Implementierung des Policy-Kontinuums

Markus Galda

8. Januar 2008

Technische Universität Braunschweig
Institut für Betriebssysteme und Rechnerverbund

Diplomarbeit

Implementierung des Policy-Kontinuums

von
cand. wirt.-inf. Markus Galda

Aufgabenstellung und Betreuung:
Prof. Dr.-Ing. L. Wolf. und Dipl.-Wirt.-Inf. T. Klie

Braunschweig, den 8. Januar 2008

Erklärung

Ich versichere, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Braunschweig, den 8. Januar 2008

Kurzfassung

Die vorliegende Diplomarbeit befasst sich mit der Implementierung des Policy-Kontinuums im Zusammenhang mit Policy Based Network Management. Besonderer Focus liegt hierbei auf dem Refinement Prozess und den dabei entstehenden Konflikten und Problemen. Die Arbeit erläutert die Grundlagen und benötigte Konzepte für die Implementierung. Nach dem Entwurf und der Modellierung soll das Policy-Kontinuum prototypisch in Java umgesetzt werden. Der Prototyp benutzt dabei das vorgestellte Konzept eines Reizwortkataloges, um ein automatisches Policy-Refinement durchzuführen.

Abstract

This thesis deals with the implementation of the policy continuum when it comes to policy based network management. The so-called refinement-process with all its problems and conflicts within this concept will be of special interest. The thesis illustrates the basics and useful concepts for the implementation. The policy-continuum-program is to be prototypically implemented in Java after a concept and a program-model are developed. The prototype uses the introduced concept of a keyword-file to perform an automated policy-refinement.

[Hier wird später die Aufgabenstellung eingefügt.]

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Problemstellung und Motivation

Die meisten der heutigen Netzwerke sind sehr komplex und wachsen stetig. Geräte und Anforderungen an Netzwerke sind dementsprechend unterschiedlicher und unübersichtlicher denn je.
Der Aufwand für Systemadministratoren, was Installation und Wartung betrifft, wird immer umfangreicher und zeitintensiver. Plötzlich auftretende Ereignisse oder Fehlerfälle erfordern immer wieder ein menschliches Eingreifen. Nicht nur die Ereignisse und Datenströme in Firmen-Netzwerken und größeren Systemen, auch die einfache Überwachung von Sensoren, beispielsweise in einem Krankenhaus (Puls, Herzfrequenz, Sauerstoffgehalt des Blutes, etc.), stellt hohe Anforderungen an das überwachende System bzw. an den überwachenden Menschen. [14]
Um dieser Situation Herr zu werden und große und komplexe Netzwerke verwalten und beherrschen zu können, gilt *Policy Based Network Management* (PBNM) als gute und effiziente Methode. PBNM bietet hier eine Art von Automatismus: mit Hilfe von Policies - im Grunde "einfache" Regeln - lassen sich Systeme oder Netze beschreiben und selbst verwalten.

Damit solche Regeln von der hohen allgemein-sprachlichen Ebene in eine maschinenverständliche Form gebracht werden können, bedarf es eines Prozesses, welcher die abstrakten Policies vereinfacht und ihre Komplexität deutlich reduziert. Die daraus entstandenen, low-level Policies lassen sich dann wiederum in konkrete (Geräte-) Konfigurationen überführen. Dieser Prozess der Überführung von high-level Policies in low-level Policies wird Refinement genannt.

Da dieses Refinement jedoch automatisch - also ohne Benutzerinteraktion - ablaufen soll, müssen dem System Informationen über seinen Aufbau und der durchzuführenden Aktionen bei auftretenden Phänomenen vorliegen. Das Programm bzw. das Gerät, welches ein Refinement durchführt, muss demnach Informationen über die Netztopologie zu seiner Verfügung haben.
Eine Beschreibung der Netztopologie könnte beispielsweise mit Hilfe von Semantic Web Technologien, wie Ontologien, modelliert und über die *Web Ontology Language* (OWL, Kapitel 2.4.2) zur Verfügung gestellt werden. Informationen bezüglich der auftretenden Zustände und auszuführenden Aktionen könnten in einer Art Wissensdatenbank abgelegt sein, auf die das System Zugriff hat. Low-Level-Policies, die aus dem Refinement resultieren, müssen dann ebenfalls in einer maschinenlesbaren Form abgespeichert werden, um sie später weiter automatisch verarbeiten zu können.

Die Problemstellung dieser Arbeit besteht folglich in der Darstellung und Modellierung des o.g. Refinements, der dabei entstehenden Probleme und Konflikte, sowie der

Problematik der Informationsbereitstellung, -verarbeitung und -speicherung für ein automatisches Refinement.

1.2 Kapitelübersicht

Kapitel 2 soll einige Grundlagen erläutern und Definitionen zur Thematik liefern. Auf Begriffe wie das PBNM selbst, Policy, der Refinement-Prozess und das Policy-Kontinuum soll detailliert eingegangen werden. Ebenso werden Probleme und Konflikte bei der Erstellung von Policies und im gesamten Prozess beschrieben und diskutiert. Auch die o.g. Konzepte des Semantic Web, die OWL und die dazu benötigten Sprachkonstrukte werden dargestellt.
Kapitel 3 beinhaltet eine Anforderungsanalyse des zu implementierenden Programms, ein daraus resultierender Entwurf mit Modellierungsansätzen, UML-Diagrammen und groben Ablaufbeschreibungen. Auch Konzepte bezüglich der Formate für die Datenverarbeitung der Implementierung werden erläutert.
In Kapitel 4 wird die endgültige Implementierung detailliert beschrieben und einige Besonderheiten dargelegt. Ein Programmtest sowie ein Vergleich mit anderen Implementierungen schließen sich in Kapitel 5 an. Das letzte Kapitel 6 soll abschließend die Ergebnisse und Probleme der Implementierung zusammenfassen und einen kurzen Ausblick auf eventuelle zukünftige Erweiterungen/Erweiterungsmöglichkeiten des Programms geben.

2 Grundlagen

Da es sich bei den folgenden Ausführungen und der späteren Implementierung um Policy-Based Network Management (im Folgenden PBNM) und den damit verbundenen Mechanismen handelt, sollen im Folgenden zunächst einige Grundlagen des PBNM dargelegt sowie der Begriff der Policy genauer erläutert werden.
Anschließend wird der Refinement-Prozess mit den dabei auftretenden Problemen vorgestellt, sowie das Policy-Kontinuum definiert werden. Abschließend werden die Konzepte des Directory Enabled Networking (DEN-ng) sowie des Common Information Model (CIM) vorgestellt.

2.1 Policy-Based Network Management

Beim PBNM geht es darum, ein Netz bzw. ein System mit Hilfe von Regeln und Richtlinien zu verwalten. Diese Regeln - auch Policies genannt - dienen der Konfiguration der einzelnen Systemkomponenten. Hervorgerufen wurde das Interesse an einem solchen Konstrukt durch immer größer und komplexer werdende Netze und die damit verbundene Schwierigkeit, diese Systeme effizient zu konfigurieren und zu administrieren.

Die Literatur bietet mehrere, und auch durchaus unterschiedliche, Definitionen des Begriffes Policy. Zum einen lassen sich mit Hilfe von Policies Ziele verfolgen bzw. umsetzen. Andererseits wird der Begriff Policy oft mit dem Begriff Ziel gleichgesetzt.

Auch die Form einer Policy kann durchaus unterschiedlich sein. So kann sie ein Dokument, eine Tabelle mit Optionen, eine Folge logischer Aussagen zur Automatisierung operationaler Entscheidungen oder ein Werkzeug sein, um Business-Ziele und Service-Prioritäten darzustellen und durchzusetzen.
John Strassner definiert eine Policy wie folgt: „Eine Policy ist eine Menge von Regeln, die dazu benutzt werden, den Status eines oder mehrerer Objekte zu verwalten und die Änderung bzw. Beibehaltung des Status zu kontrollieren.“[6]

Im Wesentlichen lässt sich eine Policy in drei Komponenten unterteilen, was in der Literatur ebenfalls durchaus unterschiedlich getan wird. So gibt es z.B. eine Unterteilung in

1. *Event*,
2. *Action* und
3. *Rule*.

Policy-*Events* werden als Zustände eines Systems und ihre operationale Realisierung betrachtet. *Actions* sind die Antwort(en) bzw. die Aktion(en), die das System im Falle des Eintretens eines Events geben bzw. durchführen soll. *Rules* wiederum werden als die Mechanismen bezeichnet, welche die Events mit den Actions verknüpfen und in einen Zusammenhang bringen [2].

Der Begriff „Rule“ wird jedoch nicht immer synonym verwandt. Es existieren sowohl Definitionen in denen Policy und Rule identische Bedeutung haben, als auch solche mit Teil-von-Beziehungen. Ein Beispiel einer weiteren Definitionsweise zeigt Abbildung 2.1 in der eine Rule als Teil einer Policy gesehen wird.

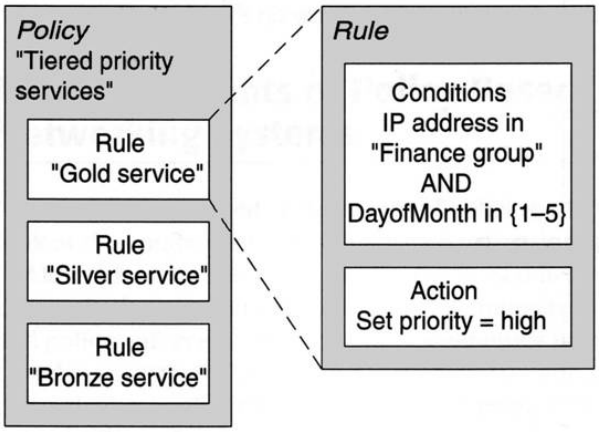


Abbildung 2.1: Rule als Teil einer Policy [1]

In der hier vorliegenden Arbeit soll ebenfalls eine Dreiteilung von Policies zugrunde gelegt werden. Anders als die bereits vorgestellten Teilungen gilt für die folgenden Kapitel und die Implementierung eine Unterteilung gemäß der DEN-ng-Policy (siehe Kapitel 2.3) in:

1. *Event*,
2. *Condition* und
3. *Action*

Als Event bezeichnet man das Auftreten eines Ereignisses. Condition stellt hier eine beim Auftreten eines Ereignisses geltende Bedingung dar, die sowohl mit wahr als auch mit falsch belegt sein kann. Mit Action wiederum wird die auszuführende Aktion bezeichnet. [7]

Des Weiteren müssen Policies persistent, d.h. nicht flüchtig, und wiederholbar sein, um für beliebig viele Ereignisse des selben Typs zu gelten. Der einmalige oder plötzliche Zugriff bzw. die Interaktion eines Administrators mit dem System wird dagegen nicht als Policy angesehen.[12]

Policies können die verschiedensten Aufgaben erfüllen:

- **Durchsetzung von Sicherheitsrichtlinien**:
  Ressourcenzugriff wird nur für bestimmte Benutzer gewährt.

- **Quality of Service** (QoS):
  Zeitkritische Daten, wie z.B. Pakete für VoIP-Anwendungen, haben Vorrang.
- **Definition und Anwendung von Geschäftsbedingungen** (Business Rules):
  Wird eine Bestellmenge X in Auftrag gegeben, gibt es einen Preisrabatt.
- **Dienstgütevereinbarungen** (SLAs):
  Das System muss zu 99% verfügbar sein.
- **Interaktionsvoraussetzungen**:
  Der Zugriff auf eine Applikation wird nur mit einer bestimmten Verschlüsselung erlaubt.

Traditionell verwendet man Policies zur Kontrolle des Zugriffs auf Ressourcen (Zugriffsrechte für Benutzer). Darüber hinaus fordern heutige Anwendungen ebenfalls eine Verknüpfung von Policies mit Dienstgütevereinbarungen (z.B. für das Routing zeitkritischer Daten, IP-Telefonie etc.).

Ein Policy-basierter Ansatz ein System zu verwalten erlaubt es außerdem, die Regeln, die das System verwalten, von der Funktionalität, die das System selbst zur Verfügung stellt, zu trennen. Die Anpassung an neue Gegebenheiten kann somit zur Laufzeit und ohne Systemstop erfolgen. Eine Reprogrammierung oder ein Neu-Aufsetzen des Systems ist daher nicht notwendig.[2]
Einen schematischen Aufbau eines solchen PBNM-Systems gibt Abbildung 2.2. Das darin gezeigte Element des Policy Decision Points (PDP) bezeichnet in einem PBNM-System den Ort/das Modul, an dem die Policies erstellt und verfeinert werden. Ein Policy Enforcement Point (PEP) stellt dagegen das Modul dar, mit deren Hilfe die formulierten Policies durchgesetzt werden, d.h. ihre Anwendung finden sollen.

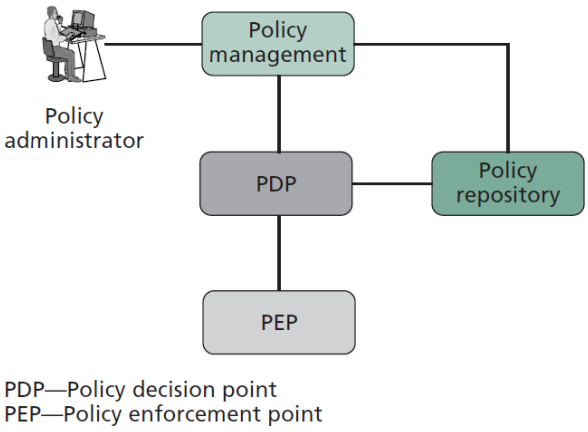


Abbildung 2.2: Schematische Darstellung eines PBNM-Systems [2]

Man betrachtet Policies auch als ein "zentrales Werkzeug (...), um die kontinuierlich steigende Komplexität und Dynamik verteilter Systeme"[12] besser beherrschen zu können. Indem ein System, im hier vorliegenden Fall ein Netzwerk, durch die Policies in der Lage ist selbst zu entscheiden, wie es sich beim Eintreffen bestimmter Situationen verhalten soll, ist keine Benutzerinteraktion mehr notwendig. Man verspricht sich

durch diese Selbstverwaltung eine Automatisierung.

Um Policies für ein PBNM zu erstellen und zu definieren, bedarf es eines Prozesses der als Policy Refinement bezeichnet wird. Dieser Prozess (siehe Kapitel 2.2) und das damit verbundene Policy-Kontinuum (in Kapitel 2.2.5 beschrieben) sollen im folgenden Kapitel 2.2 näher betrachtet werden.

2.1.1 Anwendungsmöglichkeiten

Die Literaturrecherche zur vorliegenden Ausarbeitung hat gezeigt, dass das Thema des Policy Based Managements in vielen Bereichen zur Anwendung kommt. Auch die weiter unten vorgestellten Konzepte bieten eine hohe Vielfalt an Einsatzmöglichkeiten. So zeigt sich, dass Policies auch für den Bereich von (kabellosen) SensorNetzwerken und HomeCare[15] interessant ist. Weiterhin halten Policies Einzug in e-Health Anwendungen.[16] Auch die Anwendung von Policies zur autonomen Fahrzeugsteuerung sei hier genannt. [17] Policies werden darüber hinaus dazu verwendet, die Flexibilität von Services zu erhöhen. [18] Die weiter unten in Kapitel 2.2.1 angesprochenen Probleme beim Refinement finden sich auch im Bereich der Sensor-Netzwerke wieder.[19] Die Konstrukte der OWL-Ontologien (Kapitel 2.4.2) eignen sich zudem zum Aufruf von Services.[20] Und sogar spezielle Überlegungen zur Behandlung von Telefonanrufen, bei IP-Telefonie eine Anrufkontrolle mit Hilfe von Ontologien durchzuführen, existeren. [21]

2.2 Der Refinement-Prozess und das Policy-Kontinuum

Innerhalb des PBNM findet der Prozess des Refinements statt. Es dient im Grunde dazu, Business-Ziele in Netzwerk-Ziele zu transformieren. Man versucht dabei wirtschaftliche Zielvorgaben in technische Policies umzuwandeln. Man spricht oft auch von High-Level Policies (Business-Ziele), die auf eine für Maschinen geeignete Form, Low-Level-Policies (technische Zielvorgaben), projiziert werden sollen. Die Beziehungen zwischen diesen Zielvorgaben und den letztendlichen technischen Policies bilden eine *Policy-Hierarchie*.

Die Literatur unterscheidet im Wesentlichen drei Ansätze zum Policy-Refinement:

1. Zielbasierter Ansatz (Kapitel 2.2.2)
2. Model-Checking (Kapitel 2.2.3)
3. Fallbasierte Schlussfolgerungen (Kapitel 2.2.4)

Eine weitere, vierte Betrachtungsweise stellt das Prinzip des Policy-Kontinuums dar, welches in Kapitel 2.2.5 noch einmal genauer betrachtet wird.

Bei der Erstellung und Formulierung von Policies können unterschiedliche Probleme und Konflikte (Kapitel 2.2.1) auftreten, die zunächst etwas genauer betrachtet werden sollen, bevor sich Kapitel 2.2.2 mit den Ansätzen zum Refinement beschäftigt.

2.2.1 Probleme beim Refinement

Ganz besonders beim Prozess des Policy-Refinements kann es zu Konflikten und sich ausschließenden Zuständen kommen. Das menschliche Gehirn ist durch Kombination formaler und intuitiver Regeln sowie informaler Vorgänge in der Lage, Konflikte zu erkennen, zu verhindern und mehr oder minder optimal zu lösen. Automatisierte Systeme sind jedoch gezwungen einen erheblich formaleren Ansatz zur Erkennung, Vermeidung und Lösung von Konflikten zu wählen. Dies kann sogar zu einer völligen Fehlfunktion - hier zum Absturz - des Systems führen.
So kann es durchaus sein, dass sich definierte (Business-)Ziele, aus denen später Policies entstehen sollen, zuwider laufen. Policy-Konflikte treten laut IETF immer dann auf, sobald sich die Actions zweier Policies widersprechen. Die Instanz, welche die Policy ausführt, wäre demnach nicht in der Lage zu entscheiden, welche Action auszuführen ist. Implementierungen von Policy-Systemen müssen deshalb eine Konflikterkennung und/oder -vermeidung bzw. -lösung anbieten, um diese Situation auszuschließen. [22]

Abbildung 2.3 beschreibt eine Reihe von unterschiedlichen Konflikt-Typen. Gut zu erkennen ist darin die Unterteilung in drei „Arten“ von Konflikten:

1. den Konflikt der *Modalitäten*,
2. den Konflikt der *Ziele* aufgrund von Befehlen/*Anweisungen*, und
3. den Konflikt der *Ziele* aufgrund von *Autorität*.

Weiterhin unterteilt sich die erste Kategorie in Positiv/Negativ-Konflikt und Konflikt zwischen Pflicht und Befugnis.

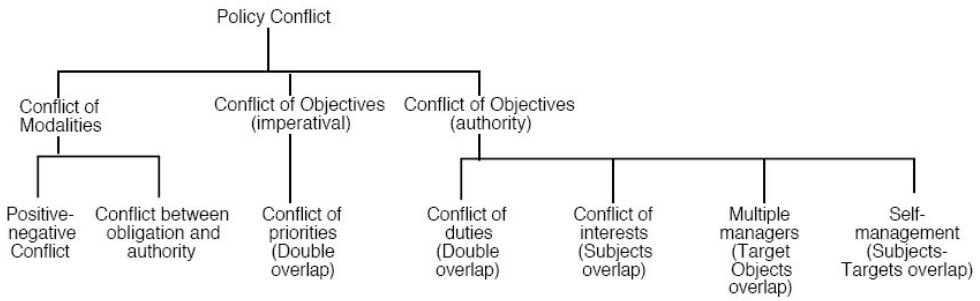


Abbildung 2.3: Klassifizierung von Policy-Konflikten [3]

Zwei sich zuwider laufende Anweisungen stellen hier den Positiv/Negativ-Konflikt dar, wogegen eine Anweisung, zu deren Ausführung man nicht die Befugnis hat den zweiten Fall beschreibt. Als Beispiel seien hier die gleichzeitig eintreffenden Befehle genannt: „an der nächsten Ecke links“ und „an der nächsten Ecke rechts“ oder dass jemand die Aufgabe bekommt, einen Mitarbeiter zu entlassen, dafür aber nicht die benötigte Hierarchiestufe hat.
Der einfachste Fall des Zielkonflikts ist der Konflikt von Prioritäten - auch doppelte Überlappung genannt. Er kann dann auftreten, wenn beispielsweise Personalkosteneinsparung und Personalentwicklung die gleiche Priorität haben.

Daneben existieren, wie in der Abbildung 2.3 zu sehen, noch weitere Konflikte. Unter anderem der Interessenkonflikt, welcher dann auftritt, wenn zum Beispiel zwei Manager mit gleicher Befugnis und gleicher Gewichtung ihrer Anweisungen, unterschiedliche Interessen an einem Projekt haben (Zeitansprüche/Qualitätsansprüche oder Vermarktung/Forschung). Weiterhin ist es möglich, dass Policy-Actions, die offensichtlich durch verschiedenartige Events hervorgerufen wurden, durch veränderte Gegebenheiten plötzlich interagieren. Policy-Actions können also nach gewisser Zeit durch veränderte Bedingungen konfliktär sein. [23]

Im Gegensatz zu Policy-Konflikten, definieren sich Policy-*Fehler* wie folgt: Policy-Fehler treten immer dann auf, wenn der Versuch eine Policy-Action durchzusetzen fehl schlägt. Dies kann die Folge eines temporären Zustandes oder dauerhaften Unterschiedes zwischen Policy-Action-Anforderung und Policy-Durchsetzungsfähigkeit des Gerätes sein. [22]

Bei allen Konflikten (und auch Fehlern) ist das Konzept der Überlappung von Policies von essentieller Bedeutung. Dort wo ein Teil einer Policy ebenfalls Teil mindestens einer weiteren Policy ist, spricht man von Überlappung. Haben zwei Policies keine Gemeinsamkeiten bzw. gemeinsame Elemente, können auch keine Konflikte zwischen ihnen auftreten. [3] Alle hier genannten Konfliktarten können beim Refinement auftreten. Die verschiedenen Vorgehensweisen beim Refinement selbst werden im Folgenden genauer erklärt.

2.2.2 Zielbasiertes Policy-Refinement

Vereinfacht ist der zielbasierte Ansatz das Erreichen von Zielen durch Formulierung von dazu notwendigen Teilzielen und die Bildung einer Zielhierarchie (siehe Abb 2.4).

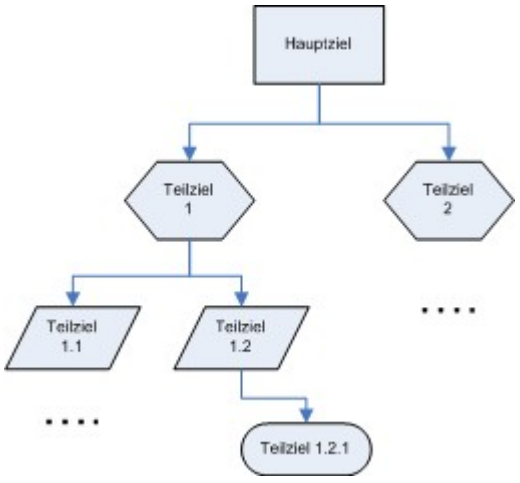


Abbildung 2.4: Zielhierarchie

Da es hier aber um das spezielle Thema des PBNM geht, hat auch der zielbasierte Ansatz eine spezielle Vorgehensweise. Der zielbasierte Ansatz zum Policy-Refinement beschäftigt sich damit, einen bestimmten Quality of Service-Grad zu erreichen. Netz-

Verfügbarkeit und Zugangskontrolle zum Netz gehören dabei ebenso zum QoS wie dynamische Bandbreitenkontingentierung. Man versucht also die oben angesprochenen SLAs zu befriedigen und leitet die QoS-Policies aus den SLA-Vorgaben ab. Um ein gegebenes High-Level-Ziel zu erreichen werden mit Hilfe von Zielausarbeitung und abduktiver Argumentation (Ableitung unbekannter Ursachen aus bekannten Effekten oder Konsequenzen) Strategien abgeleitet, die dem High-Level-Ziel gerecht werden.
Es geht hier also hauptsächlich um die Erreichung eines bestimmten QoS-Grades der in vorher definierten SLAs festgelegt wurde.[24]

2.2.3 Policy-Refinement mit Hilfe von Model-Checking

Refinement per Model-Checking ist eine Erweiterung des zielbasierten Ansatzes. Mit Hilfe von temporaler Logik und der Analyse reaktiver Systeme wird ein zielbasiertes Refinement ermöglicht.[4] Dabei werden High-Level-Ziele in verfeinerte Ziele umgewandelt, um darauf weitere Prozesse anzuwenden und so die notwendigen Informationen zur Zielerfüllung zu extrahieren. Vereinfacht lässt sich der gesamte Prozess wie in Abbildung 2.5 darstellen.

Model-Checking stellt ein Verfahren dar, welches es ermöglicht zu verifizieren, ob ein Merkmal oder eine Funktion für ein System Gültigkeit besitzt.[25] Das Model-Checking selbst wird, in Kombination mit linearer temporaler Logik, dazu verwandt, Policies zu analysieren.

Dabei sind zwei Funktionalitäten von Bedeutung: Zielmanagement-Aufgaben und die Policy-Refinement-Mechanismen. Die Zielmanagement-Aufgaben untergliedern sich in die Bereiche Zielausarbeitung und Zielauswahl. Während der Zielausarbeitung dokumentiert und klassifiziert der Benutzer (hier ein Administrator bzw. Experte für das vorliegende Problem) Ziele anhand eines Zielgraphen und speichert diese in einer Zielgraphdatenbank.
Dabei werden schon erste Refinement-Muster angewandt, um die Ziele in Teilziele aufzuteilen, die in logischer Verknüpfung das Hauptziel ergeben. In der Zielauswahl-Phase wählt ein Berater bzw. qualifizierter Benutzer die Ziele aus dem Zielgraphen aus, welche am besten mit seinen Vorstellungen und Business-Zielen korrespondieren.

Die Policy-Refinement Mechanismen beinhalten

- eine *Formulierung von Voraussetzungen* um die Zielerfüllung zu charakterisieren,
- eine *Abfrage des Systemverhaltens* zur Koordination der Ausführung des Datenerfassungssystems und des internen Ablaufs,
- die *Anwendung der übersetzten Policies*, welche die Policy-Elemente aus dem Systemverhalten, sowie den daraus gelieferten Daten, abstrahiert,
- und das Transformieren bzw. Speichern einsetzbarer Policies.

Ausgelöst werden diese Mechanismen durch eine Anfrage ein Policy Refinement auszuführen. [4] Eine Anfrage stellt in diesem Fall eine Nachricht dar, die einen Policy-verwandten Dienst anfordert. Dies können sowohl eine Menge von Policy-Regeln, als

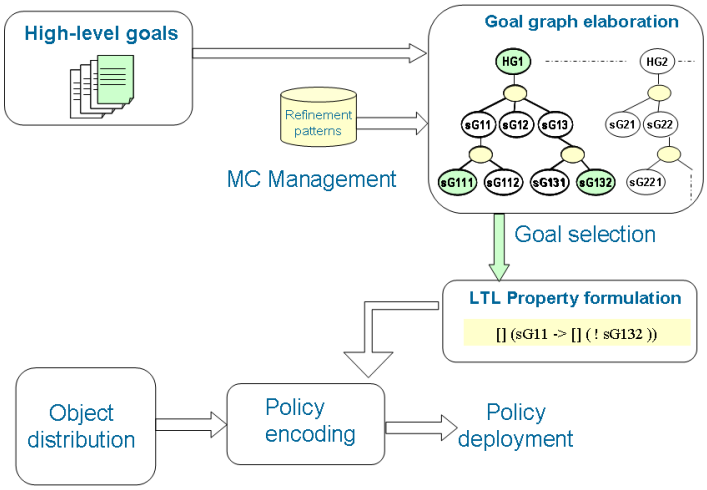


Abbildung 2.5: Schematische Darstellung des Refinements durch Model-Checking, in Anlehnung an [4]

auch die durchzusetzenden Policy-Actions sein. Wird eine Anfrage von einem Policy Enforcement Point (PEP) an einen Policy Decision Point (PDP) gesendet, spricht man von einem „policy decision request“. [22]

2.2.4 Policy-Refinement durch fallbasierte Schlussfolgerungen

Policies in spezielle Konfigurationen umzuwandeln stellt in jedem speziellen Fall neue spezifische Anforderungen an den Refinement-Prozess. Mit Hilfe von fallbasierten Schlussfolgerungen versucht man allgemeinere Transformationsregeln zu finden, die nicht zwingend ein spezifisches Problem und eine entsprechend spezifische Transformation in eine Policy voraussetzen.
Eine (relativ allgemeine) schematische Darstellung eines PBNM-Systems mit fallbasierten Schlussfolgerungen ist in Abbildung 2.6 zu sehen.

Dabei werden drei Arten von Refinement-Szenarios unterschieden:

1. Transformation mit Hilfe statischer Regeln
2. Transformation durch Nachschlagen innerhalb einer Policy-Tabelle
3. Transformation durch fallbasierte Schlussfolgerungen

Szenario 1 stellt dabei den einfachsten Fall dar. Durch seine Schlichtheit kann es hilfreich bei der Vereinfachung der Policy-Sprache von System-Administratoren sein.
Hier wird angenommen, es existiert eine Menge statischer Transformations-Regeln um Policies bzw. High-Level Ziele in maschinenlesbare Konfigurationen bzw. Low-Level Ziele zu überführen. Die Regeln folgen dabei einer Policy-Sprache, die komplexer und detaillierter ist, als die Sprache der Ziel-Policies eines System-Administrators.
Die Formulierung dieser Regeln obliegt einem Experten, welcher die Details des Systems, sowie die Definitionen und Wichtigkeit diverser Ziele (z.B. die Schwierigkeit

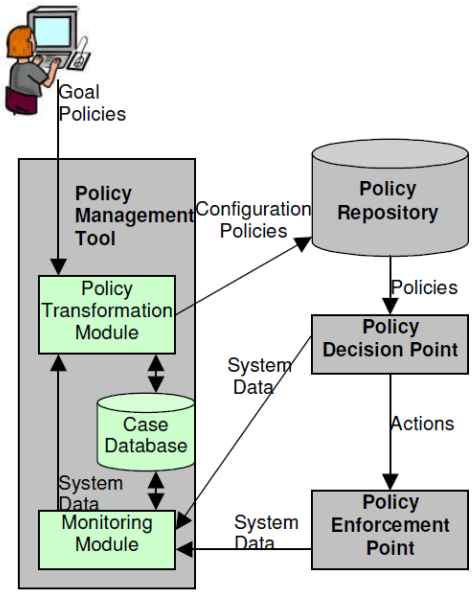


Abbildung 2.6: Architektur eines PBNM-Systems mit fallbasierter statischer Policy-Transformation, [5]

eine hohe Systemverfügbarkeit im Sinne von benötigter Systemleistung, Sicherheitsrichtlinien, etc. umzusetzen) genau kennt. Ein Übersetzungsmodul transformiert die so definierten Ziele in Low-Level Konfigurationen, indem es die Definitionen der Transformationsregeln zu Hilfe nimmt. Dieses Verfahren eignet sich besonders für Systeme, in denen Gruppen von Administratoren, mit jeweils unterschiedlichen (Zugriffs-)Rechten der Mitglieder bezüglich der Manipulation von Policies, existieren.

Szenario 2 macht von der Annahme gebrauch, dass das Transformationsmodul eine Tabelle zur Verfügung hat, welche die Policies enthält, die für das System zutreffend sind. Ein Systemadministrator kann nun eine Anfrage an das Modul mit einer Menge von Konfigurationsparametern stellen, um eine Menge von Zielen zu erhalten, die mit Hilfe der gegebenen Eingaben erreicht werden können. Das System muss nun die Ergebnis-Policies mit den Zielvorgaben abgleichen und diejenigen auswählen, welche zur Zielerreichung notwendig sind.

In Szenario 3 wiederum bedient sich das Transformationsmodul dem Wissen, des beobachteten Verhaltens des Systems der Vergangenheit, um das jetzige und zukünftige Verhalten voraus zu sagen. Das System „lernt“ sozusagen aus vergangenem operationalem Verhalten.
Notwendig dafür ist das Anlegen einer Datenbank, welche die vergangenen Verhaltensweisen und Aktionen beinhaltet. Die Fälle bilden dabei eine Kombination aus Systemkonfigurationsparametern und Geschäftszielen, die durch eine spezifische Kombination der Parameter erreicht wurden. Werden nun Parameter für ein neues Geschäftsziel benötigt, wird die Datenbank zu Rate gezogen um Parameter zu erhalten, die dem Ziel gerecht werden.
Dabei wird entweder der am ehesten zutreffende Fall gewählt, oder eine Interpolation zwischen den Parametern und einer Fall-Menge durchgeführt, um angemessene Para-

meter zu identifizieren.
Die Effektivität dieser Methode hängt jedoch stark davon ab, wie gut bzw. zahlreich die Menge der vorhandenen vergangenheitsbezogenen Daten ist. Bei kurzer Systemlaufzeit existieren keine bzw. kaum Daten auf deren Basis Entscheidungen getroffen werden können.

Dieses Problem kann entweder mit Hilfe von Heuristiken oder mit einer Menge von Initialfällen, die beim Systemstart bereits in der Datenbank vorhanden sind, gemildert und evtl. sogar umgangen werden. Ebenso könnte die Datenbank mehr Informationen enthalten als notwendig sind. Dies wären zum Beispiel Parameter, die in keinem Zusammenhang zum neuen Ziel stehen, aber bei vergangenen, passenden Vorgängen von Relevanz waren.
Auch könnten Inkonsistenzen vorhanden sein, wenn beispielsweise zwei passende Fälle in unterschiedlichen Zielen resultieren (siehe auch Kapitel 2.2.1). Und schliesslich besteht die Möglichkeit fehlender oder fehlerhafter Messwerte in Folge von Messfehlern bzw. Problemen bei der Messung (plötzlicher Systemausfall während der Aufnahme eines Falles in die Datenbank, etc.). [5]

2.2.5 Das Policy-Kontinuum

Generell versucht man beim Refinement von der höchsten (Business) Ebene auf die niedrigste (technische) Ebene zu gelangen. Dieses Vorgehen, das Durchlaufen verschiedener Ebenen bis auf die Niedrigste, wird im Konzept des Policy-Kontinuums verwendet. Sämtliche in Abbildung 2.7 gezeigten Ebenen des Kontinuums befassen sich dabei mit den spezifischen Problemen und Anwendungen, die auf der jeweiligen Schicht zu finden sind. Bei der Verfeinerung der Policies von Business-Zielen in technische Ziele werden die verschiedenen Ebenen - auch Sichten (engl. views) genannt - durchlaufen.

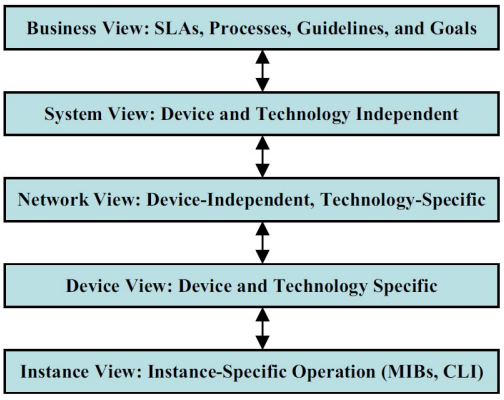


Abbildung 2.7: Das Policy-Kontinuum [6]

Ziel des Policy-Kontinuums ist es nun, mit Hilfe der Sichten, von high-level Policies auf low-level Policies (und umgekehrt, was für die vorliegende Ausarbeitung jedoch nicht von Relevanz ist) zu schliessen und zu formulieren. Auf Grund von Informationskombinationen der jeweiligen Schichten sollen so letztendliche technische Konfi-

gurationen und maschinenlesbare Befehle erzeugt werden. Dieses Konzept wird auch für die Implementierung nützlich sein. 3.2.2

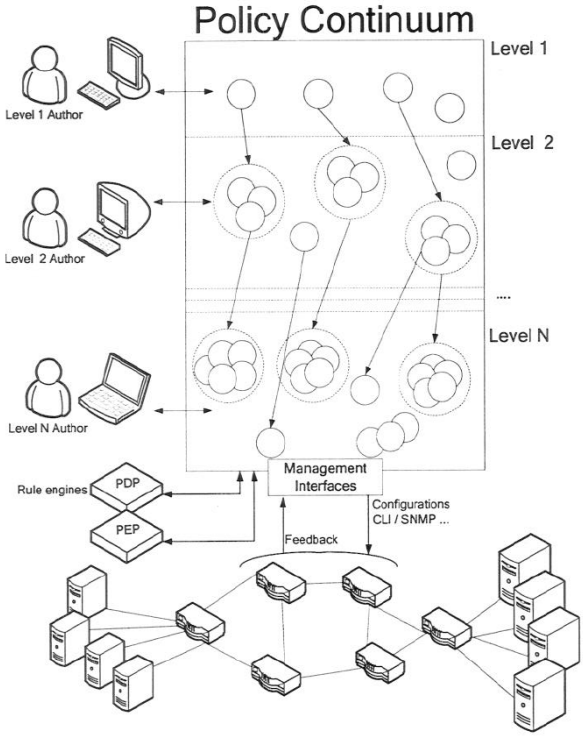


Abbildung 2.8: Abstrakte Darstellung des Policy-Kontinuums, aus [7]

Grundlage des Policy-Kontinuums ist die Überlegung, dass einzelne Policies verschiedene Aspekte, oder Ebenen, einer Menge zueinander in Relation stehender Policies verkörpern. Zusammen genommen rufen sie ein bestimmtes Verhalten des Netzwerkes vor. Jede der in Abbildung 2.7 gezeigten Ebenen ist dabei auf einen anderen „Kundenkreis“ bzw. Teilbereich des Netzwerkes, mit individuellen Anforderungen und/oder Bedürfnissen bezüglich Informationen, spezialisiert.
Ist ein Benutzer beispielsweise nur an Informationen bezüglich einer Dienstgütevereinbarung interessiert, sind Informationen über das Routing von Paketen oder die Art und Weise der Router, wie bei der Weiterleitung von Paketen gepuffert wird, für ihn uninteressant. Ein Netzwerk-Administrator wiederum hat andere Anforderungen an den Informationsgrad und die Detailtiefe technischer Informationen. Für die Router-Programmierung benötigt er daher eine vollständig andere Beschreibung und Darstellung einer Policy.

Im Kern ist das Policy-Kontinuum ein Modellierungswerkzeug. Die Darstellung von Policies auf hoher Abstraktionsebene wird durch das Kontinuum stark vereinfacht. Dies ist besonders dann hilfreich, wenn ein bestimmtes gewünschtes Verhalten, im Zusammenhang mit abstrakten Konzepten wie Produkten und Diensten eines Netzwerkes oder Preisen einer Transaktion, festgelegt werden soll.
Des Weiteren stellt es Mittel bereit, um High-Level Policies mit konkreten Konfi-

gurationen in Bezug zueinander zu bringen. Spezielle Verbindungen zwischen Policies, die auf unterschiedlichen Ebenen des Kontinuums definiert werden, erzeugen so Abhängigkeiten zwischen einer Reihe von Policies. Dadurch können Veränderungen auf einer Ebene direkte Auswirkungen auf die davon abhängigen Policies anderer Ebenen haben. Diese Abhängigkeiten sind jedoch nicht zwingend vorhanden. Je nach Definition und/oder Spezifikation der Policies können Policies auch allein auf einer Ebene existieren. Abhängigkeiten zwischen Policies auf verschiedenen Ebenen sind dabei eng mit den Abhängigkeiten von Entitäten eines Informationsmodells verbunden. Ein abstraktes Beispiel des Policy-Kontinuums zeigt Abbildung 2.8.
Dabei werden drei charakteristische Eigenschaften von Policies innerhalb des Kontinuums deutlich:

1. Eine Policy kann auf jeder Ebene des Kontinuums exisitieren, ohne dabei zwingend mit Policies anderer Ebenen verknüpft zu sein. Dadurch haben Policies die Fähigkeit allein auf einer Ebene zu existieren und Entitäten zu verwalten, die nur für eine bestimmte Gruppe von Benutzern relevant ist.

2. Eine Policy kann sich auf eine Reihe von weiter unten liegenden Policies beziehen.

3. Eine Policy kann zusätzlich zu mehr als einer höher liegenden Policy gehören. Dadurch ist es möglich eine Policy zur Erfüllung von Zielen höher liegender Policies wieder zu verwenden. Dies könnte dann gegeben sein, wenn zum Beispiel mehrere Business-Policies ein allgemeines, wiederkehrendes Verhalten von einer Firewall oder einem Router verlangen.

Die Ebene, auf der eine Policy normalerweise auftritt, definiert auch wie die Policy letztendlich durchgesetzt wird. Policies auf niedrigeren Ebenen des Kontinuums können direkt an Systemkomponenten durchgesetzt werden, wie beispielsweise Zugangskontrollen an Servern, Firewalls oder Routern. Da diese Komponenten durch Konfigurationen zu einem bestimmten Verhalten gebracht werden können, werden Policies in entsprechenden Programm- bzw. Konfigurationscode transformiert, den die Geräte verstehen und der sie zum gewünschten Verhalten bewegt.

Im Gegensatz dazu benötigen Policies auf höheren Kontinuums-Ebenen Entscheidungskomponenten und Durchsetzungskomponenten (Policy Enforcement Point) um das gewünschte Verhalten auf die verwalteten Entitäten zu übertragen. In einem solchen Fall wird eine Policy nicht direkt in eine Konfiguration übersetzt, sondern in Skripte die von Regel-Engines weiter verarbeitet werden können. [1] Diese Engines können auf Events reagieren, Conditions auswerten und Actions auf die entsprechenden Entitäten übertragen und dort durchsetzen. Dabei können High-Level-Policies das Verhalten der Regel-Engine durchaus beeinflussen. Aus diesem Grund wird ein Stapel-Speicher angelegt, in dem diejenigen Policies gespeichert werden, die das Verhalten der jeweils darunter liegenden Ebene beeinflussen. [7]

Die Literatur formuliert für die Umsetzbarkeit des Policy-Kontinuums Anforderungen. Diese Anforderungen benutzt man, um eine formale Spezifikation des Policy-

[1] Dieses Konzept wird auch für die Implementierung im Zusammenhang mit Ponder2 noch wichtig, wie in Kapitel 2.5 noch erläutert wird.

Kontinuums abzuleiten. Gemeinsam mit einer Reihe von Grund-Operationen bilden die Anforderungen eine Anbindung an das Policy-Kontinuum. Anforderungen sind im Einzelnen:

- Das *Erstellen einer Policy*. Zu Anfang des Prozesses müssen Policies von einem „Policy-Autoren“ erzeugt werden.
- Die *Abfrage einer Policy*. Mit Hilfe von Suchkriterien soll die Möglichkeit gegeben sein, eine Menge von Policies zu identifizieren.
- *Aktualisierung* einer vorhandenen Policy. Soll eine Policy im Nachhinein verändert werden, soll die Policy nicht gelöscht und neu erstellt werden müssen.
- Das *Löschen einer Policy* aus einer bestimmten Ebene. Ist eine Policy nicht mehr notwendig oder für das Netzwerk nicht mehr angebracht, kann sie vollständig gelöscht werden
- *Analyse einer Policy*. Wurde eine Policy erstellt, muß ihre Form auf Korrektheit überprüft werden. Dies erfolgt meist durch eine syntaktische Analyse. Ebenso müssen Konflikte erkannt werden können.

Diese Anforderungen an das Policy-Kontinuum bedürfen einer hohen Flexibilität bei der Verknüpfung von Policies. Auch komplexe Beziehungen müssen vom Kontinuum abgeleitet werden können. Eine gründliche und effektive Analyse der Policies vor, während und nach jeder Veränderung ist daher unabdingbar. [7]

Das DEN-ng Informationsmodell bietet eine explizite Unterstützung der in Abbildung 2.7 beschriebenen Sichten. Daher soll es im Folgenden genauer betrachtet werden.

2.3 DEN-nG und CIM

Directory Enabled Networking–new generation, oder kurz DEN-nG genannt, ist eine Spezifikation und Initiative der Distributed Management Task Force (DMTF). [26] Eine gute Übersicht zum DEN(-ng) findet sich im „Internetworking Technology Overview“ der Firma Cisco Systems. [27] Ziel des DEN-ng ist es, einen Industriestandard zu entwickeln, wie Informationen über ein Netzwerk, seine User, Anwendungen und Daten konstruiert und in einem zentralen Verzeichnis abgespeichert werden können. [2] Es stellt eine Weiterentwicklung des DEN-Standards dar. [6]
Die „new generation“-Erweiterung beinhaltet im Wesentlichen drei Verbesserungen/Erweiterungen des vorherigen DEN:

1. Eine Policy wird als Kontinuum von verwandten Subpolicies dargestellt (wie auch in Abbildung 2.7 zu sehen),
2. die Übersetzung bzw. Umwandlung von Business-Rules in Konfigurationen für Geräte und Dienste (inklusive des Prozesses, welcher die Konfigurationen anwendet),

[2] Die DTMF ist ein Zusammenschluss von Firmen, die es sich zum Ziel gesetzt haben, offene Managementstandards zu entwickeln.[28]

3. eine weitere Verfeinerung der Verwendung von Polices um die Veränderung, entsprechend eines Modells eines endlichen Automaten, einer Entität zu kontrollieren. Somit ist es möglich zu definieren, auf welche Weise eine Entität verwaltet werden soll. [29]

Weiterhin ist das DEN-ng ein vereinigtes Modell, das heißt dass es in der Lage ist, Inhalte anderer Quellen zu importieren. Um seinen Modell-Inhalt zu generieren bedient sich DEN-ng des UML-Metamodells, um standardisierte Werkzeuge und Bausteine zur Verfügung stellen zu können. Begrifflich ist DEN-ng ein erweiterbarer, UML-basierter Framework, in den andere Modelle integriert werden können und der mit (Programmier-)Sprachen verknüpft werden kann.[6]

Grundlage der Überlegungen des DEN ist das Common Information Model (CIM). Das CIM ist ein objektorientiertes Informationsmodell der DTMF. [22] Es stellt eine allgemeine Definition von Managementinformationssystemen, Netzwerken, Anwendungen und Services bereit, und erlaubt darüber hinaus eigene Erweiterungen zu definieren. Dadurch soll die Möglichkeit gegeben sein, semantische Managementinformationen zwischen Systemen und/oder eines Netzwerkes auszutauschen. [30]

Das CIM setzt sich aus zwei Komponenten zusammen:

- Die *CIM Spezifikation* beschreibt die abstrakten Modellkonstrukte und Prinzipien des Informationsmodells detailliert.
- Das *CIM Schema* definiert die textuale Sprachdefinition zur Repräsentation des Informationsmodells. [22]

Auch im CIM-Schema existiert somit die Definition einer Policy. Die Darstellung einer CIM-Policy im UML-Format findet sich in Abbildung 2.9 wieder. Hier werden auch schon zwei der in Kapitel 2.1 genannten Komponenten einer Policy verwendet: `PolicyCondition` und `PolicyAction`.

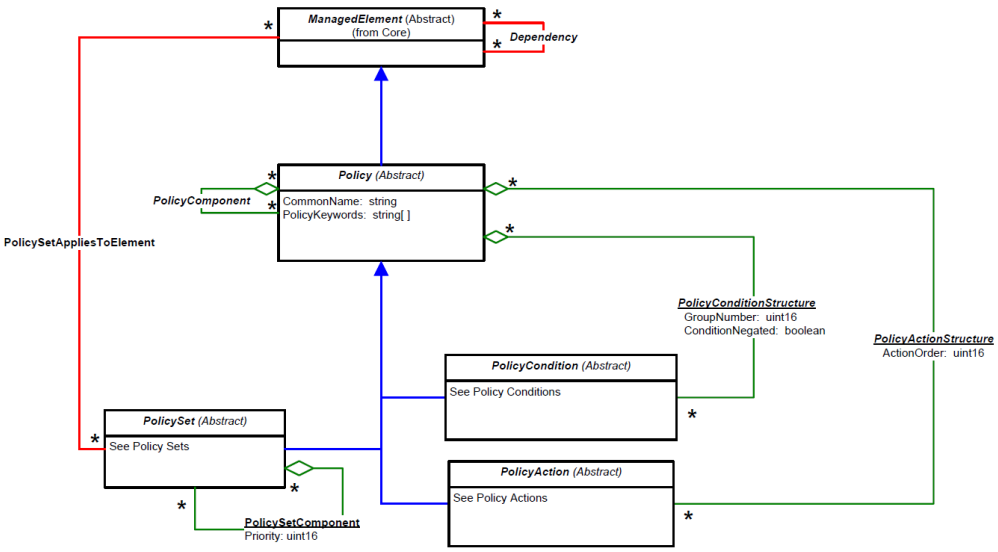


Abbildung 2.9: UML-Darstellung einer CIM-Policy, [8]

DEN-ng wiederum gründet auf dem NGOSS (New Generation Operations Software and Systems), welches es „ermöglicht Geschäftsprozesse innerhalb der Netzwerk-Infrastruktur eines Unternehmens automatisiert umzusetzen und auszuführen.“ [10] Das NGOSS-Konzept wurde im Tele-Management Forum (TMF) entwickelt. [3]

Wichtig für die spätere Implementierung des Policy-Kontinuums ist das DEN-ng Konzept einer Policy, da es - im Gegensatz zu den meisten Informationsmodellen - einen anderen Ansatz zur Repräsentierung einer Policy verfolgt. Im DEN-ng stellt eine Policy einen Container dar, welcher aus vier Komponenten besteht:

1. Den Metadaten,
2. der Event-Klausel,
3. der Condition-Klausel,
4. und der Action-Klausel.

Eine Klausel stellt hier einen Ausdruck bestehend aus einer Menge von Termen dar. [6] Policies selbst formuliert DEN-ng analog zu Abbildung 2.10.

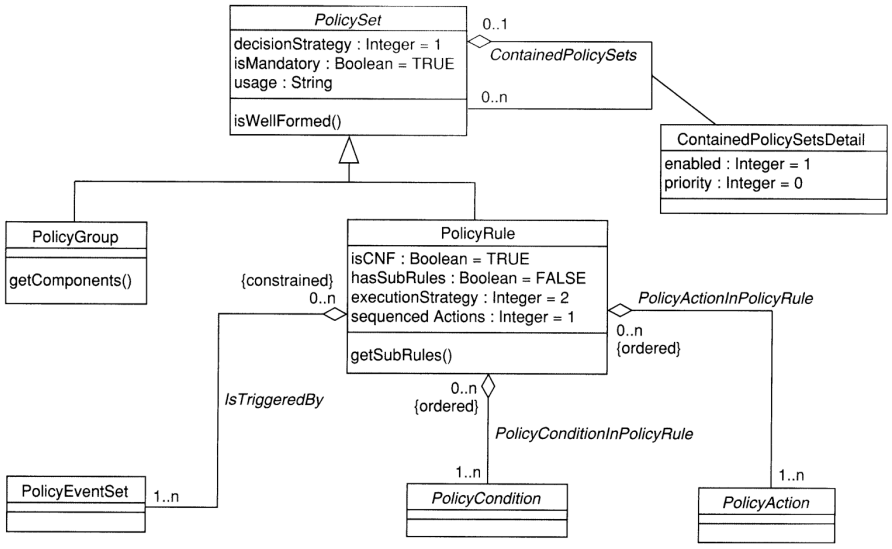


Abbildung 2.10: DEN-ng Konzept einer Menge von Policies [6]

Der Bezug zu den o.g. vier Komponenten einer Policy wird hier deutlich. Eine Policy (hier `PolicyRule`) wird von einem Event hervorgerufen. Weiterhin beinhaltet eine Policy eine Condition-Klausel (`PolicyCondition`) und eine Action-Klausel (`PolicyAction`).
Weitere Metadaten finden sich in der `PolicyGroup`, welche Policies in Gruppen faßt, sowie dem `PolicySet`. Das `PolicySet` wiederum enthält Details bezüglich der Priorität, ob eine Policy aktiv ist, ob eine Policy wohldefiniert ist, oder sogar ob eine Policy zwingend notwendig ist.

[3] Das Tele-Management Forum beschäftigt sich mit der Entwicklung von Operation Support Systems und Business Support Systems.

2.4 Semantic Web

Um die Datenformate für die Eingabe der Netztopologie (wie später im Entwurfskapitel 3.2 zu sehen) einheitlich zu gestalten, sind die Konzepte des Semantic Web von Nutzen. Das Semantic Web wurde aufgrund der Ideen von Tim Berners-Lee durch das W3C [31] initiiert. Vereinfacht soll das World Wide Web um maschinenlesbare Informationen erweitert werden. Informationen sollen so eine wohldefinierte Bedeutung bekommen. Somit soll eine bessere Zusammenarbeit zwischen Mensch und Computer gewährleistet werden.[10] Die Konzepte des Semantic Web aus Kapitel werden in [32] kurz zusammengefasst und diskutiert.
Grundlage für diese Erweiterungen sind Wissens-Repräsentationssprachen wie RDF (Resource Description Framework) (Kapitel 2.4.3) sowie OWL (Web Ontology Language) (Kapitel 2.4.2). Programme sind dadurch in der Lage, die semantischen Informationen auszulesen und mit neuen Daten in Verbindung zu bringen. Auch ein Austausch mit anderen Programmen (z.B. Software-Agenten) ist denkbar. Darüber hinaus könnten sogar Programme, die nicht für eine Zusammenarbeit konzipiert wurden, so interagieren - was momentan im World Wide Web nicht möglich wäre.
Ziel der Semantic Web Konzepte ist es, Informationen einheitlich und in verständlicher Form zur Verfügung zu stellen. Dabei setzen die Konzepte auf die Erweiterbarkeit und Möglichkeit der Verknüpfung neuer und alter Elemente untereinander. [33]

2.4.1 Ontologien

Wichtig ist dabei vor allem der Begriff der Ontologie. Eine Ontologie stellt in der Informatik ein Konzept zur formalen (d.h. maschinenlesbaren) Beschreibung von Wissen dar. [4] Anwendungen und Services arbeiten eher selten mit einer allgemeinen oder gar der selben Sprache oder Namensstruktur. Ontologien werden dazu verwendet um diese Lücke der semantischen Differenz zwischen Anwendungen und ihren Inhalten und Sprachkonzepten zu schließen. [35] Ontologien stellen eine Vielzahl von Möglichkeiten bereit, um Wissen zu verwalten [36]:

- Das gleiche Verständnis von Wissen wird durch die Verwendung einer identischen Ontologie an unterschiedlichen Orten erreicht.
- Durch Wiederverwendung bestehender Ontologien können neue Ontologien erschaffen werden.
- Wissen kann durch seine formale Beschreibung analysiert werden.
- Durch Trennung von Programmcode und Wissen ist es möglich das Wissen zu ergänzen ohne das Programm abzuwandeln.

Eine Ontologie ist z.B. in der Lage zu beschreiben, dass Java eine Programmiersprache ist. Liest eine Maschine nun semantische Informationen einer Webseite über Java, kann sie erkennen, dass es dabei um Programmiersprachen geht.

[4] In der Philosophie, dem Ursprung des Begriffes der Ontologie, steht es für die „Lehre des Seins“. [34]

2.4.2 OWL (Web Ontology Language)

Die Web Ontology Language (OWL) gehört zu den Konzepten W3C-Konzepten des Semanitc Web. Die OWL ist eine semantische Auszeichnungssprache um Ontologien im World Wide Web zu schaffen und zu verbreiten. [37] Ziel der OWL ist es, die Mächtigkeit und Möglichkeiten von XML (Extensible Markup Language) und RDF zu erweitern. Dadurch ist es möglich, hierarchische Zusammenhänge - wie sie in Netzwerken durchaus vorkommen - zu beschreiben. Maschinen sind dann in der Lage diese Hierarchien zu lesen und zu „verstehen“. OWL bietet Methoden, um Klassen mit ihren Eigenschaften zu beschreiben. Kardinalitäten und Symmetrie-Eigenschaften sind dabei ebenso von Bedeutung wie Abhängigkeiten oder Unterklassen.

Klassen, Eigenschaften und Instanzen haben dabei unterschiedliche Bedeutungen. So kann eine Klasse Eigenschaften besitzen, wohingegen eine Instanz die Ausprägung (auch Individuum) einer Klasse ist. Eine OWL-Ontologie kann mehrere Klassen mit Eigenschaften und Individuen beschreiben. Um Ontologien zu beschreiben, bedient sich die OWL des RDF, welcher im folgenden Kapitel 2.4.3 genauer beschrieben wird. Um das o.g. Beispiel der Programmiersprache Java fortzuführen, ist nachstehend eine OWL-Ontologie für Java formuliert:

```
1  <?xml version="1.0"?>
2  <rdf:RDF
3      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4      xmlns:owl="http://www.w3.org/2002/07/owl#"
5      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
6    xml:base="http://www.owl-ontologies.com/unnamed.owl">
7    <owl:Class rdf:ID="Programmiersprache"/>
8    <owl:Class rdf:ID="Java">
9      <rdfs:subClassOf rdf:resource="#Programmiersprache"/>
10   </owl:Class>
11   <owl:Class rdf:ID="Person"/>
12   <owl:Class rdf:ID="Programmierer">
13     <rdfs:subClassOf rdf:resource="#Person"/>
14   </owl:Class>
15   <owl:ObjectProperty rdf:ID="hatProgrammierer">
16     <rdfs:domain rdf:resource="#Java"/>
17     <rdfs:range rdf:resource="#Programmierer"/>
18   </owl:ObjectProperty>
19 </rdf:RDF>
```

Gut zu sehen ist hier die Zusammensetzung einer OWL-Ontologie. Sie besteht aus drei RDF-Tripeln welche die zugehörigen Zusammenhänge beschreiben. Das Beispiel beschreibt die Klasse `Programmiersprachen` als Oberklasse der Klasse `Java`. Diese `Java`-Klasse hat die Eigenschaft Programmierer zu besitzen. Die Eigenschaft `HatProgrammierer` ist Typ `Progammierer` und gilt für die Klasse `Java`. Die Klasse `Programmierer` ist ihrerseits eine Unterklasse von `Person` (hier nicht dargestellt). Weitere Programmiersprachen wie C++, PROLOG, Pascal, usw., könnten die Ontologie zusätzlich um ihre Eigenschaften und Konzepte ergänzen.

2.4.3 RDF (Resource Description Framework)

Mit Hilfe der Universal-Sprache RDF lassen sich Informationen über Ressourcen (z.B. im World Wide Web) darstellen. [38] Dabei benutzt sie Tripel der Form Subjekt, Prädikat, Objekt - also Sätzen einiger gesprochenen Sprachen (Englisch, Deutsch, etc.) durchaus ähnlich. Ein grafisches Beispiel findet sich in Abbildung 2.11, in RDF im unten aufgeführten Listing.

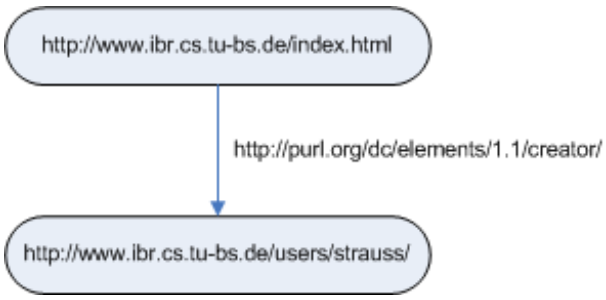


Abbildung 2.11: RDF Beschreibung [9], in Anlehnung an [10]

Ihre Syntax orientiert sich dabei an der der XML. Somit ist eine Maschinenlesbarkeit gewährleistet. Die spezielle Form der XML für RDF wird auch RDF/XML genannt. [38] Um diese Maschinenlesbarkeit von Subjekt, Prädikat und Objekt zur Verfügung stellen zu können, benutzt RDF Uniform Resource Identifier (URI). Die optionale *Fragment Identifier-Komponente* einer URI erlaubt eine indirekte Identifizierung der Quelle, die angibt welche Information beschrieben wird. Diese Quelle kann dabei unter anderem ein Teil oder eine Untermenge einer Hauptquelle sein. [39] In RDF werden Elemente, die durch eine URI-Referenz identifiziert werden können, als Ressource definiert.
Beispiel eines RDF-Tripels sei hier die Webseite des Instituts für Betriebssysteme und Netze mit ihrem „Autor“ Frank Strauß: (in Anlehnung an [10])

```
1 http://www.ibr.cs.tu-bs.de/index.html has a creator
2   whose value is Frank Strauß
```

In RDF würde diese Information, äquivalent zur Abbildung 2.11, wie folgt aussehen:

```
1 <?xml version="1.0"?>
2 <rdf:RDF
3   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4   xmlns:dc="http://purl.org/dc/elements/1.1/creator/">
5   <rdf:Description about="http://www.ibr.cs.tu-bs.de/index.html">
6     <dc:Creator>http://www.ibr.cs.tu-bs.de/users/strauss/</s:Creator>
7   </rdf:Description>
8 </rdf:RDF>
```

Definiert man eine Ontologie in RDF, so definiert man ein RDF Schema (RDFS). Dieses Schema definiert alle Bedingungen und Beziehungen des speziellen Anwendungsfalls. RDF Schema ist also eine Erweiterung von RDF um Definitionen von bestimmten, speziellen Typen und Ressourcen. Dadurch können z.B. auch Vererbungsbeziehungen bzw. Unterklassen (Vgl. `rdfs:subClasOf` im OWL-Listing)und Ein-

schränkungen (`rdfs:range`) oder sogar neue Domains (`rdfs:domain`) geschaffen werden (siehe Kapitel 2.4.2). Daher dient RDFS dazu eine Art „Dialekt“ für die eigene Ontologie zu schaffen. Somit ist gewährleistet, dass das RDF Schema korrekt interpretiert wird. [40]

2.4.4 SPARQL (SPARQL Protocol And RDF Query Language)

SPARQL ist eine Anfragesprache für RDF. Der Begriff SPARQL selbst ist ein rekursives Akronym. Er steht für ***S***PARQL ***P***rotocol ***A***nd ***R***DF ***Q***uery ***L***anguage. SPARQL ist, ebenso wie RDF, ein W3C-Standard und wird demnach auch vom W3C weiterentwickelt. SPARQL kann dazu verwendet werden, um Anfragen über diverse Datenquellen zu stellen. Dabei ist es unerheblich, ob die Daten im nativen RDF-Format vorliegen, oder die Daten in einem RDF-konformen Modell gespeichert sind. Die Anfragesprache beinhaltet demnach auch Methoden für Anfragen an einen gerichteten OWL-RDF-Graphen. SPARQL ist deshalb in der Lage sowohl UND-Verknüpfungen als auch ODER-Verknüpfungen zu interpretieren. Die Ergebnisse einer SPARQL-Anfrage können entweder als Ergebnismengen oder als RDF-Graphen ausgegeben werden. Aktuell liegt SPARQL als Proposed Recommendation vor.[5] [41]

Eine Beispielanfrage an das oben definierte RDF Schema der Programmiersprache Java könnte wie folgt aussehen:

```
1 PREFIX Programmiersprache: <http://example.com/exampleOntology#>
2 SELECT ?x ?y
3 WHERE {
4   ?x Programmiersprache:Java ?y.
5   ?y Programmiersprache:hatProgrammierer ?y.
6       }
```

Anfragen werden gemäß der RDF-Definition in Tripeln formuliert. Eine Anfrage versucht eine Übereinstimmung mit diesem Tripel zu finden. Prädikat und Objekt der Anfrage sind feste Werte. Das Subjekt ist eine Variable ohne jegliche Einschränkungen. Im o.g. Beispiel stellen x und y die Lösungsvariaben dar. Die Anfrage soll alle Werte x und y ausgeben, für die die Programmiersprache Java existiert und diejenigen Java-Werte, die Programmierer besitzen. (Vgl. [42])

2.5 Ponder2

Ponder ist eine Policy-Spezifikations-Sprache, die vom Imperial College London entwickelt wurde. Sie bietet Werkzeuge und Dienste, um Policies zu spezifizieren, zu analysieren und zu vollstrecken. *Ponder2* ist eine signifikante Weiterentwicklung der Ponder-Spezifikation mit neu entworfenem Framework.[43] Während Ponder noch für Netzwerk- und Systemmanagement ausgelegt war, wurde Ponder2 als vollständig erweiterbarer Framework entwickelt, der auf verschiedenen Abstraktionsebenen agieren kann. Ponder2 ist ein universelles Objekt-Management-System zur Verwaltung von

[5] Eine Proposed Recommendation ist ein voll entwickelter technischer Report, der nach gründlicher Überprüfung zu einem Standard vorgeschlagen werden könnte.

Managed Objects.[44]
Managed Objects sind Entitäten in Ponder2, die in der Lage sind, Nachrichten in der Ponder2-eigenen Sprache *PonderTalk*[6] zu verarbeiten. Ein Managed Object wird dabei typischerweise in Java formuliert. Um eine Verbindung zwischen PonderTalk-Nachrichten und Java-Methoden herzustellen werden Java-Annotationen verwendet. [45]

2.5.1 Elemente des Ponder2-Frameworks

Der Ponder2-Framework besteht aus folgenden Einheiten und Diensten:

- Einem *Domain Service*.
  Domains bieten Mittel, um Managed Objects für Management-Zwecke hierarchisch zu gruppieren. Eine Gruppierung erfolgt explizit, d.h. Objekte müssen explizit in eine Domain eingefügt, bzw. aus einer Domain entfernt werden. Domains sind ebenfalls Managed Objects. Domains enthalten jedoch nur Verweise auf die in ihnen enthaltenen Managed Objects, nicht die Objekte selbst. Ein Managed Object kann allerdings in verschiedenen Domains enthalten sein. [46]

- Einem *Obligation Policy Interpreter*.
  Wird das Auftreten eines Events an Ponder2 gemeldet, gleicht der Obligation Policy Interpreter diese Benachrichtigung gegenüber den registrierten Policy-Events ab. Er wertet die Condition der Policy aus und löst, bei erfolgreicher Auswertung, die zugehörige Policy-Action aus. Der Obligation Policy Interpreter verarbeitet dabei ECA-Policies (Event, Condition, Action) [47]

- Einem *Command Interpreter*.
  Der Command Interpreter wertet die XML-Kommandos aus, die Ponder2 erhält. Dies können Befehle wie `<add name=\newobject\>...</add>` sein, oder der Befehl, ein bestimmtes Objekt oder eine Anweisung zu benutzen (hier `<use>...</use>`). Die Befehle für den Ponder2 Framework werden dabei in *PonderTalk* formuliert[48]

- Dem *Authorisation Enforcement*.
  Der Authorisation Framework von Ponder2 unterstützt Authorisation Policies bei der Verwaltung und Kontrolle der Interaktionen zwischen Managed Objects. Im Ponder2 Authorisation Framework (PAF) können Authorisation Policies einheitlich spezifiziert und durchgesetzt werden. Sowohl Subjekt als auch das Zielobjekt der Action werden dabei geschützt. Abbildung 2.12 zeigt die vier Policy Enforcement Points des PAF. PEP 1 und PEP 2 werden zur Durchsetzung von Authorisation Policies auf Subjekt-Seite verwendet. PEP 3 und 4 dagegen sorgen für Durchsetzung von Authorisation Policies auf der Zielseite. [11]

[6] PonderTalk basiert auf der Programmiersprache Smalltalk und wird zur Interaktion mit der Ponder2 System Management Konsole verwendet

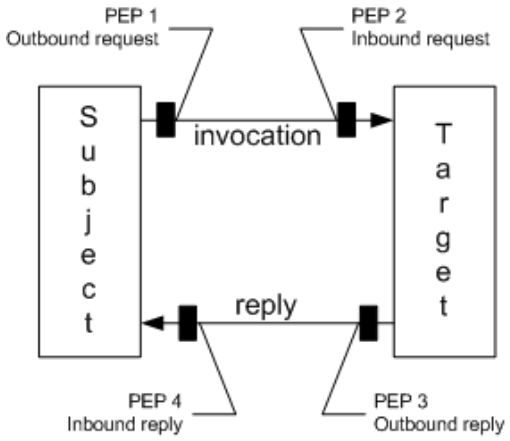


Abbildung 2.12: Policy Enforcement im Ponder Authorisation Framework (PAF), [11]

2.5.2 Policy-Enforcement im Ponder2-Framework

Durch die Durchsetzung von Policies am PEP1 der Abbildung 2.12, ist es möglich Authorisation Policies zu spezifizieren, welche die Subjekte daran hindern Aktionen auszuführen, die ihnen selbst oder ihren Domains schaden würden. Beispielswiese wird verhindert, dass ein Web Browser eine Anfrage an einen Webserver sendet, der sich auf der Ausschlussliste befindet. Weiterhin kann eine Durchsetzung an PEP4 verhindern dass ein Subjekt eine Antwort einer Action akzeptiert, die seine Integrität gefährdet.

Auf der Zielseite kann PEP2 dazu verwendet werden, um Authorisation Policies für traditionelle Zugangskontrollen durchzusetzen. Werden Authorisation Policies an PEP3 durchgesetzt, gibt es zusätzlich die Möglichkeit, die Daten des Ziels zu schützen. Eine Kompromittierung der Daten durch Preisgabe von Informationen einer Action, die nicht hätten veröffentlicht werden sollen, wird so verhindert.

Der o.g. PAF unterstützt sowohl negative als auch positive Authorisation Policies. In Kapitel 2.2.1 wurden Probleme durch verschiedene Policies mit gleichem Ausführungsergebnis vorgestellt. Um solche Konflikte zu verhindert bietet der PAF eine Konfliktlösungsstrategie, basierend auf der Dringlichkeit/dem Vorrang von Domains, die bereits während der Laufzeit Konflikte behandelt. [11]

2.5.3 Definition einer Ponder-Policy

Policies bilden die Event-Condition-Action-Regeln in Ponder2. Eine Policy wird in XML geschrieben und beschreibt den Eventtyp, auf den die Policy reagiert, und die Argumente, die das Event benutzt. Ebenso werden optionale Bedingungen, die erfüllt sein müssen (Conditions), und die Aktion(en), die ausgeführt werden sollen, definiert. Eine Policy mit dem Namen „`testpolicy`“, die auf das Event „`testevent`“ reagiert würde im Ponder-Policy-Format wie folgt aussehen:

```
1 <use name="/policy">
2   <add name="testpolicy">
3     <use name="/template/policy">
```

```
4       <create type="obligation" event="/event/testevent" active="true
            ">
5         <arg name="colour"/>
6         <arg name="intensity"/>
7         <condition>
8           <and>
9             <eq>!colour;<!-- -->red</eq>
10            <gt>!intensity;<!-- -->34</gt>
11          </and>
12        </condition>
13        <action>
14          <use name="/dom1/counter">
15            <inc/>
16          </use>
17        </action>
18      </create>
19    </use>
20  </add>
21 </use>
```

Der Typ „obligation“ zeigt an dass eine ECA-Policy folgt. Sobald die Policy erstellt ist, wird sie aktiv, was durch das Argument active=“true“ sicher gestellt wird. Die Policy benutzt die Argumente „colour und“ „intensity“, die von der Condition bereit gestellt werden. Die Condition-Klausel ist optional und beinhaltet einfache boolesche Ausdrücke zum Vergleich von String- und Integer-Werten. Im o.g. Beispiel wird überprüft ob die Farbe rot ist und ihre Intensität 34 entspricht. Jegliche Kombination von *und*, *oder*, *nicht*, *gleich*, *größer*, *größer-gleich*, *kleiner* oder *kleiner-gleich* ist dabei denkbar. Der eingefügte Kommentar <!-- --> stellt sicher, dass die Argumente und die zu überprüfenden Ausprägungen getrennt bleiben. Die abschließende Action fügt die Policy in die Domäne „counter“ ein. Das Action-Element ist bei einer Ponder-Policy zwingend erforderlich. Eine Ponder2-Action wird in Ponder2-XML geschrieben. Ihre Ausführung erfolgt erst, wenn Event und Condition der Policy erfüllt sind.[49]

2.5.4 Event-Types in Ponder2

Damit der Ponder2-Framework Events identifizieren und zuordnen kann, muss zuvor ein Event-Template entsprechend des auftretenden Ereignisses mit Hilfe der EventFactory erstellt werden. Ein Event-Template ist die tatsächliche Ausprägung eines Events zusammen mit seinen (optionalen) Argumenten. Instanzen eines Event-Templates werden erstellt und danach von den Policies aufgegriffen, die sich bei diesem speziellen Event-Template „angemeldet“ haben. [50]

Ein Event-Template für das Beispiel „testevent“ mit den Argumenten „colour“ und „intensity“ wird in PonderTalk wie folgt erstellt:

```
1 template := root/event/testevent create: #( "colour" "intensity" )
```

Das Senden eines „testevents“ mit PonderTalk an den Ponder2-Framework geschieht dann folgendermaßen:

```
1 testevent create: #( "red" 35 )
```

3 Anforderungsanalyse, Modellierung und Entwurf

Vor Betrachtung der Implementierung, werden im Folgenden zunächst einige Vorüberlegungen bezüglich Anforderungen, Architektur, Ablauf des Refinements und Formaten dargelegt.

3.1 Anforderungsanalyse

Das zu implementierende Programm soll unter bestimmten Voraussetzungen und unter gegebenen Bedingungen laufen, die nachfolgend dokumentiert werden.

3.1.1 Voraussetzungen bzw. gegebenes Szenario

Gegeben sei ein Netzwerk dessen Topologie bekannt ist. Die verbindenden Leitungen zwischen den Routern und Clients, sowie sonstigen Netzwerkkomponenten sind vorhanden. Was fehlt ist die Konfiguration des Netzes, bzw. Regeln, die das Verhalten in bestimmten Situationen steuern. Die Topologie soll eine Netzwerk-Ontologie im RDF-Format sein, die mit OWL erstellt wurde. Da PBNM sicher auch für den Business-Bereich interessant ist, wird das für die Implementierung relevante Netz dem einer kleinen Firma nachempfunden.

Das zu implementierende Programm soll nicht verteilt agieren, sondern auf einem einzelnen Rechner laufen. Daher fallen Netzwerkverbindungen über LDAP, HTTP, usw. hier nicht ins Gewicht. Die Architektur des Programms (Policy-Editor) soll sich teilweise an die von der IETF existierende Vorgabe einer Standard-Architektur eines Policy-Moduls anlehnen:

Der Policy-Editor kann als Policy Management Tool (PMT) verstanden werden. Ein Policy Decision Point (PDP) ist die Stelle innerhalb des PMT, an der entschieden wird, wie mit einer Policy verfahren wird und wo auch das Refinement stattfindet. Der Policy Enforcement Point schließlich ist der Punkt, an dem die Policy-Konfigurationen tatsächlich umgesetzt werden. Im hier vorliegenden Fall werden das in Abbildung 3.1 genannte Policy Management Tool (PMT) sowie der Policy Decision Point (PDP) nicht so groß werden, um sie als einzelne Module implementieren zu können. Eine „Verschmelzung“ zu einem Modul ist hier durchaus denkbar. Da der Policy Enforcement Point (PEP) „außerhalb“ des Refinements liegt, d.h. seine Arbeit erst beginnt, wenn das Refinement abgeschlossen ist und die tatsächlichen Policies umgesetzt werden sollen, ist er für die vorliegende Arbeit nicht von Bedeutung.

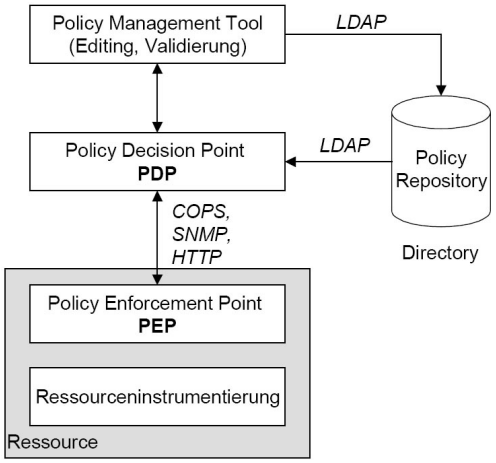


Abbildung 3.1: Die IETF-Policy-Architektur (Vgl.[12])

3.1.2 Ziel der Implementierung

Der Policy-Editor soll aus zwei Eingaben (Netztopologie und Policies) eine Ausgabe produzieren, welche die Konfiguration in maschinenlesbarer Form zur weiteren Verarbeitung enthält. Darüber hinaus sollte es zumindest einen Hinweis für den Benutzer im Falle eines Policy-Konfliktes, wie in Kapitel 2.2.1 angesprochen, geben. Eingegebene Policies sollen gespeichert werden können. Auch ein nachträgliches Bearbeiten einer bereits eingegebenen Policy soll ebenso möglich sein, wie das Löschen vorhandener Policies. Sind abgespeicherte Policies einer früheren Session in einer Datei vorhanden, sollen diese eingelesen werden können, um mit deren Bearbeitung und Vervollständigung fortfahren zu können. Weiterhin soll es dem Benutzer möglich sein das Programm seinen Bedürfnissen anzupassen. Durch einfaches Hinzufügen von Reizworten in den Katalog (siehe Kapitel 3.2.2) kann das Programm erweitert bzw. ergänzt werden. So können Spezialfälle, die nur im zu bearbeitenden Netzwerk vorkommen können, oder Netzwerkkomponenten die spezifisch für das Netzwerk sind, vom Programm verarbeitet werden.

3.2 Entwurf und Modellierung

Zu klären bleibt, wie die Anforderungen und Ziele durch den Policy-Editor umgesetzt werden können. Der erste grobe Entwurf dazu ist in Abbildung 3.2 für den Ablauf, sowie Abbildung 3.3 für ein Beispiel-UML-Diagramm dargestellt: Demnach soll der Policy-Editor die Topologie als Eingabe bekommen. Gibt ein Benutzer nun eine Policy ein, soll der Editor aus dem Reizwortkatalog ein Schlagwort zur Identifikation von Geräten bzw. Bedingungen suchen. Das Refinement erfolgt dann mit Hilfe dieser Reizworte. In Kapitel 3.2.2 wird dieser Prozess detaillierter beschrieben. Durch Kombination der Informationen aus der Topologie und den aus den Katalog gewonnenen Zuordnungen soll schließlich eine Policy entstehen. Um diese für Maschinen lesbar zu machen, wird die entstandene Policy in das maschinenlesbare Format eine Ponder-

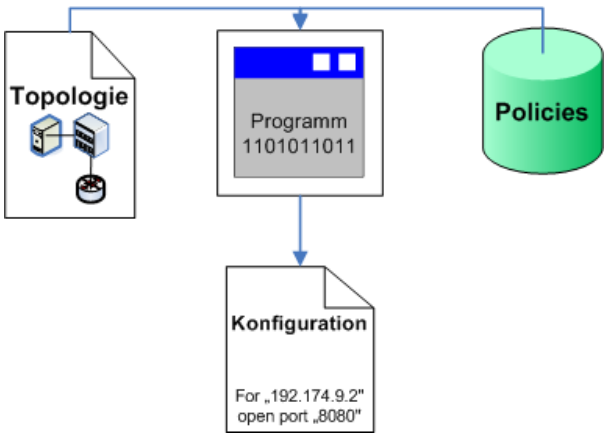


Abbildung 3.2: Grobe Darstellung der Implementierung

Policy (siehe in den Grundlagen 2.5, sowie hier in Kapitel 3.2.1) transformiert.
Aus diesem groben Ablauf ergibt sich nun ein Entwurf in UML, der sich wie in Abbildung 3.3 darstellt:

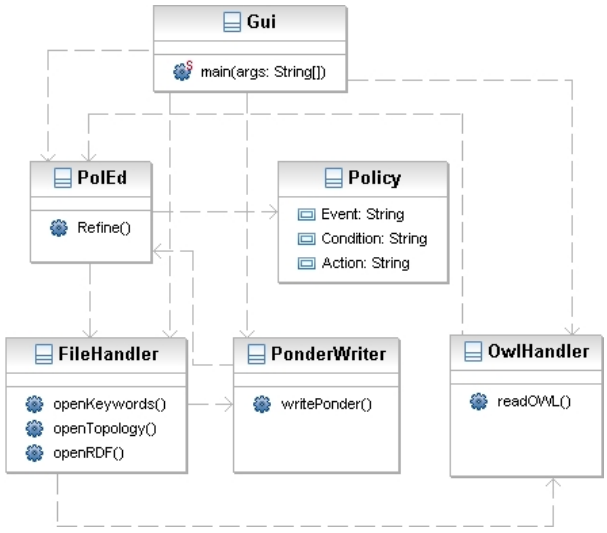


Abbildung 3.3: Grobentwurf in UML

3.2.1 Datenformate

Die externen Datenformate werden mit Hilfe von Bibliotheken bereitgestellt bzw. durch Transformation erreicht. Einerseits dient die Netzwerktopologie als Eingabe. Wie bereits in Kapitel 1.1 angesprochen, bietet das Semantic Web mit dem Konzept der Web Ontology Language (OWL) (Kapitel 2.4.2) hier gute Möglichkeiten, die in der folgenden Implementierung nutzbar (Kapitel 3.2.1) sind. So lässt sich die Topologie mit Hilfe der Jena-Bibliothek in einen Datentyp „Model“ einlesen, der einen gezielten Zugriff,

so wie Anfragen in der Anfragesprache SPARQL, zulässt. Somit ist es möglich Policies mit tatsächlichen Elementen des zu modellierenden bzw. zu verwaltenden Netzes zu verknüpfen und speziell darauf anzupassen. Zum anderen sollen dem Programm die oben angesprochenen Policies zur Konfiguration des Netzes eingegeben werden. Diese Eingabe wird als einfacher String erfolgen. Der Policy-Editor soll daraus schließlich eine maschinenlesbare Ausgabe formulieren.

Um diese Ausgabe später möglichst einfach und universell weiter verarbeiten zu können, soll sich der Policy-Editor den Konstrukten des Ponder-Konzepts bedienen. Die verfeinerten Policies werden also nach dem Refinement in Ponder-Policies transformiert und in eine Datei geschrieben.

Jena

Mit Hilfe der Jena-Bibliothek ist es möglich, eine vorhandene OWL-Ontolgoie (Kapitel 2.4.2) einzulesen und diese zu verarbeiten. Jena ist ein Open-Source Semantic-Web Framework der Hewlett-Packard Labs Semantic Web Research. [51] Jena stellt umfangreiche Klassen und Methoden für RDF, RDFS und OWL, sowie SPARQL zur Verfügung und bietet darüber hinaus eine regelbasierte Ableitungs-Engine. Wie in Abbildung 3.4 zu sehen, findet sich in der Jena-Bibliothek auch die `ModelFactory`, welche für die Erstellung von RDF-Modellen benutzt wird. Die OWL-Netztopologie lässt sich also mit Hilfe des Jena-Frameworks einlesen und in einem Modell abspeichern. Auch Anfragen an dieses Modell können mit Methoden der Jena-Bibliothek mittels der Anfragesprache SPARQL (Kapitel 2.4.4) formuliert werden. Diese Konstrukte und Methoden werden in der Implementierung ihre Anwendung finden.

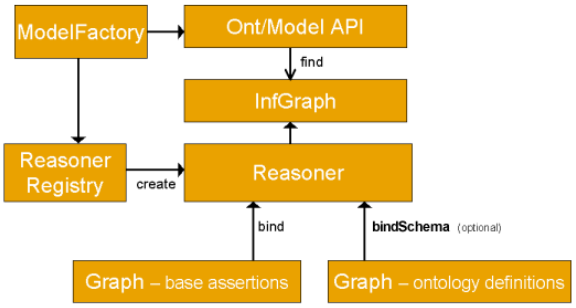


Abbildung 3.4: Aufbau der Jena-Ableitungs-Engine

Ponder2

Da auch das Ausgabeformat eindeutig gestaltet sein muss, wird die von Ponder2 in Kapitel 2.5 definierte Form einer Ponder-Policy genutzt. Somit können die vom Policy-Editor verfeinerten und transformierten Policies später vom Ponder2-Framework weiter verarbeitet werden. Der in Abbildung 3.1 gezeigte Teil des Policy Enforcement

Points könnte somit von Ponder2 übernommen werden. Eine zusätzliche Implementierung eines Management-Systems ist also demnach erforderlich.[1]
Die Nutzung von Ponder2 und Transformation der verfeinerten Policies in Ponder-Policies hat einen weiteren Vorteil. Ein Vergleich des ursprünglichen Sprachformates mit dem der Policy-Srache ist obsolet. Durch Ponder2 wird sichergestellt, dass die gesprochene Sprache in maschinenlesbare Form codiert wird. Die benutzte Policy-Sprache drückt damit exakt die Konzepte und gewollten Befehle aus, die die Eingabe beabsichtigte. [53]

3.2.2 Refinement durch Reizworte

Bereits in Kapitel 2.2 wurden einige Methoden zum Refinement vorgestellt. Ähnlich einem zielbasierten Ansatz, soll hier vorgegangen werden. Der Refinement Algorithmus des Policy-Editors führt ein zweistufiges Refinement über drei Ebenen des Policy-Kontinuums durch. Da in Kapitel 3.1.1 bereits Einschränkungen bezüglich des Einsatzortes sowie des Inputs definiert wurden, werden beim Refinement nicht alle Ebenen berücksichtigt. So erfolgt die Verfeinerung nur über die Ebenen „Business View“, „Device View“ und „Instance View“. Grafisch lässt sich der Refinement-Prozess anhand des Policy-Kontinuums wie in Abbildung 3.5 darstellen. Demnach spielen System- und Netzwerk-Ebene aufgrund der Vorbedingungen hier eine untergeordnete Rolle und werden nicht beachtet.

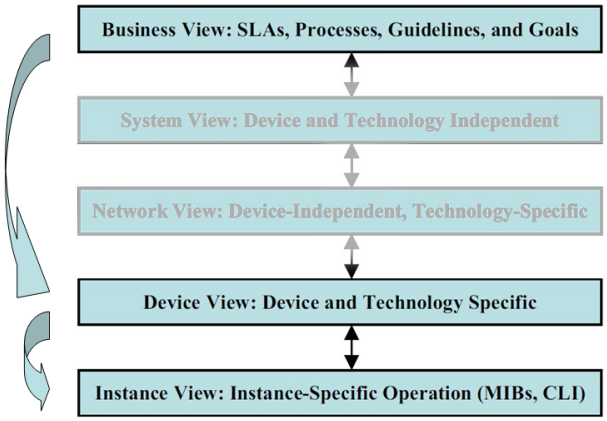


Abbildung 3.5: Zweistufiges Refinement, in Anlehnung an das Policy-Kontinuum

Die einzugebenden Policies liegen zumeist nicht in maschinenlesbarer Form vor. Wie auch auf Abbildung 2.7 des Policy-Kontinuums zu erkennen, kann die obere Stufe noch SLAs, Zielvorgaben, etc. enthalten, die üblicherweise in „normaler“ Sprache vorliegen. Um diese Sprache nun in Maschinensprache übersetzen zu können, bedarf es eines Konstrukts, welches dem Programm ermöglicht, Eingaben eindeutig zuzuweisen.

Es muss also eine Art von Wissensrepräsentation und -definition vorliegen, um die Eingaben für Maschinen lesbar zu machen. Hierfür soll das Konzept des Reizwortkatalo-

[1] Ein Tutorial zur Verwendung des Ponder-Frameworks findet sich in [52]

ges bzw. einer Keyword-Datei dienen. Der Katalog beinhaltet zeilenweise jeweils ein Tupel bestehend aus Reizwort und daraus resultierender Übersetzung für die Maschine. Abbildung 3.6 zeigt beispielhaft einen solchen Katalog. Somit wird sichergestellt, dass der Policy-Editor die Eingaben korrekt interpretiert.

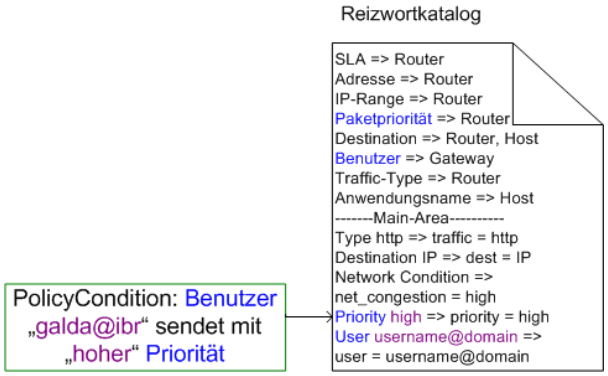


Abbildung 3.6: Beispiel eines Reizwortkataloges

Der tatsächliche Aufbau des Reizwortkataloges wurde für die Implementierung in zwei Bereiche unterteilt. Im oberen ersten Bereich befinden sich die Schlagworte mit deren Hilfe die Device-Ebene des Kontinuums (Kapitel 2.7) erreicht werden soll. Hier sollen ausschließlich Worte für die spätere Identifikation von zugehörigen Netzwerkgeräten zu finden sein. Als Trennsymbole zwischen oberem und unterem Bereich wird (mindestens) ein Minus-Zeichen verwendet. Der untere Bereich enthält nun diejenigen Schlagworte, die später in eine Konfiguration überführt werden sollen. Somit soll die untere Instance-Ebene erreicht werden. Ein Beispiel-Reizwortkatalog, der für den Policy-Editor verwendbar wäre, findet sich im Handbuch B.1.

3.2.3 Ablauf des Refinements

Um von der obersten (high-level) Ebene auf die unterste (low-level) zu gelangen, wird vom Policy-Editor ein 2-stufiges Refinement über 3 Ebenen durchgeführt. Da System- und Netzwerk-Ebene an dieser Stelle bekannt und vernachlässigbar sind, verläuft das Refinement somit über die Ebenen Business View, Device View und Instance View. Somit bedient sich der Algorithmus des Refinements mit Hilfe von Reizworten sowohl Elementen des zielbasierten Ansatzes (Kapitel 2.2.2), als auch Elementen des Refinements durch fallbasierte Schlussfolgerungen (Kapitel 2.2.4).

Ablauf des Refinements:

1. Eingabe einer Policy in einem bestimmten Format (getrennt durch Komma, Hochkomma und Dollarsymbol) (siehe auch im Handbuch B.2 und im folgenden Implementierungskapitel 4.2.3) (Business View). Eine High-Level-Policy wird als String in das Programm eingegeben. Dabei werden die Elemente (Event, Condition, Action) durch Komma getrennt. Operatoren werden mit dem Dollar-Symbol umschlossen und Parameter mit einfachen Hochkommata, um so eine eindeutige Identifizierung zu ermöglichen.

2. Extrahieren des Events, der Condition und der Action. Der Policy-String wird in seine Elemente unterteilt. Der Benutzer erhält so die Möglichkeit einer visuellen Prüfung der Richtigkeit seiner Eingabe.
3. Durchsuchen des Events nach zugehörigem Gerät durch Abgleich der einzelnen Worte im Reizwortkatalog (1. Schritt des Refinements → Device View).
4. Bei Fund: Angabe und Abspeicherung des gefundenen Gerätes. Wird ein Gerät gefunden, wird der Benutzer informiert und der Geräte-Oberbegriff (z.B. Router, Client, Gateway, etc.) zusammen mit der Policy abgespeichert.
5. Darstellung der bisher gesammelten Daten in einer Tabelle. Jede Policy wird nach ihrer Eingabe und dem ersten Refinement-Schritt in einer Tabelle dargestellt. Der Benutzer hat somit die Möglichkeit einzelne Policies nachträglich zu ändern oder sogar ganz zu löschen.
6. Sind alle Eingaben gemacht, bzw. alle Policies in der Tabelle aufgelistet, erfolgt die Abspeicherung sämtlicher bereits eingegebener Policies in einer Datei. So kann später mit den Policies weiter gearbeitet werden.
7. Transformation jeder einzelnen Policy in eine Ponder-Policy. Dabei werden die bereits vorliegenden Informationen (Event, Condition, Action, Device) genutzt und in weiteren Schritten um Daten ergänzt:
   - Für jede eingegebene Policy: Extrahieren von Parametern und Operatoren aus der Condition. Wie oben beschrieben sind Operatoren und Parameter im normalen Text durch Sonderzeichen markiert. Die Werte innerhalb der Markierungen werden ausgelesen und in die Ponder-Policy übertragen.
   - Suche des relevanten Schlagwortes aus dem Reizwortkatalog. Der Hauptbereich des Reizwortkataloges wird nun nach Schlagworten durchsucht. Die zu erfolgende Aktion wird abgeleitet und die Daten in die Ponder-Policy übertragen. Somit erreicht das Refinement die Low-Level Ebene (Instance View)

Grafisch stellt sich der Ablauf des Refinements wie in Abbildung 3.7 dar. Eine detaillierte grafische Darstellung der darin stattfindenden Transformation in Ponder findet sich im Anhang A.4.

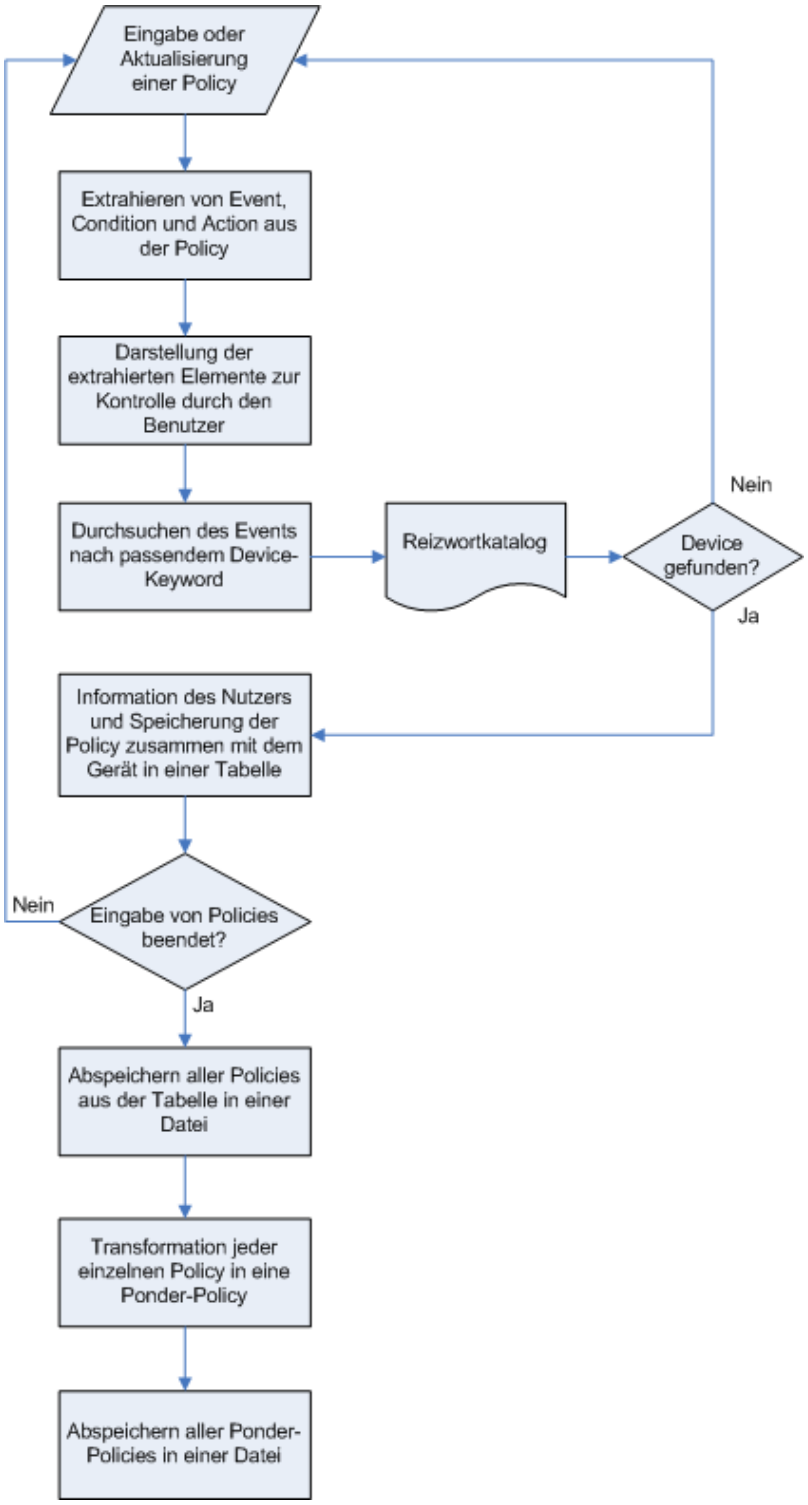


Abbildung 3.7: Ablauf des Refinement-Prozesses

4 Implementierung des Policy-Kontinuums

In diesem Kapitel wird die konkrete Implementierung der in Kapitel 3.2 diskutierten Konstrukte und Methoden erläutert. Besonderer Fokus liegt dabei auf den Methoden bezüglich des Refinements und der Transformation in Ponder.

4.1 Verwendete Technologien und Bibliotheken

Die Implementierung des Policy-Editors erfolgte in Java in der Version 1.6. Java wurde gewählt, da es in Zusammenhang mit der Jena-Klassenbibliothek (siehe unten) gute Möglichkeiten zur Verarbeitung von OWL-Ontologien bietet.

4.1.1 Protégé

Um OWL-Ontologien im RDF Format über die Netztopologie zu erstellen, wurde Protegé in der Version 3.3.1 (Build 430) verwandt (siehe Abbildung 4.1). Protégé ist ein grafischer OWL-Editor der Universität Manchester. Er ermöglicht die einfache Kreation von Ontologien im RDF Format nach dem Prinzip des „What you see is what you get“. Die in Protégé grafisch angelegten Ontologien werden im maschinenlesbaren RDF Format gespeichert.[54]

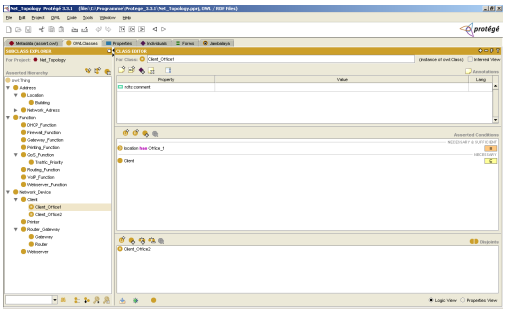


Abbildung 4.1: Protegé OWL-Editor

4.1.2 Jena

Jena ist eine Klassenbibliothek die Methoden und Konstrukte zur Verarbeitung von RDF-Dateien in Java bietet. Der Jena-Framework wird in der Version 2.5.4 verwendet.

Folgende Klassen der Jena-Bibliothek wurden bei der Implementierung verwendet:

- `jena.query.*` Stellt die Anfrage-Engine für SPARQL zur Verfügung.
- `jena.sparql.util.StringUtils` Wird bei einer Anfrage benötigt, um den Anfrage-String in SPARQL umzuwandeln.
- `jena.rdf.model.*` Beinhaltet Methodem zum Erstellen und Manipulieren von RDF-Modellen. Es wird daher zum Einlesen von OWL-Ontologien und zur Anfrage an RDF-Modelle verwendet.
- `jena.util.FileManager` Bietet die Möglichkeit RDF-Dateien in Modelle einzulesen.

4.1.3 Ponder-Policy

Um die spätere Verarbeitung eines Policy Enforcement Points der vom Policy-Editor verfeinerten Policies zu gewährleisten, wandelt der Editor die Policies am Ende des Refinements in das von Ponder2 (Kapitel 2.5) definierte Format einer Ponder-Policy um.

4.2 Implementierung des Policy-Editors

Die Implementierung orientiert sich an der in Kapitel 3.2 vorgestellten Architektur. Eine grafische Übersicht über die gesamte finale Programmarchitektur des implementierten Policy-Editors findet sich im Anhang A.1. Tabelle 4.1 zeigt die Pakete der Architektur des Policy-Editors.

PAKET	BESCHREIBUNG
PolEd.editor	enthält die Klassen für die Erstellung, Bearbeitung und Speicherung von Policies, sowie zur Dateiverarbeitung
PolEd.gui	enthält die grafische Oberfläche und die main-Methode zur Darstellung des Policy-Editors

Tabelle 4.1: Pakete des Policy-Editors

Die Programmklassen des Policy-Editors werden in Tabelle 4.2 dargestellt und kurz beschrieben.

PAKET.KLASSE	BESCHREIBUNG
editor.`Policy`	Beschreibt das interne Policy-Datenformat des Policy-Editors. Eine Policy besteht hier aus dem gesamten Policy-String, dem Event-, Condition- und Action-Teil der Policy und dem zugehörigen Netzwerkgerät(Device). Methoden zur Bestimmung und Extraktion der jeweiligen Teile sowie zur Stringumwandlung für späteres Schreiben in eine Datei werden hier bereit gestellt.
editor.`FileHandler`	Beinhaltet Methoden zum Schreiben von und Lesen aus Dateien zur Informationsgewinnung für das Programm. Liest evtl. vorhandene Policies aus einer Datei und durchsucht den Reizwortkatalog. Liest aus OWL-Dateien im RDF-Format. Schreibt verfeinerte Ponder-Policies in eine Datei.
editor.`OwlHandler`	Beinhaltet Methoden zur Verarbeitung von RDF-Dateien im OWL-Format. Liest Daten mit Hilfe des `FileHandlers` in ein OWL-Model und stellt Methoden zur Anfrage an das Modell zur Verfügung.
editor.`PolEd`	Stellt die Methoden zur Bearbeitung von Policies, Extraktion von Parametern sowie Operatoren, Aufteilung und Identifizierung von eingegebenen Policies in seine Bestandteile. Auch eine Konflikterkennung findet sich hier.
editor.`PonderWriter`	Hier befinden sich Methoden um teil-verfeinerte Policies in Ponder2-Policies zu transformieren und dabei einen weiteren Schritt des Refinements durchzuführen.
gui.`PolEdGui`	Die grafische Benutzeroberfläche des Policy-Editors.

Tabelle 4.2: Klassen des Policy-Editors

4.2.1 Der Policy-Datentyp

Bevor die Policies in Ponder-Policies transformiert werden, werden sie als eigener Policy-Datentyp gespeichert. Eine Policy enthält im Policy-Editor immer die Elemente:

1. Policy,
2. Event,
3. Condition,
4. Action,
5. Device.

Dies wird auch im Konstruktor der Klasse `Policy` deutlich:

```
1 public Policy(String policy, String event, String condtition, String
      action, String device) {
2     this.policy = policy;
3     this.event = event;
4     this.condition = condtition;
5     this.action = action;
```

```
6     this.device = device;
7   }
```

Neben get()- und set()-Methoden enthält der Policy-Datentyp noch eine `toString()`-Methode. Diese schreibt die fünf Elemente der Policy in einen einzelnen String und liefert diesen zurück.

```
1 @Override
2   public String toString() {
3     String policy_string = policy + " || " + event + " || " +
          condition
4         + " || " + action + " || " + device;
5     return policy_string;
6   }
```

Da sie die `toString()`-Methode der Superklasse (`Object`) überschreibt, ist sie mit `@Override` gekennzeichnet.

4.2.2 Der implementierte Refinement-Prozess

Wie bereits in Kapitel 3.2 beschrieben, führt der Policy-Editor ein zweistufiges Refinement durch. Die Methoden dafür verteilen sich auf die verschiedenen Klassen der Implementierung. Wichtig dabei sind:

- Die Extraktion von Event, Condition und Action aus den Benutzereingaben.
- Die Suche nach Schlagworten im Reizwortkatalog.
- Die Anfrage an die Topologie.
- Die Transformation in Ponder-Policies.

Die für diese Aktionen notwendigen Methoden werden in den folgenden Unterkapiteln genauer beschrieben.

4.2.3 Extraktion von Event, Condition und Action aus der Benutzereingabe

Die Eingabe einer Policy erfolgt in der grafischen Benutzeroberfläche des Policy-Editors im Segment des Editors. Hier bietet das Programm dem Benutzer eine Eingabemaske, wie in Abbildung 4.2, in die er eine High-Level-Policy in einem bestimmten Format eingeben kann. Zur eindeutigen Identifizierung der drei Klauseln einer Policy, werden die Teile durch Kommata getrennt. Parameter werden in einfache Hochkommata geschrieben. Sollen zwei zu überprüfende Bedingungen einer Condition mit einem Operator wie UND, ODER, etc. verknüpft werden, so muss dieser Operator in Dollar-Zeichen gesetzt werden.

Die Extraktionsmethode für eine eingegebene Policy wird von einem `ActionListener` des Knopfes „Check“ ausgelöst. Nach einer Abfrage ob genug Kommas in der Policy vorhanden sind bzw. das Eingabefeld nicht leer ist - somit werden falsche Benutzereingaben vermieden - wird der Text des Eingabefeldes in eine lokale Variable geschrieben.

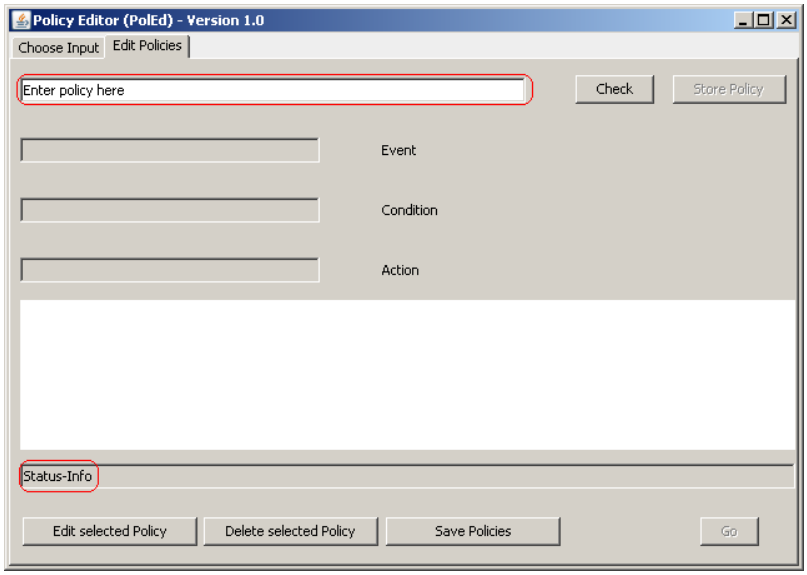


Abbildung 4.2: Eingabemaske des Policy-Editors

```
1 if (PolEd.commaCounter(PolicyField.getText()) < 2 || PolicyField.
      getText() == null) {
2         editorInfoField
3             .setText("Error in Policy String!! - Either empty or too
                  few commas");
4       }
5       else {
6         String local_pol = PolicyField.getText();
```

Anschliessend wird diese Variable durch Aufrufen von Extraktionsmethoden der Klasse `PolEd` in die einzelnen Klauseln zerlegt. Die Methoden benutzen dabei die Kommata in der Policy als Start- bzw. Endmarkierung.

```
1 String local_event = PolEd.extractEvent(local_pol);
2 String local_condition = PolEd.extractCondition(local_pol);
3 String local_action = PolEd.extractAction(local_pol);
```

Beispielhaft sei hier die Methode `extractEvent()` aufgeführt:

```
1 String event = null;
2     if (policy == null) {
3       throw new IllegalArgumentException(
4           "Policy is empty!! - Please insert a Policy");
5     }
6     int endofevent = policy.indexOf(',');
7     event = policy.substring(0, endofevent);
8     return event;
```

Ist die Policy leer, wird eine Exception ausgeworfen. Der Index des ersten Kommas wird als Endmarkierung für das Event gesetzt. Nun wird das Event als Teil-String vom Anfang bis zur Komma-Markierung extrahiert und ausgegeben. Zur Sicherheit wird noch die Anzahl der Hochkommata Modulo 2 gerechnet um zu gewährleisten, dass

bei der Eingabe keine Markierung vergessen wurde. Ist dies geschehen, werden die Ergebnisse wie in Abbildung 4.3 im Policy-Editor dargestellt.

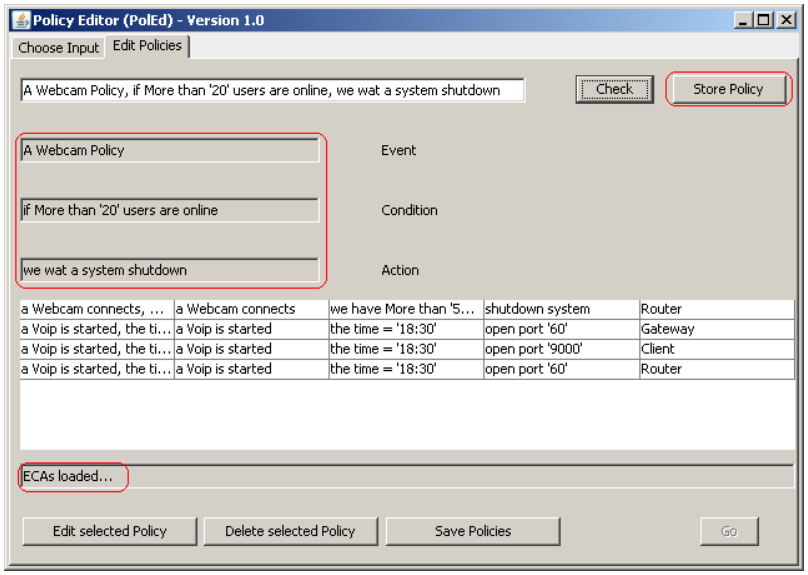


Abbildung 4.3: Extrahierte Klauseln einer eingegebenen Policy

4.2.4 Suche im Reizwortkatalog

Die Suche innerhalb des Reizwortkataloges beinhaltet - wie der Katalog selbst - zwei Bereiche: die Suche nach dem Gerät und nach einem Konfigurations-/Hauptwort. Beide Methoden werden im Folgenden beschrieben.

Suche nach dem Netzwerkgerät:

Der Algorithmus teilt den Eingabe-Satz in seine Einzelworte auf und vergleicht jedes Wort mit Hilfe der `FileHandler`-Methode `checkDeviceKeywords()` mit jedem Eintrag im Device-Bereich des Reizwortkataloges (Kapitel 3.6).

```
1     String[] eventsearch = part.split("\\s");
2     for (int i = 0; i < eventsearch.length; i++) {
3       if (FileHandler.checkDeviceKeywords(keyword_file, eventsearch[i
            ]) != null)
4         return FileHandler.checkDeviceKeywords(keyword_file,
5             eventsearch[i]);
```

Wurde ein Device-Keyword gefunden, liefert die Methode dies zurück und informiert den Benutzer über den Fund. Dieses Device wird im späteren Programmverlauf zusammen mit der zugehörigen Policy in eine Tabelle geschrieben.

Die checkDeviceKeywords()-Methode öffnet den angegebenen Reizwortkatalog und prüft jedes gefundene Wort zu Anfang einer Zeile mit dem übergebenen Reizwort.

Dabei wird die Länge des Reizwortes zugrunde gelegt, um die gesamte Zeile auf diese Länge zu kürzen. Sollte eine eingelesene Zeile kleiner sein, als das zu suchende Schlüsselwort, wird die Zeile übersprungen. Ist der Vergleich positiv, wird das entsprechend relevante Netzwerkgerät zurück gegeben und die Datei geschlossen.

```
1       keyword_row = br.readLine();
2       char first_char = keyword_row.charAt(0);
3       while (keyword_row != null && first_char != '-') {
4       if (keyword_row.length() < keyword.length()) {
5           keyword_row = br.readLine();
6         } else {
7         String check = keyword_row.substring(0, keyword.length());
8         if (check.equals(keyword)) {
9           return keyword_row.substring(keyword.length() + 1,
10              keyword_row.length());
11        }
12        keyword_row = br.readLine();
13        first_char = keyword_row.charAt(0);
14        }
15      }
16      br.close();
```

Webcam connects, Mo...	Webcam connects	More than '50' users o...	activate congestion by...	Router

Abbildung 4.4: Policy mit identifiziertem Netzwerkgerät in der Tabelle des Policy-Editors

Bei der Überprüfung wird die Länge des zu suchenden Wortes als Längenbegrenzung für das gefundene Wort genutzt. Die eingelesene Zeile des Kataloges wird auf diese Länge gekürzt. Somit passen nur Worte der selben Länge auf das zu überprüfende Wort. Stimmt der so gekürzte Zeilen-String mit dem Reizwort überein, wird das dahinter liegende Wort extrahiert und zurückgeliefert. Dieses Wort identifiziert die Geräteklasse, für die das Event Gültigkeit besitzt. Wurde ein Netzwerkgerät identifiziert, wird es zusammen mit der Policy in der Tabelle abgespeichert (Abbildung 4.4) und der Benutzer über den Fund informiert (Abbildung 4.5).

Abbildung 4.5: Information über gefundenes Netzwerkgerät

Suche nach Condition- bzw. Action-Reizwort

Ähnlich der Suche nach dem Netzwerkgerät durchsucht die Methode `getMainKeyword()` mit Hilfe des `FileHandlers` den Katalog nach Reizworten für Condition und Action. Der eingegebene Policy-Teil wird in seine Einzelworte zerlegt und anschließend an die Methode `checkMainKeywords()` übergeben.

```
1     String[] eventsearch = part.split("\\s");
2     for (int i = 0; i < eventsearch.length; i++) {
```

```
3       if (FileHandler.checkMainKeywords(keyword_file, eventsearch[i])
              != null)
4         return FileHandler.checkMainKeywords(keyword_file,
5             eventsearch[i]);
```

4.2.5 Anfragen an die Netztopologie

Während der Transformation in eine Ponder-Policy (Kapitel 4.2.6) wird mit Hilfe des `FileHandlers` die Topologie eingelesen und über den `OwlHandler` eine SPARQL-Anfrage an das Modell gestellt, um die im Netzwerk befindlichen entsprechenden Geräte auszulesen. Zunächst erfolgt eine Überprüfung, ob sich Elemente des Netzwerkgerätes in der Topologie befinden.

```
1     if (!containsDevice(model, device)) {
2       System.out.println(device + "not found!!");
3       return null;
4     }
```

Anschließend wird der Anfrage-String formuliert...

```
1     String queryString = StringUtils
2         .join(
3             "\n",
4             new String[] {
5                 "PREFIX pf: <http://www.owl-ontologies.com/
                      Ontology1196156810.owl#>",
6                 "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax
                      -ns#>",
7                 "SELECT ?subject ",
8                 "WHERE {" + " ?subject rdf:type pf:" + device
9                     + " . ", "}" });
```

...,mit Hilfe der Jena-Bibliotheksmethoden verarbeitet und die eigentliche Anfrage gestellt.

```
1     Query query = QueryFactory.create(queryString);
2     QueryExecution qexec = QueryExecutionFactory.create(query, model)
           ;
```

Danach wird die Ergebnismenge durchlaufen und jedes Element an einen String übergeben.

```
1     String device_name = null;
2     try {
3       ResultSet results = qexec.execSelect();
4       for (; results.hasNext();) {
5         QuerySolution soln = results.nextSolution();
6         String name = extractName(soln);
7         device_name = device_name + name;
8       }
9     }
```

Abschließend wird die Anfrage beendet und der Ergebnis-String zurück geliefert.

```
finally {
      qexec.close();
    }
    return device_name;
```

4.2.6 Transformation in Ponder-Policies

Die Transformationsmethode in eine Ponder-Policy ist vom Umfang her die größte der Implementierung. Sie holt sich während der Transformation Daten aus der Topologie mit Hilfe des `OwlHandlers`. So werden die tatsächlichen Individuen der Netzwerkkomponenten identifiziert und in der Policy verarbeitet. Weiterhin vollführt sie den zweiten Schritt des Refinements indem während der Transformation abermals der Reizwortkatalog, allerdings diesmal im Hauptbereich, nach Schlagworten durchsucht wird (wie in Kapitel 4.2.4 beschrieben).

Zuerst werden sämtliche Policies aus einem Policy-Vektor ausgelesen. Jede gefundene Policy wird dann an die Methode `generatePonderPolicy()` übergeben, die die tatsächliche Umwandlung durchführt.

```
    int list_length = 0;
    for (int i = 0; i < policy_vector.size(); i++) {
      if (policy_vector.get(i) != null) {
        list_length++;
      }
    }
    for (int i = 0; i < list_length; i++) {
      generatePonderPolicy(policy_vector.get(i), keyword_file, model)
          ;
    }
```

Die Transformation orientiert sich am Ponder-Policy-Schema (Kapitel 2.5) und erzeugt die entsprechenden Elemente in Verknüpfung mit den Policy-Daten und den im Reizwortkatalog gefundenen Informationen. Werden mehrere Individuals eines Netzwerkgerätetyps gefunden, so wird für jedes einzelne Gerät eine neue Policy formuliert. Zuletzt wird die Policy als String an den `FileHandler` übergeben und in eine Datei geschrieben.

Es erfolgt zunächst eine Kontrolle, ob sich das zu bearbeitende Netzwerkgerät in der Topologie befindet. Falls nicht, wird in Kommentarklammern eine Fehlermeldung angehängt.

```
    if (!OwlHandler.containsDevice(model, policy.getDevice())) {
      ponder_policy = ponder_policy + "<!-- " + policy.getDevice()
          + " not found!!-->";
```

Wie in Kapitel 2.5.4 beschrieben wurde, benötigt der Ponder2-Framework zur Identifikation eines Events einen entsprechenden Event-Type. Der für die zu erstellende Policy geltende Eventtype wird deshalb zu Anfang jeder Ponder-Policy als Kommentar hinterlegt. Dabei wird zunächst das Event nach einem verbindenden Operator durchsucht. Wird ein Operator gefunden, werden die beiden Bedingungen in ihre Bestandteile aufgeteilt und diese im Befehl der Template-Erstellung abgespeichert. Bei keinem Operator geschieht dies entsprechend nur für eine Bedingung.

```
1         if (PolEd.hasSecondOperator(policy.getEvent())) {
2           if (PolEd.extractParam(PolEd
3               .getFirstPart(policy.getEvent())) != null
4               && PolEd.getSecondPart(policy.getEvent()) != null) {
5             ponder_policy = ponder_policy
6                 + "<!-- template := root/event/"
7                 + PolEd.getDeviceInitWord(PolEd
8                     .getFirstPart(policy.getEvent()),
9                     keyword_file)
10                + " create: #( \""
11                + PolEd.getDeviceKeyword(PolEd
12                    .getFirstPart(policy.getEvent()),
13                    keyword_file)
14                + "\" "
15                + "\"value\" "
16                + "\""
17                + PolEd.getDeviceKeyword(PolEd
18                    .getSecondPart(policy.getEvent()),
19                    keyword_file) + "\" " + "\"value\""
20                + " )-->" + "\n";
21          } else if (PolEd.extractParam(PolEd.getFirstPart(policy
22              .getEvent())) != null) {
23            ponder_policy = ponder_policy
24                + "<!-- template := root/event/"
25                + PolEd.getDeviceInitWord(PolEd
26                    .getFirstPart(policy.getEvent()),
27                    keyword_file)
28                + " create: #( \""
29                + PolEd.getDeviceKeyword(PolEd
30                    .getFirstPart(policy.getEvent()),
31                    keyword_file)
32                + "\" "
33                + "\"value\" "
34                + "\""
35                + PolEd.getDeviceKeyword(PolEd
36                    .getSecondPart(policy.getEvent()),
37                    keyword_file) + "\" )-->" + "\n";
38          } else if (PolEd.extractParam(PolEd.getSecondPart(policy
39              .getEvent())) != null) {
40            ponder_policy = ponder_policy
41                + "<!-- template := root/event/"
42                + PolEd.getDeviceInitWord(PolEd
43                    .getFirstPart(policy.getEvent()),
44                    keyword_file)
45                + " create: #( \""
46                + PolEd.getDeviceKeyword(PolEd
47                    .getFirstPart(policy.getEvent()),
48                    keyword_file)
49                + "\" "
50                + "\""
51                + PolEd.getDeviceKeyword(PolEd
52                    .getSecondPart(policy.getEvent()),
53                    keyword_file) + "\" " + "\"value\""
54                + " )-->" + "\n";
55          }
56        }
57        else {
```

```
58            ponder_policy = ponder_policy
59                + "<!-- template := root/event/"
60                + PolEd.getDeviceInitWord(policy.getEvent(),
                      keyword_file)
61                + " create: #( \""
62                + PolEd.getDeviceKeyword(policy.getEvent(),
63                    keyword_file) + "\" " + "\"value\""
64                + " )-->" + "\n";
65          }
```

Eine Überprüfung der Anzahl gefundener Netzwerkgeräte schließt sich hieran an. Werden mehrere Geräte gefunden, wird für jedes einzelne eine Policy formuliert. Beim Fund nur eines Gerätes wird mit der Erzeugung des Ponder-Textes begonnen.

```
1       if (OwlHandler.getDeviceCount(model, policy.getDevice()) < 2) {
2         ponder_policy = ponder_policy + "<use name=\"/policy\">" + "\
              n";
```

Das zu bearbeitende Netzwerkgerät in Kombination mit einer Laufvariable wird als Name der Policy verwendet. Die Eventbezeichnung leitet sich aus dem Wort ab, welches das Netzwerkgerät identifiziert hat.

```
1         ponder_policy = ponder_policy + "<add name=\""
2             + OwlHandler.getDeviceName(model, policy.getDevice())
3             + "_policy_" + index + ">" + "\n";
4         ponder_policy = ponder_policy
5             + "<use name=\"/template/policy\">" + "\n";
6         ponder_policy = ponder_policy
7             + "<create type=\"obligation\" event=\"/event/"
8             + PolEd.getDeviceInitWord(policy.getEvent(),
9                 keyword_file) + " active=\"true\">" + "\n";
```

Nun wird getestet, ob der Condition-Teil der Policy eventuell zwei Überprüfungen enthält. Danach erfolgt die Suche nach dem Haupt-Reizwort im Reizwortkatalog - jeweils für den ersten bzw. zweiten Teil der Condition.

```
1         if (PolEd.hasSecondOperator(policy.getCondition())) {
2           if (PolEd.getMainKeyword(policy.getCondition(),
3               keyword_file) != null) {
4             String condition_arg = PolEd.getMainKeyword(policy
5                 .getCondition(), keyword_file);
6             ponder_policy = ponder_policy + "<arg name=\""
7                 + condition_arg + "/>" + "\n";
8             String second_arg = PolEd.getMainKeyword(PolEd
9                 .getSecondPart(policy.getCondition()),
10                keyword_file);
11            ponder_policy = ponder_policy + "<arg name=\""
12                + second_arg + "/>" + "\n";
13          }
14          ponder_policy = ponder_policy + "<condition>" + "\n";
```

Jetzt wird der die beiden Teile verbindende Operator extrahiert und in die Ponder-Policy übernommen. Ebenso werden die Operatoren innerhalb der Teil-Elemente gesucht und in die Policy geschrieben.

```
1           ponder_policy = ponder_policy + "<"
```

```
2               + PolEd.getConnector(policy.getCondition()) + ">"
3               + "\n";
4           ponder_policy = ponder_policy
5               + "<"
6               + PolEd.getOperator(PolEd.getFirstPart(policy
7                   .getCondition()))
8               + ">"
9               + "!"
10              + PolEd.getMainKeyword(PolEd.getFirstPart(policy
11                  .getCondition()), keyword_file) + ";"
12              + "<!-- -->";
```

Ob der Parameter als Integer oder String vorliegt wird anschließend kontrolliert und danach in die Policy übernommen.

```
1           if (PolEd.extractParam(PolEd.getFirstPart(policy
2               .getCondition())) != null) {
3             if (PolEd.hasNumber(PolEd.extractParam(PolEd
4                 .getFirstPart(policy.getCondition())))) {
5               ponder_policy = ponder_policy
6                   + PolEd.turnToInt(PolEd
7                       .extractParam(PolEd
8                           .getFirstPart(policy
9                               .getCondition())));
10            } else {
11              ponder_policy = ponder_policy
12                  + PolEd
13                      .extractParam(PolEd
14                          .getFirstPart(policy
15                              .getCondition()));
16            }
17          }
```

Kann kein Parameter gefunden werden, wird eine Fehlermeldung in Kommentarform in die Policy geschrieben.

```
1           else {
2             ponder_policy = ponder_policy
3                 + "<!-- WARNING! No parameter found!! -->";
4           }
```

Analog wird mit dem zweiten Teil der Policy verfahren.

```
1           ponder_policy = ponder_policy
2               + "</"
3               + PolEd.getOperator(PolEd.getFirstPart(policy
4                   .getCondition())) + ">" + "\n";
5           ponder_policy = ponder_policy
6               + "<"
7               + PolEd.getOperator(PolEd.getSecondPart(policy
8                   .getCondition()))
9               + ">"
10              + "!"
11              + PolEd.getMainKeyword(PolEd.getSecondPart(policy
12                  .getCondition()), keyword_file) + ";"
13              + "<!-- -->";
14          if (PolEd.extractParam(PolEd.getSecondPart(policy
15              .getCondition())) != null) {
```

```
16              if (PolEd.hasNumber(PolEd.extractParam(PolEd
17                  .getSecondPart(policy.getCondition())))) {
18                ponder_policy = ponder_policy
19                    + PolEd.turnToInt(PolEd.extractParam(PolEd
20                        .getSecondPart(policy
21                            .getCondition())));
22              } else {
23                ponder_policy = ponder_policy
24                    + PolEd.extractParam(PolEd
25                        .getSecondPart(policy
26                            .getCondition()));
27              }
28            }
29            else {
30              ponder_policy = ponder_policy
31                  + "<!-- WARNING! No parameter found!! -->";
32            }
33            ponder_policy = ponder_policy
34                + "</"
35                + PolEd.getOperator(PolEd.getSecondPart(policy
36                    .getCondition())) + ">" + "\n";
37          }
```

Enthält die Condition nur einen Operator, wird analog zum oben beschriebenen Verfahren die Policy erstellt.

```
1           else {
2             if (PolEd.getOperator(policy.getCondition()) != null) {
3               if (PolEd.getMainKeyword(policy.getCondition(),
4                   keyword_file) != null) {
5                 String condition_arg = PolEd.getMainKeyword(policy
6                     .getCondition(), keyword_file);
7                 ponder_policy = ponder_policy + "<arg name=\""
8                     + condition_arg + "/>" + "\n";
9               }
10              ponder_policy = ponder_policy + "<condition>" + "\n";
11              ponder_policy = ponder_policy
12                  + "<"
13                  + PolEd.getOperator(policy.getCondition())
14                  + ">"
15                  + "!"
16                  + PolEd.getMainKeyword(policy.getPolicy(),
17                      keyword_file) + ";" + "<!-- -->";
18              if (PolEd.extractParam(policy.getCondition()) != null) {
19                if (PolEd.hasNumber(PolEd.extractParam(policy
20                    .getCondition()))) {
21                  ponder_policy = ponder_policy
22                      + PolEd.turnToInt(PolEd
23                          .extractParam(policy
24                              .getCondition()));
25                } else {
26                  ponder_policy = ponder_policy
27                      + PolEd.extractParam(policy
28                          .getCondition());
29                }
30              }
31              else {
32                ponder_policy = ponder_policy
```

```
33                    + "<!-- WARNING! No parameter found!! -->";
34              }
35              ponder_policy = ponder_policy + "</"
36                  + PolEd.getOperator(policy.getCondition())
37                  + ">" + "\n";
38            } else {
39              ponder_policy = ponder_policy
40                  + "<!-- Warning, no operator found!! -->"
41                  + "\n";
42            }
43          }
```

Das umschließende Argument für die Condition wird beendet. Das Startargument für die Action wird geschrieben und die Action nach einem Hauptreizwort durchsucht. Wie für die Condition wird auch hier nach einem Parameter gesucht und dieser in die Policy übernommen.

```
1         ponder_policy = ponder_policy + "</condition>" + "\n";
2         ponder_policy = ponder_policy + "<action>" + "\n";
3         if (PolEd.hasParam(policy.getAction())) {
4           ponder_policy = ponder_policy
5               + "<use name=\"/dom1/"
6               + PolEd.getMainKeyword(policy.getAction(),
7                   keyword_file) + "='"
8               + PolEd.extractParam(policy.getAction()) + "'"
9               + ">" + "\n";
10        } else {
11          ponder_policy = ponder_policy
12              + "<use name=\"/dom1/"
13              + PolEd.getMainKeyword(policy.getAction(),
14                  keyword_file) + ">" + "\n";
15        }
```

Abschließend werden die Argumente geschlossen, ein Kommentar als Markierung des Endes einer Policy geschrieben und die Laufvariable erhöht.

```
1         ponder_policy = ponder_policy + "<inc/>" + "\n";
2         ponder_policy = ponder_policy + "</use>" + "\n";
3         ponder_policy = ponder_policy + "</action>" + "\n";
4         ponder_policy = ponder_policy + "</create>" + "\n";
5         ponder_policy = ponder_policy + "</use>" + "\n";
6         ponder_policy = ponder_policy + "</add>" + "\n";
7         ponder_policy = ponder_policy + "</use>" + "\n";
8         ponder_policy = ponder_policy + "<!-- End of "
9             + policy.getDevice() + "_policy_" + index + " -->"
10            + "\n";
11        ponder_policy = ponder_policy + "\n";
12        index++;
```

Wird für mehr als ein Gerät eine Policy benötigt, so wird eine for-Schleife über alle identifizierten Geräte gestartet, die die oben aufgeführten Operationen für jedes Gerät ausführt.

```
1 for (int i = 0; i < OwlHandler.getDeviceCount(model, policy.getDevice
      ()); i++)
```

Ist die Policy für das eingegebene Gerät bearbeitet, wird der erzeugte String mit Hilfe des `FileHandlers` in eine Datei geschrieben.

```
1     FileHandler.writePonderPolicies(ponder_string);
```

4.2.7 Behandlung von Policy-Konflikten

Die potentielle Existenz von Konflikten wird bereits beim ersten Schritt des Refinements überprüft. Dabei finden zwei Arten von Überprüfungen statt:

- Wurde die aktuelle Policy bereits gespeichert und muß daher nicht weiter verarbeitet werden?
- Behandelt die aktuelle Policy eventuell einen Bereich bzw. ein Element, welches in einer vorherigen Policy schon Bestandteil war? (Hier wird das bereits in Kapitel 2.2.1 angesprochene Konzept der Überlappung geprüft)

Doppelte Policies werden durch einen simplen Vergleich des gesamten Policy-Strings identifiziert:

```
1 public static boolean hasBeenPolicy(String policy, JTable table) {
2     int rowcount = table.getRowCount();
3     for (int i = 0; i < rowcount; i++) {
4       if (policy.equals(table.getValueAt(i, 0))) {
5         return true;
6       }
7     }
8     return false;
9   }
```

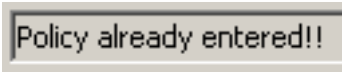


Abbildung 4.6: Identifizierung einer bereits vorhandenen Policy

Wird also nach der Eingabe und Aufruf dieser Methode „true“ zurückgegeben, wird die Policy verworfen. Der Benutzer wird über diesen Vorgang wie in Abbildung 4.6 zu sehen informiert.

Die Identifizierung einer Überlappung erfolgt durch einen Vergleich des Initialwortes der aktuellen Policy mit sämtlichen Initialworten aller bereits gespeicherten Policies:

```
1 public static boolean possibleConflict(String policy, String
      keyword_file,
2       JTable table) {
3     int rowcount = table.getRowCount();
4     for (int i = 0; i < rowcount; i++) {
5       if (getDeviceInitWord(policy, keyword_file).equals(
6           getDeviceInitWord(table.getValueAt(i, 0).toString(),
7               keyword_file))) {
8         return true;
9       }
10    }
11    return false;
```

```
12      }
```

Abbildung 4.7: Meldung über das Auftreten eines möglichen Konfliktes

Wurde eine solche Überlappung entdeckt wird die Policy zwar gespeichert, aber der Benutzer über die Möglichkeit des Auftretens eines Konfliktes informiert (siehe Abbildung 4.7).

5 Test und Vergleich

Nach der Beschreibung der Implementierung soll nun überprüft werden, ob der Policy-Editor den in Kapitel 3.1.1 definierten Anforderungen gerecht wird. Anhand von Anwendungsbeispielen aus dem Bereich eines Firmen-Netzwerkes soll dies getestet werden. Die dazu benötigte Topologie-Ontologie wird kurz vorgestellt. Besonderes Augenmerk liegt hier auf dem Prozess des Refinements. Der zweite Abschnitt dieses Kapitels befasst sich mit dem Vergleich anderer Implementierungen.

5.1 Test des Policy-Editors

Der Policy-Editor wurde auf einem Toshiba-Notebook mit einem Pentium-M 735 Prozessor mit 1.60 GHz und 1 GB RAM, unter Windows XP (Version 2002, Service Pack 2) getestet.

5.1.1 Verwendete Netzwerk-Topologie

Als Eingabe für den Policy-Editor dient die Netztopologie des bereits erwähnten 3.1.1 Firmennetzwerkes. Das für den Test gewählte Netzwerk hat den in Abbildung 5.1 zu sehenden Aufbau. Es besteht aus zwei Teilbereichen - dem Hauptbüro und dem Nebenbüro. Die beiden Büros sind über einen jeweiligen Router miteinander verbunden. Innerhalb der Büros befinden sich jeweils ein Webserver, sowie im Hauptbüro 30 Clients und im Nebenbüro 10 Clients. Zusätzlich ist jeweils an den Router ein Netzwerkdrucker angeschlossen. Die Ontologie enthält also einige für ein Firmennetzwerk durchaus typische Geräte für die es gilt Policies zu formulieren und zu verfeinern.

Die Eingabe der Netztopologie erfolgt im OWL-Format. Grafisch stellt sich die Topologie-Ontologie wie in Abbildung A.2 im Anhang dar. Hier ist auch zu erkennen, dass die Ontologie weitere Informationen enthält, wie zum Beispiel Funktionen, Standorte, etc. Beispielhaft zeigt die Aufstellung A.3 im Anhang einen Ausschnitt der OWL-Topologie in RDF-Form. Das Listing zeigt die Beschreibung des in der Topologie befindlichen Routers mit der Bezeichnung „Router_Office_1“.

Im alltäglichen Betrieb eines Firmen-Netzwerkes treten die unterschiedlichsten Anwendungsfälle auf, die hohe Anforderungen an die Konfiguration des Netzes stellen. So werden Daten vom Haupt- ins Nebenbüro übertragen, Telefonate per VoIP geführt, Emails versendet, Webcams angeschlossen und Konferenzen geführt, etc. Policies bieten hier, wie in den Grundlagen beschrieben, gute Möglichkeiten um diesen Anforderungen zur Laufzeit und ohne Benutzerinteraktion gerecht zu werden. Auftretende

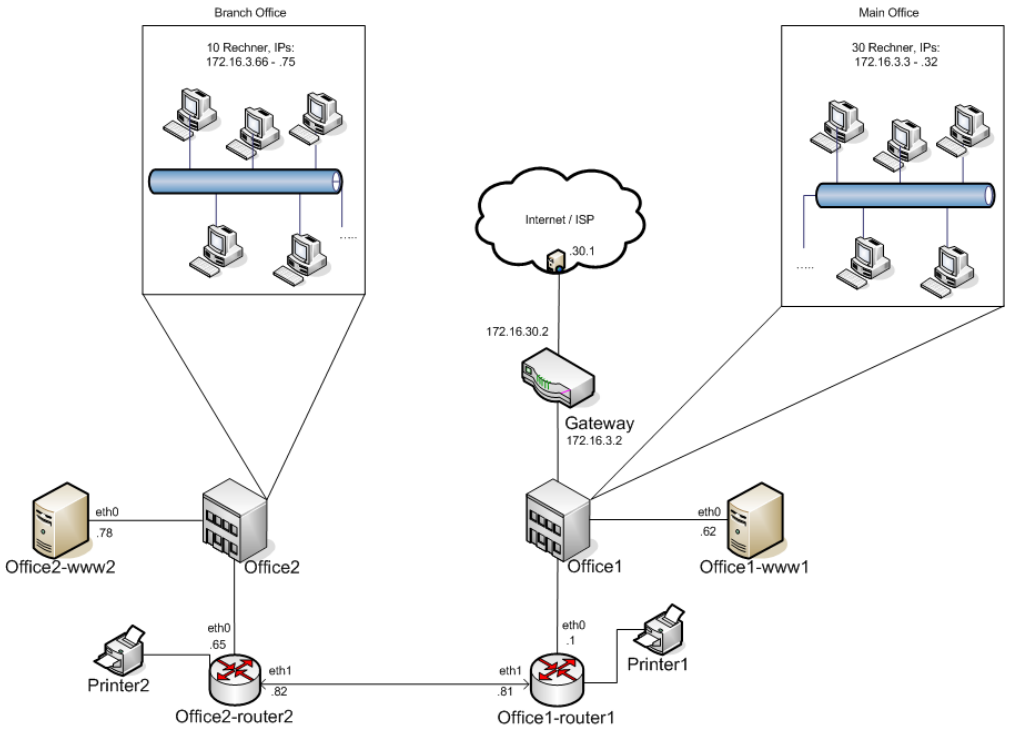


Abbildung 5.1: Netztopologie des Testnetzwerkes

Events haben eindeutige Aktionen zur Folge, sofern bestimmte Bedingungen (Condition) erfüllt sind. Es ist somit möglich Gerätekonfigurationen aus Anwendungsfällen abzuleiten.

5.1.2 Der Test-Reizwortkatalog

Die beim Test auftretenden Testfälle müssen, damit sie vom Policy-Editor erkannt werden können, zunächst im Reizwortkatalog vermerkt werden. Für den Test dienen die Fälle, dass Emails übertragen werden, VoIP-Telefonate geführt werden und Webcams angeschlossen werden, als Beispiel. Diese Reizworte müssen nun im Katalog vermerkt werden um Policies für die betreffenden Netzwerkgeräte formulieren zu können. Ebenso die damit in Verbindung stehenden Schlagworte für die Conditions und Actions müssen im Reizwortkatalog vorhanden sein. Wird eine Email versandt, wurden dafür die Reizworte „senderdomain“ und „receiverdomain“ definiert, die jeweils in Router-IPs resultieren. Eine resultierende Aktion könnte sein, die Verschlüsselung zu aktivieren, damit eine abgefangene Email die zwischen den Büros versendet wurde, nicht gelesen werden kann. Soll ein VoIP-Telefonat geführt werden und sind mehr als 20 Clients bzw. Benutzer Online, könnte ein Datenstau für die zeitkritischen VoIP-Daten auftreten. Daher wird als Aktion die Paketpriorität auf „hoch“ gesetzt. Bleibt der Fall der Webcam-Konferenz. Diese Daten sind ebenfalls zeitkritisch und bedürfen einer hohen Priorität. Weiterhin muss ein Personenkreis definiert werden, der an der Konferenz teilnimmt. Mit diesem Anwendungsfall soll ein Policy-Konflikt simuliert werden. Es werden zwei Policies in das System eingegeben. Die erste wird, analog zur VoIP-Policy die Anzahl der Benutzer überprüfen. Eine zweite Policy soll überprüfen, ob diejenige Adresse, die die Webcam angeschlossen hat, zum Teilnehmerkreis gehört. Ist dies der Fall, soll ein Port für die Webcam-Konferenz geöffnet werden. Der Reizwortkatalog könnte beispielsweise wie folgt aussehen:

```
1 Voip Client
2 Webcam Gateway
3 Email Router
4 -Main-Keyword-Area-
5 senderdomain Sdomain
6 receiverdomain Rdomain
7 scrambling encryption
8 priority packetpriority
9 users usercount
10 open port
11 close port
12 connectorrange IPrange
13 isIn domain
```

5.1.3 Test des Refinement-Prozesses

Für den Test des Refinement-Prozesses mit Hilfe des Reizwortkataloges werden analog zu den eben diskutierten Testfällen vier Policies formuliert:

1. Email sent, senderdomain = ‘Branchoffice’AND receiverdomain = ‘Mainoffice ’, activate scrambling
2. Voip session started, there are More than ‘20 ’users connected, set priority = ‘high ’
3. Webcam is connected, there are More than ‘15 ’users connected, set priority = ‘high ’

4. Webcam is connected, connectorrange = ‘172.16.3.66−75 ’AND isIn = ‘ConferenceDomain ’, open ‘5100’

Diese Policies werden nach Eingabe des Reizwortkataloges und der OWL-Netztopologie in den Policy-Editor eingegeben.

Erster Refinement Schritt

In einem ersten Schritt wird der Reizwortkatalog nach den passenden Netzwerkgeräten durchsucht. Bei einem Konflikt, der durch die Eingabe zweier Policies mit gleichem Event hervorgerufen wird, soll dieser angezeigt werden. Abbildung 5.2 zeigt das Ergebnis des ersten Refinement Schrittes. Die Fehlermeldung im Info-Panel unten links

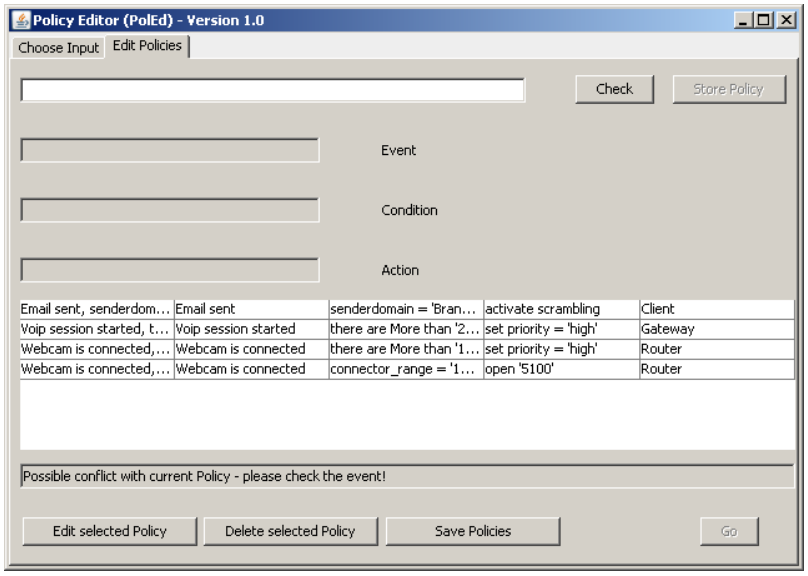


Abbildung 5.2: Eingelesene Test-Polices mit Hinweis auf Policy-Konflikt

zeigt deutlich, dass ein möglicher Konflikt identifiziert wurde. In der Tabelle des Policy-Editors (Abbildung 5.2) ist zu sehen, dass für jede Policy ein entsprechendes Netzwerkgerät gefunden wurde (letzte Spalte). Demnach wurden die Policy-Klauseln mit den Extraktionsmethoden `extractEvent()`, `extractCondition()` und `extractAction()` korrekt ausgelesen und die Reizworte mit der `checkDeviceKeywords()`-Methode gefunden. Die Speicherung der Policies funktioniert laut Info-Panel „Policies written to file: C:\Dokumente und Einstellungen\Administrator\Eigene Dateien\workspace\PolEd\ policy_file.txt“ ebenfalls. Die gespeicherte Policy-File sieht wie folgt aus:

```
1 Email sent, senderdomain = 'Branchoffice' $AND$ receiverdomain = '
      Mainoffice', activate scrambling|| Email sent|| senderdomain = '
      Branchoffice' $AND$ receiverdomain = 'Mainoffice'|| activate
      scrambling|| Router
2 Voip session started, there are More than '20' users connected, set
      priority = 'high'|| Voip session started|| there are More than '
      20' users connected|| set priority = 'high'|| Client
```

```
3 Webcam connects, there are More than '15' users connected, set
      priority = 'high'|| Webcam connects|| there are More than '15'
      users connected|| set priority = 'high'|| Gateway
4 Webcam connects, connectorrange = '172.16.3.66-75' $AND$ isIn = '
      ConferenceDomain', open '5100'|| Webcam connects||
      connectorrange = '172.16.3.66-75' $AND$ isIn = 'ConferenceDomain
      '|| open '5100'|| Gateway
```

Zweiter Refinement Schritt

Der zweite Schritt des Refinements beinhaltet drei Elemente:

1. die Anfrage an die OWL-Topologie,
2. die Suche nach einem Hauptreizwort,
3. die Transformation in eine Ponder-Policy.

Ein Druck auf den Knopf „Go“ sollte demnach eine Datei mit verfeinerten Ponder-Policies liefern. Das Infopanel bestätigt die erfolgreiche Ausführung der Transformation durch Angabe des Speicherortes der Ponder-Policy-File (Abbildung 5.3). Öffnet man die Datei an angegebener Stelle, finden sich darin die verfeinerten Policies im Ponder-Format. Exemplarisch seien hier die Policies für jeweils ein gefundenes Netz-

Abbildung 5.3: Transformation in Ponder-Policies durchgeführt

werkgerät aufgeführt, in Klammern ist die Anzahl der erstellten Policies angegeben. Router-Policy (beim Test zwei erstellt):

```
1 <!-- template := root/event/Email create: #( "Router" "value" )-->
2 <use name="/policy">
3 <add name="Router_Office_1_policy_0>
4 <use name="/template/policy">
5 <create type="obligation" event="/event/Email active="true">
6 <arg name="Sdomain/>
7 <arg name="Rdomain/>
8 <condition>
9 <AND>
10 <eq>!Sdomain;<!-- -->Branchoffice</eq>
11 <eq>!Rdomain;<!-- -->Mainoffice</eq>
12 </condition>
13 <action>
14 <use name="/dom1/encryption>
15 <inc/>
16 </use>
17 </action>
18 </create>
19 </use>
20 </add>
21 </use>
22 <!-- End of Router_policy_0 -->
```

Client-Policy (beim Test 40 erstellt):

```
<!-- template := root/event/Voip create: #( "Client" "value" )-->
<use name="/policy">
<add name="Client_Office1_Client3_policy_41>
<use name="/template/policy">
<create type="obligation" event="/event/Voip active="true">
<arg name="usercount/>
<condition>
<gt>!usercount;<!-- -->20</gt>
</condition>
<action>
<use name="/dom1/packetpriority='high'>
<inc/>
</use>
</action>
</create>
</use>
</add>
</use>
<!-- End of Client_policy_41 -->
```

Gateway-Policy der dritten Policy (beim Test eine erstellt):

```
<!-- template := root/event/Webcam create: #( "Gateway" "value" )-->
<use name="/policy">
<add name="Gateway_1_policy_42>
<use name="/template/policy">
<create type="obligation" event="/event/Webcam active="true">
<arg name="usercount/>
<condition>
<gt>!usercount;<!-- -->15</gt>
</condition>
<action>
<use name="/dom1/packetpriority='high'>
<inc/>
</use>
</action>
</create>
</use>
</add>
</use>
<!-- End of Gateway_policy_42 -->
```

Gateway-Policy der vierten Policy (beim Test eine erstellt):

```
<!-- template := root/event/Webcam create: #( "Gateway" "value" )-->
<use name="/policy">
<add name="Gateway_1_policy_43>
<use name="/template/policy">
<create type="obligation" event="/event/Webcam active="true">
<arg name="IPrange/>
<arg name="domain/>
<condition>
<AND>
<eq>!IPrange;<!-- -->172.16.3.66-75</eq>
<eq>!domain;<!-- -->ConferenceDomain</eq>
</condition>
```

```
13 <action>
14 <use name="/dom1/port='5100'>
15 <inc/>
16 </use>
17 </action>
18 </create>
19 </use>
20 </add>
21 </use>
22 <!-- End of Gateway_policy_43 -->
```

Insgesamt lässt sich festhalten, dass der Policy-Editor seine Aufgabe erfüllt und den Anforderungen aus Kapitel 3.1.1 gerecht wird. Alle aus der Topologie ausgelesenen Geräte werden beim Refinement berücksichtigt. Die entsprechenden Event-Templates für ein späteres Enforcement durch den Ponder2-Framework werden ebenfalls geschrieben. Besonders hervorzuheben ist die Möglichkeit, neue Reizworte in den Katalog einzufügen. Dadurch wächst die Skalierbarkeit des Policy-Editors stark an. Der Policy-Editor ist somit theoretisch in der Lage, für jedes beliebige Netzwerk und jeden Spezialfall ein Refinement durchzuführen - sofern die entsprechenden Reizworte im Katalog zu finden sind. Deutlich wird hier das Problem, dass sich hinter dem Konstrukt des Reizwortkataloges verbirgt. Sollen sehr viele Anwendungsfälle verwaltet werden, wird der Katalog leicht unübersichtlich. Ein weiteres Problem ist sicherlich die notwendige Eindeutigkeit der Reizworte, um eine Maschinenlesbarkeit garantieren zu können. In gesprochenen Sprachen ist eine Mehrdeutigkeit und Wiederverwendbarkeit eines Wortes für mehrere Fälle durchaus üblich. Der Policy-Editor unterliegt hier also einer starken Beschränkung.

5.1.4 Performanz des Policy-Editors

Da das Konzept des Refinements des Policy-Editors auf dem Reizwortkatalog beruht, bleibt zu testen, ob die Größe des Reizwortkataloges einen signifikanten Einfluß auf die Laufzeit hat. Dafür wurden Reizwortkataloge mit unterschiedlich vielen Einträgen erstellt. Getestet wurde der Zeitaufwand eine Policy zu verfeinern. Dabei wurden dem Policy-Editor nacheinander Kataloge mit über

- 500 Zeileneinträgen,
- 1.000 Zeileneinträgen,
- 10.000 Zeileneinträgen,
- 15.000 Zeileneinträgen,
- 60.000 Zeileneinträgen,
- 120.000 Zeileneinträgen,
- 500.000 Zeileneinträgen, und
- 1.000.000 Zeileneinträgen

zur Verfügung gestellt. Die zu suchenden Reizworte des Hauptbereiches befanden sich dabei in den letzten neun Zeilen des Kataloges. Eine auffallende Verlangsamung trat

jedoch erst ab einer Katalog-Größe von mehr als 60.000 Zeilen auf. Der Policy-Editor benötigte für das Refinement mit Hilfe der Kataloge mit den Größen 500−10.000 Zeilen weniger als eine Sekunde (ca. *0,2 Sekunden* für 500 Einträge, ca. *0,4 Sekunden* für 1.000 Einträge und ca. *0,85 Sekunden* für 10.000 Einträge). Bei 15.000 Zeilen dauerte das Refinement etwa *1 Sekunde*. Um 60.000 Einträge zu durchsuchen, waren bereits ca. *1,7 Sekunden* notwendig. 120.000 Zeilen schlugen mit *2,5 Sekunden* zu Buche. Die Suche in einem Katalog mit über 500.000 Einträgen benötigte schon ungefähr *8,2 Sekunden*. Das Refinement mit einem Katalog einer Größe von über 1 Mio. Einträgen dauerte gar *50,8 Sekunden*. Zu erkennen ist ein nicht-lineares, eher progressives Wachstum der Laufzeit.

Diese Laufzeiten sind jedoch nicht unbedingt als schlecht zu bewerten, da während des Refinements mehrfach auf den Katalog zugegriffen wird. Sowohl für die Suche nach einer Geräteklasse, als auch nach einem Haupt-Schlagwort, werden die Zeilen des Kataloges durchsucht. Auch die Überprüfung, ob ein Konflikt vorliegt, hat eine Suche im Reizwortkatalog zur Folge. Ebenso ist das eingangs genannte Testsystem nicht aktuell. Neuere, schnellere Rechner mit entsprechend höherer Taktfrequenz erledigen das Refinement deutlich schneller. (Bereits ein Pentium-M 740 Prozessor der nächsten Generation mit 1.73 GHz Taktfrequenz und 1 GB RAM war im Schnitt ca. 10−15% schneller.)

Der Test mit über 1 Mio. Zeilen stellt hier ein Extremum dar. Für ein Firmen-Netzwerk ist dieser Fall nicht unbedingt relevant. Allein der „Duden der deutschen Rechtschreibung“ enthält nur ca. 130.000 Stichwörter. [55] Daher wird der Testfall mit 120.000 Zeileneinträgen als realistische Obergrenze betrachtet. Mit ca. 2,5 Sekunden, um 120.000 Schlagworte zu durchsuchen ist der Policy-Editor daher eher als schnell für das Refinement einer einzelnen Policy anzusehen.

Weitere Tests wurden aufgrund der angesprochenen „Obergrenze“ von 120.000 Einträgen mit dem Mittelwert von 60.000 Zeilen durchgeführt. Die oben genannten Laufzeiten bezogen sich auf die Laufzeit für das Refinement einer einzigen Policy. In tatsächlichen Netzwerken werden jedoch sicherlich deutlich mehr Policies formuliert werden müssen, um möglichst viele Anwendungsfälle abzudecken. Um ein aussagekräftiges Testergebnis zu erhalten, wurden 100 Policies zur Verfeinerung und Übersetzung in den Policy-Editor eingelesen. Auf dem vorgestellten Testsystem benötigte der Policy-Editor dafür ca. *40 Minuten und 36 Sekunden*. Dabei wurden 1125 Ponder-Policies geschrieben.

Der Policy-Editor ist demnach zwar in der Lage auch sehr große Reizwortkataloge in relativ geringer Zeit zu verarbeiten, zeigt aber bei einem realistischen Test mit 100 Policies bereits eine recht hohe Laufzeit. Da das Policy-Refinement aber meist vor dem Einsatz der verfeinerten Policies im Policy Enforcement Point geschieht, wäre es trotz allem durchaus möglich, den Policy-Editor auch für große Netzwerke einzusetzen. Sind die Policies erst einmal formuliert und auf die technische Ebene verfeinert worden, ist es am PEP, die schnell auftretenden Events zu verarbeiten und die entsprechenden Policies durchzusetzen. Der Refinement-Prozess wird daher nicht als zeitkritisch angesehen. Weiterhin ist anzumerken, dass die Übersetzung mehrerer Business-Policies in technische Konfigurationen, von Hand sicherlich deutlich länger benötigt.

5.2 Vergleich mit anderen Implementierungen

Quelltexte zum direkten Vergleich des Policy-Editors mit anderen Implementierungen sind so gut wie nicht frei verfügbar. Autoren schützen ihren Programmcode und beschreiben in Ausarbeitungen und Papers meist nur das Vorgehen und die Möglichkeiten. Ein Vergleich kann also nur dahingehend stattfinden, die Ziele und Möglichkeiten der Implementierungen gegenüber zu stellen. Implementierungen von John Strassner, wie eine Fallstudie zum bio-Inspired PBNM [56] oder der Prototyp für Autonomic Networking [57], behandeln meist nur Quality of Service- und DiffServ-Aspekte. So ist der Autonomic Networking Prototyp ausschließlich für die Durchsetzung von Service Level Agreements geschrieben worden. Die bio-Inspired PBNM Fallstudie versucht eine Verbindung zwischen dem Policy-Kontinuum und der molekularen Regulation in Organismen herzustellen. Von dort wird dann auf DiffServ geschlossen und die Fallstudie dahingehend erweitert, das Policy-Kontinuum für Differentiated Services anwendbar zu machen. Metriken, wie die Erweiterbarkeit oder die Skalierbarkeit, lassen sich hier schlecht anlegen, da diese Implementierungen nur einen sehr kleinen und sehr speziellen Teil der Möglichkeiten des PBNM abdecken.

Aktuellere Implementierungskonzepte von Strassner et al benutzen ein Ontology-Mapping [53] zur automatischen Policy-Übersetzung bzw. ein formales Modell des Policy-Kontinuums, um das Kontinuum selbst zu manipulieren [7]. Das Ontology-Mapping stellt dabei ein völlig neues Konzept dar. Es läuft in zwei Phasen ab:

1. Phase: Abbildung der Policy-Sprachkonzepte auf eine Interlingua-Ontologie und umgekehrt.
   Dieser Prozess kann entweder manuell oder halb-automatisch, mit anschließender Validierung durch einen menschlichen Experten, ablaufen.
2. Phase: Dynamische Generierung einer Abbildung von der Quell-Policy-Sprachontologie auf die Ziel-Policy-Sprachontologie.
   Dieses Mapping basiert auf den in Phase 1 erstellten Interlingua-Ontologien.

Das Konzept des formalen Policy-Kontinuum-Modells spezifiziert Basisoperationen, die auf das Modell angewendet werden können. Dazu gehören:

- Das Erstellen einer Policy.
- Hinzufügen eines Events zu einer Policy.
- Entfernen eines Events aus einer Policy.
- Hinzufügen einer Policy zum Modell.
- Entfernen einer Policy aus dem Modell.
- Aktualisieren einer Policy des Modells.
- Ausgabe der „Kinder“ einer Policy.
- Ausgabe von Policies einer bestimmten Kontinuums-Stufe.
- Ausgabe aller Policies.
- Ausgabe der „Eltern“ einer oder mehrerer Policies.

- Ausgabe generalisierter Policies (Policies unterer Kontinuums-Stufen).

Der im formalen Modell vorgestellte Prozess des Policy-Authorings wird in drei Schritte unterteilt:

1. Verifizierung der Korrektheit und der strukturellen semantischen Integrität der Änderungen durch die Policy am Kontinuum.
2. Analyse des Konfliktpotentials der Policies einer Ebene.
3. Aufruf des Refinements.

Grafisch stellt sich der Autoring-Prozess wie in Abbildung 5.4 gezeigt dar. Bei beiden

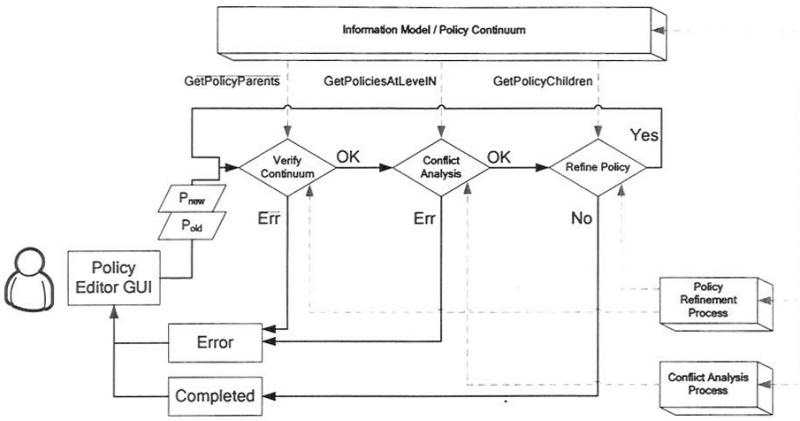


Abbildung 5.4: Policy-Authoring Prozess [7]

Konzepten wird der Refinement Prozess nicht beschrieben, so dass ein Vergleich mit der vorliegenden Implementierung des Policy-Editors nicht möglich ist.
Der von Rubio vorgestellte „methologische Ansatz“ verwendet ein zielbasiertes Refinement per Zielgraph.[13] Dabei werden eine Zieldatenbank angelegt und Refinement-Muster bei der Policy-Verfeinerung zu Hilfe genommen. Der Refinement-Prozess wählt dann diejenigen Policies aus, die das gewünschte Ziel erfüllen. Die Architektur des Frameworks für diese Implementierung findet sich in Abbildung 5.5. Dieses Konzept ist mit dem des Policy-Editors vergleichbar. Anstelle einer Zieldatenbank benutzt der Policy-Editor einen Reizwortkatalog, um ein Refinement durchzuführen. Allerdings liegen die Reizworte für den Policy-Editor nicht als Graph vor.

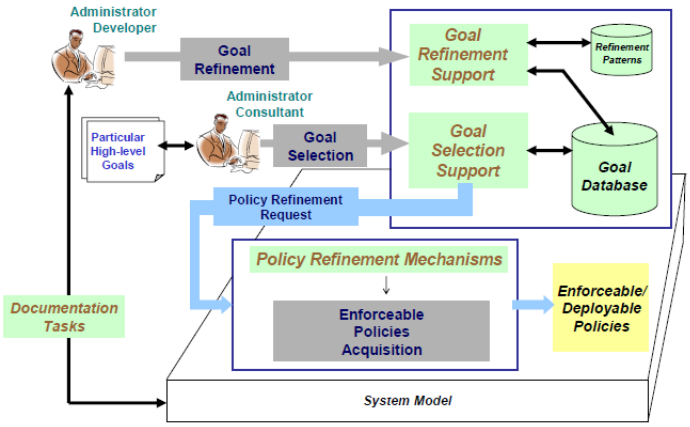


Abbildung 5.5: Architektur des Ansatzes von Rubio [13]

Anders sieht es dagegen mit einer Implementierung von Benjamin Ernst zum Thema „Automatisches Policy-Refinement mit Hilfe von semantischen Infrastruktur-Informationen“, genannt SEMPR, aus. Sie stellt sozusagen den „Vorgänger“ der hier vorliegenden Ausarbeitung dar und beschäftigt sich mit dem Refinement mit Hilfe von OWL-Services. Ebenso wie der Policy-Editor wird dabei eine OWL-Ontologie zur Modellierung einer Netztopologie und als Eingabe verwendet. Der größte Unterschied besteht offensichtlich in der Verwendung von Services. SEMPR kombiniert Services, um das gewünschte Ergebnis zu erhalten. Policies werden somit als Services formuliert durch Service-Composition bereits bestehender Services geschaffen. Auch das Refinement per Services macht von den Ebenen des Policy-Kontinuums Gebrauch. Ebenso wie der Policy-Editor werden jedoch auch in der SEMPR-Implementierung nicht alle Ebenen benutzt, bzw. die „Struktur des Policy-Kontinuums (...) aufgeweicht“[10]. Der Policy-Editor vergleicht Schlagworte aus einer Datei mit den gefundenen Elementen einer Policy. Die Refinement-Engine der SEMPR-Implementierung sucht mit Hilfe eines Matchmakers die passenden Web Services. Der Ablauf des Refinements bei SEMPR generiert jedoch, im Unterschied zum Policy-Editor, aus Policy-Action und -Condition das Ziel des Refinements. Das so generierte Ziel muss von einem Web-Service erfüllt werden können. Der Policy-Editor verlangt bei der Erstellung einer Policy dagegen auch die Definition der auszuführenden Aktion.

Legt man nun an beide Implementierungen die Metrik der Skalierbarkeit an, so verhalten sich die Implementierungen durchaus ähnlich. Einzig der Einsatzbereich stellt hier den größten Einflußfaktor dar. Die SEMPR-Implementierung wurde speziell für Home-Networks entwickelt. Dort eingesetzte Geräte und Sensoren sind sicherlich ebenso vielfältig, wie im Bereich von für den Policy-Editor relevanten Firmennetzwerken. Erweitert man die Policy-Ontologien um entsprechende Anwendungsfälle, werden diese von der SEMPR-Implementierung berücksichtigt. Der Policy-Editor bietet diese Möglichkeit der Erweiterbarkeit durch die Ergänzung von Reizworten für die vorliegenden Anwendungsfälle und die im Netzwerk vorkommenden Geräte. Allerdings kann die Erweiterbarkeit durch das Konstrukt des Reizwortkataloges als um einiges simpler und schneller betrachtet werden. Vorteil der SEMPR-Implementierung

gegenüber dem Policy-Editor bleibt allein die verteilte Architektur. Allerdings benötigt SEMPR zur Ausführung des Refinements bestehende und funktionierende Netzwerkverbindungen und eventuell sogar einen Internetzugang, um die Ontologien einlesen zu können. Dies ist beim Policy-Editor nicht der Fall, da er als Stand-Alone-Lösung auf einem Rechner läuft.

6 Fazit

6.1 Zusammenfassung

In der vorliegenden Ausarbeitung wurden die Grundlagen des Policy Based Network Management vorgestellt. Besonderes Augenmerk galt dabei dem Policy-Kontinuum und verschiedenartigen Ansätzen ein Policy-Refinement durchzuführen.
Directory Enabled Networking - new generation wurde vorgestellt und seine Grundkonzepte erläutert. Der Vorgänger des DEN-ng, das Common Information Model, wurde ebenfalls kurz dargelegt. Auch die Grundkonzepte des Semantic Web mit den Ausprägungen OWL und RDF wurden erläutert, da sie in der Implementierung ihre Anwendung fanden. Darauf folgte Eine Darstellung des Ponder2-Frameworks für Managed Objects. Die dabei vorgestellte Definition einer Ponder-Policy fand ebenso ihre Anwendung in der Implementierung. Somit wurden Umfang und Anforderungen der Implementierung bereits reduziert bzw. definiert.

Eine Anforderungsanalyse übernahm diese ersten Einschränkungen und definierte weitere Ziele einer Implementierung. Die ersten Konzepte wurden in UML formuliert und eine vorläufige Architektur des Policy-Editors beschrieben. Verwendete Bibliotheken und Konzepte wie Jena und Ponder2 wurden kurz angesprochen.
Es folgte die Konzeption eines Refinement-Algorithmus mit Hilfe eines Reizwortkataloges. Dabei werden Reizworte aus einer Datei ausgelesen, um drei verschiedene Ebenen des Policy-Kontinuums zu durchlaufen. Somit wurde es möglich, von High-Level-Policies auf Low-Level-Policies zu schließen. Die Transformation in eine Ponder-Policy konvertiert die verfeinerten Policies in ein maschinenlesbares Format.

Anschließend wurde mit Hilfe der vorgestellten Klassenbibliotheken (Jena, Ponder) der Policy-Editor in Java prototypisch implementiert. Er besteht aus einer grafischen Oberfläche und den Klassen und Methoden des Editors. Der Prototyp liest Netzwerkontologien im OWL-Format ein, um diese im späteren Refinement zu verarbeiten. Ein Benutzer hat die Möglichkeit, Policies in den Editor einzugeben, bereits eingegebene Policies zu löschen und zu editieren. Eine Abspeicherung der bereits formulierten Policies ist ebenfalls möglich. Durch den Abgleich der in den formulierten Policies enthaltenen Worte mit denen der im Reizwortkatalog befindlichen Schlagworte, wird ein Refinement ermöglicht. Die Aufgabe der Überführung in konkrete Gerätekonfigurationen wird durch die Transformation in eine Ponder-Policy dem Ponder-Framework überlassen.

Der Test des Policy-Editors erfolgte mit einer OWL-Topologie eines kleinen Firmen-Netzwerkes. In diesem Bereich bietet sich ein PBNM an, um das Verhalten des Netzwerkes bei auftretenden Ereignissen zu kontrollieren. Die Ontologie wurde mit Hilfe

von Protegé erstellt und dem Programm als Eingabe zur Verfügung gestellt. Es wurden Test-Policies formuliert und die dafür benötigten Reizworte im Reizwortkatalog hinterlegt.

6.2 Bewertung

Durch die Bereitstellung von Reizworten in einem Katalog kann einem PBNM-System umfangreiche Möglichkeiten zur Konfiguration zur Verfügung stellen. Durch die Netzwerk-Ontologien war der Policy-Editor in der Lage, diejenigen Geräte zu identifizieren, die tatsächlich von einer Policy betroffen sind, und sie namentlich in der Policy hinterlegen. Weitere Netzwerkgeräte und Anwendungsfälle zu verwalten, stellt kein Problem für das vorgestellte Konzept dar. Sie können einfach in den Reizwortkatalog aufgenommen werden. Ein automatisches Refinement ist somit für eine große Anzahl verschiedener Netze möglich. Auch eine Anwendung in einem Bereich, der nicht der eines Firmennetzwerkes entspricht, ist denkbar. Auch außerhalb des PBNM könnte dieses Konzept Andwendung finden, z.B. zur Verwaltung von Systemen, die zwar eine netzwerkartige Struktur aufweisen, jedoch kein (Computer-)Netzwerk sind. Solange die System-Topologie als OWL-Ontologie vorliegt und die Geräteklassen, sowie Anwendungsfälle, im Reizwortkatalog zu finden sind, kann ein Policy-Refinement stattfinden.

Der Ansatz hat allerdings auch gezeigt, dass die Eindeutigkeit der Reizworte ein Problem darstellen kann. Gesprochene Sprachen erlauben die Mehrdeutigkeit eines Begriffes. Für den Policy-Editor müssen Informationen aus dem Reizwortkatalog jedoch eindeutig sein, da es sonst zu Policy-Konflikten kommen kann. Weiterhin kann der Reizwortkatalog, durch die Vielfalt der zu verwaltenden Geräteklassen und Anwendungsfälle, extrem groß werden. Dadurch leidet die Übersichtlichkeit und damit auch die simple Erweiterbarkeit des Reizwortkataloges. Zudem kann eine außerordentliche Größe mit entsprechend vielen Reizworten die Performanz des Policy-Editors negativ beeinflussen. Besonders bei der Konflikt-Identifizierung würde dies ins Gewicht fallen. Auch die Suche nach einem Reizwort kann sich entsprechend zeitlich verlängern, wenn sich beispielsweise das gesuchte Wort am Ende eines Kataloges mit über 120.000 Zeilen befindet.

6.3 Ausblick

In den Grundlagen wurden bereits die Vorteile und erheblichen Möglichkeiten des PBNM vorgestellt. Eine weitere Erforschung und Entwicklung von Systemen in diesem Bereich ist daher unabdingbar. Wie die Literaturrecherche (Kapitel 2.1.1) gezeigt hat, findet eine intensive Forschung zum PBNM, Policy-Refinement und OWL-Ontologien, statt. Das Interesse am Thema und der hohe Forschungsaufwand sind dabei durchaus berechtigt. Die verschiedenen Anwendungsbereiche des PBNM und unterschiedlichen Konzepte, ein Refinement durchzuführen, zeigen, dass das Potential des PBNM längst nicht ausgeschöpft ist. Durch Weiterentwicklungen, sowohl in den

Bereichen der OWL-Ontologien und der PEP-Frameworks [1], als auch auf Hardwareseite, könnte ein automatisches Refinement Einzug in Netzwerke und andere Systeme halten.

In Bezug auf die vorgestellte Implementierung könnte das oben angesprochene Problem der Mehrdeutigkeit der Sprache durch die Überprüfung von Wort-Kombinationen gemildert werden. Dadurch wäre die im Test aufgefallene hohe Laufzeit für das Refinement vieler Policies, bei gleichzeitig vielen vorhandenen Reizworten, weiter reduzierbar. Auf der Hand liegt die Erweiterung des Policy-Editors durch die Anbindung an den vorgestellten Ponder2-Framework, um die verfeinerten Ponder-Policies mit Hilfe des Ponder Authorisation Frameworks durchzusetzen. Eine andere sinnvolle Weiterentwicklung des Policy-Editors wäre zudem die verteilte Anwendung innerhalb eines Netzwerkes.

[1] Eine neue Version des Ponder2-Frameworks ist laut Projektwebseite bereits in Arbeit[44]

Literaturverzeichnis

[1] D. Kosiur. *Understanding Policy-Based Networking*. Robert Ipsen, Wiley Networking Council Series, 2001.

[2] A. R. Choudhary. Policy-Based Network Management. *Bell Labs Technical Journal*, Volume 9(1), 2004.

[3] J. D. Moffett, M. S. Sloman. Policy Hierarchies for distributed Systems Management. In *IEEE JSAC Special Issue on Network Management*, volume Volume 11, Number 9, 1993.

[4] J. Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegekas, G. Pavlou. A Functional Solution for Goal-oriented Policy Refinement. *Proceedings of Seventh IEEE International Workshop on Policies for Distributed Systems and Networks*, 2006.

[5] M. S. Beigi, S. Calo, D. Verma. Proceedings of Fifth IEEE International Workshop on Policies for Distributed Systems and Networks. In *Policy Transformation Techniques in Policy-based Systems Management*, pages 13–22, 2004.

[6] J. C. Strassner. *Policy-Based Network Management - Solutions for the next Generation*. Morgan Kaufman Publishers, 2004.

[7] S. Davy, B. Jennings, J. Strassner. The Policy Continuum - A Formal Model. In *Modelling Autonomic Communications Environments -MACE2007*, pages 65–78, 2007.

[8] Distributed Management Task Force. Schematische Darstellungen der CIM-Policy. `http://www.dmtf.org/standards/cim/cim_schema_v217/CIM_Policy.pdf`.

[9] World Wide Web Consortium (W3C). RDF Concepts. `http://www.w3.org/TR/rdf-concepts/`.

[10] B. Ernst. Automatisches Policy-Refinement mit Hilfe von semantischen Infrastruktur-Informationen. Diploma Thesis, Technische Universität Braunschweig, 2007.

[11] Imperial College London. Ponder2 Authorisation Enforcement. `http://ponder2.net/cgi-bin/moin.cgi/Ponder2Authorisation`.

[12] A. Keller, H. Ludwig. Policy-basiertes Management: State-of-the-Art und zukünftige Fragestellungen. *in: Praxis der Informationsverarbeitung und Kommunikation*, Volume 27(Issue 2), 2004.

[13] J. R. Loyola. *A Methodological Approach to Policy Refinement in Policy-based Management Systems*. PhD thesis, Universitat Politècnica de Catalunya, 2007.

[14] E. Lupu, N. Dulay, M. Sloman, J.Sventek, S. Heeps, S. Strowes, K. Twidle, S.-L. Keoh, A. Schaeffer-Filho. AMUSE: Autonomic Management of Ubiquitous e-Health Systems. *Concurrency and Computation: Practice and Experience*, 2007.

[15] K. J. Turner, G. A. Campbell, F. Wang. Policies for Sensor Networks and Home Care Networks. In *In Proceedings of the 7th. International Conference on New Technologies for Distributed Systems*, pages 273–284, 2007.

[16] G. Russello, C. Dong, N. Dulay, J. Singh, J. Bacon, K. Moody. A Policy-Based Framework for e-Health Applications. In *In Proceedings of the UK e-Science All Hands Meeting*, 2007.

[17] E. Asmare, N. Dulay, E. Lupu, M. Sloman. Towards Self-managing Unmanned Autonomous Vehicles. In *In Proceedings for the 2nd Systems Engineering for Autonomous Systems DTC Technical Conference - Edinburgh 2007*, 2007.

[18] S. Reiff-Marganiec, K.h J. Turner. A Policy Architecture for Enhancing and Controlling Features. In *In Proceedings of the Feature Interactions in Telecommunication Networks VII*, pages 239–246, 2003.

[19] G.A. Campbell. Sensor Network Policy Conflicts. In *In Proceedings of the Ninth International Conference on Feature Interactions in Software and Communication Systems 2007, ICFI07*, 2007.

[20] M. H. Burstein. Ontology Mapping for Dynamic Service Invocation on the Semantic Web. In *In Proceedings of the AAAI Spring Symposium on Semantic Web Services*, 2004.

[21] G. A. Campbell, K. J. Turner. Ontologies to support Call Control Policies. In *In Proceedings of the 3rd Advanced International Conference on Telecommunications (AICT'07), IEEE Computer Society*, pages 5.1–5.6, 2007.

[22] A. Westerinen, et al. Terminology for Policy-Based Management. Technical report, IETF, 2001.

[23] F. Wang, K. J. Turner. Policy Conflicts in Home Care Systems. In *In Proceedings of the Ninth International Conference on Feature Interactions in Software and Communication Systems 2007, ICFI07*, 2007.

[24] A. K. Bandara, E. C. Lupu, A. Russo, N. Dulay, M. Sloman, P. Flegakas, M. Charalambides, G. Pavlou. Policy Refinement for DiffServ Quality of Service Management. *IFIP/IEEE International Symposium*, 2005.

[25] Katia Hristova and Yanhong A. Liu. Improved Algorithm Complexities for Linear Temporal Logic Model Checking of Pushdown Systems. In *VMCAI*, pages 190–206, 2006.

[26] Distributed Management Task Force. Organisations-Webseite. `http://www.dmtf.org/`.

[27] Cisco Systems. *Internetworking Technology Handbook*. Cisco Systems, 1999.

[28] Distributed Management Task Force. Organisations-Webseite. `http://www.wbemsolutions.com/tutorials/DMTF/dmtftutorial.pdf`.

[29] J. Strassner. DEN-ng: Achieving Business-Driven Network Management. In *Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP*, pages 753–766, 2002.

[30] Distributed Management Task Force. Organisations-Webseite. `http://www.dmtf.org/standards/cim/`.

[31] World Wide Web Consortium (W3C). Semantic Web. `http://www.w3.org/2001/sw/`.

[32] F. Gruber. Semantic Web – Ein Henne-Ei-Problem. `http://cms.fh-augsburg.de/report/2006/Gruber_Frank__Semantic_Web/Semantic_Web.pdf`, 2006.

[33] Tim Berners-Lee, Jim Hendler, Ora Lassila. The SemanticWeb. *Scientific American*, 284 (5), 2001.

[34] Meyers Lexikon. Ontologie - Philosophisch. `http://lexikon.meyers.de/meyers/Ontologie`.

[35] S. Heeps, J. Sventek, N. Dulay, A. E. Schaeffer Filho, E. C. Lupu, M. Sloman S. Strowes. Dynamic Ontology Mapping for Interacting Autonomous Systems. In *Proceedings of the Second International Workshop on Self-Organizing Systems, IWSOS2007*, pages 255–263, 2007.

[36] Ying Ding, Dieter Fensel, Hans-Georg Stork. The Semantic Web: from Concept to Percept. *Österreichische Gesellschaft für Artificial Intelligence (OGAI)*, 2003.

[37] World Wide Web Consortium (W3C). OWL Reference. `http://www.w3.org/TR/owl-ref/`.

[38] World Wide Web Consortium (W3C). RDF Syntax. `http://www.w3.org/TR/rdf-syntax-grammar/`.

[39] T. Berners-Lee, et al. Uniform Resource Identifier (URI): Generic Syntax. Technical report, IETF, 2005.

[40] S. Decker, F. van Harmelen, J. Broekstra, M. Erdmann, D. Fensel, I. Horrocks, M. Klein, S. Melnik. The semantic web - on the respective roles of XML and RDF. *Institute of Electrical and Electronics Engineers (IEEE) Internet Computing*, 4(5):63–74, 2000.

[41] World Wide Web Consortium (W3C). SPARQL. `http://www.w3.org/TR/rdf-sparql-query/`.

[42] Hewlett Packard Labs Semantic Web Research. SPARQL Tutorial. `http://jena.sourceforge.net/ARQ/Tutorial/query1.html`.

[43] Imperial College London. PONDER Implementation Guide - Extending the Ponder toolkit to support the enforcement of users' defined obligation policies. `http://www-dse.doc.ic.ac.uk/Research/policies/ponder/docs/PONDERImplementationGuide.pdf`, 2003.

[44] Imperial College London. Ponder2 Projekt-Webseite. `http://ponder2.net/`.

[45] Imperial College London. Ponder2 Managed Objects. `http://ponder2.net/cgi-bin/moin.cgi/ManagedObjects`.

[46] Imperial College London. Ponder2 Domain Service. `http://ponder2.net/cgi-bin/moin.cgi/DomainService`.

[47] Imperial College London. Ponder2 Obligation Policy Interpreter. `http://ponder2.net/cgi-bin/moin.cgi/ObligationPolicyInterpreter`.

[48] Imperial College London. Ponder2 XML Befehle. `http://ponder2.net/cgi-bin/moin.cgi/Ponder2XML`.

[49] Imperial College London. Ponder2 Ponder-Policy. `http://ponder2.net/cgi-bin/moin.cgi/Ponder2UsingPolicies`.

[50] Imperial College London. Ponder2 Event-Types. `http://ponder2.net/cgi-bin/moin.cgi/Ponder2UsingEventTypes`.

[51] Hewlett Packard Labs Semantic Web Research. Unternehmenswebseite des HP Semantic Web Research Labs. `http://www.hpl.hp.com/semweb/`.

[52] P. A. Bonatti, D. Olmedilla. Semantic Web Policies: Where Are We and What Is Still Missing? In *In Proceedings of the 2nd International Conference on Rules and Markup Language for the Semantic Web, RuleML-2006*, 2006.

[53] K. Barrett, J. Strassner, S. van der Meer, W. Donnelly. Determining the Feasibility of Policy Translation. In *Modelling Autonomic Communications Environments - MACE2007*, pages 95–113, 2007.

[54] H. Knublauch, R. W. Fergerson, N. Fridman Noy, M. A. Musen. The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications. In *International Semantic Web Conference*, pages 229–243, 2004.

[55] Dudenredaktion (Hrsg.). *Duden, die deutsche Rechtschreibung : auf der Grundlage der neuen amtlichen Rechtschreibregeln*. Dudenverlag, Mannheim [u.a.], 24., völlig neu bearb. und erw. Aufl. edition, 2006.

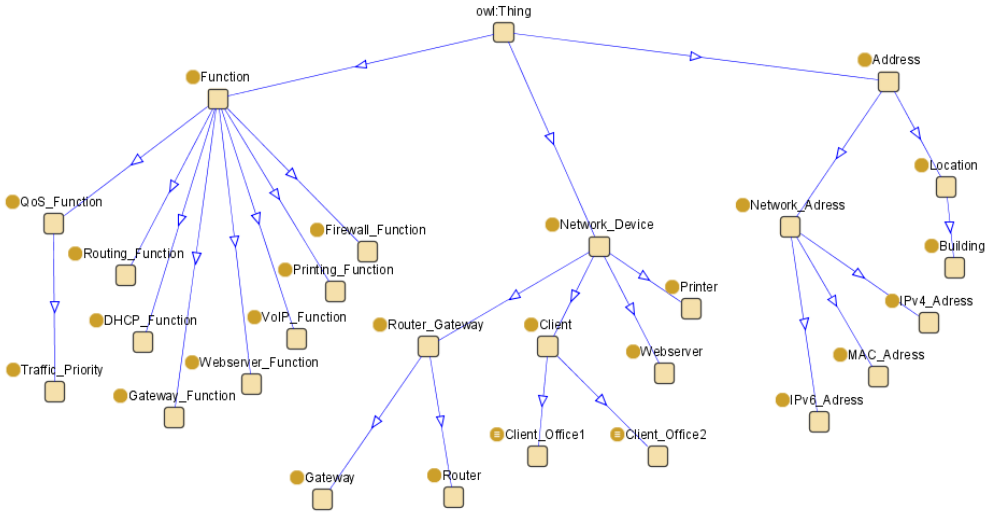
[56] J. Strassner, S. Balasubramaniam, K. Barrett, W. Donnelly, S. van der Meer. Bio-inspired Policy Based Management (bioPBNM) for Autonomic Communications Systems. In *Proceedings of the Seventh IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'06)*, pages 3–12, 2006.

[57] S. van der Meer, A. Davy, S. Davy, R Carroll, B. Jennings, J. Strassner. Autonomic Networking: Prototype Implementation of the Policy Continuum. In *The 1st International Workshop on Broadband Convergence Networks, 2006. BcN 2006*, pages 1–10, 2006.

A Anhang

A.1 UML Diagramm des Policy-Editors

A.2 OWL-Netztopologie



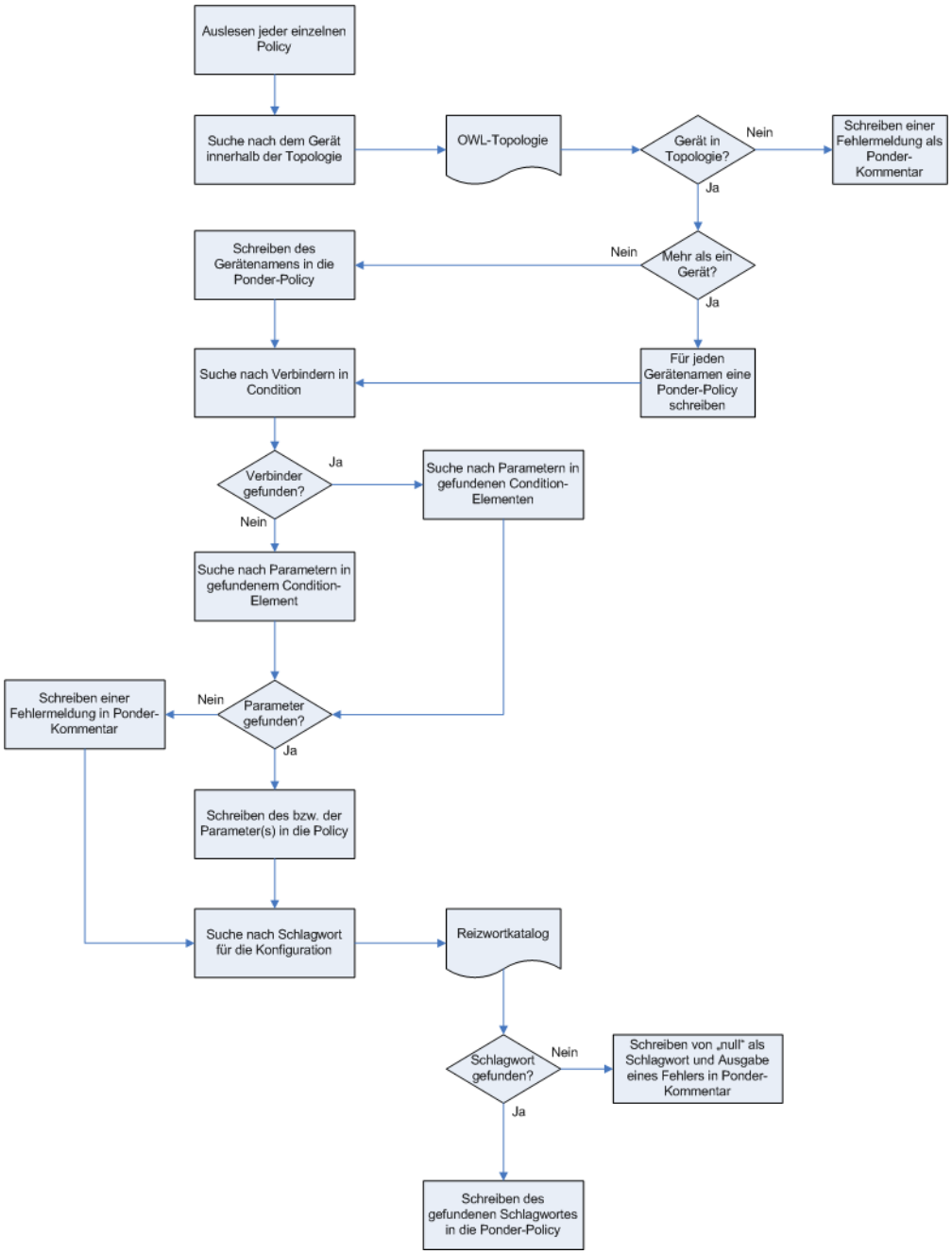
A.3 Ausschnitt der verwendeten OWL-Netztopologie im RDF-Format

Beschreibung der Instanz „Router_Office_1“ der Geräteklasse „Router“:

```
<rdf:Description rdf:about="#Router_Office_1">
    <rdf:type rdf:resource="#Router"/>
    <is_connected_to rdf:resource="#Printer_1"/>
    <is_connected_to rdf:resource="#Webserver_www_1"/>
    <is_connected_to rdf:resource="#Gateway_1"/>
    <is_connected_to rdf:resource="#Router_Office_2"/>
    <is_connected_to rdf:resource="#Client_Office1_Client1"/>
    <is_connected_to rdf:resource="#Client_Office1_Client2"/>
    <is_connected_to rdf:resource="#Client_Office1_Client3"/>
    <is_connected_to rdf:resource="#Client_Office1_Client4"/>
    <is_connected_to rdf:resource="#Client_Office1_Client5"/>
    <is_connected_to rdf:resource="#Client_Office1_Client6"/>
    <is_connected_to rdf:resource="#Client_Office1_Client7"/>
    <is_connected_to rdf:resource="#Client_Office1_Client8"/>
    <is_connected_to rdf:resource="#Client_Office1_Client9"/>
    <is_connected_to rdf:resource="#Client_Office1_Client10"/>
    <is_connected_to rdf:resource="#Client_Office1_Client11"/>
    <is_connected_to rdf:resource="#Client_Office1_Client12"/>
    <is_connected_to rdf:resource="#Client_Office1_Client13"/>
    <is_connected_to rdf:resource="#Client_Office1_Client14"/>
    <is_connected_to rdf:resource="#Client_Office1_Client15"/>
    <is_connected_to rdf:resource="#Client_Office1_Client16"/>
    <is_connected_to rdf:resource="#Client_Office1_Client17"/>
```

```
24      <is_connected_to rdf:resource="#Client_Office1_Client18"/>
25      <is_connected_to rdf:resource="#Client_Office1_Client19"/>
26      <is_connected_to rdf:resource="#Client_Office1_Client20"/>
27      <is_connected_to rdf:resource="#Client_Office1_Client21"/>
28      <is_connected_to rdf:resource="#Client_Office1_Client22"/>
29      <is_connected_to rdf:resource="#Client_Office1_Client23"/>
30      <is_connected_to rdf:resource="#Client_Office1_Client24"/>
31      <is_connected_to rdf:resource="#Client_Office1_Client26"/>
32      <is_connected_to rdf:resource="#Client_Office1_Client25"/>
33      <is_connected_to rdf:resource="#Client_Office1_Client27"/>
34      <is_connected_to rdf:resource="#Client_Office1_Client28"/>
35      <is_connected_to rdf:resource="#Client_Office1_Client29"/>
36      <is_connected_to rdf:resource="#Client_Office1_Client30"/>
37      <has_function rdf:resource="#isFirewall"/>
38      <has_adress rdf:resource="#IPv4_Adress_router1"/>
39      <has_function rdf:resource="#isRouter"/>
40      <has_function rdf:resource="#hasPriority"/>
41    </rdf:Description>
```

A.4 Ablauf der Transformation einer Policy in Ponder



B Handbuch

Dieses Handbuch beschreibt die Benutzung und einige Anwendungsfälle des Policy-Editors.

B.1 Start des Programms - Datei-Auswahl

Um den Policy-Editor zu starten, genügt ein Doppelklick auf die ausführbare Jar-Datei „PolEd.jar“. Zum Programmstart bietet sich dem Nutzer das Bild wie in Abbildung B.1 dargestellt:

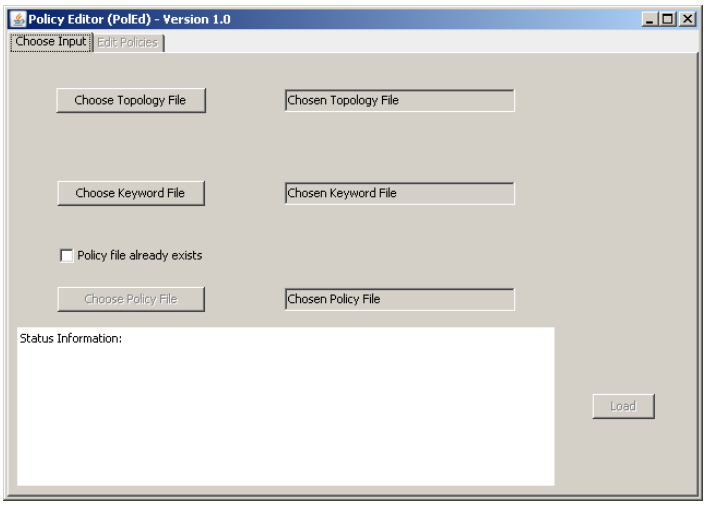


Abbildung B.1: Ansicht nach dem Öffnen des Policy-Editors

Zu beachten sind die deaktivierten Elemente „Load“-Knopf, sowie „Editor“-Reiter (siehe Abbildung B.2). Diese werden erst aktiviert, nachdem eine OWL-Netztopologie und eine Keyword-Datei (Reizwortkatalog) ausgewählt wurden.

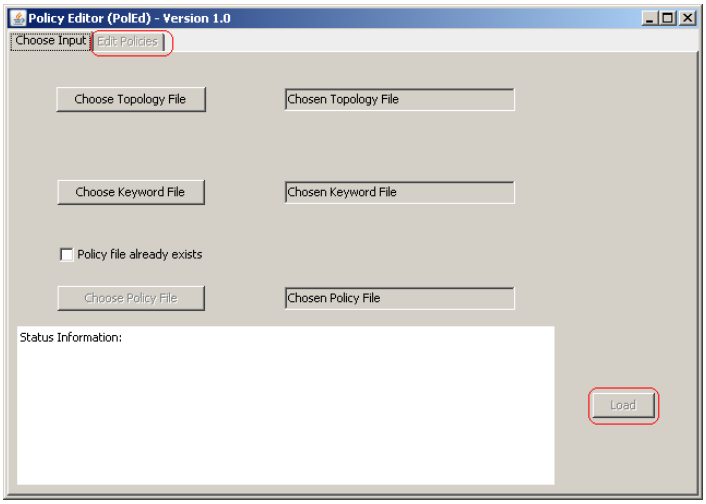


Abbildung B.2: Deaktivierte Elemente

Wählen Sie zunächst eine Topologie-Datei.

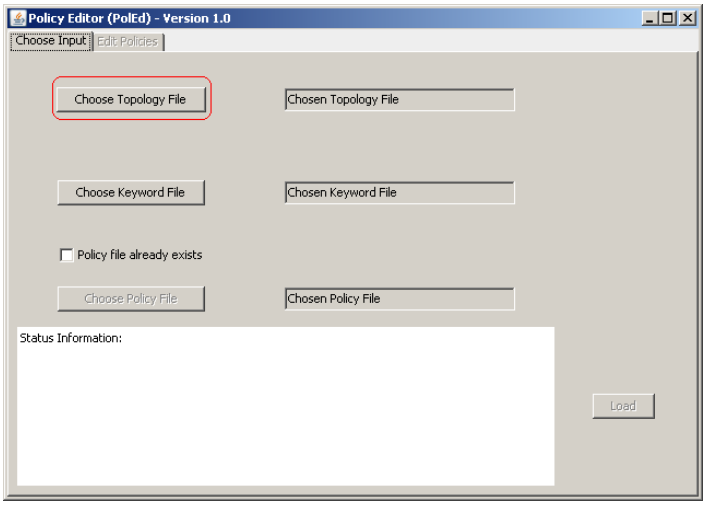


Abbildung B.3: Auswahl der Topologie

Nach Druck auf den „Choose Topology File“-Button öffnet sich ein Fenster zur Dateiauswahl. Beachten Sie, dass nur Dateien der Endung „owl“ zulässig sind.

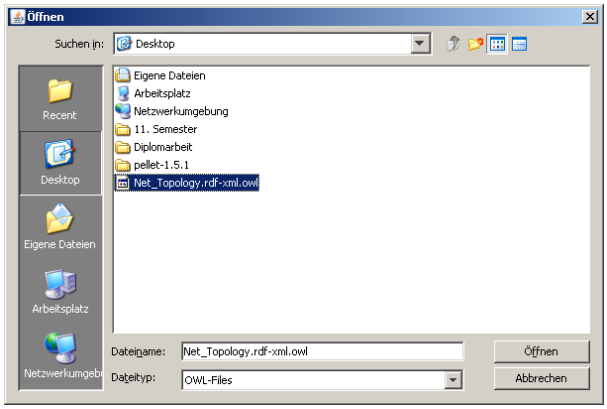


Abbildung B.4: Datei-Auswahl-Fenster

Nach Auswahl der Datei wird dies im Hauptfenster bestätigt, wie Abbildung B.5 zeigt.

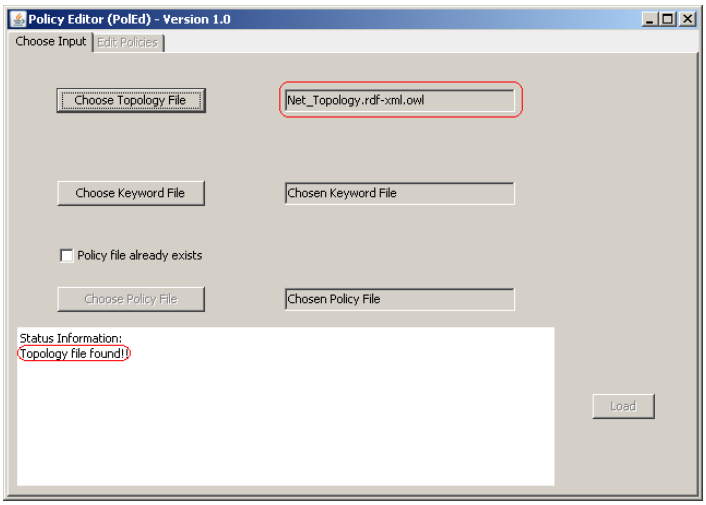


Abbildung B.5: Topologie ausgewählt

Die Auswahl des Reizwortkataloges erfolgt analog, wobei hier keine Einschränkung bezüglich der Dateiendung vorliegt. (Abbildung B.6) Beim Anlegen bzw. Ergänzen eines Reizwortkataloges achten Sie bitte auf die interne Formatierung gemäß dem Schema:

- Pro Zeile ein Reizwort [Leerzeichen] gefolgt vom zugehörigen Netzwerkgerät,
- als Trennung zwischen Device- und Main-Area mindestens ein „-“,
- Pro Zeile ein Reizwort [Leerzeichen] gefolgt von der zugehörigen Aktion bzw. dem maschinenverständlichen Konfigurationspendant.

Ein Beispiel-Reizwortkatalog gestaltet sich wie folgt:

```
1  Webcam Router
2  Voip Gateway
3  Address Client
4  -Main-Keyword-Area-
5  clock time
6  time time
7  ip IP
8  address IP
9  users usercount
10 shutdown shutdown
11 priority priority
12 open port
13 close port
```

Ist ein Reizwortkatalog gewählt, wird dies vom Programm bestätigt. Ab jetzt lässt sich auch der „Load“-Button betätigen.

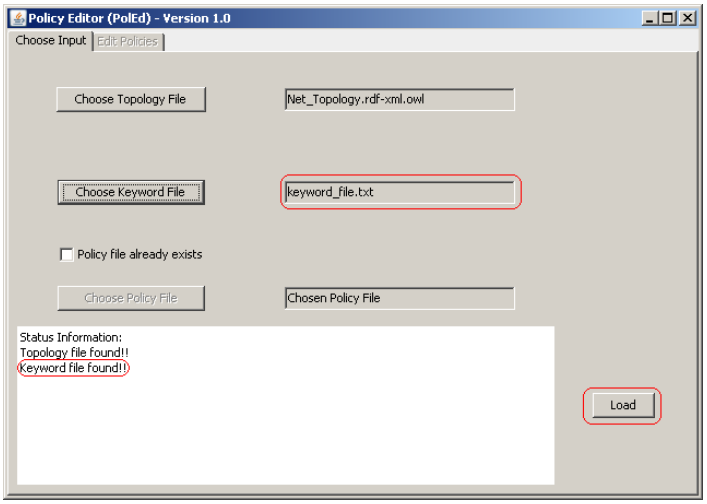


Abbildung B.6: Reizwort-Katalog ausgewählt

Zusätzlich besteht die Möglichkeit, bereits gespeicherte Policies aus vorherigen Sitzungen zu importieren. Die Dateiauswahl erfolgt analog der in Abbildung B.4 gezeigten Vorgehensweise.

Werden Policies abgespeichert erfolgt dies nach dem Schema:

- Zunächst die gesamte Policy,
- gefolgt von einem „||“-Trennsymbol,
- danach die Elemente Event, Condition und Action, jeweils durch „||“ getrennt,
- zuletzt das identifizierte Netzwerkgerät für die Policy.

Eine Policy-Datei könnte wie folgt aussehen:

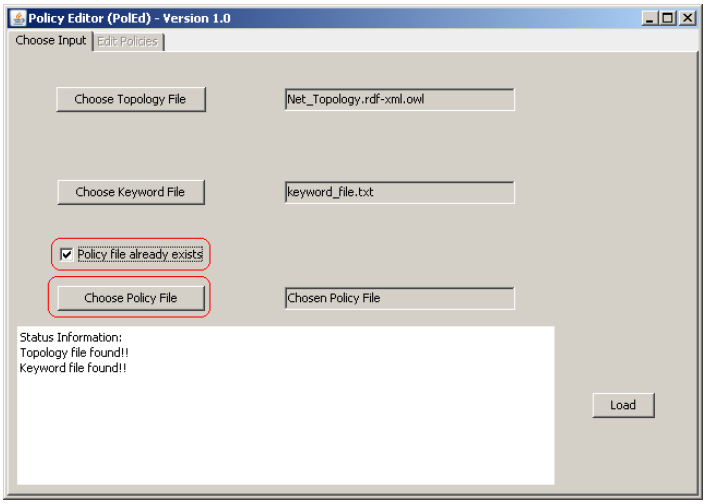


Abbildung B.7: Optional: Auswahl bereits bestehender Policies

```
1 Webcam connects, we have More than '50' users online $AND$ priority =
      'high', shutdown system || a Webcam connects|| we have More
     than '50' users online $AND$ priority = 'high'|| shutdown system
      || Router
2 Voip is started, the time = '18:30', open port '60' || a Voip is
     started || the time = '18:30' || open port '60' || Gateway
3 Voip is started, the time = '18:30', open port '9000' || a Voip is
     started || the time = '18:30' || open port '9000' || Client
4 Webcam policy, if More than '20' users are online, we want a system
     shutdown || a Webcam policy|| if More than '20' users are online
     || we want a system shutdown|| Router
```

Hier ist bereits die Eingabeformatierung von Policies mit einigen Besonderheiten zu erkennen, die später in Anhang B.2 noch erläutert werden.

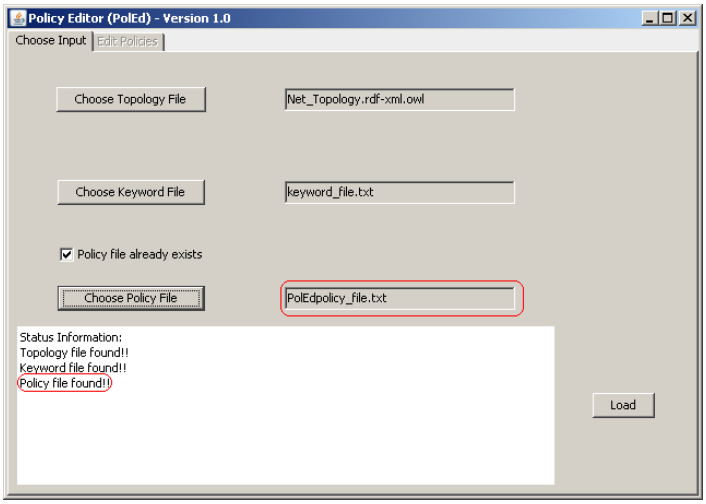


Abbildung B.8: Vorhandene Policies gewählt

Sind alle Datei-Auswahlen getroffen, betätigen Sie bitte den „Load“-Knopf. Die gefundenen Dateien werden eingelesen und der Policy-Editor wird aktiviert. Eine weitere Dateiauswahl ist jetzt nicht mehr möglich! Sollten sie eine falsche Datei gewählt haben und dies feststellen, nachdem der „Load“-Knopf betätigt wurde, müssen Sie das Programm neu starten.

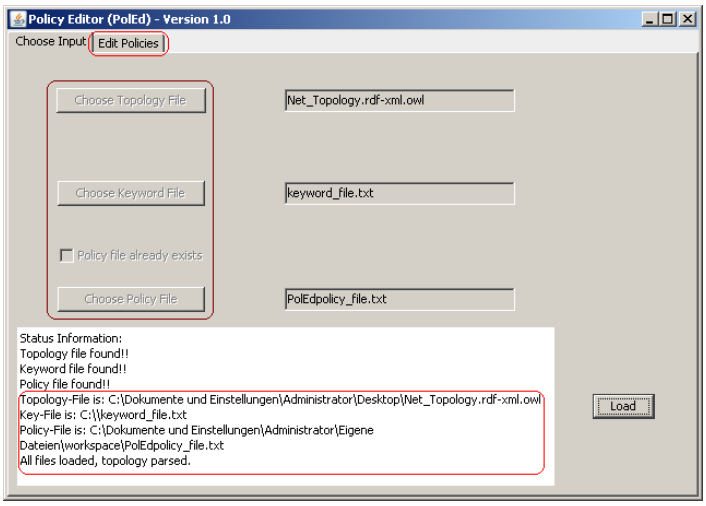


Abbildung B.9: Alle Dateien ausgewählt, Editor freigeschaltet

Fahren Sie nun mit dem Policy-Editor fort.

B.2 Der Policy-Editor

Nach der Dateiauswahl erfolgt die eigentliche Arbeit mit Policies. Hat der Benutzer keine Policies aus einer Datei importiert, bietet sich das Bild B.10. Wurden Policies eingelesen, sieht der Editor wie in Abbildung B.11 aus. Die importierten Policies wurden dann in die Tabelle übernommen. Sie haben dann bereits die Möglichkeit, vorhandene Policies zu bearbeiten bzw. zu löschen. Im oberen Bereich ist das Formularfeld markiert, hier werden neue Policies eingegeben, bzw. bereits vorhandene editiert. Im unteren Bereich befindet sich eine Status-Anzeige, die den Benutzer mit Informationen über den aktuellen Stand bzw. eventuelle Fehlermeldungen versorgt. Der große weiße Bereich beinhaltet, entweder direkt zu Anfang (wie in Abbildung B.11) oder nach Speicherung der ersten Policy, eine Tabelle der bereits formulierten Policies.

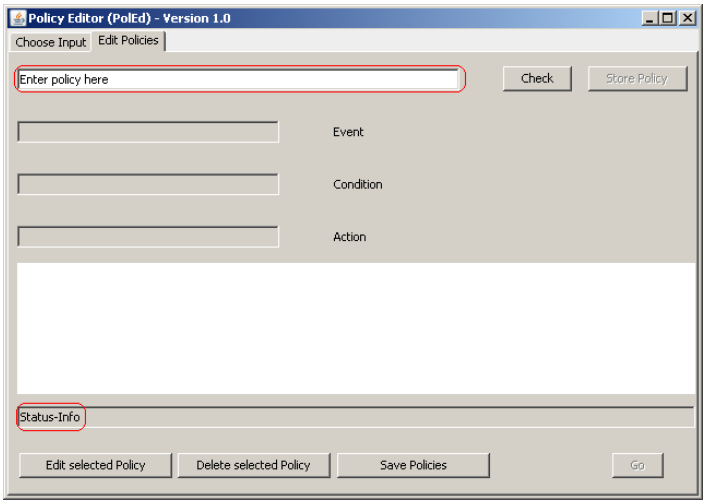


Abbildung B.10: Editor, Ansicht ohne geladene Policies

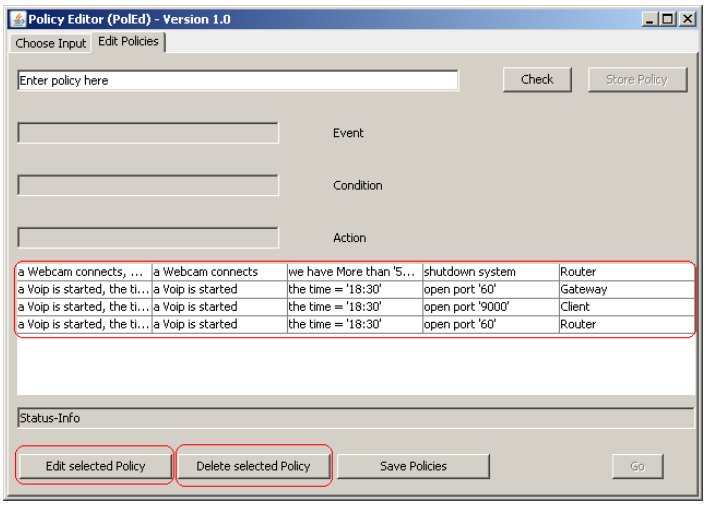


Abbildung B.11: Ansicht bei eingelesenen Policies

Eine neue Policy können Sie anlegen, indem Sie diese in das obere Formularfeld eintragen. Beachten Sie dabei bitte folgende Vorgaben:

- Der Editor „spricht“ nur Englisch!
- Da Policies aus drei Elementen (Event, Condition, Action) bestehen, werden diese durch ein Komma getrennt: „a Webcam connects *(Event)*, More than ’50’ users online *(Condition)*, shutdwon the system *Action*.
- In der Action bzw. der Condition können Parameter bzw. Bedingungen enthalten sein. Parameter werden in einfache Hochkommata gestellt. Operatoren - auch zur Verknüpfung zweier Bedingungen - werden entweder in Großbuchstaben oder als mathematische Operatoren eingegeben. Dabei sind folgende Operatoren zulässig:

- AND für und
- OR für oder
- NOT für nicht
- != für ungleich
- >= für grösser-gleich
- <= für kleiner-gleich
- More für größer
- Less für kleiner
- = für gleich

Haben Sie eine Policy entsprechend der o.g. Formatierung eingegeben, betätigen Sie den „Check“-Knopf. Die Policy wird eingelesen und in ihre Bestandteile zerlegt (Abbildung B.13). Wollen Sie die Policy nachträglich ändern, tun Sie dies in der noch im Formularfeld stehenden Policy und drücken Sie abermals „Check“.

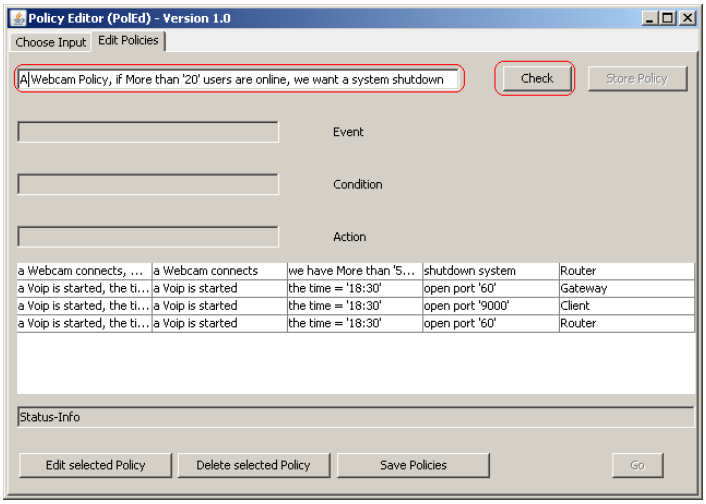


Abbildung B.12: Eingegebene Policy

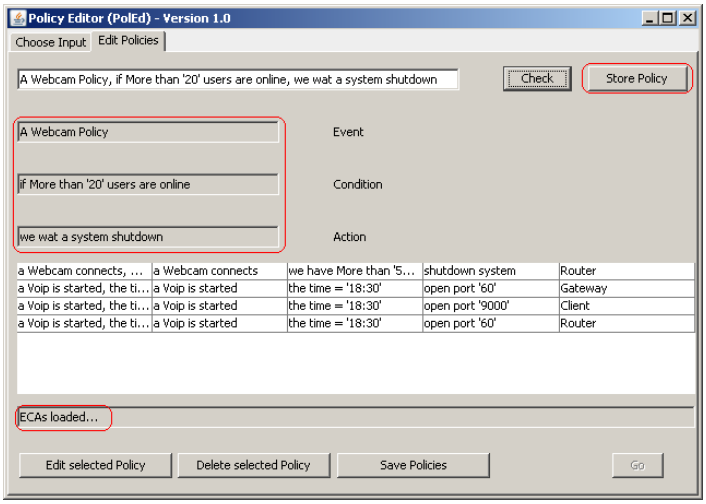


Abbildung B.13: Editor: nach Drücken des Check-Buttons

Sind Sie mit der Aufteilung der Policy einverstanden, betätigen Sie den „Store“-Knopf. Das Programm übernimmt die Policy in die Tabelle und bestätigt - im Falle eines Fundes innerhalb des Reizwortkataloges - bereits die Zuordnung zu einem Netzwerkgerät.(Abbildung B.14)

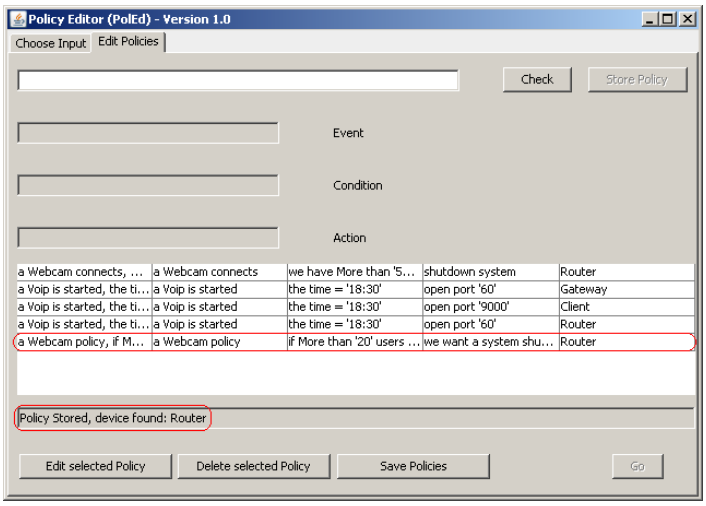


Abbildung B.14: Editor: Policy in die Tabelle übernommen

Zum Löschen einer Policy genügt eine Auswahl der entsprechenden Policy aus der Tabelle und ein Druck auf „Delete selected Policy“. Die gewählte Policy wird aus der Tabelle entfernt und die Aktion bestätigt.

Soll eine bereits eingegebene Policy nachträglich verändert werden, wählen Sie die Policy aus der Tabelle aus und betätigen „Edit selected Policy“. Die gewählte Policy wird in das Formularfeld geladen und die Elemente in den entsprechenden Feldern dargestellt. Haben Sie die Policy nach Ihren Wünschen verändert, drücken Sie „Check“ und - nach erfolgreicher Übernahme der Daten in die ECA-Felder sowie Bestätigung

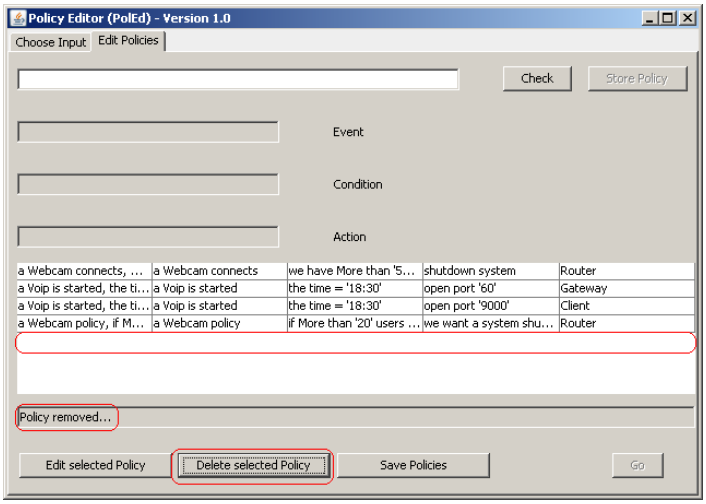


Abbildung B.15: Editor: Löschen einer Policy

des gefundenen Gerätes - auf „Store Policy“. Eine Meldung im Info-Panel bestätigt die veränderten Daten (Abbildung B.17).

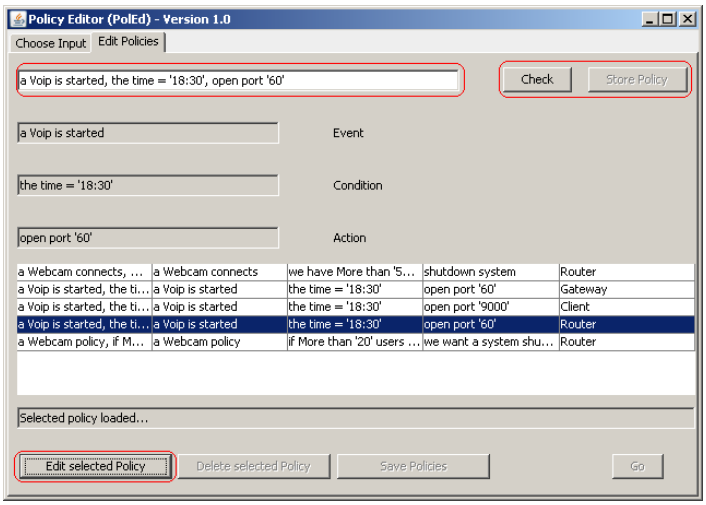


Abbildung B.16: Editor: Bearbeiten einer bereits eingegebenen Policy

Haben Sie die Eingabe sämtlicher Policies beendet und wollen zur Transformation in Ponder-Policies übergehen, bzw. möchten die bereits eingegebenen Daten für eine spätere Session speichern, drücken Sie auf „Save Policies“. Der Speicherort der Policies wird im Info-Panel dargestellt (Standard: Benutzer-Home-Verzeichnis mit dem Dateinamen „policy_file.txt“). Sobald dies geschehen ist, wird der „Go“-Knopf aktiviert (Abbildung B.18) und der zweite Refinement-Schritt kann gestartet werden (Abbildung B.19).

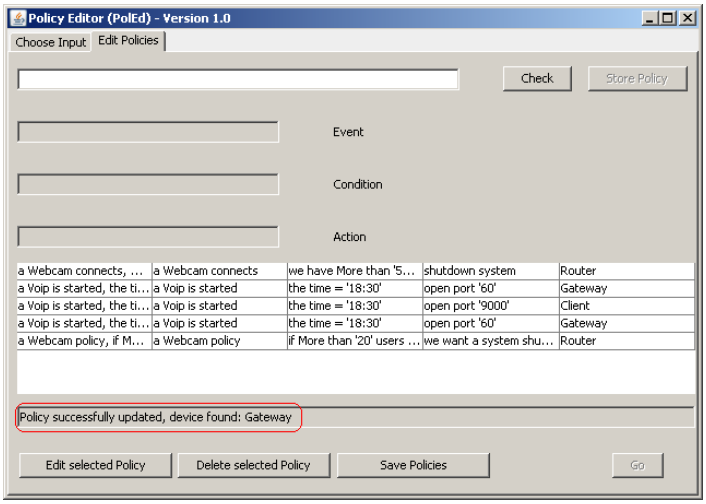


Abbildung B.17: Editor: Fertig bearbeitete Policy

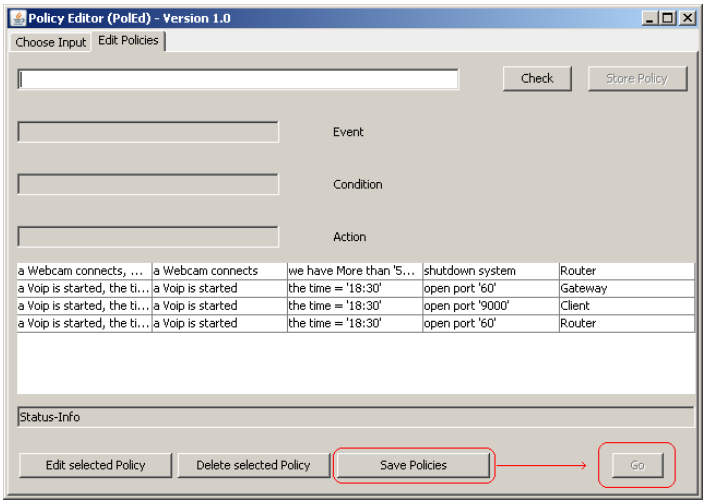


Abbildung B.18: Editor: Go-Button inaktiv

Nach Betätigung des „Go“-Buttons werden die in der Tabelle befindlichen Policies in Ponder2-Policies überführt. Eventuelle Fehler dabei finden sich als Kommentar, bzw. als Variablenname „null“ in der Ponder-Datei wieder. Ist der Vorgang abgeschlossen, wird dies mit dem Speicherort (Standard ist das Benutzer-Home-Verzeichnis mit dem Dateinamen „ponder_policies.txt“) bestätigt und der Vorgang ist abgeschlossen.

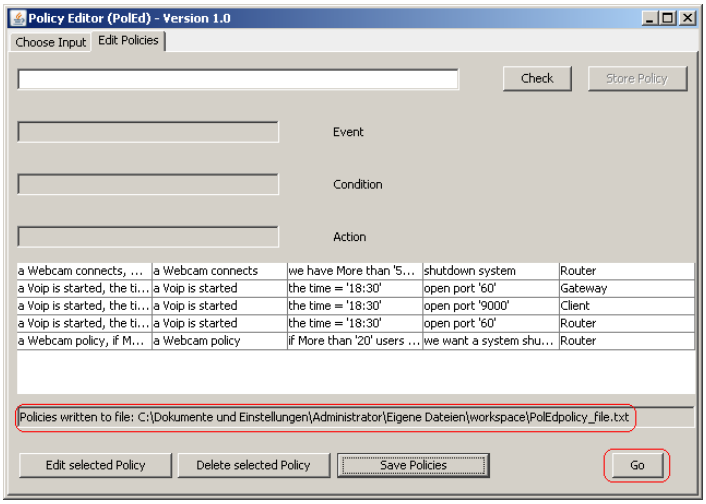


Abbildung B.19: Editor: Go-Button aktiviert

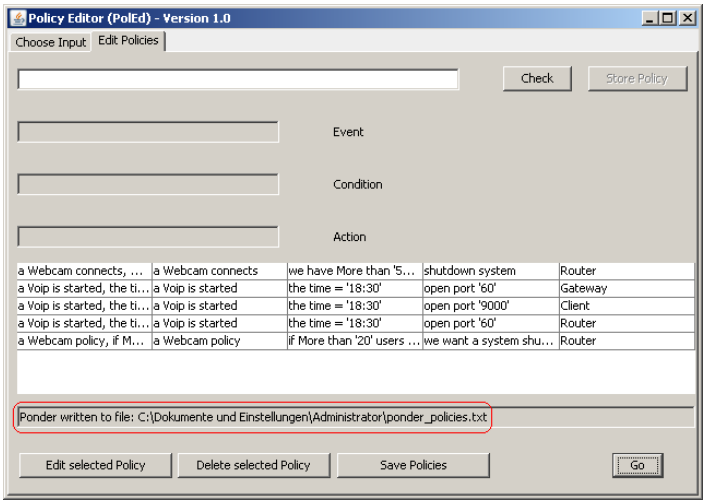


Abbildung B.20: Editor: Ponder-Transformation abgeschlossen.

Sie können das Programm nun schließen und die Ponder-Policies weiter verarbeiten bzw. in einem Editor Ihrer Wahl betrachten.