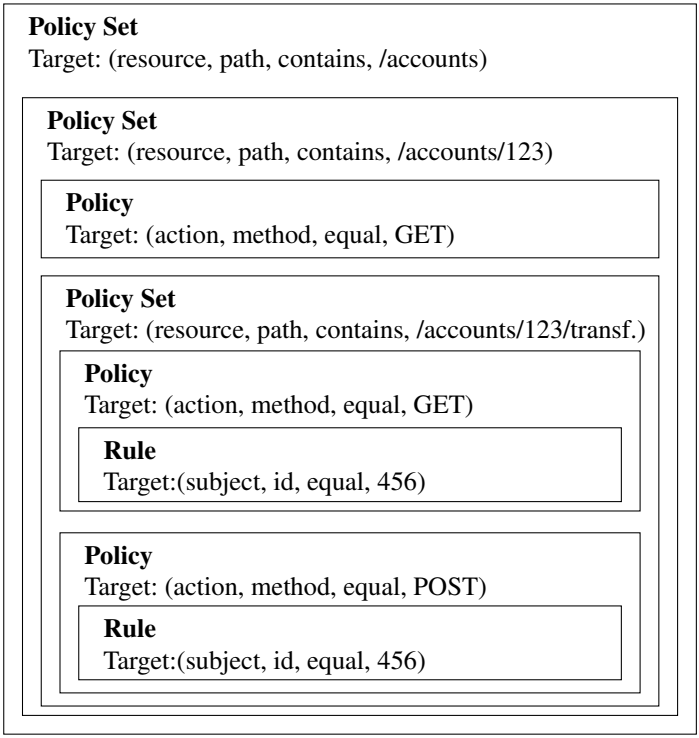

Abbildung 3.7: Beispielhaftes XACML-Regelwerk für RESTful Services

zuvor beschrieben wurde. Werden beispielsweise in das in Abbildung 3.7 dargestellte Regelwerk weitere Konten hinzugefügt, bedeutet dies, dass das übergeordnete Policy Set zusätzliche untergeordnete Elemente bekommt. Mit einer wachsenden Anzahl an Konto-Ressourcen wächst demnach auch die durchschnittliche Antwortzeit (die Zeitspanne vom Eintreffen der Zugriffsanfrage beim Zugriffskontrollsystem und der Rückgabe der Zugriffsentscheidung) für den Zugriff auf solche Ressourcen, da ein XACML-Mechanismus die Elemente sequentiell verarbeiten muss.

Da XACML-Regelwerke von Haus aus in Bäumen organisiert sind, können zusätzliche Baumstrukturen verwendet werden, um dieses Problem zu reduzieren. Dabei wird eine große Menge von Kinderknoten wiederum in einem Baum aufgeteilt. Dieser Baum dient der Verwaltung von Intervallen von untergeordneten Elementen. Mit XACML ist die Aufteilung in solche zusätzlichen Baumstrukturen grundsätzlich keine triviale Aufgabe, da das Regelwerk beliebige Arten von Zugriffsbedingungen abbilden kann. An dieser Stelle soll jedoch lediglich ein Fokus auf die Ressourcenbedingung der Adresse einer Ressource gesetzt werden, bei der eine solche Aufteilung gefahrlos gemacht werden kann, um die Performanz zu steigern.

Abbildung 3.8 zeigt wie ein zusätzlicher Binärbaum zur Beschleunigung der Auswertungszeiten in ein XACML-Regelwerk integriert werden kann. Der Baum des Regelwerks aus Abbildung 3.8 kann acht Konto-Ressourcen verwalten. Bei der Erstellung des Baums wird dazu in einem ersten Schritt die Menge der Konto-Ressourcen in zwei Intervalle unterteilt, für welche jeweils ein XACML-Policy Set angelegt wird, welches anwendbar wird, wenn die angefragte Ressource in dessen Intervall liegt. Dies wird so lange wiederholt bis lediglich eine einzelne Ressource übrig bleibt. In Abbildung 3.8 werden eckige

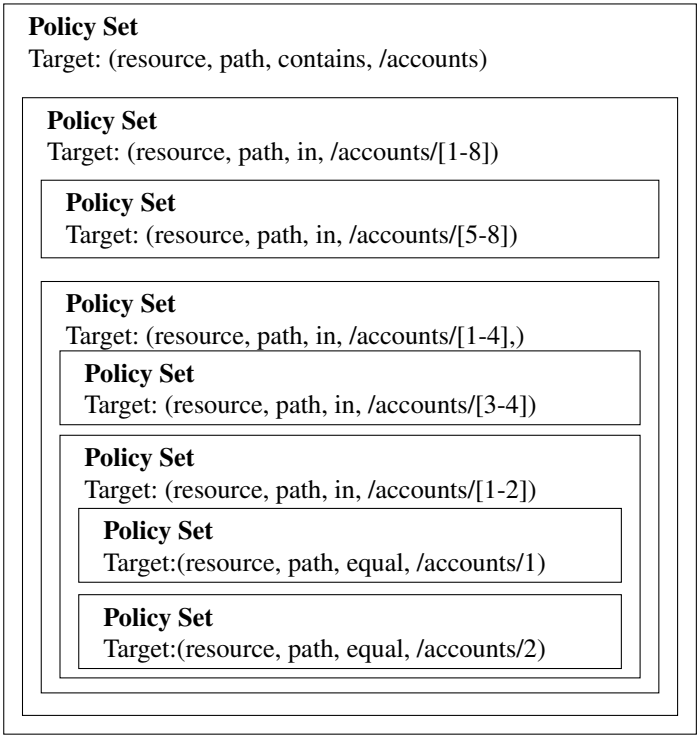

Abbildung 3.8: Gruppierte Konto-Ressourcen

Klammern verwendet, um Intervalle zu symbolisieren.

Die Verwendung zusätzlicher Bäume beschleunigt insbesondere für große Ressourcenmengen die Auswertung der Zugriffszeit deutlich. Auf der anderen Seite wird die Komplexität des Regelwerks ebenfalls deutlich erhöht, da zusätzliche Policy Sets integriert werden. Dies stellt einen wesentlichen Nachteil für die Wartbarkeit des Regelwerks dar. Das heißt, beim Einsatz von XACML für RESTful Services muss abgewogen werden zwischen Performanz und Wartbarkeit des Regelwerks. Dieses Abwiegen könnte durch den Einsatz effizienterer Datenstrukturen vereinfacht werden. Allerdings ist dies aufgrund des generischen Designs von XACML nicht möglich, da in einem XACML-Regelwerk an jeder Stelle beliebige Bedingungen formuliert werden können.

In Abhängigkeit der Rechenkapazitäten des Geräts, welches die Zugriffsanfragen verarbeitet und der Anzahl an Ressourcen unterhalb einer Listenressource, kann die Baumstruktur variiert werden. Ein binärer Baum verursacht größere Pflegeaufwände als ein Baum, welcher auf breiteren Intervallen pro Knoten beruht. Experimentelle Ergebnisse (vgl. Abschnitt 3.6) haben gezeigt, dass unter bestimmten Umständen etwa 100 Knoten pro Intervall problemlos ohne merklichen Zeitverlust verarbeitet werden können. Dies bedeutet, dass anstelle eines Binärbaums auch ein Baum mit jeweils 100 Unterknoten verwendet werden kann. Für die Wartbarkeit ist dies ein deutlicher Vorteil, da eine geringere Tiefe im Baum berücksichtigt werden muss.

3.5 Analyse von XACML mit zusätzlichen Binärbäumen

Bei einer Zugriffsanfrage müssen alle Elemente mit dem selben Elternknoten sequentiell ausgewertet werden. Dies führt zu einer durchschnittlichen Anzahl an Auswertungsoperationen der Ordnung $O(n)$ mit n als der Anzahl der Elemente. Das Design von XACML lässt jegliche Art von Attributbedingung zu, so dass eine Minimierung der Anzahl der Auswertungsoperationen nur bedingt möglich ist. Ein XACML-Regelwerk ist ein Baum von Entscheidungsknoten mit beliebigen Attributbedingungen innerhalb der Knoten. Bei einer großen Anzahl von Objekten des gleichen Typs ist eine direkte Adressierung nicht möglich, wenn man den Vorgaben des Standards folgt. Eine Minimierung ist lediglich durch eine Transformation in einen (balancierten) Baum möglich. Die Transformation ist schematisch in Abbildung 3.9 dargestellt.

Jedes Policy Set verwaltet ein Intervall von Ressourcen und hat wiederum höchstens zwei untergeordnete Policy Sets, so dass ein Binärbaum aus Policy Sets entsteht. An dieser Stelle sei darauf hingewiesen, dass die Knoten eines Binärbaums üblicherweise Werte enthalten anstelle von Intervallen. Jeder Wert teilt dabei ein Intervall in zwei gleich große Teile. Dieses Verhalten kann jedoch aufgrund des Designs von XACML nicht exakt abgebildet werden. Durch die Verwendung des Binärbaums aus Policy Sets, kann die Identifikation der angefragten Ressourcen erheblich schneller ausgeführt werden, da Suchoperationen in Binärbäumen mit Ordnung $O(log_2(n))$ durchgeführt werden, während eine sequentielle Verarbeitung lediglich mit Ordnung $O(n)$ durchgeführt werden kann [28].

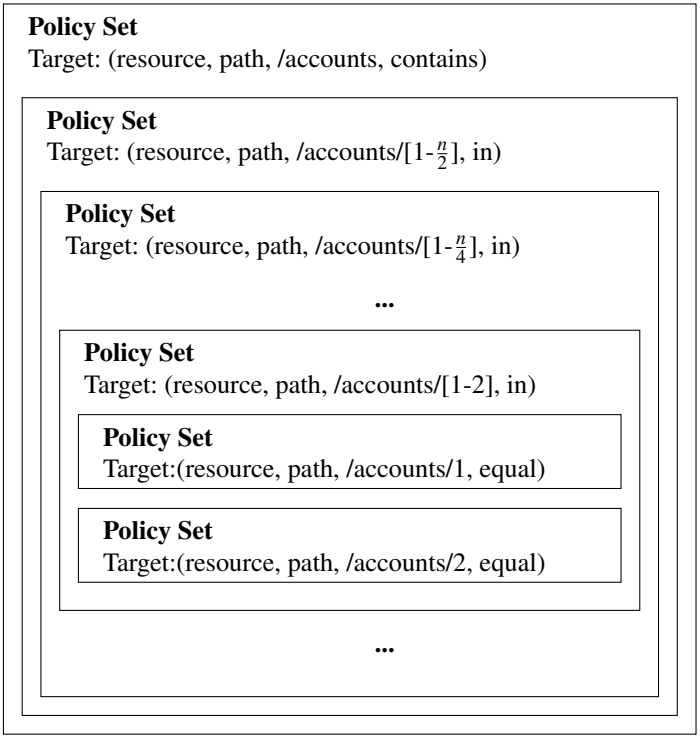

Abbildung 3.9: Binärbaum aus XACML Policy Sets

Die Antwortzeiten für Zugriffsanfragen steigen auf ein nicht mehr akzeptables Niveau, wenn die Anzahl der Ressourcen wächst und keine Minimierungen vorgenommen werden. Dabei können selbst bei einer relativ kleinen Menge von Ressourcen bereits lange Antwortzeiten erreicht werden. Im vorherigen Abschnitt wurde eine Transformation gezeigt, die einen Baum aus Policy Sets verwendet und einen effizienteren Weg darstellt, um Zugriffskontrolle für RESTful Services mit XACML zu implementieren [66].

Auf der einen Seite hilft die Optimierung dabei die Performanz zu verbessern, um damit die Antwortzeiten für Zugriffsanfragen deutlich zu reduzieren. Auf der anderen Seite jedoch müssen zusätzliche Policy Sets hinzugefügt werden, um den Baum zu erstellen. Dies führt zu einem erhöhten Speicherbedarf. Anstelle von $n+1$ Policies oder Policy Sets (das übergeordnete Policy Set zuzüglich jeweils einer Policy oder eines Policy Sets für jede Ressource) hat der hier verwendete Binärbaum

$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + ... + 1 = \sum_{i=0}^{\log_2(n)} \frac{n}{2^i} \tag{3.8}$$

Policies und Policy Sets. Gleichung (3.9) zeigt, dass sich die Anzahl an Policies und Policy Sets für eine große Anzahl an Ressourcen in etwa verdoppelt.

$$\lim_{n\to\infty} \sum_{i=0}^{\log_2(n)} \frac{n}{2^i} = 2n \tag{3.9}$$

Dies bedeutet, dass der Preis die Optimierung der Antwortzeit einer Zugriffsanfrage von $O(n)$ zu $O(log_2(n))$ die Verdopplung des Speicherbedarfs ist.

Für konkrete Zahlenwerte kann erneut das Beispiel der Bankanwendung betrachtet werden mit dem folgenden Adressierungsschema: http://example.org/accounts/{aId}/transfers/{tId}/details/{dId}.

Existieren 10 Konten in der Anwendung mit jeweils 10 Überweisungen, welche wiederum jeweils 10 Detaileinträge als untergeordnete Ressourcen besitzen, so ergeben sich insgesamt 1110 Ressourcen (10 Konten, 100 Überweisungen, 1000 Detaileinträge).

Da die sequentielle Auswertung aller Ressourcen sehr zeitaufwendig ist, kann das Regelwerk für die Zugriffsentscheidungen nach Abbildung 3.9 strukturiert und somit eine schnelle Identifikation der auszuwertenden Policies gewährleistet werden. Nach Gleichung (3.9) kann gesagt werden, dass sich die benötigte Anzahl an Policy Sets durch diese Optimierung verdoppelt zu 2220 Policy Sets nur für die schnelle Identifikation der angefragten Ressource. Die Policy für eine Ressource sollte sich mindestens auf eine Zugriffsmethode beziehen. Der Policy werden Rules untergeordnet, welche den Zugriff auf die Ressource kontrollieren. In diesem Beispiel werden 2 Zugriffsmethoden und 5 Rules für den Zugriff auf jede Ressource angenommen. Dies bedeutet, dass das Regelwerk 2220 Policy Sets, $2220 * 2 = 4440$ Policies und $4440 * 5 = 22220$ Rules beinhaltet. Da jedes dieser Elemente ein Target beinhaltet, existieren insgesamt $2220 + 4440 + 22220 = 28880$ Targets in einem Binärbaum. Testergebnisse zeigen, dass das Regelwerk in einer Datei mit einer Größe von 22 MB gespeichert werden kann. Im effektiven Einsatz mit der XACML-Implementierung Balana [16] resultiert für dieses Regelwerk eine Arbeitsspeicherbelegung von 130 MB.

Wird die Bezeichnung $m(r)$ verwendet als durchschnittliche Anzahl an deklarierten Zugriffsmethoden pro Ressource r und die durch-

schnittliche Anzahl an Berechtigungen, welche für eine Methode zur Zugriffskontrolle erstellt werden als $p(r)$ geschrieben, so lässt sich die Funktion zur Bestimmung der Antwortzeit t einer Zugriffsanfrage ausdrücken als:

$$t(r,n) = c * log_2(n) * m(r) * p(r); c = const. \tag{3.10}$$

Hierbei ist n die Anzahl der Ressourcen innerhalb des Regelwerks. Es wurde bereits erwähnt, dass die Optimierung der Antwortzeit einhergeht mit einem erhöhten Speicherverbrauch. Darüber hinaus existieren jedoch noch weitere Nachteile bei der Verwendung von Bäumen. Beim Hinzufügen und Entfernen von Ressourcen müssen neue Policies und Policy Sets hinzugefügt und entfernt werden. Um die Optimierung der Antwortzeit auch langfristig gewährleisten zu können, muss der Binärbaum in solch einem Fall neu balanciert werden. Die Balancierung verursacht hohe zusätzliche Kosten in Form von Rechenleistung [28]. Kostenintensive Operationen sollten für häufig wiederkehrende Aufgaben soweit wie möglich vermieden werden. Ein weiterer Nachteil bei der Verwendung eines Baums ist die deutliche Erhöhung der Administrationsaufwände für den Fall, dass das Regelwerk manuell geändert und geprüft wird. Diese Erhöhung liegt in der Verdopplung der Anzahl der Targets begründet.

3.6 Experimentelle Ergebnisse

Zur Evaluierung der gefundenen Optimierungsansätze wurde eine Reihe von Tests mit verschiedenen Regelwerken durchgeführt. Eine erste Menge von Regelwerken enthielt 10 Bedingungen an die URI von

10 Ressourcen. Für jede HTTP-Methode (GET, POST, PUT, DELETE) wurde eine einzelne Policy mit jeweils einer Rule definiert, so dass insgesamt 40 Rules existierten. Ein zweites und drittes Testset beinhaltete weitere 100 beziehungsweise weitere 1000 Ressourcen, so dass insgesamt 440 beziehungsweise 4440 Rules existierten. Für jedes Regelwerk wurden vier funktionell identische, aber strukturell unterschiedliche XACML-Regelwerke untersucht:

- Eine nicht optimiertes Regelwerk mit einer flachen Struktur, welche dem Muster aus Abbildung 3.1 folgt. Die Zugriffsbedingungen werden in diesem Regelwerk also in genau einer übergeordneten Policy mit einer großen Anzahl untergeordneter Rules abgebildet. Die Targets der Rules enthalten komplexe Bedingungen.

- Eine gemäß Abbildung 3.6 normalisierte Policy. Im Vergleich zur ersten Messreihe, sind die Rules nach ihrem Effect sortiert und es kommt der Combining Algorithm *FirstAppicable* zum Einsatz.

- Eine strukturoptimierte Form mit den beiden in Abschnitt 3.3 zuerst beschriebenen Optimierungen. Das in dieser Messreihe verwendete Regelwerk wird nicht mehr durch eine Vielzahl an Rules auf der gleichen Ebene abgebildet, sondern es werden Policy Sets eingesetzt, so dass eine Baumstruktur im Regelwerk entsteht.

- Eine optimierte Variante mit allen in Abschnitten 3.3 beschriebenen Optimierungen.

Als XACML-Mechanismus wurde Balana [16] eingesetzt. Die Messungen wurden auf einem Dual Core System (Intel i7-3250M, 2,90GHz) mit 8GB reserviertem Arbeitsspeicher durchgeführt. Jeder Testdurchlauf wurde mindestens 20-mal wiederholt.

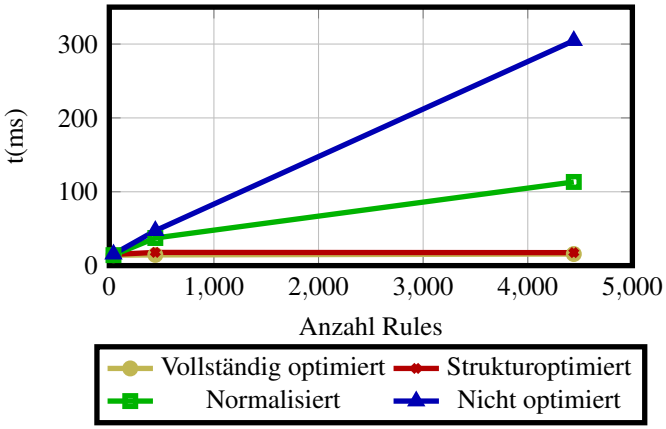

Abbildung 3.10: Durchschnittliche Antwortzeiten

Abbildung 3.10 zeigt die durchschnittlichen Antwortzeiten. Bei kleinen Ressourcenmengen unterscheiden sich die Abweichungen der Antwortzeiten nur unwesentlich. Mit steigender Ressourcenzahl und damit auch steigender Komplexität im Regelwerk kommen die Auswirkungen der Optimierungen jedoch deutlicher zum Vorschein. Während die durchschnittliche Antwortzeiten für das vollständig optimierte Regelwerk bei nahezu konstant 15ms bleibt, wächst die durchschnittliche Antwortzeit für das nicht optimierte Regelwerk auf 304ms.

Der wesentliche Teil der Optimierung kommt durch die Strukturanpassungen. Durch die enthaltenen Optimierungen wird eine erste Baumstruktur eingeführt und es werden Redundanzen eliminiert. Diese Optimierungen führen dazu, dass bei der Auswertung einer Zugriffsanfrage sehr schnell eine Entscheidung berechnet werden kann. Da der Anstieg der durchschnittlichen Antwortzeiten bei steigender Anzahl von Ressourcen nur sehr klein ist und gleichzeitig ein Initialisierungsaufwand beim Programmaufruf existiert, der groß ist im Vergleich zum Anstieg, erscheint die Entwicklung der Antwortzeiten für eine steigende Anzahl von Ressourcen konstant.

Normalisierung hat nur einen spürbaren Einfluss bei großen Mengen von Ressourcen und einer damit verbundenen Komplexität im Regelwerk. Zudem gibt es starke Schwankungen in der Laufzeit mit Abweichungen von bis zu 200% vom durchschnittlichen Wert. Die übrigen Varianten haben eine Abweichung von etwa 25% von der durchschnittlichen Antwortzeit.

Einfluss von gruppierten Ressourcen Nachdem die im vorherigen Abschnitt die Auswirkungen der in Kapitel 3.3 beschriebenen Optimierungen gemessen wurden, werden in diesem Abschnitt die Auswirkungen des Designs von Regelwerken für RESTful Services auf die Antwortzeiten gemessen.

Wie in den vorherigen Abschnitten erwähnt, wächst die durchschnittliche Antwortzeit bei einer großen Menge von Ressourcen, welche ein und derselben Listenressource untergeordnet sind. Wird die Anzahl zu groß, können aufgrund der sequentiellen Verarbeitung von Policy Sets auch die in Abschnitt 3.3 beschriebenen Optimierungen nicht für eine ausreichende Performanz sorgen, da eine große Anzahl

von Ressourcen im Regelwerk dem gleichen Element untergeordnet ist. Deshalb müssen in solch einem Fall mehrere Ressourcen in Intervallen zusammen gefasst werden, um eine verbesserte Performanz zu erzielen, indem eine weitere Baumstruktur eingeführt wird.

Um den Einfluss dieser Gruppierung in Intervalle von Ressourcen zu testen, wurden Regelwerke mit 10, 100, 1.000 und 10.000 Ressourcen erstellt. Dann wurden die unterschiedlichen Antwortzeiten für Zugriffsanfragen gemessen. Dabei wurde zunächst auf Gruppierung von Ressourcen verzichtet. Das bedeutet, dass etwa alle Ressourcen, deren Pfade dem Template */accounts/123/transfers/{transferid}* entsprichen, unter der gleichen Ressource mit dem Pfad */accounts/123/transfers* im Regelwerk untergeordnet sind. Danach wurden die Ressourcen gruppiert, so dass zusätzliche Baumstrukturen im Regelwerk entstehen. Hierdurch wird die maximale Anzahl an Kinderknoten pro Knoten reduziert und zusätzliche Tiefe im Baum eingeführt. Bei der Gruppierung wurde ein 10er Baum verwendet, was bedeutet, dass jedes Policy Set wiederum 10 untergeordnete Policy Sets verwalten kann. Die zusätzlichen Bäume haben also eine Tiefe von 1 (100 Ressourcen), 2 (1.000 Ressourcen) oder 3 (10.000 Ressourcen). Hierdurch verändert sich beispielsweise das Regelwerk für 100 Ressourcen derart, dass nicht mehr ein Element 100 untergeordnete Elemente besitzt, sondern dass dieses Element nur noch 10 untergeordnete Elemente besitzt, welche aber jeweils wiederum selbst 10 untergeordnete Elemente besitzen.

Abbildung 3.11 zeigt den Einfluss von Gruppierung in Abhängigkeit einer wachsender Zahl von Ressourcen. Ohne die Gruppierung von Ressourcen ist zu erwarten, dass die durchschnittliche Antwortzeit linear mit der steigenden Anzahl an Ressourcen wächst. Wenn Gruppierung von Ressourcen eingesetzt wird, ist zu erwarten, dass die

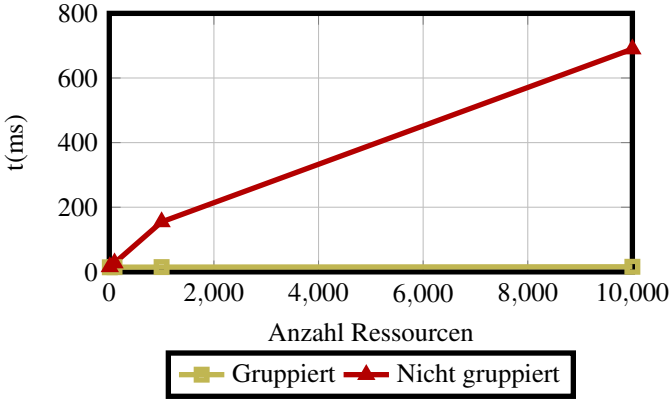

Abbildung 3.11: Einfluss der Gruppierung bei wachsender Anzahl von Ressourcen

Antwortzeit lediglich in logarithmischer Ordnung wächst. Während das lineare Wachstum bei den gemessenen Werten als schwache Korrelation interpretiert werden kann, ist das logarithmische Verhalten nur sehr bedingt erkennbar, da bei der Verarbeitung der Zugriffsanfrage ebenfalls ein Initialisierungsaufwand anfällt, welcher groß gegenüber der Zunahme der Antwortzeiten ist.

Im ersten Fall wächst dabei die Antwortzeit einer Zugriffsanfrage von 14*ms* bei 10 Ressourcen auf 689*ms* bei 10.000 Ressourcen. Im zweiten Fall wächst die Antwortzeit lediglich um 2*ms* und ist deshalb klein im Vergleich zu einem Initialisierungsaufwand von etwa 14*ms*, der durch die XACML Implementierung verursacht wird. Die Ant-

wortzeit einer Zugriffsanfrage steigt von 14.1*ms* bei 10 Ressourcen auf 15.7 bei 10.000 Ressourcen.

Verhalten unter Last Die praktische Eignung dieses Ansatzes wurde in einem Lasttest untersucht und mehrere Messreihen erstellt [64]. Dabei wurden in neuen Regelwerken synthetische Regeln erstellt. Das erste dieser Regelwerke wurde für den Schutz von 10 Ressourcen ausgelegt. Für die 4 wichtigen HTTP-Methoden wurde jeweils eine Zugriffsregel erstellt.

Im zweiten Regelwerk wurden jeweils 10 untergeordnete Ressourcen einer Ressource hinzugefügt, resultierend in insgesamt 110 Ressourcen. Dabei ist eine Ressource einer anderen Ressource untergeordnet, wenn der Pfad der anderen Ressource ein Teil des Pfads der Ressource ist. Beispielsweise ist eine Ressource mit dem Pfad */accounts/123/transfers* der Ressource mit dem Pfad */accounts/123* untergeordnet.

Im dritten Regelwerk wurden wiederum 10 untergeordnete Ressourcen an jede Ressource angeheftet, so dass das Regelwerk 1110 Ressourcen abdeckt. So entstanden Regelwerke, welche 40, 440 und 4440 Regeln enthielten, die in einer Baumstruktur angeordnet wurden. Um Aussagen über die Skalierbarkeit machen zu können, wurden simultane Anfragen verschickt. Die Anzahl der simultanen Anfragen entsprach 10, 100 und 1000. Die Abbildungen 3.12, 3.13 und 3.14 zeigen die Testergebnisse. Es ist zu erkennen, dass die Antwortzeiten bei steigender Ressourcenanzahl nur leicht anwachsen.

Die Tests wurden auf einem Dual Core System durchgeführt und 8 GB Arbeitsspeicher reserviert für das Testen. Die XACML-Implementierung, welche für die Durchführung verwendet wurde, ist

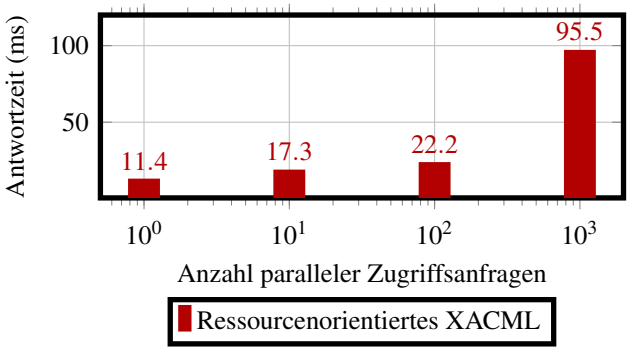

Abbildung 3.12: Durchschnittliche Antwortzeit für ein Regelwerk mit 40 Regeln

Balana [16]. Balana ist eine Open Source Implementierung, welche vom Unternehmen WSO2 bereitgestellt wird. Für jeden Test wurde ein Minimum von 10 Durchläufen ausgeführt. Bei jeder Zugriffsanfrage wurden zufällige Attributwerte verwendet. In einem ersten Test-Setup wurden die Antwortzeiten unmittelbar im Quellcode des Zugriffskontrollsystems gemessen, so dass kein Protokolloverhead entstanden ist. In einem zweiten Test-Setup konnten die Ergebnisse verifiziert werden in dem ein Framework zum automatisierten Testen genutzt wurde. Zum Einsatz kam dabei Apache JMeter [82]. In diesem Test-Setup wurde der durch JMeter entstehende HTTP-Protokolloverhead mitberücksichtigt und es wurde das REST Profile von XACML eingesetzt, welches beschreibt, wie XACML selbst als RESTful Service aufgesetzt werden kann. Das erste und das zweite Test-Setup haben sehr ähnliche

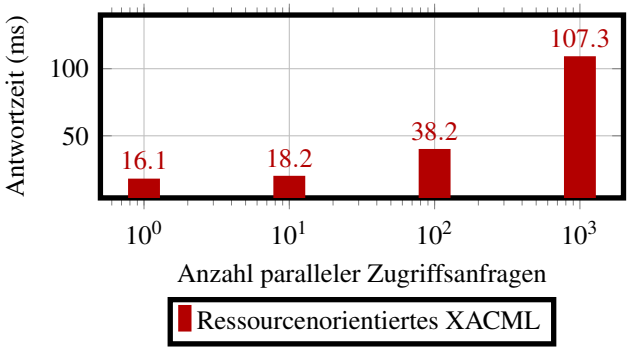

Abbildung 3.13: Durchschnittliche Antwortzeit für ein Regelwerk mit 440 Regeln

Ergebnisse geliefert. Im zweiten Test-Setup gab es durchschnittlich etwas höhere Zeiten, da hier der Verarbeitungsaufwand von HTTP mit gemessen wurde. Die Abbildungen 3.12, 3.13 und 3.14 zeigen die Testergebnisse aus dem ersten Test-Setup. Da die Ergebnisse aus dem zweiten Test-Setup jeweils nur um etwa 3-4 ms erhöht sind, wird auf die zusätzliche Darstellung verzichtet.

Für optimierte XACML-Regelwerke ist die Antwortzeit gestiegen, wenn die Anzahl an XACML-Rules sich erhöht hat und/oder wenn die Anzahl paralleler Anfragen sich erhöht hat. Eine funktionell identische, aber nicht optimierte Variante mit sequentiellem Aufbau des XACML-Regelwerks mit 4440 Regeln hat für die Auswertung einer einzigen Anfrage etwa 300ms gebraucht.

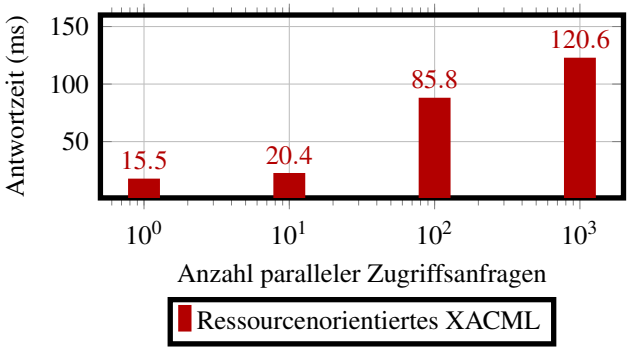

Abbildung 3.14: Durchschnittliche Antwortzeit für ein Regelwerk mit 4440 Regeln

3.7 Probleme beim Einsatz von XACML

Das Modell der attributbasierten Zugriffskontrolle kann als Generalisierung von verschiedenen etablierten Zugriffskontrollmodellen angesehen werden. Wie bereits zuvor beschrieben, ist XACML sowohl ein Mechanismus als auch eine Sprache, die zur Umsetzung von ABAC eingesetzt werden können und die sich als Standard für ABAC etabliert haben. Die gebotene Flexibilität des unterliegenden ABAC-Models eröffnet die Möglichkeit detaillierte Regelwerke zu formulieren. Allerdings birgt der Einsatz von XACML auch einige Gefahren. So ist es möglich, dass die Wartbarkeit von Regelwerken nicht mehr gegeben ist, wenn Komplexität innerhalb des Regelwerks stark gestiegen ist, weil die Anzahl der Regeln zu groß geworden ist.

Ebenfalls ist es möglich, dass starke Schwankungen bei der Verarbeitungsdauer von Zugriffsanfragen auftreten. Dies kann damit begründet werden, dass die Verarbeitungsgeschwindigkeit abhängig davon ist, wie das Regelwerk organisiert ist. Lange Antwortzeiten für Zugriffsanfragen können die Konsequenzen sein in Umgebungen, die eine sehr feingranulare Zugriffskontrolle benötigen.

Wie zuvor beschrieben kann es zu Performanzproblemen bei der Verarbeitung einer Zugriffsanfrage kommen, wenn das Regelwerk geschrieben wird ohne die Auswirkung der Struktur innerhalb des Regelwerks zu berücksichtigen. Ebenfalls wurde gezeigt, dass es beim Einsatz in RESTful Services zu einem Konflikt zwischen der Optimierung der Performanz und dem Erhalt der Wartbarkeit kommt, sobald die Komplexität im Regelwerk einen gewissen Wert übersteigt. Ein dritter, kritischer Punkt ist die steigende Restriktivität innerhalb des Regelwerks. Mit jedem zusätzlichen Knoten können einem XACML-Regelwerk nur weitere Zugriffsbedingungen hinzugefügt werden. Dies führt dazu, dass Redundanz bei der Angabe der Zugriffsbedingungen auftreten kann. Wird die ressourcenorientierte Struktur eingehalten, so können weitere Zugriffsbedingungen nicht zusammengeführt werden. Dies ist dadurch bedingt, dass einzelne Policies oder Rules nicht wiederverwendet werden können. Es existiert lediglich die Möglichkeit, komplette Policy Sets auszulagern und damit auch wiederzuverwenden.

Balance zwischen Performanz und Wartbarkeit In den vorherigen Abschnitten wurden Optimierungsansätze für die Antwortzeit von Zugriffsanfragen und die Wartbarkeit von XACML-Regelwerken beschrieben. Diese wurden anhand einer Kostenfunktion abgeleitet,

die den Aufwand für die Verarbeitung einer Zugriffsanfrage berechnet. Für den Einsatz in einer ressourcenorientierten Umgebung hat sich herausgestellt, dass die Strukturierung des Regelwerks analog zur Strukturierung der RESTful Services deutliche Gewinne bei der Verarbeitungsgeschwindigkeit und der Wartbarkeit ermöglicht. Mit einer wachsenden Anzahl von Ressourcen reichen diese Optimierungen jedoch nicht mehr aus, so dass die Gruppierung von mehreren Ressourcen in einer zusätzlichen Baumstruktur vorgenommen werden musste, um eine schnelle Verarbeitungsgeschwindigkeit zu gewährleisten. Dies wiederum hat allerdings zwei entscheidende Nachteile:

- Die Wartbarkeit des Regelwerks wird deutlich schlechter, da zusätzliche Policy Sets integriert werden, was die Komplexität des gesamten Regelwerks erhöht.
- Die Integration von neuen Ressourcen wird erheblich erschwert, da das Regelwerk ab einem bestimmten Ressourcenzuwachs neu balanciert werden muss. Geschieht dies nicht, so führt dies zu einem Ungleichgewicht bei der Gruppierung der Ressourcen, welches sich wiederum in schlechteren Antwortzeiten ausdrückt.

Die Ursache, dass zwischen Performanz und Wartbarkeit abgewogen werden muss, liegt im generischen Design von XACML begründet. Große Datenmengen sind mit XACML kaum handhabbar, da XACML auch Policy Sets oder Policies mit äußerst vielen untergeordneten Elementen sequentiell verarbeiten muss. In einer ressourcenorientierten Umgebung jedoch haben alle Ressourcen einen eindeutigen Schlüssel: die URI. Es ist möglich, XACML für die Verwendung dieses Schlüssels zu optimieren, was aber zu den beiden genannten Nachteilen führt.

Diese lassen sich nicht mit Hilfe von XACML lösen, da die Verwendung effizienterer Datenstrukturen (im Vergleich zur sequentiellen Verarbeitung von Listen) nur auf Kosten der Wartbarkeit möglich ist.

Fehleranfälligkeit Zusätzlich lässt sich bereits aus Kapitel 2.2 erahnen, dass ein in XACML geschriebenes Regelwerk sehr schnell eine enorme Größe annehmen kann. Messungen haben gezeigt, dass beispielsweise ein Regelwerk mit etwa 4500 Regeln aus etwa 260.000 Zeilen XML besteht. Bei der Verarbeitung dieser Mengen an XML-Daten durch Menschen kann es sehr leicht zu Fehlern kommen. Durch die Flexibilität und die ausführliche Syntax wird XACML schwierig und fehleranfällig [140, 15]. Da es sich zudem um ein kompositionelles Autorisierungsverfahren handelt, können sich Fehler auf eine sehr große Menge von Zugriffsversuchen auswirken. Dies bedeutet, dass eine weitgehende softwaretechnische Analyse des Regelwerks zwingend erforderlich ist [139, 91]. Doch auch mit softwaretechnischer Unterstützung bei der Wartung bleibt das Problem der schwierigen Integration neuer Regeln in das Regelwerk.

Wiederverwendbarkeit Grundsätzlich sind XACML-Regelwerke dazu ausgelegt, Zugriffsbedingungen wiederverwenden zu können. Dies führt jedoch dazu, dass die Struktur gemäß der Wiederverwendbarkeit angelegt werden muss. Eine Wiederverwendung von Policies und Rules ist nicht möglich. Lediglich Policy Sets können ausgelagert und an verschiedenen Stellen mehrfach eingebunden werden.

3.8 Stand der Forschung

Der Bedarf nach effizienter Zugriffskontrolle für Web Services wird belegt durch die Forschungsaktivitäten in dieser Richtung. Deklarative Autorisierung für RESTful Services wird in [48] behandelt. Die Autoren beschreiben einen auf XML basierenden Ansatz, der Ressourcenorientierung berücksichtigt. Die Zugriffsentscheidung wird basierend auf der URI der angefragten Ressource, der verwendeten HTTP-Methode und einer Benutzeridentität getroffen. Allerdings fundiert dieser Ansatz nicht auf attributbasierter Zugriffskontrolle. Vielmehr verwenden die Autoren sogenannte *rule*- und *perm*-Knoten. In einem *rule*-Knoten werden genau eine Ressource und die zugehörigen Zugriffsberechtigungen verknüpft. Die Zugriffsberechtigungen werden in *perm*-Knoten gespeichert, welche HTTP-Methoden abbilden. Schließlich werden in einer Benutzertabelle alle *rule*-Knoten der anfragenden User gespeichert und so die Verbindung von Ressource, HTTP-Methode und User hergestellt. Es handelt sich bei dem Ansatz demnach um eine auf REST zugeschnittene Variante von Zugriffskontrolllisten. Die Abbildung von stark heterogenen Zugriffsbedingungen ist mit einem solchen Ansatz nicht möglich.

Ein weiterer Ansatz, der sich mit Autorisierung für RESTful Services beschäftigt, wird in [22] behandelt. Die Autoren präsentieren einen Ansatz, welcher auf dem rollenbasierten Zugriffskontrollmodell aufgebaut ist und einen zentralen Security Service verwendet. Der Security Service übernimmt die Authentifizierung und Autorisierung des Clients. Der Client erhält im Falle einer erfolgreichen Autorisierung ein Zugriffstoken, welches der Web Service wiederum beim Security Service verifizieren kann. Hierzu verwaltet der Security Service Rollen und Benutzergruppen, die die Zugriffsrechte abbilden. Da auch dieser

Ansatz auf einem klassischen Zugriffskontrollmodell basiert, ist es nur bedingt möglich, heterogene Zugriffsbedingungen abzubilden.

Ein dezentraler Ansatz zur Implementierung von Zugriffskontrolle im Web of Things wird in [104] gezeigt. Dieser Ansatz verwendet Zugriffsmatrizen, um die Zugriffsberechtigungen von einzelnen Subjekten auf die Ressourcen eines Dings zu speichern. Im Web of Things kann mit jedem Ding über eine oder mehrere REST-Schnittstellen kommuniziert werden. Einzelne Benutzer werden über HTTP-Header identifiziert. Die Zugriffsmatrix findet die hinterlegten Zugriffsrechte anhand des authentifizierten Benutzers und der eindeutigen Adresse der angefragten Ressource. Innerhalb der Matrizen werden Zugriffsberechtigungen in Form eines CRUD-Flags hinterlegt. Das heißt, für jede der Operationen *Create*, *Read*, *Update*, *Delete* signalisiert ein boolscher Wert, ob die Operation ausführbar ist.

Ein Ansatz für die Verwaltung von Zugriffsrechten im Internet of Things wird in [50] vorgestellt. Die Autoren beschreiben, dass Zugriffskontrolle im Internet of Things skalierbar, wartbar, effizient und einfach verständlich sein muss. Zudem sind die Autoren der Überzeugung, dass bisherige Ansätze und Lösungen zur Umsetzung von Zugriffskontrolle nach den rollen- und attributbasierten Modellen diese Eigenschaften nicht aufweisen. Insbesondere werden die Skalierbarkeit, Nutzbarkeit und Wartbarkeit kritisiert. Die Arbeit stellt einen Ansatz vor, der auf Capabilities (Capability Based Access Control - CapBAC) fundiert. Capabilities funktionieren ähnlich wie Zugriffskontrolllisten. Die beiden Ansätze unterscheiden sich dadurch, dass Zugriffskontrolllisten dem zu schützenden Objekt/der zu schützenden Ressource zugeordnet werden, während Capabilities dem anfragenden Subjekt zugeordnet werden. Das anfragende Subjekt erhält also Nachweise, auf welche Objekte oder Ressourcen es zugreifen darf. In [50]

kann der Besitzer eines Dings signierte Capabilities an Dritte aushändigen, welche es den Dritten ermöglichen, bestimmte Operationen auf dem Ding auszuführen. Dieser Ansatz erfordert es, dass jedes Ding einem Besitzer zugeordnet wird und dieser Besitzer die Verwaltung der Zugriffsberechtigungen steuert. Es obliegt dem anfragenden Subjekt bei einem Zugriffsversuch aus der Menge aller seiner Capabilities die passende Capability auszuwählen und als Nachweis zu verwenden.

In [146] wird ein regelbasierter Ansatz für das Industrial Internet of Things (IIoT) beschrieben. Maschinen, Sensoren, Aktoren und andere Ressourcen werden durch Gateways abgeschirmt. Auf diesen Gateways werden Zugriffsanfragen durch Kontextinformationen in Form von Attributen bereichert. Diese Attribute werden in einem Context Enhancement Protocol (CEP) transportiert und beim Zugriff auf die Ressource ausgewertet. Die Berechnung der Zugriffsentscheidung erfolgt dezentral auf den Gateways. Zugriffsregeln werden als einfache boolsche Ausdrücke formuliert.

Die generellen Vorteile der Absicherung von Web Services mit ABAC werden in [152] vorgestellt. Die Autoren beschreiben, dass bekannte Zugriffskontrollmodelle wie DAC, MAC oder RBAC zu statisch und grobgranular sind, so dass sie sich nicht für den Einsatz in serviceorientierten Architekturen eignen. Der Ansatz orientiert sich architektonisch und sprachlich an XACML. Das Einsatzgebiet sind serviceorientierte Architekturen, welche das Simple Object Access Protocol (SOAP) verwenden.

Weitere, sehr ähnliche Ansätze, die ABAC für SOAP adressieren, werden in [129, 84] vorgestellt. In [129] präsentieren die Autoren ein Web Service ABAC Framework, welches XACML nutzt, um nicht autorisierte SOAP-Anfragen zu blockieren. In [84] werden SAML und XACML eingesetzt für die Sicherung von SOAP-basierten Web

Services für die Heimautomatisierung. Beide Ansätze untersuchen die grundsätzliche Eignung von attributbasierter Zugriffskontrolle am Beispiel von XACML für SOAP. Einzelne SOAP-Anfragen werden von einem XACML-Zugriffssystem autorisiert.

In [32] wird ein attributbasierter Ansatz für serviceorientierte Architekturen im Allgemeinen beschrieben. Auch bei diesem Ansatz wird XACML eingesetzt. Die Autoren beschreiben eine dreiteilige Architektur bestehend aus einem Service Management Module, einem Authentication und Token Management Module und einem Access Control Module. Das Access Control Module verwendet feste Tupel der Form <*Subject,Resource,Environment*> als Grundlage, was die Auswertung von Zugriffsanfragen erheblich vereinfacht. Benutzerinformationen werden in einer ausgezeichneten Datenbank hinterlegt und für die Autorisierung ausgewertet.

In [20] beschreiben die Autoren, dass eine Herausforderung bei der Absicherung von Web Services das Design von effektiven Zugriffskontrollmechanismen ist. Die Autoren adressieren das Problem mit einem kontextbezogenen Ansatz. RBAC wird dabei um Kontextinformationen wie etwa Zeit und Standort erweitert.

3.9 Zusammenfassung

In diesem Kapitel wurden die folgenden wissenschaftlichen Beiträge beschrieben:

- **Optimierungsstrategien für XACML-Regelwerke:** es wurden Strategien beschrieben, wie sich XACML-Regelwerke im allgemeinen effizienter gestalten lassen. Dazu wurde eine Kos-

tenfunktion aufgestellt, mit der sich die Evaluationskosten in einem XACML-Regelwerk berechnen lassen.

- **XACML-Regelwerke für RESTful Services:** Ressourcenorientierung ist eine elementare Grundlage von REST, welche sich bei der Strukturierung von Regelwerken ausnutzen lässt, um die Verarbeitungsgeschwindigkeit von Zugriffsanfragen zu verbessern. Hierzu muss das Regelwerk ressourcenorientiert gestaltet werden. Analysen und experimentelle Ergebnisse weisen die Effizienz dieser Methode nach.

- **Abwägung zwischen Effizienz und Wartbarkeit:** die Verbesserung der Verarbeitungsgeschwindigkeit geht zu Lasten der Wartbarkeit eines Regelwerks. Für große Mengen von Ressourcen werden Regelwerke zu komplex, um eine einfache Wartung durchführen zu können. Eine erste Version von RestACL löst dieses Problem, indem es Ressourcenorientierung in die Gestaltung des Regelwerks einfließen lässt.

Die Richtlinien zur Erstellung von effizienten XACML- Regelwerken für RESTful Services bestehen aus zwei wesentlichen Grundlagen. Die erste Grundlage ist die effiziente Verwendung der XACML- Deklarationssprache. Hierbei können für die effiziente Gestaltung die zuvor aus der Kostenfunktion abgeleiteten Deklarationsregeln heran gezogen werden. Werden diese bei der Gestaltung eines Regelwerks zur Absicherung von RESTful Services berücksichtigt, so entsteht eine klare Struktur innerhalb des Regelwerks, welches es ermöglicht die Wartungsaufwände zu reduzieren und gleichzeitig die Verarbeitung effizienter gestaltet. Wird die Anzahl an zu verwaltenden Ressourcen jedoch

zu groß, so sollte eine zweite Grundlage befolgt werden: die Gruppierung von Ressourcen in zusätzlichen Baumstrukturen. Hierbei wird eine höhere Verarbeitungsgeschwindigkeit erreicht. Dies geschieht jedoch auf Kosten der Wartbarkeit, welche sich durch zusätzliche Elemente innerhalb des Regelwerks wesentlich verschlechtert.

Ein Vorteil der beschriebenen Deklarationsregeln für XACML ist die Einführung von klaren Hierarchien in der Struktur von XACML-Regelwerken. Änderungen am Regelwerk sind bei kleinen Datenmengen auf einfache Weise möglich und ziehen keine langwierigen Transformationsprozesse nach sich. XACML sieht für die Manipulation von Regelwerken keinen standardisierten Weg vor. Der naheliegendste Weg ist deshalb das Hinzufügen von neuen Policy Sets, Policies und Rules, unterstützt durch einen graphischen Editor [95]. Die manuelle Veränderung komplexer Regelwerke ist jedoch sehr fehleranfällig, da die finale Zugriffsentscheidung bei XACML immer eine Komposition verschiedener Effekte ist.

Durch die Verwendung der beschriebenen Methoden treten jedoch auch Nachteile auf. Neben der Fehleranfälligkeit bei Änderungen ist die Wiederverwendung von Zugriffsbedingungen sehr schwierig. So müssen gleiche Zugriffsbedingungen für verschiedene Ressourcen mehrfach explizit angegeben werden. Grundsätzlich würde es ein XACML-Regelwerk zwar erlauben, diese Zugriffsbedingungen weiter oben im Baum zu verwenden, jedoch würde dann die Ressourcenorientierung verloren gehen. Schließlich ist die Wartbarkeit problematisch.

Kapitel 4

RestACL - Die REST Access Control Language

Im folgenden Kapitel wird die REST Access Control Language (RestACL) detailliert beschrieben. Eine erste Version von RestACL wurde in [64] beschrieben und dient als Grundlage für die im folgenden beschriebenen Arbeiten. In diesem Kapitel werden eine Regelspezifikationssprache und Zugriffsmechanismen vorgestellt, mit denen attributbasierte Zugriffskontrolle in effizienter Weise für RESTful Services ermöglicht wird. Dabei erfolgt die Abgrenzung von indexbasierten Verfahren gegenüber kompositionellen Verfahren. Die Beiträge dieses Kapitels wurden in [69] veröffentlicht. Das Kapitel beinhaltet eine Analyse von RestACL, welche in [65] publiziert wurde.

4.1 Indexbasierte Zugriffskontrollmechanismen

Gegenüber der ersten Version von RestACL unterscheidet sich der in diesem Kapitel vorgestellte Mechanismus in drei wesentlichen Punkten:

1. Ein neues Architekturkonzept dient der Integration eines Zugriffskontrollsystems mit Anwendungen, die durch RESTful Services aufgebaut sind.

2. Die Modifikationen der Syntax und der Architektur erlauben eine einfachere Wiederverwendung von Zugriffsberechtigungen.

3. Eine Implementierung des Zugriffskontrollmechanismus verwendet effizientere Algorithmen zur Identifikation von Ressourcen und zugehörigen Zugriffsregeln innerhalb von kurzer Zeit. Aufgrund der Eindeutigkeit von Ressourcenadressen können indexbasierte Verfahren angewendet werden, um die Verarbeitungsgeschwindigkeit auch bei sehr großen Mengen von Ressourcen konstant zu halten.

RestACL ist zugeschnitten auf die Anforderungen, die durch das Schutzbedürfnis der Ressourcen von RESTful Services entstehen. Das Erstellen, Bearbeiten und auch die Durchsetzung der Zugriffsberechtigungen kann zusammen mit dem Erstellen, Bearbeiten und dem Zugriff auf Ressourcen umgesetzt werden. Der beschriebene Mechanismus ermöglicht eine schnelle Entscheidungsfindung auch bei großen Datenmengen. Die Implementierung verwendet indexbasierte Auswertungsverfahren anstelle von kompositionellen Verfahren, welche

der generische XACML-Mechanismus verwenden muss. Aus diesem Grund kann der RestACL-Mechanismus eingesetzt werden, um die Entscheidungsfindung auch dann noch auf Anfragebasis durchzuführen, wenn die Anzahl an Ressourcen und Anfragen sehr groß wird. Das bedeutet, dass jede Anfrage an einen RESTful Service individuell ausgewertet werden kann.

4.2 Anforderungen

Es existieren vier wesentliche Merkmale, welche ein Zugriffskontrollsystem für REST erfüllen muss. Mit der Erfüllung dieser Merkmale wird sichergestellt, dass die Vorteile von REST nicht verloren gehen und die Zugriffskontrolle auf effiziente Art und Weise durchgeführt werden kann.

(R1) *Autorisierung auf Anfragebasis*: Zustandslose Kommunikation zwischen Client und Server ist ein wesentliches Konzept von REST. Um diesen Grundsatz zu gewährleisten, muss Autorisierung auf Anfragebasis durchgeführt werden. Dies bedeutet, dass es möglich sein muss, jede Anfrage individuell zu autorisieren. Wenn die Autorisierung nicht auf Anfragebasis durchgeführt wird, kann dies zu Zugriffsentscheidungen führen, welche nicht konform sind mit dem aktuellen Regelwerk. Dies lässt sich relativ einfach am Beispiel eines Statuswechsels einer Ressource verdeutlichen. Wechselt eine Ressource ihren Status, kann dies auch dazu führen, dass sich die Zugriffsberechtigungen ändern. Verändert eine Ressource ihren Status zwischen zwei Anfragen, kann die Zugriffsentscheidung für die zweite Anfrage nach dem Statuswechsel eine andere sein als sie es vor der ersten Anfrage war. Werden die Zugriffsentscheidungen für beide

Anfragen nicht vollständig unabhängig voneinander berechnet, kann ein unerlaubter Zugriff die Folge sein. Werden Zugriffsberechtigungen mittels Token oder innerhalb von Sessions vergeben, müssen zusätzliche Aufwände eingeplant werden, um die Validität der vergebenen Rechte nach jeder Anfrage erneut zu prüfen. Es ist daher effektiver, jede Anfrage individuell zu genehmigen oder zu verbieten.

(R2) *Häufige Änderungen des Regelwerks*: Je nach Anwendungsfeld der Applikation können häufig neue Ressourcen erzeugt und existierende Ressourcen gelöscht werden. Bei diesen Ereignissen kann es notwendig sein, eine Anpassung am Regelwerk vornehmen zu müssen. Auch bei Zustandswechseln von Ressourcen ist es denkbar, dass Änderungen bei den Zugriffsberechtigungen notwendig werden. Für das Zugriffskontrollsystem bedeutet dies, dass eine schnelle Reaktion auf diese Ereignisse möglich sein muss. Geschieht ein Zustandswechsel oder wird eine neue Ressource erzeugt und folgt unmittelbar auf dieses Ereignis eine Zugriffsanfrage, so muss das Zugriffskontrollsystem in der Lage sein, unmittelbar mit einer Entscheidung in Abhängigkeit der neuen Berechtigungen und des neuen Zustands zu antworten. Deshalb muss das Zugriffskontrollsystem eine einfache Integration und Manipulation von Zugriffsrechten ermöglichen. Systeme, die schnelle Zugriffsentscheidungen erst nach aufwendiger Transformation gewährleisten können, sind demnach nicht geeignet.

(R3) *Unterstützung von Vielfältigkeit*: Eine wachsende Anzahl von Ressourcen kann begleitet werden von einer wachsenden Heterogenität innerhalb der Menge der Zugriffsberechtigungen. Das Zugriffskontrollsystem muss deshalb in der Lage sein, eine möglichst große Bandbreite von verschiedenartigen Zugriffsberechtigungen zu unterstützen. Beispielsweise ist es denkbar, dass eine Anwendung erweitert wird, um den Benutzern neue Funktionen oder Dienste an-

zubieten. Das Zugriffskontrollsystem sollte dies unterstützen ohne große Anpassungsaufwände zu erfordern. Andernfalls wird das Zugriffskontrollsystem möglicherweise der limitierende Faktor bei der Bereitstellung neuer Funktionen und Dienste.

(R4) *Skalierbarkeit*: Da die Anzahl an Ressourcen innerhalb einer Applikation schnell auf eine sehr große Zahl wachsen kann, muss das Zugriffskontrollsystem möglichst effizient in Bezug auf Speicherverbrauch und Antwortzeiten sein. Skalierbarkeit ist ein entscheidender Vorteil in verteilten Systemen, die den Prinzipien von REST folgen. Dies bedeutet, dass auch das Zugriffskontrollsystem mit einer wachsenden Zahl von Ressourcen umgehen können muss. Andernfalls könnte das Zugriffskontrollsystem zum Flaschenhals der Applikation werden oder die gesamte Applikation wird ineffizient in Bezug auf Speicherverbrauch, so dass ein nicht mehr akzeptables Niveau erreicht wird.

4.3 Architekturkonzept

Abbildung 4.1 zeigt eine Übersicht der Komponenten, die zur Entscheidungsfindung über die Durchführbarkeit einer Anfrage an einen RESTful Service notwendig sind.

Ein standardisierter *RESTful Client* (beispielsweise ein Browser) sendet eine Ressourcenanfrage zu einem *RESTful Server*. Dabei verwendet der Client Protokolle wie beispielsweise HTTP, welche dem Client eine bestimmte Menge von Operationen anbieten, die vorgeben, wie mit der angefragten Ressource interagiert werden kann. Bevor der Server die Anfrage des Client bearbeitet, kann er den Client auffordern sich zu authentifizieren. Für den Fall, dass HTTP das eingesetzte Pro-

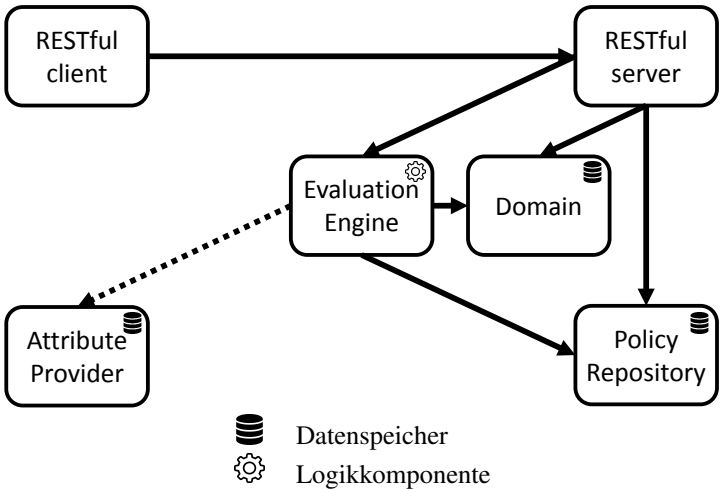

Abbildung 4.1: Architekturkonzept

tokoll ist, können hierfür bekannte und etablierte Mechanismen wie Basic Authentication oder Digest Authentication verwendet werden [72].

Nachdem der Client sich gegebenenfalls authentifiziert hat, muss eine Zugriffsentscheidung getroffen werden. Die Kommunikationsprotokolle kennen üblicherweise keine standardisierten Lösungen zur Durchführung der Zugriffskontrolllogik, da die Zugriffsentscheidung eine Entscheidung ist, welche der Server autonom treffen kann. An dieser Stelle gilt es demnach, den Zugriffskontrollmechanismus auf Seite des Servers zu integrieren. Würde der Server die Anfrage des

Clients bearbeiten bevor die Zugriffsberechtigung berechnet ist oder parallel dazu, so könnte ein nicht-autorisierter Zugriff die Konsequenz sein.

Der RESTful Server muss beim Eintreffen einer Ressourcenanfrage eine RestACL-Zugriffsanfrage formulieren. Diese Zugriffsanfrage wird von der Ressourcenanfrage abgeleitet, um verschiedene Attribute ergänzt und dann an die *Evaluation Engine* übergeben. Das Ergänzen der Zugriffsanfrage um weitere Attribute ist anwendungsabhängig. Der Server ist verantwortlich, sicherheitsrelevante Attribute auszuwählen und der Zugriffsanfrage hinzuzufügen. Denkbar ist beispielsweise, dass in manchen Anwendungen die IP-Adresse des Client ein sicherheitsrelevantes Attribut ist. In diesem Fall muss der Server die IP-Adresse mit zur Zugriffsanfrage hinzufügen. In anderen Anwendungen hingegen können andere Attribute sicherheitsrelevant sein.

Nachdem der Server die Zugriffsanfrage formuliert hat, leitet er diese an die Evaluation Engine weiter. Dort wird die Zugriffsanfrage bearbeitet und eine Zugriffsentscheidung getroffen. Bevor die Evaluation Engine die Zugriffsentscheidung berechnet, können noch externe Attributquellen hinzugezogen werden, um weitere Attribute der Zugriffsanfrage hinzuzufügen. Beispielsweise ist es denkbar, dass der Server eine Benutzerkennung als Attribut der Zugriffsanfrage beifügt. Die Evaluation Engine könnte dann beispielsweise einen Identity Provider hinzuziehen und verschiedene Attribute des Benutzers abfragen.

Nachdem mögliche *Attribute Provider* zusätzliche Attribute bereitgestellt haben, kann die Evaluation Engine mit der Berechnung der Zugriffsentscheidung beginnen. Die Evaluation Engine trifft die Zugriffsentscheidung basierend auf den in der Zugriffsanfrage enthaltenen Attributen, der Ressourcenstruktur (gemäß der Beschreibung in der *Domain*) und den in der Domain referenzierten Policies. Die

Domain stellt eine Menge an Verweisen von einer Ressource auf die Policies dar, welche für den Zugriff auf die jeweilige Ressource ausgewertet werden müssen. Die referenzierten Policies werden aus einem *Policy Repository* geladen und für die Entscheidungsfindung ausgewertet.

Danach wird die Zugriffsentscheidung an den Server zurückgeliefert. Der Server ist dann verantwortlich diese Zugriffsentscheidung durchzusetzen. Das bedeutet, er kann die Ressourcenanfrage des Clients weiter verarbeiten, wenn der Zugriff gewährt wurde oder die Anfrage des Clients mit einer entsprechenden Fehlermeldung zurückweisen, wenn der Zugriff verweigert wurde. Beispielsweise kann beim Einsatz von HTTP der Server mit einem *HTTP 403 Forbidden* antworten für den Fall, dass die Zugriffsanfrage abgelehnt wurde. Für den Fall, dass die Zugriffsentscheidung positiv ausfällt, kann der Server damit beginnen, die Anfrage des Client zu verarbeiten. Im Falle von HTTP als zugrundeliegendem Protokoll hängt die Antwort des Servers dann unter anderem von der Zugriffsmethode des Clients ab (beispielsweise *HTTP 200 Ok* für eine lesende Anfrage oder *HTTP 201 Created* für eine Anfrage zum Erstellen einer Ressource).

Neben der Schnittstelle zur direkten Kommunikation mit der Evaluation Engine existieren zwei weitere Schnittstellen zwischen dem Server und den Zugriffskontrollkomponenten. Der Server hat die Möglichkeit Einträge in der Domain zu erstellen, zu verändern oder zu löschen. Wird eine Ressource erstellt oder gelöscht, so muss auch der entsprechende Eintrag in der Domain erstellt beziehungsweise gelöscht werden. Zudem können über diese Schnittstelle die anzuwendenden Policies festgelegt werden. Die dritte Schnittstelle dient der Verwaltung der Policies. Über diese Schnittstelle können neue Policies erstellt werden, existierende verändert oder gelöscht werden.

Natürlich müssen auch die Schnittstellen zum Zugriff auf die Domain und das Policy Repository geschützt werden. Wie die Regeln zum Schutz dieser Schnittstellen aussehen, hängt von der Anwendung ab. Beispielsweise ist es denkbar, dass lediglich ein Administrator die Schnittstellen nutzen darf. In Kapitel 6.2 wird jedoch auch ein Anwendungsbeispiel beschrieben, bei dem Benutzer Ressourcen anlegen können und selbst die Zugriffsrechte für diese Ressourcen von außen festlegen können. Dazu werden eine Reihe von Standard-Policies angewendet, welche beim Erstellen einer Ressource verlinkt werden.

Nachrichtensequenz Abbildung 4.2 zeigt ein vereinfachtes Sequenzdiagramm für die Verarbeitung einer Anfrage an einen RESTful Server. Ein RESTful Client sendet eine Anfrage an den Server, welcher wiederum eine Zugriffsanfrage aus der ursprünglichen Anfrage ableitet. Diese Zugriffsanfrage kann mit verschiedenen Attributen, wie etwa der aktuellen Uhrzeit, versehen werden. Neben der angefragten Ressource (identifiziert über die URI) muss die Zugriffsanfrage die Zugriffsmethode beinhalten (beispielsweise HTTP GET). Weitere Attribute sind optional und müssen dem Kontext der Anwendung entsprechend gewählt werden. Beispielsweise können zusätzliche Attribute der Authentifikationsprozedur entnommen werden, sie können Teil der Ressource selbst, Teil der Metadaten der Ressourcen oder Anwendungsdaten sein.

Die Formulierung der Zugriffsanfrage kann entweder durch die Anwendung (die Serverimplementierung) selbst geschehen oder als Teil eines generischen Frameworks. Wird eine fest definierte Menge von Attributen verwendet, um die Zugriffsentscheidung zu bestimmten, so können diese Attribute auf eine automatisierte Art und Weise

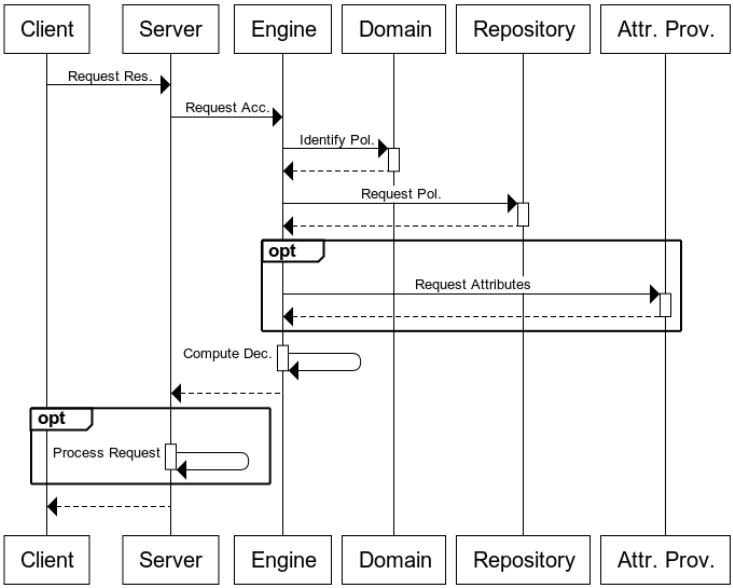

Abbildung 4.2: RestACL Nachrichtensequenz

ausgewählt und der Zugriffsanfrage hinzugefügt werden. Details der Automatisierung hängen von der jeweiligen Anwendung und den benötigten Attributen ab. Der Server schickt die Zugriffsanfrage dann zur Evaluation Engine und wartet mit der Verarbeitung der Anfrage bis er eine Zugriffsentscheidung erhält, bevor er mit der eigentlichen Verarbeitung der Anfrage fortfahren kann. Im ersten Teil der Auswer-

tung identifiziert die Evaluation Engine die auszuwertenden Policies basierend auf den Angaben innerhalb der Domain. Das Resultat ist eine Liste von Identifiern von Policies. Außerdem werden die zugehörigen Policies aus dem Policy Repository geladen. Im zweiten Teil der Auswertung kann die Evaluation Engine optional zusätzliche Attribute bei entsprechenden Providern erfragen. Im dritten Teil der Auswertung berechnet die Evaluation Engine schließlich die Zugriffsentscheidung. Das Resultat wird zurück an den Server geliefert, welcher nun die Entscheidung durchsetzen muss, indem er die Anfrage entweder bearbeitet oder zurückweist.

4.4 Definitionen

Bevor die Details der Sprache diskutiert werden, soll eine Menge von Definitionen aus [62] abgrenzen, welches Vokabular RestACL verwendet.

Definition (Attribut): Ein Attribut ist ein Tripel $a := (c, d, v)$ bestehend aus einer Kategorie c, einem Designator d und einem Wert v. Der Wert hat einen Datentypen wie etwa Integer oder Zeichenkette. Die Menge A sei die Menge aller möglichen Attribute.

Ein Beispiel für ein Attribut ist $(Subjekt, Name, Marc)$. Die Kategorie (Subjekt) dient dazu ein oder mehrere Attribute einer Entität zuzuordnen. Die Kategorie kann also zum Gruppieren von Attributen verwendet werden. Etwa kann eine Zugriffsbedingung vom Namen des anfragenden Subjekts und dem Namen der angefragten Ressource abhängen. Um eindeutig zwischen den beiden Namen unterscheiden zu können, wird die Kategorie genutzt. Die Kategorie dient also dazu,

ein Attribut einer bestimmten Entität zuzuordnen. Eine Entität besitzt immer nur Attribute der gleichen Kategorie. Der Name ist in diesem Beispiel der Designator, der zusammen mit der Kategorie das Attribut identifiziert.

Definition (Entität): Eine Entität $e := (\{a_1, ..., a_n\})$ ist die Menge aller Attribute $\{a_1, ..., a_n\}$ mit $n \in \mathbb{N}, a_i \in A$ welche zur gleichen Kategorie gehören.

Beispielsweise könnte ein Regelwerk fordern, dass *der Name des anfragenden Subjekts gleich dem Wert Marc sein muss*. Ein Attribut $a_{marc} = (subject, name, marc)$ kann dann innerhalb einer Attributbedingung verwendet werden um zu überprüfen, ob ein Attribut a_x aus der Menge der Attribute einer Entität eine Attributbedingung $ac(equal, a_x, a_{marc})$ erfüllt.

Definition (Attributbedingung): Eine Attributbedingung ist eine Funktion $ac : F \times A^n \rightarrow \{true, false\}$ mit F als Menge von vergleichenden Funktionen. Eine Attributbedingung ist erfüllt, wenn die Funktion den Wert $true$ zurückliefert.

Ein Regelwerk kann als logische Konkatenation von Attributbedingungen interpretiert werden. Etwa kann ein Regelwerk eine logische Konjunktion zwischen zwei Attributbedingungen beschreiben: *wenn die Attributbedingungen ac_1 UND ac_2 erfüllt sind, wird der Zugriff gewährt*. Während der Auswertung einer Zugriffsanfrage wird eine Menge von Attributen gegen die Attributbedingungen verglichen.

Definition (Policy): Eine Policy ist eine Funktion $p : \mathcal{P}(AC) \times \mathbb{N} \rightarrow E$ aus der Potenzmenge der Attributbedingungen $\mathcal{P}(AC)$ in die Menge der Effekte E mit $E := \{Permit, Deny, Undetermined\}$. Poli-

cies sind zudem mit einer Priorität $p \in \mathbb{N}$ versehen, welche aus den natürlichen Zahlen stammt.

RestACL ist ein ABAC-Mechanismus, welcher die Zugriffslogik in Policies hinterlegt. Diese werden gemäß Abbildung 4.3 in einem Policy Repository gespeichert. Die Identifikation der auszuwertenden Policies wird mit Hilfe einer Domain vorgenommen. In der Domain werden Ressourcen anhand ihrer Adresse beziehungsweise anhand des Pfads als Teil der Adresse identifiziert. Jeder Ressource können sogenannte Access-Elemente zugeordnet werden, welche den Zugriff auf die Ressource steuern und welche eine direkte Verknüpfung zwischen Policy und Ressource bilden.

Definition (Ressource): Eine Ressource r ist das zu schützende Objekt. Sie ist ein Tripel $r := (add, \{ae_1, ..., ae_n\}, \{r'_1, ..., r'_m\})$ bestehend aus einer Adresse add, einer Menge von Access-Elementen $\{ae_1, ..., ae_n\}$ und einer Menge von r untergeordneten Ressourcen $\{r'_1, ..., r'_m\}$.

Wird auf eine Ressource zugegriffen, so sind die Ressourcenadresse und die Zugriffsmethode zwingende Bestandteile der Zugriffsanfrage. Während die Ressourcenadresse genutzt wird, um die Ressource innerhalb der Domain zu identifizieren, wird die Zugriffsmethode genutzt, um die anzuwendenden Policies aus den zugehörigen Access-Elementen zu identifizieren.

Definition (Access-Element): Ein Access-Element $ae : \mathcal{P}(M) \rightarrow \mathcal{P}(P)$ ordnet eine oder mehrere Zugriffsmethoden einer oder mehrerer Policies zu. Es handelt sich dabei also um eine Abbildung aus der Potenzmenge der Methoden $\mathcal{P}(M)$ in die Potenzmenge der Policies

$\mathcal{P}(P)$ mit M als Menge der Zugriffsmethoden und P als Menge der Policies.

Betrachtet man das Access-Element als einen Zwischenschritt bei der Policy-Identifikation, so kann die Domain verstanden werden als Abbildung von den Ressourcen in die Potenzmenge der Policies.

Definition (Domain): Eine Domain ist eine Funktion $d : R \rightarrow \mathcal{P}(P)$ von der Menge der Ressourcen R in die Potenzmenge der Policies $\mathcal{P}(P)$.

Die Evaluation Engine berechnet Zugriffsentscheidungen basierend auf Zugriffsanfragen, welche von Anfragen auf Ressourcen abgeleitet werden. Attribute, welche für die Entscheidungsfindung benötigt werden, können von mehreren Attribute Providern geliefert werden. Dabei können verschiedene Arten von Attribut Providern zum Einsatz kommen. Ein Attribute Provider kann beispielsweise ein Identity Provider sein, der basierend auf einem eindeutigen Identifier Attribute zu einem bestimmten Subjekt liefert. Ein anderer Attribute Provider kann aktuelle Kontextdaten wie die Uhrzeit oder das Datum liefern. Sobald alle bekannten Attribute für die Berechnung der Zugriffsentscheidung vorliegen, kann die Evaluation Engine anhand zweier Funktionen die Entscheidung berechnen.

$$d : R \rightarrow \mathcal{P}(P) \tag{4.1}$$

$$f : \mathcal{P}(A) \times \mathcal{P}(P) \rightarrow E \tag{4.2}$$

d ist die Domain-Funktion und wird zur Identifikation der auszuwertenden Policies verwendet. Als Eingangsparameter dient die

Ressourcenadresse. f berechnet die Zugriffsentscheidung basierend auf einer Menge von Attributen und einer Menge von Policies. Policies haben Prioritäten und die Policy, die gleichzeitig anwendbar ist und die höchste Priorität hat, bestimmt die Zugriffsentscheidung. Die Summe aus Domain und Policies ist demnach das Regelwerk.

4.5 Trennung von Domain und Policy

Aus den Funktionen (4.1) und (4.2) ist bereits ersichtlich, dass die Verarbeitung von Zugriffsanfragen in zwei Teilfunktionen unterteilt ist: die Identifikation von Policies in der Domain und die Auswertung der Policies gegen Attribute der Zugriffsanfrage. Abbildung 4.3 visualisiert den logischen Zusammenhang zwischen Domain und Policy Repository, so wie er von der Evaluation Engine genutzt wird.

Trifft eine Zugriffsanfrage bei der Evaluation Engine ein, so wird zunächst innerhalb der Domain die Ressource identifiziert, die angefragt wurde. Die Ressourcenadresse dient dabei als eindeutiger Schlüssel, der zur Identifikation genutzt wird. Auf jede Ressource kann mit einer Teilmenge der Methoden der einheitlichen Schnittstelle eines RESTful Service zugegriffen werden. Von der Methode wird schließlich auf einen Stapel von Policies verwiesen, welcher ausgewertet werden muss. Dies bedeutet, dass eine Zugriffsanfrage immer Ressourcenadresse und Methode beinhalten muss. Dies ist bei einer Anfrage an einen RESTful Service immer gegeben.

Wird beispielsweise ein Zugriff auf *Resource A* mit *Methode 1* angefragt, so nutzt die Evaluation Engine die Adresse von *Resource A* als eindeutigen Schlüssel, um an einen Index zu gelangen, an dem die Zuordnung von Methoden und Policies für diese Ressource

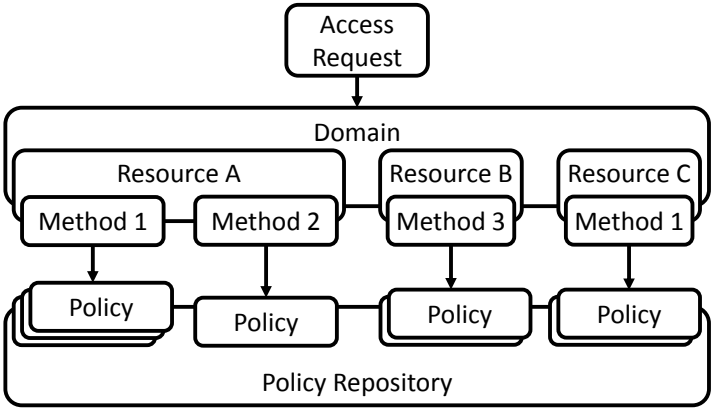

Abbildung 4.3: Trennung von Domain und Policy

hinterlegt ist. Über den Eintrag zu *Methode 1* gelangt die Evaluation Engine an die Identifier einer oder mehrerer Policies. Der Stapel von auszuwertenden Policies wird aus dem Policy Repository geladen (in Abbildung 4.3 über Pfeile als logische Verknüpfung angedeutet). Es ist beispielsweise implementierungsabhängig denkbar, dass die Policies aus einer Datei geladen werden. Andere Implementierungen könnten die Policies wiederum im Hauptspeicher verwalten und wieder andere Implementierungen könnten die Dateien auf einem File-Server gespeichert haben.

Dieser Ansatz erlaubt es, die Antwortzeiten von der Anzahl an Ressourcen und Policies zu entkoppeln und auf ein Minimum zu reduzieren. Hierdurch können große Mengen von Ressourcen und

Policies gespeichert werden, ohne die Performanz bei der Verarbeitung zu beeinflussen.

Die Trennung zwischen Domain und Policy, also die Trennung zwischen Anwendungsstruktur und Zugriffsbedingungen, erlaubt eine schnelle Identifikation der auszuwertenden Policies und damit eine insgesamt schnelle Auswertung der Anfrage, da zu jeder Anfrage nur eine sehr kleine Teilmenge der Policies ausgewertet werden muss. Durch die Verwendung eines eindeutigen Schlüssels als Verweis auf eine Policy wird es möglich effiziente, indexbasierte Datenstrukturen für die Auswertung zu verwenden. Beispielsweise können Hash-Tables verwendet werden, welche in der Praxis nahezu konstante Zugriffszeiten bieten und welche unabhängig von der Menge an Einträgen sind, die sie verwalten. Hash-Tables speichern Schlüssel-Wert-Paare und eignen sich deshalb hervorragend, um den eindeutigen Schlüssel zusammen mit den Verweisen auf die tatsächlichen Policies zu speichern. Dank dieser indexbasierten Datenstruktur wird die Verarbeitungszeit von Zugriffsanfragen entkoppelt von der Komplexität des Regelwerks. Dies ist ein erheblicher Vorteil im Vergleich zu kompositionellen Verfahren. Gleichzeitig ist die Verwendung von indexbasierten Verfahren allerdings nur möglich, wenn eindeutige Schlüssel als Index genutzt werden können.

Gleichzeitig hilft die Trennung dabei Wartungsaufwände zu reduzieren. Beispielsweise ist es denkbar, dass für viele Ressourcen die gleichen Zugriffsbedingungen gelten. Diese brauchen lediglich einmal in Form einer Policy formuliert werden und im Policy Repository abgelegt werden. Bei Änderungen an den Zugriffsbedingungen müssen lediglich Änderungen an einer Policy vorgenommen werden. Werden die Bedingungen dagegen beispielsweise in XACML formuliert, muss die gesamte Komposition von Attributbedingungen geändert wer-

den, welches potentielle Seiteneffekte hat oder gegebenenfalls müssen Bedingungen mehrfach geändert werden.

4.6 Abstrakte Syntax

In diesem Abschnitt werden die abstrakte Syntax und die Semantik von RestACL beschrieben. Die Sprache basiert auf drei wesentlichen Datenstrukturen. Die Evaluation Engine nutzt die in der Domain dokumentierte Ressourcenstruktur zur Identifikation der auszuwertenden Policies. Die erste Datenstruktur bildet deshalb die Domain und die darin enthaltenen Ressourcen ab. Die zweite Datenstruktur bildet die Policies ab, welche Anwendung bei der Entscheidungsfindung finden. Und schließlich wird die dritte Datenstruktur verwendet, um Zugriffsanfragen und Zugriffsentscheidungen zu formulieren.

Abbildung 4.4 zeigt die abstrakte Syntax einer RestACL-Domain. Eine Domain deklariert den *host*, für den das System verantwortlich ist, in URI-Notation. Die Domain kann zudem eine oder mehrere *Resources* enthalten. Resources verweisen auf untergeordnete Resources und bilden somit eine verschachtelte Struktur. Die Syntax dient dem Abbilden von Teil-Ganzes-Hierarchien, ähnlich wie es das Composite Strukturmuster tut [47]. Resources verweisen zudem auf *Access*-Elemente und *ParameterizedAccess*-Elemente. Access-Elemente definieren eine oder mehrere *methods* und verknüpfen Resources mit *Policies*. Policies sind in einem extra Diagramm abgebildet, um die Lesbarkeit zu erhöhen. ParameterizedAccess-Elemente können verwendet werden, um Untermengen von Ressourcen zu adressieren. Deshalb ist hier die Angabe von URI-Parametern (Query Strings) möglich, die Selektionen auf Listenressourcen durchführen. Resources können eben-

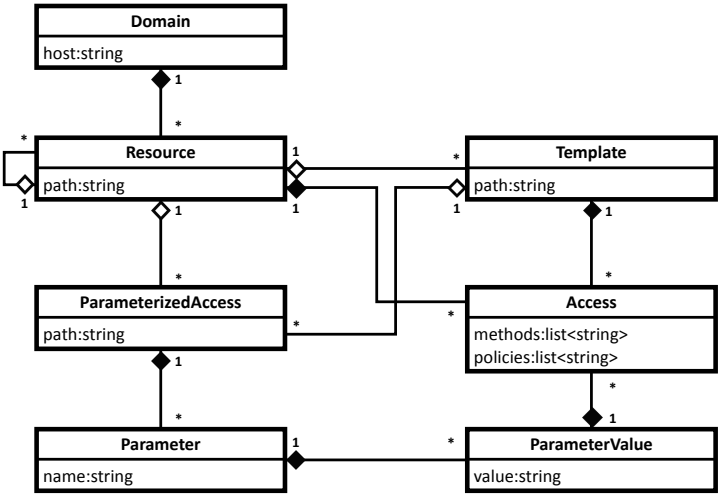

Abbildung 4.4: Abstrakte Syntax der Domain

falls *Templates* enthalten, welche als Subressourcen beziehungsweise Mengen von Subressourcen interpretiert werden können. Templates repräsentieren eine Menge von Resources, welche durch ein URI Template [2] identifiziert werden können. Templates selbst können keine untergeordneten Ressourcen besitzen. Existieren mehrere Variablen innerhalb eines URI Templates und müssen für diese verschiedene Zugriffsberechtigungen abgebildet werden, so müssen mehrere RestACL-Templates angelegt werden. Beispielsweise können drei RestACL-Templates verwendet werden, um Zugriffsberechtigungen für die URI

Templates */accounts/{accountId}*, */accounts/{accountId}/transfers* und */accounts/{accountId}/transfers/{transferId}* abzubilden. Eine alternative Abbildungsmöglichkeit wäre die Verschachtelung von RestACL-Templates und RestACL-Resources bei der eine Variable innerhalb eines URI Templates genau einem RestACL-Template zugeordnet wird. Da jedoch die Standardisierung von URI Templates mehrere Variablen pro Template zulässt, wurde auf die Verschachtelung von Templates und Resources verzichtet.

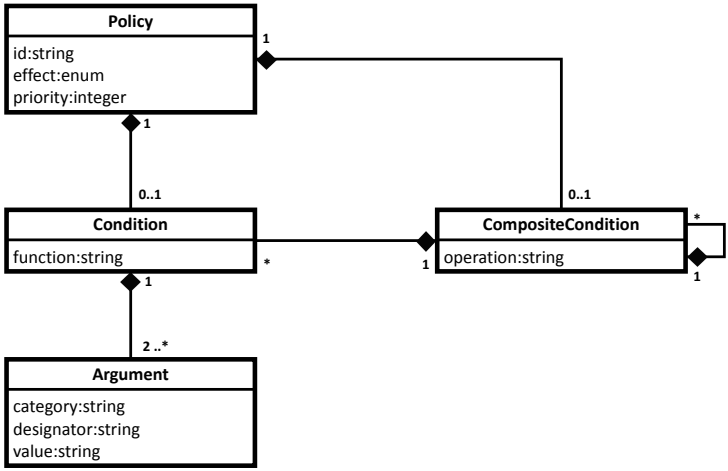

Abbildung 4.5: Abstrakte Syntax der Policy

Abbildung 4.5 zeigt die abstrakte Syntax einer RestACL-Policy. Eine Policy kann entweder eine *Condition* oder eine *CompositeCondition*

beinhalten. Conditions werden verwendet, um Attributbedingungen zu formulieren (beispielsweise muss die *Abteilung* der anfragenden Person den Wert *Entwicklung* haben, damit Zugriff gestattet werden kann). Deshalb haben sie eine Vergleichsfunktion (*function*). Eine CompositeCondition erlaubt es, komplexere Bedingungen zu formulieren. Sie ist aus ein oder mehr Conditions aufgebaut und kann wiederum andere CompositeConditions beinhalten, was bedeutet, dass auch hier wieder eine Teil-Ganzes-Hierarchie abgebildet wird (vgl. Composite Strukturmuster [47]). Die enthaltenen Elemente sind mittels einer logischen Operation verknüpft (beispielsweise AND, OR). Eine Condition muss mindestens zwei *Arguments* aufweisen, welche während der Entscheidungsfindung mit Hilfe der Vergleichsfunktion ausgewertet werden. In Abhängigkeit der Vergleichsfunktion können aber auch mehr Argumente angegeben werden. Argumente können entweder feste Werte (*Values*) sein (beispielsweise *Entwicklung*) oder Werte in der Zugriffsanfrage referenzieren mit Hilfe von *Categories* (beispielsweise *Subjekt*) und *Designators* (beispielsweise *Abteilung*). Im zweiten Fall wird bei der Auswertung in der Zugriffsanfrage nach dem Wert des Attributs mit der gleichen Category und dem gleichen Designator gesucht.

Eine Policy hat eine *id*, welche dazu verwendet wird, um den Verweis von einer Resource aus der Domain auf eine Policy zu ermöglichen. Zudem muss eine Policy einen *Effect* besitzen, entweder *Permit* oder *Deny*. *Undetermined* hingegen ist kein erlaubter Wert einer Policy, sondern kann lediglich das Resultat der Auswertung einer Zugriffsanfrage sein. Die Verwendung von Effects ermöglicht Black Listing, White Listing und Kombinationen aus beiden Ansätzen [88]. Ein weiteres Merkmal einer Policy ist die *Priority*. Der Zugriff auf eine Resource kann in mehreren Policies geregelt werden. Diese Poli-

cies können zu unterschiedlichen Effects führen, wenn sie anwendbar sind. In solch einem Fall muss die wichtigste Policy gefunden werden, welche schließlich die Zugriffsentscheidung festlegt. XACML setzt hierfür Combining-Algorithmen ein. Der RestACL-Ansatz nutzt hingegen Priorities, da sie verschiedene Vorteile bieten:

- Prioritäten bieten eine große Flexibilität bei der Erstellung von Kombinationen aus Black und White Listings. Beim Erstellen des Regelwerks können dadurch auch komplizierte Zugriffsberechtigungen intuitiv abgebildet werden.
- Prioritäten erlauben eine einfache Anpassung des Regelwerks bei sich ändernden Umständen. Müssen Zugriffsberechtigungen angepasst werden, so reicht möglicherweise bereits das Ändern der Priorities um das gewünschte Verhalten zu erzielen.
- Prioritäten ermöglichen eine schnelle Entscheidungsfindung, da Policies mit Prioritäten sortiert werden können und damit in absteigender Priorität abgearbeitet werden können. Sobald die erste Policy anwendbar ist, brauchen die folgenden Policies nicht weiter ausgewertet werden.

Innerhalb eines Policy Repositories dürfen nicht mehrere Policies die gleiche Priorität besitzen. Besäßen zwei Policies die gleiche Priorität, würde ein Konflikt entstehen. Allerdings wird hierdurch die Notwendigkeit eingeführt Prioritäten zu pflegen. Werden neue Policies angelegt, muss sichergestellt werden, dass die neu vergebene Priorität nicht bereits zuvor vergeben wurde. Eine weitere Optimierung dieses Mechanismus ist die Verwendung von lokalen Prioritäten. Dabei besitzen Policies nicht mehr eine allgemein gültige Priorität, sondern

nur noch eine Priorität für den Zugriff auf eine bestimmte Ressource. Diese lokale Priorität entspricht also der Reihenfolge der Auflistung der Policies in einem Access-Element innerhalb der Domain. Lokale Prioritäten werden in dieser Arbeit nicht betrachtet und können in zukünftigen Arbeiten detailliert untersucht werden.

Zum Beispiel könnte eine Policy *P1* Zugriff mit einer Priorität von *1* erlauben, eine Policy *P2* Zugriff mit einer Priorität von *2* verbieten und schließlich eine Policy *P3* Zugriff mit einer Priorität von *3* wieder erlauben. Die Verwendung eines Combining Algorithm wie XACMLs PermitOverrides könnte dieses Verhalten nicht ausdrücken. Mit XACML müsste der Combining Algorithm FirstApplicable eingesetzt werden, was gleichbedeutend mit der Abarbeitung einer Prioritätenliste ist. In diesem Fall beschreibt die Reihenfolge die Priorität.

In [94] wird eine Transformationsvorschrift beschrieben, die es ermöglicht alle XACML Combining Algorithm zu FirstApplicable zu transformieren. Ziel der Transformation ist eine schnellere Verarbeitung von Zugriffsanfragen. Deshalb ist es die flexibelste und gleichzeitig effizienteste Art, Policies mit Priorities zu kombinieren und anhand dieser die Entscheidungsfindung zu gestalten.

Abbildung 4.6 zeigt die abstrakte Syntax von RestACL *Requests* und *Responses*. Ein *Request* enthält die *URI* der angefragten Resource und die *method*, die verwendet wird, um mit der Ressource zu interagieren. Zusätzlich kann die Zugriffsanfrage mehrere *Attributes* haben. Der *value* eines *Attribute* kann innerhalb eines *Argument* in einer Policy referenziert werden mit Hilfe der *category* und des *designator*. Ein *Response* beinhaltet die finale Entscheidung: entweder *Permit*, *Deny* oder *Undetermined*. Im Gegensatz zu XACML wird nicht zwischen einer unbestimmten Entscheidung und einem Fehlerfall unterschieden, sondern in diesen Fällen *Undetermined* zurückgeliefert, da das anfra-

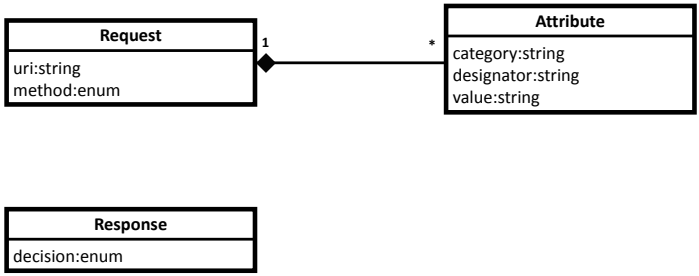

Abbildung 4.6: Syntax von Request und Response

gende Subjekt nicht über Details der Entscheidungsfindung informiert werden sollte. Diese Informationen könnte ein Angreifer ausnutzen. XACML unterscheidet zwischen den Resultaten *Not Applicable* und *Indeterminate*, *Indeterminate (Permit)* und *Indeterminate (Deny)*

4.7 Konkrete Syntax

Dieser Abschnitt beschreibt die konkrete Syntax von Domains, Policies, Requests und Responses. Es werden einige Beispiele gegeben, wie Zugriffskontrolle mit Hilfe der Sprache umgesetzt werden kann.

Domains Eine REST-Anwendung ist immer ressourcenorientiert. Deshalb ist es eine sinnvolle Idee, eine Sprache zur Definition von Zugriffsberechtigungen ebenfalls ressourcenorientiert zu gestalten. Durch

Ressourcenorientierung kann beispielsweise eine einfache Integration mit beliebten Modellierungswerkzeugen (beispielsweise Swagger [135]) sichergestellt werden, da diese Werkzeuge einen URI-Graphen zur Strukturierung der Services verwenden. Außerdem wird durch die Anwendung von Ressourcenorientierung ein schlüsselbasierter ABAC-Mechanismus möglich. Dieser wiederum ermöglicht die Verwendung von effizienten Datenstrukturen wie beispielsweise Hash-Tables. Zu jeder Ressource wird eine Menge von Zugriffsmethoden gespeichert. Zu jeder Zugriffsmethode wiederum gibt es einen Stapel mit Verweisen auf Policies. Das Verweisen auf Policies stellt ebenfalls einen schlüsselbasierten Mechanismus dar. Zusammen beschreiben diese beiden Mechanismen die Domain.

Die Domain ist identisch zum Aufbau der Ressourcen der Anwendung, die auf den REST-Prinzipien basiert, strukturiert. Dies bedeutet, dass die Domain hierarchisch angeordnete Resources enthält. Jede Resource kann Access-Elemente enthalten, welche von Zugriffsmethoden auf Policies verweisen. Innerhalb von Access-Elementen können mehrere Methoden und Policies angegeben werden.

Dieses Zugriffskonzept reduziert die potentiell anwendbaren Policies, welche pro Zugriffsanfrage ausgewertet werden müssen, dramatisch. XACML ist ein kompositioneller Ansatz, bei dem die Zugriffsentscheidung von der Komposition der Attributbedingungen abhängt. RestACL hingegen ist ein indexbasiertes Verfahren, welches anhand des Index zunächst die kleine Teilmenge von Policies identifiziert, welche ausgewertet werden müssen.

Abbildung 4.7 zeigt eine Abstraktion der Datenstruktur wie sie RestACL einsetzt, um einen Teil des Regelwerks abzubilden. Auflistung 4.1 zeigt die konkrete Syntax zum Beispiel aus Abbildung 4.7. Die eingesetzte Implementierung verwendet JSON als Datenformat, da es

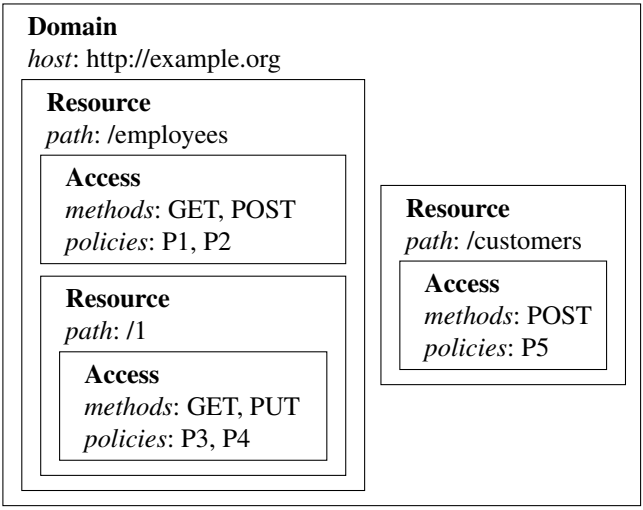

Abbildung 4.7: Domain mit zwei Ressourcen

im Umfeld von REST etabliert ist. Grundsätzlich ist das Datenformat jedoch nicht eingeschränkt. JSON-Objekte sind einfache Mengen von Schlüssel-Wert-Paaren, welche mit geschwungenen Klammern als Objekte gruppiert werden. Ein Wert kann dabei entweder atomar sein, eine Liste von Werten sein (notiert in eckigen Klammern) oder wiederum selbst ein Objekt sein. Das in Abbildung 4.7 wird in Auflistung 4.1 dargestellte. Das abgebildete Beispiel der Domain kann wie folgt interpretiert werden:

```
1  {
2    "host" : "http://example.org",
3    "resources" : [{
4      "path" : "/employees",
5      "access" : [{
6        "methods" : ["GET, POST"],
7        "policies" : ["P1", "P2"]
8      }],
9      "resources" : [{
10       "path" : "/1",
11       "access" : [{
12         "methods" : ["GET, PUT"],
13         "policies" : ["P3", "P4"]
14       }]
15     }]
16   },{
17     "path" : "/customers",
18     "access" : [{
19       "methods" : ["POST"],
20       "policies" : ["P5"]
21     }]
22   }]
23 }
```

Auflistung 4.1: Domain in JSON-Notation

GET- oder POST-Zugriff auf die Ressource mit dem Pfad */employees* auf dem Host *http://example.org* (also die Ressource mit der URI *http://example.org/employees*) wird erlaubt oder verboten, wenn die in den Policies *P1* oder *P2* enthaltenen Attributbedingungen erfüllt werden. GET- und PUT-Zugriff auf die Ressource mit der

Adresse *http://example.org/employees/1* wird erlaubt oder verboten, wenn die in den Policies *P3* oder *P4* enthaltenen Attributbedingungen erfüllt werden. POST-Zugriff auf die Ressource mit der Adresse *http://example.org/customers* wird erlaubt oder verboten, wenn die in Policy *P5* enthaltenen Attributbedingungen erfüllt werden.

Parameter Ein wichtiges Konzept, welches bei RESTful Services häufig zum Einsatz kommt um Listen-Ressourcen zu filtern, ist die Anwendung von URI-Parametern. Beispielsweise kann eine Liste von Angestellten mit Hilfe eines URI-Parameters so gefiltert werden, dass nur Mitarbeiter einer bestimmten Abteilung dargestellt werden. Hierzu wird hinter die Adresse der Ressource ein Schlüssel-Wert-Paar angehängt. Beispielsweise lässt sich die Angestelltenliste mit einem *department*-Filter versehen. Sollen nun lediglich die Angestellten aus der Entwicklung angezeigt werden, so kann die entsprechende URI verwendet werden: *http://example.org/employees?department=development*. Es ist denkbar, dass unterschiedliche Zugriffsregeln für gefilterte oder ungefilterte Listen gelten sollen. Beispielsweise könnte die Liste mit Angestellten aus der Entwicklung für die Mitarbeiter aus dieser Abteilung zugänglich sein, die ungefilterte Liste mit allen Angestellten jedoch nur für das Management. Deshalb muss eine Sprache zur Formulierung von Zugriffsberechtigungen das Konzept der URI-Parameter unterstützen. Hierfür muss die Domain-Datenstruktur ergänzt werden.

Das Abbilden von auf URI-Parametern bedingten Zugriffsrechten wird durch ParameterizedAccess-Elemente möglich. Abbildung 4.8 zeigt ein strukturelles Beispiel für ein ParameterizedAccess-Element und Auflistung 4.2 zeigt die konkrete Syntax für dieses Element. Das dargestellte Element hat den folgenden Effekt: für den Fall, dass die un-

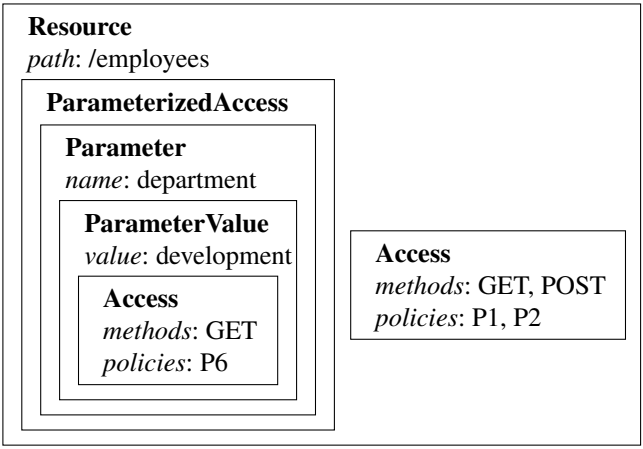

Abbildung 4.8: ParameterizedAccess mit einem Parameter

gefilterte Angestelltenliste angefragt wird, werden die Policies *P1* und *P2* zur Entscheidungsfindung herangezogen. Wird jedoch die gefilterte Liste angefragt, so wird zusätzlich die Policy *P6* ausgewertet. Dies bedeutet, dass die am höchsten priorisierte und anwendbare Policy aus der Menge *{P1, P2, P6}* die Zugriffsentscheidung für die Ressource *http://example.org/employees?department=development* festlegt, wenn eine GET Anfrage an die Ressource geschickt wird.

```
1  {
2    "resources" : [{
3      "path" : "/employees",
4      "access" : [{
5        "methods" : ["GET, POST"],
6        "policies" : ["P1", "P2"]
7      }],
8      "parameterizedAccess": [{
9        "parameters": [{
10         "name": "department",
11         parameterValues: [{
12           "value": "development",
13           "access": [{
14             "methods": ["GET"],
15             "policies": ["P6"]
16           }]
17         }]
18       }]
19     }]
20   }]
21 }
```

Auflistung 4.2: ParameterizedAccess in JSON Notation

Templates Templates können verwendet werden, wenn mehrere Ressourcen des gleichen Typs eine oder mehrere Zugriffsregeln gemeinsam haben. Beispielsweise kann jeder Angestellte berechtigt sein, seine Account-Informationen zu aktualisieren. Der Account des Angestellten mit der Kennung *1* kann mit dem Pfad */employees/1* adressiert werden. Eine PUT Anfrage zu dieser Ressource kann dann für den

Angestellten mit Kennung *1* erlaubt werden. Ganz analog gilt für den Angestellten mit der Kennung *2*, dass eine PUT Anfrage auf die Ressource mit dem Pfad */employees/2* möglich sein sollte. Wird für jeden Benutzer explizit eine solche Regel angelegt, führt dies zu unnötiger Redundanz. Modellierungswerkzeuge wie Swagger oder RAML verwenden deshalb URI Templates, um Ressourcen desselben Typs zu modellieren [2]. Ein Beispiel für ein URI Template ist *http://example.org/employees/{id}*, wobei *{id}* der Teil ist, welcher in einer konkreten Anfrage mit einem expliziten Wert ersetzt wird. Eine Sprache zur Formulierung von Zugriffsberechtigungen sollte Templates unterstützen, da diese den Wartungsaufwand für das Regelwerk erheblich reduzieren können.

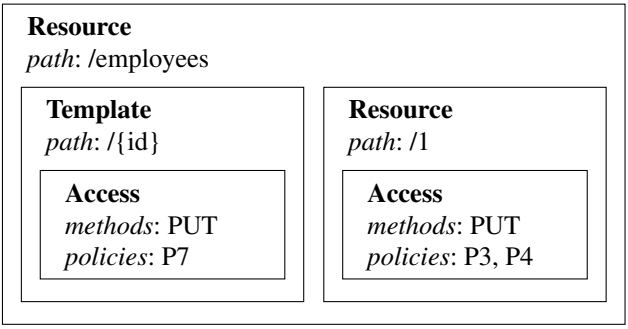

Abbildung 4.9: Policy mit Template zum Identifizieren von Ressourcen

Abbildung 4.9 zeigt wie Templates in die Ressourcenhierarchie der Domain integriert werden können. Wie in den vorherigen Abschnitten beschrieben, können Ressourcen untergeordnete Subressourcen besit-

zen. Ganz analog zu einer Ressource kann auch ein Template einer übergeordneten Ressource zugewiesen werden. Während der Evaluationsphase wird für eine konkrete Anfrage dann beides ausgewertet: die anwendbaren Templates und explizit angegebene Ressourcen. Das bedeutet, dass Templates zusätzliche Policies für explizite Ressourcen angeben. Beispielsweise werden für eine PUT Anfrage auf die Ressource mit dem Pfad */employees/1* die Policies *P3*, *P4* und *P7* ausgewertet. Auflistung 4.3 zeigt die zu diesem Beispiel gehörende Syntax in JSON-Notation.

```
1  {
2    "resources" : [{
3      "path" : "/employees",
4        "resources" : [{
5          "path": "/{id}",
6          "access": [{
7            "methods": ["PUT"],
8            "policies": ["P7"]
9          }]
10       },{
11         "path": "/1",
12         "access": [{
13           "methods": ["PUT"],
14           "policies": ["P3", "P4"]
15         }]
16       }]
17     }]
18   }]
19 }
```

Auflistung 4.3: Template in JSON Notation

Policies Die tatsächliche Zugriffskontrolllogik wird in Policies gespeichert. Diese sind dazu ausgelegt das attributbasierte Zugriffskontrollmodell umzusetzen. Dies ermöglicht die Erstellung verschiedenster Varianten von Zugriffsregeln, die eine große Bandbreite von Anwendungen unterstützen können. Während die Domain-Datenstruktur die Beziehung zwischen Ressourcen ausdrückt und gleichzeitig den Ressourcen Policies zuweist, so wird die Policy-Datenstruktur dafür verwendet um Conditions zu formulieren, welche die Zugriffsentscheidung festlegen. Diese Conditions vergleichen dabei Attributwerte einer Zugriffsanfrage mit Sollwerten. Abbildung 4.10 zeigt ein strukturelles Beispiel einer Policy, während Auflistung 4.4 die konkrete Syntax für dieses Beispiel liefert.

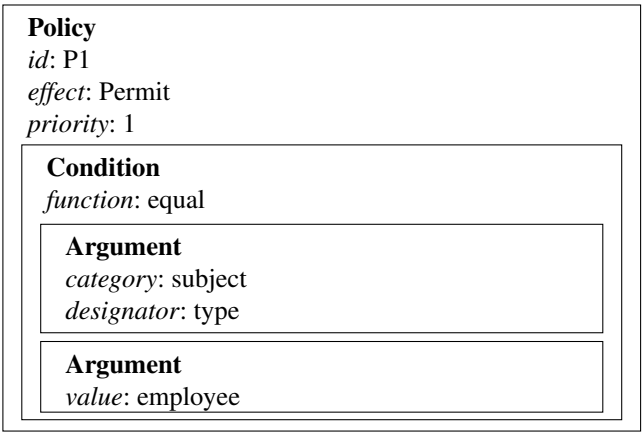

Abbildung 4.10: Policy mit einer Attributbedingung

```
{
  "policies": [{
    "id": "P1",
    "effect": "Permit",
    "priority": "1",
    "condition": {
      "function": "equal",
      "arguments": [{
        "category": "subject",
        "designator": "type"
      },{
        "value": "employee"
      }]
    }
  }]
}
```

Auflistung 4.4: Policy in JSON-Notation

Die Condition aus Abbildung 4.10 überprüft den *type* eines Subjekts. Der in der Zugriffsanfrage angegebene type wird dann mit dem Wert *employee* verglichen. Hierbei kommt die *function* zum Einsatz, welche einen Boolschen Wert zurückliefert. Berechnet sich die function zu *true*, trifft die Bedingung zu und die Policy wird anwendbar. Wie in der Domain-Datenstruktur zu sehen ist, können mehrere Policies zur Entscheidungsfindung beitragen. Diese können unterschiedliche Effekte haben. Um zu einem eindeutigen Ergebnis zu kommen, wird jeder Policy eine Priorität zugewiesen. Die anwendbare Policy mit der höchsten Priorität bestimmt die finale Zugriffsentscheidung. Die Zugriffsentscheidung entspricht dem Effect dieser Policy. Categories werden eingesetzt, da verschiedene Entitäten innerhalb einer

Zugriffsanfrage Attribute mit gleichem Namen besitzen können. Etwa können ein Subjekt und eine Ressource beide jeweils eine eindeutige Kennung und einen Namen haben. Um diese Attribute den richtigen Entitäten zuordnen zu können, besitzen Attribute eine Category.

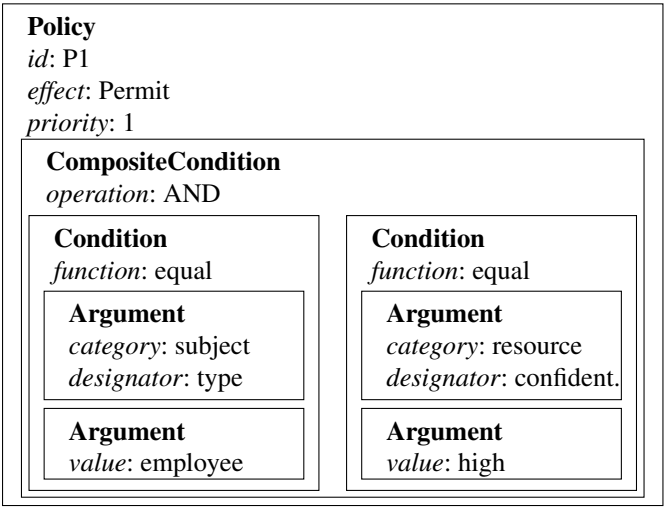

Abbildung 4.11: Policy mit zwei Attributbedingungen

Soll die in Abbildung 4.10 dargestellte Policy derart erweitert werden, dass auf die zu schützenden Ressourcen nur noch zugegriffen werden darf, wenn der type des anfragenden Subjekts gleich *employee* ist und wenn der type der Ressource gleich *account* hat, müssen zwei Conditions mit einer logischen Operation (AND) verknüpft werden.

```
1  {
2    "policies": [{
3      "id": "P1",
4      "effect": "Permit",
5      "priority": "1",
6      "compositeCondition": {
7        "operation": "AND",
8        "conditions": [{
9          "function": "equal",
10         "arguments": [{
11           "category": "subject",
12           "designator": "type"
13         },{
14           "value": "employee"
15         }]
16       },{
17         "function": "equal",
18         "arguments": [{
19           "category": "resource",
20           "designator": "confidentiality"
21         },{
22           "value": "high"
23         }]
24       }]
25     }
26   }]
27 }
```

Auflistung 4.5: CompositeCondition in JSON-Notation

Die Sprache sieht hierfür CompositeConditions vor, welche es ermöglichen verschiedene Kombinationen von logischen Ausdrücken

auf Conditions anzuwenden. CompositeConditions bestehen deshalb aus einer *Operation* (beispielsweise *AND, OR, XOR*) und mehreren Conditions, welche mit Hilfe der *Operation* verknüpft sind.

Abbildung 4.11 zeigt eine Policy mit einer CompositeCondition, welche die beiden zuvor erwähnten Bedingungen verknüpft. In Auflistung 4.5 wird die zugehörige konkrete Syntax dargestellt. CompositeConditions können somit dazu verwendet werden, beliebig komplexe boolsche Ausdrücke zu formulieren.

Request und Response Abbildung 4.12 zeigt eine Abstraktion der Datenstrukturen für Zugriffsanfragen (*Request*) und Zugriffsentscheidungen (*Response*). Zugriffsanfragen beinhalten eine Liste von Attributen, die zur Entscheidungsfindung heran gezogen werden. Die Zugriffsentscheidung ist ein einzelnes Datum, mit dem Wert *Permit*, *Deny* oder *Undetermined* für den Fall, dass keine anwendbaren Policies gefunden wurden.

Eine Zugriffsanfrage (ein Request) muss eine eindeutige Adresse - eine *URI* - und eine der Methoden der einheitlichen Schnittstelle des RESTful Service beinhalten. Darüber hinaus kann die Zugriffsanfrage eine beliebige Anzahl von Attributen haben. Ein Attribut ist dabei ein Triple bestehend aus *Category*, *Designator* und *Value*. Wie bereits in den vorherigen Abschnitten beschrieben, werden *Category* und *Designator* dazu verwendet einen *Value* in der Anfrage zu identifizieren. Der *Value* wird dann während der Entscheidungsfindung ausgewertet. Die konkrete Syntax aus Auflistung 4.6 zeigt das Beispiel aus Abbildung 4.12.

Request
URI: http://example.org/employees/1
method: GET

Attribute
category: subject
designator: type
value: employee

Attribute
category: resource
designator: confidentiality
value: high

Response
decision: Permit

Abbildung 4.12: Request und Response

```
{
  "uri": "http://example.org/employees/1",
  "method": "GET",
  "attributes": [{
    "category": "subject",
    "designator": "type",
    "value": "employee"
  },{
    "category": "resource",
    "designator": "confidentiality",
    "value": "high"
  }]
}
```

Auflistung 4.6: Request in JSON-Notation

Ein *Response* ist eine sehr einfache Datenstruktur, welche lediglich die finale Zugriffsentscheidung beinhaltet. Auflistung 4.7 zeigt die konkrete Syntax einer Zugriffsentscheidung, wie sie von Evaluation Engine an den RESTful Server geschickt wird.

```
1 {
2   "decision": "Permit"
3 }
```

Auflistung 4.7: Response in JSON-Notation

4.8 Sprachinterpretation

In diesem Abschnitt werden Details zur Interpretation der Sprache beschrieben. Es werden die einzelnen Evaluationsphasen erläutert und geeignete Algorithmen beschrieben, die die Entscheidungsfindung beschleunigen.

Evaluationsphasen Eine Zugriffsanfrage durchläuft zwei oder drei Phasen bevor eine Zugriffsentscheidung getroffen wird. Dabei sind die erste und dritte Phase verpflichtend und müssen durchlaufen werden, während die zweite Phase optional ist und übersprungen werden kann. Die zweite Phase wird bei der Einbindung eines oder mehrerer Attribute Provider benötigt.

(P1) *Policy Identifikation*: In der ersten Phase werden die zu evaluierenden Policies identifiziert. Hierfür wird die Ressourcenadresse ausgewertet und die zugehörigen Access-Elemente aus der Domain geladen. Wenn die Zugriffsmethode in der Zugriffsanfrage und im Access-Element übereinstimmen, werden die Policy-Referenzen des

Access-Elements für die spätere Auswertung gesichert. Dies bedeutet, dass das Domain-Modell, welches im vorherigen Abschnitt beschrieben wurde, in der ersten Phase ausgewertet wird. Im folgenden Abschnitt wird der Algorithmus dieser Evaluierung formell beschrieben.

(P2) (Optional) *Sammeln von Attributen*: Die Zugriffsanfrage kann eine beliebige Anzahl von Attributen beinhalten. Vertrauenswürdige Clients können beispielsweise ihre eigenen Attribute innerhalb der Zugriffsanfrage senden. Darüber hinaus bietet das Zugriffskontrollsystem noch die Möglichkeit weitere Attributquellen in Form von Attribute Providern anzubinden. Beim Eintreffen einer Zugriffsanfrage können diese zusätzlichen Attributquellen konsultiert werden, so dass weitere Attribute zur Evaluierung verwendet werden können.

Beispielsweise könnte ein RESTful Server lediglich eine unvollständige Liste von Attributen senden oder externe Identity Provider könnten zum Einsatz kommen, um anfragende Subjekte zu identifizieren. In solchen Fällen können Attribute Provider zusätzliche Attribute bereitstellen. Ebenfalls können Umgebungsattribute wie die aktuelle Uhrzeit oder ähnliches durch Attribute Provider bereitgestellt werden. Beim Konsultieren eines Attribute Provider können verschiedene Strategien zum Einsatz kommen, welche einen Einfluss auf die Performanz haben in Abhängigkeit davon, wie die Attribute Provider an das Zugriffskontrollsystem angebunden sind. Dies kann entweder in **(P2)** oder **(P3)** geschehen. Das bedeutet, dass zunächst alle bekannten Attribute der beteiligten Entitäten gesammelt werden können bevor die Berechnung der Zugriffsentscheidung beginnt. Alternativ können während der Berechnung der Zugriffsentscheidung die in den Policies angegebenen Attribute abgefragt werden.

(P3) *Entscheidungsfindung*: In dieser Phase wird schließlich die tatsächliche Zugriffsentscheidung berechnet. Eine priorisierte Liste

von Policies, welche in **(P1)** identifiziert wurde, wird mit den Attributen der Zugriffsanfrage und den in Phase **(P2)** geladenen Attributen abgeglichen. Das Resultat dieser Berechnung wird an den RESTful Server weitergeleitet. In den folgenden Abschnitten wird der Algorithmus zur Berechnung der Zugriffsentscheidung formell beschrieben. In Abhängigkeit der gewählten Strategie zum Sammeln von Attributen kann diese Phase parallel zu oder nach Abschluss von **(P2)** ausgeführt werden.

Evaluationsalgorithmen In diesem Abschnitt werden die wesentlichen Algorithmen aus den Phasen **(P1)** und **(P3)** mit Hilfe von Pseudocode beschrieben.

Policy Identifikation: In **(P1)** wird ein sehr schneller Such- beziehungsweise Look-Up-Algorithmus benötigt, um die Policies zu identifizieren, welche für den Zugriff auf eine bestimmte Ressource ausgewertet werden müssen. Hash-Tables sind ein idealer Kandidat für solch einen Algorithmus, da sie für eben solche Zwecke entworfen wurden. Sie weisen im praktischen Einsatz eine äußerst starke Performanz auf [28]. Aus diesem Zweck wird Hashing in der Referenzimplementierung verwendet. Der Mapping-Algorithmus identifiziert anhand von *scheme*, *authority* und *path* das Ressource-Element innerhalb des Domain-Modells, welches der angefragten Ressource zugeordnet werden kann. Dazu werden in der Hash-Table die Access-Elemente gespeichert, welche der Ressource zugeordnet sind. Ein Beispiel für einen Eintrag in einer Hash-Table ist in Tabelle 4.1 dargestellt. Der Schlüssel *http://www.example.org/employees* wird als Hash-Wert der Adresse der angefragten Ressource abgebildet. Der zugeordnete Wert beinhaltet die Access-Elemente gemäß dem Eintrag in der Domain.

Hash	Value
h(http://www.example.org/employees)	[{“methods”: [“GET”], “policies”: [“P1”, “P2”]}]

Tabelle 4.1: Datenstruktur der Hash-Table

Formell kann der Algorithmus beschrieben werden wie in Algorithm 3 dargestellt. Dabei werden die folgenden Abkürzungen verwendet: r ist eine RestACL-Resource, a ist ein Access-Element, pa ist ein ParameterizedAccess-Element, pm ist ein Parameter, m ist die *method* eines RestACL-Requests beziehungsweise die *method* eines Access-Elements und p ist der Identifier einer RestACL-Policy.

Der erste Schritt zur Identifikation der zugeordneten Policies ist die Berechnung eines Hash-Wertes über *scheme*, *authority* und *path* der URI der angefragten Ressource. Das Zugriffskontrollsystem vergleicht diesen Wert dann mit einer Reihe von im Vorfeld berechneten Hash-Werten. Für jede in der Domain eingetragene Ressource existiert dabei ein im Vorfeld berechneter Wert in einer Hash-Table. Dies bedeutet, dass beim Eintreffen einer Zugriffsanfrage eine Hash-Suche durchgeführt wird und das Resultat dieser Suche ein Resource-Element r ist, inklusive der zugehörigen Access- und ParameterizedAccess-Elemente $\{a_1, ..., a_{n_1}\}_r$.

In einem zweiten Schritt werden die *methods* $\{a_i.m_1, ..., a_i.m_{n_2}\}$ der Access-Elemente $a_i \in \{a_1, ..., a_{n_1}\}$ verglichen mit der angefragten Zugriffsmethode $q.m$. Ist die Zugriffsmethode in einem Set von Methoden eines Access-Elements enthalten, werden alle Policy-Identifier dieses Access-Element zur Auswertung vorgemerkt. Nach dem Vergleich aller Access-Elemente entsteht somit die vollständige Liste

Input: Request q
Output: Collection of Policies $\mathcal{P}_r$

$r = search(hash(q.uri))$;
$\mathcal{P}_r := \emptyset$
$\forall\ a_i \in \{r.a_1, ..., r.a_{n_1}\}$
  $if\ (q.m \in \{a_i.m_1, ..., a_i.m_{n_2}\})$
    $\mathcal{P}_r = \mathcal{P}_r \cup \{a_i.p_1, ..., a_i.p_{n_3}\}$;
$\forall\ pa_i \in \{r.pa_1, ..., r.pa_{n_4}\}$
  $\forall\ pm_j \in \{pa_i.pm_1, ..., pa_i.pm_{n_5}\}$
    $if\ (q.uri.query \cong pm_j)$
      $\forall\ a_k \in \{pm_j.a_1, ..., pm_j.a_{n_6}\}$
        $if\ (q.m \in \{a_k.m_1, ..., a_k.m_{n_7}\})$
          $\mathcal{P}_r = \mathcal{P}_r \cup \{a_k.p_1, ..., a_k.p_{n_8}\}$;
$return\ \mathcal{P}_r$;

$n_1, ..., n_8 \in \mathbb{N}$

Algorithmus 3: Policy Identifikation

von Policies, welche ausgewertet werden müssen. Das Resultat der gesamten Berechnung ist eine Liste von Policy-Identifiern.

Schließlich werden noch die ParameterizedAccess-Elemente ausgewertet. Hier ist ein zusätzlicher Arbeitsschritt notwendig, um Paramatername und Parameterwert aus der Zugriffsanfrage und dem Resource-Element abzugleichen. Konnte ein Parameter erfolgreich abgeglichen werden, so werden die dem Parameterwert zugeordneten Access-Elemente analog zum zweiten Schritt überprüft und gegebenenfalls die Menge der auszuwertenden Policies erweitert.

Templateauswertung: Eine besondere Herausforderung bei der Auswertung einer Zugriffsanfrage ist die Auswertung von URI Templates innerhalb der Domain. Durch die Verwendung von Templates geht die Eindeutigkeit der Ressourcenadresse verloren. Eine URI aus der Zugriffsanfrage kann dann gleichzeitig auf einen Pfad und verschiedene Templates in der Domain abgebildet werden. Dies kann insbesondere dann problematisch werden, wenn viele Templates verwendet werden. Für jedes verwendete Template muss ein Pattern Matching gemacht werden, um zu prüfen, ob die konkrete URI mit dem Template übereinstimmt. Dies ist vergleichsweise sehr zeitaufwendig.

Um dieses Problem zu umgehen, kann die Annahme einer geschlossenen Welt gemacht werden. Dies bedeutet, dass angenommen wird, dass der RESTful Service alle Ressourcen kennt. Dies ist eine valide Annahme, da das Anlegen einer Ressource eine normale Interaktion über die einheitliche Schnittstelle ist. Beispielsweise wird bei HTTP die POST-Methode genutzt, um eine Ressource anzulegen. Beim Anlegen der Ressource, kann dann die URI der neuen Ressource mit existierenden Templates abgeglichen und für die neue Ressource ein Eintrag in der Domain erstellt werden. Wird wiederum ein neues Template erstellt, so kann dieses beim Erstellen gegen die konkreten

URIs in der Domain abgeglichen werden und die zusätzlichen Verweise auf Policies in der Domain eingetragen werden. Da alle Ressourcen bekannt sind, ist sichergestellt, dass auch alle Ressourcen, die durch das Template abgebildet werden, in der Domain eingetragen sind. Analog müssen beim Löschen eines Templates alle Ressourcen, deren Adresse dem URI Template entspricht, angepasst werden.

Mit diesem Verfahren ist es lediglich beim Anlegen einer Ressource oder eines Templates notwendig, konkrete URIs gegen URI Templates abzugleichen. Bei allen folgenden Anfragen, kann dann der eindeutige Eintrag in der Domain verwendet werden, ohne ein zusätzliches Pattern Matching durchführen zu müssen.

Entscheidungsfindung: Ein schneller Algorithmus mit einer geringen durchschnittlichen Berechnungszeit wird in Phase **(P3)** benötigt, um eine Zugriffsentscheidung in Abhängigkeit der gegebenen Attribute und der identifizierten Policies zu finden. Die in diesem Kapitel beschriebene, prototypische Implementierung speichert alle Policies in einer weiteren Hash-Table im Hauptspeicher. So kann auf die im ersten Schritt identifizieren Policies sehr schnell zugegriffen werden. Aber auch wenn die Policies nicht im Hauptspeicher gehalten werden, ist eine verhältnismäßig schnelle Identifikation möglich, da nur eine kleine Teilmenge der Policies in den Hauptspeicher geladen werden muss.

Verschiedene Policies können verschiedene Effekte haben und somit müssen anwendbare Policies mit unterschiedlichen Effekten zu einem Ergebnis kombiniert werden. Dies bedeutet, dass ein schneller Kombinationsalgorithmus benötigt wird. Priorisierung bietet den Vorteil einer schnellen Entscheidungsfindung, da der Evaluationsalgorithmus stoppen kann, sobald die anwendbare Policy mit der höchsten Priorität gefunden wurde. Werden die Policies nun in absteigender

Reihenfolge der Prioritäten evaluiert, kann der Evaluationsalgorithmus nach der ersten anwendbaren Policy stoppen. Ebenfalls bietet Priorisierung eine hohe Flexibilität bei der Erstellung und Kombination von Policies. Der Evaluationsmechanismus kann deshalb formell beschrieben werden, wie in Algorithm 4 dargestellt.

Input: Collection of Policies $\mathcal{P}_r$, Collection of Attributes $\mathcal{A}$
Output: Effect e

$\forall\ p_i \in prioritize(\{p_1, ..., p_{n_4}\} \in \mathcal{P}_r)$
$\quad if\ (isApplicable(p_i, \mathcal{A}))$
$\quad\quad return\ p_i.effect;$
$return\ Undetermined;$

$n_4 \in \mathbb{N}$

Algorithmus 4: Entscheidungsfindung

Der Evaluationsalgorithmus iteriert über eine nach Prioritäten sortierte Liste von Policies $p_i \in \mathcal{P}_r$. Policies mit gleichen Prioritäten sind nicht erlaubt. Für jede Policy wird die Anwendbarkeit der Policy geprüft, indem die Sammlung von Attributen $\mathcal{A}$ mit der logischen Bedingung der Policy abgeglichen wird. Wie zuvor beschrieben, ist eine Policy anwendbar, wenn die zugehörige Condition beziehungsweise CompositeCondition erfüllt ist. Das bedeutet, eine Policy ist anwendbar, wenn alle in der Bedingung geforderten Attribute in der Zugriffsanfrage enthalten sind und die geforderten Werte aufweisen. Ist die Policy anwendbar, liefert der Algorithmus den Effect der Policy zurück und stoppt. Für den Fall, dass die Policy nicht anwendbar ist,

fährt der Algorithmus mit der Überprüfung der Policy mit der nächst höheren Priorität fort. Ist keine der identifizierten Policies anwendbar, so ist das Resultat der Zugriffsanfrage unbestimmt *Undetermined*. Die Verantwortung über die Genehmigung des Zugriffs liegt dann beim RESTful Server.

4.9 Analyse

Der RestACL-Mechanismus kann im Vergleich zu den generischen Datenstrukturen eines XACML-Mechanismus auf effizienteren Datenstrukturen aufgebaut werden. So können zur Optimierung der Verarbeitungszeit Hash-Tables verwendet werden. Deshalb ist es möglich, Antwortzeiten der Ordnung $O(1)$ zu erreichen.

Um die Bedeutung dieser Eigenschaft hervorzuheben, soll wie bei der Analyse von XACML in Kapitel 3.5 beispielhaft von durchschnittlich 2 Zugriffsmethoden und 5 Regeln für den Zugriff auf eine Ressource ausgegangen werden. Durch den Einsatz von Hash-Tables wird die vollständige Auswertung auf eine Maximalzahl von $2*5=10$ Elementen reduziert. Dies stellt eine dramatische Reduktion der potentiell auszuwertenden Elemente im Vergleich mit XACML dar, wo die Anzahl an potentiell auszuwertenden Elementen abhängig ist von der Anzahl der zu schützenden Ressourcen.

Dies wiederum bedeutet, dass der signifikanteste Teil der Entscheidungsfindung die Auswertung der Attributbedingungen ist. Aus der Perspektive der Verarbeitungszeit ist dies ein bedeutender Vorteil, da sich die Verarbeitungszeit $t(r)$ nun berechnen lässt als:

$$t(r) = c * m(r) * p(r), c = const. \tag{4.3}$$

mit $m(r)$ als durchschnittliche Anzahl an deklarierten Zugriffsmethoden pro Ressource r und $p(r)$ als durchschnittliche Anzahl an Berechtigungen, welche für eine Methode existieren.

Vergleich man die Gleichungen (3.10) und (4.3), so lässt sich schnell erkennen, dass in (3.10) eine Abhängigkeit von der Gesamtanzahl der Ressourcen n existiert, welche in (4.3) nicht vorkommt. Dies bedeutet, dass die Verarbeitungszeit unabhängig von der Anzahl an Ressourcen ist und das System deshalb sehr viel besser als XACML die Skalierungsforderung **(R4)** erfüllt.

An dieser Stelle sei erwähnt, dass ideales Hashing für RESTful Services nicht möglich ist. Dies liegt darin begründet, dass ein RESTful Service in der Theorie eine unbeschränkte Anzahl von Ressourcen bearbeiten kann und deshalb keine perfekte Hash-Funktion gefunden werden kann. Als Konsequenz kann es passieren, dass verschiedene Ressourcenadressen auf denselben Hashwert abgebildet werden, was einer Kollision beim Auflösen des Hashwerts entspricht. Kollisionen können mit Hilfe von Techniken wie *Chaining* aufgelöst werden. Allerdings erhöht Chaining wiederum die Suchzeiten zur Identifikation der auszuwertenden Regeln. Dennoch performen auch solche Hash-Tables überaus gut in der Praxis [28]. Zusammenfassend kann festgehalten werden, dass mittels effizienter Datenstrukturen die Skalierunganforderung **(R4)** deutlich effektiver durch RestACL unterstützt wird.

Betrachtet man nun den Speicherbedarf von RestACL im Vergleich zur Lösung mit Hilfe der Binärbaums in XACML, so lässt sich festhalten, dass aus genereller Sicht Hash-Tables einen höheren Speicherbedarf als Binärbäume haben, da sie leere Buckets enthalten, in denen Werte gespeichert werden könnten. Wie jedoch zuvor gezeigt wurde, verdoppelt der Binärbaum zur Optimierung der Verarbeitungszeit den Speicherbedarf des Regelwerks in XACML. Auf der anderen Seite

können fortgeschrittene Konzepte wie *Extendible Hashing* zur Reduktion des Speicherbedarfs angewendet werden [39]. Solche Hash-Tables haben eine maximale Größe von $2n$, wenn n Schlüssel-Wert-Paare gespeichert werden müssen. Deshalb kann man davon ausgehen, dass die theoretischen Größen der Datenstrukturen sich in der gleichen Dimension bewegen.

Allerdings kann RestACL zwei Charakteristika ausweisen, welche den Speicherbedarf substantiell reduzieren. Das erste Charakteristikum ist das Design der Domain. Generische Lösungen müssen relativ komplexe Inhalte abbilden, um das ABAC-Modell zuzüglich zu den domänenspezifischen Daten abzubilden. Beispielsweise müssen für ein Regelwerk zur Identifikation einer Ressource in XACML-Policy Sets und Policies inklusive Target-Beschreibung angelegt werden. Diese beinhalten verschiedene Metadaten wie etwa den Datentyp. Diese Metadaten sind bei maßgeschneiderten Lösungen überflüssig, da sie implizit als gegeben angenommen werden können. Das zweite Charakteristikum ist lose Kopplung zwischen Domain und Policy Repository. Die lose Kopplung vereinfacht die Wiederverwendbarkeit, da Policies modular und einfach lesbar geschrieben werden können. Innerhalb der Domain ist lediglich eine Referenzierung anstelle einer expliziten Angaben zum Inhalt der Policy notwendig. Es wird lediglich die Referenz mehrfach verwendet.

In Kapitel 3.5 wurde eine Analyse von XACML vorgenommen anhand eines beispielhaften Regelwerks mit 28880 Policy Sets, Policies und Rules. Wird dieses Regelwerk in RestACL geschrieben, kann die Dateigröße von 22 MB auf 1 MB reduziert werden. Im effektiven Einsatz mit einer RestACL-Implementierung werden 10 MB Arbeitsspeicherbedarf gemessen, im Vergleich zu 130 MB beim Einsatz von XACML mit Balana.

Analyseergebnis RestACL wurde ressourcenorientiert entworfen, so dass der Einsatz von Datenstrukturen, die eine effiziente Verarbeitung garantieren, möglich ist. RestACL bietet Performanzsteigerungen und reduzierten Speicherbedarf, so wie es Anforderung **(R4)** an ein Zugriffskontrollsystem in einer RESTful-Umgebung erwartet. Die Effizienz und das Design von REST ermöglichen die Autorisierung einzelner Zugriffe unabhängig von den Ergebnissen vorheriger Zugriffe, weshalb das HATEOAS-Prinzip nicht verletzt wird, so wie es Anforderung **(R1)** verlangt. Die Unterscheidung zwischen Domäne und eigentlichem Regelwerk und die einfache Datenstruktur der beiden Elemente ermöglichen eine schnelle und unkomplizierte Wartung der Zugriffsrechte durch den RESTful Server, wenn dies gefordert ist. Deshalb können Änderungen am Regelwerk auf eine schnelle Art und Weise integriert werden, so wie es Anforderung **(R2)** verlangt. Da RestACL noch immer ein attributbasierter Ansatz ist, unterstützt es von Haus aus eine große Bandbreite verschiedener Anwendungen mit unterschiedlichen Zugriffsanforderungen. Damit kann auch Anforderung **(R3)** als erfüllt angesehen werden.

4.10 Experimentelle Ergebnisse

Einer der Hauptnachteile von XACML ist die Tatsache, dass die Verarbeitungszeit für eine Zugriffsanfrage davon abhängig ist, wie das Regelwerk geschrieben ist [133]. Um die Effizienz von RestACL zu messen, wurde deshalb der Mechanismus mit der performanzoptimierten Variante des XACML-Regelwerks mit Bäumen (vgl. Kapitel 3) verglichen. Wie bereits zuvor beschrieben, ist der Nachteil der performanzoptimierten Variante ein erhöhter Speicherbedarf bei XACML.

Der erhöhte Speicherbedarf resultiert aus einem Overhead, welcher aufgrund des generischen Designs von XACML nicht intern verarbeitet werden kann, sondern explizit im Regelwerk geschrieben steht. Die prototypische Implementierung von RestACL ist in Java geschrieben und verwendet HashMaps als Hashing-Algorithmus zur Minimierung der Auswertungszeiten. Es entsteht kein zusätzlicher Overhead für das Erstellen des Regelwerks. Die Testumgebung ist gegenüber Kapitel 3.6 unverändert.

Ansatz	Ressourcen	Verarbeitungszeit	Speicherbedarf
XACML	10	14.1ms	11MB
XACML	100	14.5ms	21MB
XACML	1000	14.7ms	112MB
XACML	10000	14.7ms	731MB
RestACL	10	0.3ms	6MB
RestACL	100	0.2ms	6MB
RestACL	1000	0.4ms	8MB
RestACL	10000	0.3ms	30MB
RestACL	100000	0.4ms	133MB
RestACL	1000000	0.3ms	850MB

Tabelle 4.2: Antwortzeiten und Speicherverbrauch für optimiertes XACML und RestACL

Tabelle 4.2 zeigt die durchschnittliche Verarbeitungszeit für eine Zugriffsanfrage und den Arbeitsspeicherbedarf, der benötigt wird um das Regelwerk in optimiertem XACML und RestACL umzusetzen. Man kann sehen, dass die Antwortzeiten in etwa auf einem konstanten Level bleiben. Für die optimierte XACML-Lösung kann dies damit be-

gründet werden, dass ein Initialisierungsaufwand gegeben ist, welcher groß ist im Vergleich mit der Verarbeitungszeit für die Zugriffsanfrage. Die Messungen des RestACL-Mechanismus bestätigen die Annahme von konstanten Antwortzeiten aufgrund der Anwendung von Hashing zur Identifikation der auszuwertenden Regeln. Der Preis für die Performanzoptimierung bei XACML kann beim Speicherbedarf abgelesen werden. RestACL ist in der Lage bei gleichem Speicherbedarf etwa 100 mal so viele Ressourcen zu kontrollieren wie XACML. Zudem ist die Auswertungszeit eine Größenordnung kleiner.

In diesem Test wurden für jede Ressource 1 bis 4 Zugriffsmethoden festgelegt mit jeweils 5 Regeln/Policies, welche die Attributbedingungen beinhalteten. Je Regel/Policy wurde eine Attributbedingung festgelegt. Zugriffsanfragen haben 1 bis 10 Attribute beinhaltet. Ein externer Attribut Provider wurde nicht verwendet. Weitere Messungen mit XACML waren auf dem Testsystem nicht möglich, da diese zu einem Speicherüberlauf geführt haben.

Ansatz	Ressourcen	Verarbeitungszeit	Varianz
-	1	3ms	1ms
RestACL	10	3ms	1ms
RestACL	100	3ms	1ms
RestACL	1000	3ms	1ms
RestACL	10000	4ms	1ms
RestACL	100000	4ms	2ms
RestACL	1000000	4ms	3ms

Tabelle 4.3: Verarbeitungszeit und Varianz für parallele Zugriffsanfragen

Eine weitere wichtige Größe ist das Verhalten bei der Bearbeitung von parallelen Zugriffsanfragen. Um das Verhalten zu testen, wurden mehrere Zugriffsanfragen simultan in einem Ramp-Up-Test mit Apache JMeter abgeschickt. Mit diesem Test ist es möglich, die gesamten Aufwände durch das Zugriffskontrollsystem in einer Real-World-Umgebung zu bestimmen. Es ist zu beachten, dass für die Tests synthetische URIs verwendet wurden. Dies ist gefahrlos möglich, da die URIs auf Hash-Werte abgebildet werden und somit kein Unterschied bei der Identifikation der auszuwertenden Policies besteht. Tabelle 4.3 zeigt die Resultate des Ramp-Up-Tests. Die erste Zeile der Tabelle zeigt einen Referenzwert, der für den Vergleich herangezogen werden kann. Dieser Wert wurde gemessen ohne den Einsatz irgendeiner Zugriffskontrolle und bildet somit den Overhead durch Apache JMeter ab im Vergleich zu Tabelle 4.2.

Das Setup für die Tests wurde wie folgt festgelegt: Jeder Test wurde mit 100 Threads durchgeführt mit einer Ramp-Up-Periode von 1 Sekunde. Dies bedeutet, dass alle 10ms eine neue Zugriffsanfrage an das Zugriffskontrollsystem gesendet wurde. Dies entspricht wiederum 100 Zugriffsanfragen pro Sekunde. Für jedes Regelwerk wurde der Ramp-Up-Test mindestens 10 mal durchgeführt, so dass für jedes Regelwerk mindestens 1000 Anfragen gesendet wurden. Der RESTful Service wurde mittels JAX-RS [79, 24] erstellt, welches eine Java-Spezifikation zur Erstellung von RESTful Services ist. Das Zugriffskontrollsystem wurde als Singleton (vgl. [47]) instanziiert, um mehrfache Instanzen des gleichen System zu verhindern. Mehrfache Instanzen auf derselben Maschine hätten bei einer großen Anzahl an Ressourcen zu Speicherüberläufen geführt.

Es lässt sich erkennen, dass das Zugriffskontrollsystem nahezu keinen Einfluss auf die Verarbeitungszeit einer HTTP Anfrage hat. Der

wesentliche Anteil ist hierbei die Formulierung der Zugriffsanfrage inklusive der Extraktion der Attribute aus dem HTTP-Request. Das Verhalten für eine große Anzahl paralleler Anfragen bestätigt die in den ersten Tests gefundenen Ergebnisse.

4.11 Verwandte Arbeiten

RestACL unterscheidet sich gegenüber XACML grundlegend durch die Trennung von Domain und Policy Repository. XACML und andere Verfahren, welche Zugriffsberechtigungen in Baumstrukturen organisieren, müssen bei großen Datenmengen Methoden anwenden, um die effiziente Auswertung sicherzustellen. XEngine [94, 93] beispielsweise tut dies indem Attributwerte innerhalb des Regelwerks auf ein festes Intervall bestimmter Größe beschränkt werden. Hierdurch wird es möglich, die Zugriffsentscheidungen für alle Kombinationen der möglichen Werte vor dem Eintreffen einer Zugriffsanfrage zu berechnen. Mit Hilfe eines Entscheidungsbaums können die anwendbaren Regeln für bestimmte Attributwerte ermittelt werden. Die Zugriffsentscheidungen müssen dann noch leicht auswertbar gruppiert werden, so dass zum Zeitpunkt der Zugriffsanfrage keine Auswertung des eigentlichen Regelwerks mehr stattfinden muss, sondern nur noch in den bereits berechneten Ergebnissen die zugehörige Entscheidung gefunden werden muss. XEngine eliminiert hierdurch den kompositionellen Aspekt von XACML. Dieses Verfahren ist äußerst effizient hinsichtlich der Bearbeitungzeiten von Zugriffsanfragen. Wie bereits zuvor erwähnt werden hierdurch die Auswertungszeiten von der Komplexität des Regelwerks entkoppelt. Es entstehen konstante Antwortzeiten für Zugriffsanfragen, welche lediglich von der Anzahl der verwen-

deten Attribute abhängen. Gleichzeitig hat dieses Vorgehen jedoch auch einige Nachteile. Insbesondere ist das Verfahren nur in statischen Umgebungen sinnvoll, da das Aufstellen und Erweitern des Entscheidungsbaums eine aufwändige Operation darstellt und eine inkrementelle Erweiterung des PDD nicht beschrieben wird. Darüber hinaus geht durch dieses Verfahren Flexibilität und Ausdrucksstärke verloren. So können etwa keine Zeichenketten zur Laufzeit verglichen werden, was die Ausdruckstärke erheblich einschränkt. Schließlich wird in [121] ein extremer Speicherbedarf als Nachteil genannt.

Sollen bei einem auf einem Entscheidungsbaum basierenden Verfahren alle Möglichkeiten zur Formulierung von Zugriffsberechtigungen erhalten bleiben, muss der Entscheidungsbaum zur Laufzeit ausgewertet werden, so wie in [121] beschrieben. Bei der Verwendung von RestACL hingegen ist der Einsatz von Entscheidungsbäumen aufgrund der Trennung von Domain und Policy Repository nicht notwendig. Der kompositionelle Aspekt entfällt bei einem solchen Verfahren.

Das indexbasierte Verfahren hat vielmehr Ähnlichkeiten mit einem instanzbasierten Verfahren, dessen Grundgedanken in [137] beschrieben werden. Beim instanzbasierten Verfahren können Regelwerke individuell den zu schützenden Ressourcen zugewiesen werden. Es existiert somit nicht mehr nur ein allgemeingültiges Regelwerk, sondern pro zu schützender Ressource ein Regelwerk. Diese Regelwerke müssen beim Eintreffen einer Zugriffsanfrage geladen werden und können dann ausgewertet werden. So entstehen viele kleine Regelwerke deren Auswertung eine geringe Komplexität aufweist. Das bedeutet, die Regelwerke müssen beim Eintreffen der Zugriffsentscheidung geladen werden, um das Zugriffskontrollsystem zu instanziieren.

In [137] beschreiben die Autoren instanzbasierte Zugriffskontrollbäume mit einer festen Struktur bestehend aus Subjekt-, Aktions- und

Ressourcenbedingungen in Form von Data Objects. Diese werden in einem Baum für ausschließlich positive Zugriffsentscheidungen organisiert. Negative Zugriffsentscheidungen werden nicht berücksichtigt.

Eine feste Struktur zur Teilung der Zugriffsberechtigungen wird bei RestACL durch die Ressourcenorientierung von RESTful Services intuitiv gegeben, so dass keine zusätzliche Struktur vorgegeben werden muss. Anhand der Ressourcenadresse kann vorselektiert werden, welche Zugriffsregeln ausgewertet werden müssen. Anders als bei einem instanzbasierten Verfahren ist es nicht zwingend notwendig Regelwerke zur Laufzeit zu laden. Ein RestACL-System kann vollständig in den Hauptspeicher geladen werden, so dass beim Eintreffen einer Zugriffsanfrage direkt eine Auswertung stattfinden kann. Technisch wäre eine solche Lösung auch mit einem auf XACML basierenden System denkbar. Allerdings ist der Speicherbedarf für ein solches System erheblich höher als bei RestACL, da XACML als kompositioneller Ansatz konzipiert wurde und ein instanzbasiertes Verfahren eine Optimierung für große Datenmengen in der Cloud darstellt. RestACL hingegen wurde jedoch derart konzipiert, dass der indexbasierte Ansatz elementarer Bestandteil des Systems ist.

4.12 Zusammenfassung

In diesem Kapitel wurden die folgenden wissenschaftlichen Beiträge beschrieben:

- **Indexbasierte Zugriffskontrollmechanismen:** Durch die Verwendung von eindeutigen Schlüsseln beziehungsweise eines Schlüsselattributs, lässt sich attributbasierte Zugriffskontrolle

sehr effizient gestalten, da der kompositionelle Aspekt von generischen Ansätzen entfällt und effiziente Datenstrukturen, wie etwa Hash-Tables, eingesetzt werden können.

- **Sprache und Architektur:** Es wurde die Semantik von RestACL in abstrakter und konkreter Syntax beschrieben. Die Interpretationsverfahren zu dieser Syntax wurden vorgestellt. Architekturkonzepte demonstrieren, wie der Zugriffskontrollmechanismus in RESTful Services integriert werden kann und wie einzelne Komponenten kommunizieren.

- **Analyse und Untersuchung:** RestACL wurde mathematisch analysiert, so wie bereits zuvor XACML analysiert wurde. In experimentellen Untersuchungen konnten die unterschiedlichen Eigenschaften der beiden Mechanismen nachgewiesen werden.

Es wurde ein neuartiger Zugriffskontrollmechanismus eingeführt, welcher für den Einsatz bei RESTful Services konzipiert ist. Dazu wurden zunächst die Anforderungen an ein Zugriffskontrollsystem definiert. Der Ansatz ist inspiriert von XACML und automatisiert Optimierungsansätze, welche gemäß Kapitel 3 beim Einsatz von XACML für RESTful Services berücksichtigt werden sollten. Regelwerke werden identisch zur Ressourcenstruktur aufgebaut, was einen intuitiven Weg ermöglicht, attributbasierte Zugriffskontrolle für RESTful Services umzusetzen.

Wesentliches Merkmal des Mechanismus ist die Trennung zwischen Domain und Policy Repository. Diese Trennung ermöglicht Interpretern eine effizientere Auswertung von Zugriffsanfragen durchzuführen. Es können indexbasierte Datenstrukturen eingesetzt werden. Hierdurch wird die Identifikation des Teils des Regelwerks, der für

das Berechnen der Zugriffsentscheidung ausgewertet werden muss, erheblich beschleunigt und die Auswertung dieses Teils vom übrigen Regelwerk entkoppelt.

Der Evaluierungsalgorithmus ist einfach gestaltet, da er auf simplen Suchoperationen und Priorisierung beruht. Hiermit werden Seiteneffekte vermieden, die dazu führen können, dass Evaluationszeiten sowie Aufwände für Wartung und Fehlerbehebung anwachsen. Die Verwendung von Effekten ermöglicht es White und Black Listing sowie Kombinationen davon umzusetzen. Die eingesetzten Formate - wie das weitverbreitete JSON - ermöglichen eine einfache Integration mit existierenden Modellierungswerkzeugen für RESTful Services.

Die Sprache wurde einer Analyse unterzogen und in experimentellen Tests mit XACML verglichen. RestACL zeigt verbesserte Performanz und einen reduzierten Speicherverbrauch im Vergleich zum Einsatz einer funktionell identischen Implementierung mit XACML.

Zukünftige Arbeiten können sich detailliert noch offenen Fragestellungen widmen. So kann etwa untersucht werden, in wie weit sich die Flexibilität und Wartbarkeit von Regelwerken durch die Verwendung von lokalen Prioritäten verbessern lässt. Lokale Prioritäten eröffnen die Möglichkeit, Policies in unterschiedlicher Gewichtung wiederzuverwenden und Aufwände bei der Verwaltung von Prioritäten zu minimieren.

Eine weitere Möglichkeit für zukünftige Untersuchungen ist die Integration mit Entwicklungswerkzeugen für RESTful Services. In [112] wurde die Integration von RestACL und Swagger prototypisch getestet. Es wurde gezeigt, dass die Erstellung einer Beschreibung eines RESTful Service mit Swagger auch dazu verwendet werden kann, ein zugehöriges Regelwerk zu generieren. Basierend auf diesen Ergebnissen können weitere Detailuntersuchungen erfolgen.

Kapitel 5

Vergleich von RestACL und XACML

In diesem Kapitel wird ein Vergleich von XACML und RestACL mit Hilfe von formalen Methoden vorgenommen. Die Ergebnisse wurden in [68] publiziert.

5.1 Einleitung

Es werden Transformationsvorschriften beschrieben, welche es erlauben, ein RestACL-Regelwerk in ein XACML-Regelwerk umzuwandeln und umgekehrt. Diese formellen Transformationsvorschriften untermauern die Ausdrucksstärke beim Formulieren von RestACL-Regelwerken, da sie demonstrieren, dass die Ausdrucksstärke der

beiden Sprachen äquivalent ist. Ebenfalls werden die Korrektheit und die Vollständigkeit von RestACL durch die vorgestellten Transformationsvorschriften belegt.

- Die Korrektheit bezieht sich auf das Eingabe-Ausgabe-Verhalten. RestACL ist funktional korrekt, wenn zu jeder Kombination von Eingabe-Parametern der zu erwartende Ausgabeparameter berechnet wird. Die Eigenschaft der Korrektheit kann durch eine Transformationsvorschrift nach XACML als etabliertem Standard bewiesen werden. Sind alle XACML-Regelwerke korrekt und sind die Regelwerke von RestACL eine Teilmenge der Regelwerke von XACML, so sind auch alle RestACL-Regelwerke korrekt.
- Die Vollständigkeit einer Sprache bezieht sich auf deren Ausdruckstärke. Eine Sprache ist vollständig, wenn sie den Gegenstand ausdrücken kann, für den sie entworfen wurde. Diese Eigenschaft kann durch eine umgekehrte Transformation von XACML zu RestACL belegt werden. Ist es möglich alle attributbasierten Regelwerke in XACML zu formulieren und sind die Regelwerke von XACML eine Teilmenge der Regelwerke von RestACL, so ist es auch möglich, alle attributbasierten Regelwerke mit RestACL zu formulieren.

Ohne Beschränkung der Allgemeinheit werden dabei die Kernelemente von XACML und RestACL betrachtet. Das bedeutet, dass die XACML-Elemente Obligation, Advice und Condition nicht betrachtet werden. Obligations und Advices sind Konzepte, welche nach der Entscheidungsfindung zum Tragen kommen. Deshalb können sie uneingeschränkt übernommen werden und eine detaillierte Analyse

ist nicht notwendig. Conditions sind eine zusätzliche Möglichkeit in XACML Attributbedingungen auszudrücken. Sie unterscheiden sich im grundsätzlichen Aufbau nicht von einem Target. XACML-Targets unterliegen jedoch einigen Restriktionen, die bei Conditions nicht gegeben sind. So dürfen beispielsweise mehrere Attributwerte aus der Zugriffsanfrage in einer Condition verglichen werden. Bei RestACL wird auf die Betrachtung von ParameterizedAccess-Elementen und Templates verzichtet. ParameterizedAccess-Elemente sind analog zu Access-Elementen zu betrachten, die einer gesonderten Form der Ressourcenadresse zugeordnet werden. Templates sind lediglich eine Vereinfachung zur Formulierung von Regelwerken und beschreiben Ressourcen mit gleichartiger Adresse.

5.2 Formelle Beschreibung von XACML

XACML nutzt Policy Sets, Policies und Rules, um ein Regelwerk abzubilden. Jedes dieser Elemente hat ein Target, welches über eine Menge von Attributbedingungen festlegt, ob das Element auf eine Zugriffsanfrage anwendbar ist. Policy Sets und Policies verwenden einen Combining Algorithm, welcher zur finalen Entscheidungsfindung beiträgt, falls mehrere untergeordnete Elemente anwendbar sind. Rules haben einen Effect, welcher eine Teilentscheidung ausdrückt.

Für den formellen Vergleich von XACML und RestACL wird eine abstrakte Notation eingeführt. Bereits in Kapitel 4 wurde eine Definition für ein Attribut eingeführt, welche an dieser Stelle noch einmal wiederholt werden soll.

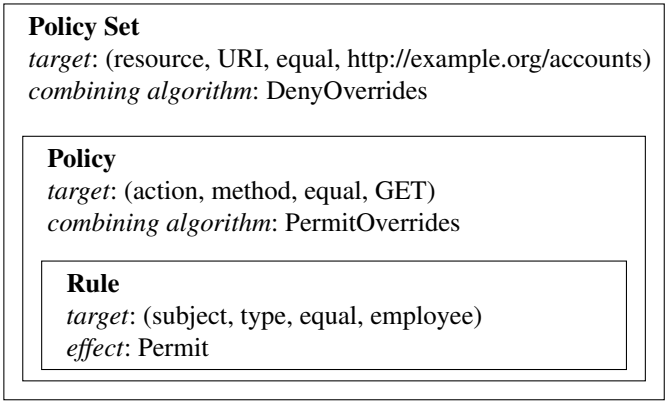

Abbildung 5.1: XACML-Regelwerk zum Schutz einer Ressource

Definition (Attribute): ein Attribut wird als ein Tripel $a := (c, d, v)$ notiert, bestehend aus einer Category c, einem Designator d und einem Value v. Category, Designator und Value haben alle einen Datentyp wie beispielsweise Integer oder Zeichenketten. Weiter sei A eine Menge von Attributen.

Ebenfalls wurde bereits in Kapitel 4 eine Definition für eine Attributbedingung für RestACL eingeführt. XACML ist bei der Formulierung von Attributbedingungen in Targets restriktiver als die vorherige Definition es zulässt. So dürfen in einem Target nicht mehrere Attributwerte aus der Zugriffsanfrage verglichen werden, sondern es darf lediglich ein Attribut aus der Zugriffsanfrage mit einem statischen Wert abgeglichen werden. Deshalb gilt nicht wie in der vorherigen

Definition zur Attributbedingung $ac : F \times A^n \rightarrow \{true, false\}$, sondern $ac : F \times A^2 \rightarrow \{true, false\}$.

Notation (Attributbedingung): eine Attributbedingung wird als Quadrupel $ac := (c, d, f, v)$ notiert, bestehend aus einer Category c und einem Designator d, welche dazu verwendet werden, den Value eines Attributs aus der Zugriffsanfrage mit Hilfe einer Vergleichsfunktion $f \in F$ mit einem statischen Value v abzugleichen. Weiter sei AC eine Menge von Attributbedingungen.

Notation (XACML-Target): ein Target wird als $t := (ac_1 \circ ... \circ ac_n)$ notiert, mit $ac_i \in AC$, $i \in 1, ..., n$ und $\circ$ als Repräsentation einer logischen Konjunktion oder Disjunktion. Weiter sei T eine Menge von Targets.

Notation (XACML-Rule): eine Rule wird als $r := (t, e)$ notiert, mit $t \in T$ und e als resultierendem Effect $e \in \{Permit, Deny\}$.

Notation (XACML-Policy): eine Policy wird als Tripel $p := (R, t, l)$ notiert, bestehend aus R als Liste von Rules, welche enthalten sind in p, $t \in T$ und l als Combining Algorithm.

Notation (XACML-Policy Set): ein Policy Set wird als Quadrupel $ps := (PS, P, t, l)$ notiert, bestehend aus PS als Liste von ps untergeordneten Policy Sets, P als Liste von ps untergeordneten Policies, $t \in T$ und l als Combining Algorithm.

Mit diesen Definitionen kann das in Abbildung 5.1 gezeigte Regelwerk beschrieben werden als:

$$
\begin{array}{lcl}
t_{ps_1} & = & ((resource, URI, equal, example.org/accounts)) \\
t_{p_1} & = & ((action, method, equal, GET)) \\
t_{r_1} & = & ((subject, type, equal, Employee)) \\
ps_1 & = & (\emptyset, \{p_1\}, t_{ps_1}, DenyOverrides) \\
p_1 & = & (\{r_1\}, t_{p_1}, PermitOverrides) \\
r_1 & = & (t_{r_1}, Permit)
\end{array}
$$

Man kann sehen, dass eine Rule auf eine Zugriffsanfrage anwendbar wird, wenn die Targets der Rule selbst, der übergeordneten Policy und allen übergeordneten Policy Sets anwendbar sind. Beispielsweise wird r_1 anwendbar auf eine Zugriffsanfrage, wenn die Targets t_{r_1}, t_{p_1} und t_{ps_1} auf die Zugriffsanfrage anwendbar sind.

Notation (Target Path): ein Target Path wird als $t'_r := \{t_r \wedge t_{p_r} \wedge t_{ps_{1_r}} \wedge ... \wedge t_{ps_{n_r}}\}$ notiert, bestehend aus einer Menge von Targets, welche anwendbar sein müssen, damit eine XACML-Rule r anwendbar auf eine Zugriffsanfrage ist. Dabei ist t_r das Target der Rule r, t_{p_r} das Target der Policy p_r, welche r beinhalten und $t_{ps_{1_r}} ... t_{ps_{n_r}}$ als die Targets aller übergeordneten Policy Sets von p_r.

5.3 Formelle Beschreibung von RestACL

Neben der formellen Beschreibung von XACML, muss auch RestACL formell beschrieben werrden. Zur Abgrenzung von XACML werden griechische Buchstaben verwendet. Ohne Beschränkung der Allgemeinheit wird dabei auf die Betrachtung von Template- und

ParameterizedAccess-Elementen verzichtet, da diese lediglich Spezialfälle der Ressourcenadresse abbilden.

Notation (RestACL-Policy): eine Policy wird als Tripel $\rho := (\varepsilon, \pi, (ac_1 \circ ... \circ ac_n))$ notiert, mit $\varepsilon \in \{Permit, Deny\}$ als Effekt, $\pi \in \mathbb{N}$ als Priorität und $\{ac_1, ..., ac_n\}$ als Menge von Attibutbedingungen analog zu XACML.

Notation (RestACL-Access): ein Access-Element besteht aus zwei Mengen und wird als ein Paar $\alpha := (\{\mu_1, ...\mu_n\}, \{\rho_1, ..., \rho_m\})$ notiert, mit μ_i als Methode der einheitlichen Schnittstelle, $i \in 1, ..., n$ und ρ_j als RestACL-Policy, $j \in 1, ..., m$.

Notation (RestACL-Resource): ein Resource-Element besitzt zwei Mengen und wird als Tripel $\sigma := (\phi, \{\alpha_1, ..., \alpha_n\}, \{\sigma'_1, ..., \sigma'_m\})$ notiert, mit ϕ als Pfad einer Ressource (beispielsweise */accounts/123*), α_i als Access-Elemente, $i \in 1...n$ und σ'_j als untergeordnete Ressourcen der Ressource σ, $j \in 1...m$

Notation (RestACL-Domain): eine Domain wird als Paar $\varphi := (\upsilon, \{\sigma_1, ..., \sigma_n\})$ notiert mit υ als Host, für den das Zugriffskontrollsystem verantwortlich ist und σ_i als Ressourcen in dieser Domain, $i \in 1...n$.

Abbildung 5.2 zeigt die RestACL Domain, die in diesem Kapitel als Beispiel dient. Diese Domain kann mit den eingeführten Notationen geschrieben werden als:

$$
\begin{array}{lll}
\alpha_1 & = & (\{GET, PUT\}, \{P1, P2, P3\}) \\
\alpha_2 & = & (\{DELETE\}, \{P4\}) \\
\alpha_3 & = & (\{POST\}, \{P5\}) \\
\sigma_1 & = & (/accounts, \{\alpha_1\}, \sigma_2) \\
\sigma_2 & = & (/123, \{\alpha_2\}, \emptyset) \\
\sigma_3 & = & (/transfers, \{\alpha_3\}, \emptyset) \\
\varphi & = & (example.org, \{\sigma_2, \sigma_3\})
\end{array}
$$

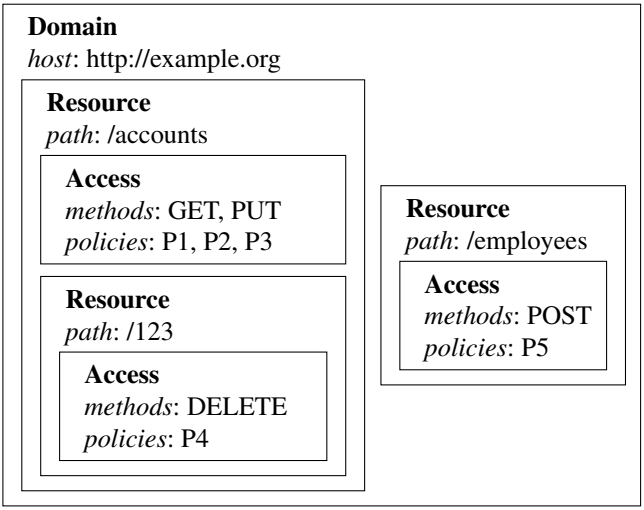

Abbildung 5.2: RestACL Domain

5.4 Korrektheit von RestACL

In diesem Absatz wird beschrieben, wie ein in RestACL geschriebenes Regelwerk in ein XACML-Regelwerk transformiert werden kann. Die Transformation dient auch dazu einen zweckmäßigen und praktischen Weg aufzuzeigen, um die Semantik von RestACL zu formulieren. Wie in vorherigen Absätzen zu sehen war, werden bei RestACL individuelle Zugriffsberechtigungen auf Ressourcen und Methoden abgebildet. Um diese Abbildung in XACML zu ermöglichen, wird im ersten Schritt der Transformation ein Wurzel-Policy-Set erstellt, welches den Host der RestACL-Domain kontrolliert. Darüber hinaus werden für jede Ressource untergeordnete Policy Sets erstellt. Da Ressourcenadressen eindeutig sind, kann der Combining Algorithm *FirstApplicable* in diesen Policy Sets verwendet werden. Die Adressen von hierarchisch angeordneten Ressourcen werden bei der Erstellung der Policy Sets aufgelöst. Es wird also immer der vollständige *path* der URI verwendet.

Präziser ausgedrückt werden für alle Ressourcen σ_i, welche in der RestACL Domain φ hinterlegt sind, Policy Sets ps_{σ_i} wie folgt erstellt:

$$ps_{\sigma_i} := (\emptyset, P_{\sigma_i}, t_{ps_{\sigma_i}}, FirstApplicable)$$

P_{σ_i} wird dabei in einem zweiten Schritt berechnet. Das Target $t_{ps_{\sigma_i}}$ hat exakt eine Attributbedingung, welche formuliert werden kann als:

$$t_{ps_{\sigma_i}} = ((resource, path, \phi_{\sigma_i}, equal))$$

Für die RestACL-Domain φ wird ein Wurzel-Policy-Set ps_{root} wie folgt erstellt:

$$ps_{root} := (\cup_i ps_{\sigma_i}, \emptyset, t_{ps_{root}}, FirstApplicable)$$

Wie zuvor enthält auch dieses target $t_{ps_{root}}$ lediglich eine Attributbedingung:

$$t_{ps_{root}} = ((domain, host, \upsilon_{\varphi}, equal))$$

Beispielsweise können die XACML-Policy Sets für das RestACL-Regelwerk aus Abbildung 5.2 geschrieben werden als:

$$\begin{array}{lcl} t_{ps_{root}} & = & ((resource, host, equal, example.org)) \\ ps_{root} & = & (\cup_i ps_{\sigma_i}, \emptyset, t_{ps_{root}}, FirstApplicable) \\ \\ t_{ps_{\sigma_1}} & = & ((resource, path, equal, /accounts)) \\ ps_{\sigma_1} & = & (\emptyset, P_{\sigma_1}, t_{ps_{\sigma_1}}, FirstApplicable) \\ \\ t_{ps_{\sigma_2}} & = & ((resource, path, equal, /accounts/123)) \\ ps_{\sigma_2} & = & (\emptyset, P_{\sigma_2}, t_{ps_{\sigma_2}}, FirstApplicable) \\ \\ t_{ps_{\sigma_3}} & = & ((resource, path, equal, /transfers)) \\ ps_{\sigma_3} & = & (\emptyset, P_{\sigma_3}, t_{ps_{\sigma_3}}, FirstApplicable) \end{array}$$

Im zweiten Schritt ist es notwendig, die Access-Elemente von Ressourcen σ_i innerhalb eines RestACL-Regelwerks auf XACML-Policies P_{σ_i} zu übersetzen. Hierfür wird für jede Methode μ_i in einem Access-Element α_j eine XACML-Policy nach folgender Vorschrift erstellt:

$$p_{i_{\alpha_j}} := (R_{i_{\alpha_j}}, t_{p_{i_{\alpha_j}}}, FirstApplicable)$$

Das Target von $p_{i_{\alpha_j}}$ hat dabei erneut exakt eine Attributbedingung, welche geschrieben werden kann als:

$$t_{p_{i_{\alpha_j}}} = ((action, method, \mu_{i_{\alpha_j}}, equal))$$

Beispielsweise können die XACML-Policies für die Kontenliste (/accounts) im RestACL-Regelwerk aus Abbildung 5.2 geschrieben werden als:

$$
\begin{aligned}
t_{p_{1\alpha_1}} &= (action, method, equal, GET) \\
t_{p_{2\alpha_1}} &= (action, method, equal, PUT) \\
p_{1\alpha_1} &= (R_{1\alpha_1}, t_{p_{1\alpha_1}}, FirstApplicable) \\
p_{2\alpha_1} &= (R_{2\alpha_1}, t_{p_{2\alpha_1}}, FirstApplicable)
\end{aligned}
$$

Diese XACML-Policies werden den im ersten Schritt erstellten Policy-Sets untergeordnet:

$$P_{\sigma_1} = \{p_{1\alpha_1}, p_{2\alpha_1}\}$$

Im dritten Schritt wird schließlich jeweils eine XACML-Rule r_{ρ_i} für jede RestACL-Policy ρ_i, welche in den Access-Elementen α_j aufgeführt ist, erstellt. Dies geschieht wie folgt:

$$r_{\rho_i} := ((a_{1_{\rho_i}} \circ ... \circ a_{n_{\rho_i}}), \varepsilon_{\rho_i})$$

Abbildung 5.3 zeigt ein Beispiel für eine RestACL-Policy $P1$. Das Target für eine XACML-Rule r_1 aus dieser RestACL-Policy kann geschrieben werden als:

$$
\begin{aligned}
ac_{1_{P1}} &= (subject, type, equal, employee) \\
ac_{2_{P1}} &= (resource, type, equal, account) \\
t_{r_1} &= ac_{1_{P1}} \wedge ac_{2_{P1}}
\end{aligned}
$$

Die neuen XACML-Rules werden in absteigender Ordnung der Prioritäten der zugehörigen RestACL-Policy zu den jeweiligen XACML-Policies hinzugefügt. Beispielsweise könnten die Policies $P1, P2$ und $P3$ zum Schutz der Kontenliste die Prioritäten $\pi_{\rho_{P1}} = 1, \pi_{\rho_{P2}} = 2$ und

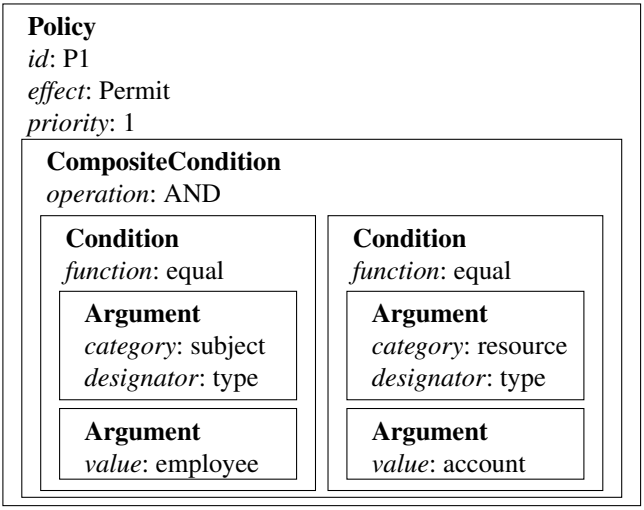

Abbildung 5.3: RestACL Policy

$\pi_{\rho_{P3}} = 3$ haben. Dadurch kann schließlich bestimmt werden, dass $R_{1_{\alpha_1}} = \{r_{P1}, r_{P2}, r_{P3}\}$ und $R_{2_{\alpha_1}} = \{r_{P1}, r_{P2}, r_{P3}\}$.

5.5 Vollständigkeit von RestACL

In diesem Abschnitt wird beschrieben, wie XACML-Policy Sets, Policies und Rules für eine RESTful API in ein RestACL-Regelwerk transformiert werden können, inklusive einer Ressourcenhierarchie

und priorisierten Policies für jede Ressource. Mit Hilfe der Transformation wird gezeigt, dass RestACL die gleiche Ausdrucksstärke hat wie XACML.

Überführung von Rules Eine Anfrage an einen RESTful Service beinhaltet immer die Adresse der Ressource (die URI) und die Zugriffsmethode mit der zugegriffen werden soll. Im Weiteren werden diese beiden Attributbedingungen, welche die Adresse und die Zugriffsmethode betreffen, als ac_{res} beziehungsweise ac_{met} bezeichnet. Um eine RestACL-Policy aus einer XACML-Rule zu erstellen, müssen alle Attributbedingungen entlang des Target Path t'_r betrachtet werden. Alle diese Bedingungen werden Teil der Attributbedingungen der RestACL-Policy mit Ausnahme von ac_{res} und ac_{met}.

Während ac_{met} dazu verwendet wird ein Access-Element zu erstellen, wird ac_{res} dazu benötigt, die Hierarchie von Ressourcen innerhalb der Domain abzubilden.

Ein formeller Weg diese Vorschrift zu beschreiben ist der Folgende: für jede XACML-Rule r im gesamten XACML-Regelwerk, wird eine passende RestACL-Policy ρ_r erstellt. Da die Targets innerhalb des Target Path logisch konjugiert sind und logische Konjunktionen kommutativ sind, kann geschrieben werden:

$$t'_r = ((ac_1 \circ ... \circ ac_n) \wedge ac_{res} \wedge ac_{met})$$

In einem ersten Schritt kann ein abgeleiteter Target Path t''_r erstellt werden, welcher die beiden Bedingungen ac_{res} und ac_{met} nicht beinhaltet:

$$t''_r = (ac_1 \circ ... \circ ac_n)$$

Danach wird eine RestACL-Policy ρ_r wie folgt erstellt:

$$\rho_r := (e_r, \pi_{\rho_r}, t_r'')$$

Dies bedeutet, dass die verbleibenden Attributbedingungen in t_r'' aus dem Target Path in die RestACL-Policy übernommen werden. Ebenfalls wird der Effekt der XACML-Rule übernommen. Die Priorität π_{ρ_r} der RestACL-Policy wird in einem späteren Schritt berechnet.

In einem zweiten Schritt wird ein Access-Element α_r für die RestACL-Domain erstellt. Dieses wird aus ac_{met} wie folgt abgeleitet:

$$\alpha_r := (\{\mu_1, ..., \mu_n\}_{ac_{met}}, \{\rho_r\})$$

Die Werte der Attributbedingung ac_{met} werden also übersetzt auf die Liste von Zugriffsmethoden, beispielweise:

$$ac_{met} = (action, method, POST, equal) \rightarrow \{\mu_1, ..., \mu_n\}_{ac_{met}} = \{POST\}$$

Sollte der Target Path der XACML-Rule keinerlei Bedingungen hinsichtlich der Zugriffsmethoden beinhalten, müssen alle Methoden der einheitlichen Schnittstelle in die Liste der Methoden des Access-Elements von α_r aufgenommen werden. In diesem Fall ist ac_{met} eine immer erfüllte Bedingung. Die neue RestACL-Policy ρ_r wird dann der Policy-Liste von α_r hinzugefügt.

Im dritten Schritt wird das Resource-Element σ_r aus der Attributbedingung ac_{res} abgeleitet. Der Pfad ϕ_r ist nach der Definition von ac_{res} durch die Attributbedingung festgelegt. Wenn also eine Ressourcenadresse die Bedingung ac_{res} erfüllt, wird σ das Access-Element α_r der Liste von Access-Elementen von σ_r hinzugefügt. Deckt die Attributbedingung ac_{res} die Pfade von mehreren Ressourcen ab, müssen

dementsprechend mehrere Resource-Elemente erstellt oder identifiziert werden und das Access-Element α_r muss zu all diesen hinzugefügt werden. Existiert keine Einschränkung hinsichtlich der Ressourcenadresse im Target Path der XACML-Rule, so muss α_r zu jedem Resource-Element in der RestACL-Domain hinzugefügt werden. Beispielsweise könnte der Administrator eines Systems unbeschränkten Zugriff auf alle Ressourcen des Systems haben, so dass Zugriffsbedingungen existieren, die keinen Bezug zu bestimmten Ressourcen haben.

Mit Hilfe dieser drei Schritte kann das XACML-Regelwerk aus Abbildung 5.1 wie folgt transformiert werden. Der Target Path der XACML-Rule besteht aus drei Targets: dem Target des Policy Sets, dem Target der Policy und dem Target der Rule. Jedes Target enthält jeweils eine Attributbedingung. Im ersten Schritt kann die Attributbedingung des Policy Sets als ac_{res} identifiziert werden und die Attributbedingung der Policy als ac_{met}. Deshalb gilt:

$$t_r'' = (ac_r) = (subject, type, equal, employee)$$

Da die ursprüngliche XACML-Rule den Effekt *permit* hat, wird eine neue RestACL-Policy mit dem gleichen Effekt erstellt:

$$\rho_r := (permit, \pi_{\rho_r}, (subject, type, equal, employee))$$

Die Berechnung der Priorität erfolgt gemäß des nächsten Abschnitts. Im zweiten Schritt wird ein Access-Element aus a_{met} erstellt. Da die Bedingung lediglich den GET-Zugriff betrifft, kann das Access-Element erstellt werden als:

$$\alpha_r := (GET, \{\rho_r\})$$

Schließlich wird im dritten Schritt ein Resource-Element aus ac_{res} abgeleitet, indem die Bedingung an den Pfad der Ressource ausgewertet und der RestACL-Domain hinzugefügt wird:

$$\sigma_r := (/accounts, \{\alpha_r\}, \emptyset)$$

5.6 Berechnung von Prioritäten

Um die Transformation abschließen zu können, müssen in einem letzten Schritt noch die Prioritäten der RestACL-Policies berechnet werden. Diese hängen von den im XACML-Regelwerk verwendeten Combining Algorithm ab. Beispielsweise müssen für eine XACML-Policy mit dem Combining Algorithm *PermitOverrides* und mehreren untergeordneten XACML-Rules, die Rules eine höhere Priorität bekommen, welche auch den Effekt *Permit* haben. Analoges gilt für den Combining Algorithm *DenyOverrides*. Beim Combining Algorithm *FirstApplicable* müssen Prioritäten in absteigender Reihenfolge vergeben werden. Der Combining Algorithm *OnlyOneApplicable* funktioniert nach der gleichen Logik wie *FirstApplicable* mit der Ausnahme, dass im Falle von mehreren anwendbaren Resultaten ein unbestimmtes Ergebnis zurück geliefert wird.

Für RestACL ist der Einsatz von Prioritäten vorgesehen, da diese ein einfach zu verstehendes Konzept darstellen. Außerdem wurde in [94, 93] gezeigt, dass der Combining Algorithm *FirstApplicable* im Mittel schnellere Antwortzeiten aufweist und Prioritäten

analog zu diesem Combining Algorithm funktionieren. Die Anwendung des Combining-Prinzips kann sehr komplex werden, wenn es eine große Menge von Policy Sets und Policies mit unterschiedlichen Algorithmen gibt. Deshalb ist auch die Berechnung von Prioritäten sehr komplex. Die Auswertung von XACML-Targets wird in einem Top-Down-Ansatz durchgeführt. Die Entscheidungsfindung hingegen wird dann in einem Bottom-Up-Prozess berechnet. Deshalb ist es bei der Bestimmung von Prioritäten wichtig zu beachten, welche Effekte von anwendbaren Rules sich durchsetzen und bis an die Wurzel des XACML-Baums durchgereicht werden.

Abbildung 5.4 zeigt ein XACML-Regelwerk mit verschiedenen Combining Algorithm. Im Abschnitt zuvor wurde beschrieben, dass ein solches Regelwerk in vier RestACL-Policies $\rho_{r_1}, ..., \rho_{r_4}$ transformiert wird. Für jede XACML-Rule entsteht eine RestACL-Policy. Der Combining Algorithm von p_1 zeigt, dass die Priorität von ρ_{r_1} größer sein muss als die Priorität von ρ_{r_2}, da sich der Effekt von r_1 gegen den Effekt von r_2 durchsetzt, wenn beide Rules anwendbar sind.

Deshalb wird als erstes $\pi_{\rho_{r_1}} = 1$ gesetzt und $\pi_{\rho_{r_2}} = 0$. Analog kann aus dem Combining Algorithm von p_2 abgeleitet werden, dass $\pi_{\rho_{r_3}} = 1$ und $\pi_{\rho_{r_4}} = 0$ gesetzt werden muss. Aus dem Combining Algorithm von ps_1 können zwei weitere Relationen abgeleitet werden. Zunächst kann abgeleitet werden, dass die Priorität von ρ_{r_2} größer sein muss als die Priorität von ρ_{r_3}, da sich ρ_{r_2} durchsetzt, wenn r_1 nicht anwendbar ist. Da r_1 nicht anwendbar sein darf damit r_2 den resultierenden Effekt bestimmt, wird eine weitere RestACL-Policy benötigt. Diese muss die Attributbedingung von t''_{r_2} enthalten und die negierten Attributbedingung von t_{r_1}. Die Policy wird als $\rho_{\overline{r_1} r_2}$ notiert und bekommt die Priorität $\pi_{\rho_{\overline{r_1} r_2}} = 2$. Die Priorität muss größer sein, da die Regel von einem übergeordneten Policy Set abgeleitet wurde.

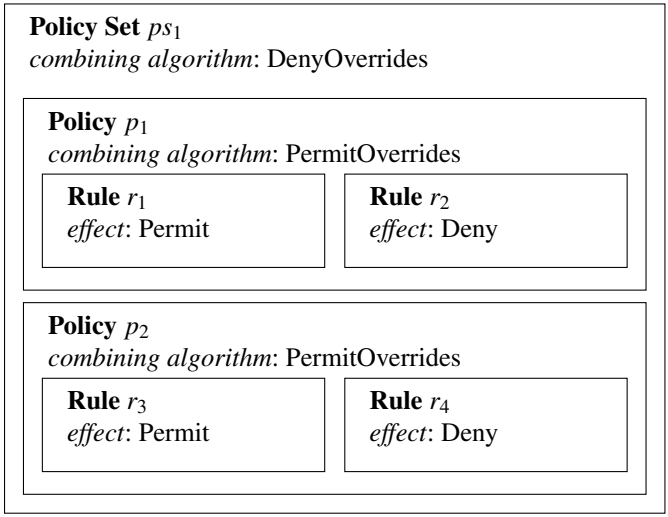

Abbildung 5.4: Verschiedene Combining Algorithm und Effects

Die zweite Relation sagt, dass die Priorität von ρ_{r_4} größer sein muss als die Priorität von ρ_{r_1}, wenn r_3 nicht anwendbar ist. Auch hier ist eine zusätzliche RestACL-Policy notwendig, welche die Attributbedingungen von t''_{r_4} und die negierte Attributbedingungen von t_{r_3} beinhaltet. Die Priorität wird zugewiesen $\pi_{\rho_{\overline{r_3}r_4}} = 2$. Da die neuen Policies von ps_1 abgeleitet werden, bekommen sie auch den dominierenden Effekt von ps_1, in diesem Fall *Deny*. Müssten neue Policies erstellt werden und wäre *PermitOverrides* der Combining Algorithm, so würden die neuen Policies den Effekt *Permit* bekommen. Im Fall von *FirstAppi-*

cable sind keine neuen Policies notwendig. Hier werden Prioritäten anhand der Reihenfolge der untergeordneten Elemente angepasst.

Im Folgenden wird ein formeller Weg beschrieben, wie die Prioritäten berechnet werden können. Der Effekt *Permit* wird notiert als *P*, der Effekt *Deny* als *D*, der Combining Algorithm *PermitOverrides* als *PO*, der Combining Algorithm *DenyOverrides* als *DO* und der Combining Algorithm *FirstApplicable* als *FA*. Initial wird allen RestACL-Policies die Priorität 0 zugewiesen. Danach werden die Prioritäten der RestACL-Policies basierend auf den Combining Algorithm der XACML-Policies erhöht.

$$\forall r \in p : \pi_{\rho_r} = \begin{cases} 1 & \text{if } e_r = P \wedge l_{p_r} = PO \\ 1 & \text{if } e_r = D \wedge l_{p_r} = DO \\ pos_p^{-1}(r) & \text{if } l_{p_r} = FA \end{cases}$$

mit $pos_p^{-1}(r)$ als Funktion, die die Position von r in inverser Anordnung zurückliefert. Beispielsweise liefert $pos_p^{-1}(r)$ bei insgesamt 3 XACML-Rules die Werte 2, 1, 0 für die erste, zweite und dritte XACML-Rule.

Danach werden in rekursiver Vorgehensweise neue RestACL-Policies erstellt und Prioritäten erhöht. Dies geschieht solange bis der Wurzelknoten des XACML-Regelwerks erreicht und verarbeitet wurde. Es sei π_{max} die höchste Priorität aller RestACL-Policies hinsichtlich eines XACML-Policy-Sets zu Beginn einer Iteration der Prioritätenberechnung für *ps*. Neue RestACL-Policies werden nach der folgenden Vorschrift erstellt:

$$\rho_{new} = \rho_{r_i \overline{r_j} \ldots \overline{r_k}}$$

für alle $r_j ... r_k \in SUB_{ps}$ mit SUB_{ps} als Menge von XACML-Rules in untergeordneten Policies von ps. Außerdem müssen $r_j ... r_k$ in gleichen untergeordneten Policy Sets oder Policies sein wie r_i. Die RestACL-Policies müssen erstellt werden, wenn entweder die Bedingung (5.1) oder (5.2) und gleichzeitig die Bedingung (5.3) erfüllt sind:

$$l_{ps} = PO \wedge e_{r_i} = P \wedge \forall e_x \in \{e_{r_j}, ..., e_{r_k}\} : e_x = D \tag{5.1}$$

$$l_{ps} = DO \wedge e_{r_i} = D \wedge \forall e_x \in \{e_{r_j}, ..., e_{r_k}\} : e_x = P \tag{5.2}$$

$$\forall \pi_{\rho_{r_x}} \in \{\pi_{\rho_{r_j}, ..., \pi_{\rho_{r_k}}}\} : \pi_{\rho_{r_x}} > \pi_{\rho_{r_i}} \tag{5.3}$$

Neu erstellte RestACL-Policies erhalten die Priorität:

$$\pi_{\rho_{new}} = \pi_{max} + 1$$

Betrachtet man das eingehende Beispiel erneut, so hat r_2 den Effect *Deny* und ps_1 den Combining Algorithm *DenyOverrides*. Nun müssen alle ps_1 untergeordneten Rules betrachtet werden, welche den Effect *Permit* haben. Wenn eine oder mehrere Rules verhindern, dass r_2 die dominierende Rule wird, so muss eine neue Policy erstellt werden, welche die Bedingungen von r_2 um die negierte Bedingung der jeweiligen Rule ergänzt. Im eingehenden Beispiel ist r_1 im gleichen Zweig wie r_2. Bedingung (5.2) ist also erfüllt. Die XACML-Rule verhindert, dass r_2 die Zugriffsentscheidung bestimmt. Dies liegt darin begründet, dass p_1 erfordert, dass r_1 eine höhere Priorität bekommt. Bedingung (5.3) ist also auch erfüllt. Als Konsequenz muss eine neue Policy erstellt werden.

Nachdem die neuen RestACL-Policies erstellt wurden, müssen gegebenenfalls die Prioritäten erhöht werden. Dies geschieht nach folgender Vorschrift:

$$\forall r \in ps : \pi_{\rho_r} = \begin{cases} \pi_{max} + 1 & \text{if } (5.4) \vee (5.5) \\ \pi_{\rho_r} + off_{ps}(r) & \text{if } l_{ps} = FA \end{cases}$$

$$l_{ps} = PO \wedge e_r = P \wedge \exists r' \in SUB_{ps} : e_{r'} = D \wedge \pi_{r'} \geq \pi_r \tag{5.4}$$

$$l_{ps} = DO \wedge e_r = D \wedge \exists r' \in SUB_{ps} : e_{r'} = P \wedge \pi_{r'} \geq \pi_r \tag{5.5}$$

Dies bedeutet, dass die Prioritäten erhöht werden müssen, wenn die XACML-Rule den dominierenden Effekt des Combining Algorithm hat oder wenn der Combining Algorithm *FirstApplicable* ist. Im ersten Fall erhält die Policy die Priorität $\pi_{max} + 1$, so dass sie die am höchsten priorisierte Policy wird. Im zweiten Fall wird eine Funktion $off_{ps}(r)$ benötigt, welche einen Offset berechnet, der zur Priorität hinzugefügt wird. Der Offset entspricht der Anzahl an Rules, welche in allen Policy Sets und Policies enthalten sind und nach dem Policy Set oder der Policy evaluiert werden, welche *r* enthält. Beispielsweise könnte ein erstes Policy Set 2 untergeordnete Rules haben, ein zweites Policy Set könnte 4 untergeordnete Rules haben und ein drittes Policy Set könnte 3 untergeordnete Rules haben. Der Offset ist in diesem Fall 7 für das erste Policy Set, 3 für das zweite Policy Set und 0 für das dritte Policy Set.

Da die Prioritätenberechnung ein wesentlicher aber sehr komplexer Vorgang ist, soll ein umfangreiches Beispiel den Prozess verdeutlichen. Abbildung 5.5 zeigt den Strukturbaum eines XACML-Regelwerks. Es

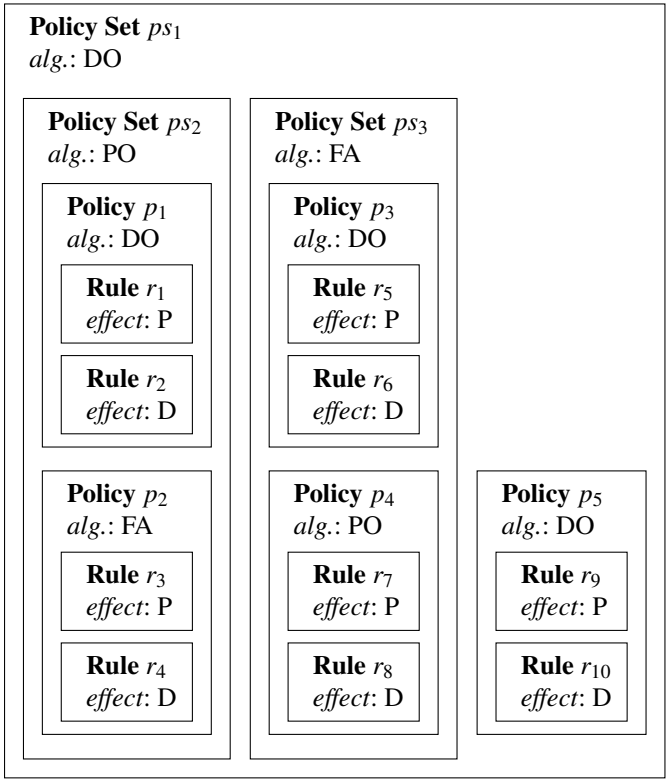

Abbildung 5.5: XACML Combining Algorithm und Effects

sind 3 Policy Sets ps_1, ps_2 und ps_3 vorhanden mit den Policies p_1 bis p_5. Die Policies enthalten die Rules r_1 bis r_{10}.

Initial werden alle Prioritäten der RestACL-Policies auf 0 gesetzt. Danach startet die Prioritätenberechnung basierend auf den Combining Algorithm der XACML-Policies. In Abhängigkeit der Effekte werden die Prioritäten erhöht auf 1. Da beispielsweise die XACML-Rule r_2 das Ergebnis der XACML-Rule r_1 überschreibt, wird $\pi_{\rho_{r_2}} = 1$ gesetzt.

$$
\begin{aligned}
p_1 \quad &\Rightarrow \quad \pi_{\rho_{r_2}} = 1 \\
p_2 \quad &\Rightarrow \quad \pi_{\rho_{r_3}} = 1 \\
p_3 \quad &\Rightarrow \quad \pi_{\rho_{r_6}} = 1 \\
p_4 \quad &\Rightarrow \quad \pi_{\rho_{r_7}} = 1 \\
p_5 \quad &\Rightarrow \quad \pi_{\rho_{r_{10}}} = 1
\end{aligned}
$$

Danach kann mit der Prioritätenberechnung begonnen werden, welche durch die XACML-Policy-Sets bestimmt wird. Die Berechnung beginnt mit ps_2. Es müssen alle Kombinationen gefunden werden, welche dazu führen, dass der Effekt *Permit* bis ps_2 gereicht wird. Diese Kombinationen brauchen die höchste Priorität unterhalb von ps_2.

$$ps_2 \quad \Rightarrow \quad \pi_{\rho_{r_1 \overline{r_2}}} = 2$$

r_1 liefert *Permit* genau dann wenn r_2 nicht anwendbar ist. Wenn r_2 anwendbar ist, wird p_1 den Effekt von r_1 überschreiben. Deshalb muss eine neue RestACL-Policy angelegt werden mit einer höheren Priorität als ρ_{r_2}. Der Effekt dieser neuen Policy ist *Permit*.

Ebenfalls kann aus ps_2 abgeleitet werden, dass die Priorität von ρ_{r_3} größer sein muss als die Priorität von ρ_{r_2}.

$$ps_2 \quad \Rightarrow \quad \pi_{\rho_{r_3}} = 2$$

r_3 liefert *Permit* unabhängig davon, ob r_4 anwendbar ist oder nicht. Deshalb bekommt ρ_{r_3} eine höhere Priorität als ρ_{r_2} und es muss keine neue Regel angelegt werden.

Policy Set ps_3 hat den Combining Algorithm *FirstApplicable*, weshalb keine neuen RestACL-Policies benötigt werden. Stattdessen werden die existieren RestACL-Policies angepasst hinsichtlich ihrer Prioritäten. Dies bedeutet, die Policies hinsichtlich der XACML-Rules unter p_3 müssen eine höhere Priorität haben als die Policies hinsichtlich der XACML-Rules unter p_4. Deshalb werden die Prioritäten ρ_{r_5} und ρ_{r_6} im gleichen Maße erhöht. Der Offset, um den sie erhöht werden, beträgt 2, da in p_4 2 Rules existieren.

$$\begin{aligned} p_3 \quad &\Rightarrow \quad \pi_{\rho_{r_6}} = 3 \\ p_4 \quad &\Rightarrow \quad \pi_{\rho_{r_5}} = 2 \end{aligned}$$

Schließlich müssen noch die Prioritäten berechnet werden, welche sich durch ps_1 ergeben. ps_1 hat den Combining Algorithm *DenyOverrides*, weshalb Regelkombinationen gefunden werden müssen, welche ein *Deny* produzieren.

$$ps_1 \quad \Rightarrow \quad \pi_{\rho_{r_2\overline{r_3}}} = 4$$

r_2 produziert *Deny*, wenn r_3 nicht anwendbar ist. Die neue Policy $\rho_{r_2\overline{r_3}}$ hat den Effekt *Deny*. Die Priorität muss höher sein als jede bisher vergebene Priorität. Analog können 3 weitere Kombinationen gefunden werden.

$$\begin{aligned} ps_1 \quad &\Rightarrow \quad \pi_{\rho_{r_4\overline{r_3}\overline{r_1}}} = 4 \\ ps_1 \quad &\Rightarrow \quad \pi_{\rho_{r_8\overline{r_5}\overline{r_7}}} = 4 \\ ps_1 \quad &\Rightarrow \quad \pi_{\rho_{r_{10}}} = 4 \end{aligned}$$

Damit ergeben sich die in Tabelle 5.1 abgebildeten Prioritäten. Da bei RestACL Prioritäten eindeutig sein müssen, müssen die Prioritäten nach der Position in der Tabelle vergeben werden.

Pos.	π	$\rho_{r_i \ldots r_j}$	ε
13	4	$\rho_{r_{10}}$	D
12	4	$\rho_{r_2\overline{r_3}}$	D
11	4	$\rho_{r_4\overline{r_3}\overline{r_1}}$	D
10	4	$\rho_{r_8\overline{r_5}\overline{r_7}}$	D
9	3	ρ_{r_6}	D
8	2	ρ_{r_3}	P
7	2	ρ_{r_5}	P
6	2	$\rho_{r_1\overline{r_2}}$	P
5	1	ρ_{r_2}	D
4	1	ρ_{r_7}	P
3	0	ρ_{r_1}	P
2	0	ρ_{r_4}	D
1	0	ρ_{r_8}	D
0	0	ρ_{r_9}	P

Tabelle 5.1: Resultierende Prioritäten

5.7 Verwandte Arbeiten

Die Bestimmung von Prioritäten hat ähnliche Ziele wie der Einsatz von Entscheidungsbäumen, so wie bei XEngine [94, 93] und anderen

Verfahren [121] beschrieben. Ziel des Einsatzes des Entscheidungsbaums bei XEngine ist das Erstellen eindeutiger Regeln, so dass beim Eintreffen einer Zugriffsentscheidung nur eine Regel anwendbar sein kann, zu der eine bereits berechnete Zugriffsentscheidung existiert. Anders als bei XEngine wird der Entscheidungsbaum in [121] erst zur Laufzeit ausgewertet, um die anwendbaren Regeln zu bestimmen. Das Ziel der hier beschriebenen Priorisierung ist die Bestimmung der Regeln, welche zuerst ausgewertet müssen beim Eintreffen einer Zugriffsanfrage, da diese Regeln dominanter gegenüber anderen Regeln sind und die Zugriffsentscheidung bestimmen. Das gemeinsame Ziel aller Verfahren ist also die schnelle Identifikation anwendbarer Regeln.

Die Vorgehensweisen unterscheiden sich jedoch in einem Aspekt grundlegend. XEngine setzt darauf eindeutige Attributwertkombinationen zu finden, welche keine Überlappungen mit anderen Kombinationen haben, so dass zu einer Zugriffsanfrage auch nur eine Regel anwendbar ist. Auch der Entscheidungsbaum in [121] verwendet konkrete Attributwerte. Der Priorisierungsprozess hingegen stützt sich weniger auf konkrete Attributwerte, sondern basiert alleine auf der Struktur des Regelwerks. Einschränkungen auf bestimmte Attributwerte werden nicht vorgenommen. Die Priorität einer Regel wird alleine anhand der Struktur des XACML Regelwerks und der darin enthaltenden Combining Algorithm und Effects bestimmt.

Ein Entscheidungsbaum liefert für jede Attributwertkombination die anwendbaren Regeln. Der XEngine-Ansatz nutzt dies, um neue Regeln zu erzeugen, so dass eine Eindeutigkeit innerhalb der Regeln gegeben ist. Eine Zugriffsanfrage gegen die neue Menge von Regeln wird immer genau einer neuen Regel zugeordnet. Der Ansatz aus [121] wertet zur Laufzeit die anwendbaren Regeln aus. Bei diesem Ansatz ist es möglich, dass mehrere Regeln gleichzeitig an-

wendbar sind. Die Zugriffsentscheidung wird dann durch Combining Algorithm bestimmt. Der hier beschriebene Priorisierungsansatz nutzt keinen Entscheidungsbaum, der auf konkreten Werten beruht, sondern berechnet Prioritäten von Regeln aufgrund der Struktur eines XACML-Regelwerks. Durch den Einsatz verschiedener Combining Algorithm und Effects entsteht eine komplexe Baumstruktur in deren Knoten unterschiedliche Gewichtungen auftreten können. Beispielsweise kann eine Regel r_1 mit dem Effekt *Permit* von einer Regel r_2 mit dem Effekt *Deny* dominiert werden, wenn der zugeordnete Combing Algorithm *DenyOverrides* ist. Wird in einer höheren Ebene des Entscheidungsbaums jedoch der Combining Algorithm *PermitOverrides* angwendet, so wird r_1 dominanter. Die Voraussetzunge hierfür ist jedoch, dass das Ergebnis von r_2 nicht zuvor bereits das Ergebnis von r_1 überschrieben hat. Die gesamte Berechnung von Prioritäten ist unabhängig von konkreten Attributwerten und -bedingungen, sondern beruht alleine auf der Struktur des XACML-Regelwerks. Hierdurch unterscheiden sich der Einsatz von Entscheidungsbäumen und die Berechnung von Prioritäten entscheidend, auch wenn das gemeinsame Ziel das Bestimmen der anwendbaren Regeln ist. Beim Priorisierungsansatz müssen keine Einschränkungen auf konkrete Attributwerte vorgenommen werden.

Die Berechnung von allgemein gültigen Prioritäten für XACML-Rules zeigt, dass das Überführen eines XACML-Regelwerks in ein RestACL-Regelwerk möglich ist. Hierdurch ist es möglich, den kompositionellen Aspekt von XACML aufzulösen und durch einen index- und prioritätenbasierten Ansatz zu ersetzen, ohne dabei die Korrektheit der Zugriffsberechtigungen zu beeinflussen. Das Auflösen des kompositionellen Aspekts wird bei XEngine ebenfalls durchgeführt, um die Verarbeitungsgeschwindigkeit von Zugriffsanfragen zu verbessern. XEngine setzt dabei auf Eindeutigkeit von Regeln, welche möglich

wird, indem Attributwerte eingeschränkt werden. Bei der in diesem Kapitel beschriebenen Transformation sind keine Einschränkungen dieser Art notwendig. Der Unterschied zwischen beiden Ansätzen ist also, dass XEngine anhand von Werten unterscheidet, während der hier beschriebene Ansatz anhand der Struktur unterscheidet. XEngine generiert dazu eine Menge neuer Regeln, welche den eingeschränkten Wertebereich abdecken und in einer flachen Struktur angeordnet sind. Der in diesem Kapitel dargstellte Priorisierungsmechanismus berücksichtigt die Struktur des Regelwerks und führt komplementäre Regeln ein, welche es ermöglichen aus einer hierarchischen Struktur mit unterschiedlichen Combining Algorithmen eine priorisierte Liste zu berechnen ohne dabei Wertebereiche zu kennen oder gar einzuschränken.

5.8 Zusammenfassung

In diesem Kapitel wurden die folgenden wissenschaftlichen Beiträge beschrieben:

- **Korrektheit und Vollständigkeit von RestACL:** die beiden Eigenschaften wurden durch Transformationsvorschriften nach und von XACML belegt. XACML ist ein etablierter Standard. Durch die Existenz von Transformationsvorschriften können Korrektheit und Vollständigkeit belegt werden.
- **Semantik von XACML:** Die Berechnung von Prioritäten ist ein Beitrag, um die Semantik von XACML Combining Algorithmen zu erläutern. Es ist eine Methode, um die Wichtigkeit einzelner Zugriffsbedingungen zu bestimmen.

RestACL ist ressourcenorientiert. Zugriffsberechtigungen, die nicht explizit eine Bedingung an die Ressourcenadresse stellen, müssen bei jeder Ressource angegeben werden. Solange die Abkehr von der Ressourcenorientierung eine Ausnahme ist, ist dies ein vernachlässigbarer Mehraufwand, da lediglich zusätzliche Verweise hinzugefügt werden müssen. Existieren jedoch viele Zugriffsberechtigungen, die keinen Bezug zu Ressourcen haben, so könnte die Menge der Verweise sehr groß werden und einen negativen Einfluss auf die Verarbeitungsgeschwindigkeit haben.

In diesem Kapitel wurden XACML und RestACL formal beschrieben und miteinander verglichen. Es wurden formelle Methoden eingesetzt, um sowohl XACML als auch RestACL zu beschreiben. Die Korrektheit und Vollständigkeit von RestACL wurde mit Hilfe von Transformationsverfahren belegt. Gleichzeitig untersucht dieser Abschnitt einige Charakteristika von XACML. Die Priorisierung von XACML-Rules ist durch den kompositionellen Aspekt sehr kompliziert. Verschiedene Effects und Combining Algorithm ergeben komplexe Strukturen, welche das Verständnis von Regelwerken erschweren. In diesem Kapitel wurde ein systematischer Ansatz vorgestellt mit dem Prioritäten von XACML-Rules innerhalb des Regelwerks berechnet werden können. Durch die Verwendung von Prioritäten wird es leichter ein Regelwerk zu verstehen und zu interpretieren.

Kapitel 6

Anwendungsmöglichkeiten

In diesem Kapitel werden die Einsatzmöglichkeiten von RestACL aufgezeigt. Es wird zunächst ein Ansatz beschrieben, der eine systematische Umsetzung von HATEOAS mittels eines Zugriffskontrollsystems ermöglicht. Dieser Ansatz wurde in [61] veröffentlicht. Anschließend werden situationsabhängige Zugriffskontrolle und ein Tokenverfahren für IoT-Umgebungen beschrieben. Diese beiden Ansätze wurden in [62, 63] und [101] publiziert.

6.1 Autorisierungsabhängiges Hypermedia

In den vorherigen Kapiteln wurde bereits die Bedeutung von Hypermedia As The Engine Of Application State erläutert. HATEOAS ist eines der wichtigsten Prinzipien von REST, wenn die Vorteile des

Architektursstils bestmöglich genutzt werden sollen. Mit Hilfe von HATEOAS ist es möglich, Client und Server zu entkoppeln. Dadurch wird es ermöglicht, standardisierte Clients einzusetzen. Außerdem trägt die Idee hinter HATEOAS wesentlich zur Skalierbarkeit eines verteilten Systems bei, da es Verwaltungsaufwände auf Seiten des Servers erheblich reduziert.

Aber die Realisierung von HATEOAS ist herausfordernd, da keine systematischen Ansätze existieren, mittels derer HATEOAS umgesetzt werden kann. Deshalb bleibt die Umsetzung eine Aufgabe der Entwickler einer Anwendung. In diesem Abschnitt wird eine Methode vorgestellt, die aufzeigt wie HATEOAS systematisch angewendet werden kann. Dabei werden REST-API-Modelle mit dem RestACL-Zugriffskontrollmechanismus integriert. Hierdurch können nicht-autorisierte Zugriffsversuche und damit überflüssiger Netzwerkverkehr reduziert werden. Es werden auf das anfragende Subjekt zugeschnittene Repräsentationen von Ressourcen erstellt, welche keine Verweise enthalten, denen das anfragende Subjekt nicht folgen darf. Hierzu ist es notwendig den RestACL-Mechanismus derart zu erweitern, dass eine Unterscheidung zwischen statischen und dynamischen Ressourcen möglich ist. Darüber hinaus wird eine 2-phasige Autorisierungsprozedur vorgestellt, die bestimmt, welche Verweise in der zugeschnittenen Repräsentation der Ressource enthalten sein müssen und welche nicht enthalten sein dürfen.

HATEOAS erfordert es, dass der Server dem anfragenden Client alle nächstmöglichen Statusübergänge zusammen mit der Antwort auf die Anfrage des Client sendet [11] (Kapitel 5). Diese Statusübergänge werden als Hypermedia transferiert. Deshalb ist es sinnvoll, Statusübergänge zwischen Ressourcen als Metadaten zu modellieren. Existiert ein Metadatenmodell, so kann dieses Modell ausgenutzt wer-

den, um Zugriffskontrolle zu gestalten. Zustandsübergänge, welchen der anfragende Client nicht folgen darf, werden dann nicht in die Antwort an den Client integriert. Dies hilft dabei die Sicherheit zu erhöhen, unnötigen Netzwerkverkehr zu vermeiden (insbesondere in der Maschine-zu-Maschine-Kommunikation) und die Benutzerfreundlichkeit zu verbessern, da so vermieden wird, dass Benutzer zu Ressourcen navigieren, auf die sie nicht zugreifen dürfen.

Der im Folgenden beschriebene Ansatz kombiniert REST-API-Modelle und feingranulare Zugriffskontrolle für RESTful Services. Anstatt die gleichen Repräsentationen von Ressourcen für alle Clients zu verwenden, werden Zugriffsberechtigungen im Vorfeld berücksichtigt und in zugeschnittenen Repräsentationen umgesetzt.

Beispielszenario Um die Idee des hier beschriebenen Ansatzes zu verdeutlichen, soll ein praktisches Szenario verwendet werden. Industrie 4.0 [89] ist ein Gebiet auf dem verschiedene Subjekte wie Kunden, Zulieferer, Anbieter, Kontrolleure und intelligente Maschinen zusammen auf denselben Ressourcen agieren. Ein Beispiel für eine solche Umgebung ist eine Fabrik, in der komplexe Produkte hergestellt werden, bei denen der Kunde die Möglichkeit hat, Komponenten des Produkts anzupassen, auch wenn die Produktion schon gestartet ist. Als Beispiel für eine solche Fabrik könnte die Automobilbranche dienen. Kunden könnten etwa die Konfiguration des Interieurs noch anpassen, auch wenn Karosserie und Motor schon in Produktion sind.

Betrachtet man das Produkt aus REST-Perspektive, so ist es eine Ressource und die parametrisierbaren Komponenten des Produkts sind untergeordnete Ressourcen. Abbildung 6.1 zeigt den Ausschnitt eines Statusdiagramms. Innerhalb der Boxen ist der jeweilige Sta-

tusname, während die Übergänge zwischen den Stati den Methoden einer REST-Schnittstelle entsprechen. In diesem Szenario wird für alle Ressourcen und deren untergeordnete Ressource das gleiche Statusdiagramm verwendet, um das Beispiel einfach zu halten und sich auf die Ideen von autorisierungsabhängigen HATEOAS zu fokussieren. Dabei ist zu beachten, dass reale Ressourcen wesentlich komplexere Statusdiagramme haben.

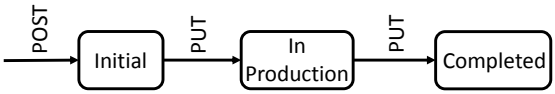

Abbildung 6.1: Einfaches Zustandsdiagramm für Ressourcen (Quelle: [61])

Im Beispiel kann ein Kunde ein Produkt bestellen, indem er durch Senden einer POST-Anfrage an *http://example.org/products* eine Erstellungsanfrage an die Fabrik richtet. Die Produktressource ist solange in einem initialen Status, bis ein Mitarbeiter der Fabrik entscheidet, die Produktion zu beginnen und die Ressource mittels einer PUT-Anfrage aktualisiert. Diese Anfrage schickt er an die Adresse der zuvor erstellten Ressource: *http://example.org/products/{id}*. Von nun an darf der Kunde zwar lesend auf die Ressource zugreifen, jedoch kann der Kunde die Ressource nicht mehr aktualisieren. Dies dürfen nur noch die Mitarbeiter der Fabrik.

Aber der Kunde kann individuelle Teile des Produkts hinzufügen. Dazu muss der Kunde eine POST-Anfrage an die Ressource mit

dem Pfad */products/{id}/parts* senden. Dies kann der Kunde solange tun, bis ein Mitarbeiter der Fabrik den Status dieser Listenressource auf *Completed* ändert. Ebenfalls kann der Kunde einzelne Teile anpassen, welche noch nicht in Produktion, also noch im Status *Initial* sind. Hierfür kann der Kunde eine PUT-Anfrage an die Ressource *http://example.org/products/{id}/parts/{id}* senden.

Tabelle 6.1 zeigt die REST-API des Produktszenarios. Diese API wird in diesem Abschnitt durchgehend verwendet. Die Tabelle zeigt die genannten Ressourcen, die möglichen Zugriffsmethoden und die Subjekte, welcher die Zugriffe durchführen dürfen.

	Resource	**Met.**	**Subjects**
1	/products	POST	Customer
2	/products/{id}	GET	Customer, Worker
3	/products/{id}	PUT	Worker
4	/products/{id}/parts	GET	Customer, Worker
5	/products/{id}/parts	POST	Customer
6	/products/{id}/parts	PUT	Worker
7	/products/{id}/parts/{id}	GET	Customer, Worker
8	/products/{id}/parts/{id}	PUT	Customer, Worker

Tabelle 6.1: Produkt API

Aus der Szenariobeschreibung lässt sich erkennen, dass verschiedene Zugriffsmethoden entweder exklusiv dem Kunden, exklusiv den

Mitarbeitern oder beiden Subjekttypen zur Verfügung stehen. Zusätzlich hängen die Zugriffsberechtigungen einiger Ressourcen auch vom aktuellen Status der Ressource ab. So sind die Anfragen zum Aktualisieren eines Produkts (Operation 3), das Erstellen von neuen Komponenten (Operation 5), das Aktualisieren der Komponentenliste (Operation 6) und das Aktualisieren einzelner Komponenten (Operation 8) nur in bestimmten Ressourcenzuständen möglich.

6.1.1 Hypermedia Navigation Model

Neben dem bereits ausführlich beschriebenen Zugriffskontrollmechanismus (RestACL) ist das REST-API-Model die zweite entscheidende Grundlage für die Umsetzung einer Lösung für autorisierungsabhängiges Hypermedia. Detailliertes Wissen über die Struktur der REST-API, genau genommen über die verfügbaren Ressourcen und die Navigationspfade, ist unabdingbar. Es existieren eine Reihe von Metamodellen für REST-APIs, inklusive industriegetriebenen Sprachen wie Swagger [135], OpenAPI[105] oder RAML[113]. Ebenso gibt es wissenschaftliche Arbeiten, die sich der Thematik widmen [127, 87, 123, 53]. Obwohl HATEOAS ein zentrales Feature von REST-APIs ist, bilden die meisten Metamodelle es nicht ab. Dies bedeutet, die Metamodelle bieten keine Möglichkeit Hypermedia-basierte Verknüpfungen darzustellen. Dieses Defizit wird mit dem Metamodell aus [51, 52] angegangen. Auch die OpenAPI Spezifikation 3.0 greift die Modellierung auf und sieht das *Link*-Konstrukt vor, um Verknüpfungen zwischen Ressourcen zu beschreiben.

Das Hypermedia-beschreibende Metamodell, umfasst alle möglichen Navigationsbeziehungen zwischen Ressourcen. Dennoch können zu bestimmten Zeitpunkten diese Navigationsbeziehungen unter Um-

ständen nicht traversiert werden. Das zuvor beschriebene Szenario beinhaltet Beispiele dafür. Es gibt zwei grundlegende Ansätze dieses Problem anzugehen. Die erste Möglichkeit ist es, das Ressourcenmodell zu erweitern, so dass der interne Status und auch die Relationen zwischen den Ressourcen inklusive der Operationen gepflegt werden können. Dieser Ansatz wird in [127] verfolgt. Der zweite Ansatz hingegen beschäftigt sich weniger mit dem internen Status der Ressource, sondern prüft, welche Relationen aktiv sind, also dem anfragenden Client gezeigt werden sollen. Hierbei kann nicht nur der interne Status der Ressourcen berücksichtigt werden, sondern auch vollständig unabhängige Parameter wie etwa die Zeit der Anfrage oder der geographische Standort des Clients. Grundlage hierfür ist die Integration eines ABAC-Mechanismus mit dem Metamodell. Hierdurch wird eine mächtige Lösung möglich, welche systematisch Hypermedia-getriebene REST-APIs realisiert.

Zuvor wurde bereits die REST-API (vgl. Tabelle 6.1) beschrieben und der Status einzelner Ressourcen dargestellt (vgl. Abbildung 6.1). Es wird deutlich, dass keinerlei Informationen bekannt sind, wie die Relationen zwischen den Ressourcen aussehen. Weder die API noch das Zustandsdiagramm beinhalten die Information, wie die Ressourcen verknüpft sind. Ein Metamodell hilft bei genau diesem Problem.

Abbildung 6.2 zeigt die Beziehungen zwischen den Ressourcen aus dem Produktszenario. Die graphische Notation ist [117] entnommen. Eine POST-Anfrage auf die Produktliste erstellt eine neue Produktressource. Aus Abbildung 6.1 erkennt man, dass die Ressource mittels einer PUT-Anfrage aktualisiert werden kann. Das Metamodel ergänzt nun die Information, dass der Client in diesem Fall zurück zur aktualisierten Ressource gelangt. Zusätzlich zeigt das Metamodel, dass es Zustandsübergänge zur Komponentenliste und den bereits hinzugefüg-

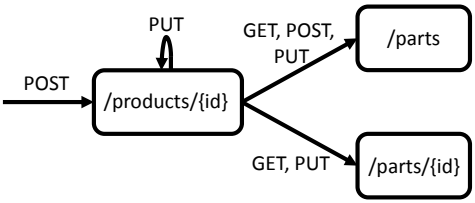

Abbildung 6.2: Hypermedia Navigation Model (Quelle: [61])

ten Komponenten gibt. Der Zustandsübergang zur Komponentenliste kann mit GET-, POST- oder PUT-Anfragen durchlaufen werden, während der Zustandsübergang zu individuellen Komponenten mit GET- oder PUT-Anfragen durchlaufen werden kann.

Beispielsweise kann ein Kunde ein Produkt (*/product/1*) erstellt und bereits zwei Komponenten (mit den Pfaden */product/1/parts/1* und */product/1/parts/2*) hinzugefügt haben. Tabelle 6.2 listet die vom Metamodel abgebildeten Hypermedia-Beziehungen. Nach Abbildung 6.2 hat die Produktressource eine Referenz auf sich selbst, welche mittels PUT-Anfragen durchlaufen werden kann. Die Komponentenliste (die untergeordnete */parts* Ressource von */product/1*) kann mit GET-, POST- oder PUT-Anfragen erreicht werden. Beide bereits hinzugefügten Komponenten (*/parts/1* und */parts/2*) können direkt über eine Hypermedia-Referenz vom Produkt aus erreicht werden.

Resource	Methods
/products/1	PUT
/products/1/parts	GET, POST, PUT
/products/1/parts/1	GET, PUT
/products/1/parts/2	GET, PUT

Tabelle 6.2: Metadatenbeispiel für die Ressource /product/1

6.1.2 Maßgeschneiderte Repräsentationen

HATEOAS erfordert es, dass einem anfragenden Subjekt alle möglichen Zustandsübergänge, welchen das Subjekt folgen kann, als Bestandteil der Antwort genannt werden. Für das Beispiel bedeutet dies, dass eine Anfrage an die Produktressource alle in Tabelle 6.2 aufgeführten Statusübergänge beinhalten muss. Dies entspricht allen Hypermedia-Elementen wie sie vom Hypermedia-Navigations-Modell beschrieben werden. Im Allgemeinen wird Zugriffskontrolle dabei nicht berücksichtigt. Als Konsequenz werden dem anfragenden Subjekt möglicherweise Hypermedia-Elemente geliefert, welche das Subjekt nicht nutzen kann beziehungsweise dessen Nutzung das Subjekt lediglich zu einer Fehlermeldung führt. Im Beispiel werden dem Kunden etwa Hypermedia-Elemente geliefert, um die Produktressource zu aktualisieren, obwohl er dies nicht darf. Andere Subjekte hingegen dürfen diese Operation ausführen. Die Mitarbeiter der Fabrik können die Produktressourcen aktualisieren. Aber die Mitarbeiter dürfen wiederum andere Operationen nicht ausführen, wie beispielsweise das Hinzufügen von neuen Komponenten, welches den Kunden vorbehalten ist.

Dieses Problem kann dadurch behoben werden, dass Zugriffsberechtigungen ausgewertet und berücksichtigt werden, bevor die Antwort an das anfragende Subjekt zurückgeliefert wird. Operationen, welche in keinem Fall durchgeführt werden können, dürfen nicht in die Antwort an den Client integriert werden, so dass der Client nichts von den Operationen mitbekommt. Werden Zugriffsberechtigungen im Vorfeld berücksichtigt, so muss die Ressourcenrepräsentation auf das anfragende Subjekt maßgeschneidert werden. Die maßgeschneiderte Repräsentation der Ressource enthält dann lediglich die Statusübergänge, welche der Client ausführen darf oder die Statusübergänge, welche in Zukunft möglicherweise ausführbar sein werden. Dies hilft dabei die Sicherheit zu erhöhen, unnötigen Netzwerkverkehr zu vermeiden und das Benutzererlebnis zu verbessern, indem Frustration bei menschlichen Benutzern reduziert wird.

6.1.3 2-phasige Autorisierung

Um eine Antwort mit einer maßgeschneiderten Repräsentation einer Ressource an das anfragende Subjekt versenden zu können, ist ein 2-phasiger Autorisierungsprozess notwendig. In der ersten Phase wird die eigentliche Anfrage autorisiert. In der zweiten Phase werden die Zugriffsberechtigungen für die möglichen Zustandsübergänge bestimmt. Es ist wichtig zwischen den beiden Phasen zu unterscheiden. In den beiden Phasen werden unterschiedliche Attributtypen ausgewertet. Nicht alle Attribute können bei der Auswertung der Zugriffsanfragen in der zweiten Phase herangezogen werden, da ihre Werte nicht konstant sind (vgl. Abschnitt 6.1.4)

Abbildung 6.3 zeigt die Autorisierung der ursprünglichen Anfrage. Der Ablauf beginnt mit der initialen Anfrage eines Subjekts, wie et-

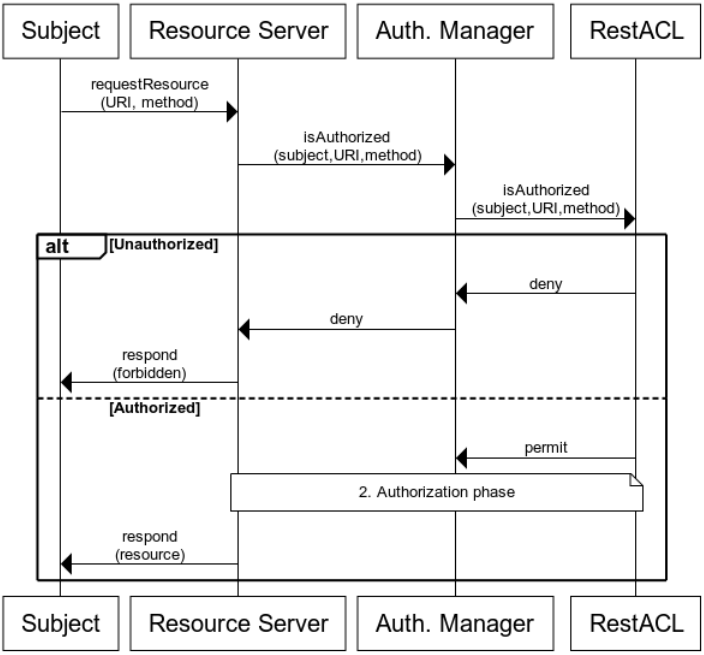

Abbildung 6.3: Erste Autorisierungsphase

wa einem Kunden, einem Mitarbeiter der Fabrik oder einer anderen Maschine. Die Anfrage erreicht den Resource Server (Res. Server), welche die Anfrage unmittelbar an den Authorization Manager (Auth.

Manager) weiterreicht. Der Authorization Manager formuliert eine Zugriffsanfrage und reicht sie an das Autorisierungssystem (RestACL) weiter. Das Autorisierungssystem prüft die Berechtigung und sendet das Resultat dieser Prüfung zurück an den Authorization Manager. Dieser ist dafür verantwortlich die Zugriffsentscheidung durchzusetzen. Beispielsweise kann der Resource Server mit einem *HTTP 403 Forbidden* antworten, wenn der Zugriff nicht erlaubt wird. Wird der Zugriff erlaubt, so kann die angefragte Resource ausgeliefert werden. Bevor die Resource jedoch ausgeliefert wird, wird sie in einer zweiten Phase maßgeschneidert für das anfragende Subjekt.

Abbildung 6.4 zeigt die neu hinzukommende zweite Phase der Autorisierung. Wird die ursprüngliche Anfrage durch das Autorisierungssystem genehmigt, beginnt der Authorization Manager mit der Erstellung der maßgeschneiderten Repräsentation der Ressource bevor der die Zugriffsentscheidung an den Resource Server weiterleitet. Erst wenn diese Phase abgeschlossen ist, kann der Resource Server die Ressource an das Subjekt ausliefern.

Hierfür sendet er zunächst eine Anfrage an das Hypermedia Navigation Model (Nav. Model), um die möglichen Zustandsübergänge zu erfahren. Das Navigation Model kennt die Zustandsübergänge für alle Ressourcen. In unserem Beispielszenario ist das Navigation Model etwa in Kenntnis darüber, dass die Produktressource Zustandsübergänge zum Aktualisieren des Produkts, zum Hinzufügen von Komponenten und zum Aktualisieren von Komponenten hat. Das Navigation Model liefert die Zustandsübergänge als Paare von URIs und Zugriffsmethoden zurück an den Authorization Manager. Der Authorization Manager iteriert über diese Liste und führt für jeden Listeneintrag eine Zugriffsanfrage beim Autorisierungssystem aus. Wird ein hypothetischer Zugriff gewährt, so wird der Zustandsübergang in die Antwort

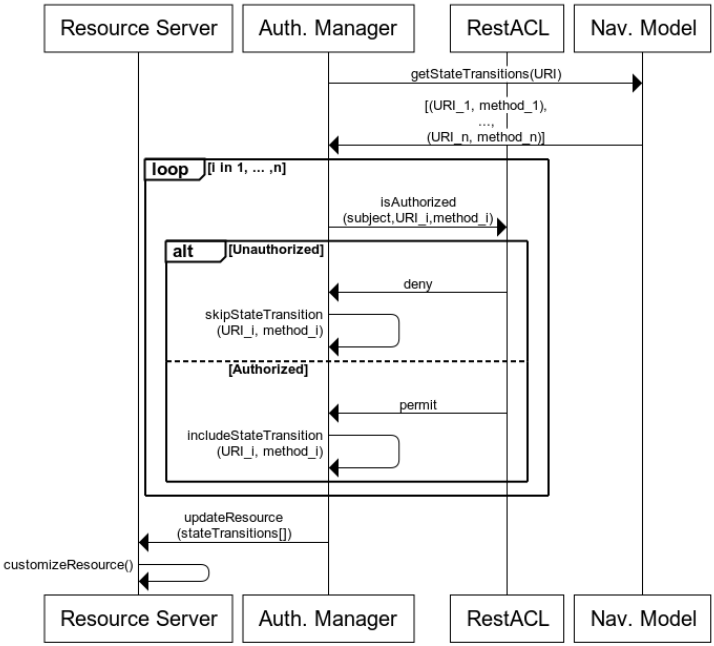

Abbildung 6.4: Zweite Autorisierungsphase

integriert. Wird der Zugriff ausgeschlossen, so wird der Zustandsübergang nicht mit in die Antwort eingefügt. Sobald alle Zustandsübergänge abgearbeitet wurden, antwortet der Authorization Manager und liefert alle Zustandsübergänge an den Resource Server zurück. Der

Resource Server muss die Repräsentation der Ressource entsprechend den Vorgaben des Authorization Manager anpassen und zurück an das anfragende Subjekt liefern.

Subjekt, Resource Server und Autorisierungssystem sind bekannte Komponenten einer REST-Umgebung. Authorization Manager und Hypermedia Navigation Model hingegen sind neue Architekturkomponenten. Der Authorization Manager agiert als Koordinator der Zugriffkontrolloperationen. Er koordiniert den Prozess der Maßschneiderung der Repräsentation der Ressource. Dies bedeutet, er versteckt alle Zugriffskontrolloperationen vor dem Resource Server, ohne selbst die Zugriffsentscheidung zu fällen.

Beispiel An dieser Stelle soll das Produktbeispiel näher betrachtet werden. 2 Policies sollen darstellen, wie der Zugriff für die Ressourcen gesteuert werden kann.

Die erste Policy gewährt dem Kunden Zugriff. *P1* (vgl. Auflistung 6.1) erlaubt den Zugriff, wenn das anfragende *subject* den *type Customer* hat und die angefragte *resource* im *Initial state* ist.

Die zweite Policy aus dem Industrie 4.0 Szenario gewährt Mitarbeitern der Fabrik Zugriff. *P2* (vgl. Auflistung 6.2) gewährt Zugriff, wenn das anfragende *subject* den *type Worker* hat und die *resource* nicht im *Completed state* ist. Mitarbeiter können also mit Hilfe dieser Policy nur auf Ressourcen zugreifen, welche noch nicht fertiggestellt wurden.

```
{
  "policies": [
    {
      "id": "P1",
      "description": "Customer access policy",
      "effect": " Permit ",
      "priority": "1" ,
      "compositeCondition": {
        "operation": "AND",
        "conditions": [
          {
            "function": "equal",
            "arguments": [
              {"category": "subject",
               "designator": "type"},
              {"value": "Customer"}
            ]
          },{
            "function": "equal",
            "arguments": [
              {"category": "resource",
               "designator": "state"},
              {"value": "Initial"}
            ]
          }
        ]
      }
    }
  ]
}
```

Auflistung 6.1: Zugriffsberechtigungen des Kunden

```
{
  "policies": [
          {
      "id": "P2",
      "description": "Worker access policy",
      "effect": " Permit ",
      "priority": "2" ,
      "compositeCondition": {
        "operation": "AND",
        "conditions": [
          {
            "function": "equal",
            "arguments": [
              {"category": "subject",
               "designator": "type"},
              {"value": "Worker"}
            ]
          },{
            "function": "unequal",
            "arguments": [
              {"category": "resource",
               "designator": "state"},
              {"value": "Completed"}
            ]
          }
        ]
      }
    }
  ]
}
```

Auflistung 6.2: Zugriffsberechtigungen der Mitarbeiter

```
{
  "host": "http://example.org"
        "resources":[{
    "path":"/products/1",
    "access":[
      {"methods":["PUT"],"policies":["P2"]}
    ],
    "resources":[{
      "path":"/parts",
      "access":[
        {"methods":["POST"],"policies":["P1"]},
        {"methods":["PUT"],"policies":["P2"]}
      ],
      "resources":[{
        "path":"/{id}",
        "access":[
          {"methods":["PUT"],"policies":["P1, P2"]}
        ]
      }
    ]
  }
}
```

Auflistung 6.3: Zuweisung der Policies

Entsprechend der Beschreibung des Szenarios müssen diese beiden Policies den Ressourcen zugewiesen werden, so wie in Auflistung 6.3 dargestellt. Hierdurch können die Zugriffsbedingungen des Szenarios umgesetzt werden.

Sowohl Kunden als auch Mitarbeiter können lesende Zugriffe (mittels einer *GET*-Anfrage) an die Produktressource, die Kompo-

nentenliste und die einzelnen Komponenten richten. Deshalb muss dieser Zugriff hier nicht eingeschränkt werden. Aber nur Mitarbeiter können die Produktressource mittels einer *PUT*-Anfrage an die Ressource */product/1* aktualisieren. Andersherum können nur Kunden neue Komponenten über eine *POST*-Anfrage der untergeordneten Ressource */parts* hinzufügen. Die Policy stellt sicher, dass diese Anfragen nur gewährt werden, wenn die Ressourcen den entsprechenden Status haben.

Wenn der Kunde die Produktressource anfragt, wird in der zweiten Autorisierungsphase das Hypermedia Navigation Model konsultiert, welches alle möglichen Zustandsübergänge für die angefragte Ressource liefert. Für jeden hypothetischen, nächsten Zustandsübergang führt der Authorization Manager eine Zugriffsanfrage aus. Das Aktualisieren der Produktressource ist kein Zustandsübergang, den der Kunde durchführen kann. Eine *PUT*-Anfrage ist den Mitarbeitern der Fabrik vorbehalten (Policy *P2*). Dieser Zustandsübergang wird also nicht in die Antwort an den Kunden integriert.

6.1.4 Statische und dynamische Attribute

Ein entscheidender Einfluss auf ein Autorisierung berücksichtigendes HATEOAS-System ist die Zeit. RESTful Services, welche HATEOAS unterstützen, weisen eine zustandslose Kommunikation zwischen Client und Server auf. Der Client fragt eine Ressource an und der Server antwortet mit einer Repräsentation der Ressource inklusive der möglichen Zustandsübergänge. Es werden jedoch keinerlei Bedingungen gestellt, wann ein Zustandsübergang erfolgen muss oder kann. Die Zustandübergänge können unmittelbar nach der ersten Anfrage erfolgen, aber es können auch Minuten, Stunden, Tage oder gar Wochen ver-

gehen ehe ein Zustandsübergang erfolgt. Als Konsequenz ist es nicht immer möglich die Zugriffsberechtigungen für zukünftige Anfragen zu bestimmen. Können sich Attributwerte zwischen der ersten und der zweiten Anfrage ändern, so kann bei der Verarbeitung der ersten Anfrage keine verlässliche Aussage getroffen werden, ob der Zustandsübergang erlaubt ist. Deshalb ist es notwendig zwischen statischen und dynamischen Attributen zu unterscheiden.

Definition (Statische/Dynamische Attribute): Ein Attribut ist dynamisch, wenn sich der Wert des Attributs zwischen zwei aufeinanderfolgenden Zugriffsanfragen ändern kann. Andernfalls wird das Attribut als statisch bezeichnet.

Der Blick auf das Produktbeispiel zeigt den Unterschied zwischen statischen und dynamischen Attributen. Das anfragende Subjekt hat einen *type* wie *customer* oder *worker*. Die Werte dieses Attributs ändern sich zwischen den Anfragen des Subjekts nicht, so dass es als statisch deklariert werden kann. Ein Beispiel für ein dynamisches Attribut ist der Ressourcenstatus. Im Beispiel können die Ressourcen den Status *Initial*, *In Production* oder *Completed* annehmen. Durch das Aktualisieren einer Ressource kann der Status der Ressource geändert werden.

Ein potentieller Zustandsübergang kann zu jedem beliebigen Zeitpunkt in der Zukunft durchgeführt werden. Daher ist es nicht möglich, die Werte von dynamischen Attributen zu diesem Zeitpunkt zu kennen und eine verlässliche Aussage über die Zugriffsberechtigungen zu treffen. Deshalb wird für den Fall, dass dynamische Attribute für die Entscheidungsfindung benötigt werden, eine unbestimmte Entscheidung für die Bedingung mit dem dynamischen Attribut festgelegt.

Die Policies aus Auflistung 6.1 und Auflistung 6.2 enthalten jeweils ein statisches und ein dynamisches Attribut. Diese sind durch eine UND-Verknüpfung kombiniert. Bei der Berechnung von zunkünftigen Zugriffsentscheidungen wird die Teilbedingung mit dem dynamischen Attribut als unbestimmt festgelegt. Dies bedeutet, dass lediglich die Teilbedingung ausgewertet wird, welche das statische Attribut enthält. Wird eine der Policies für eine zukünftige Anfrage ausgewertet, so bestimmt allein der *type* des *subject* die Anwendbarkeit, da die Teilbedingung hinsichtlich des *state* der *resource* dynamisch ist, somit in einer zukünftigen Anfrage erfüllt sein könnte und deshalb als unbestimmt festgelegt wird.

Die Tabellen 6.3 und 6.4 zeigen die Resultate der Auswertung einer Kombination von statischen und dynamischen Attributen in Abhängigkeit der Erfüllung der Teilbedingungen. Ist die Zugriffsentscheidung unbestimmt, fügt der Authorization Manager die betroffene Ressource mit in die Repräsentation der Ressource ein. Die tatsächliche Entscheidung über den Zugriff kann dann erst bei der Durchführung der Anfrage getroffen werden.

Statische Teilbedingung erfüllt	**Dynamische Teilbedingung erfüllt**	**Resultat**
nein	nein	Nicht anwendbar
nein	ja	Nicht anwendbar
ja	nein	Unbestimmt
ja	ja	Unbestimmt

Tabelle 6.3: UND-Verknüpfung

Statische Teilbedingung erfüllt	**Dynamische Teilbedingung erfüllt**	**Resultat**
nein	nein	Unbestimmt
nein	ja	Unbestimmt
ja	nein	Effect der Policy
ja	ja	Effect der Policy

Tabelle 6.4: ODER-Verknüpfung

Wird beispielsweise die Policy aus Auflistung 6.1 für eine zukünftige Anfrage eines Kunden ausgewertet, liefert diese eine unbestimmte Entscheidung aufgrund des *state*-Attributs der *resource*. Dessen Wert kann sich bis zur tatsächlichen Anfrage noch ändern. Wird die Policy hingegen für eine zukünftige Anfrage des Mitarbeiters ausgewertet, so ist die Teilbedingung mit dem statischen Attribut nicht erfüllt und kann auch in der Zukunft nicht erfüllt werden. Die Policy ist also nicht anwendbar.

6.1.5 Autorisierungsabhängige Antworten

In [92] beschreiben die Autoren einen Ansatz, wie HATEOAS in bestehende Systeme integriert werden kann, welche nicht die Vorgaben von REST unterstützen. Die Kernidee ist die Verwendung von *Link*-Headern, um Zustandsübergänge zu kommunizieren. Jeder Zustandsübergang wird dabei als eigener Header der Antwort hinzugefügt. Hierfür wird zwischen Client und Server eine Komponente eingeführt, welche die Antwort vom Server abfängt und in einer Look-Up-Tabelle

nach den Zustandsübergängen sucht. Die Zustandsübergänge werden in einem State-Chart gespeichert. Sind die Zustandsübergänge bekannt, so kann die Komponente Hypermedia-Links zu den nächsten Zuständen der Antwort hinzufügen.

Authorization-aware HATEOAS erweitert diesen Ansatz um die Integration von Zugriffskontrolle. Es werden nicht einfach alle möglichen Zustandsübergänge der Antwort hinzugefügt, sondern lediglich diejenigen, welche auch vom Client durchgeführt werden können. Genau genommen werden diejenigen nicht hinzugefügt, welche der Client sicher nicht ausführen darf, da ausschließlich statische Attribute für die Bestimmung der Zugriffsberechtigungen in der Zukunft heran gezogen werden können. Attributbedingungen mit dynamischen Attributen werden nicht berücksichtigt und es wird ein Hyperlink hinzugefügt.

Das Zugriffskontrollsystem evaluiert eine Anfrage und wenn die Entscheidung lautet, den Zugriff zu gewähren oder der Zugriff kann möglicherweise in der Zukunft gewährt werden, wird ein Link-Header vom Authorization-Manager hinzugefügt. Im Produktszenario könnte der Kunde auf die Produktressource mit einem lesenden Zugriff mittels einer *GET*-Anfrage zugreifen. Diese Anfrage ist in Auflistung 6.4 dargestellt.

```
1 GET /products/1 HTTP/1.1
2 Host: example.org
```

Auflistung 6.4: HTTP-Anfrage

Ohne Authorization-aware HATEOAS würde der Server für alle Clients die gleiche Antwort ausliefern samt aller bekannten Zustandsübergänge, unabhängig von den Zugriffsrechten der Clients. Diese

Antwort ist in Auflistung 6.5 abgebildet.

```
1 HTTP/1.1 200 OK
2 Link: </products/1>; verb="Put"
3 Link: </products/1/parts>; verb="Get,Post,Put"
4 Link: </products/1/parts/1>; verb="Get,Put"
```

Auflistung 6.5: Autorisierungsunabhängige Antwort

Wird Authorization-aware HATEOAS integriert, so kann die Ressource auf den Benutzer zugeschnitten werden. Der Zuschnitt hängt dann von verschiedenen Attributen ab, etwa dem internen Status der Ressource oder dem Typ des anfragenden Subjekts. Fragt beispielsweise der Kunde die Produktressource an, so wird der Hypermedia-Link zum Aktualisieren der Ressource nicht integriert, weil der Kunde niemals die Ressource aktualisieren darf (vgl. Tabelle 6.1). Gleiches gilt für das Aktualisieren der Liste von Komponenten. Auch diese darf der Kunde nicht aktualisieren. In Auflistung 6.6 ist die maßgeschneiderte Antwort für den Kunden dargestellt. Im Vergleich mit der nicht maßgeschneiderten Variante fehlt der Link-Header zum Aktualisieren der Produktressource und auch die Aktualisierungsoperation im Link-Header der Komponentenliste fehlt. Diese Ressourcen werden durch Policy *P2* geschützt, welche bei einem Zugriff durch den Kunden nicht anwendbar ist.

```
1 HTTP/1.1 200 OK
2 Link: </products/1/parts>; verb="Get,Post"
3 Link: </products/1/parts/1>; verb="Get,Put"
```

Auflistung 6.6: Maßgeschneiderte Antwort Kunde

Eine vollkommen andere Repräsentation der Ressource erhalten die Mitarbeiter der Fabrik (vgl. Auflistung 6.7). Hier ist der Link zum

Aktualisieren der Ressource in der Antwort enthalten. Auch die Aktualisierungsoperation für die Komponentenliste ist in der Antwort integriert. Aber es fehlt die Operation zum Anlegen neuer Komponenten. Die Operation ist dem Kunden vorbehalten. Die *POST*-Methode für die Ressource */products/1/parts* ist nicht möglich. Diese Ressource werden durch Policy *P1* geschützt, welche bei einem Zugriff durch den Mitarbeiter nicht anwendbar ist.

```
1 HTTP/1.1 200 OK
2 Link: </products/1>; verb="Put"
3 Link: </products/1/parts>; verb="Get,Put"
4 Link: </products/1/parts/1>; verb="Get,Put"
```

Auflistung 6.7: Maßgeschneiderte Antwort Mitarbeiter

Es wurde die Verwendung des *Link*-Headers erweitert und das *verb*-Property hinzugefügt. Dies ist ein einfacher Weg um dem Client mitzuteilen, welche Methoden verwendet werden können, um auf die Ressource zugreifen zu können.

Die Verwendung des *Link*-Headers ist ausreichend in der Maschine-zu-Maschine-Kommunikation. Sind menschliche Benutzer die anfragenden Subjekte, so muss die Repräsentation der Ressource über ein anschauliches User Interface erfolgen. Hierzu ist ein weiterer Renderer notwendig, der beispielsweise HTML in Abhängigkeit der integrierten *Link*-Header anpasst.

6.1.6 Experimentelle Ergebnisse

Durch die zusätzlichen Bearbeitungsaufwände entstehen Verzögerungen, welche eine kritische Größe darstellen können. Werden die Aufwände zu groß, so könnte das Autorisierungssystem zu einem Flaschen-

hals werden, welches die gesamte Anwendung verlangsamt. Deshalb ist es von großer Bedeutung, die Aufwände zu analysieren und in der Praxis zu messen. Die Gesamtverarbeitungszeit t einer Ressourcenanfrage mit autorisierungsabhängiger Prozessierung kann bestimmt werden als die Summe der Zeit t_1 für die Anfrage selbst plus die Zeit t_2 für die Prozessierung in autorisierungsabhängiger Weise:

$$t = t_1 + t_2 \tag{6.1}$$

$$t_1 = t_{network} + t_{resource} + t_{access} \tag{6.2}$$

$$t_2 = t_{model} + n * (t_{access} + t_{manipulation}) \tag{6.3}$$

Die Definition der einzelnen Zeitintervalle ist in Tabelle 6.5 aufgelistet. Darüber hinaus ist n die Nummer der möglichen nächsten Zustände, welche von der Ressource aus erreicht werden können.

Die erste Phase des 2-phasigen Autorisierungsprozess ist dargestellt in t_1 als t_{access}, während t_2 die autorisierungsabhängige Prozessierung beschreibt.

Um die Funktionalität zu prüfen und die Performanz zu messen, wurde ein Test-Setup erstellt. In diesem Setup sind der Resource Server und der Authorization Manager als einzelne Softwarekomponente implementiert. Zusätzlich wurden das Zugriffskontrollsystem und der Model Service auf der gleichen Maschine aufgesetzt. Um die Auswirkungen von Netzwerklaufzeiten zu eliminieren wurde auch der Client auf der gleichen Maschine ausgeführt. Die Tests wurden auf einem

$t_{network}$	Die Netzwerklaufzeit für Anfrage und Antwort zwischen anfragendem Subjekt und Server und die Zeit, die für die Verarbeitungs des unterliegenden Protokollstapels (etwa IP, TCP, HTTP) notwendig ist.
$t_{resource}$	Die Zeit, die benötigt wird, um eine Zugriffsanfrage vom Resource Server an den Authorization Manager zu schicken und die zugehörige Entscheidung durchzusetzen.
t_{access}	Die Zeit, die benötigt wird, um eine Zugriffsanfrage vom Authorization Manager an das Zugriffskontrollsystem zu schicken und die zugehörige Entscheidung zu berechnen.
t_{model}	Die Zeit, die benötigt wird, um eine Anfrage vom Authorization Manager an den Model Service zu schicken und die zugehörige Antwort zu empfangen.
$t_{manipulation}$	Die Zeit, die benötigt wird, um eine Antwort an den Client zu manipulieren.

Tabelle 6.5: Antwortzeiten

Quadcore-Rechner mit 2.90 GHz Taktfrequenz ausgeführt und 2 GB Arbeitsspeicher wurden ausschließlich für die Tests reserviert.

Es wurde eine Analyse anhand der RESTful Services aus dem Beispielszenario durchgeführt. Das bedeutet, dass die REST-API aus Tabelle 6.1 mit Hilfe von JAX-RS [79] erstellt wurde sowie das dazugehörige Metamodel. Das Zugriffskontrollsystem wurde aufgesetzt, um ein Produkt mit 2 Komponenten abzusichern. An dieser Stelle sei auf die detaillierte Analyse des RestACL-Systems im vorherigen Kapitel

verwiesen. Dort wurde gezeigt, dass die Verarbeitungsgeschwindigkeiten auf Seiten des Zugriffskontrollsystems unabhängig von der Anzahl der Ressourcen in der Domain ist. Deshalb ist für die Aufwandsberechnung bereits eine kleine Menge von Ressourcen ausreichend.

In den Tests wurden das Produkt, die Komponentenliste und die individuellen Komponenten angefragt. Für jede Ressource wurden verschiedene Anfragen erstellt, welche die Benutzertypen (*Customer* oder *Worker*) genauso wie die Ressourcenstatus (*Initial*, *In Production* oder *Completed*) abdecken. Jede Anfrage wurde mindestens 10-mal durchgeführt.

Tabelle 6.6 zeigt die Auswertung der Antwortzeiten. Minimale, maximale und durchschnittliche Antwortzeiten sind in Millisekunden abgebildet. Es ist zu sehen, dass die Antwortzeiten sehr klein sind und nur leichte Abweichungen existieren. Die größten Abweichung zwischen Minimum und Maximum treten beim Zugriffskontrollsystem auf. Aber auch hier liegt die Abweichung unterhalb von 2ms.

	$t_{resource}$	t_{access}	t_{model}	$t_{manipulation}$
min.	0.002	0.270	0.344	0.001
max.	0.015	2.073	0.792	0.031
avg.	0.003	0.532	0.445	0.009

Tabelle 6.6: Messergebnisse

Gemäß der obigen Gleichung kann die durchschnittliche Verzögerung für eine Ressource mit 6 Zustandsübergängen (wie das Beispiel aus Auflistung 6.5) berechnet werden als:

$$t_2 = 0.445ms + 6 * (0.532ms + 0.009ms) = 3.691ms$$

Etwa 3-4ms sind klein im Vergleich mit Netzwerklaufzeiten zwischen Client und Server. Dies gilt sogar in lokalen Netzwerken. Beispielsweise wurde t als 20ms gemessen im Test-Setup. Deshalb ist der zusätzliche Verarbeitungsaufwand vernachlässigbar. Dies kann sich allerdings ändern, wenn deutlich mehr als 6 Zustandsübergänge verarbeitet werden müssen. Ebenso kann sich die Manipulationszeit deutlich erhöhen, wenn die Repräsentation der Ressource für einen menschlichen Benutzer angepasst werden muss. In einem Maschine-zu-Maschine-Szenario reicht die Inklusion von *Link*-Headern aus, aber die Repräsentation der Ressource muss für menschliche Benutzer angepasst werden und höhere Verarbeitungsaufwände sind zu erwarten.

6.1.7 Stand der Forschung

Autorisierungsabhängige Repräsentationen von Ressourcen wurden bis dato sehr wenig in der Wissenschaft diskutiert. *Adaptive Hypertext and Hypermedia* ist ein Forschungsgebiet, welches sich mit dem Zuschneiden von Ressourcen auf den anfragenden Client beschäftigt. Dabei werden Benutzerkenntnisse und Vorlieben berücksichtigt [23]. Ansätze aus diesem Forschungsgebiet versuchen das Benutzererlebnis zu verbessern, indem sie dem Benutzer Repräsentation liefern, welche der Benutzer gerne sehen möchte [57, 85]. Deshalb fokussieren sich diese Ansätze darauf, den Benutzer zu modellieren. Die Umsetzung von Zugriffsrechten spielt in diesem Forschungsgebiet keine Rolle.

6.2 Situationsabhängige Zugriffskontrolle

Dieser Abschnitt zeigt ein weiteres Anwendungsgebiet von ABAC. Hier wird beschrieben, wie der Zugriff auf sensible Daten eines Sensornetzwerks situationsabhängig umgesetzt werden kann. Dazu wird der attributbasierte Zugriffskontrollmechanismus mit einem Situationserkennungssystem kombiniert. Dadurch wird es möglich, anhand aktueller Sensorwerte eine Situation zu erkennen und Zugriff in Abhängigkeit der erkannten Situation zu gewähren oder zu verweigern. Hierbei können sowohl statische als auch dynamische Attribute für die Bestimmung der Zugriffrechte verwendet werden.

Als attributbasierter Mechanismus kommt erneut RestACL zum Einsatz, da es in den Anwendungsszenarien um den Schutz von Web Services geht. Allerdings ist der Ansatz nicht zwangsläufig an RestACL geknüpft. Die Ideen und Lösungen ließen sich auch mit einem anderen attributbasierten Zugriffsmechanismus wie etwa XACML umsetzen. Neben dem Zugriffskontrollsystem basiert der situationsabhängige Ansatz auf einem Situationserkennungssystem. Das Situationserkennungssystem entdeckt in IoT-Umgebungen das Auftreten bestimmter Situationen. Eine solche Umgebung besteht im Regelfall aus verschiedenen Geräten, welche unterschiedliche Sensoren aufweisen. Basierend auf den Sensorwerten, können Zustände der Umgebung entdeckt werden. Aus der Aggregation von diesen Zuständen können Situationen abgeleitet werden. In diesem Kontext kann eine Situation also als *Statuswechsel innerhalb der IoT-Umgebung* definiert werden. Beispielsweise kann die Situation *Serverraum überhitzt* mittels eines Temperatursensors erkannt werden. Sobald die Sensorwerte einen bestimmten Grenzwert überschreiten und damit eine Statusänderung eintritt, tritt die Situation auf und das Situationserkennungssystem meldet

dies. SitOPT [46, 56, 148] ist ein solches Situationserkennungssystem, welches Situationen anhand von Low-Level-Sensordaten erkennt.

Anwendungsszenarien Im Folgenden werden zunächst zwei Anwendungsszenarien beschrieben in denen ein situationsabhängiges Zugriffskontrollsystem benötigt wird.

Szenario 1: Ambient Assisted Living Eines der Anwendungsszenarien ist die Unterstützung im Bereich von Ambient Assisted Living [62]. In diesem Szenario können mögliche Notfallsituationen, wie etwa der Sturz einer älteren Person, durch die Aggregation von Sensordaten erkannt werden. Dazu können beispielsweise Beschleunigungs- oder Bewegungssensoren eingesetzt werden. Sobald die Situation auftritt, kann ein Situationserkennungssystem automatisch erkennen, dass ein Notfall vorliegt und eine Benachrichtigung an einen Notfallkontakt verschicken. Allerdings ist es so, dass Situationserkennung keine vollkommen verlässliche Erkennungsrate haben kann. Deshalb ist es sinnvoll zu überprüfen, ob tatsächlich eine Notfallsituation vorliegt. Ist die Ambient Assisted Living-Umgebung mit einer oder mehreren Kameras ausgestattet, kann über die Bilder der Kameras von außerhalb geprüft werden, ob die Notfallsituation tatsächlich vorliegt. Im Szenario müssen drei unterschiedliche Zugriffsfälle betrachtet werden.

- Ein besorgtes Familienmitglied ist berechtigt, jederzeit auf die Bilder der Kamera zuzugreifen.
- Ein Rettungsdienst bekommt automatisch Zugriff, wenn eine Notfallsituation vorliegt. Diese Zugriffsberechtigung erlischt,

sobald die Situation nicht mehr gegeben ist oder nach einem bestimmten Zeitintervall.

- Andere Personen dürfen niemals auf die Kamera zugreifen. Andernfalls wäre es beispielsweise für Einbrecher sehr einfach, das Opfer und Wertgegenstände auszuspähen. Insbesondere könnte ein Einbrecher abwarten, bis das Opfer das Haus verlässt und danach gezielt in der Wohnung nach Wertgegenständen suchen.

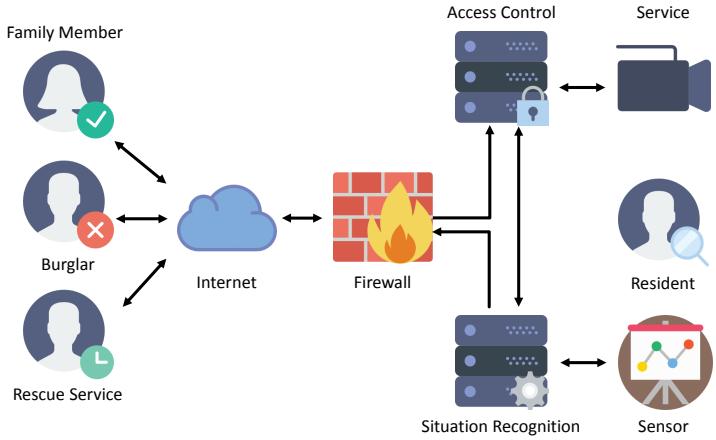

Abbildung 6.5: Situationsabhängige Zugriffskontrolle im Ambient Assisted Living (Quelle: [62])

Abbildung 6.5 zeigt das Ambient Assisted Living Szenario. Die drei unterschiedlichen Arten des Zugriffs (permanent gewährt, tempo-

rär gewährt, permanent verboten) aus diesem Szenario werden über die drei Personen auf der linken Seite abgebildet. Alle Personen können versuchen über das Internet auf die Services zuzugreifen. Eine Firewall blockiert den Zugriff für nicht registrierte Personen. Registrierte Personen können Anfragen an den Service senden. Diese Anfragen werden von einem Zugriffskontrollsystem abgefangen und entweder autorisiert oder abgeblockt. Das Situationserkennungssystem erkennt Situationen basierend auf verschiedenen Sensorwerten und sendet Benachrichtigungen an registrierte Benutzer. Beispielsweise könnten sich das Familienmitglied und der Rettungsdienst für Benachrichtigungen im Falle einer Notfallsituation registriert haben. Darüber hinaus informiert das Situationserkennungssystem auch das Zugriffskontrollsystem über das Auftreten von Situationen. Hierdurch wird es möglich, den Zugriff situationsabhängig zu steuern.

Szenario 2: Industrie 4.0 Ein zweites Szenario wird durch das IoT selbst eröffnet [63]. Die Vorteile des IoT ermöglichen Smart Applications (wie etwa Smart Factories [86]). Aber mit den Chancen steigen auch die Sicherheitsrisiken. Das hier vorgestellte Szenario basiert auf [58]. In schnellen, autonomen Produktionsumgebungen, sind nicht nur Echtzeitanforderungen essentiell, auch Sicherheitsaspekte müssen bedacht werden. Ausschließlich autorisiertem Personal oder autorisierten Softwarekomponenten darf Zugriff auf typische Ressourcen wie Sensoren, Antriebselemente oder Prozessinformationen gewährt werden. Die Folgen von nicht autorisierten Zugriffen können vom Verlust von Firmengeheimnissen bis zur Sabotage der Produktion reichen. Oft ist der Fall gegeben, dass der Zugriff auf eine Ressource vom aktuellen Kontext abhängt, also der Situation. Beispielsweise ist es

vorstellbar, dass ein externer Wartungsingenieur entfernten Zugriff auf Maschinen in der Produktionsumgebung bekommt, wenn ein Problem auftritt. In anderen Situationen soll die Produktionsumgebung nicht aus der Ferne kontrollierbar sein. Insbesondere die Einsehbarkeit und die Steuerbarkeit von Robotern müssen kontrolliert werden. Eine solche Zugriffssteuerung erfordert situationsabhängige Zugriffskontrolle.

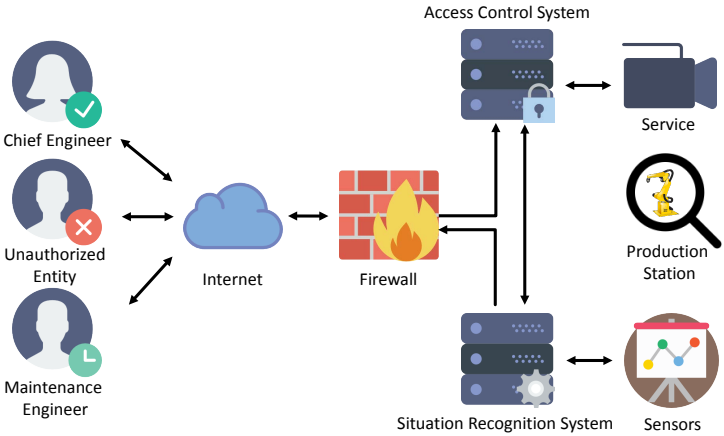

Abbildung 6.6: Situationsabhängige Zugriffskontrolle für Industrie 4.0 (Quelle: [63])

In diesem Szenario steht ein Produktionsroboter eines Automobilherstellers im Zentrum. Der Roboter schneidet Metallteile, welche zur Herstellung eines Fahrzeugs benötigt werden. Während des Schneidens kann es zu folgenden Situationen kommen:

- Das Metallteil wird falsch auf der Schneidefläche platziert, so dass es nicht den geforderten Zuschnitt aufweist.
- Der Laser kann beim Schneiden überhitzen und infolgedessen nicht mehr in der Lage, sein korrekte Schnitte auszuführen.
- Das Metallteil wird korrekt zugeschnitten, kann aber nicht aus dem Rahmen entnommen werden und blockiert die weitere Verwendung des Roboters.

Alle drei Situationen können mit Hilfe eines Situationserkennungssystems erkannt werden. Um nun die Erkennung zu verifizieren, kann der externe Wartungsingenieur Zugriff auf verschiedene Ressourcen bekommen. Hierdurch wird es möglich, das Problem unmittelbar anzugehen. Beispielsweise könnte erneut Zugriff auf eine Kamera gewährt werden, welche die Maschine überwacht oder auf den Temperatursensor des Lasers. Der Wartungsmitarbeiter kann dann die notwendigen Schritte zur Behebung des Problems einleiten. Abbildung 6.6 zeigt dieses Szenario.

Zugriff auf hochsensible Informationen sollte nur dann gewährt werden, wenn bestimmte Situationen vorliegen. Informationen über die internen Abläufe und Produktionsprozesse können beispielsweise über den Kamerazugriff bekannt werden. Deshalb bedarf es einer strengen Zugriffskontrolle. Im Beispielszenario sind drei Fälle zu unterscheiden.

- Interne Ingenieure können zu jeder Zeit auf die Kamera zugreifen.
- Der externe Wartungsingenieur erhält nur im Fehlerfall Zugriff.

- Nicht autorisierte Mitarbeiter und andere Subjekte dürfen niemals auf die Kamera zugreifen.

Situationserkennung Abbildung 6.7 zeigt den SitOPT-Ansatz, welcher verschiedene Komponenten zur Situationserkennung beinhaltet. Die beiden Kernkomponenten sind der Situation Recognition Service (SitRS) und die Resource Management Platform (RMP). Die SitRS sind dazu in der Lage, Situationen basierend auf Situation Templates zu erkennen. Ein Situation Template ist ein Modell zur Definition von Bedingungen. Präziser ausgedrückt knüpft das Template Sensorwerte an Bedingungen, welche mit logischen Operatoren wie AND, OR oder XOR verbunden werden. Die RMP fungiert als ein Gateway zu den Sensoren und bietet verschiedene Adapter, um Sensorwerte anzubinden. Hierdurch werden Sensorwerte über eine einheitliche Schnittstelle in Form von RESTful Services zugänglich [54, 55].

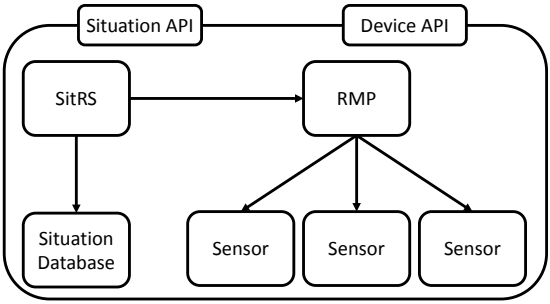

Abbildung 6.7: Architektur von SitOPT (Quelle: [63])

Ein registriertes Gerät kann einen oder mehrere Benutzer haben. Initial wird nur den Besitzern des Geräts Zugriff auf das Gerät gewährt. Sobald ein Gerät angebunden wurde, können Benutzer oder Anwendungen Situationen registrieren, welche sich auf Sensoren des Geräts beziehen. Hierzu muss zunächst ein Situation Template angelegt werden mit den Bedingungen für Sensorwerte. Dieses ist ein Eingangsparameter für den SitRS, welcher das Template in eine ausführbare Repräsentation transformiert. Die Repräsentation wird in einer Laufzeitumgebung ausgeführt, so dass die Situationen permanent überwacht werden. SitOPT bietet zwei Schnittstellen zu Anwendungen an. Die Situation API ermöglicht die Registrierung von Situation Templates inklusive Benachrichtigungen für den Fall des Auftretens. Die Device API erlaubt die Registrierung neuer Geräte, welche durch die SitRS überwacht werden sollen. Für den Fall, dass eine Situation aufgetreten ist, wird eine Benachrichtigung an einen Callback verschickt und das Auftreten der Situation wird in einer Datenbank (Situation Database - SitDB) protokolliert.

6.2.1 Situationsabhängige Zugriffskontrolle

Ein Zugriffskontrollmechanismus in einem solchen Szenario muss sehr effizient und flexibel sein. Dies bedeutet, das System muss in der Lage sein, Zugriffsentscheidungen in sehr kurzer Zeit zu treffen, um sicherzustellen, dass die Servicedaten aktuell sind. Diese Eigenschaft muss auch noch für große Mengen von Diensten und Web Services möglich sein. Weiterhin muss das System flexibel hinsichtlich dem Hinzufügen und Entfernen von Diensten sein. Es muss also einfach möglich sein, Dienste zu erstellen, zu ändern oder zu entfernen. Das Gleiche gilt für die Regelwerke, welche den Zugriff auf die Dienste

und die produzierten Daten kontrollieren. Aus den beiden unterschiedlichen Anwendungsszenarien wird auch deutlich, dass das System dazu in der Lage sein muss, eine große Bandbreite an Regelwerken zu unterstützen. Die folgenden Anforderungen müssen erfüllt werden, um ein situationsabhängiges Zugriffskontrollsystem zu ermöglichen.

(R5) *Schnelle Reaktionszeiten*: Der Status von Geräten, Sensoren und anderen Ressourcen kann sich schnell und häufig ändern und somit auch die Zugriffsberechtigungen. Deshalb muss das Zugriffskontrollsystem in der Lage sein, Änderungen sehr schnell zu adaptieren. Wenn ein Gerät seinen Status ändert und unmittelbar danach erfolgt ein Zugriffsversuch, so müssen die Zugriffsberechtigungen in Abhängigkeit des aktuellen Status berechnet werden. Deshalb muss das System eine enge Integration in den situationsabhängigen Kontext ermöglichen. Zusätzlich müssen Änderungen schnell angenommen werden.

(R6) *Ausdrucksstärke*: Ein situationsabhängiges Zugriffskontrollsystem muss hohe Flexibilität und feingranulare Zugriffskontrolle ermöglichen. Aus den unterschiedlichen Anwendungsszenarien wird deutlich, dass eine große Bandbreite verschiedener Attribute abgedeckt werden können muss.

(R7) *Integration und Administration*: Die Anbindung von neuen Geräten, Sensoren und Templates muss genauso möglich sein wie die einfache Administration von existierenden Geräten, Sensoren und Templates. Hierfür bedarf es einer sorgfältig ausgewählten Menge von initialen Zugriffsberechtigungen, welche während der Registrierung erstellt werden. Ebenfalls ist eine strukturierte Schnittstelle zur Verwaltung von Zugriffsberechtigungen notwendig, die durch Menschen und automatisierte Systeme gleichermaßen genutzt werden kann.

Die Anforderungen **(R5)** und **(R6)** werden durch das RestACL-System und die in Kapitel 4 beschriebenen Anforderungen größtenteils

abgedeckt. Lediglich die Integration der Zugriffskontroll- und Situationserkennungssysteme muss nun derart gestaltet werden, dass die Merkmale des Zugriffskontrollsystems, welche die Erfüllung der Anforderungen garantieren, erhalten bleiben.

Die Situationskategorie Da ABAC ein idealer Kandidat für flexible Zugriffskontrolle ist, kann der situationsabhängige Zugriffkontrollmechanismus auf dem ABAC-Modell aufgebaut werden. Grundlage hierfür ist die Einführung einer *Situation*-Kategorie, die ganz analog zu *Subjekt* oder *Ressource* eine eigene Kategorie sein kann. Entitäten dieser Kategorie haben bestimmte Attribute. Beispielsweise informiert das SitOPT-System das Zugriffskontrollsystem, wenn eine Situation auftritt und wenn eine Situation nicht mehr vorliegt. Es wird also die Änderung im Vorliegen einer Situation signalisiert. Deshalb haben Situationsentitäten *occurred*- und *time*-Attribute, welche festhalten, ob eine Situation gerade vorliegt und wann die letzte Änderung war. Diese Attribute werden vom SitRS an das RestACL-System geliefert. Das RestACL-System verwendet seine Attribute Provider, um diese Informationen zu speichern.

Da der Zugriff nach dem Auftreten einer Situation für eine bestimmte Zeit gelten soll, benötigen die Situationsentitäten zusätzlich ein *accessInterval*-Attribut. Beispielsweise kann ein Subjekt während des Registrierungsprozesses ein Regelwerk anlegen, welches Zugriff gewährt, wenn die Situation in den letzten 20 Minuten aufgetreten ist. Das *accessInterval*-Attribut ist sozusagen eine Stoppuhr, welche nach einer bestimmten Zeitspanne abläuft. Sobald die Stoppuhr abgelaufen ist, wird der Zugriff nicht weiter gestattet. Das *accessInterval*-Attribut wird beim Registrierungsprozess eines Situation Template automatisch

gesetzt, kann aber jederzeit durch das Subjekt, welches das Situation Template registriert hat, angepasst werden.

Die Verwendung des *occurred*-Attribut kann dazu genutzt werden, Zugriffsberechtigungen zurückzunehmen, bevor das *accessInterval* abgelaufen ist. Beispielsweise sendet das SitOPT-System eine Benachrichtigung, sobald eine Situation nicht mehr vorliegt. Das *occurred*-Attribut wird dann auf *false* gesetzt und die situationsabhängigen Zugriffsberechtigungen ändern sich.

Für das zweite Beispielszenario hängt die Zugriffsentscheidung bei einer Anfrage des Wartungsgingenieur unter anderem von den Attributen *occurred*, *time* und *accessInterval* der Situation ab. Beispielweise könnten diese Attribute die in Tabelle 6.7 dargestellten Werte annehmen.

Att.	**Category**	**Designator**	**Value**
a_1	situation	id	123
a_2	situation	occurred	true
a_3	situation	time	12:00:00 01.01.2019
a_4	situation	accessInterval	1200000 ms

Tabelle 6.7: Beispiel Situationsentität

Es ist zu beachten, dass die Situationsentität immer die drei Attribute *occurred*, *time* und *accessInterval* haben muss, andernfalls kann keine situationsabhängige Berechnung der Zugriffsentscheidung stattfinden. Ebenfalls muss vom Zugriffskontrollsystem die Zeit festgehalten werden, zu der die Zugriffsanfrage eingetroffen ist. Trifft die Anfrage beispielsweise um *12:10:00 30.06.2019* ein, so wird das *time* Attribut gemäß Tabelle 6.8 gesetzt.

Att.	Category	Designator	Value
a_5	environment	time	12:10:00 30.06.2018

Tabelle 6.8: Beispiel Environment-Attribut

Alleine die Präsenz der Attribute ist jedoch noch nicht ausreichend um situationsabhängig Zugriff zu gewähren. Die Attributwerte müssen in einer bestimmten Konstellation vorliegen. So muss der Zeitpunkt, zu dem die Zugriffsanfrage eingetroffen ist, in das *accessInterval* fallen. Formeller ausgedrückt müssen für die Attribute die folgenden Bedingungen gegeben sein:

$$v_{a_2} = true \tag{6.4}$$

und

$$v_{a_3} < v_{a_5} < v_{a_3} + v_{a_4} \tag{6.5}$$

6.2.2 Systemarchitektur

Abbildung 6.8 zeigt die notwendigen Komponenten um Situationserkennung und Zugriffskontrolle zu situationsabhängiger Zugriffskontrolle zu integrieren. In der Abbildung wird zwischen vier Schichten unterschieden. In der untersten Schicht wird die reale Welt abgebildet. Das heißt, hier sind physische Objekte wie Geräte und Sensoren angesiedelt. Darauf aufgebaut werden Service-Schicht, Sicherheits-Schicht und Client-Schicht. Diese werden mit RESTful Services gekoppelt.

Die Service-Schicht beinhaltet die zu schützenden Services. Beispielsweise sind hier die Kamera-Services aus den Anwendungsszena-

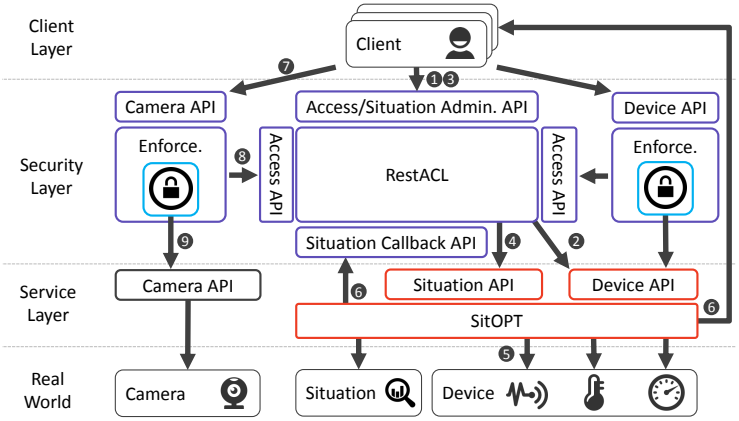

Abbildung 6.8: Systemarchitektur für situationsabhängige Zugriffskontrolle

rien zu finden. Ganz allgemein werden hier alle Services angesiedelt, welche permanente oder situationsabhängige Zugriffskontrolle benötigen. Es ist zu beachten, dass der SitRS eine Reihe von Diensten zum Verwalten der Situationserkennung anbietet. Diese beinhalten beispielsweise die Registrierung von Sensoren oder Situation Templates oder den Zugriff auf die Geräte, welche von der RMP überwacht werden. Diese Dienste finden sich ebenfalls in der Service-Schicht wieder.

Die Security-Schicht ist dafür verantwortlich, dass lediglich berechtigte Subjekte Zugriff auf die Dienste der darunter liegenden Schicht bekommen. Hierfür werden Enforcement Points eingesetzt, welche

die eingehenden Anfragen kontrollieren. Dafür fängt der Enforcement Point die Anfrage an den Service ab, leitet daraus eine Zugriffsanfrage ab und schickt diese an das RestACL-System. Das RestACL-System identifiziert die Policies, die ausgewertet werden müssen. Dabei werden Identifier aus der Zugriffsanfrage verwendet, um Attribute bei Attribute Providern zu erfragen. Nachdem die Zugriffsentscheidung berechnet wurde, wird sie an den Enforcement Point geschickt, welcher die Anfrage entweder an die Service-Schicht weiterleitet oder die Anfrage zurückweist. Der Enforcement Point bietet die gleiche Service-Schnittstelle wie der darunter liegende Service und fügt die Durchführung von Zugriffskontrolllogik hinzu.

Wenn ein Client ein Gerät, einen Sensor oder ein Template registriert, so kann er dazu die Access/Situation Administration API verwenden. In der Sicherheits-Schicht wird überprüft, ob der Client berechtigt ist, diese Aktion durchzuführen (beispielsweise wird überprüft, ob der Client berechtigt ist, einen Sensor für ein bestimmtes Gerät zu registrieren). Ist dies der Fall, so leitet das RestACL-System die Anfrage an das SitOPT-System weiter. Registriert der Client ein Situation Template, so registriert sich auch das RestACL-System als Callback, um über Situationsänderungen informiert zu werden. Tritt eine Situation auf, wird das RestACL-System informiert. Das System passt die Situationsattribute an, indem bei den Attribute Providern die entsprechenden Werte hinterlegt werden. Sobald eine Zugriffsanfrage eintrifft, kann das Zugriffskontrollsystem dann basierend auf aktuellen Situationsdaten entscheiden.

Das situationsabhängige Zugriffskontrollsystem funktioniert wie in Abbildung 6.8 dargestellt:

1. Ein Client registriert Geräte (inklusive Sensoren) über die Ac-

cess/Situation Administration API. Das RestACL-System legt initiale Attribute für die Geräte und Sensoren an.

2. Das RestACL-System leitet die Anfrage an das SitOPT-System weiter.
3. Ein Client registriert ein Situation Template über die Access/-Situation Administration API. Das RestACL-System legt eine Situationsentität an und speichert diese bei einem Attribute Provider.
4. Das RestACL-System leitet die Anfrage zum Anlegen des Situation Templates an das SitOPT-System weiter und registriert sich selbst als Callback.
5. Das SitOPT-System überwacht die Geräte und Sensoren hinsichtlich des Auftretens der Situation.
6. Wenn die Situation auftritt, informiert der SitRS alle Callbacks (inklusive des RestACL-Systems) über das Eintreten der Situation. Das RestACL-System aktualisiert die Daten der Situationsentität beim zugehörigen Attribute Provider.
7. Ein Client versucht einen Service aufzurufen und schickt eine Anfrage an den Service.
8. Der zugehörige Enforcement Point fängt die Anfrage ab und überprüft über das RestACL-System, ob der Client die Berechtigung hat.
9. Der Enforcement Point leitet die Anfrage in Abhängigkeit der Zugriffsentscheidung an den Service weiter oder blockiert die Anfrage.

Als Teil der Integration der beiden Systeme, wird eine Administrationskomponente benötigt, welche es verschiedenen Clients ermöglicht Policies, Attribute und die Zuordnung zu Geräten und Sensoren zu kontrollieren. Clients möchten beispielsweise Geräte registrieren und die Zugriffsberechtigungen dazu verwalten. Die Access/Situation Administration-Komponente ermöglicht dies. Hier können verschiedene Operationen zum Verwalten von Zugriffsberechtigungen und Attributen für Geräte und Sensoren durchgeführt werden. Ebenfalls wird eine Situation Callback API benötigt, welche es dem Situationserkennungssystem ermöglicht Situationsänderungen an das RestACL-System zu melden.

6.2.3 Dienste und Schnittstellen

Da das situationsbezogene Zugriffskontrollsystem verschiedene Dienste anbietet, können Clients über mehrere Aktionen mit dem System interagieren. Zunächst sind dies Aktionen zur Situationserkennung wie etwa das Registrieren von Geräten, Sensoren und Situation Templates. Darüberhinaus existieren Aktionen zur Steuerung der Zugriffskontrolle, welche durch das RestACL-System in der Access/Situation Administration API angeboten werden. Schließlich können Clients noch auf die Web Services zugreifen, für die sie die entsprechenden Berechtigungen haben. Diese Web Services können Anwendungen wie etwa die Kamera sein oder einen direkten Zugriff auf jene Geräte abbilden, die Sensordaten zur Situationserkennung liefern. Die Zugriffe auf die Anwendungen und Geräte erfolgt über die jeweilige Application API (beispielsweise die Camera API) beziehungsweise Device API. Das Situationserkennungssystem hingegen nutzt die Situation Callback API zum Aktualiseren von Situationsdaten.

Dienst	Kategorie	Beschreibung
Register situation	Situation	Clients registrieren Situationen, in denen sie benachrichtigt werden wollen.
Deregister situation	Situation	Clients deregistrieren Situationen.
Register device	Situation	Clients registrieren Geräten, die zur Situationserkennung notwendig sind.
Deregister device	Situation	Clients deregistrieren Geräte.
Register sensor	Situation	Clients registrieren Sensoren eines Geräts, welche bei der Situationserkennung verwendet werden.
Deregister sensor	Situation	Clients deregistrieren Sensoren.
Access sensor values	Device	Clients greifen auf aktuelle Sensordaten zu oder setzen Parameter von Antriebselementen oder ähnlichen Komponenten.

Tabelle 6.9: Schnittstelle zum Administrieren von Situationen und Geräten

Tabelle 6.9 listet die Dienste auf, die vom Client zum Administrieren der Situationserkennung genutzt werden können. Die Sicherheitsschicht fängt Aufrufe an diese Schnittstelle ab und fügt Zugriffskontrolle hinzu. Wenn ein Client eine Operation ausführen darf, wird die

Anfrage des Clients an die darunter liegende Serviceschicht weitergereicht. Dabei ist zu erwähnen, dass die RMP zusätzliche Operationen zum Aktualisieren von Sensorwerten durch die Sensoren selbst anbietet. Da diese Methoden nur für den Sensor konzipiert sind, werden die Operationen nicht in der Schnittstelle zur Client-Schicht angeboten. Da diese Services direkt verknüpft sind mit den jeweiligen Sensoren, besteht keine Notwendigkeit nach feingranularer Zugriffskontrolle.

Dienst	**Kategorie**	**Beschreibung**
Register services	Access	Clients registrieren Dienste, für die situationsbezogene Zugriffskontrolle notwendig ist.
Create attributes	Access	Clients erstellen Attribute für sich selbst und die Geräte und Sensoren, die sie besitzen.
Update, delete attributes	Access	Clients aktualisieren und löschen Attribute für sich selbst und die Geräte und Sensoren, die sie besitzen.
Create policies	Access	Clients erstellen Policies für den Zugriff auf Geräte und Sensoren, welche die gewünschte Zugriffslogik enthalten.
Update, delete policies	Access	Clients aktualisieren und löschen Policies für den Zugriff auf Geräte und Sensoren.

Fortsetzung auf nächster Seite

Fortsetzung von vorheriger Seite

Dienst	Kategorie	Beschreibung
Assign policies	Access	Clients ändern die Zuordnung von Policies und Geräten oder Sensoren. Policies können von einer Vielzahl von Geräten und Sensoren verwendet werden, während Attribute genau einer Entität zugeordnet sind.
Check policies	Access	Enforcement Points überprüfen die Zugriffsberechtigungen.

Tabelle 6.10: Schnittstelle zum Administrieren der Berechtigungen

Tabelle 6.10 listet die Zugriffskontrolloperationen auf, welche zum Administrieren der Zugriffskontrolle angeboten werden. Alle Operationen werden in der Security-Schicht bearbeitet. Da das Zugriffskontrollsystem auf RestACL basiert und die angebotenen Dienste als RESTful Services umgesetzt sind, kann das Zugriffskontrollsystem nicht nur dem Schutz des SitOPT-System dienen, sondern auch seine eigenen Services vor ungewünschtem Zugriff schützen.

Tabelle 6.11 beinhaltet die anwendungsbezogenen Operationen, die der Client ausführen kann. Diese Operationen sind in der Service-Schicht angesiedelt und beim Zugriff auf die Dienste findet über die Enforcement Points Zugriffskontrolle statt.

SitAC Service-Schnittstellen Die im vorherigen Abschnitt beschriebenen Dienste sind als RESTful Services implementiert. Es existieren

Dienst	Kategorie	Beschreibung
Access services	Applicat.	Clients interagieren (lesen, schreiben oder löschen) mit Web Services, wenn ihnen (situationsbezogener) Zugriff gewährt wird.

Tabelle 6.11: Anwendungsschnittstelle, wie etwa die Kamera API

drei Hauptressourcen (*/situations*, */services* und */devices*) mit verschiedenen untergeordneten Ressourcen, welche die Access/Situation Administration API, Device API und Situation Callback API implementieren. Die Access API hingegen als schichtinterne Schnittstelle zum Abfragen von Zugriffsberechtigungen ist bei der prototypischen Implementierung als Java-Bibliothek umgesetzt, welche RestACL-Anfragen als JSON-Objekte entgegen nimmt und mit RestACL-Anworten im JSON-Format antwortet.

Tabelle 6.12 zeigt die möglichen Operationen auf der */situations*-Ressource. Die Situationsliste (*/situations*) liefert bei einer GET-Anfrage eine vollständige Liste aller registrierten Situation Templates. Eine neues Situation Template kann mittels eines POST auf die Listenressource angelegt werden. Dabei muss eine Liste von Geräten angegeben werden, welche mit Hilfe des Situation Template überwacht werden sollen. Das RestACL-System überprüft dann, ob dem anfragenden Subjekt die Berechtigungen für den Zugriff auf das Gerät und seine Sensoren zugewiesen sind. Ist dies für mindestens einen Sensor nicht der Fall, so wird das Situation Template nicht angelegt und eine Fehlermeldung zurück geliefert.

Resource	Methoden	Beschreibung
/situations	GET, POST	Lesen der Liste der Situation Templates, Registrieren eines neuen Situation Template.
/situations/{id}	GET, DELETE	Lesen eines Situation Template. Löschen eines Situation Template.
/situations/{id}/access	PUT	Aktualisieren der Zugriffsberechtigungen für das Situation Template.
/situations/{id}/attributes	PUT	Aktualisieren der Attribute der Situation.

Tabelle 6.12: SitAC Situation API

Wenn das Anlegen gewährt wird, wird eine Situationsentität erstellt und bei einem Attribute Provider gespeichert. Die Situationsentität kann durch das SitOPT-System aktualisiert werden. Hierfür stellt das RestACL-System untergeordnete Ressourcen bereit. Über den Pfad */situation/{id}/attributes* kann das SitOPT-System die Attribute der Situationsentität aktualisieren. Wenn die Situation eintritt, kann das SitOPT-System eine PUT-Anfrage mit den aktuellen Daten für die Attribute *occurred* und *time* an die Ressource senden. Auch die Ressource */situation/{id}/attributes* benötigt Schutz vor unbefugtem Zugriff. Hierfür wird vom RestACL-System eine Policy erstellt, die ausschließlich dem SitOPT-System Zugriff gewährt.

Darüber hinaus existiert eine untergeordnete Ressource */situation/{id}/access*, über welche Zugriffsberechtigungen verwaltet werden können, indem der Zugriff auf die */situation/{id}*-Ressource gesteuert wird. Bei der untergeordneten */access*-Ressource können RestACL-Access-Elemente hinterlegt werden. Auch die */situation/{id}/access*-Ressource benötigt Schutz vor unbefugter Manipulation, so dass das RestACL-System ausschließlich dem Ersteller des Situation Template Zugriff auf diese untergeordnete Ressource gewährt.

Die zu schützenden Services sind über den Pfad */services* erreichbar. ähnlich wie die Situationsliste hat die Serviceliste die untergeordneten Ressourcen */attributes* und */access*, welche zum Abbilden der Zugriffskontrolle dienen. Auch hier dürfen lediglich die registrierenden Subjekte auf die untergeordneten Ressourcen zugreifen. Tabelle 6.13 zeigt die möglichen Operationen auf der */services*-Ressource.

Resource	**Methoden**	**Beschreibung**
/services	GET, POST	Lesen der Liste der Services, Registrieren eines neuen Services.
/services/{id}	GET, DELETE	Lesen eines Service. Löschen eines Services.
/services/{id}/access	PUT	Aktualisieren der Zugriffsberechtigungen für den Service.
/services/{id}/attributes	PUT	Aktualisieren der Attribute des Services

Tabelle 6.13: SitAC Service API

Die Geräteliste wird über den Pfad */devices* angesprochen. Wie bei Situation und Service existieren auch für die Geräte die untergeordneten Ressourcen */attributes* und */access*. Die möglichen Operationen auf der */devices*-Ressource sind in Tabelle 6.14 aufgelistet.

Resource	**Methoden**	**Beschreibung**
/devices	GET, POST	Lesen der Liste der Geräte, Registrieren eines neuen Geräts.
/devices/{id}	GET, DELETE	Lesen eines Geräts. Löschen eines Geräts.
/devices/{id}/access	PUT	Aktualisieren der Zugriffsberechtigungen für das Gerät.
/devices/{id}/attributes	PUT	Aktualisieren der Attribute des Geräts.

Tabelle 6.14: SitAC Geräte API

Allerdings gibt es weitere untergeordnete Ressourcen der */devices*-Ressource. Jedes Gerät kann eine Menge von Sensoren haben. Ein Sensor muss einen eindeutigen Namen haben. Der Zugriff und die Zugriffssteuerung für die Sensoren funktioniert wiederum analog zu Situation, Service und Gerät und ist in Tabelle 6.15 dargestellt.

6.2.4 Beispiele im Industrie 4.0 Szenario

In einer ressourcenorientierten Umgebung gibt es bei jeder Aktion mindestens zwei involvierte Entitäten: die angefragte Ressource und

Resource	Meth.	Beschreibung
/devices/{id}/sensors	GET, POST	Lesen der Liste der Sensoren, Registrieren eines neuen Sensors.
/devices/{id}/sensors/{name}	GET, DEL.	Lesen eines Sensors. Löschen eines Sensors.
/devices/{id}/sensors/{name}/access	PUT	Aktualisieren der Zugriffsberechtigungen für den Sensor.
/devices/{id}/sensors/{name}/attributes	PUT	Aktualisieren der Attribute des Sensors.

Tabelle 6.15: SitAC Sensor API

das anfragende Subjekt. Die angefragte Ressource ist einfach über die URI zu identifizieren. Zur Identifikation des anfragenden Subjekts ist jedoch eine Authentifikation notwendig.

Authentifizierung Authentifizierung kann über verschiedene Wege durchgeführt werden. Beispielsweise ist einer dieser Wege der Einsatz von standardisierten HTTP-Authentifizierungsmechanismen wie Basic

und Digest Authentication. Wichtiger als die Frage wie authentifiziert wird, ist es jedoch, dass authentifiziert wird. Wenn der Client sich nicht authentifizieren kann, gewährt das RestACL-System keinen Zugriff.

Sobald sich ein Subjekt authentifiziert hat, kann das Zugriffskontrollsystem die Zugriffsanfrage auswerten und bestimmen, ob das Subjekt Zugriff erhält. Hierfür erstellt der Enforcement Point eine Zugriffsanfrage inklusive der Adresse der angefragten Ressource (beispielsweise */devices/1/sensors/thermometer*), der Zugriffsmethode (beispielsweise *GET*) und einer Kennung des Subjekts. Anhand dieser Parameter kann das Zugriffskontrollsystem weitere Attribute aus seinen Attribut Providern laden. Beispielsweise kann das Subjekt ein Attribut *responsibility* mit dem Wert *maintenance* besitzen. Es existieren keinerlei Einschränkungen hinsichtlich der Verwendbarkeit von Attributen und auch eine Vielzahl von Attributquellen (Attribute Providern) sind möglich.

Registrierungsprozedur Entscheidet sich ein Subjekt dazu eine neue Komposition von Geräten zu registrieren, welche hinsichtlich bestimmter Situationen überwacht werden sollen, so muss das Subjekt zunächst alle Geräte der Komposition inklusive der Sensoren der Geräte (vgl. Abbildung 6.8 - Nachricht 1) registrieren. Hierfür muss für jedes Gerät eine Registrierungsanfrage an das Zugriffskontrollsystem gesendet werden. Die Anfrage muss eine eindeutige Gerätekennung, eine Liste von Besitzern und optional eine Gerätebeschreibung enthalten. Das RestACL-System limitiert den Zugriff auf diese Geräte dann auf die Subjekte aus der Besitzerliste. Danach wird eine Reihe von Einträgen für das Gerät in der *Domain* vorgenommen. Dies sind die Einträge für das Gerät selbst und die Verwaltung seiner Attribute,

Einträge zum Registrieren der Sensoren des Geräts und zum Verwalten derer Attribute sowie Einträge zum Ändern der Zugriffsberechtigungen. Zusätzlich speichert das RestACL-System ein Besitzerattribut für das Gerät bei einem Attribute Provider. Dabei ist zu beachten, dass ein Gerät mehrere Benutzer haben kann. In diesem Fall haben alle Besitzer Zugriff auf das Gerät und auch alle Besitzer können die Zugriffsrechte ändern. Nachdem alle diese Operationen erfolgreich durchgeführt wurden, leitet das RestACL-System die Registrierungsanfrage an das SitOPT-System weiter (vgl. Abbildung 6.8 - Nachricht 2).

Beispielsweise könnte der Chefingenieur aus dem Industrie 4.0 Szenario einen neuen Roboter zur Situationserkennung registrieren wollen. Der Chefingenieur wird mit der Kennung *1* identifiziert (als Ressource im System wird er damit über den Pfad */users/1* eindeutig identifiziert). Die Anwendung zur Registrierung sendet dann eine POST-Anfrage an das RestACL-System mit eben dieser Kennung und der Gerätekennung. Mit JSON-Daten als Übertragungsformat sieht die Anfrage wie folgt aus:

```
1  POST /devices HTTP/1.1
2
3  {
4    "deviceId" : "1234",
5    "deviceDescription" : "Robot that cuts metal parts",
6    "deviceOwners" : ["/users/1"]
7  }
```

Auflistung 6.8: Anlegen eines Geräts

Wenn eine solche Anfrage beim RestACL-System eintrifft, werden Policies angelegt, welche den Zugriff für die Subjekte aus der Besitzerliste erlauben. Die Policies werden im Policy Repository gespeichert.

```
1  {
2    "id": "P1",
3    "effect": "Permit",
4    "priority": "1",
5    "condition": {
6      "function": "equal",
7      "arguments": [
8        {"category": "subject","designator": "uri"},
9        {"value": "/users/1"}
10     ]
11   }
12 }
```

Auflistung 6.9: Zugriffsberechtigungen für das Gerät

Danach wird die Policy für den Zugriff auf das Gerät aktiviert, indem ein neuer Eintrag in der Domain angelegt wird. Der Zugriff per GET-Anfrage an das neue Gerät, wird über die neue Policy festgelegt:

```
1  {
2    "path": "/devices/1234",
3    "access": [
4      {"methods": ["GET"],"policies": ["P1"]}
5    ]
6  }
```

Auflistung 6.10: Zuweisung zwischen Gerät und Policy

In gleicher Weise wird die neue Policy für die Registrierung von Sensoren des neuen Geräts aktiviert. Die Registrierung erfolgt per POST-Anfrage an die untergeordnete */sensor*-Ressource des Geräts.

```
1 {
2   "path": "/devices/1234/sensors",
3   "access": [
4     {"methods": ["POST"],"policies": ["P1"]}
5   ]
6 }
```

Auflistung 6.11: Zuweisung zwischen Sensor und Policy

Ebenfalls wird die neue Policy für die untergeordnete */attributes* Ressource des Geräts aktiviert. Über diese Ressource können Besitzer dann die Attribute des Geräts verwalten. Werden nur externe Attribute Provider verwendet, so ist dieser Schritt nicht notwendig, da Subjekte dann nicht die Attribute ihrer Geräte ändern dürfen und deshalb auch keine Schnittstelle für diese Operation benötig wird.

In einem letzten Schritt wird die neue Policy schließlich für die untergeordnete */access*-Ressource des Geräts aktiviert. Über diese Schnittstelle können die Besitzer dann die Zugriffsberechtigungen für das Gerät verwalten. Das Ändern der Zugriffsberechtigungen ist initial deshalb nur den Besitzern des Geräts möglich. Möchte der Chefingenieur beispielsweise dem Wartungsingenieur Zugriff in Fehlersituationen gewähren, kann dem Roboter hier eine neue Policy für eben diesen Zugriff zugewiesen werden.

Abschließend wird die Besitzerliste in Form eines *deviceOwners*-Attributs gespeichert. Dieses Attribut wird zusammen mit allen anderen Attributen des Geräts beim Attribute Provider hinterlegt.

```
1  {
2    {
3      "category" : "resource",
4      "designator" : "id",
5      "value" : "/devices/1234"
6    },
7    {
8      "category" : "resource",
9      "designator" : "deviceOwners",
10     "value" : ["/users/1"]
11   }
12 }
```

Auflistung 6.12: Besitzerattribut

Die Service- und Sensorregistrierungsprozeduren sind der Geräteregistrierungsprozedur sehr ähnlich. Es gibt lediglich einen entscheidenden Unterschied bei der Registrierung von Sensoren. Sensoren sind mit einem Gerät assoziiert und können nur registriert werden, wenn Zugriffsberechtigungen für das Gerät existieren. Das heißt, eine POST-Anfrage an die untergeordnete */sensors*-Ressource des Geräts muss möglich sein. Hier muss das RestACL-System also bereits Zugriffsberechtigungen auswerten, was beim Registrieren von Geräten und Services nicht der Fall ist. Auch für Sensoren und Services wird eine initiale Policy erzeugt, welche den Zugriff auf die registrierenden Subjekte beschränkt. Die Zugriffsrechte können über die untergeordneten */access*-Ressourcen der jeweiligen Ressource von den Besitzern angepasst werden.

Situationsabhängige RestACL-Policies Möchte ein Subjekt den Zugriff auf einen bestimmten Service situationsabhängig gestalten, so muss das Subjekt zunächst ein Situation Template registrieren. Das Registrieren eines Situation Template ist nur möglich, wenn das registrierende Subjekt Zugriffsrechte für alle Sensoren hat, welche im Situation Template ausgewertet werden. Ist dies der Fall, so erstellt das RestACL-System eine Situationsentität und speichert diese beim Attribute Provider. Dabei werden Attribute hinterlegt, die beschreiben, ob die Situation vorliegt und wann zuletzt ein Wechsel des Status stattgefunden hat. Diese Informationen über die Situation (*occurred*, *time* und *accessInterval*) können in den RestACL-Policies ausgewertet werden. Für den Zugriff auf die Kamera aus dem Industrie 4.0 Szenario kann beispielsweise die Policy aus Auflistung 6.13 verwendet werden.

```
1  {
2    "id" : "PError",
3    "description" : "A policy that grants access in case
4      of an error situation. The policy can be applied
5      to the camera resource.",
6    "effect" : "Permit",
7    "priority" : "2" ,
8    "compositeCondition" : {
9      "operation" : "AND",
10     "conditions" : [
11       {
12         "function" : "equal",
13         "arguments" : [
14           {"category" : "subject", "designator" :
15                                        "responsibility"},
16           {"value" : "maintenance"}
17         ]
```

```
18        },{
19          "function" : "equal",
20          "arguments" : [
21            {"category" : "situation", "designator" :
22                                          "occurred"},
23            {"value" : "true"}
24          ]
25         },{
26          "function" : "between",
27          "arguments" : [
28            {"category" : "situation","designator" :
29                                        "time"},
30            {"category" : "environment","designator" :
31                                         "time"},
32            {
33              "function" : "add",
34              "arguments" : [
35                {"category" : "situation",
36                   "designator" : "time"},
37                {"category" : "situation",
38                   "designator" : "accessInterval"}
39              ]
40            }
41          ]
42        }
43      ]
44    }
45  }
```

Auflistung 6.13: Situationsabhängige Policy

Die Policy gewährt Zugriff, wenn drei Bedingungen erfüllt sind.

- Das anfragende Subjekt hat ein *responsibility*-Attribut mit dem Wert *maintenance*, welches beschreibt, dass es sich um einen Wartungsingenieur handelt.
- Die Situation muss aufgetreten sein. Das heißt, das *occurred*-Attribut der Situation muss den Wert *true* haben.
- Die Zeit, zu der der Zugriff auf die Kamera erfolgt, muss innerhalb des *accessInterval* liegen. Hierzu wird von der *situation time* ausgegangen und ein Zeitfenster bis zum Ablauf des *accessInterval* geöffnet. Der Zeitpunkt der Zugriffsanfrage wird über ein *environment time*-Attribut bestimmt. Alle drei Zeitpunkte werden als Argumente der *between*-Funktion ausgewertet.

Es ist leicht erkennbar, dass für ein korrektes Funktionieren verlässliche Attributquellen notwendig sind. Insbesondere stellt sich die Frage, welche Subjekte welchen Entitäten Attribute zuweisen dürfen. Dies ist von der Anwendung abhängig und lässt sich technisch über die Beschränkung des Zugriffs auf die untergeordneten */attributes*-Ressourcen steuern.

Die zweite und dritte Bedingung erfordern es zwingend, dass die Situationsattribute aktuell sind. Deshalb ist es von höchster Wichtigkeit, dass SitRS das RestACL-System informiert, sobald es Änderungen am Status einer Situation gibt, also einen Wechsel beim *occurred*-Attribut der Situation geschehen ist. SitRS kann dies über Callbacks erledigen. Dazu muss sich das RestACL-System als Callback für jede Situation registrieren. Ist dies geschehen, so werden die Informationen über die Situation aktuell gehalten und bei den Attribute Providern des RestACL-Systems gespeichert zur Auswertung bei Zugriffsanfragen.

Situationsabhängiger Zugriff In dem zuvor beschriebenen Szenario können standardisierte REST Clients eingesetzt werden, um Service-Daten von den Web Services zu erfragen. Diese Web Services werden durch einen vorgelagerten Enforcement Point geschützt. Dies bedeutet, der Client sendet eine Anfrage an die Kamera-Ressource, der Enforcement Point fängt diese Anfrage ab (vgl. Abbildung 6.8 - Nachricht 7). Hat der Client sich authentifiziert, so formuliert der Enforcement Point eine Zugriffsanfrage und leitet diese an das RestACL-System (vgl. Abbildung 6.8 - Nachricht 8). Das RestACL-System erfragt zusätzliche Attribute bei seinen Attributquellen und wertet die Zugriffsanfrage dann aus. Wird der Zugriff gewährt, so leitet der Enforcement Point die Anfrage an den Web Service weiter (vgl. Abbildung 6.8 - Nachricht 9). Andernfalls wird die Anfrage zurückgewiesen (beispielsweise durch eine HTTP 403 Antwort).

Die Weiterleitung einzelner Anfragen hängt von der Entscheidung des RestACL-Systems ab, welcher für jede neue Anfrage eine neue Bewertung vornimmt. Dies stellt sicher, dass Anforderung **(R5)** erfüllt wird. Sobald die Registrierungsprozeduren und die Zugriffskonfiguration erledigt wurden, werden alle Schritte automatisch innerhalb des situationsabhängigen Zugriffssystems erledigt. Versucht ein Client auf eine Resource zuzugreifen, muss er sich lediglich über Standardmethoden authentifizieren. Zusätzliche Interaktion mit dem Client ist nicht notwendig.

6.2.5 Experimentelle Ergebnisse

Um den situationsabhängigen Ansatz zu untersuchen wurden zwei Methoden eingesetzt:

- Zum einen wurde das implementierte System mit den zuvor aufgestellten Anforderungen abgeglichen.
- Zum anderen wurden Tests anhand der Szenarien vorgenommen.

Zu Evaluierungszwecken wurden das Situationserkennungssystem und das Zugriffskontrollsystem auf demselben Rechner installiert, so dass Netzwerklaufzeiten bei Messungen vernachlässigt werden können. Dies bedeutet, dass keine zusätzlichen Verzögerungen auftreten zwischen dem Erkennen einer Situation und der Anpassung der Zugriffsberechtigungen. Lediglich die Änderung von Attributwerten muss durchgeführt werden. Da dies eine äußerst einfache Operation ist, sind hierfür sehr kleine Antwortzeiten von weniger als 1ms notwendig. So wird gewährleistet, dass die richtigen Zugriffsberechtigungen unmittelbar nach dem Auftreten und Erkennen einer Situation verfügbar sind, so wie es in (**R5**) gefordert wird.

Es wurde eine Lösung erstellt, welche die Koexistenz von situationsabhängigen und situationsunabhängigen Regelwerken ermöglicht. Beide Arten von Regelwerken sind nicht auf ein bestimmtes Themenfeld eingeschränkt, sondern können für jede Art von Attribut eingesetzt werden. Neben Anwendungsgebieten wie Industrie 4.0 und Ambient Assisted Living sind etwa Smart Cities oder generell Event-gesteuerte Umgebungen interessante Einsatzbereiche. Attribute und Kategorien können frei gewählt und kombiniert werden, so wie es (**R6**) erfordert.

In den vorherigen Abschnitten wurde die Registrierungsprozedur im Detail beschrieben. Hierdurch wird deutlich, dass Geräte und Sensoren niemals ohne Zugriffsschutz in einer solchen Umgebung existieren, so wie es in (**R7**) gefordert wird.

Es wurde bereits erwähnt, dass unterschiedliche Formen des Zugriffs existieren. Zugriffsberechtigungen können permanent oder tem-

porär vergeben werden. Um die funktionale Korrektheit des Systems zu prüfen, wurde eine Reihe von Tests durchgeführt, welche die unterschiedlichen Formen des Zugriffs abbildet. Für jede Form (permanent erlaubt, permanent verboten, temporär erlaubt und temporär verboten) wurde überprüft, ob das System zu jedem Zeitpunkt die richtige Zugriffsentscheidung berechnet. Dies bedeutet, dass die richtige Entscheidung berechnet werden musste bevor die Situation eingetreten ist (s_1), nachdem die Situation eingetreten ist (s_2), nachdem das Zeitfenster für temporäre Zugriffe/Verbote abgelaufen ist (s_3) und nachdem die Situation nicht mehr vorliegt (s_4). Tabelle 6.16 listet die zu erwarteten Entscheidungen für jeden dieser Fälle. Es wurde erfolgreich verifiziert, dass das System in jedem dieser Fälle die richtige Entscheidung trifft.

Access Type	s_1	s_2	s_3	s_4
Permanently grant	permit	permit	permit	permit
Permanently forbid	deny	deny	deny	deny
Temporary grant	deny	permit	deny	deny
Temporary forbid	permit	deny	permit	permit

Tabelle 6.16: Zugriffsarten

6.2.6 Stand der Forschung

Situationsabhängige Zugriffskontrolle ist ein Thema, das bisher nur äußerst wenig in der Forschung diskutiert wurde.

Ein spezialisiertes Zugriffskontrollmodell wird in [17, 111] beschrieben. Der Fokus dieser Arbeit ist ein Modell für das Gesundheitswesen. Es wird ein Situationsschema beschrieben, welches auf

Patientendaten, Krankenakten und Data-Requestern basiert. Zugriff hängt von kontextuellen Parametern anstelle von Rollen ab, welche die Data-Requester inne haben. Es wird eine spezialisierte Lösung beschrieben, welche das RBAC-Modell um einen medizinischen Kontext erweitert.

Situationsbezogene Zugriffskontrolle für autonome, dezentralisierte Systeme wird in [151] beschrieben. Auch hier wird ein rollenbasierter Ansatz gewählt, um Zugriffskontrolle in dynamischen und skalierenden Systemen umzusetzen. Die Autoren erweitern das hierarchische RBAC-Modell aus [8]. Es werden Situationseinschränkungen eingeführt, welche auf die Relation zwischen Benutzer und Rolle einwirken.

6.3 Integration mit Tokenverfahren für den Einsatz im IoT-Umfeld

Im vorherigen Abschnitt wurde beschrieben, wie RestACL mit Situationserkennung kombiniert werden kann, um den Zugriff auf Dienste und IoT-Geräte zu steuern. Dabei wurden Enforcement Points eingesetzt, welche den Zugriff auf Dienste und Geräte unterbinden können, so dass keine unautorisierten Zugriffe auf die Dienste und Geräte erfolgen. In diesem Abschnitt wird ein auf Token basierender Mechanismus beschrieben, der dem gleichen Zweck dient. Dienste und Geräte können nur mit validen Token genutzt werden, welche von einem zentralen Autorisierungssystem ausgestellt werden, so dass keine unautorisierten Zugriffe erfolgen, auch wenn eine direkte Verbindung zum Dienst oder Gerät möglich ist. Der Ansatz wurde in [101] veröffentlicht.

Da heutzutage vermehrt IoT-Geräte zum Einsatz kommen, steigt der Bedarf nach Lösungen für die Zugriffskontrolle. Da bisherige Zugriffskontrollmodelle und -mechanismen für konventionelle Geräte und Umgebungen entwickelt wurden, zielt dieser Ansatz auf die Anpassung und Adaption von existierenden Lösungen ab. Die wesentliche Idee des Ansatzes ist die Auslagerung von Zugriffskontrolllogik, so dass auf leistungsschwachen IoT-Geräten keine komplexe Logik ausgeführt werden muss.

IoT-Anwendungen weisen eine große Heterogenität auf, was wiederum die Wahrung von Sicherheitsaspekten schwieriger macht [153]. Beispielsweise ist die physische Sicherheit von Maschinen und menschlichen Operateuren ein wichtiger neuer Aspekt, den es zu bedenken gilt, wenn ein IoT-System umgesetzt werden soll. IT-Sicherheitskonzepte sind oft für konventionelle Systeme mit hohen Rechen- und Speicherkapazitäten entworfen worden. IoT-Geräte hingegen bieten oft nur sehr begrenzte Ressourcen hinsichtlich CPU-Performanz, Speicherkapazität oder auch Energieversorgung [81]. Diese Limitierungen müssen berücksichtigt werden, wenn eine sichere Umgebung für IoT-Geräte geschaffen werden soll. Genau wie das Internet der Dinge selbst, müssen auch die Sicherheitsmodelle skalierbar, leichtgewichtig und verwaltbar sein [49].

Im folgenden wird untersucht, wie bewährte Technologien eingesetzt werden können, um in IoT-Umgebungen nutzbar zu sein, so dass gleichzeitig die Sicherheit verbessert wird und die Vorteile von IoT-Umgebungen erhalten bleiben. Weiterhin muss analysiert werden, wie die große Heterogenität und die Dynamik von IoT-Umgebungen mit flexiblen Regelwerken und dynamischer Konfiguration abgebildet werden können.

Internet der Dinge Das Internet der Dinge beschreibt die Integration und Vernetzung von Alltagsgegenständen mit eingebettetem System in das Internet [141]. Durch Kommunikationsfähigkeit über IP-Adressierung werden die physischen Objekte zu *Cyber-Physical Systems* (CPS) eines Netzwerks. Als Konsequenz sind die Geräte über Sensoren und Antriebselemente dazu in der Lage, Daten zu überwachen, auszuwerten, zu optimieren und zu steuern. Aber wie bereits erwähnt, fehlt es diesen Geräten an Charakteristika, welche traditionelles Equipment hat [49, 81]:

- **Rechengeschwindigkeit und Speicherkapazität:** Aus Kostengründen sind Geräte stark auf bestimmte Anwendungsfälle zugeschnitten. Deshalb weisen einfache Geräte oft nur geringe Rechen- und Speicherkapazitäten auf.
- **Energieversorgung:** Mobile Geräte werden oft über Batterien versorgt, so dass ihre Energieversorgung limitiert ist. Deshalb müssen die Geräte so wenig Energie wie möglich verbrauchen.
- **Bandbreite und Konnektivität:** Die Konnektivität wird beeinflusst, wenn sich Geräte mit hoher Geschwindigkeit bewegen. Kabellose Kommunikation ist oft mit geringerer Bandbreite verbunden.
- **Aktualisierbarkeit und Verfügbarkeit:** Aufgrund der Verteiltheit ist es eine Herausforderung, das Gesamtsystem mit Updates zu versehen. Ebenfalls ist es schwierig die Verfügbarkeit der Geräte zu gewährleisten.

Anwendungsszenario Ein Anwendungsgebiet für einen solchen Mechanismus ist beispielsweise ein Smart Home, indem Heizung, Licht oder auch Türen aus der Enfernung gesteuert werden können. So ist es beispielsweise denkbar, dass die Haustür oder das Garagentor automatisch entriegelt werden, sobald sich bestimmte Geräte in der Nähe befinden. Ebenfalls ist es denkbar, dass ein Hausbewohner auf einem Gerät wie etwa einem Smart Phone benachrichtigt wird, sobald die Türklingel betätigt wird. Der Hausbewohner könnte dann etwa einen Briefkasten entriegeln und es einem Postboten ermöglichen, Pakete zu deponieren. Hierfür werden verschiedene Regelwerke benötigt die beispielsweise Bedingungen wie *ist ein Hausbewohner* oder *ist in der Nähe des Hauses* kontrollieren können.

```
1  {
2    "uri":"https://www.example.com",
3    "resources":[{
4      "path":"/garage/state",
5      "access":[{
6        "methods":["GET, PUT"],
7        "policies":["P1"]
8      }]
9    }]
10 }
```

Auflistung 6.14: RestACL Domain zur Garagensteuerung

Auflistung 6.14 zeigt einen Ausschnitt aus der RestACL-Domain, die im Beispielszenario verwendet wird. Mit einer Garage als Gegenstand im Internet der Dinge kann über `GET`- oder `PUT`-Anfragen kommuniziert werden. Das Zulassen dieser Anfragen wird über die Policy `P1` kontrolliert. Eine `PUT`-Anfrage auf die Ressource öffnet oder

schließt das Garagentor. Über eine `GET`-Anfrage kann der aktuelle Status abgefragt werden, ob das Garagentor geöffnet oder geschlossen ist.

```
{
  "policies":[{
    "id":"P2",
    "effect":"permit",
    "priority":"1",
    "condition":{
      "function":"equal",
      "arguments":[{
        "category":"device",
        "designator":"code"
      },{
        "value":"123456789"
      }]
    }
  }]
}
```

Auflistung 6.15: RestACL Policy zur Garagensteuerung

Nun ist es denkbar, dass während der Abwesenheit der Hausbewohner die Nachbarn die Garage verwenden können. Hierfür könnte eine neue Policy verwendet werden, die in einem bestimmten Zeitraum Zugriff gewährt, wenn das anfragende Gerät ein Gerät des Nachbarn ist (identifiziert über eine Gerätekennung - `device code`). Diese Policy kann einfach zu der bereits existierenden Policy in der Domain hinzugefügt werden. Dabei muss lediglich beachtet werden, dass die Policies keine gleichen Prioritäten haben dürfen. Auflistung 6.15 zeigt eine Policy, welche die Gerätekennung prüft und den Zugriff auf eine

Ressource gewährt, wenn die Kennung einen bestimmten Wert hat.

Auflistung 6.16 zeigt eine Zugriffsanfrage, welche dem Nachbarn das Öffnen des Garagentors ermöglicht.

```
{
  "uri":"https://my.smarthome.com/garage/state",
  "method":"PUT",
  "attributes":[{
    "category":"device",
    "designator":"code",
    "value":"123456789"
  }]
}
```

Auflistung 6.16: RestACL Anfrage zur Garagensteuerung

6.3.1 Systemarchitektur

Die Kernidee des vorgestellten Ansatzes ist die Auslagerung von Zugriffskontrolle mittels etablierter Technologien wie OAuth. Hierdurch kann das IoT-Gerät entlastet werden und die Rechenanforderungen von einem leistungsstärkeren Netzwerkteilnehmer übernommen werden. Dabei müssen die Einschränkungen und Bedingungen in IoT-Umgebungen berücksichtigt werden. Zusätzlich zur Entlastung der IoT-Geräte hilft die Zentralisierung von Zugriffskontrolle dabei Redundanz zu minieren, so dass die Wartbarkeit erhöht und konsistente Verwaltung vereinfacht wird. Synchronisierungsaufwände durch Änderungen von Zugriffsberechtigungen werden minimiert. Grundlage des Ansatzes ist ABAC in Kombination mit einem *User-to-Token*-Ansatz [126], um die Authentifizierungs- und Autorisierungssaufwände auf

Seiten der IoT-Geräte gering zu halten. Als Zugriffskontrollmechanismus kommt RestACL zum Einsatz.

Die Architektur hat über die schon beschriebenen Ansätze hinaus die folgenden Grundideen und Annahmen.

- **Vertrauenswürdige Client-Anwendungen:** Die Architektur basiert auf einem Token-Verfahren, welches dem OAuth *Client Credential Grant* Flow (vgl. Abschnitt 2.4) entspricht. Dieser Flow erfordert vertrauenswürdige Client-Anwendungen, so dass diese Anforderung auch für den vorgestellten Ansatz gilt. Für die Client-Anwendungen ist also keine Autorisierung notwendig, sondern lediglich eine Authentifizierung.

- **Benutzerauthentifizierung:** Neben der Authentifizierung von Client-Anwendungen muss die Autorisierung von Benutzern berücksichtigt werden, wenn die Zugriffsberechtigungen von User-Attributen abhängt. Wenn der Client-Anwendung vertraut wird, so kann auch die Client-Anwendung die Benutzerauthentifizierung übernehmen und Benutzerattribute in die Zugriffsanfrage einfügen. Ist dies nicht der Fall, so können komplexere Lösungen die Benutzerauthentifizierung an den Authorization and Access Control Server delegieren, beispielsweise über den OAuth *Authorization Code* Flow. Hierdurch würde das System zu einer ganzheitlichen Identity und Access Management Lösung. Im Folgenden liegt der Fokus jedoch auf vertrauenswürdigen Client-Anwendungen.

- **Konfigurationsvorgaben:** Alle IoT-Geräte müssen registriert werden beim Authorization and Access Control Server. Die Konfiguration des Authorization and Access Control Server muss

dazu während der Installation des IoT-Geräts hinterlegt werden. Dies kann anwendungsabhängig automatisiert geschehen. Dabei teilt das IoT-Gerät dem Authorization and Access Control Server die anzuwendenden Policies und die Lebensdauer der auszustellenden Tokens mit. Diese Informationen kann das IoT-Gerät jederzeit ändern.

- **Privacy:** Das Transportprotokoll ist grundsätzlich frei wählbar. Es muss jedoch gewährleistet werden, dass die Übertragung sicher und privat ausgeführt wird.

6.3.2 Komponenten und Kommunikationsfluss

Das Teilen von Daten ist ein wesentliches Merkmal in IoT-Umgebungen. Da ein IoT-Gerät eine Menge von Ressourcen bereitstellen kann, ist es nicht effizient, bei jeder Anfrage den Resource Owner persönlich nach Zugriffsberechtigungen zu fragen. Deshalb kommt ein attributbasiertes Zugriffskontrollsystem zum Einsatz, welches es dem Benutzer ermöglicht, verschiedene Zugriffsrechte in Form von Policies zu hinterlegen. Zusätzlich wird der leicht modifizierte OAuth *Client Credential Grant*-Flow eingesetzt, um die Tokenübermittlung zwischen Client-Anwendung, Authorization und Access Control Server und IoT-Gerät durchzuführen. Die Modifikation des Flows besteht lediglich in der Anpassung der Anfrage- und Antwortformate, der eigentliche Ablauf bleibt unverändert. Damit wird eine dynamische Zugriffskontrolle möglich, welche auf etablierten Technologien aufbaut.

Abbildung 6.9 zeigt die Komponenten und den Kommunikationsfluss. Sobald ein IoT-Gerät eingerichtet ist und die Adresse des Authorization und Access Control Server bekannt ist, startet das IoT-

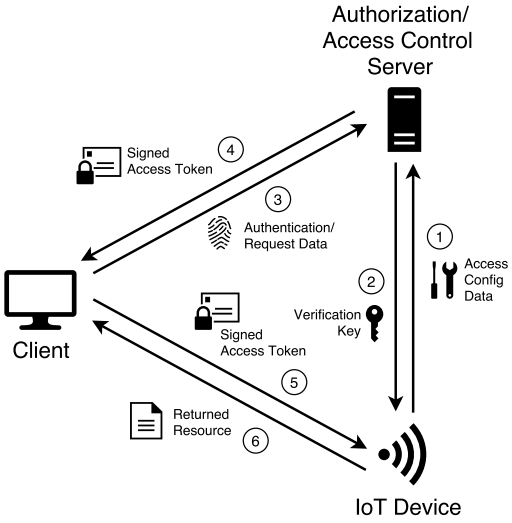

Abbildung 6.9: Architektur für verteilte Zugriffskontrolle (Quelle: [101])

Gerät den Konfigurationsprozess. Dafür sendet das IoT-Gerät Konfigurationsdaten zum Server (Nachricht 1). Diese Daten beinhalten die Ressourcenadressen, die Lebensdauer des auszustellenden Token und die Policies, welche beim Zugriff auf die Ressourcen des Geräts ausgewertet werden sollen. Sind die Konfigurationsdaten erfolgreich übermittelt worden, so antwortet der Authorization und Access Control Server mit dem Schlüssel zur Verifikation der von ihm ausgestellten

Tokens (Nachricht 2). Sobald das IoT-Gerät den Schlüssel empfangen hat, ist kein weiterer Nachrichtenaustausch notwendig. Das IoT-Gerät kann jedoch jederzeit die Konfigurationsdaten für den Zugriff beim Authorization und Access Control Server ändern, wenn dies gewünscht ist.

Fragt ein Client eine Ressource beim IoT-Gerät an, so erwartet das Gerät ein valides Token. Ist dieses Token nicht in der Anfrage enthalten, so verweist das IoT-Gerät auf den Authorization und Access Control Server. Der Zugriff auf die Ressource bleibt verwehrt.

Wendet sich ein Client an den Authorization und Access Control Server, so muss er sich zunächst authentifizieren. Der Client kann dann User-Attribute und Anfrage-Attribute an den Server übermitteln (Nachricht 3). Anfrage-Attribute sind etwa die URI der angefragten Ressource und die Zugriffsmethode. Weitere Attribute der angefragten Ressource, des anfragenden Subjekts oder der Umgebung (beispielsweise ein aktueller Zeitstempel) können durch die Attribute Provider des Authorization und Access Control Server bereitgestellt werden. Der Authorization und Access Control Server prüft danach die Policies für den Zugriff auf die Ressource und stellt gegebenenfalls ein Token aus, welches er signiert. Dieses Token wird an den Client zurückgeliefert (Nachricht 4). Der implementierte Prototyp antwortet im Erfolgsfall mit einem speziellen HTTP Status. Der Status 280 wird samt des Tokens zurückgeliefert. Wird der Zugriff verweigert, so antwortet der Server mit dem Status 480.

Abbildung 6.9 zeigt eine vereinfachte Version des Nachrichtenflusses. Bevor der Client zum ersten Mal die Ressource anfragt, ist der zuständige Authorization und Access Control Server ihm unbekannt, so dass er sich bei der ersten Anfrage an das IoT-Gerät wendet, welches ihn dann an den zuständigen Server verweist. Bei jeder wei-

teren Anfrage ist der Server bekannt und der Client kann sich wie in Abbildung 6.9 dargestellt direkt an den Server wenden.

Hat der Client ein valides Token, so kann er dieses in die Ressourcen-Anfrage einfügen und an das IoT-Gerät senden (Nachricht 5). Das IoT-Gerät prüft dann die Gültigkeit des Token. Dazu wird die Signatur geprüft und abgeglichen, ob die Parameter im Token der Anfrage entsprechen. Beispielsweise überprüft das IoT-Gerät, ob die im Token hinterlegte IP-Adresse der IP-Adresse des Client entspricht und ob das Token noch nicht abgelaufen ist. Ist die Anfrage erfolgreich, wird Zugriff auf die Ressource gewährt. Das IoT-GerÃd't antwortet erneut mit einem speziellen HTTP Status. Im Falle einer erfolgreichen Anfrage wird 290 als Status verwendet. Konnte das Token nicht erfolgreich validiert werden, so antwortet das IoT-Gerät mit dem Status 490 (Nachricht 6).

Übertragen auf das Beispielszenario bedeutet dies, das der Hausbewohner zum Öffnen der Garage ein Token vom Authorization und Access Control Server benötigt. Dazu authentifiziert sich die Client-Anwendung (beispielsweise eine Smart-Phone-Anwendung) beim Authorization und Access Control Server und schickt die Zugriffsanfrage inklusive Ressourcenadresse. Der Authorization und Access Control Server wertet die Anfrage aus und stellt ein Token aus, mit dem der Hausbewohner dann das Garagentor öffnen kann.

6.3.3 Tokenkonzept und Integrität

Token stellen die Verbindung zwischen Authorization und Access Control Server und dem IoT-Gerät dar und ersetzen die direkte Kommunikation zwischen diesen. Token sind nicht generell valide, sondern werden nur für bestimmte Anfragen ausgestellt. Die verwendeten To-

ken sind den OAuth-Token ähnlich. In Auflistung 6.17 ist ein Token dargestellt. Das Token beinhaltet einen Zeitstempel, um die Lebensdauer des Token abzubilden. Außerdem werden eine Menge von Ressourcen angegeben, die beim IoT-Gerät angefragt werden können. Dies ähnelt den Scopes der OAuth-Token. Die erlaubten Zugriffsmethoden sind ebenfalls hinterlegt. Schließlich ist noch der Client eingetragen, für welchen das Token ausgestellt wurde.

```
1  {
2    "token":{
3      "lifetime":"01.01.2017 12:01:00",
4      "resources":
5        ["https://www.example.org/garage/state"],
6      "device":"https://www.example.org/garage",
7      "methods":["GET"],
8      "client":"1.2.3.4",
9      "auth_server":
10       "https://www.example.org/authorization"
11   }, "signature": {
12     "value":"1a7b6062c6fb8fc2511036df6178 ..."
13   }
14 }
```

Auflistung 6.17: Beispieltoken

Mittels einer Signatur kann das IoT-Gerät verifizieren, dass das Token vom zugehörigen Authorization und Access Control Server ausgestellt wurde und während der Übertragung nicht manipuliert wurde. Dazu versieht der Authorization und Access Control Server das Token mit einem verschlüsselten Hash-Wert. Der Hash-Wert berechnet sich über die Werte des Tokens. Dieser Wert wird vom Authorization

und Access Control Server verschlüsselt und kann vom IoT-Gerät entschlüsselt werden. Dazu kann das IoT-Gerät den übermittelten Schlüssel aus der Konfigurationsnachricht verwenden. Diesen Wert kann das IoT-Gerät mit einer eigenen Berechnung des Hash abgleichen. Stimmen die Werte überein, so wurde das Token nicht manipuliert. Stimmen die Parameter des Tokens mit der Anfrage überein, so wurde das Token erfolgreich validiert und das IoT-Gerät gewährt Zugriff auf die Ressource.

6.3.4 Experimentelle Ergebnisse

Der Proof of Concept fokussiert sich auf die Zentralisierung von Zugriffsberechtigungen und die Integration von Tokens. Es handelt sich also nur um ein Element von vielen zur Sicherung von IT-Systemen. Der Prototyp erfordert die manuelle Eingabe der Konfiguration, also der Lebensdauer der auszustellenden Token sowie der Domain-Einträge und Policies für das Zugriffskontrollsystem. Das implementierte System basiert auf HTTP und verwendet propietäre Status Codes. Der Einsatz von HTTPS kann zusätzliche Sicherheit bieten, indem der Nachrichtenaustausch verschlüsselt wird. Die grundlegenden Konzepte eignen sich jedoch auch für andere Protokolle als HTTP und HTTPS.

Sicherheit und Privatsphäre Da in diesem Ansatz die Zugriffskontrolle auf Ebene der Anwendungsschicht durchgeführt wird, kann auf zusätzliche Sicherheitsmechanismen wie *Transport Layer Security* (TLS) und Authentifikationsprozeduren aufgebaut werden. Der Ansatz betrachtet die Integration von RestACL mit Tokens und IoT-Geräten

und sichert die Integrität von Token mittels Signaturen ab. Ende-zu-Ende-Verschlüsselung kann zusätzlich verwendet werden, um die Tokens nicht einsehbar zu transportieren. Hierdurch wird die Privatsphäre erhöht und Identitätsdiebstahl sowie Man-in-the-middle-Attacken ausgeschlossen. Um Ende-zu-Ende-Verschlüsselung zu ermöglichen, kann in der Konfigurationsphase ein symmetrischer Schlüssel ausgetauscht werden, welcher zum Verschlüsseln der Token verwendet wird. Dieser Schlüssel kann mit bekannten Verfahren wie etwa dem Diffie-Hellman-Schlüsselaustausch [35] ausgetauscht werden.

Messergebnisse Um den Ansatz zu testen, wurde ein *Raspberry Pi 3 Model B* als IoT-Gerät verwendet. Das Gerät ist ausgestattet mit einem *System on a Chip* (SoC) ARM Prozessor mit 1.2 GHz Taktfrequenz, 1 GB Arbeitsspeicher und einer kabellosen Netzwerkschnittstelle. Das 1.2 GHz Raspberry Pi 3 ist ein verhältnismäßig leistungsstarkes IoT-Gerät. Auf leistungsschwächeren Geräten wie etwa einem *ESP8266* [37] sind über die gemessenen Werte hinaus noch erheblich längere Laufzeiten zu erwarten. Der Authorization/Access Control Server und der Client wurden auf einem 3.2 GHz iMac (4 GB RAM) und einem 2 GHz MacBook (8 GB RAM) betrieben.

Auf dem Raspberry Pi wurden drei physische Komponenten bereitgestellt, welche auf drei Ressourcen abgebildet sind und mit insgesamt vier unterschiedlichen Zugriffsmöglichkeiten gesteuert werden können:

- Ein Temperatursensor, welcher die aktuelle Temperatur in einer Datei speichert, kann mit einer *GET*-Anfrage ausgelesen werden.

- Eine LED-Diode kann mit einer *PUT*-Anfrage ein- und ausgeschaltet werden.
- Eine LCD-Anzeige kann über eine *POST*-Anfrage einen Text anzeigen oder über eine *DELETE*-Anfrage den aktuellen Text löschen.

Um die Bedürfnisse von IoT-Geräten zu befriedigen, müssen Rechenaufwände und Speicherbedarf niedrig bleiben. Deshalb ist es wichtig zu vergleichen, inwiefern der Ansatz der Zentralisierung der Zugriffsberechtigungen sich gegenüber einer dezentralen Lösung abhebt. Bei einer dezentralen Lösung implementiert jedes Gerät einen eigenen Zugriffsmechanismus.

Tabelle 6.17 zeigt die benötigte Zeit für die Entscheidungsfindung und den benötigten Speicherbedarf im Vergleich zwischen zentralem und dezentralem Ansatz:

Ansatz	Laufzeit	Speicherbedarf (IoT-Gerät)
V_1: Dezentral	4.98 ms	variabel
V_2: Zentral (Asymmetrisch)	34.64 ms	1.7 MB
V_3: Zentral (Symmetrisch)	5.22 ms	1.7 MB

Tabelle 6.17: Aufwände verschiedener Implementierungen

- V_1 zeigt die Laufzeit, wenn das IoT-Gerät den RestACL- Mechanismus selbst implementiert. Dabei ist zu beachten, dass sich diese Laufzeiten erhöhen, wenn das IoT-Gerät nicht so leistungsstark ist. Der Speicherbedarf ist variabel und abhängig von der

Anzahl der verwendeten Policies. In Kapitel 4 wurden beispielsweise 6MB Speicherbedarf gemessen, um 10 Ressourcen mit jeweils 5 Policies zu schützen.

- V_2 zeigt die Performanz des zentralisierten Ansatz mit asymmetrischer Signatur (basierend auf RSA [120] und SHA-512 [33]) der Token. Dabei wird ein Token vom Authorization/Access Control Server signiert, der Client berechnet die Signatur selbst und vergleicht den errechneten Wert mit der Signatur des Servers.

- V_3 zeigt die Performanz des zentralisierten Ansatz mit symmetrischer Verschlüsselung. Dabei wird das gesamte Token mit einem in der Konfigurationsphase ausgetauschten Schlüssel verschlüsselt (Pre-Shared Key basierend auf AES [31]).

Während die Verarbeitungszeit auf dem leistungsstarken IoT-Gerät kürzer ist als beim Tokenansatz, wird ein höherer Speicherbedarf festgestellt. Der Speicherbedarf auf dem IoT-Gerät richtet sich nach der Anzahl an Ressourcen und Policies. Zudem herrscht Redundanz, so dass auf vielen Geräten ein höherer Speicherbedarf entsteht. Beim Tokenansatz hingegen kann mit einem festen Speicherbedarf auf dem IoT-Gerät gerechnet werden. Der Speicherbedarf richtet sich hier im Wesentlichen nach der eingesetzten Kryptographie-Bibliothek zur Überprüfung der Validität des Token.

Beachtenswert ist darüber hinaus, dass das RestACL-System deutlich schneller rechnet, wenn er auf einem konventionellen Gerät ausgeführt wird im Vergleich zu einem System-on-a-Chip. Gleiches kann für eine Authentifizierungsprozedur erwartet werden, welche in den dargestellten Zeiten nicht enthalten ist.

Neben der Skalierung ist die Gesamtlaufzeit einer Anfrage vom Client an das IoT-Gerät von Interesse. Durch Hinzufügen des Propagation Delay t_{Pd} und der Netzwerklaufzeit t_{Nw} berechnet sich die Gesamtlaufzeit t_{TotalV_x} bei dezentralen Ansatz V_1 und zentralem Ansatz $V_{2/3}$ als:

$$t_{TotalV_x} = t_{Pd} + t_{Nw} + t_{V_x}$$

Damit ergibt sich t_{TotalV_1} zu:

$$t_{TotalV_1} = t_{Pd_C} + t_{Pd_{IoT}} + 2 * t_{Nw_{C-IoT}} + t_{V_1}$$

mit t_{Pd_C} als Propagation Delay auf Clientseite, $t_{Pd_{IoT}}$ als Propagation Delay auf Geräteseite und $t_{Nw_{C-IoT}}$ als Netzwerklaufzeit zwischen Client und IoT-Gerät.

Die Gesamtlaufzeit bei $V_{2/3}$ hängt davon ab, ob der Client zuerst beim IoT-Gerät nach der Adresse des Authorization und Access Control Server fragen muss oder ob ihm diese bereits bekannt ist. t_{Pd_C}, $t_{Pd_{IoT}}$ und $t_{Nw_{C-IoT}}$ müssen im ersten Fall doppelt berücksichtigt werden.

$$t_{TotalV_{2/3}} = t_{C-AS} + t_{C-IoT} + t_{V_{2/3}}$$

mit

$$t_{C-AS} = t_{Pd_C} + t_{Pd_{AS}} + t_{Nw_{C-AS}}$$
$$t_{C-IoT} = t_{Pd_C} + t_{Pd_{IoT}} + t_{Nw_{C-IoT}}$$

Da das Propagation Delay und die Netzwerklaufzeit von verschiedensten Faktoren abhängen, ist eine absolute Testserie wenig aussagekräftig. Dennoch lässt sich feststellen, dass Netzwerklaufzeiten oft im Bereich von hunderten von Millisekunden liegen. Damit wird die Frage der Auslagerung und Zentralisierung von Zugriffsrechten eine Frage der Wartbarkeit und eine Frage des Speicherbedarfs. Der Speicherbedarf kann auf Geräten mit vielen Ressourcen und/oder Policies merklich reduziert werden. Insbesondere, wenn mehrere Geräte unter den gleichen Zugriffsbedingungen operieren, kann der Speicherbedarf insgesamt gesenkt werden, da Redundanz eliminiert wird. Dadurch steigt auch die Wartbarkeit der Zugriffsrechte.

6.3.5 Stand der Forschung

In [143] werden Probleme beim Auslagern und Teilen von Daten diskutiert. Der Ansatz nutzt Selective Encryption, um Daten zu verschlüsseln und so nicht öffentlich zugänglich zu machen. Geteilte Ressourcen werden verschlüsselt und aus dem Schlüssel können Token abgeleitet werden, mit denen die Ressource entschlüsselt werden kann. Der Ansatz fokussiert dabei auf Verschlüsselung und das Ableiten von Schüsseln. Zugriffsmethoden werden nicht berücksichtigt, sondern implizit ein lesender Zugriff angenommen. Zugriffsberechtigungen werden als Paare <*User, Resource*> definiert, wobei ein User auch eine Gruppe von Benutzern sein kann.

Capability Based Access Control (CapBAC) wird in [49] diskutiert. Dabei werden Token für den Zugriff auf Ressourcen eingesetzt. Die Token fungieren als Capability. Im Kontrast zu dem in diesem Abschnitt vorgestellten Ansatz werden initiale Zugriffsberechtigungen vom Besitzer der Ressource vergeben. Eine vertrauenswürdige, dritte

Partei, wie der Authorization and Access Control Server, ist nicht gegeben.

Wie bereits zuvor beschrieben, mangelt es IoT-Geräten an effizienten Entschlüsselungsalgorithmen mit denen schnellere Antwortzeiten erreicht werden könnten. In [153] wird beschrieben, dass der Entwurf von leichtgewichtigen Kryptographiesystemen noch immer eine große Herausforderung ist. Public Key-Infrastrukturen bieten Datenintegrität sowie Privatsphäre und sind geeignet für Authentifizierungsmechanismen, erfordern aber gleichzeitig einen erhöhten Kommunikationsaufwand.

Das *Constrained Application Protocol* (CoAP) wird aufgrund seiner Einfachheit und des geringen Overhead als geeignetes Protokoll für IoT-Umgebungen angesehen. HTTPS ist eng verknüpft mit TLS als Sicherheitsprotokoll und erfordert deshalb TCP als Transportprotokoll, welches wiederum langsamer verarbeitet wird als UDP. CoAP nutzt *Datagram Transport Layer Security* (DTLS) als Sicherheitsprotokoll, welches über UDP transportiert wird.

Neben der Vereinfachung von Protokollen, wird in [128] der Fokus auf die Hardware von IoT-Geräten gelegt. Auch wenn für Ende-zu-Ende-Verschlüsselung nur zu Beginn der Kommunikation eine Public Key-Infrastruktur benötigt wird, so wird dennoch während der gesamten Laufzeit einer Anwendung Speicher allokiert, so dass der eigentlichen Anwendung weniger Speicher zur Verfügung steht. Hardwareseitige Verschlüsselung auf neuen System-On-a-Chip-Geräten hingegen bieten symmetrische Verschlüsselung mit nur wenig zusätzlichen Kosten.

6.4 Zusammenfassung

In diesem Kapitel wurden die folgenden wissenschaftlichen Beiträge beschrieben:

- **Autorisierungsabhängiges Hypermedia:** es wurde ein Mechanismus vorgestellt, der die systematische Umsetzung von HATEOAS durch einen Zugriffskontrollmechanismus beschreibt.
- **Situationsabhängige Zugriffskontrolle:** situationsabhängige Zugriffskontrolle ist ein Zugriffskontrollmodell, welches mit ABAC umgesetzt werden kann. Hierzu werden Situationen als eigenständige Entitäten modelliert.
- **RestACL als RESTful Service:** es wurde beschrieben, wie RestACL als RESTful Service umgesetzt werden kann. Da RestACL zum Schutz vor unbefugten Zugriffen auf RESTful Services konzipiert wurde, ist das System in der Lage, sich selbst vor unbefugten Zugriffen zu schützen.
- **Integration mit OAuth für das IoT:** leistungsschwache IoT-Geräte können über das OAuth-Protokoll durch einen ABAC-Mechanismus geschützt werden.

In Abschnitt 6.1 wurde das HATEOAS-Paradigma mit Hilfe von autorisierungsabhängigen Repräsentationen von Ressourcen umgesetzt. Dazu wurde ein 2-phasiger Ablauf eingeführt, welcher Zugriffsberechtigungen für potentielle Zustandsübergänge berechnet. In der zusätzlichen Autorisierungsphase werden Informationen aus einem Hypermedia Navigation Model erfragt. Dank dieser Informationen ist

es gegebenenfalls möglich Ressourcenanfragen, welche vom anfragenden Subjekt nicht ausgeführt werden dürfen, zu filtern und die nicht erlaubten Hypermedia-Verweise bei der Antwort an das Subjekt wegzulassen. Dazu muss zwischen statischen und dynamischen Attributen unterschieden werden.

Der vorgestellte Beitrag verwendet *Link*-Header zur HATEOAS-Implementierung in einer Maschine-zu-Maschine-Umgebung. In Zukunft kann der Ansatz noch erweitert werden, indem anstelle der Verwendung des *Link*-Headers zusätzliche Renderer zum Einsatz kommen, welche die *Link*-Header mit der Darstellung der Ressource verknüpfen.

Abschnitt 6.2 präsentiert einen situationsabhängigen Zugriffskontrollmechanismus für schützenswerte Ressourcen im Internet of Things. Der Mechanismus basiert auf einem Situationserkennungssystem, welches Situation Templates gegen aktuelle Sensordaten auswertet. Sobald eine Situation auftritt, wird ein attributbasierter Zugriffskontrollmechanismus informiert, welcher Daten zur Situation speichert. Diese Daten können bei der Auswertung von Zugriffsanfragen berücksichtigt werden und so situationsabhängige Zugriffskontrolle ermöglichen.

In diesem Kapitel wird ein Architekturansatz beschrieben, der zeigt, wie das System zur Situationserkennung mit dem attributbasierten Zugriffskontrollmechanismus integriert werden kann. Aufgrund der generischen Eigenschaften des attributbasierten Ansatzes kann das System in verschiedenen Szenarien zum Einsatz kommen. Es wurde der Anwendungsfall im Umfeld von Industrie 4.0 ausführlich diskutiert und ein weiterer Anwendungsfall im Ambient Assisted Living vorgestellt. Weitere Anwendungsgebiete wie beispielsweise Smart Cities oder generell eventgetriebene Umgebungen sind denkbar.

In Abschnitt 6.3 wird ein auf Token basierender Ansatz vorgestellt, welcher für die Zugriffskontrolle bei leistungsschwachen IoT-Geräten

eingesetzt werden kann. Der Beitrag stellt eine Architektur vor und verwendet Nachrichtenflüsse, welche dem OAuth Client Credential Grant Flow entsprechen. Neben der Entlastung von leistungsschwachen IoT-Geräten bietet dieser Ansatz den Vorteil der Zentralisierung des Authorization Managements. Dies bedeutet eine Verbesserung der Wartbarkeit durch Reduktion von Redundanz. Gleichzeitig jedoch ist es bei einem tokenbasierten Ansatz und dynamischen Attributen notwendig, die Lebenszeiten des Token möglichst gering zu halten, da ansonsten unberechtigte Zugriffe die Folge sein können.

Kapitel 7

Zusammenfassung und Ausblick

In dieser Arbeit wurde die effiziente Gestaltung und Anwendung von attributbasierter Zugriffskontrolle für RESTful Services untersucht. Es wurde ein Zugriffskontrollmechanismus entwickelt, der sich die Eigenschaften von RESTful Services zunutze macht, um die Effizienz gegenüber generischen ABAC-Mechanismen zu steigern. Ressourcenorientierung ist das fundamentalste Merkmal von RESTful Services. Durch die Verwendung von Ressourcenadressen als eindeutige Schlüssel wird es möglich, ein indexbasiertes Evaluationsverfahren anzuwenden. Hierdurch kann sehr schnell die Teilmenge des Regelwerks identifiziert werden, die für eine Zugriffsanfrage ausgewertet werden muss.

Ein generischer Mechanismus wie XACML muss potentiell das gesamte Regelwerk sequentiell abarbeiten, um eine Zugriffsentscheidung treffen zu können. Der in dieser Arbeit entwickelte Mechanismus hingegen kann anhand der Ressourcenadresse die auszuwertenden Regeln finden, welche im Vergleich zum gesamten Regelwerk nur eine sehr kleine Menge darstellen. Mit diesem Verfahren werden also die Aufwände zum Auswerten einer Zugriffsanfrage von der Komplexität des gesamten Regelwerks entkoppelt. Hierdurch wird es möglich auch für sehr große Mengen von Ressourcen sehr feingranulare Zugriffsbedingungen zu formulieren, ohne dabei die Performanz des RESTful Service signifikant zu verschlechtern.

7.1 Antworten auf die Forschungsfragen

Zu Beginn dieser Arbeit wurden mehrere Forschungsfragen formuliert. An dieser Stelle sollen diese Forschungsfragen nun beantwortet werden:

Lässt sich eine flexible und feingranulare Autorisierung von Zugriffsanfragen für eine große Menge von Ressourcen in RESTful Services umsetzen?

- Ja, Autorisierung für individuelle Ressourcen lässt sich flexibel und feingranular gestalten, wenn attributbasierte Zugriffskontrolle eingesetzt wird und gleichzeitig ein Schlüsselattribut existiert. Dies ist für RESTful Services grundsätzlich der Fall, da Ressourcenorientierung eingesetzt wird und die Ressourcenadresse als eindeutiger Schlüssel stets Teil der Kommunikation zwischen Client und Server ist.

Wie lassen sich ein allgemein gültiges Modell und eine Sprache konzipieren, so dass keine von der Anwendung abhängigen Elemente

abgebildet werden müssen? Kann Externalisierung der Zugriffskontrolle für RESTful Services umgesetzt werden?

- Da ABAC ein generisches Modell ist, werden keine anwendungsabhängigen Elemente modelliert und ausgewertet. Lediglich Ressourcenadresse und Zugriffsmethode sind notwendig. Diese sind bei jedem RESTful Service gegeben, unabhängig von der Anwendung, die durch die Services implementiert wird. Externalisierung kann durch eine strikte Trennung von Zugriffskontrolllogik und Anwendungslogik erreicht werden. Durch die Flexibilität des Mechanismus ist es nicht notwendig, die Zugriffskontrolllogik als Teil der Anwendungslogik zu implementieren.

Welche Architekturen und Algorithmen ermöglichen ein effizientes Abbilden, Bearbeiten und Auswerten dieses Modells/dieser Sprache?

- Der Zugriffsmechanismus ist aufgeteilt in vier grundlegende Bausteine: Evaluation Engine, Domain, Policy Repository und Attribute Provider. Dabei sind drei dieser Bausteine reine Datenspeicher. Attribute werden bei Attribute Providern gespeichert oder sind bereits Teil der Zugriffsanfrage. Die eigentliche Zugriffskontrolllogik wird in separierten und wiederverwendbaren Policies in einem Policy Repository gehalten. Die Daten der Domain beinhalten die Information, für welche Ressourcen und Zugriffsmethoden welche Policies ausgewertet werden müssen. In allen drei Datenspeichern können indexbasierte Verfahren eingesetzt werden, um die Effizienz zu maximieren. Hash-Tables sind eine ideale Speicherstruktur, um schnell von einem eindeutigen Schlüssel einer Ressource, Policy oder Entität an die benötigten Daten zu gelangen.

Welche Merkmale bekannter Autorisierungsmechanismen und -standards lassen sich abbilden? Sind positive und negative Zugriffsberechtigungen, Generalisierungen und Ausnahmen, das Überschreiben

von Zugriffsberechtigungen, Gruppierungen von Berechtigungen oder auch Vererbung möglich?

- ABAC ist in der Lage, andere Zugriffskontrollmodelle abzubilden. So können etwa DAC, MAC und RBAC abgebildet werden [80]. XACML ist ein Standard, der ABAC auf generische Weise ermöglicht. RestACL ist an XACML angelehnt und spezialisiert auf RESTful Services. In dieser Arbeit wurde gezeigt, dass die Ausdrucksstärke der beiden Mechanismen identisch ist. Positive und negative Zugriffsentscheidungen lassen sich in beiden Mechanismen abbilden. Durch die Verwendung von ausgelagerten Policies und die Möglichkeit zur Angabe einer Liste von Policies für die Entscheidungsfindung wird es möglich, generelle Zugriffsberechtigungen zu formulieren, welche von verschiedenen Policies überschrieben werden können. So ist es etwa möglich, den Zugriff auf Ressourcen grundsätzlich zu verbieten. Dazu kann eine Policy genutzt werden, die keinerlei Attributbedingungen formuliert, den Zugriff verbietet und eine niedrige Priorität aufweist. Andere Policies mit einer höheren Priorität und spezialisierten Attributbedingungen können diese Policy überschreiben und den Zugriff gewähren. Da Policies leicht wiederverwendbar sind, können für kleine und große Mengen von Ressourcen die Zugriffsberechtigungen in einer oder mehreren Policies gruppiert werden.

Wie wird bei der Auswertung von Anfragen, insbesondere bei lesenden Zugriffen, verfahren? Kann eine Entscheidung unabhängig vom Inhalt der angefragten Ressourcen getätigt werden? Ist es sinnvoll und vertretbar, die Inhalte zur Entscheidung heranzuziehen?

- Es ist leicht vorstellbar, dass der Zugriff auf eine Ressource, wie etwa ein Dokument, davon abhängt, welche Informationen im Dokument gespeichert sind. Durch die Verwendung von attributbasierter Zugriffskontrolle können auch die Inhalte einer Ressource ausgewer-

tet werden. Diese Inhalte müssen dazu als Attribut der Ressource formuliert werden, so dass sie bei der Auswertung von Zugriffsberechtigungen zur Verfügung stehen.

7.2 Wissenschaftliche Beiträge der Arbeit

Die Arbeit liefert verschiedene wissenschaftliche Beiträge zu XACML. In Kapitel 3 werden allgemeine Optimierungsverfahren für die Gestaltung von XACML-Regelwerken beschrieben. Die Strukturierung von XACML-Regelwerken ist mitentscheidend für die Effizienz eines Zugriffskontrollsystems, da sie einen negativen Einfluss auf die Verarbeitungsgeschwindigkeit von Zugriffsanfragen haben kann. Wird das Regelwerk nicht von Beginn an klar strukturiert, so sind Transformationsprozesse notwendig, welche die Struktur des Regelwerks neu organisieren, um die Verarbeitungsgeschwindigkeit zu verbessern. Diese Transformationsprozesse sind jedoch nur dann praktikabel, wenn das Regelwerk statisch ist. Dynamische Änderungen von Zugriffsberechtigungen können mit solchen Prozessen nicht mehr effizient gehandhabt werden, da sie zu häufig ausgeführt werden müssen.

Ein weiterer wissenschaftlicher Beitrag zur Analyse von XACML ist die Bestimmung von Prioritäten für XACML-Rules. Dieser Beitrag ist im Rahmen des Vergleichs von XACML und RestACL in Kapitel 5 entstanden. Durch die Verwendung verschiedener Combining Algorithm und Effects und aufgrund des kompositionellen Aspekts von XACML-Regelwerken, kann ein Regelwerk sehr schnell sehr komplex werden. Dies führt dazu, dass die Fehleranfälligkeit steigt und erhöhte Wartungsaufwände notwendig sind, um fachlich korrekte Zugriffberechtigungen abzubilden. Durch die Berechnung von Prioritäten ist ein

systematischer Ansatz gegeben, der aufzeigt, an welchen Positionen eine Rule innerhalb der Komposition Einfluss nimmt.

Der umfangreichste Beitrag der Arbeit ist die Untersuchung von indexbasierten Mechanismen für attributbasierte Zugriffskontrolle. Diese Mechanismen können dann eingesetzt werden, wenn ein eindeutiger Schlüssel, also ein Schlüsselattribut, existiert. Die Strukturierung eines Regelwerks entlang des Schlüsselattributs ermöglicht den Einsatz effizienter Algorithmen, um den Teil des Regelwerks zu identifizieren, welcher die Zugriffsentscheidung bestimmt. In einem solchen Mechanismus ist der kompositionelle Aspekt von XACML nicht mehr gegeben, wodurch eine Entkopplung von der Verarbeitungsgeschwindigkeit und der Menge der Zugriffsberechtigungen stattfindet.

In Kapitel 4 wird ein indexbasierter Mechanismus für RESTful Services beschrieben. RESTful Services basieren auf Ressourcenorientierung und besitzen deshalb die Ressourcenadresse, die URI, als Schlüsselattribut. RestACL teilt das Regelwerk in Domain und Policy Repository. In der Domain wird dieses Schlüsselattribut ausgewertet. Darüber hinaus weisen RESTful Services aufgrund der einheitlichen Schnittstelle noch ein zweites Attribut auf, welches bei jeder Interaktion mit einer Ressource präsent ist: die Zugriffsmethode. Auch dieses Attribut wird in der Domain ausgewertet, um die Menge an auszuwertenden Zugriffsbedingungen weiter zu reduzieren. Allerdings ist die Zugriffsmethode kein Schlüsselattribut, da die Schnittstelle für alle Ressourcen gleich ist und somit kein Unterscheidungsmerkmal existiert. Der attributbasierte Charakter des Mechanismus wird durch leicht wiederverwendbare Policies ausgedrückt. Hier können beliebige Attributbedingungen formuliert werden.

Die Einsatzmöglichkeiten für einen solchen Mechanismus sind vielfältig. In Kapitel 6 werden Anwendungsmöglichkeiten aus drei

verschiedenen Bereichen vorgestellt. Autorisierungsabhängiges Hypermedia ist ein Ansatz, um HATEOAS systematisch mit Hilfe eines Hypermedia Navigation Models und dem Zugriffsmechanismus umzusetzen. Diese Einsatzmöglichkeit ist ein wissenschaftlicher Beitrag zur Untersuchung von REST.

Situationsabhängige Zugriffskontrolle hingegen zeigt die Mächtigkeit des attributbasierten Zugriffskontrollmodells. Es wird aufgezeigt, wie ein situationsabhängiges Zugriffskontrollmodell durch ein attributbasiertes Verfahren abgebildet werden kann. Hierfür ist es lediglich notwendig, die Situation als eigene Kategorie zu modellieren. Durch die Integration mit einem Situationserkennungssystem kann so ein situationsabhängiges Zugriffskontrollsystem entstehen.

Eine dritte Anwendungsmöglichkeit ist der Einsatz im Internet Of Things. IoT-Geräte sind unter Umständen leichtgewichtig und leistungsschwach. Auf diesen Geräten ist die Implementierung eines feingranularen Zugriffskontrollsystems deshalb nicht praktikabel. Durch den Einsatz von OAuth kann jedoch auch auf einem leichtgewichtigen Gerät feingranulare Zugriffskontrolle stattfinden, wenn die Berechtigungsverwaltung an eine leistungsstarke Netzwerkkomponente ausgelagert wird.

7.3 Ausblick

Das grundlegende Prinzip des indexbasierten Mechanismus lässt sich auch in anderen Umgebungen als RESTful Services anwenden. Überall dort, wo eindeutige Schlüssel in Zugriffsanfragen existieren, kann der Mechanismus eingesetzt werden. Attributbasierte Zugriffskontrolle lässt sich demnach deutlich effizienter gestalten, wenn ein Schlüsselat-

tribut existiert. In der Zukunft kann ein allgemein gültiges Verfahren formuliert werden, in dem Schlüsselattribute genutzt werden, um attributbasierte Zugriffskontrolle effizient zu gestalten.

Ein weiteres Themengebiet für zukünftige Forschungsarbeiten ist die Untersuchung von lokalen Prioritäten anstelle von globalen Prioritäten. Lokale Prioritäten vereinfachen die Verwaltung von Prioritäten und bieten die Möglichkeit, Policies mit unterschiedlicher Gewichtung wiederzuverwenden. Hier gilt es zu untersuchen, wie sich lokale Prioritäten im Zusammenspiel von Access-Elementen, ParameterizedAccess-Elementen und Templates verhalten.

Ein drittes Themengebiet, in dem weitere Forschung sinnvoll erscheint, ist der Einsatz von attributbasierter Zugriffskontrolle im IoT-Umfeld. Im Rahmen dieser Arbeit wurde ein tokenbasierter Ansatz untersucht, um attributbasierte Zugriffskontrolle auf IoT-Geräten zu ermöglichen. Dies ist insbesondere bei leistungsschwachen Geräten ein sinnvoller Ansatz. Es existieren jedoch auch viele leistungsstarke Geräte, welche Autorisierungsmechanismen selbst implementieren können, wodurch hybride Umgebungen enstehen, in denen ein Teil der Geräte eigenverantwortlich agieren kann, während der andere Teil von Geräten auf externe Komponenten angewiesen ist. In solch einer Umgebung stellt sich die Frage nach der Verteilung, Synchronisation und Konsistenz von Zugriffsberechtigungen.

Veröffentlichungen

[61] Marc Hüffmeyer, Florian Haupt, Frank Leymann, and Ulf Schreier. “Authorization-aware HATEOAS”. In: *CLOSER ’18 - Proceedings of the 8th International Conference on Cloud Computing and Services Science*. 2018.

[62] Marc Hüffmeyer, Pascal Hirmer, Bernhard Mitschang, Ulf Schreier, and Matthias Wieland. “SitAC – A System for Situation-Aware Access Control - Controlling Access to Sensor Data”. In: *ICISSP ’17 - Proceedings of the 3rd International Conference on Information Systems Security and Privacy*. 2017.

[63] Marc Hüffmeyer, Pascal Hirmer, Bernhard Mitschang, Ulf Schreier, and Matthias Wieland. “Situation-Aware Access Control for Industrie 4.0”. In: *Information Systems Security and Privacy*. Springer, 2018.

[64] Marc Hüffmeyer and Ulf Schreier. “An Attribute Based Access Control Model for RESTful Services”. In: *SummerSOC ’15 - Proceedings of the 9th Symposium on Service-Oriented Computing*. 2015.

[65] Marc Hüffmeyer and Ulf Schreier. “Analysis of an Access Control System for RESTful Services”. In: *ICWE ’16 - Proceedings of the 16th International Conference on Web Engineering*. 2016.

[66] Marc Hüffmeyer and Ulf Schreier. “Designing Efficient XACML Policies for RESTful Services”. In: *Web Services and Formal Methods*. Springer, 2016.

[67] Marc Hüffmeyer and Ulf Schreier. “Efficient Attribute Based Access Control for RESTful Services”. In: *ZEUS ’15 - Proceedings of the 7th Central European Workshop on Services and their Composition*. 2015.

[68] Marc Hüffmeyer and Ulf Schreier. “Formal Comparison of an Attribute Based Access Control Language for RESTful Services with XACML”. In: *SACMAT ’16 - Proceedings of the 21st Symposium on Access Control Models and Technologies*. 2016.

[69] Marc Hüffmeyer and Ulf Schreier. “RestACL - An Attribute Based Access Control Language for RESTful Services”. In: *ABAC ’16 - Proceedings of the 1st Workshop on Attribute Based Access Control*. 2016.

[101] Philipp Montesano, Marc Hüffmeyer, and Ulf Schreier. “Outsourcing Access Control for a Dynamic Access Configuration of IoT Services”. In: *IoTBDS ’17 - Proceedings of the 2nd International Conference on Internet of Things, Big Data and Security*. 2017.

[146] Kevin Wallis, Marc Hüffmeyer, Ayhan Soner Koca, and Christoph Reich. “Access Rules Enhanced by Dynamic IIoT Context”. In: *IoTBDS ’18 - Proceedings of the 3rd International Conference on Internet of Things, Big Data and Security*. 2018.

Literaturverzeichnis

[1] Internet Engineering Task Force (IETF). “RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax”. In: (2012).

[2] Internet Engineering Task Force (IETF). “RFC 6570 - URI Template”. In: (2012).

[3] Internet Engineering Task Force (IETF). “RFC 6749 - The OAuth 2.0 authorization Framework”. In: (2012).

[4] Paul Adamczyk, Patrick H. Smith, Ralph E. Johnson, and Munawar Hafiz. “REST and Web Services: In Theory and in Practice”. In: *REST: From Research to Practice*. Ed. by Erik Wilde and Cesare Pautasso. Springer, 2011.

[5] Organization for the Advancement of Structured Information Standards (OASIS). “Extensible Access Control Markup Language (XACML) Version 3.0”. In: (2013).

[6] Organization for the Advancement of Structured Information Standards (OASIS). “JSON Profile of XACML 3.0 Version 1.0”. In: (2017).

[7] Organization for the Advancement of Structured Information Standards (OASIS). “REST Profile of XACML v3.0 Version 1.0”. In: (2014).

[8] Gail-Joon Ahn and Ravi Sandhu. “Role-Based authorization Constraints Specification”. In: *ACM Transactions on Information and System Security*. Vol. 3. 2000.

[9] *Abbreviated Language For authorization*. `https://www.oasis-open.org/committees/download.php/55228/alfa-for-xacml-v1.0-wd01.doc/`. Zuletzt abgerufen: 2018-06-30.

[10] Mike Amundsen. “From APIs to Affordances: A New Paradigm for Web Services”. In: *WS-REST ’12 - Proceedings of the Third International Workshop on RESTful Design*. 2012.

[11] Mike Amundsen. *RESTful Web Clients - Enabling Reuse Through Hypermedia*. O’Reilly Media, 2017, pp. 153–172.

[12] *You are not obliged to follow my advice: Obligations and Advice in XACML*. `https://www.axiomatics.com/blog/you-are-not-obliged-to-follow-my-advice-obligations-and-advice-in-xacml/`. Zuletzt abgerufen: 2018-06-30.

[13] *Protect Critical Data and Enable Collaboration with ABAC*. `https://www.axiomatics.com/resources/why-abac/`. Zuletzt abgerufen: 2018-06-30.

[14] Neil Ayeb, Francesco Di Cerbo, and Slim Trabelsi. “Enhancing Access Control Trees for Cloud Computing”. In: *TELERISE ’16 - Proceedings of the 2nd International Workshop on TEchnical and LEgal aspects of data pRIvacy and SEcurity*. 2016.

[15] Dhouha Ayed, Marie-Noelle Lepareux, and Cyrille Martins. “Analysis of XACML policies with ASP”. In: *NTMS ’15 - Proceedings of the 7th International Conference on New Technologies, Mobility and Security*. 2015.

[16] *WSO2 XACML Implementation - Balana*. `https://github.com/wso2/balana`. Zuletzt abgerufen: 2018-06-30.

[17] Dizza Beimel and Mor Peleg. “Using OWL and SWRL to represent and reason with situation-based access control policies”. In: *Data & Knowledge Engineering*. Vol. 70. Elsevier, 2011.

[18] Messaoud Benantar. *Access Control Systems: Security, Identity Management and Trust Models*. Springer Science & Business Media, 2006, pp. 40–72.

[19] Elisa Bertino. “Policies, Access Control, and Formal Methods”. In: *Handbook on Securing Cyber-Physical Critical Infrastructure*. Vol. 1. 2012.

[20] Rafae Bhatti, Elisa Bertino, and Arif Ghafoor. “A Trust-Based Context-Aware Access Control Model for Web-Services”. In: *Distributed and Parallel Databases*. Vol. 18. Springer, 2005.

[21] Carsten Bormann, Angelo P. Castellani, and Zach Shelby. “CoAP: An Application Protocol for Billions of Tiny Internet Nodes”. In: *IEEE Internet Computing*. Vol. 16. 2012.

[22] Eric Brachmann, Gero Dittmann, and Klaus-Dieter Schubert. “Simplified Authentication and authorization for RESTful Services in Trusted Environments”. In: *ESOCC’12 - Proceedings of the 1st European Conference on Service-Oriented and Cloud Computing*. 2012.

[23] Peter Brusilovsky. “Methods and techniques of adaptive hypermedia”. In: *Adaptive Hypertext and Hypermedia*. Springer, 1998.

[24] Bill Burke. *RESTful Java with Jax-RS*. O’Reilly Media, 2009, pp. 27–69.

[25] Hakki Cankaya. “Access Control Lists”. In: *Encyclopedia of Cryptography and Security*. Ed. by Henk van Tilborg and Sushil Jajodia. Springer, 2011.

[26] Shiu-Kai Chin and Susan Beth Older. *Access Control, Security, and Trust: A Logical Approach*. CRC Press, 2011, pp. 107–174.

[27] John Cocke. “Global Common Subexpression Elimination”. In: *ACM SIGPLAN Notices - Proceedings of a symposium on Compiler optimization*. 1970.

[28] Thomas Cormen, Charles Leiserson, Robert Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, 2001, pp. 253–285.

[29] Jason Crampton and Charles Morisset. “PTaCL: A Language for Attribute-Based Access Control in Open Systems”. In: *POST ’12 - Proceedings of the First International Conference on Principles of Security and Trust*. 2012.

[30] *Microsoft Dynamic Access Control*. `https://docs.microsoft.com/en-us/windows-server/identity/solution-guides/dynamic-access-control-overview`. Zuletzt abgerufen: 2018-06-30.

[31] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES - the advanced encryption standard*. Springer, 2013.

[32] Ni Dan, Shi Hua-Ji, Chen Yuan, and Guo Jia-Hu. “Attribute Based Access Control (ABAC)-based cross-domain access control in service-oriented architecture (SOA)”. In: *CSSS ’12 - Proceedings of the 2012 International Conference on Computer Science and Service System*. 2012.

[33] Quynh Dang. “Recommendation for Applications Using Approved Hash Algorithms”. In: National Institute of Standards and Technology, Information Technology Laboratory, Computer Security Division, 2012.

[34] Cornelia Davis. “What if the Web Were not RESTful?” In: *WS-REST ’12 - Proceedings of the Third International Workshop on RESTful Design*. 2012.

[35] Whitfield Diffie and Martin Hellman. “New Directions in Cryptography”. In: *IEEE Transactions on Information Theory*. Vol. 22. IEEE Computer Society, 1976.

[36] Claudia Eckert. *IT-Sicherheit, Konzepte - Verfahren - Protokolle*. De Gruyter Oldenbourg, 2014, pp. 465–540, 635–712.

[37] *ESP8266*. `https://espressif.com/products/hardware/esp8266ex`. Zuletzt abgerufen: 2018-06-30.

[38] *Facebook API*. `https://developers.facebook.com/docs/facebook-login/permissions/`. Zuletzt abgerufen: 2018-06-30.

[39] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and Raymond Strong. “Extendible Hashing - A Fast Access Method for Dynamic Files”. In: *ACM Transactions on Database Systems*. Vol. 4. 1979.

[40] David Ferraiolo, Larry Feldman, and Greg Witte. “Exploring the Next Generation of Access Control Methodologies”. In: *National Institute of Standards and Technology, Information Technology Laboratory, Computer Security Division*. 2016.

[41] David Ferraiolo and Richard Kuhn. “Role-Based Access Controls ”. In: *National Institute of Standards and Technology, Technology Administration*. 1992.

[42] David Ferraiolo, Richard Kuhn, and Vincent Hu. “Attribute-Based Access Control”. In: *Computer*. Vol. 48. IEEE Computer Society, 2015.

[43] David Ferraiolo, Ravi Sandhu, Serban Gavrila, Richard Kuhn, and Ramaswamy Chandramouli. “Proposed NIST standard for role-based access control”. In: *TISSEC ’01 - ACM Transactions on Information and System Security*. 2001.

[44] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine, 2000.

[45] *Martin Fowler - Steps toward the glory of REST*. `https://martinfowler.com/articles/richardsonMaturityModel.html`. Zuletzt abgerufen: 2018-06-30. 2010.

[46] Ana Cristina Franco da Silva, Pascal Hirmer, Matthias Wieland, and Bernhard Mitschang. “SitRS XT – Towards Near Real Time Situation Recognition”. In: *Proceedings of the 31st Brazilian Symposium of Databases (SBBD)*. 2016.

[47] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994, pp. 127–135, 163–175.

[48] Sebastian Graf, Vyacheslav Zholudev, Lukas Lewandowski, and Marcel Waldvogel. "Hecate, Managing authorization with RESTful XML". In: *WS-REST '11 - Proceedings of the Second International Workshop on RESTful Design*. 2011.

[49] Sergio Gusmeroli, Salvatore Piccione, and Denise Rotondi. "IoT Access Control Issues: a Capability Based Approach". In: *Proceedings of the Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. 2012.

[50] Sergio Gusmeroli, Salvatore Piccione, and Domenico Rotondi. "A capability-based security approach to manage access control in the Internet of Things". In: *Mathematical and Computer Modelling*. Vol. 58. 2013.

[51] Florian Haupt, Dimka Karastoyanova, Frank Leymann, and Benjamin Schroth. "A Model-Driven Approach for REST Compliant Services". In: *ICWS '14 - 2014 IEEE International Conference on Web Services*. IEEE, 2014.

[52] Florian Haupt, Frank Leymann, and Cesare Pautasso. "A conversation based approach for modeling REST APIs". In: *WICSA '15 - 12th Working IEEE / IFIP Conference on Software Architecture*. IEEE, 2015.

[53] Florian Haupt, Frank Leymann, Anton Scherer, and Karolina Vukojevic-Haupt. “A Framework for the Structural Analysis of REST APIs”. In: *ICSA ’17 - Proceedings of the IEEE International Conference on Software Architecture*. IEEE, 2017.

[54] Pascal Hirmer, Matthias Wieland, Uwe Breitenbücher, and Bernhard Mitschang. “Automated Sensor Registration, Binding and Sensor Data Provisioning”. In: *Proceedings of the CAiSE’16 Forum, at the 28th International Conference on Advanced Information Systems Engineering (CAiSE 2016)*. Vol. 1612. CEUR Workshop Proceedings. 2016.

[55] Pascal Hirmer, Matthias Wieland, Uwe Breitenbücher, and Bernhard Mitschang. “Dynamic Ontology-based Sensor Binding”. In: *Advances in Databases and Information Systems. 20th East European Conference, ADBIS 2016, Prague, Czech Republic, August 28-31, 2016, Proceedings*. Vol. 9809. Information Systems and Applications, incl. Internet/Web, and HCI. Springer International Publishing, 2016.

[56] Pascal Hirmer, Matthias Wieland, Holger Schwarz, Bernhard Mitschang, Uwe Breitenbücher, and Frank Leymann. “SitRS - A Situation Recognition Service based on Modeling and Executing Situation Templates”. In: *Proceedings of the 9th Symposium and Summer School On Service-Oriented Computing*. Ed. by Johanna Barzen, Rania Khalaf, Frank Leymann, and Bernhard Mitschang. Vol. RC25564. Technical Paper. IBM Research Report, 2015.

[57] Kristina Höök, Jussi Karlgren, Annika Wærn, Nils Dahlbäck, Carl Gustaf Jansson, Klas Karlgren, and Benoit Lemaire. “A

Glass Box Approach to Adaptive Hypermedia". In: *Adaptive Hypertext and Hypermedia*. Springer, 1995.

[58] Eva Hoos, Pascal Hirmer, and Bernhard Mitschang. "Improving Problem Resolving on the Shop Floor by Context-Aware Decision Information Packages". In: *Proceedings of the CAiSE 2017 Forum*. CEUR Workshop Proceedings, 2017.

[59] Vincent Hu, David Ferraiolo, Richard Kuhn, and Ramaswamy Chandramouli. *Attribute Based Access Control*. Artech House, 2017.

[60] Vincent Hu, David Ferraiolo, Richard Kuhn, Adam Schnitzer an Kenneth Sandlin, Robert Miller, and Karen Scarfone. "Guide to Attribute Based Access Control (ABAC) Definition and Considerations". In: *National Institute of Standards and Technology, Information Technology Laboratory, Computer Security Division*. 2016.

[61] Marc Hüffmeyer, Florian Haupt, Frank Leymann, and Ulf Schreier. "Authorization-aware HATEOAS". In: *CLOSER '18 - Proceedings of the 8th International Conference on Cloud Computing and Services Science*. 2018.

[62] Marc Hüffmeyer, Pascal Hirmer, Bernhard Mitschang, Ulf Schreier, and Matthias Wieland. "SitAC – A System for Situation-Aware Access Control - Controlling Access to Sensor Data". In: *ICISSP '17 - Proceedings of the 3rd International Conference on Information Systems Security and Privacy*. 2017.

[63] Marc Hüffmeyer, Pascal Hirmer, Bernhard Mitschang, Ulf Schreier, and Matthias Wieland. "Situation-Aware Access Con-

trol for Industrie 4.0". In: *Information Systems Security and Privacy*. Springer, 2018.

[64] Marc Hüffmeyer and Ulf Schreier. "An Attribute Based Access Control Model for RESTful Services". In: *SummerSOC '15 - Proceedings of the 9th Symposium on Service-Oriented Computing*. 2015.

[65] Marc Hüffmeyer and Ulf Schreier. "Analysis of an Access Control System for RESTful Services". In: *ICWE '16 - Proceedings of the 16th International Conference on Web Engineering*. 2016.

[66] Marc Hüffmeyer and Ulf Schreier. "Designing Efficient XACML Policies for RESTful Services". In: *Web Services and Formal Methods*. Springer, 2016.

[67] Marc Hüffmeyer and Ulf Schreier. "Efficient Attribute Based Access Control for RESTful Services". In: *ZEUS '15 - Proceedings of the 7th Central European Workshop on Services and their Composition*. 2015.

[68] Marc Hüffmeyer and Ulf Schreier. "Formal Comparison of an Attribute Based Access Control Language for RESTful Services with XACML". In: *SACMAT '16 - Proceedings of the 21st Symposium on Access Control Models and Technologies*. 2016.

[69] Marc Hüffmeyer and Ulf Schreier. "RestACL - An Attribute Based Access Control Language for RESTful Services". In: *ABAC '16 - Proceedings of the 1st Workshop on Attribute Based Access Control*. 2016.

[70] Kantara Initiative. *User-Managed Access (UMA) Profile of OAuth 2.0*. https://docs.kantarainitiative.org/uma/rec-uma-core.html/. Zuletzt abgerufen: 2018-06-30.

[71] Internet Engineering Task Force. "RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1". In: (1999).

[72] Internet Engineering Task Force. "RFC 2617 - HTTP Authentication: Basic and Digest Access Authentication". In: (1999).

[73] Internet Engineering Task Force. "RFC 2865 - Remote Authentication Dial In User Service (RADIUS)". In: (2000).

[74] Internet Engineering Task Force. "RFC 4120 - The Kerberos Network Authentication Service (V5)". In: (2005).

[75] Internet Engineering Task Force. "RFC 4511 - Lightweight Directory Access Protocol (LDAP): The Protocol". In: (2006).

[76] Internet Engineering Task Force. "RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2". In: (2008).

[77] Internet Engineering Task Force. "RFC 6350 - vCard Format Specification". In: (2011).

[78] Internet Engineering Task Force. "RFC 7252 - The Constrained Application Protocol (CoAP)". In: (2014).

[79] *JAX-RS*. https://jax-rs-spec.java.net/. Zuletzt abgerufen: 2018-06-30.

[80] Xin Jin, Ram Krishnan, and Ravi Sandhu. "A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC". In: *DBSec '12 - Proceedings of the 26th Annual Conference on Data and Applications Security and Privacy*. 2012.

[81] Qi Jing, Athanasios V. Vasilakos, Jiafu Wan, Jingwei Lu, and Dechao Qiu. “Security of the Internet of Things: Perspectives and Challenges”. In: *Wireless Networks*. 2014.

[82] *Apache JMeter*. `http://jmeter.apache.org`. Zuletzt abgerufen: 2018-06-30.

[83] Carole S. Jordan. “Guide to Understanding Discretionary Access Control in Trusted Systems”. In: *ACM Computing Surveys*. 1987.

[84] Markus Jung, Georg Kienesberger, Wolfgang Granzer, Martin Unger, and Wolfgang Kastner. “Privacy enabled Web service access control using SAML and XACML for home automation gateways”. In: *ICITST ’11 - Proceedings of the 2011 International Conference for Internet Technology and Secured Transactions*. 2011.

[85] Craig Kaplan, Justine Fenwick, and James Chen. “Adaptive hypertext navigation based on user goals and context”. In: *User Modeling and User-Adapted Interaction*. Springer, 1993.

[86] Laura Kassner, Christoph Gröger, Jan Königsberger, Eva Hoos, Cornelia Kiefer, Christian Weber, Stefan Silcher, and Bernhard Mitschang. “The Stuttgart IT Architecture for Manufacturing”. In: vol. 291. Springer International Publishing, 2017.

[87] Markku Laitkorpi, Petri Selonen, and Tarja Systa. “Towards a model-driven process for designing restful web services”. In: *ICWS ’09 - Proceedings of the 2009 IEEE International Conference on Web Services*. 2009.

[88] Bo Lang, Ian Foster, Frank Siebenlist, Rachana Ananthakrishnan, and Tim Freeman. “A Multipolicy authorization Framework for Grid Security”. In: *NCA ’16 - Proceedings of the 5th IEEE International Symposium on Network Computing and Applications*. 2016.

[89] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, and Thomas Feld. “Industry 4.0”. In: *Business & Information Systems Engineering*. Springer, 2014.

[90] Myriam Leggieri and Michael Hausenblas. “Interoperability of Two RESTful Protocols: HTTP and CoAP”. In: *REST: Advanced Research Topics and Practical Applications*. Springer, 2013.

[91] Dan Lin, Prathima Rao, Elisa Bertino, Ninghui Li, and Jorge Lobo. “EXAM: a comprehensive environment for the analysis of access control policies”. In: *International Journal of Information Security*. Vol. 9. Springer, 2010.

[92] Olga Liskin, Leif Singer, and Kurt Schneider. “Teaching Old Services New Tricks: Adding HATEOAS Support as an Afterthought”. In: *WS-REST ’11 - Proceedings of the Second International Workshop on RESTful Design*. 2011.

[93] Alex Liu, Fei Chen, JeeHyun Hwang, and Tai Xie. “Designing Fast and Scalable XACML Policy Evaluation Engines”. In: *IEEE Transactions on Compters*. Vol. 60. 2011.

[94] Alex Liu, Fei Chen, JeeHyun Hwang, and Tai Xie. “XEngine: A Fast and Scalable XACML Policy Evaluation Engine”. In: *SIGMETRICS ’08 - Proceedings of the 2008 ACM Internatio-*

nal Conference on Measurement and Modeling of Computer Systems. 2008.

[95] Markus Lorch, Dennis Kafura, and Sumit Shah. "An XACML-based Policy Management and authorization Service for Globus Resources". In: *GRID '03 - Proceedings of the 4th International Workshop on Grid Computing*. 2003.

[96] *Microsoft Active Directory*. `https://docs.microsoft.com/en-us/windows-server/identity/ad-ds/active-directory-domain-services`. Zuletzt abgerufen: 2018-06-30.

[97] Said Marouf, Mohamed Shehab, Anna Squicciarini, and Smitha Sundareswaran. "Adaptive Reordering and Clustering-Based Framework for Efficient XACML Policy Evaluation". In: *IEEE Transactions on Services Computing*. Vol. 4. 2010.

[98] Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. "Formalisation and Implementation of the XACML Access Control Mechanism". In: *ESSoS '12 - Proceedings of the 4th Iinternational Conference on Engineering Secure Software and Systems*. 2012.

[99] Agon Memeti, Besnik Selimi, Adrian Besimi, and Betim Cico. "A Framework for Flexible REST Services: Decoupling authorization for Reduced Service Dependency". In: *MECO'15 - Proceedings 4th Mediterranean Conference on Embedded Computing*. 2015.

[100] Philip Miseldine. "Automated XACML Policy Reconfiguration for Evaluation Optimisation". In: *SESS '08 - Proceedings*

of the fourth international workshop on Software engineering for secure systems. 2008.

[101] Philipp Montesano, Marc Hüffmeyer, and Ulf Schreier. "Outsourcing Access Control for a Dynamic Access Configuration of IoT Services". In: *IoTBDS '17 - Proceedings of the 2nd International Conference on Internet of Things, Big Data and Security*. 2017.

[102] Charles Morisset and Nicola Zannone. "Reduction of Access Control Decisions". In: *SACMAT '14 - Proceedings of the 19th ACM Symposium on Access Control Models and Technologies*. 2014.

[103] Canh Ngo, Yuri Demchenko, and Cees De Laat. "Decision Diagrams for XACML PolicyEvaluation and Management". In: *Computers Security*. Vol. 49. Elsevier, 2014.

[104] Se Won Oh and Hyeon Soo Kim. "Decentralized access permission control using resource-oriented architecture for the Web of Things". In: *ICACT '14 - Proceedings of the 2014 International Conference on Advanced Communication Technology*. 2014.

[105] *OpenAPI*. `http://www.openapis.org`. Zuletzt abgerufen: 2018-06-30.

[106] Ertem Osmanoglu. *Identity and Access Management*. Syngress, 2013, pp. 1–19.

[107] Savas Parastatidis, Jim Webber, Guilherme Silveira, and Ian Robinson. "The role of hypermedia in distributed system development". In: *WS-REST '10 - Proceedings of the First International Workshop on RESTful Design*. 2010.

[108] Cesare Pautasso. “On Composing RESTful Services”. In: *Software Service Engineering*. Ed. by Frank Leymann, Tony Shan, Willen-Jan van den Heuvel, and Olaf Zimmermann. 2009.

[109] Cesare Pautasso. “RESTful Web Services: Principles, Patterns, Emerging Technologies”. In: *Web Services Foundations*. Springer, 2013.

[110] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. “Restful Web Services vs. “Big“ Web Services: Making the Right Architectural Decision”. In: *WWW ’08 - Proceedings of the 17th International Conference on World Wide Web*. ACM, 2008.

[111] Mor Peleg, Dizza Beimel, Dov Dorib, and Yaron Denekamp. “Situation-Based Access Control: Privacy management via modeling of patient data access scenarios”. In: *journal of Biomedical Informatics*. Vol. 41. Elsevier, 2008.

[112] Nicole Pfeffermann. *Erweiterung von Restmodellierungswerkzeugen um Access Control Mechanismen*. Masterarbeit Hochschule Furtwangen, 2016.

[113] *RAML*. `https://raml.org/`. Zuletzt abgerufen: 2018-06-30.

[114] Kencana Ramli, Carroline Dewi Puspa, Hanne Riis Nielson, and Flemming Nielson. “The Logic of XACML”. In: *Formal Aspects of Component Software*. Springer, 2012.

[115] Leonard Richardson. “Developers Enjoy Hypermedia, But May Resist Browser-Based OAuth authorization”. In: *WS-REST ’10 - Proceedings of the 1st International Workshop on RESTful Design*. 2010.

[116] Leonard Richardson. “Developers Like Hypermedia, But They Don’t Like Web Browsers”. In: *WS-REST ’10 - Proceedings of the 1st International Workshop on RESTful Design*. 2010.

[117] Leonard Richardson and Mike Amundsen. *RESTful Web APIs - Services for a Changing World*. O’Reilly Media, 2013, pp. 1–16, 45–56.

[118] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O’Reilly Media, 2008, pp. 79–105.

[119] Erik Rissanen, David Brossard, and Adriaan Slabbert. “Distributed Access Control Management – A XACML-Based Approach”. In: *Service-Oriented Computing*. Springer, 2009.

[120] Ronald Rivest, Adi Shamir, and Leonard Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. In: *Communications of the ACM*. Vol. 21. 1978.

[121] Santiago Pina Ros, Mario, Lischka, and Felix Gomez Marmol. “Graph-Based XACML Evaluation”. In: *SACMAT ’12 - Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*. 2012.

[122] Pierangela Samarati and Sabrina Capitani de Vimercati. “Access Control: Policies, Models, and Mechanisms”. In: *Foundations of Security Analysis and Design*. Ed. by R. Focardi and R. Gorrieri. Springer, 2001.

[123] Robson Vincius Vieira Sanchez, Ricardo Ramos de Oliveira, and Renata Pontin de Mattos Fortes. “RestML: Modeling RESTful Web Services”. In: *REST: Advanced Research Topics and Practical Applications*. Springer, 2014.

[124] Ravi Sandhu. “The authorization leap from rights to attributes: maturation or chaos?” In: *SACMAT ’12 - Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*. 2012.

[125] Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. “Role-Based Access Control Models”. In: *IEEE Computer*. Vol. 29. IEEE Computer Society, 1996.

[126] Ravi Sandhu and Pierangela Samarati. “Authentication, Access Control, and Audit”. In: *ACM Computing Surveys*. Vol. 28. 1996.

[127] Silvia Schreier. “Modeling RESTful applications”. In: *WS-REST ’11 - Proceedings of the Second International Workshop on RESTful Design*. 2011.

[128] Hossein Shafagh and Anwar Hithnawi. “Poster Abstract: Security Comes First, a Public-key Cryptography Framework for the Internet of Things”. In: *Proceedings of the 4th IEEE International Conference on Distributed Computing in Sensor Systems*. 2014.

[129] Hai-bo Shen and Fan Hong. “An Attribute Based Access Control Model for Web Services”. In: *PDCAT ’06 - Proceedings of the Seventh International Conference on Parallel and Distributed Computing*. 2006.

[130] *Shibboleth Consortium - Privacy Preserving Identity Management*. `https://www.shibboleth.net/`. Zuletzt abgerufen: 2018-06-30.

[131] Munindar Singh and Michael Huhns. *Service-Oriented Computing: Semantics, Processes, Agents*. Wiley, 2006, p. 54.

[132] Martin Spasovski. *OAuth 2.0 Identity and Access Management Patterns*. Packt Publishing, 2013, pp. 7–11.

[133] Bernard Stepien, Amy Felty, and Stan Matwin. “Challenges of Composing XACML Policies”. In: *ARES ’14 - Proceedings of the Ninth International Conference on Availability, Reliability and Security*. 2014.

[134] *Sun PDP*. `http://sunxacml.sourceforge.net/`. Zuletzt abgerufen: 2018-06-30.

[135] *Swagger*. `http://swagger.io/`. Zuletzt abgerufen: 2018-06-30.

[136] Stefan Tilkov, Martin Eigenbrodt, Silvia Schreier, and Oliver Wolf. *REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web*. dpunkt Verlag, 2015, pp. 35–52, 87–109.

[137] Slim Trabelsi, Adrien Ecuyer, Paul Cervera Alvarez, and Francesco Di Cerbo. “Optimizing Access Control Performance for the Cloud ”. In: *CLOSER ’14 - Proceedings of the 4th International Conference on Cloud Computing and Services Science*. 2014.

[138] Chwei-Shyong Tsai, Cheng-Chi Lee, and Min-Shiang Hwang. “Password Authentication Schemes: Current Status and Key Issues”. In: *International journal of Network Security*. Vol. 3. 2006.

[139] Fatih Turkmen, Jerry den Hartog, Silvio Ranise, and Nicola Zannone. “Analysis of XACML Policies with SMT”. In: *Principles of Security and Trust*. Springer, 2015.

[140] Fatih Turkmen, Jerry den Hartog, Silvio Ranise, and Nicola Zannone. “Formal analysis of XACML policies using SMT”. In: *Computers & Security*. Elsevier, 2017.

[141] Dieter Uckelmann, Mark Harrison, and Florian Michahelles. *Architecting the Internet of Things*. Springer, 2011, pp. 1–22.

[142] Shambhu Upadhyaya. “Mandatory Access Control”. In: *Encyclopedia of Cryptography and Security*. Ed. by Henk van Tilborg and Sushil Jajodia. Springer, 2011.

[143] Sabrina De Capitani di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. “Over-encryption: management of access control evolution on outsourced data”. In: *VLDB’07 - Proceedings of the 33rd international conference on Very large data bases*. 2007.

[144] Henry Vu, Tobias Fertig, and Peter Braun. “Verification of Hypermedia Characteristic of RESTful Finite-State Machines”. In: *WS-REST ’18 - Proceedings of the 9th International Workshop on Web APIs and Service Architecture*. 2018.

[145] Ray Wagner, Earl Perkins, Gregg Kreizman, Felix Gaehtgens, and Ant Allan. *Identity and Access Management 2020*. Gartner, 2013.

[146] Kevin Wallis, Marc Hüffmeyer, Ayhan Soner Koca, and Christoph Reich. “Access Rules Enhanced by Dynamic IIoT Context”. In: *IoTBDS ’18 - Proceedings of the 3rd International Conference on Internet of Things, Big Data and Security*. 2018.

[147] Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in Practice - Hypermedia and Systems Architecture*. O’Reilly Media, 2010, pp. 1–20, 93–152.

[148] Matthias Wieland, Holger Schwarz, Uwe Breitenbücher, and Frank Leymann. “Towards Situation-Aware Adaptive Workflows”. In: *Proceedings of the 13th Annual IEEE Intl. Conference on Pervasive Computing and Communications Workshops: 11th Workshop on Context and Activity Modeling and Recognition*. IEEE, 2015.

[149] World Wide Web Consortium. “Simple Object Access Protocol (SOAP) Version 1.2”. In: (2007).

[150] World Wide Web Consortium. “Web Services Description Language (WSDL) Version 2.0”. In: (2007).

[151] Stephen S. Yau, Yisheng Yao, and Vageesh Banga. “Situation-aware access control for service-oriented autonomous decentralized systems”. In: *Proceedings of the 2005 International Symposium on Autonomous Decentralized Systems*. 2005.

[152] Eric Yuan and Jin Tong. “Attribute based access control (ABAC) for Web services”. In: *ICWS ’05 - Proceedings of the 2005 IEEE International Conference on Web Services*. 2005.

[153] Zhi-Kai Zhang, Michael Cheng Yi Cho, Chia-Wei Wang, Chia-Wei Hsu, Chong-Kuan Chen, and Shiuhpyng Shieh. “IoT Security: Ongoing Challenges and Research Opportunities”. In: *Proceedings of the 7th IEEE International Conference on Service-Oriented Computing and Applications*. 2014.

Abbildungsverzeichnis

Tabellenverzeichnis