Energieeffizienz in Workflowsystemen

Technische Universität Dresden
Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Diplomarbeit

Energieeffizienz in Workflowsystemen

WEAT - Workflow-driven Energy Auto Tuning

Georg Püschel

Hochschullehrer: Prof. Dr. rer. nat. habil. Uwe Assmann
Betreuender Mitarbeiter: Dipl. Inf. Sebastian Richly

Eingereicht am: 24. März 2011

Inhaltsverzeichnis

1 Einführung **11**
1.1 Zielstellung . . . 14
1.2 Übersicht . . . 14

2 Grundlagen und weiterführende Literatur **15**
2.1 Geschäftsprozesse . . . 16
2.2 Qualität, Metrik und Energieeffizienz . . . 18
2.3 CoolSoftware . . . 20
2.3.1 Das Cool Component Model . . . 21
2.3.2 Vertragsmodellierung . . . 23
2.3.3 Architektur . . . 25
2.4 Weitere Ansätze . . . 26

3 Konzept **27**
3.1 Übersicht . . . 28
3.2 Modelle . . . 30
3.2.1 Komponentenmodell . . . 30
3.2.2 Einheitensystem und Mathematisches Modell . . . 32
3.2.3 Vertragsmodell . . . 35
3.2.4 Verhaltensmodell . . . 39
3.2.5 Workflowmodell . . . 44
3.2.6 Planmodell . . . 44
3.3 Verarbeitungskette . . . 45
3.3.1 Vereinigung paralleler Workflows . . . 46
3.3.2 Partitionierung des Workflows . . . 46
3.3.3 Contract Negotiation . . . 48
3.3.4 Bewertung der Varianten . . . 49
3.3.5 Simulation . . . 50
3.3.6 Rekombination der Partitionen . . . 51
3.3.7 Auswahl des optimalen Plans . . . 51

4 Implementierung **53**
4.1 Technologieüberblick . . . 55
4.2 Implementierung des Einheitenmodells . . . 56
4.3 Verwaltetes Komponentensystem . . . 56

4.4 Modelle und Parser 58
4.5 Planungskomponente 59
4.6 Integration von OSPP 60

5 Evaluation **63**
5.1 Anwendungsfall 64
5.2 Versuchsaufbau 65
5.3 Verbrauchsmessung 66
5.4 Hardwareressourcen 66
5.4.1 Prozessoren 67
5.4.2 Festplatten 71
5.4.3 Netzwerkadapter 74
5.5 Schnittstellen und Workflowmodell 74
5.6 Softwarekomponenten 75
5.7 Berechnungs- und Messergebnisse 79

6 Fazit und Ausblick **83**

A Algorithmen zur Simulation **89**

B Benutzeroberfläche **97**

Abbildungsverzeichnis

2.1 Referenzmodell Workflowmanagementsystem (nach [Coa95]) 17
2.2 Auto-Tuning Verarbeitungskreislauf (nach [DDF+06]) 21
2.3 CCM `structure`-Paket (nach [GWS+10a]) 22
2.4 CCM `behavior`-Paket (nach [GWS+10a]) 22
2.5 CCM `variation`-Paket (nach [GWS+10a]) 22
2.6 THEATRE-Architektur (nach [GWS+10a, GWS+10b]) 25
2.7 THEATRE-Verarbeitungskreislauf 26

3.1 Konzeptüberblick 29
3.2 Verwendete Modelle 31
3.3 Komponentenmodell 31
3.4 Beispiel Komponentenmodell 33
3.5 Einheiten-, Größen- und Dimensionsmodell 34
3.6 Qualitätenmodell 34
3.7 Mathematisches Modell 35
3.8 Vertragsmodell 36
3.9 Verhaltensmodell 40
3.10 Leistungsaufnahme für Beispiel 3.3 43
3.11 Workflowmodell 44
3.12 Planmodell 45
3.13 Vereinigung paralleler Workflows 46
3.14 Ein Workflow-Task und dessen Parallel-Tasks 47
3.15 Eine mögliche Partitionierung 47

4.1 Architektur 54
4.2 Schnittstellen des Komponentensystems 57
4.3 WEAT-Schnittstelle 59
4.4 WEAT-Architektur 61

5.1 Anwendungsfall Workflow 65
5.2 Versuchsaufbau 65
5.3 Anwendungsfall – Workflowmodell 75
5.4 Mögliches Berechnungsergebnis des Anwendungsfalls 81

B.1 Benutzeroberfläche Teil I 98
B.2 Benutzeroberfläche Teil II 99

Listings

2.1 ECL-Charakteristik (nach [GWS+10a]) 24
2.2 ECL-Komponentenprofil (nach [GWS+10a]) 24
2.3 ECL-Ressourcenprofil (nach [GWS+10a]) 24

3.1 ECL'-Profil der Ressource `RA` 38
3.2 ECL'-Profil der Komponente `C` 38
3.3 Verhaltensdefinition Beispiel 43
3.4 Rekursiver Algorithmus Contract Negotiation 48

5.1 ECL'-Profile der Prozessoren 69
5.2 Virtuelle Komponenten zur Modellierung der Grundlast 69
5.3 ECL'-Profile zur Modellierung der Grundlast 69
5.4 Verhaltensmuster der Prozessoren 70
5.5 ECL'-Profile der Festplatten 72
5.6 Verhaltensmuster der Festplatten 72
5.7 Virtuelle Komponenten – Netzwerkkarte 73
5.8 Verhaltensmuster der Netzwerkadapter 73
5.9 ECL'-Profil der Decoder-Komponente 77
5.10 Verhaltensmuster der Decoder-Komponente 77
5.11 Verhaltensmuster der Recognizer-Komponente 78
5.12 ECL'-Profil der VideoFinder-Komponente 78
5.13 Verhaltensmuster der VideoFinder-Komponente 79

A.1 Transformationsalgorithmus 90
A.2 Schedulingalgorithmus 92
A.3 Simulationsalgorithmus 94

Syntaxdefinitionen

3.1 Modellierungssprache für Komponenten 32
3.2 Grammatik für Qualitäten, Maßeinheiten und Dimensionen 33
3.3 Grammatik für mathematische Ausdrücke 35
3.4 EBNF-Definition von ECL' 36
3.5 EBNF-Definition der Behavior Language 42

Tabellen

2.1 Qualitätsmerkmale nach ISO/IEC 9126 19

5.1 Messungen – Prozessoren 68
5.2 Herstellerangaben und Messungen – Festplatten 71
5.3 Messungen – Netzwerkadapter 73
5.4 Softwareschnittstellen und Qualitäten 74

Erklärung

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst habe und ausschließlich angegebene Quellen und Hilfsmittel benutzt wurden. Stellen, deren Wortlaut oder Sinn anderen Werken entnommen sind, wurden in jedem Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht. Dies gilt ebenso für Tabellen und Abbildungen. Diese Arbeit hat in dieser oder einer ähnlichen Form noch nicht im Rahmen einer anderen Prüfung vorgelegen.

____________________ (Ort, Datum)

____________________ (Unterschrift)

1. Einführung

Die Verbesserung der Energieeffizienz von IT-Systemen wird zunehmend als Aufgabe in Unternehmen anerkannt. Zum einen sind klimapolitsche Einflussnahmen Ursache dieser Entwicklung, andererseits wirkt durch die einsetzende Ressourcenknappheit ein wachsender Preisdruck, der die Entscheidungsträger dazu drängt, ihre Rechenanlagen zu optimieren. In der IT-Industrie werden Maßnahmen zur ressourcensparenden Entwicklung, Produktion, Nutzung und Entsorgung unter dem Begriff Green IT zusammengefasst. Da ein signifikanter Anteil industrieller Anwendungen auf größeren Systemen betrieben wird, richtet sich das Hauptaugenmerk der Green IT auf Entwicklungen im Servermarkt. Obwohl Green IT durch die Verantwortungsübernahme der Unternehmen auch einen positiven Einfluss auf deren relative Positionierung gegenüber anderen Wettbewerbern hat und starke Kosteneinsparungspotentiale vorhanden sind, bestehen weiterhin umfangreiche Verbesserungsmöglichkeiten.

Einen Versuch den Energieverbrauch der IT-Wirtschaft in den Vereinigten Staaten darzustellen unternahm 2007 die U.S. Environmental Protection Agency (EPA) in [EPA07]. Das Ergebnis der Studie offenbarte eine geschätzte Verdopplung des jährlichen Energiebedarfs von 2000 bis 2006 auf 61 Milliarden kWh. Zudem setzten sich die Autoren mit Vorhersagen bis zum Jahr 2011 bei Eintritt unterschiedlicher Szenarien (auf Grundlage erwarteter technischer Verbesserungen) auseinander. Insofern das historische Verbrauchswachstum erhalten bliebe, prognostizierten sie die Verdopplung der annualen Leistungsaufnahme der US-IT-Wirtschaft bis 2011. Im besten Fall wäre eine Einsparung von 55% möglich, wenn sofort eine aggressive Umsetzung der bekannten Ausrüstungs- und Infrastrukturoptimierungsmaßnahmen eingeleitet wurden wäre. Es kann heute, am Ende dieses Prognosezeitraumes, davon ausgegangen werden, dass letzterer Fall nicht eingetreten ist.

Die EPA schlug neben der Verbesserung der Hardware auch einige Maßnahmen für Softwaresysteme vor. Insbesondere betrachteten die Studienverantwortlichen die Entwicklung der Virtualisierung mit Sorge, da diese, im Gegensatz zur direkten Ausführung von Software, zusätzliche Systemressourcen («Overhead») in Anspruch nimmt. Heute kann jedoch davon ausgegangen werden, dass Virtualisierung aufgrund der gemeinsamen Verwendung von Hardware durch mehrere virtuelle Systeme sogar Energie eingespart [VMw08]. Ebenfalls sahen die Autoren Potential im effektiven Einsatz von Mehrkerntechnologien und der Reduktion der Ressourcenzugriffe durch die Software. Ein weiterer Punkt war zudem die Möglichkeit, auf Unregelmäßigkeiten des bekannten Workloads auf High-Performance-Rechnern zu reagieren.

Die Verringerung von Ressourcenzugriffen (Polling) beschreibt auch [Gar07]. Vor allem das Betriebssystemes solle diese Aufgabe wahrnehmen, da es durch Caching-Maßnahmen auf die Interaktion zwischen Hard- und Software Einfluss nehmen kann. Für Serveranwendungen jedoch tritt ein anderes Problem in den Vordergrund: Sie müssen ihre Verfügbarkeit ständig sicherstellen und erlauben deshalb keinerlei Schlafzustand. Diese Diagnose stellen Barroso und Hölzle in [BH07]. So wird auch gezeigt, das Server weiterhin die Hälfte ihrer maximalen Leistungsaufnahme benötigen, wenn nahezu keine Last mehr auf das System einwirkt. Als Ziel definieren sie deshalb *Energy Proportional Computing*, also eine konstant hohe Energieeffizienz in sämtlichen Auslastungsgraden. Dazu ist der Ausbau der Spannungsregulierung von Geräten notwendig, vor allem aber die Einführung

von aktiven Low-Power-States. Im Gegensatz zu passiven Ruhezuständen können in diesen immer noch Grundaufgaben des Systems erfüllt werden, ohne die Verfügbarkeit der Dienste zu beeinflussen.

Hardware-Performancezustände können durch das *Advanced Configuration and Power Interface (ACPI)*, früher *Advanced Power Management (APM)*, reguliert werden. Die auslastungsgetriebene Steuerung der Ressourcen bezieht jedoch nur Informationen der Gegenwart in die Entscheidung über einen Zustandswechsel ein. Viel höhere Effizienz könnte durch einen Blick in die Zukunft erreicht werden. Dazu ist die detaillierte Kenntnis der durchzuführenden Aufgabe notwendig. David J. Brown beschreibt in [BR10] eine mögliche Vorgehensweise. Anders als in traditionellen Systemen soll dabei die Performance einer Anwendung nicht mehr im Vordergrund stehen. Durch Definition von nicht-funktionellen Bedingungen kann anstattdessen eine Mindest-Performance oder Qualität definiert werden. Auf dieser Grundlage kann die Suche nach der energieeffizientesten Konfiguration eines Systems, welche die Erfüllung funktionaler und nicht-funktionaler Bedingungen festlegt, als Optimierungsproblem verstanden werden. Dazu muss das System folgende drei Fähigkeiten besitzen:

1. Die Konstruktion eines *Energiemodells* (Power Model), in dem Aussagen getroffen werden, wie und wo Energie verbraucht wird und wie das System zu manipulieren ist.

2. Ein *Aufwandsmodell* zur Bestimmung, wie ein Workload oder Task das System in Anspruch nimmt, und welche Bedingungen (Constraints) dabei eingehalten werden müssen. Diese Informationen sind durch Monitoring oder explizite Modellierung zu gewinnen.

3. Ein Algorithmus, der durch heuristische Bewertungen oder vollständige Analyse eine optimale Systemkonfiguration erzeugt. Dies wird auch als *Kapazitätsplanung* bezeichnet.

Die CoolSoftware-Forschungsgruppe verfolgt diese Idee in ihrem Ansatz *Energy Auto Tuning (EAT)* [GWS+10a] weiter. Dabei werden nicht nur Performance-Bedingungen definiert, sondern darüber hinaus Anforderungen beliebiger Qualitätscharakteristiken. Es wurde bereits ein Komponentenmodell sowie eine Architektur konstruiert auf deren Grundlage die energieeffiziente Kapazitätsplanung aufbauen wird. Die konkrete Implementierung des Systems steht noch aus.

Im praktischen, industriellen Einsatz kann ein solches EAT-System vor allem Energie einsparen, wenn es in der IT-Infrastruktur des Unternehmen an einem zentralen Knotenpunkt verwendet wird. Dort können ihm sämtliche Informationen über die betriebenen Ressourcen und Komponenten zur Verfügung gestellt werden und es kann, insofern diese steuerbar sind, die gewonnenen Entscheidungen bei der Koordination des Gesamtsystems berücksichtigen. Eine häufig im Unternehmen eingesetzte Kompositionsplattform für Softwaredienste ist das *Workflow Management System*. Dieses stellt darüber hinaus noch eine weitere Funktion zur Verfügung: Die Komposition von Anwendungen schließt immer einen definierten Workflow ein, der auch als *strukturierter Kontrollfluss* betrachtet

werden kann. Somit steht für das EAT-System eine weitere Wissengrundlage nach dem Planungszeitpunkt in Form zukünfiger Workloads bereit. Dies führt zu einer genaueren Planung der dynamischen Rekonfiguraion der Komposition und der manipulierbaren Soft- und Hardware. Die Energieeffizienz kann so um einen bedeutenden Schritt verbessert werden.

1.1. Zielstellung

Um eine Workflow energieeffizient auszuführen, muss eine große Menge von möglichen Konfigurationsverläufen verglichen werden; ein einzelner Verlauf beschreibt dabei die Erfüllung der funktionalen und nicht-funktionalen Bedingungen in jedem Teil-Workflow. Es ist eine Entscheidung zu treffen, in welchen Teil-Workflows (Partitionen) des bereits sicheren und deterministischen vorhersagbaren Planungszeitraumes welche Konfiguration optimal ist. Der Workflow kann dazu in eine Menge Partitionen unterteilt werden, die jeweils zuerst einzeln und im Anschluss in gemeinsamer Wirkung betrachtet werden. Je nach funktionellen Eigenschaften des Workflows und der Präferenzen des Benutzers kann für jede dieser Partitonen eine Konfiguration (Variante) bestimmt werden, die unter allen betrachteten die höchste Energieeffizienz verursachen wird. Die Vorhersage des Energieverbrauches benötigt System-, Qualitäts- und Verhaltensmodelle sowie eine anschließende Simulation der voraussichtlichen Vorgänge während des betrachteten Intervalls. Der simulierte Verbrauch kann anschließend zur Bewertung der Auswahl mit der im Modell definierten Qualität in Beziehung gesetzt werden, so dass letztendlich eine optimale Konfiguration vorliegt.

Der hier verfolgte Ansatz hat das Ziel, einen strukturierten Kontrollfluss auf Grundlage eines durch ein Workflow Management System definierten Workflows in ein Energy-Auto-Tuning-System zu integrieren. Dabei wird zunächst ein Konzept für ein prototypisches EAT-System entwickelt, welches mit zusätzlichen Funktionen ausgestattet wird, die die Umsetzung sämtlicher oben erläuterter Aufgaben ermöglicht. Anschließend soll mithilfe des entwickelten Ansatzes ein funktionierendes Workflow-getriebenes, adaptives System implementiert sowie die zur Modellierung notwendige Methodik an einem vereinfachten Anwendungsfall veranschaulicht werden.

1.2. Übersicht

Diese Arbeit gliedert sich wie folgt: In Kapitel 2 werden mehrere Arbeiten vorgestellt, die eine grundlegende Wissensbasis zum Verständnis des Problems und seiner Lösung zur Verfügung stellen, oder bereits Versuche im gleichen Problemfeld durchführten. Das Kapitel 3 zeigt den konkreten Lösungsansatz anhand notwendiger Modelle und Verarbeitungsschritte. Anschließend wird dieses Konzept in Kapitel 4 mithilfe verschiener Java-basierter Technologien implementiert. Kapitel 5 zeigt einen Versuch auf Grundlage eines konkreten Anwendungsfalls und eines methodischen Vorgehens, ein zugehöriges Modell zu entwickeln. Zuletzt, im Kapitel 6, werden einige ungelöste Probleme und zukünftige Fragestellungen aufgezeigt.

2. Grundlagen und weiterführende Literatur

In diesem Kapitel werden einige Konzepte beschrieben, die Grundlage des zu entwickelnden Ansatzes sein werden oder deren Ziel es ist, ähnliche Problemstellungen zu lösen. Anfangs werden *Geschäftsprozesse* und *Workflows* erläutert. Letztere sind die Berechnungsgrundlage des Planungsvorganges; sie repräsentieren den auszuführenden Kontrollfluss, der Ursache eines Aufwandes ist, welcher bei der Effizienzberechnung minimiert werden soll. Die zu maximierende Komponente des Effizienzverhältnisses ist die angebotene Qualität des Systems während der Ausführung des Workflows. Deshalb wird im anschließenden Absatz in die Themen *Qualität, Metrik und Energieeffzienz* eingeführt. Es folgt eine Beschreibung des Ansatzes der *CoolSoftware*-Forschungsgruppe, deren Modellsatz Vorbild bei der Gestaltung neuer, angepasster Modelle im Kapitel 3 sein wird. Am Ende wird kurz auf einige weitere Ansätze eingegangen, die mit der Berechnung effizienter Systemkonfigurationen in Verbindung gebracht werden können, aber nicht das gleiche Aufgabenspektrum beinhalten wie das CoolSoftware-Projekt.

2.1. Geschäftsprozesse

Ein Unternehmen muss, um Umsatz zu erwirtschaften, Dienstleistungen oder produzierte Güter an Kunden anbieten und absetzen. Die dazu notwendigen Tätigkeiten werden in ihrer zeitlichen Abfolge als Wertschöpfungskette bezeichnet. Zur Ablauforganisation werden dabei Geschäftsprozesse eingesetzt. Ein Geschäftsprozess umfasst einen wertmehrenden Vorgang bezüglich eines Produktes oder einer Diensleistung für einen Kunden aus administrativen, operativen und unterstützenden Aktivitäten oder Aufgaben. Wird der Geschäftsprozess durch einen Modellierer definiert, kann er als Handlungsanweisung und Entscheidungsgrundlage der Organisationseinheiten des Unternehmens dienen. Zur Identifikation, Gestaltung, Dokumentation, Implementierung, Steuerung und Verbesserung von Geschäftsprozessen wird das Geschäftsprozessmanagement [vBR10] eingesetzt.

Die Modellierung von Geschäftsprozessen (engl.: Business Process Modeling (BPM)) beschreibt rechnergestützt den Prozess, so dass er daraufhin analysiert oder verbessert werden kann. Zudem kann sie zur Planung neuer Prozesse verwendet werden. Um den Prozess darzustellen werden verschiedene, meist grafische, Notationen eingesetzt, wie zum Beispiel BPMN [Obj09].

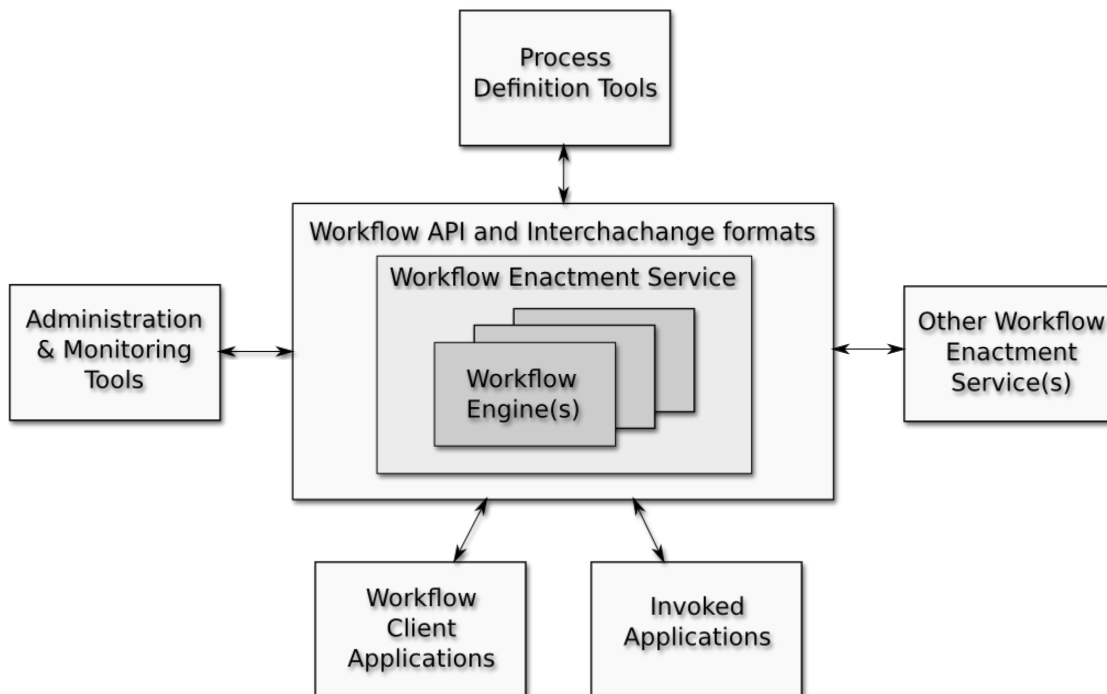
Workflows

Die Durchführung und Überwachung von Geschäftsprozessen kann durch Rechnersysteme unterstützt werden. Eine technische Repräsentation eines Prozesses wird als *Workflow* bezeichnet. Die Workflow Management Coalition[1] beschäftigt sich mit der Standardisierung von Workflows und Workflow-unterstützenden Systemen. Im *Workflow Reference Model* [Coa95] wird der Workflow definiert als:

Definition 2.1 (Workflow) *Die rechnergetriebene Unterstützung oder Automation eines Geschäftsprozesses als Ganzes.*

[1] http://www.wfmc.org

Abbildung 2.1. Referenzmodell Workflowmanagementsystem (nach [Coa95])



Zur Modellierung, Verwaltung und Ausführung von Workflows wird ein *Workflow Mangement System* (WfMS) verwendet. Auch dieses wird im Referenzmodell definiert:

Definition 2.2 (Workflow Mangement System) *Ein System, welches Workflows durch Verwendung von Software vollständig definiert, verwaltet und sie wie in der Workflowlogik beschriebenen Reihenfolge ausführt.*

Das WfMS vermittelt zwischen menschlichen oder technischen Akteuren, die einzelne Aufgaben das Workflows durchführen. Der Kontrollfluss des Workflows wird in einer *Process Definition* repräsentiert:

Definition 2.3 (Process Definition) *Die technische Repräsentation eines Prozesses, welche die manuelle Definition sowie die Workflowdefinition beinhaltet.*

Die Process Definition beschreibt das Workflowmodell. Durch das WfMS werden Instanzen dieses Modells ausgeführt. Zur Modellierung werden graphische oder textuelle Notationen eingesetzt. Beispiele sind die Web Service Business Process Execution Language (WS-BPEL) [OAS07] und XML Process Definition Language (XPDL) [2.108]. Ein Workflowmodell kann entweder vollständig *strukturiert* sein oder nur einige Regeln enthalten, die in bestimmten Situationen Kontrollflussstrukturen definieren. Wird vollständig auf eine vordefinierte Struktur verzichtet, so wird der Workflow als *Ad-Hoc-Worflow*

bezeichnet. Die WFMC beschreibt auch ein Architektur-Referenzmodell für Workflow Mangement Systeme. Dieses ist in Abbildung 2.1 dargestellt. Die zentrale Komponente der Architektur ist der *Workflow Enactment Service*.

Definition 2.4 (Workflow Enactment Service) *Ein Softwaredienst, der aus einer oder mehrerer Workflow-Engines bestehen kann und dazu dient Workflow-Instanzen zu erzeugen, zu verwalten und auszuführen. Anwendungen können durch diesem Dienst mithilfe des Workflow Programming Interface (WAPI) kommunizieren.*

Das WfMS kann durch *Client Applications* oder *Invoked Applications* mit Ressourcen interagieren. Client Applications sind *Worklist Handler*, sie verwalten die Aufgabenlisten der Benutzer. Aufgaben können durch Benutzeranwendungen verarbeitet und durchgeführt werden. Invoked Applications dagegen werden direkt vom WfMS aufgerufen, um eine Aufgabe durchzuführen. Durch WAPI können mehrere Enactment Services miteinandar kommunizieren – dies ermöglicht den Aufbau eines verteilten WfMS. *Process Definition Tools* sind Werkzeuge, die der Modellierung von Worklows dienen. Die Komponente *Administration & Monitoring Tool* bietet unterstützende Werkzeuge für Prozessverwaltung und -beobachtung. Während der Enactment Service vor allem zur Aktivierung es Workflows dient, benutzt er *Workflow Engines* zu deren Ausführung:

Definition 2.5 (Workflow Engine) *Eine Engine bietet eine Laufzeitumgebung für eine Workflow Instanz.*

Aufgaben einer Engine sind unter anderem die Interpretation und Ausführung der Process Definition, Instanz- und Sitzungsverwaltung für Benutzer sowie der Aufruf von Invoked Applications.

Open Service Process Platform

Das Workflow Mangement System Open Service Process Platform (OSPP) [HRR+08] bietet grundlegende Funktionen des beschriebenen Referenzmodells und implementiert darüber hinaus aktuelle Forschungsansätze des Workflowmanagements. Zu diesen gehören rollenbasierte [RGmS10] und reflexible Workflows durch Belief-Desire-Intention [RBA08]. OSPP wird im vorgestellten Ansatz als WfMS eingesetzt.

2.2. Qualität, Metrik und Energieeffizienz

Der Begriff *Effizienz* wird synonym mit dem des *Wirkungsgrades* verwendet und bezeichnet das Verhältnis der Größe einer bestimmten *Leistung* zur Größe eines dafür notwendigen *Aufwandes*. *Energieeffizienz* beschreibt das Verhältnis einer bestimmten Leistung zum *Energieverbrauch*. Die Leistung eines Vorganges in einem Informationssystem kann verstanden werden als die *Qualität* des Vorganges. Der Verhältniswert der Größe des Energieverbrauches ist die Summe der durch den Vorgang ausgelösten Einzelverbräuche in den jeweils genutzten Systemressourcen.

Tabelle 2.1 Qualitätsmerkmale nach ISO/IEC 9126

Merkmal	**Untermerkmale**
Funktionalität	Angemessenheit, Richtigkeit, Interoperabilität, Sicherheit, Ordnungsmäßigkeit
Zuverlässigkeit	Reife, Fehlertoleranz, Wiederherstellbarkeit, Konformität
Benutzbarkeit	Verständlichkeit, Erlernbarkeit, Bedienbarkeit, Attraktivität, Konformität
Effizienz	Zeitverhalten, Verbrauchsverhalten, Konformität
Änderbarkeit	Analysierbarkeit, Modifizierbarkeit, Stabilität, Testbarkeit
Übertragbarkeit	Anpassbarkeit, Installierbarkeit, Koexistenz, Austauschbarkeit, Konformität

Qualität ist ebenfalls eine Verhältnisgröße, denn sie bezieht sich auf eine zuvor definierte Menge von Anforderungen an ein Objekt. Der Standard *EN ISO9000:2005* bezeichnet Qualität daher als *Grad, in dem ein Satz inhärenter Merkmale Anforderungen erfüllt.* Mithilfe einer *Metrik* kann die Qualität quantifiziert werden. Metriken bezeichnen einerseits die Methode der *Operationalisierung* der Qualität, andererseits eine konkrete *Kennzahl.* Die Operationalisierungsmethode umfasst eine Messung, mit deren Hilfe eine Messgröße der Qualität mit einer Maßeinheit verglichen wird, und eine Skalierung, die den gemessenen Wert auf eine Skala der Größe anpasst.

Standardisierte Qualitätsmodelle Die Qualität eines Informationssystems ist eine zusammengesetzte Größe. Durch ein *Qualitätsmodell* werden die Teilgrößen der Qualität genauer beschrieben. Es existieren verschiedene Qualitätsmodelle, wie beispielsweise *FURPS* (Functionality, Usability, Reliablity, Performance, Supportability), die zurückgezogene *DIN 66272* oder *ISO/IEC 9126.* Letzterer Standard definiert *Qualitätsmerkmale und -untermerkmale (engl. (sub) quality characteristics)* als Teilgrößen der Qualität eines Softwaresystems wie in Tabelle 2.1.

Für die Beschreibung der Qualität zur Berechnung der Energieeffizienz eines Vorganges in einem Informationssystem ist nur eine Teilmenge dieser Merkmale betrachtenswert. Insbesondere ist in diesem Modell die Effizienz, das Verbrauchsverhalten und damit die Energieeffizienz selbst ein Qualitätsmerkmal des Systems. Grundsätzlich kann deshalb die Energieeffizienzbestimmung als eine Abwägung zwischen unterschiedlichen Qualitätsmerkmalen begriffen werden. Jedes (Unter-)Merkmal benötigt eine eigene Metrik. Das Ziel eines energieeffizienten Systems ist es, dem Benutzer eine bestimmte, durch ihn abgewogene, Qualität zur Verfügung zu stellen und diese ins Verhältnis zur verbrauchten

Energie zu setzen. Dabei wird nur der eingeschränkte Merkmalsraum betrachtet, der für den Benutzer tatsächlich eine Rolle spielt, also für ihn nützlich ist. Die Qualität, die der Benutzer letztendlich erfährt, kann deshalb auch als *Nutzen (engl. utiltity)* bezeichnet werden. Die in diesem Merkmalsraum eingeschlossenen Merkmale werden durch sogenannte *Benutzermetriken (engl. user mertrics, [FS04])* bewertet.

Die Abwägung der Qualitätsmerkmale geschieht durch Wichtung ihrer Kennzahlen (Metriken). Der konkrete Nutzen ist somit die Summe der gewichteten Kennzahlen, der Wert der Energieeffizienz ergibt sich wie folgt:

$$energyefficiency = \frac{utility}{energy} = \frac{\sum_{i=0}^{n} (w_i * usermetric_i)}{energy}$$

Qualität in der Softwareanalyse Bereits während der Anforderungsanalyse werden in der Softwareentwicklung Qualitäten als Nicht-funktionale Eigenschaften (bzw. Anforderungen) identifiziert. In [MB01] wird der Analysevorgang beschrieben. Zu den Laufzeit-bezogenen, nicht-funktionellen Eigenschaften der Software werden dort gezählt:

- Benutzbarkeit
- Konfigurierbarkeit und Haltbarkeit
- Korrektheit, Verlässlichkeit und Verfügbarkeit
- Performancebezogene Quality Of Service (QoS)
- Sicherheit
- Skalierbarkeit

Weiterhin werden Quellen für diese Anforderungen genannt. Dies können beispielsweise sein: Die Mindestanforderungen (Constraints) eines Systems, insbesondere der Hardware; direkte oder abgeleitete Endbenutzer- oder Softwareentwicklervorgaben; aus Funktionen (Features) abgeleitete Bedingungen sowie aktuelle Marktbedingungen. Die Anforderungen müssen dabei das SMART-Schema [MK95] erfüllen. Dies setzt voraus, dass die Qualitätsmerkmale spezifisch, erreichbar, realisierbar und letztendlich während der gesamten Softwareentwicklung konsistent nachvollziehbar sein müssen.

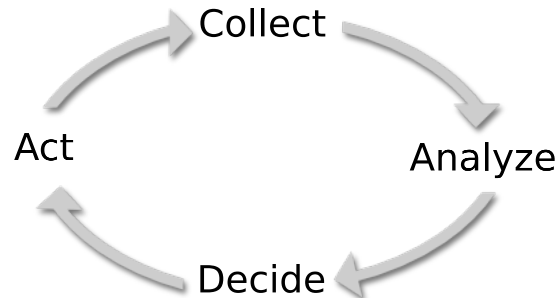
Dienstgüte Netzwerkanwendungen betreffend wird häufig der Begriff *Quality of Service (QoS)* verwendet. QoS stellen Anforderungen an Netzwerkverbindungen wie Latenzzeit, Jitter, Paketverlustrate und Datendurchsatz. Insofern ein QoS-Aspekt messbar ist wird er als QoS-Charakteristik bezeichnet.

2.3. CoolSoftware

Das CoolSoftware-Projekt[2] stellt in seinem Ansatz eine effizienzorientierte Verbindung zwischen Software und von ihr verwendeter Hardware her. Dazu wird ein Modell der

[2] http://www.reuseware.org/index.php/CoolSoftware

Abbildung 2.2. Auto-Tuning Verarbeitungskreislauf (nach [DDF+06])



Eigenschaften und Beziehungen zwischen Softwarekomponenten [Szy02] und anderen Softwarekomponenten oder Hardwareressourcen erzeugt. Das zugrundeliegende Metamodell wird als *Cool Componenten Model (CCM)* bezeichnet. Ziel ist es, das Softwaresystem auf Grundlage einer über den Modelldaten gefällten Entscheidung möglichst energieeffizient anzupassen. Dazu wird ein Auto-Tuning-System [DDF+06] verwendet. Den Verarbeitungskreislauf eines solchen Systems zeigt Abbildung 2.2. Das System ermittelt notwendige Modelldaten (*Collect*) und untersucht mögliche Konfigurationen (Varianten) hinsichtlich der Energieeffizienz (*Analyze*). Dies geschieht zum Beispiel durch Simulation eines modellierten Arbeitsaufwandes (Workloads). Danach kann das System die energieeffizienteste Konfiguration auswählen (*Decide*). Abschließend werden die modellierten Komponenten rekonfiguriert (*Act*). Diese Konfiguration des Systems anhand einer Energieeffizienzbewertung wird als Energy Auto Tuning (EAT) [GWS+10a] bezeichnet.

2.3.1. Das Cool Component Model

Die von CoolSoftware verwendeten Modelle repräsentieren sämtliche für die Effizienzentscheidung notwendigen Informationen. Mit den modellierten Daten sollen Vorhersagen über die jeweilige Effizienz einer möglichen Konfiguration erbracht werden. Die validen Konfigurationen sind durch Contract Negotiation [MS07] zu bestimmen.

Modellierung von Komponentensystem und Ressourcen Zur Modellierung einer hierarchischen IT-Infrastruktur enthält CCM das Paket `structure`, dessen Klassenstruktur Abbildung 2.3 zeigt. Es ermöglicht die Definition von Schnittstellen und möglicher Interaktionen zwischen diesen. Die Klasse `ComponentType` sowie ihre Spezialisierungen `UserType`, `ResourceType` und `SWComponentType` beschreiben verschiedene Komponenten des Systems; letztere beiden können durch Selbstaggregation hierarchisch zusammengesetzt werden. Die Kommunikationsschnittstellen einer Komponente wird durch die Klasse `PortType` beschrieben, die Kommunikationsrichtung der Schnittstelle durch eine `Direction`. Valide Verbindungen zwischen Schnittstellen repräsentieren `PortConnectorTypes`. Zudem besitzt jede Komponente einen Vertrag (`Contract`).

Abbildung 2.3. CCM `structure`-Paket (nach [GWS+10a])

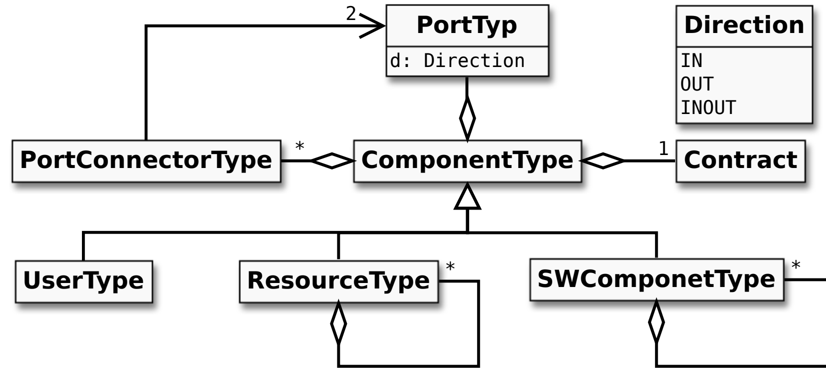


Abbildung 2.4. CCM `behavior`-Paket (nach [GWS+10a])

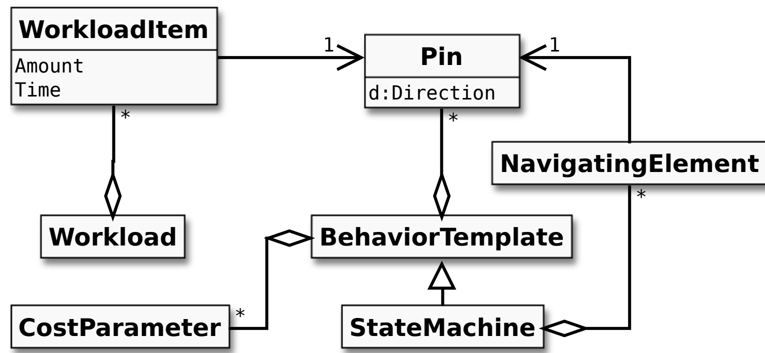
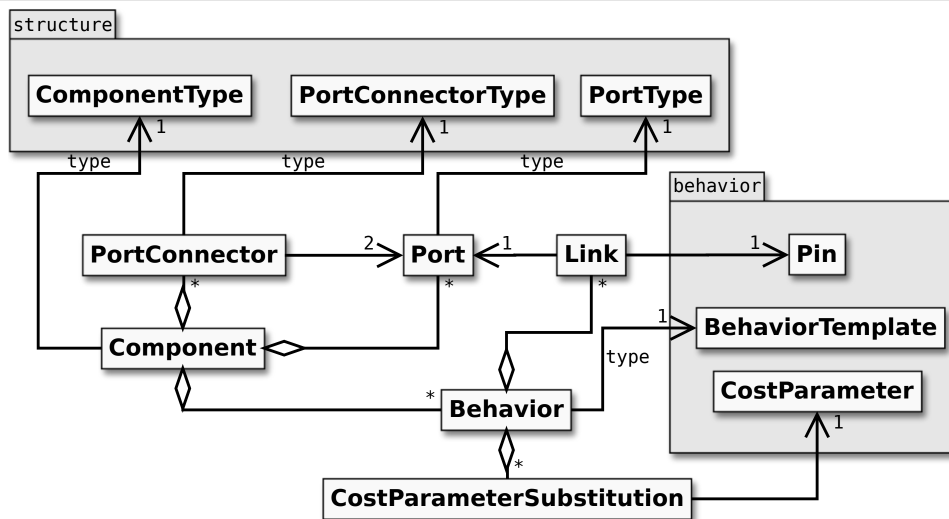


Abbildung 2.5. CCM `variation`-Paket (nach [GWS+10a])



Verhaltensmodellierung Das Verhalten der Komponenten beschreibt das `behavior`-Paket (Abbildung 2.4). Die zentrale Klasse `BehaviorTemplate` repräsentiert ein durch `CostParameter` adaptierbares Verhaltensmuster. Es kann unterschiedlich implementiert werden, sei es durch eine Zustandsmaschine, deren Zustände und Transitionen Energieverhalten einbeziehen, oder durch Heuristiken. Mithilfe der Klassen `Pin` und `NavigatingElement` kann später (siehe `variation`-Paket) das Verhalten einer Komponente mit Ereignissen im Verhaltensmuster assoziiert werden. Auch ein aus `Workloaditems` zusammengesetzter `Workload` kann durch diese Konzepte beschrieben werden. Das Verhaltensmodell dient maßgeblich zur Bestimmung des Energieaufwandes eines Workloads.

Varianzmodellierung Die Verbindung zwischen den beiden bisherigen Paketen stellt das `variation`-Paket, Abbildung 2.5, her. Sowohl für Klassen des `structure`-Paketes, als auch für das `BehaviorTemplate` können konkrete Varianten definiert werden. Zudem wird das Verhaltensmuster durch `CostParameterSubstitutions` parametrisiert. Durch die Klasse `Link` lässt sich, wie oben bereits erwähnt, das Verhalten der Komponente mit diesem Muster verknüpfen.

2.3.2. Vertragsmodellierung

Die Beschreibung der funktionellen Eigenschaften und damit verbundenen Energieverbräuchen erfolgt auf Grundlage der *Energy Contract Language (ECL)*, deren Vorbild die *Component Quality Modeling Language (CQML)* [Aag01, RZ03] ist.

Die Maßeinheit eines Qualitätsmerkmales wird in ECL als Qualitätscharakteristik beschrieben. Listing 2.1 zeigt die Definition einer Charakterisitik in ECL für einen fiktiven Stream-Videoplayer. Der Wert der Charakteristik wird aus dem Objekt der Klasse VideoStream abgeleitet. Ressourcen können in ECL wie in Listing 2.2 modelliert werden. Die Netzwerkschnittstelle besitzt die beiden Zustände `connected` und `disconnected`, die beide unterschiedliche Leistungsaufnahmen generieren. Im verbundenen Zustand kann eine Verbindung mit maximalem Datendurchsatz von 600 kbit/s aufgebaut werden. Auch während der Transition zwischen den Zuständen wird Energie und besonders Zeit verbraucht. Eine Transition kann durch ein Ereignis ausgelöst werden. Der letzte Ausdruck definiert den initialen Zustand der Ressource.

Der Vertrag der Softwarekomponente VideoPlayer wird in Listing 2.3 gezeigt. Es sind jegliche Zuständsübergänge erlaubt, durch den `precedence`-Ausdruck definiert der Modellierer den Zustand `highQuality` als vorziehenswürdig. Beide Zustände definieren eine Mindestqualität, die notwendig ist, um die jeweilige Framerate zu garantieren.

Listing 2.1 ECL-Charakteristik (nach [GWS+10a])

```
characteristic framerate(stream:VideoStream){
 domain numeric integer frames/second;
 value stream.getFrameRate();
}
```

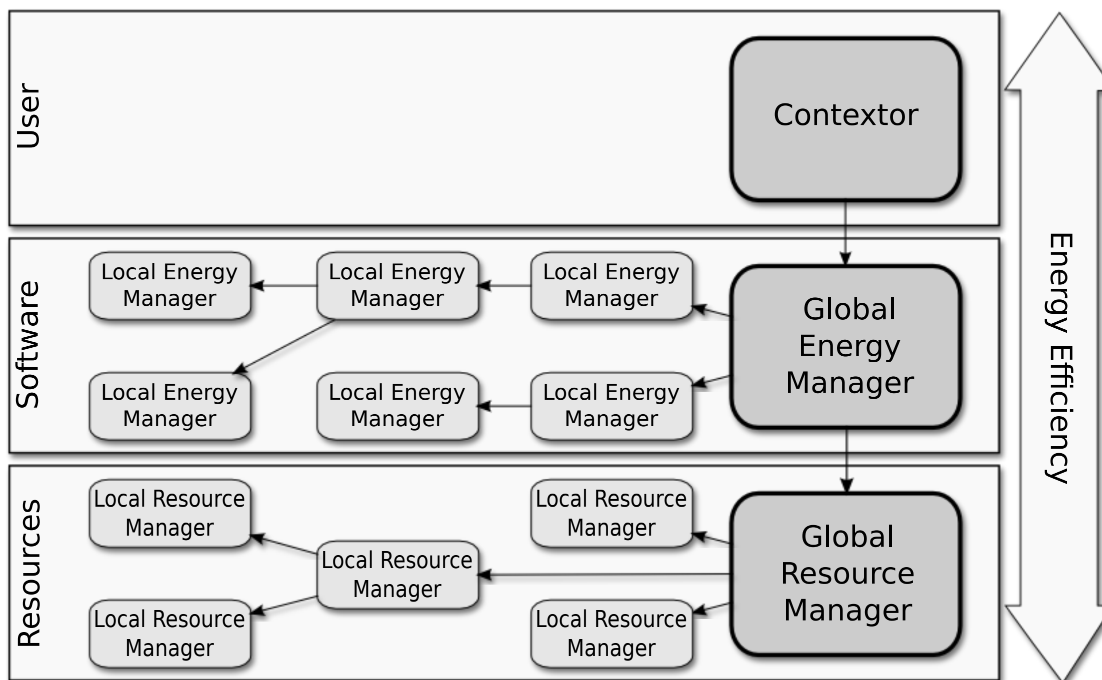
Listing 2.2 ECL-Komponentenprofil (nach [GWS+10a])

```
profile NetworkContract for Network {
 state connected{
  energy-effect 0.8 Watt;
  provides characteristic bandwidth = 600;
 }
 state disconnected{
  energy-effect 0.4 Watt;
 }
 transition connected -> disconnected{
  whenever event-occurs disconnect();
  delay 1.0 seconds;
  energy-effect 0.5 Watt;
 }
 transition disconnected -> connected{
  whenever event-occurs send();
  delay 10 seconds;
  energy-effect 30.0 Watt;
 }
 initial-state disconnected;
}
```

Listing 2.3 ECL-Ressourcenprofil (nach [GWS+10a])

```
profile PlayerContract for VideoPlayer {
 state highQuality{
  uses characteristic bandwidth = 450;
  provides characteristic framerate = 30;
 }
 state lowQuality{
  uses characteristic bandwidth = 150;
  provides characteristic framerate = 10;
 }
 transition any-state -> any-state{}
 precedence highQuality, lowQuality;
}
```

Abbildung 2.6. THEATRE-Architektur (nach [GWS+10a, GWS+10b])

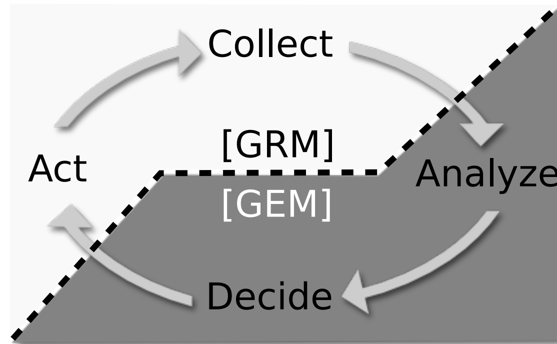


2.3.3. Architektur

Das von CoolSoftware beschriebene *THree-layer Energy Auto Tuning Runtime Environment (THEATRE)* organisiert in drei Schichten notwendige Informationen über Benutzer, Energieverträge, Komponenten und Ressourcen sowie die Zugriffspunkte der Ressourcen. Diese Struktur wird in Abbildung 2.6 veranschaulicht. Auf unterster Architekturebene, dem *Resource Layer* wird eine Ressourcenhierarchie aufgespannt. Als Ressourcen werden unter anderem auch Betriebs- und Dateisysteme benannt. Jede Ressource wird jeweils durch einen *Local Resource Manager* verwaltet, der deren Steuerung ermöglicht. Die gesamten Ressourcenhierarchie wird durch eine zentralen *Global Resource Manager* verwaltet. In [GWS+10b] wurde bereits die Schnittstelle der Resource Manager spezifiziert. Die *Energy Manager* der darüberliegenden Ebene *Software Layer* verfügen über das Vertragswissen der jeweiligen Softwarekomponente. Die *Local Energy Manager (LEM)* werden ebenfalls von einem übergeordneten *Global Energy Manager (GEM)* verwaltet. Auf dem *User Layer* werden durch *Contextors* die Benutzerbedürfnisse quantifiziert. Diese umfassen tatsächlich durch den Benutzer wahrnehmbare Qualitäten, die sogenannten *User Metrics* [FS04].

Die Zuständigkeiten von `Resource-` und `EnergyMgr` können auf das Autotuning-Verfahren direkt übertragen werden. Wie Abbildung 2.7 zeigt, ist der Resource Layer für die Informationsbereitstellung und Analyse der Ressourcen verantwortlich, während auf dem Software Layer eine Entscheidung herbeigeführt und rekonfiguriert wird.

Abbildung 2.7. THEATRE-Verarbeitungskreislauf



2.4. Weitere Ansätze

Neben CoolSoftware existieren einige weitere Ansätze für energieeffziente Softwaresysteme. In [FZJ08] wird versucht, verschiedene Anwendungen energieeffzient auf mobilen Geräten auszuführen. Ihren Ansatz bezeichnen Fei et. al. als *Dynamic Software Mangement (DSOM)*. Unterschiedliche QoS-Konfigurationen werden mit Vorhersagen über benötigte Energie in Beziehung gesetzt. Die Wertung einer Konfiguration erfolgt durch eine vom Benutzer gestellte Priorisierung der Qualitätsmerkmale.

Auch Flinn et. al. [FPS02] beschäftigen sich bereits länger mit dem Problem energieeffizienter Software. Sie beschreiben ihr System *Spectra*, welches ebenfalls die energieeffiziente Ausführung mobiler Software fokussiert. Spectra überwacht das Verhalten und den Ressourcenverbrauch von Software und entscheidet über deren Ausführungsort im mobilen verteilten System. Eine gewichtige Rolle spielt in mobilen System zudem die Ressourcenverfügbarkeit, die besonders aufgrund der unstetigen Netzwerkverbindung ständig variiert.

In [ONR+06] wird die energieeffiziente verteilte Softwareplattform *AutoPower* für ein Netzwerk autonomer Roboter entwickelt. Dabei werden absolute Constraints durch Softwarekomponenten definiert. Ihr Framework erlaubt es ihnen, die Batterierestlaufzeit der autonomen Systeme bei Durchführung eines definierten Workloads einzuschätzen. Einzelne Glieder eine für eine Aufgabe notwendigen Verarbeitungskette werden im Anschluss auf die Aktoren verteilt. Dabei nimmt der Entscheidungsprozess besonders Rücksicht auf Kommunikationskosten zwischen diesen Aktoren.

Verschiedene Softwarearchitekturen verteilter Systeme untersuchten die Autoren von [SEMM08] bezüglich deren Energieeffizienz. Sie entwickelten ein Framework, das auf Grundlage von Befehlen in einer virtuellen Maschine den Energieverbrauch beim Einsatz eines der Architekturstile bestimmt. Es werden sowohl die Rechenkosten der Komponenten, als auch deren Kommunikationskosten in Betracht gezogen.

3. Konzept

In diesem Kapitel wird ein Konzept entwickelt, in dem der oben beschriebene EAT-Ansatz mit einer vorangehenden Workflow-Verarbeitung integriert wird. Ein energieeffizientes Workflow Management System benötigt eine Vielzahl von Informationen, aus denen Entscheidungen abgeleitet werden können, die eine optimale Konfiguration für das verwaltete System bestimmen. Da das dazu notwendige Planungssystem mit beliebigen Klientensystemen kooperieren soll, muss es auch vom WfMS entkoppelt werden. Das WfMS stellt nur eine Anfangsinformation bereit, die eine Verarbeitungskette im Planungssystem in Gang setzt. Die anfangs Workflow-spezifischen Berechnungen liefern wiederum Daten, die an ein angeschlossenes Energy-Auto-Tuning-System, dessen Konzeptvorbild CoolSoftware ist, weitergereicht werden. Beide Teilsysteme - Workflowverarbeitung und EAT-System - umfasst der hier vorgestellte Ansatz.

3.1. Übersicht

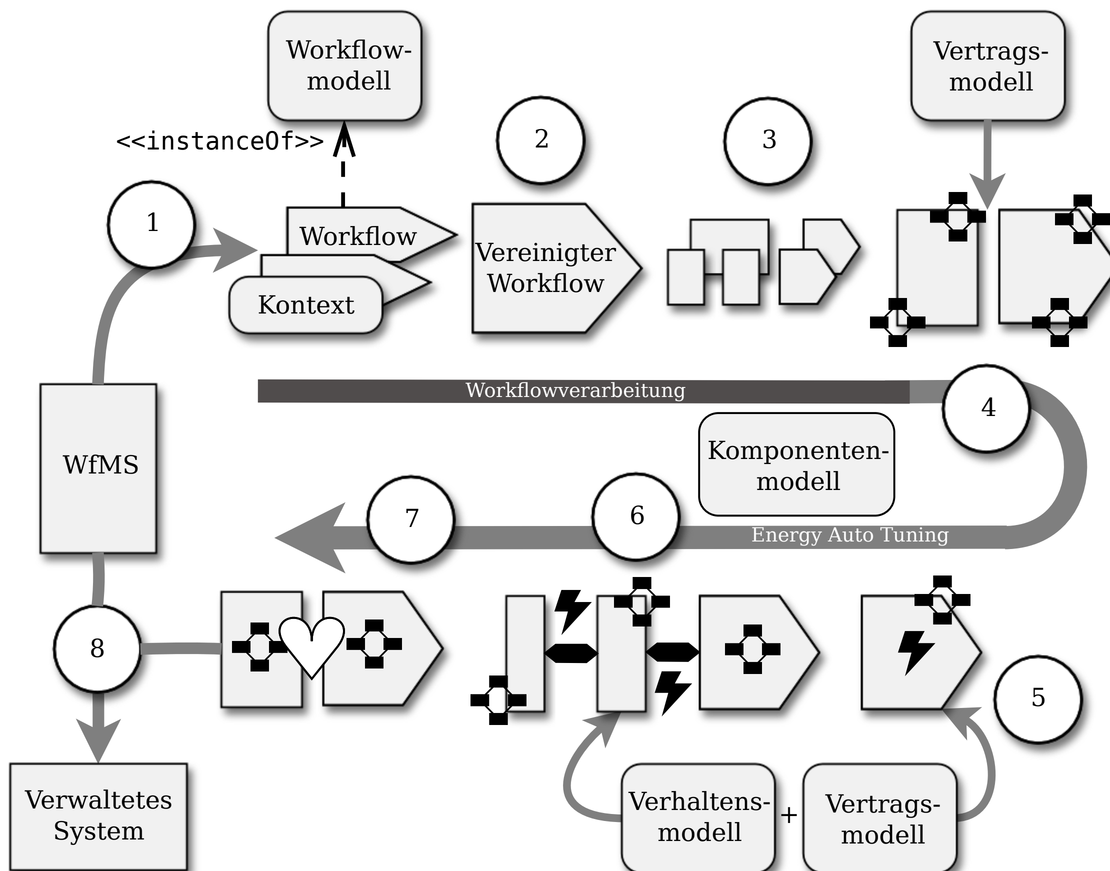
Abbildung 3.1 zeigt eine Übersicht der Verarbeitungsschritte. Die vom WfMS eingespeisten Informationen (1) bestehen aus benutzerdefinierten Qualitätswichtungen, die den Kontext eines Workflows bilden sowie einem Modell des Workflows selbst. Die konventionierte Kommunikation mit dem Planungssystem bedingt den Einsatz eines gemeinsamen *Workflowmetamodells*. Klientensysteme müssen deren Daten auf Grundlage dieses Metamodells vor der Kommunikation transformieren. Dabei kommt zudem das *Komponentenmodell* zum Einsatz. Dieses verwendet sämtliche im System benötigten Modelle. Es repräsentiert Abhängigkeits- und Kommunikationsbeziehungen der verwalteten Ressourcen und Komponenten und ermöglicht die Berechnung einer validen Konfiguration.

Da ein WfMS eine zentrale Infrastruktur in der Unternehmens-IT ist, wird es von mehreren Benutzern für mehrere Vorgänge (Workflows) gleichzeitig verwendet. Deshalb muss das System in der Lage sein, parallele Workflows zu planen (2). Dazu kann das WfMS als weiteren Kontext den relativen Startzeitpunkt nach Beginn der Planung jedes (Teil-)Workflows zur Verfügung stellen. Das Planungssystem erzeugt aus diesen Informationen ein *vereinigtes* Workflowmodell.

In Schritt (3) wird der Workflow in Planungseinheiten *partitioniert*. Da eine Reihe von Partitionierungen des Workflows möglich sind, muss eine Vielzahl von Teil-Workflows betrachtet werden. Die Ursache dieser Variabilität ist vor allem die nicht bekannte Ausführungsreihenfolge von parallelen Workflow-Tasks. Alle folgenden Berechnungen finden für die Gesamtmenge der generierten Partitionen statt.

Mit Beginn des nächsten Schrittes (4) sind die Berechnungsschritte dem EAT-System zuzurechnen. Zunächst können aus den Partitionen funktionale Anhängigkeiten in Form von Operationszugriffen extrahiert werden. Diese Abhängigkeiten beschreiben die funktionalen Mindestanforderungen an das System. Mögliche Implementatoren definiert das Komponentenmodell. Der Aufbau einer vollständigen Komposition, die auch die Ressourcen und Komponenten beschreibt, auf denen diese oberflächlichen Implementatoren aufbauen, unterliegt einer umfangreichen Varianz. Diese wird durch ein *Vertragsmodell* beschrieben, dessen Grundlage die Energy Contract Language ist. Das Vertragsmodell definiert mehrere Zustände sowie Transitionen zwischen diesen. Jeder Zustand kann

Abbildung 3.1. Konzeptüberblick



minimale Qualität bedingen oder maximale Qualität garantieren. Sowohl in Zuständen als auch Transitionen werden Energieverbräuche definiert. Die Erfüllung von funktionellen Bedingungen zwischen Komponenten und Ressourcen variiert in der jeweiligen Maximalqualität der nicht-funktionellen Bedingungen. Die Auswahl anhand dieser Vorgaben wird durch das Verfahren *Contract Negotiation* vorgenommen.

Aus dem für jede Partition berechneten Aufwand (Workflow) und den für die Partition bestimmten Konfigurationsvarianten muss in Schritt (5) ein Energieverbrauch ermittelt werden. Dies geschieht durch *Simulation*. Das Simulationsmodell beschreibt das Verhalten des Systems bezüglich definierter Operationen und Zustände. Das *Verhaltensmodell* wird in diesem Ansatz durch den Benutzer modelliert und beschreibt den Kontrollfluss während der Operationen. Die Partition, die ein Teil-Workflow ist, wird vor der Simulation in dieses Modell transformiert. Die Simulation selbst bestimmt den *Lastenergieaufwand*, der im Verhaltensmodell definiert ist und während einer Operation anfällt, sowie die dafür benötigte Zeitspanne. Die in dieser Zeit anfallende, zusätzliche *Ruheleistungsaufnahme* («Grundlast») wird dem Vertragsmodell entnommen.

Jede Partition beschreibt nun eine Zeitspanne mit unveränderter Variante. Die Variantenumschaltung geschieht zwischen den Partitionen, die durch Kombination wieder zusammengesetzt werden. Während der Umschaltung kann, zum Beispiel für Aufwände bei der Aktivierung oder Deaktivierung eine Komponente oder Ressource, oder durch Zustandsübergänge, die im Vertragsmodell definiert wurden, zusätzlicher Aufwand entstehen. Dieser wird ebenfalls durch Simulation in Schritt (6) berechnet.

Das Planungssystem ist nun in der Lage, eine Bewertung der generierten Partitionierungen durchzuführen. Die Qualitätswerte, die aus den jeweils in einer Variante gewählten Komponentenbeschreibungen (im Vertragsmodell) entnommen werden, können zusammen mit dem Benutzergewichtungen und dem simulierten Aufwand zu einem *Energieeffizienzwert* verrechnet werden. Die effizienteste mit einer Partitionierung assoziierte Variantenauswahl wird als *Konfigurationsplan* installiert (7).

Nach der Installation beginnt das WfMS am anfangs definierten Zeitpunkt mit der Ausführung des Workflows. Bei Aktivierung eines Tasks wird dieser dem Planungssystem mitgeteilt und löst eventuelle, im Konfigurationsplan definierte, Rekonfigurationen aus (8). Versucht das WfMS einen Task vor dem geplanten Zeitraum auszuführen, kann es durch das Planungssystem während des fehlenden Zeitraums blockiert werden, um den energieeffizienten Plan nicht zu invalideren. Die detaillierten Modelle und Verarbeitungsschritte werden in den folgenden Abschnitten beschrieben.

3.2. Modelle

Dieser Abschnitt beschreibt die Modelle für Komponentensystem, Verträge, Verhalten und Workflowinstanz. Die Modelle sind nicht vollkommen voneinander getrennt angelegt, sondern importieren Klassen der jeweils abhängigen Modelle, so dass ein mehrschichtiges Gesamtmodell entsteht. Abbildung 3.2 zeigt, wie die einzelnen Teilmodelle von einander abhängen.

Die Syntaxdefinitionen dieses Kapitels sind in einem Dialekt der *Erweiterten Backus Naur Form* (EBNF) [ISO] notiert. Die Notation beinhaltet mehrere Produktionsregeln einer kontextfreien Grammatik. Die jeweils erste Regel ist das Wurzelelement der (Teil-)Sprache. Links des Pfeiles ist der Name des Nichtterminalsymboles genannt, rechts dessen Regelkörper. **Fettgedruckte** Bezeichner sind Terminalsymbole, *kursive* Bezeichner Nichtterminale und unterstrichene Symbole sind Kontrollkonstrukte des EBNF-Dialekts. Ein in '{}' eingeschlossener Ausdruck kann beliebig oft wiederholt werden, durch '|' getrennte Ausdrücke sind alternativ anzuwenden. Runde Klammern umfassen mehrere Alternativen. In '«»' eingeschlossene Ausdrücke sind durch beliebige Zeichenketten zu ersetzen, sie sind die Parameter der jeweiligen Produktionsregel. Ausdrücke in '??' sind optional.

3.2.1. Komponentenmodell

Das Komponentenmodell (Abbildung 3.3) beschreibt Komponenten, Ressourcen sowie deren Schnittstellen und Beziehungen. Es werden ausschließlich die für die Berechnung notwendigen Informationen modelliert. Zunächst lassen sich mithilfe von `Interfaces` die

Abbildung 3.2. Verwendete Modelle

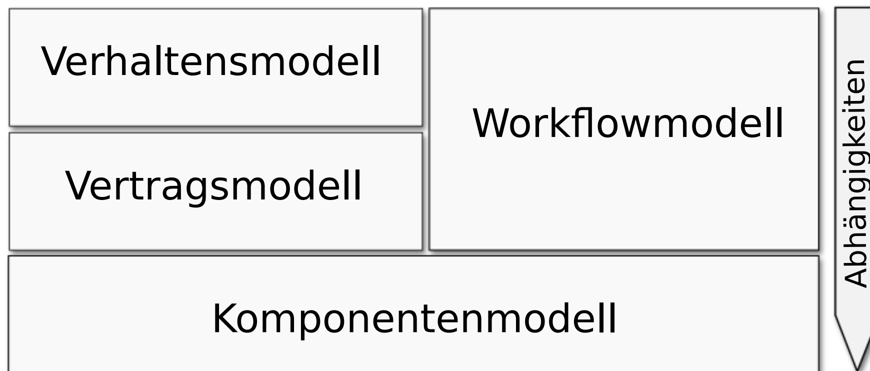
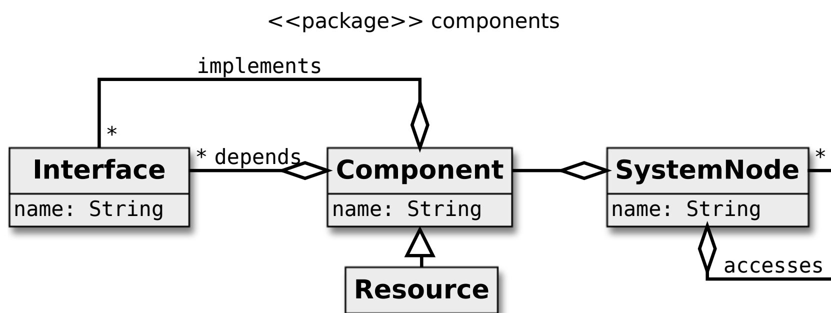


Abbildung 3.3. Komponentenmodell



angebotenen und benötigten Schnittstellen einer Komponente (`Component`) definieren. Auch eine `Resource` ist eine Komponente, wird aber konzeptionell von ihr getrennt. Der Komponentenraum wird zudem durch `SystemNode`s strukturiert. Die Klasse, die sich in einem kompositen Muster selbst aggregiert, beschreibt einen Container innerhalb des Systems, in dem sämtliche Komponentenschnittstellen für im Container enthaltene Komponenten sichtbar sind. Weiterhin ist der Zugriff auf sämtliche implementierte Schnittstellen der direkt untergeordneten Container (`accesses`-Assoziation) möglich. Dieses Muster entspricht beispielsweise dem Zugriff eines verteilten Systems durch Proxy-Objekte auf eine entfernte Ressource. Durch `SystemNode`s kann eine Komponente zudem repliziert werden, ohne mehrfach ihr Verhalten zu beschreiben. Dazu wird eine Komponente von mehreren Knoten gleichzeitig aggregiert. Das Komponentenmodell ist die Grundlage aller anderen Modelle. Deshalb besitzen sämtliche Klassen einen eindeutigen referenzierbaren Bezeichner.

Obwohl die Komponenten des Systems selbstbeschreibend sein sollten, das heißt das Komponentenmodell kann automatisch aus dem System extrahiert werden, ist es von Vorteil, dem Benutzer eine Möglichkeit zu bieten, Komponenten von Hand zu beschreiben. Auf diese Weise kann er *virtuelle* Komponenten definieren, die zwar nicht durch das WEAT-System verwaltet werden (z.B. aufgrund fehlender Treiber), aber trotzdem zum

Syntaxdefinition 3.1 Modellierungssprache für Komponenten

ComponentSpec	→	{(*Component* |*Resource*)} {*SystemNode*}
OperationRef	→	*Compound*
InterfaceRef	→	*Compound*
Component	→	`component` «name» `implements {` {InterfaceRef} `}`
ComponentRef	→	«name»
Resource	→	*Component* `threads` «number»
SystemNodeRef	→	«name» `:` «number»
SystemNode	→	`sysnode` «name» `:` «number» `includes {` {*ComponentRef*} `}`
Compound	→	«base» {`.` «succ»}

Aufwand einer Operation beitragen. So wird es möglich, die Grundlast des Systems in virtuellen Komponenten abzubilden, sowie deren Lastverbräuche darzustellen. Die Sprache umfasst dabei nicht sämtliche Fähigkeiten des Modells, sondern nur jene, um virtuelle Schnittstellen und Komponenten zu definieren und letztere realen Systemknoten unterzuordnen. Syntaxdefinition 3.1 zeigt die dafür notwendige Grammatik. Sie enthält auch eine Notation für Referenzen beliebiger Entitäten des Modells, welche in den Grammatiken der folgenden Modelle wiederverwendet werden können. Somit entsteht eine *Grammatikhierarchie*.

Virtuelle Komponenten beschreiben keine weiteren funktionellen Abhängigkeiten, diese werden anstattdessen aus den assoziierten Verträgen abgeleitet. Die Regel `SystemNode` definiert keinen neuen Knoten, sondern fügt die virtuelle Einheit einem existierenden hinzu. Die Bezeichner der Schnittstellen sind wie in allen modernen objektorientierten Sprachen zusammengesetzt aus einem Namensraum und einem zusätzlichen Terminal. Die einzelnen Teile des Namensraumpräfixes sind durch Punkte getrennt.

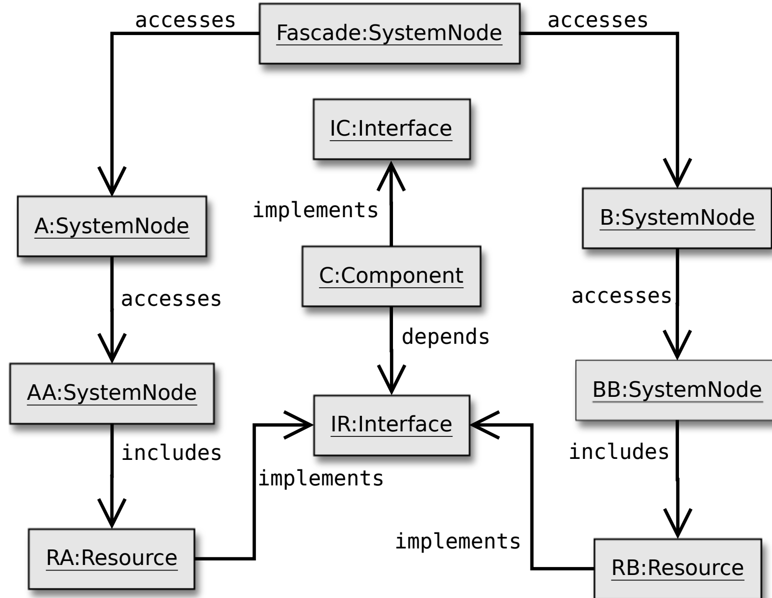
Beispiel 3.1 *Abbildung 3.4 zeigt eine beispielhafte Instanz des Komponentenmodells. Ein* `Fascade`*-Knoten (in dem beispielsweise das WfMS arbeiten könnte) hat Zugriff auf zwei Knoten* `A` *und* `B`*; diese bieten gleiche Softwarekomponenten. Da beide funktionell und nicht-funktionell identische Eigenschaften besitzen, können sie auch durch ein identisches Objekt* `C` *repräsentiert werden. Die Softwarekomponente bedingt die Erfüllung einer Schnittstelle* `IR` *und implementiert selbst die Schnittstelle* `IC`*. Eine weitere Ebene bilden die Knoten* `AA` *und* `BB`*. Beide stellen eine Implementierung der Schnittstelle* `IR` *zur Verfügung. Auf diese Weise wird der operative Aufwand, der in der Komponente* `C` *erzeugt wird, durch die Wahl verschiedener Konfigurationen ebenfalls unterschiedlich sein.*

3.2.2. Einheitensystem und Mathematisches Modell

In den auf diesen Abschnitt folgenden Modellen werden mehrfach Qualitäten, Zeitspannen und Verbräuche definiert. Die Verarbeitung dieser Modellentitäten verursacht folgende Anforderungen:

(1) ein Verfahren, Einheiten zu definieren und auf deren Grundlage verschiedene Beträge umzurechnen

Abbildung 3.4. Beispiel Komponentenmodell



Syntaxdefinition 3.2 Grammatik für Qualitäten, Maßeinheiten und Dimensionen

OperationQuality	→	*characterisicName*`(`*InterfaceRef*`.`«operation»`)`
Unit	→	`[`{«u» | `(` | `)` | `*` | `/` | *Unit*}`]`
Dimension	→	*Unit*

(2) ein Verfahren zur Definition der Qualitäten

(3) ein mathematisches Modell für Berechnungsvorschriften

Zur Lösung der Anforderung (1) kommt das in 3.5 gezeigte Modell zum Einsatz. Die Klasse `Dimenison` definiert eine physikalische Dimension, die durch unterschiedliche Größen implementiert werden kann. Größen typisieren Einheiten (`Unit`) und Beträge (`Amount`). Auf diese Weise kann zwischen Beträgen unterschiedlicher Einheiten der identischen Größen oder Dimension umgerechnet werden. Die Klasse `Amount` unterstützt auch die Verrechnung der Beträge (in der Abbildung wird nur eine Auswahl von möglichen Operationen gezeigt).

Die Definition von Qualitätsmerkmalen wird durch das Modell in Abbildung 3.6 ermöglicht. Die Benennung einer konkreten Qualitätscharakteristik beinhaltet die Referenzierung einer modellierten Schnittstelle, sowie die Definition einer Operation. Letztere ist nicht Teil des Komponentenmodells, da der Operationsbezeichner nur zur Strukturierung von Simulation und Vertrag notwendig ist. Der Bezeichner kann auch als ein beliebiger *Aspekt* der Schnittstelle verstanden werden. Eine Charakteristik trägt ebenfalls einen Namen sowie ihre Größendimension.

Abbildung 3.5. Einheiten-, Größen- und Dimensionsmodell

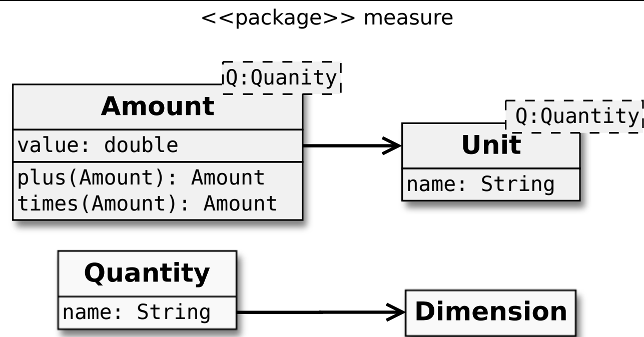
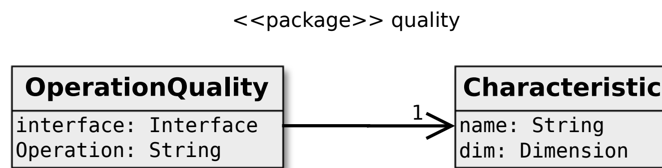


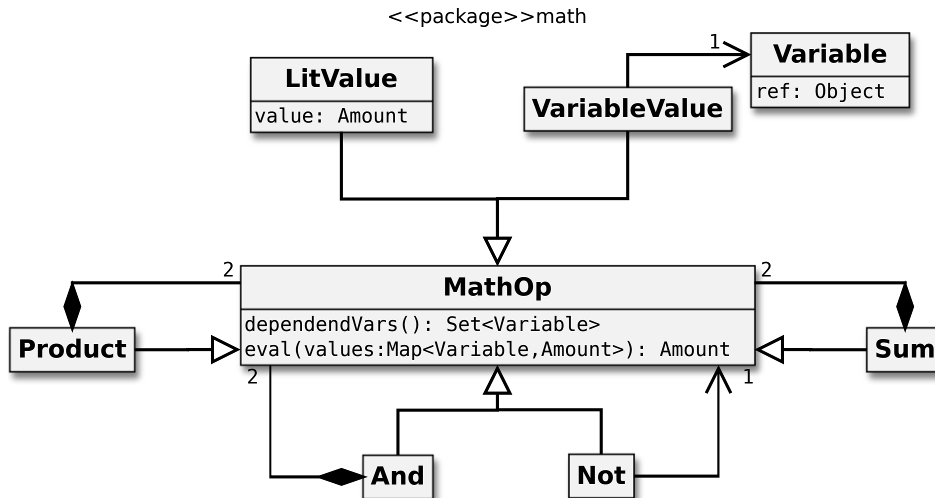
Abbildung 3.6. Qualitätenmodell



Um diese Modelle auf die benutzerdefinierten Verträge zu übertragen, müssen eine zusätzliche Hilfsgrammatik 3.2 definiert werden. Das Nichtterminal `Unit` dient zur Definition von Einheiten. Charakteristiken werden hier nicht definiert, sondern nur referenziert. Die Erzeugung von Charakteristiken wird später in der Grammatik des Vertragsmodells vorgenommen. Dimension und Einheit besitzten identische Syntax, erzeugen aber unterschiedlich klassifizierte Objekte.

Ein mathematisches Modell (3) zeigt Abbildung 3.7. Die Schnittstelle `MathOp` besitzt zwei Methoden: `eval` dient zur Auswertung der Vorschrift auf Grundlage einer Abbildung von Variablen auf Werte, `dependendVars` kann rekursiv sämtliche Variablen ermitteln, um die Formel abzuschließen. Die Implementierungen repräsentieren beispielsweise Summen, Differenzen, Produkte, Divisionen, Literale und auch logische Verknüfungen. Letztere ermöglichen bedingte Berechnungen. Die Semantik der logischen Prädikate entspricht der 'C'-Semantik (die Sprache C). Alle von Null verschiedenen Werte repräsentieren dabei den logischen Wert «wahr». Eine Klasse `VariableValue` definiert eine auszuwertende Variable. Diese Variable kann beliebige Objekte referenzieren; so kann das Modell sowohl für die Berechnung von Qualitätswerten als auch Verbräuchen eingesetzt werden. In der Abbildung sind die Mehrzahl der möglichen Operatoren ausgespart, um an dieser Stelle die Übersichtlichkeit zu bewahren. Auch die zugehörige Grammatik 3.3 zeigt nur das dem Modell entsprechende Muster. Es wird eine strikte Klammersyntax verwendet; somit entfällt die Notwendigkeit, den Operatoren Prioritäten zuzuweisen.

Abbildung 3.7. Mathematisches Modell



Syntaxdefinition 3.3 Grammatik für mathematische Ausdrücke

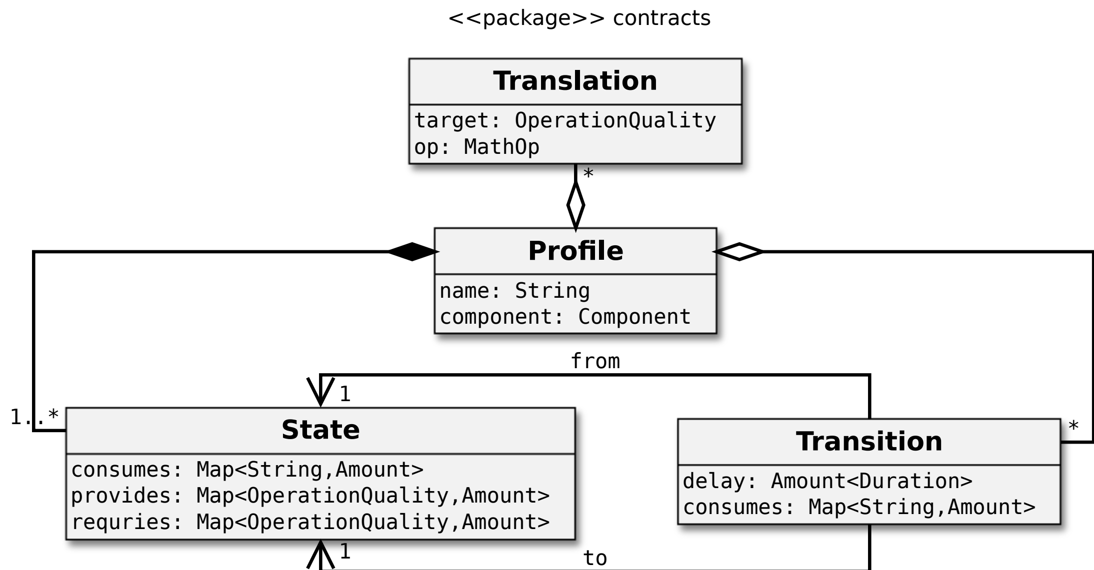
MathOp	→	(*Division* | *And* | *Not* ...)
Division	→	`(` *MathOp* `/` *MathOp* `)`
And	→	`(` *MathOp* {`&` *MathOp*} `)`
Not	→	`!` *MathOp*
...		

3.2.3. Vertragsmodell

Jeder Komponente kann ein Vertrag zugewiesen werden. Das Modell zeigt Abbildung 3.8. Der Vertrag besteht aus einem Komponentenprofil `Profile`, welches eine Zustandsmaschine mit unterschiedlichen Performancezuständen der Komponente beschreibt. Jeder Zustand kann Qualitätsminima `requires` beschreiben, die verwendete Komponenten erfüllen müssen, bevor er aktiviert werden kann, sowie Qualitätsmaxima `provides`, die in der mit dem Vertrag assoziierten Komponente in diesem Zustand zur Verfügung stehen. Verbräuche repräsentiert das Attribut `consumes`. Im EAT-System sind dies Leistungsaufnahmen. Tatsächlich können in dem konstruierten Modell auch andere, *beliebige* Verbrauchsdimensionen definiert werden. Dies könnten beispielsweise monetäre Beträge sein. Verweilt eine Komponente über einer Zeitspanne in einem Zustand, wird der Verbrauchwert aggregiert. Im Fall des EAT-Systems ist der aggregierte Wert ein konkreter Energiewert.

Den Zustandsübergang repräsentiert die Klasse `Transition`. Sie hat eine definierte Zeitspanne `delay` und ebenfalls Vebrauchsdefinitionen `consumes`. Der Zustandswechsel ist bei der Betrachtung von Wokflows, beziehungsweise strukturierten Kontrollflüssen, von besonderer Signifikanz, da in den längeren Planungszeitraumen der Rekonfigurationsaufwand einen wichtigen Verbrauchsfaktor darstellt.

Abbildung 3.8. Vertragsmodell



Syntaxdefinition 3.4 EBNF-Definition von ECL’

ECLSpec	→	{*Characteristic*} {*Profile*}
Characteristic	→	**`characteristic`** «name» **`of`** *Dimension*
Profile	→	**`profile`** «name» **`for`** «comp» **`{`** {*Translation*} {*State*} {*Transition*} **`}`**
Translation	→	**`translate`** *OperationQuality* **`=`** *MathOp*
State	→	**`state`** «name» **`{`** {(*Requires* | *Provides* | *Consumes*)} **`}`**
Requires	→	**`requires`** *Constraint*
Provides	→	**`provides`** *Constraint*
Consumes	→	**`consumes`** «qualifier» «value» *Unit*
Constraint	→	*OperationQuality* **`=`** «value» *Unit*
Transition	→	**`transition`** «from» **`->`** «to» **`{`** {(*Consumes* | *Delays*)} **`}`**
Delays	→	**`delays`** «value» *Unit*

Ein weiteres Konzept des Modells ist die `Translation`. Mit ihr können Berechnungsvorschriften `formula` aus Charakteristiken für neue Charakteristiken `target` angelegt werden. Insbesondere lassen sich somit auch Charakteristiken ausdrücken, die einen höheren Qualitätswert durch geringere Beträge (*decreasing values*) repräsentieren. Dazu werden sie einfach auf neue inverse Charakteristiken abgebildet. Die Verarbeitungsalgorithmen fokussieren später nur die *Maximierung der Qualität* bei Minimierung eines Verbrauchswertes.

Die Verträge sind vollständig vom Benutzer (in der Praxis dem Komponentenhersteller) zu beschreiben. Die Sprache ist an ECL angeleht und wird zu Abgrenzung als *ECL'* bezeichnet. Die Grammatik zeigt Syntaxdefinition 3.4.

Beispiel 3.2 *Für das in Abbildung 3.4 dargestellte Komponentensystem definieren die Listings 3.2 und 3.1 zwei Verträge. Der Vertrag der Komponente `RA` beschreibt zwei Zustände `on` und `off`. Nur im aktivierten Zustand kann eine Qualität*

```
qualityIR(IR.operationR)
```

mit skalarer Dimension gewährleistet werden. Dabei wird Energie verbraucht, ebenso wie in den definierten Transitionen. Die Komponente `C` invertiert die hier als fallend angenommene Qualität. Durch den logischen Ausdruck

```
(qualityIR(IR.operationR)> 0[])
```

kann eine Division durch Null vermieden werden, da in diesem Fall das Ergebnis der rechtshändigen Operation bezüglich des ''-Infixoperators mit dem jeweiligen binären Wert multipliziert wird. Wird die Schnittstelle `IR` in diesem Beispiel durch die Ressource `RA` erfüllt, kann die Komponente `C` nur im Zustand `s1` operieren, da der maximale Wert der inversen Qualität* $1/0.03 = 33\frac{1}{3}$ *ist.*

Listing 3.1 ECL'-Profil der Ressource RA

```
characteristic qualityIR of []

profile RAContract for RA{
 state on{
  provides qualityIR(IR.operationR) = 0.03 []
  consumes power 10 [W]
 }

 state off{}

 transition off -> on{
  delays 2000 [ms]
  consumes power 20 [W]
 }

 transition on -> off{
  delays 1000 [ms]
  consumes power 15 [W]
 }
}
```

Listing 3.2 ECL'-Profil der Komponente C

```
characteristic qualityIR of []
characteristic qualityIRInverse of []
characteristic qualityC of []

profile CContract for C{
 translate qualityIRInverse(IR.operationR) =
     ((qualityIR(IR.operationR) > 0[]) * (1 /
     qualityIR(IR.operationR)))

 state s1{
  provides qualityC(IC.operationC) = 100 []
  requires qualityIRInverse(IR.operationR) = 10 []
 }

 state s2{
  provides qualityC(IC.operationC) = 200 []
  requires qualityIRInverse(IR.operationR) = 50 []
 }
}
```

3.2.4. Verhaltensmodell

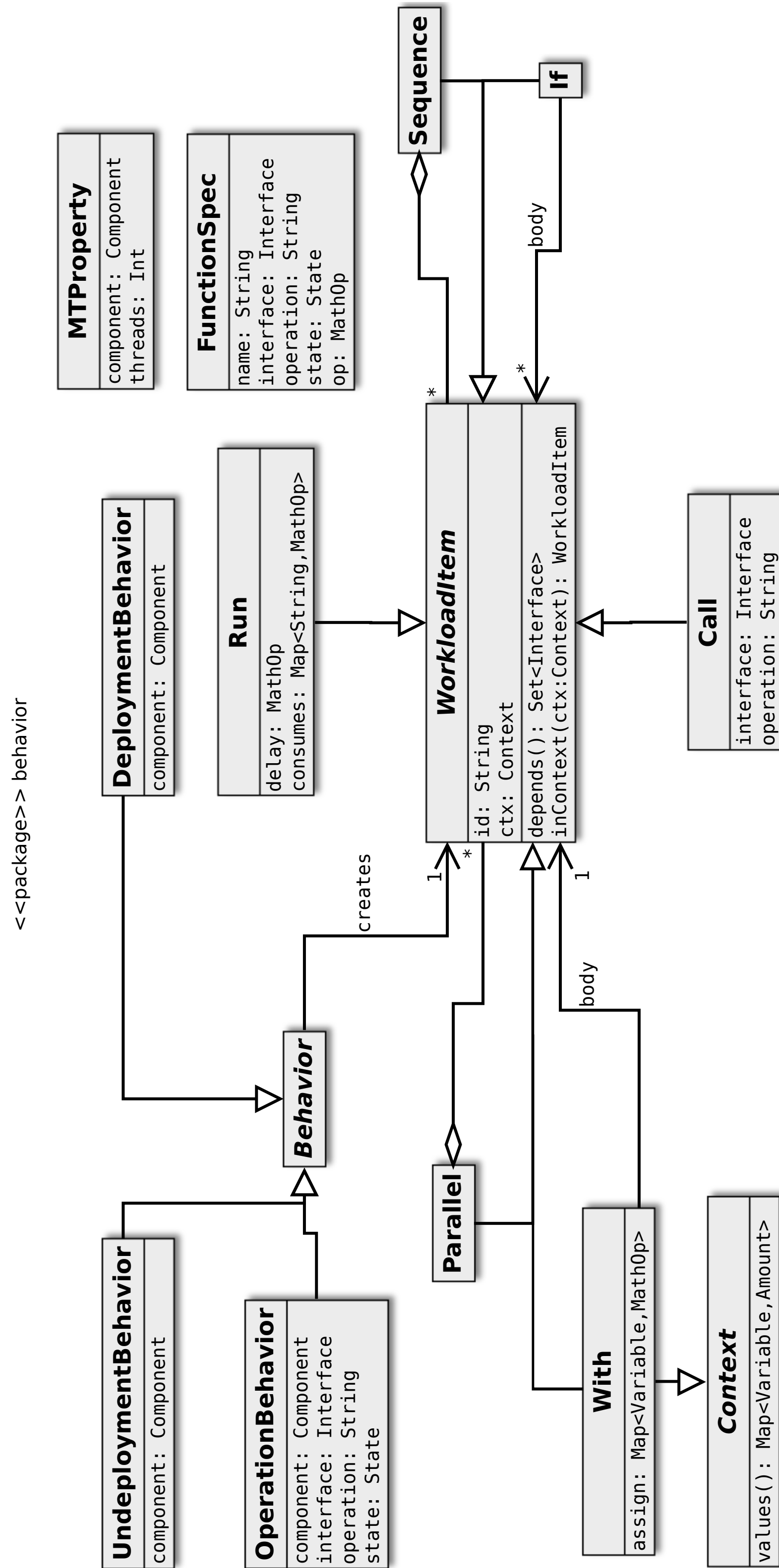
Um den zeitlichen Aufwand eines Operationsaufrufes prognostizieren zu können und damit seinen energetischen Aufwand, muss das Verhalten von Komponenten simuliert werden. Die Simulation benötigt ein *Verhaltensmodell*, welches das System hinreichend genau beschreibt, um eine gewünschte Aussagekraft zu garantieren. Das Modell besteht aus einem Kontrollflussgraphen, der die Interaktion zwischen Softwarekomponenten und Hardwareressourcen repräsentiert und einigen Funktionen zur Berechnung von Zeit- und Energieaufwänden. Es wird ebenfalls durch eine weitere Grammatik vom Benutzer definiert.

Die Klassen des Verhaltensmodells zeigt Abbildung 3.9. Ein `Behavior` ermöglicht die Definition des erzeugten Workloads bei Eintritt eines Ereignisses. Im Modell sind drei verschiedene Ereignistypen verfügbar. Zum einen kann durch ein `OperationBehavior` der Workload bei Aufruf einer Operation beschrieben werden. Die Gültigkeit des Musters schränkt der Parameter `state` auf einen im Vertragsmodell definierten Zustand ein. Insofern dieses Feld undefiniert bleibt, gilt das Verhaltensmuster für sämtliche Zustände. Dies erlaubt auch die Modularisierung der Kontrollflussbeschreibung. Da Operationen nur *Aspekte* einer Schnittstelle sind, das heißt keiner tatsächlichen Methode oder Operation entsprechen müssen, können beliebig neue eingeführt werden. Auf diese Weise ist es möglich, ein zustandsloses Muster als Teilkontrollfluss über sämtlichen Zuständen zu definieren. Die Verhaltensmuster `DeploymentBehavior` und `UndeploymentBehavior` können zusätzlich den Aufwand einer von ihnen referenzierten Komponente bei ihrer Aktivierung oder Deaktivierung beschreiben. Beide Muster sind zustandslos, da sie eine statische Eigenschaft der Komponente definieren.

Der Workload besteht aus `WorkloadItems`. Jedes Item kann durch einen `Context` parametrisiert werden. Die Methode `values` des Kontextes erzeugt je nach Implementierung eine Zuordnung von Variablenwerten. Ein während der Simulation zur Verfügung gestellter *globaler Kontext* kann beispielsweise die virtuelle Simulationszeit definieren. Zudem ist es möglich, einen Systemzustand zu hinterlegen, welcher der Simulation zeitvariante Metriken des Gesamtsystems zur Verfügung stellt. Eine weitere Implementierung ist das `With`-Item. Mit ihm kann ein anderes Item in einen *zur Laufzeit berechneten Kontext* versetzt werden. Dazu definiert der Modellierer Variablenzuweisungen auf Grundlage des oben vorgestellten mathematischen Modells. Die durch einen Kontext definierten Zuweisungen müssen bei der simulativen Traversierung des Kontrollflusses auf die jeweiligen Items angewendet werden. Wird während der Traversierung ein `With`-Kontext überschritten wird er dem bisherigen Kontext hinzugefügt und überschreibt eventuelle vorherige Zuweisungen der in ihm neu definierten Variablen und Funktionen.

Ein `Run`-Item beschreibt einen Vorgang auf einer Hardwareressource für eine definierte Zeitspanne. Die Dauer des Vorganges wird durch eine Berechnungsvorschrift (des mathematischen Modells) definiert. Dies ermöglicht die Beschreibung von Operationen *beliebiger Zeitkomplexität*. Dazu verwenden die Berechnungsvorschriften Variablen und Funktionen, deren Werte und Parameter während der Simulation durch den jeweils auf das `Run`-Objekt angewendeten Kontext entnommen werden. Die Eigenschaft `consumes` beschreibt wie im Vertragsmodell verschiedene Aufwände. Im Fall einer elektrischen

Abbildung 3.9. Verhaltensmodell



Leistungsaufnahme ist dieser Aufwand mit einer *Utilization* (Auslastung) gleichzusetzen. Eine höhere Auslastung entspricht höherer Leistungsaufnahme (anderenfalls muss die Auslastung nicht modelliert werden). Durch die im globalen Kontext erzeugten Zeitparameter ist zudem eine zeitvariante Modellierung der Leistung möglich. Bei sequentieller Wiederholung eines `Run`-Items mit zeitabhängiger Leistungsaufnahme wird diese in jedem Simulationschritt angepasst.

Die Klasse `Call` löst während der Simuation ein neues, synchrones Ereignis aus und beschreibt somit den Aufruf eines anderen Verhaltensmusters des gleichen Zustandes. Die bedingte Ausführung eines Items kann durch ein `If` definiert werden, welches auf die im mathematischem Modell enthaltenen Prädikate zurückgreift. `Sequence` und `Parallel` erlauben die Modellierung mehrerer, entweder sequentiell oder parallel ausgeführter, Items.

Werden unterschiedliche Verhaltensmuster für verschiedene Zustände definiert, kann entweder der Kontrollfluss unterschiedlich strukturiert sein, oder ein oder mehrere Kontrollflusselemente haben unterschiedliche Zeit- und Energieaufwände. Da im letzten Fall die wiederholte Definition der Kontrollflussstruktur überflüssig ist, können durch `FunctionSpec`s solche Aufwände zustandsvariabel zur Simulationszeit bestimmt werden. Zudem können durch Funktionen auch Metriken berechnet werden, die außerhalb der Verhaltensmusterdefinition einer Komponente oder Ressource wiederverwendet werden. Weiterhin lassen sich auf diese Weise auch Standardwerte von Variablen festlegen. Die Funktion trägt einen der betroffenen Schnittstelle untergeordneten Namen «Schnittstelle.Funktionsname».

Die Klasse `MTProperty` ermöglicht die Modellierung von Multitasking-fähigen Komponenten und Ressourcen. Dazu kann eine feste Anzahl `threads` definiert werden. Wird keine solche Eigenschaft für eine Komponente definiert, muss angenommen werden, dass diese beliebig viele nebenläufige Operationen ausführen kann. Ressourcen sollten ohne explizite Modellierung von Nebenläufigkeit nur genau eine gleichzeitige Operation unterstützen.

Syntaxdefinition 3.5 EBNF-Definition der Behavior Language

BehaviorSpec	→	{*Behavior*}
Behavior	→	`when` (*Undeploy* | *Deploy* | *Operation*)
OperationBehavior	→	*OperationRef* `in` *ComponentRef* ?`@` «state» *WorkloadItem*?
Undeploy	→	`undeploy` *ComponentRef* *WorkloadItem*
Deploy	→	`deploy` *ComponentRef* *WorkloadItem*
WorkloadItem	→	{*Parallel* | *Call* | *Run* | *Sequence*} `*` «multiplicity»
With	→	`with (` *Assignment* `)` *WorkloadItem*
Assignment	→	{«variablename» `=` *MathOp*}
Sequence	→	`[` {*WorkloadItem*} `]`
Parallel	→	`{` {*WorkloadItem*} `}`
Call	→	`call` *OperationRef*
Run	→	`run` *MathOp* ?`consumes (` {«effort» *MathOp*} `)`?
If	→	`if` *MathOp* *WorkloadItem*
Function	→	`def` *Compound* `in` *ComponentRef* ?`@` «state»? `=` *MathOp*
MTProperty	→	`enable` «n» `threads in` *ComponentRef*

Das Verhaltensmodell beschreibt der Benutzer (oder Komponenten-/Ressourcenhersteller) mit der in Syntaxdefinition 3.5 dargestellten Grammatik. Mithilfe des Sequenzoperators '*' kann die automatische Wiederholung eines `WorkloadItems` impliziert werden. Dies erleichtert die Definition eines zeitvarianten Kontrollflusses. Durch eine durch diesen Operator automatisch erzeugten Variable `step` kann dazu der Sequenzschritt bestimmt und die Zeitvarianz gesteuert werden.

Beispiel 3.3 *Eine beispielhafte Verhaltensdefinition für das in 3.2.1 eingeführte Systembeispiel zeigt Listing 3.3. Während des Aufrufs der `operationC` wird die `operationR` sequentiell in einer Wiederholung ausgeführt, wobei eine Variable `scale` erzeugt wird, die den Energieverbrauch der `operationR` steuert und durch eine Funktionsdefinition mit dem Standardwert «1» definiert wird. Das zweite Verhaltensmuster demonstriert einen einzelnen Aufruf, der im Zustand `s2` durchgeführt werden würde. Das anschließende `UndeployBehavior` definiert zwei nebenläufige Aufrufe. Letztendlich wird auch ein Muster für den Zustand `on` der Ressource RA definiert. In jedem Simulationsschritt $t = 0 \ldots 9$ beträgt die Gesamtleistungaufnahme für die Ressource `R`*

$$P_R = scale * \sqrt[3]{t} + P_{RB}$$

, wobei die Grundlast $P_{RB} = 10W$ ist, wie im Vertragsmodell für den Zustand `on` modelliert. Die Last-Leistungsaufnahme wurde hier mithilfe der Funktion `pwf` beschrieben. Abbildung 3.10 zeigt den Verlauf der Leistungsaufnahme über den gesamten Zeitraum einer Operation `operationR` unter der Vorraussetzung $scale = 1$.

Listing 3.3 Verhaltensdefinition Beispiel

```
1  when IC.operationC in C@s1[
2   with call IR.operationR*2
3   with(IR.scale = 0.5) call IR.operationR
4  ]
5
6  when IC.operationC in C@s2 call IR.operationR*4
7
8  when deploy C{
9   with(IR.scale = 1.5) call IR.operationR
10  with(IR.scale = 1.9) call IR.operationR
11 }
12
13 def IR.pwr in RA@on = (IR.scale*root(3,step))
14 def IR.scale in RA@on = 1[]
15
16 when IR.operationR in RA@on run 100 [ms] consumes(power IR.pwr)*10
```

Abbildung 3.10. Leistungsaufnahme für Beispiel 3.3

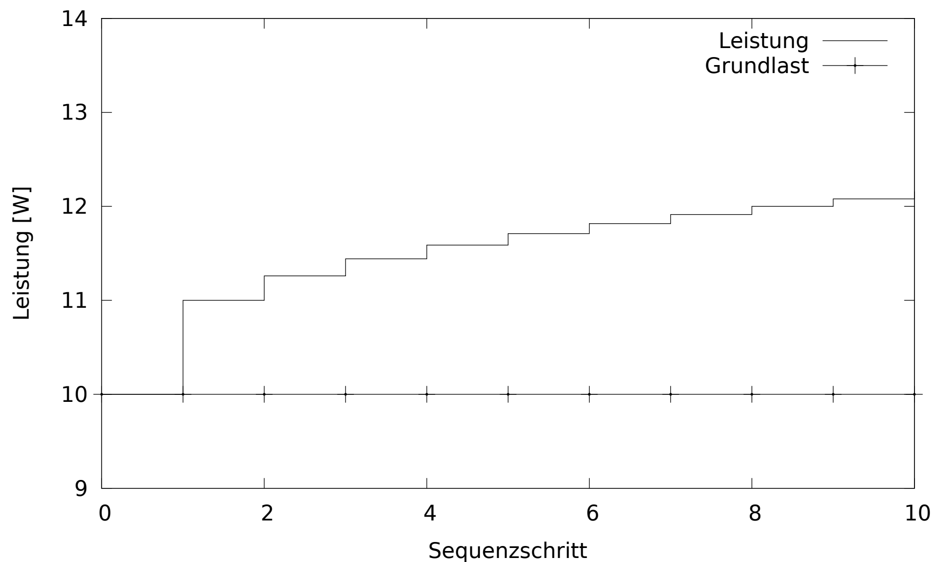
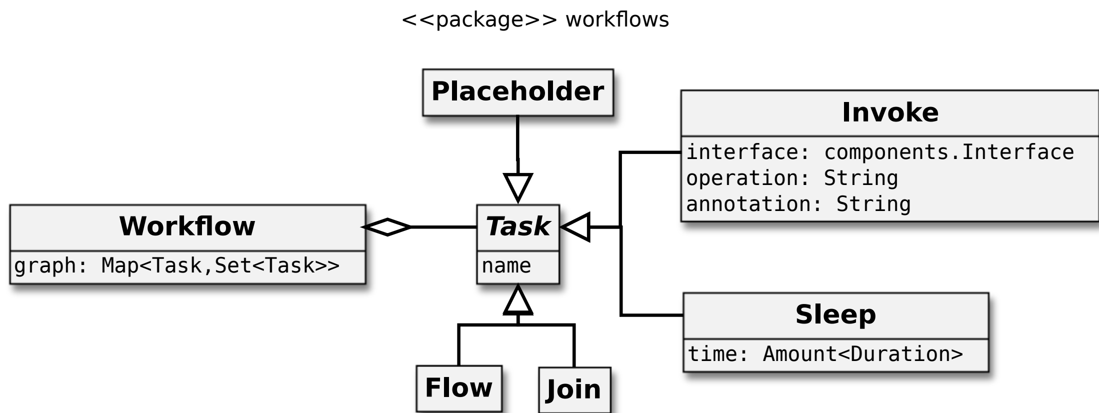


Abbildung 3.11. Workflowmodell



3.2.5. Workflowmodell

Mithilfe des in Abbildung 3.11 gezeigten Workflowmodells können zeitlich und funktionell terminierte Teilworkflows repräsentiert werden. Das Klient-WfMS soll dessen Workflowmodell in das hier vorgestellte transformieren, wodurch der Workflow für das WEAT-System interpretierbar wird. Ein `Workflow`-Objekt aggregiert eine Menge von Tasks und verknüpft diese mit den jeweiligen Nachfolgern (Attribut `graph`). Die abstrakte Klasse `Task` wird implementiert durch die Klasse `Invoke`, die Operationsaufrufe bezüglich des im Komponentenmodell definierten Systems referenziert, sowie durch die Klasse `Sleep`, mit der sich eine im Workflow verstreichende Zeit modellierten lässt. Die Implementierungen `Flow` und `Join` dienen zur Definition paralleler Workflowabschnitte. Das Modell stellt ausschließlich funktionell und zeitlich sichere Informationen dar, enthält somit weder Klassen zur Repräsentation interaktiver Tasks, noch Klassen für bedingte Verzweigungen. Zudem wird vorausgesetzt, dass im Modell keine Zyklen vorkommen.

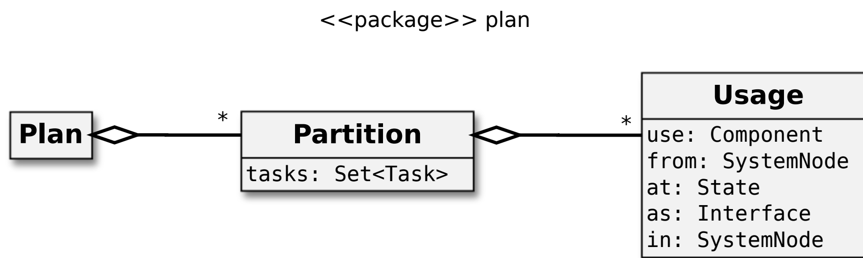
Eine besondere Eigenschaft der Klasse `Invoke` ist das Attribut `annotation`. Mithilfe dieses Attributes und der im Verhaltensmodell 3.2.4 eingeführten Grammatik kann dort ein Kontext erzeugt werden, der Komplexitätsparameter für die anschließende Simulation definiert. Dazu wird das Nichterminal `Assignment` genutzt.

Fehlende Task-Semantiken, zum Beispiel Start, End oder Assign, wie sie in BPEL definiert wurden, sind von diesem Workflowmetamodell ausgeschlossen, so dass sie nicht in die Planung einbezogen werden. Anstattdessen können durch `Placeholder` die entstehenden Lücken im Workflowgraphen geschlossen werden, falls dies aus strukturerhaltenden Gründen notwendig ist.

3.2.6. Planmodell

Die vorgestellten Modelle können im Folgenden verwendet werden, um einen *Konfigurationsplan* zu ermitteln. Um den Plan zu repräsentieren wird ein letztes Modell benötigt,

Abbildung 3.12. Planmodell

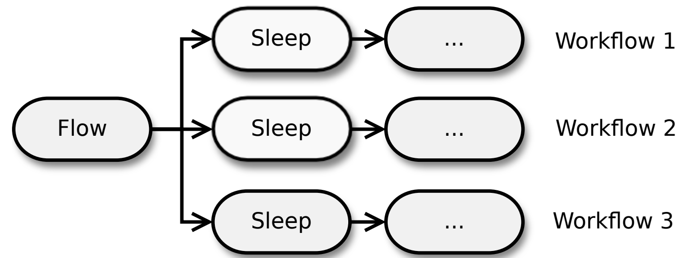


das Planmodell (Abbildung 3.12). Es beschreibt den Plan als eine Menge von disjunkten Partitionen. Jede Partition enthält eine Menge von Tasks des verarbeiteten Workflows, sowie eine Menge von `Usages`; letztere kann als *Variante* des Systems bezeichnet werden. Sie beschreibt die Kommunikationswege und Zustände, die notwendig sind, um die funktionalen und nicht-funktionalen Anforderungen der in der jeweiligen Partition verwendeten Komponentendienste zu erfüllen. Dabei wird eine Komponente (`use`), die in einem Systemknoten (`from`) verfügbar ist und einen bestimmten Zustand (`at`) eingenommen hat, zur Erfüllung einer Schnittstelle (`as`) in einem weiteren, womöglich auch demselben Systemknoten (`in`) eingesetzt.

3.3. Verarbeitungskette

Bevor die Planberechnung begonnen werden kann, muss das WfMS den von ihm durchzuführenden Workflow in obiges Workflow(-meta-)modell transformieren. Da diese Transformation je nach Ursprungsmodell des WfMS unterschiedlich ist, wird sie hier nicht dargestellt. Wie bereits oben erwähnt, sind dabei aber einige Vorgaben umzusetzen: Erstens können keine bedingten oder interaktiven Tasks geplant werden. Diese finden keine Entsprechung im gezeigten Workflowmodell, da sie unsichere Informationen repräsentieren. Ein interaktiver Task kann nicht geplant werden, da seine Ausführungszeit unbekannt ist. Jede verstrichene Zeit muss jedoch während der Planung in einem Energieverbrauch abgebildet werden. Bedingte Ausführungen können erst geplant werden, wenn das Bedingungsprädikat ausgewertet wird und somit der tatsächliche Kontrollfluss bekannt ist. Das hier vorgestellte Konzept nimmt die eventuelle (möglicherweise Wahrscheinlichkeits-getriebene) Vorhersage der bedingten Ausführungen dem WfMS nicht ab, da dieses aufgrund des bereits dort implementieren Wissens über die verwendeten Prädikate derartige Planungen selbst durchführen kann. Die zweite Einschränkung betrifft die Nicht-Betrachtung von zyklischen Vorgängen. Anstattdessen wird eine Planungstiefe des Worfklows gewählt und eventuelle Zyklen werden als wiederholte Sequenzen dargestellt, bis diese Planungstiefe erreicht ist. Der Grund dieser Vorgehensweise ist die ebenfalls unmögliche Vorraussagbarkeit der Wiederholungshäufigkeit eines Zyklus. Entweder wird dieser unendlich oft wiederholt, was ebenfalls eine Wiederholung des berechneten Planes zur Folge hätte, oder der Zyklus wird durch eine Bedingung unterbrochen, die wie bereits erläutert, nicht verarbeitet werden kann.

Abbildung 3.13. Vereinigung paralleler Workflows



Eine weitere Voraussetzung für den Beginn der Berechnung ist die Erzeugung eines Kontextes. Dieser enthält neben der Information, wann die gleichzeitig zu planenden Workflows relativ zum Planungsbeginn ausgeführt werden sollen, eine Information über die Benutzerpräferenzen bezüglich der definierten Qualitätsmetriken. Dazu beinhaltet der Kontext Prioritätsfaktoren, die diese Metriken wichten. Die Summe sämtlicher Wichtungen sollte *Eins* betragen.

3.3.1. Vereinigung paralleler Workflows

Insofern mehrere Workflows gleichzeitig und zu unterschiedlichen Startzeitpunkten geplant werden sollen, können diese durch ein einfaches Verfahren zu einer vereinigten Planungsgrundlage transformiert werden. Die Vorgehensweise zeigt Abbildung 3.13. Das Startelement ist ein `Flow`-Task, der für jeden Workflow zunächst in einen `Sleep`-Task verzweigt. Dieser Tasks definiert die Wartezeit, die für einen Workflow im Kontext beschrieben wurde und wird mit dem eigentlichen Start-Task des Workflows verbunden.

3.3.2. Partitionierung des Workflows

Während der späteren Ausführung des Workflows durch das WfMS soll zwischen verschiedenen Konfigurationen gewechselt werden. Dazu ist es notwendig, den Workflow in Zeitintervalle zu partitionieren. Jedes Intervall beschreibt eine Menge von Workflow-Tasks, für die die gleiche Konfiguration aufrecht erhalten wird. Da bisher noch keine Informationen zum Zeitaufwand der durch den Workflow ausgelösten Operationen generiert wurden, kann die Partitionierung nur anhand der Workflowstruktur vorgenommen werden. Die Synchronisation zwischen WfMS und Planungssystem findet jeweils am Startzeitpunkt eines Tasks statt. Deshalb werden pauschale Intervallgrenzen jeweils vor jedem Task gesetzt. Diese Vorgehensweise kann nur in ausschließlich sequentiellen Workflowabschnitten angewandt werden. In parallelen Abschnitten dagegen beeinflusst die nicht vorhersehbare Nebenläufigkeit mehrerer Tasks die Ausführungszeit. Deshalb wird in diesem Fall eine Auswahl von Möglichkeiten erzeugt, die jeweils beschreiben, welche Tasks parallel auszuführen sind. Ein Beispiel zeigt Abbildung 3.14.

Der Task C kann parallel zu den Tasks B, D, E oder F durchgeführt werden. Sämtliche zu einem Task parallelisierbaren Tasks können durch Traversierung des gerichteten

Abbildung 3.14. Ein Workflow-Task und dessen Parallel-Tasks

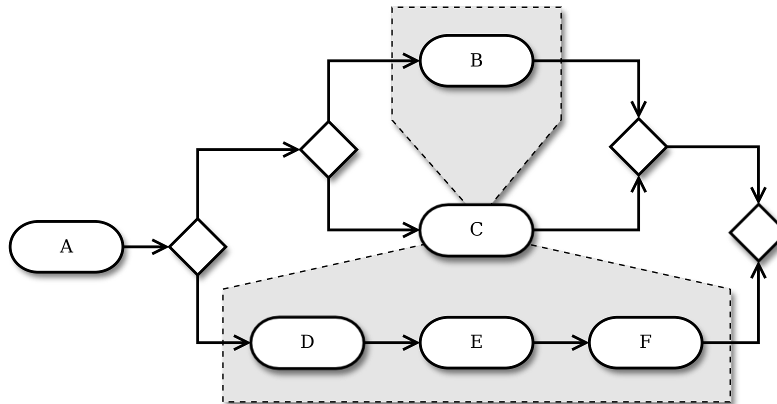
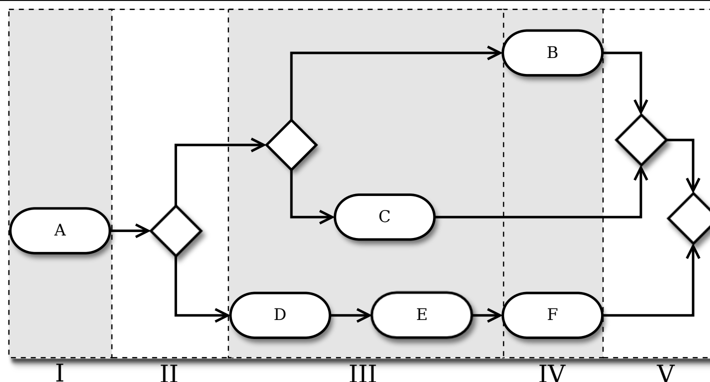


Abbildung 3.15. Eine mögliche Partitionierung



Workflowgraph gefunden werden[1]. Ein Task ist t ist zu einem Teilgraphen $P = (V_P, E_P)$ des gerichteten Workflowgraphen G genau dann parallelisierbar, wenn für alle Tasks $t_p \in V_P$ kein Weg von t nach t_p und kein Weg von t_p nach t in G gefunden werden kann.

Abbildung 3.15 zeigt eine mögliche Partitionierung. Task A wurde in eine einzelne Partition I tranformiert, da er keine möglichen Parallel-Tasks besitzt. `Flows` und `Joins` werden dargestellt als $\diamond$. Da ihnen kein simulationsrelevanter Zeitverbrauch zugerechnet wird, werden Partitionen, in denen außschließlich diese Task-Arten sowie `Placeholders` auftreten, in den nächsten Schritten übergangen werden. Die Partitionen III und IV enthalten parallele Tasks (`Inovke` oder `Sleep`) mit determiniertem Zeitverbrauch oder funktionelle Abhängigkeit und sind somit «echte» Partitionen.

Es werden *sämtliche mögliche Partitionierungen* berechnet. Da dieses Vorgehen eine hohe Rechenkomplexität verursacht, kann eine maximale Anzahl zu berechnender Möglichkeiten definiert werden, um den Algorithmus zeitlich zu beschränken. Die folgenden Berechnungsschritte werden jeweils für *jede* dieser Möglichkeiten vorgenommen.

[1] Zur Einführung in die Graphentheorie siehe beispielsweise [Wes00].

Listing 3.4 Rekursiver Algorithmus Contract Negotiation

```
1  FUNCTION negotiate
2  INPUT   constraints: Set<Contraint>
3  INPUT   variant: Set<Usage>
4  OUTPUT   Set<Set<Usage>>
5  BEGIN
6   IF constraints == ∅ THEN RETURN {variant}
7   ELSE BEGIN
8    VAR c ← x ∈ contraints
9    VAR constraints ← contraints \ q
10   VAR impls ← set of all usages satisfiing c
11   VAR out ← {}
12   FOR EACH i IN impls DO BEGIN
13    VAR variant_next ← variant ∪ {i}
14    VAR constraints_next ← set of all unsatified constraints in
         variant_next
15    out ← out ∪ negotiate(contraints_next,variant_next)
16   END
17   RETURN out
18  END
19 END
```

3.3.3. Contract Negotiation

Jede erzeugte Partition besitzt ein oder mehrere funktionelle Anforderungen, die sich in den Schnittstellenreferenzen der `Invoke`-Tasks manifestieren. Die Aufgabe der Contract Negotiation ist die Erzeugung aller möglichen System*varianten* zur Erfüllung dieser Anforderungen. Jede geforderte Schnittstelle muss in einer Variante durch eine Implementierung erfüllt werden. Dabei sind die Kommunikationswege, die im Komponentenmodell festgelegt wurden, zu beachten sowie eventuell nicht-funktionelle Anforderungen des Vertragsmodells. Eine Variante kann durch eine Menge von `Usage`s (Abbildung 3.12) repräsentiert werden.

Die Contract Negotiation kann durch den rekursiven Alogrithmus `negotiate` in Listing 3.4 umgesetzt werden. Der erste Eingabeparameter ist eine Menge nicht-funktioneller Minimalbedingungen (Constraints) bezüglich eines Systemsknotens der Form

$$(OperationQuality, SystemNode) \mapsto Amount$$

. Auf diese Weise kann gefordert werden, dass eine Komponente, die auf dem benannten Systemknoten verfügbar ist, einen bestimmten Minimalbetrag einer Qualitätscharakteristik erfordert, insofern dies im Vertragsmodell festgelegt wurde. Der zweite Parameter ist eine Variante, die initial leer ist (keine Schnittstellen wurden mit Implementatoren belegt).

In Zeile 6 wird zuerst überprüft, ob alle Bedingungen erfüllt sind. Ist dies der Fall, so ist die Variante vollständig und wird ausgegeben. Ansonsten wählt der Algorithmus in Zeile 8 eine beliebige unerfüllte Bedingung aus und entfernt diese (Zeile 9) aus der Menge unerfüllter Bedingungen. Anschließend, in Zeile 10 werden alle möglichen `Usage`s erzeugt,

die die jeweilige Bedingung erfüllen. Dies geschieht durch einen einfachen Suchalgorithmus, der Komponenten auf allen Systemknoten in allen verfügbaren Zuständen zu der jeweiligen funktionellen Abhängigkeit ermittelt und dann die Erfüllung der nicht-funktionellen Bedingungen prüft. Zu jeder dieser Möglichkeiten kann eine neue Variante konstruiert und nach neuen, bisher unerfüllten Bedingungen durchsucht (Zeile 12f.) werden. Nun wird, in Zeile 15 ein rekursiver Aufruf mit der erweiterten Variante erzeugt.

Funktionelle Bedingungen, wie sie initial durch den Workflow impliziert werden, können durch eine binäre Pseudocharakteristik 'default' und eine Pseudooperation mit gleichem Namen ausgedrückt werden. So lassen sich die Abhängigkeiten des in Abschnitt 3.2.1 gewählten Beispiels wie folgt definieren:

$$(default(IC.default), FascadeNode) \mapsto 1[]$$

. Diese Bedingung beschreibt die funktionelle Abhängigkeit des Workflows von der Schnittstelle `IC`. Der Skalar hat keine Einheit ('[]'). Nichtfunktionelle Bedingungen werden während der Contract Negotiation für die unterschiedlichen Performancezustände der Komponente `C` generiert (siehe Verträge 3.1 und 3.2). Sie lauten je nach der gerade im Berechnungsschritt betrachteten Implementierung der Schnittstelle `IR` :

$$(qualityIR(IR.operationR), AA) \mapsto 0.1[]$$

und

$$(qualityIR(IR.operationR), BB) \mapsto 0.1[]$$

3.3.4. Bewertung der Varianten

Die Varianten können bereits nach diesem Berechnungsschritt einer Vorbewertung unterzogen werden, welche die Bestimmung einer pauschalen Vorziehungswürdigkeit in folgenden Verarbeitungsschritten erlaubt. Die Bewertung basiert zunächst auf dem schon berechenbaren Nutzen (*utility*) der Variante. Grundlage der Nutzwertbestimmung ist die Festlegung von Prioritäten bezüglich sichtbarer Qualitätscharakteristiken durch den Benutzer im Kontext bei Beginn des Planungsvorganges. Sichtbar sind sämtliche Qualitäten, die durch Komponenten zur Verfügung gestellt werden, deren Schnittstellenoperationen direkt durch den Workflow aufgerufen werden. Da aber Charakteristiken unterschiedlichen Dimensionen angehören können, sind deren absolute Beträge nicht vergleichbar. Um die Vergleichbarkeit herzustellen, ist es notwendig die Charakteristiken einer Normierung zu unterwerfen. Diese kann bei der Annahme der Existenz einer Einheit der jeweiligen Dimension mit linearer Skala aufgrund eines festen systemweiten Maximalwertes durchgeführt werden. Dieses Maximum ist über den in sämtlichen Verträgen festen und abgeleitenen (`Translations`), angebotenen Werten zu bestimmen. Sei $0 \leq p(c_i) \leq 1$ die festgelegt Prioriät der Charakteristik c_i und $q(c_i, v)$ der Wert der dieser Charakteristik

bei Verwendung der Variante v, sowie $c_{i_{max}}$ deren systemweites Maximum, dann ist der Nutzwert der Variante

$$u(v) = \sum_{i=1}^{n} \frac{p(c_i) * q(c_i, v)}{c_{1_{max}}}$$

.

3.3.5. Simulation

Jeder Partition wurden eine Reihe von möglichen Varianten zugeordnet. Werden die Partitionen in das Verhaltensmodell transformiert, kann für jede der Varianten ein Zeit- und Energieaufwand durch Simulation bestimmt werden. Die Transformation wird durch den in Listing A.1 gezeigten Algorithmus durchgeführt. Im Wesentlichen wird eine Transformation von `Invoke`- zu `Call`-, `Sleep`- zu `Run`- und `Flow`- zu `Parallel`-Objekten vollzogen. Zudem werden aus den erzeugten sequentiellen Tasks `Sequence`-Objekte generiert. Im Fall paralleler Abschnitte ist ein rekursiver Aufruf des Algorithmus notwendig, um auch aus den nebenläufigen «Threads» Sequenzen zu erstellen.

Für die Simulation selbst sind zwei weitere Algorithmen nötig. Neben der eigentlichen Simulation (Listing A.3) erzeugt eine Funktion `schedule` (Listing A.2 einen Abhängigkeitsgraphen über die in Warteschlangen eingereihten `Workload`-Items. Mithilfe dieses Graphen können die sequentiellen Ausführungen forciert und die Warteschlangen miteinander synchronisiert werden. Außerdem wird beim Scheduling der Kontext des jeweiligen Items gesetzt und dieses dabei repliziert (dies entspricht einer Instanziierung des modellierten Items auf die Simulationslaufzeitebene). Der Kontext eines erzeugten Items enspricht immer dem des aufrufenden Items. Nur im Fall der `With`-Items wird dieser Kontext zusätzlich durch den dort definierten überschrieben.

Die Simulation ist zeitdiskret und wird auf Modellebene durchgeführt. Zu jedem Simulationszeitpunkt werden Items gesucht, deren Startzeitpunkt bereits abgelaufen ist und die keine unverarbeiteten Abhänigkeiten im Abhängigkeitsgraphen besitzen. Für jede Ressource werden jeweils immer nur so viele Items gleichzeitig entnommen, wie aufgrund der Modellierung von `MTProperty`-Objekten im Verhaltensmodell möglich sind, oder, wenn keine Modellierung vorgenommen wurde, eine pauschale Anzahl. Im Fall einer Ressource wird dazu von Single-Threading ausgegangen, Komponenten können beliebig viele nebenläufige Operationen durchführen. Zudem wird der Simulationszeitschritt mit den `consume`-Verbräuchen der aktiven Objekte verrechnet, so dass am Ende der Vorhersage ein Gesamtenergieverbrauch und ein Zeitverbrauch zur Verfügung steht. Da die Simulation mit jeweils einer Variante v und einem Teil-Workflow p parametrisiert wurde, wird der bestimmte Energieverbrauch im Folgenden als

$$E_O(p, v)$$

bezeichnet. Aus den Energieverbräuchen eines aus diesen Teil-Workflows zusammengesetzten Workflows kann später der zu minimierende Energieaufwand des möglichen Konfigurationsplanes (bisher ohne Betrachtung der Aufwände für Zustandswechsel) bestimmt werden.

3.3.6. Rekombination der Partitionen

Der nächste Schritt ist das Wiederzusammensetzen des Workflows durch Kombination aller Partitionen und jeweils assoziierter möglicher Varianten. Ein Kombination ist gültig, wenn sie sämtliche Tasks des Ursprungsworkflows enthält und keine zwei Partitionen einen gemeinsamen Task beschreiben. Auch die Kombination besitzt eine ungünstige Zeitkomplexität und wird durch eine maximale Anzahl zu erzeugender Partitionen begrenzt. Die Kombination ist eine Sequenz von Partitionen, aus denen der Workflow zusammenengesetzt werden kann sowie eine fixe Zuordnung einer Variante zu jeder dieser Partitionen. Insofern zwei aufeinander folgende Partitionen unterschiedliche Varianten zugeordnet wurden, muss eine Rekonfiguration zwischen diesen stattfinden. Die Rekonfiguration verursacht durch die beschriebenen Zustandsübergänge des Vertragsmodells sowie der (`Un-`)`DeploymentBehavior`-Definitionen des Verhaltensmodells zusätzlichen Aufwand. Während ersterer Aufwand durch direkte Entnahme aus dem Vertragsmodell extrahiert werden kann, wird der Aufwand des Verhaltensmodell erneut simuliert. Beide Aufwandsquellen bestimmen zudem eine Zeitspanne, die für die für die Rekonfiguration notwendig ist. Die maximale Zeitspanne der beiden Aufwände kann ebenfalls durch das Vertragsmodell auf eine Grundlast abgebildet werden. Sei

$$c = (p_0, v_0) \dots (p_i, v_i) \dots (p_n, v_n)$$

eine Kombination der Länge $n + 1$ aus Partitionen p_i, denen jeweils eine Variante v_i zugeordnet wurde. Wenn $E_R(v_s, v_e)$ der Energieaufwand zur Rekonfiguration zwischen Varianten v_s und v_e ist sowie $t_R(v_s, v_e)$ dessen Zeitaufwand und $E_B(v, t)$ der Energieaufwand, der über die Zeitspanne t bei Konfiguration v anfällt , dann ist

$$\overline{E_R(c)} = \sum_{i=0}^{n-2} (E_R(v_i, v_{i+1}) + E_B(v_i, t_R(v_i, v_{i+1})))$$

der Gesamtenergieaufwand der Rekonfiguration. Es ist außerdem zu beachten, dass durch die Partitionierung der parallelen Workflowabschnitte unterschiedliche Sequenzfolgen der gleichen Kombination von Partitionen möglich sind. Bei einer vollständigen Betrachtung muss der Aufwand für jede Folge eigenständig bestimmt werden.

3.3.7. Auswahl des optimalen Plans

Auf Grundlage der bestimmten Nutzwerte und Energieaufwände können sämtliche Kombinationen nun bewertet und eine optimale Kombination ausgewählt werden. Diese optimale Kombination entspricht dem Konfigurationsplan; durch die Zuweisung einer Variante für jede Partition kann auch für jeden Task des Workflows die jeweils zu installierende Konfiguration ermittelt werden. Für die Kombination

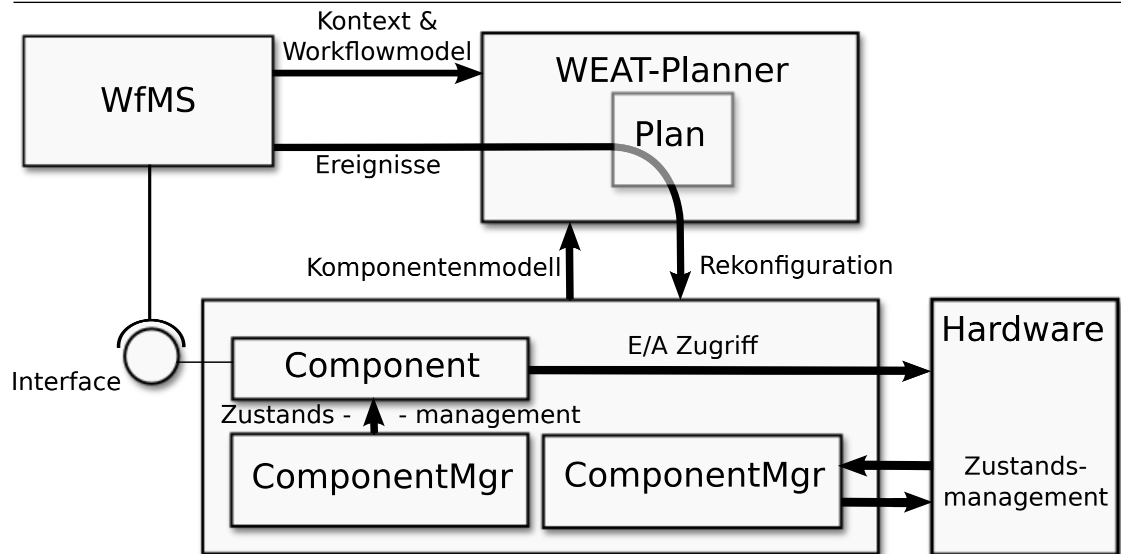
$$c = (p_0, v_0) \dots (p_i, v_i) \dots (p_n, v_n)$$

gilt:

$$efficiency = \frac{\sum_{i=0}^{n}(u(v_i))}{\overline{E_R(c)} + \sum_{i=0}^{n}(E_O(p_i, v_i))}$$

4. Implementierung

Abbildung 4.1. Architektur



Dieses Kapitel beschreibt, wie das vorgestellte Konzept prototypisch implementiert wird. Anfangs wird in Technologien eingeführt, die als Basis der Implementierungen dienen. Das Komponentensystem, welches durch WEAT verwaltet werden soll, muss mit zusätzlichen Schnittstellen ausgestattet werden, die eine Kontrolle durch das Planungs- und Konfigurationsmodul gestattet. Obwohl das System damit einen begrenzten Whitebox-Charakter erhält, bleibt es auch völlig unabhängig ausführbar. Das Planungsmodul und das zugreifende WfMS arbeiteten jeweils auf einer weiteren Schicht. Die Schnittstelle des Komponentensystems wird möglichst abstrakt definiert und das Planungsmodul bleibt weitgehend zustandslos, so dass Fehler zwischen den Schichten kaum übertragen werden können. Ein instabiles Workflowsystem kann im Fehlerfall neugestartet werden, ohne andere Systemteile zu invalideren. Letztendlich erhält das System eine graphische Benutzeroberfläche, in der versucht wird durch Graphen die unterschiedlichen Planungsinformationen darzustellen.

Architektur

Die zur Modellerzeugung, Planung und Konfiguration notwendige Architektur zeigt Abbildung 4.1. Das WfMS überträgt den zu planenden Workflow mitsamt des zugehörigen Kontextes an das Planungssystem. Nach Abschluss der Planung wird mit der Ausführung der Workflows begonnen. Dabei signalisiert das WfMS Start und Ende einer Task-Ausführung dem Planungssystem, das daraufhin, auf Grundlage des erzeugten Plans, das verwaltete Komponentensystem rekonfiguriert. Da während der Planerzeugung eventuell mögliche parallele Partitionen im optimalen Plan erzeugt wurden, welche die nebenläufige Ausführung einer genau fixierten Menge von Tasks definieren, muss das Planungssystem

die synchrone Ereignissignalisierung so lang blockieren, bis der früheste Startzeitpunkt der Partition überschritten wurde. Dieser Zeitpunkt kann anhand der während der Simulation bestimmten Zeitaufwände jeder Partition ermittelt werden.

Das Komponentensystem verwaltet neben Softwarekomponenten zusätzliche *Manager*. Ein solcher `ComponentMgr` steuert den im jeweiligen Vertrag einer Softwarekomponente modellierten Zustand oder den einer Hardwareressource. Die Komponente, insofern sie aktiviert ist, stellt eine Schnittstelle zur Verfügung, über die das WfMS direkt nach Abschluss der Ereignissignalisierung auf den jeweiligen Dienst zugreifen kann. Eine weitere Aufgabe das Komponentensystems ist die Introspektion – die Erzeugung des Komponentenmodells, welches vor der Planberechnung vom System angelegt wird.

4.1. Technologieüberblick

Java Die gewählte Basistechnologie ist Java[1]. Einerseits bietet diese Sprache und die mit ihr einbegriffene Technologie eine breite Plattformunabhängigkeit sowie eine große Auswahl an Softwarebibliotheken, die für die Implementierung verschiedener testbarer Komponenten hilfreich sind. Andererseits basiert bereits OSPP auf Java, somit wird keine Technologiebrücke zwischen dem WfMS und den zu entwickelnden Komponenten benötigt. Ein Nachteil von Java ist, dass die Kontrolle der durch das Workflow-EAT gesteuerten Hardwareressourcen nur durch zusätzliche Werkzeuge außerhalb der Java Virtual Machine (JVM) funktionieren kann.

(D)OSGI Desweiteren verwendet OSPP die OSGI[2]-Implementierung Equinox[3] der Eclipse - Foundation. OSGI ist ein Komponentenframework für lokale Komponenten. Das von OSGI definierte Lebenszyklusmodell bietet bereits Möglichkeiten zur (Un-)Deployment der verwalteten Komponenten. Um darüber hinaus verteilte Systeme zu betreiben, werden WebServices verwendet, die auf SOAP [W3C00] und WSDL [W3C01] basieren. Der WebService-Stub wird durch Apache CXF/DOSGI[4] implementiert.

Scala Während Java eine imperative und objektorientierte Programmiersprache ist, wurden bereits einige Sprachen mit abweichenden Konzepten für die JVM definiert. Die Compiler dieser Sprachen erzeugen aus dem Programmcode Java-Klassendateien. Scala[5] ist eine funktionale, objektorientierte Sprache. Sie ist wesentlich flexibler und anpassbarer als Java, wordurch ihre Sprachkonstrukte ausdrucksstärker sind und der Beschreibungsaufwand von Algorithmen sich im Verhältnis zu Java oft nur noch auf einen kleinen Teil beschränkt. Dies macht, insofern der Entwickler konventionell programmiert, den Code verständlicher und verringert die Zeit, in der die Software geändert werden kann. Eine besondere, in den Scala-Sprachkern aufgenommene Funktion ist die Unterstützung

[1] http://www.java.com
[2] Open Service Gateway Initiative, http://www.osgi.org
[3] http://www.eclipse.org/equinox
[4] http://cxf.apache.org/distributed-osgi.html
[5] http://www.scala-lang.org

von internen und externen *Domain Specific Languages* (DSLs). Interne DSLs werden durch die Definition gewöhnlicher Scala-Funktionen gebildet, die Klammersyntax vermeiden und teilweise wie natürliche Sprachen wirken. Diese Form von DSLs werden direkt im Scala-Code verwendet und sind eng an die Möglichkeiten der Programmiersprache gebunden. Externe DSLs dagegen sind völlig frei definierbar und werden von in Scala erzeugten Parsers aus Textdateien eingelesen. Die Konstruktion dieser Parser wird durch die Scala-API stark vereinfacht. Mithilfe von *Parser Combinators* ist es möglich, verschiedene, für bestimmte Tokensematiken vordefinierte, Parserreferenzen einfach zu konkatenieren. Sobald eine solche Konkatenation zu einem erfolgreichen Parsing-Ergebnis führt, lassen sich die eingelesenen und bereits strukturierten Tokensemantiken in ein Objektmodell umwandeln. Die Parser-API wird in der WEAT-Implementierung zur Verarbeitung der benutzerdefinierten Modelltexte eingesetzt.

Eclipse RAP OSPP besitzt eine Web-Oberfläche, die mit Hilfe des Eclipse-Runtime-Projektes RAP (Rich Ajax Platform) implementiert wurde. RAP ermöglicht es, eine Anwendung mit Eclipse RCP - Oberfläche im Webbrowser zu betreiben. Dazu ist keine Änderung am Anwendungskode notwendig, es muss lediglich eine andere Implementierung der RCP-Oberflächenelemente geladen werden (Single-Sourcing). Ein OSGI-integrierter Web-Server (Jetty) ermöglicht dann den Zugriff auf die Webanwendung. Die OSPP-Oberfläche wird durch die Visualisierung des Planungsvorganges erweitert.

4.2. Implementierung des Einheitenmodells

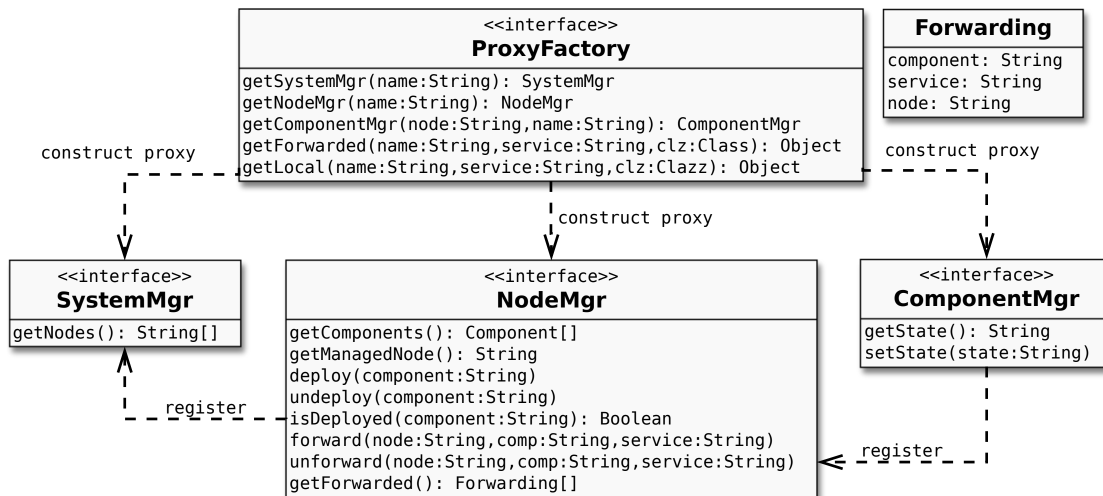
Um die im Einheitensystem definierten Klassen zu realisieren, kommt die auf JSR-275 [Pro] basierende Implementierung JScience[6] zum Einsatz. Die JAVA-Measurement-API bietet wie im Konzeptmodell die Klassen `Unit` und `Dimension`. Eine physikalische Dimension kann elektrische Stromstärke, Länge, Masse, Temperatur, Zeit, Stoffmenge, ein Skalar oder eine Verknüpfung dieser sein. Die Charakteristiken eines Informationssystems basieren zumeist auf den zeitlichen und skalaren (sämtliche Informationseinheiten sind skalar) Dimensionen. Desweiteren werden durch JScience die meisten SI und Nicht-SI-Einheiten definiert. Auch die Klasse `Amount` wird zur Verfügung gestellt, die durch eine Maßeinheit typisiert wird. Mithilfe der Programmierschnittstelle können Beträge einfach umgerechnet werden. Für die Verarbeitung von in Zeichenketten dargestellten Beträgen, Einheiten und Dimensionen wird auch Parsing angeboten.

4.3. Verwaltetes Komponentensystem

Abstrakte Schnittstellen Die Laufzeitumgebung wird als zusätzliche Abstraktionsschicht um das Komponentensystem angelegt. Da das System sowohl vom konkret verwendeten WfMS als auch vom Komponentensystem entkoppelt funktionieren soll, werden dessen Schnittstellen zunächst unabhängig von Implementierungen definiert.

[6] http://jscience.org/

Abbildung 4.2. Schnittstellen des Komponentensystems



Die Struktur des Komponentensystems entspricht dem in 3.2.1 entwickelten Modell. Dabei werden Knoten und Komponenten durch Manager (`NodeMgr` und `ComponentMgr`) verwaltet. Ein zentraler `SystemMgr` dient als Verzeichnisdienst für sämtliche verfügbare Knoten. Die Schnittstellen werden in Abbildung 4.2 dargestellt.

Die Managerkomponenten registrieren ihre Instanz bei der jeweils auf der nächsthöheren Ebene gelegenen; die `NodeMgr` zudem an ihren «Superknoten» (Assoziationen `accesses` im Komponentenmodell). Die Schnittstellenmethoden, die zur Registrierung notwendig sind, werden erst durch die erweiterten Schnittstellen (s.u.) definiert, da sie sich nicht in der Domäne des WEAT-Komponentenmodells befinden. Entscheidend ist hier nur die Zurverfügungstellung der Information, um später das Komponentenmodell zu gewinnen. Komponenten besitzen zwei getrennte Zustandsräume. Einerseits wird ihr Deploymentzustand durch den Container kontrolliert, welcher wiederum der Steuerung durch den `NodeMgr` unterliegt. Andererseits lassen sich Zustände aus dem Vertragsmodell über die Schnittstelle `ComponentMgr` steuern. Der `NodeMgr` erlaubt den Zugriff auf Informationen über seinen Namen, seine untergeordneten Knoten, die verwalteten Komponenten und deren Deploymentzustand. Im Fall, dass der Dienst einer Komponente in einem Knoten eingesetzt werden soll (`Usage`), wird ein `Forwarding` erzeugt. Der Dienst, der durch das Forwarding von einem Systemknoten in einen anderen weitergeleitet wird, muss dann vom `NodeMgr`[7] im Container verfügbar gemacht werden. Die `ProxyFactory` ermöglicht die Implementierung eines verteilten Komponentensystems. Sie erzeugt aus den globalen Identifikatoren Proxyobjekte zum Zugriff auf den jeweiligen Dienst. Die Methode `getLocal` ermöglicht dem System auch den Zugriff auf nicht-weitergeleitete Dienste, die aber lokal zur Verfügung stehen.

[7] Die Parameterabkürzungen in den Klassen `NodeMgr` und `ProxyFactory` stehen für (n)ode, (c)omponent und (s)ervice.

Tabelle 4.1. Konvention der WebService-Addressen

Schnittstelle	WebService-Adressmuster
SystemMgr	http://<Knoten>/SystemMgr
NodeMgr	http://<Knoten>/NodeMgr
ComponentMgr	http://<Knoten>/<Komponente>/ComponentMgr
Schnittstelle (local)	http://<Knoten>/local/<Komponente>/<Schnittstelle>
Schnittstelle (forwarded)	http://<Knoten>/forwarded/<Schnittstelle>

DOSGI-Erweiterungen Zur Implementierung der Schnittstellen werden diese in fünf OSGI-Bundles modularisiert. Die oben einfgeführten Schnittstellen beinhalten ein eingeständiges Bundle *RTE (Runtime Environment)*, so dass das Planungsmodul, welches auf diese zugreift, unabhänig von der Implementierung des Komponentensystems arbeiten kann. Das Bundle *DOSGI-RTE* spezialisiert die Schnittstellen und fügt die notwendigen JAX-WS[8]-Annotationen hinzu. Es enthält zudem die DOSGI-Implementierung der `ProxyFactory`. `SystemMgr` und `NodeMgr` werden in seperaten Bundles implementiert, um sie eigenständig ausführen zu können.

Sämtliche durch das Komponentensystem verwaltete Komponenten müssen einen `ComponentMgr` implementieren und diesen über die OSGI-Service-Registry am `NodeMgr` anmelden. Da OSGI-Manifeste zwar Auskunft über ihre Paketabhängigkeiten, aber nicht über benötigte und angebotene Schnittstellen geben, werden diese um die Einträge *Cool-Implements* und *Cool-Depends* erweitert, welche ebendiese Informationen dem `NodeMgr` zur Verfügung stellen.

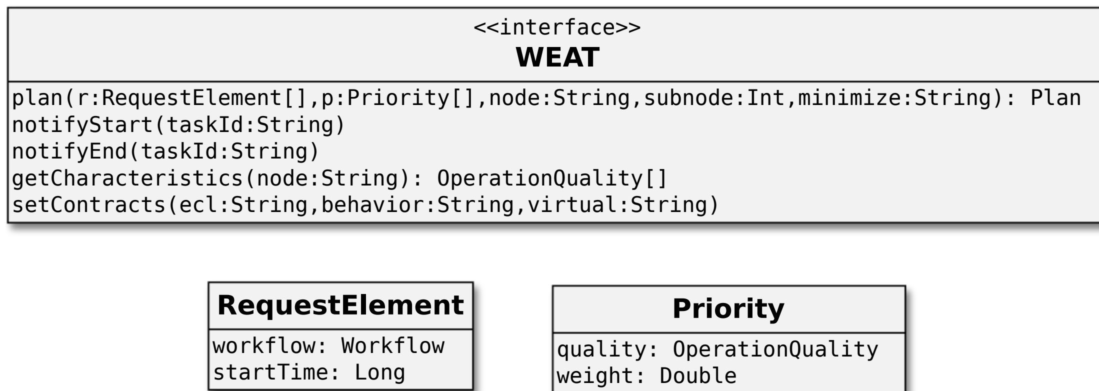
Um eine möglichst einfache Kommunikation zwischen den verschiedenen Objekten zu gewährleisten, werden die DOSGI-Endpoints auf konventionierten URLs publiziert, welche dem Muster in Tabelle 4.1 folgen. Eine Vereinfachung ist die Verwendung von Hostnamen und TCP-Port als Knotenbezeicher[9]. Durch Verwendung der Portnummer können mehrere Knoten auf einem Netzwerkknoten verwaltet werden.

4.4. Modelle und Parser

Die Modelle (siehe 3.2) werden in Scala definiert. Während der Workflow bereits als Modell im WfMS vorhanden ist und das Komponentenmodell durch die o.g. Schnittstellen konstruiert werden kann, müssen Parser das Vertrags- und Verhaltensmodell erzeugen. Zu diesem Zweck werden *Combinated Parsers* eingesetzt. Um bereits definierte Objekte anhand eines Referenzbezeichners wiederzufinden, wird ein Scala-Objekt (dies entspricht einer Java-Klasse nach dem Singleton-Entwurfsmuster) namens `Model` als Assoziativspeicher (Cache) eingesetzt. Die Einführung folgender Namensräume, ermöglicht in dieser Modellbasis gleiche Objektnamen in unterschiedlichen (Teil-)domänen des Gesamtmodells:

[8] Java API for XML - Web Services, http://jcp.org/aboutJava/communityprocess/final/jsr224
[9] Beispiel für einen Knotenbezeicher: localhost:9000

Abbildung 4.3. WEAT-Schnittstelle



- `component` für Komponenten, Schnittstellen und Systemknoten
- `characteristic` für Charakterisitiken
- `profile` für ECL*-Verträge
- `behavior` für Verhaltensdefinitionen

Ein Nachteil dieser Vorgehensweise ist die fehlende Möglichkeit der Vorwärtsreferenzierung, da ein Objekt erst gefunden werden kann, wenn es vorher bereits durch den Parser erzeugt wurde. Dies ist auch der Grund, warum die Modellierungssprachen beispielsweise eine Reihenfolge von Zuständen und Transitionen bei der Definition von ECL-Verträgen vorraussetzen.

4.5. Planungskomponente

Vor den in 3.3 aufgeführten Arbeitsschritten müssen sämtliche Informationen aus dem erzeugten Objektmodell verarbeitet werden. Die Algorithmen werden vollständig in Scala umgesetzt. Der Zugriff auf das Planungssystem erfolgt über einen WebService, der die einkommenden Ereignisse des WfMS an das Scala-Modul propagiert. Dieser WebService wird von einem weiteren Bundle *WEATServer* ausgeführt und besitzt die in Abbildung 4.3 gezeigte Schnittstelle. Die ebenfalls dort gezeigten Klassen ermöglichen den Informationsaustausch mit dem Klientsystem. Mithilfe einer `Priority` lässt sich das Gewicht einer Usermetrik festlegen. Die Klasse `RequestElement` assoziiert jeden zu planenden Workflow mit einem Startzeitpunkt. Neben dieses Informationen benötigt die `plan`-Funktion außerdem den Ort (Systemknoten), von dem das WfMS auf die Dienste des Komponentensystems zugreift sowie die zu minimerende Aufwandsgröße (`minimize`).

Der Zugriff von Java-Code auf Scala-Komplilate ist zwar ohne weiteres möglich, aber dennoch umständlich, da Klassennamen oft aus Entwicklersicht nicht intuitiv zuordenbar

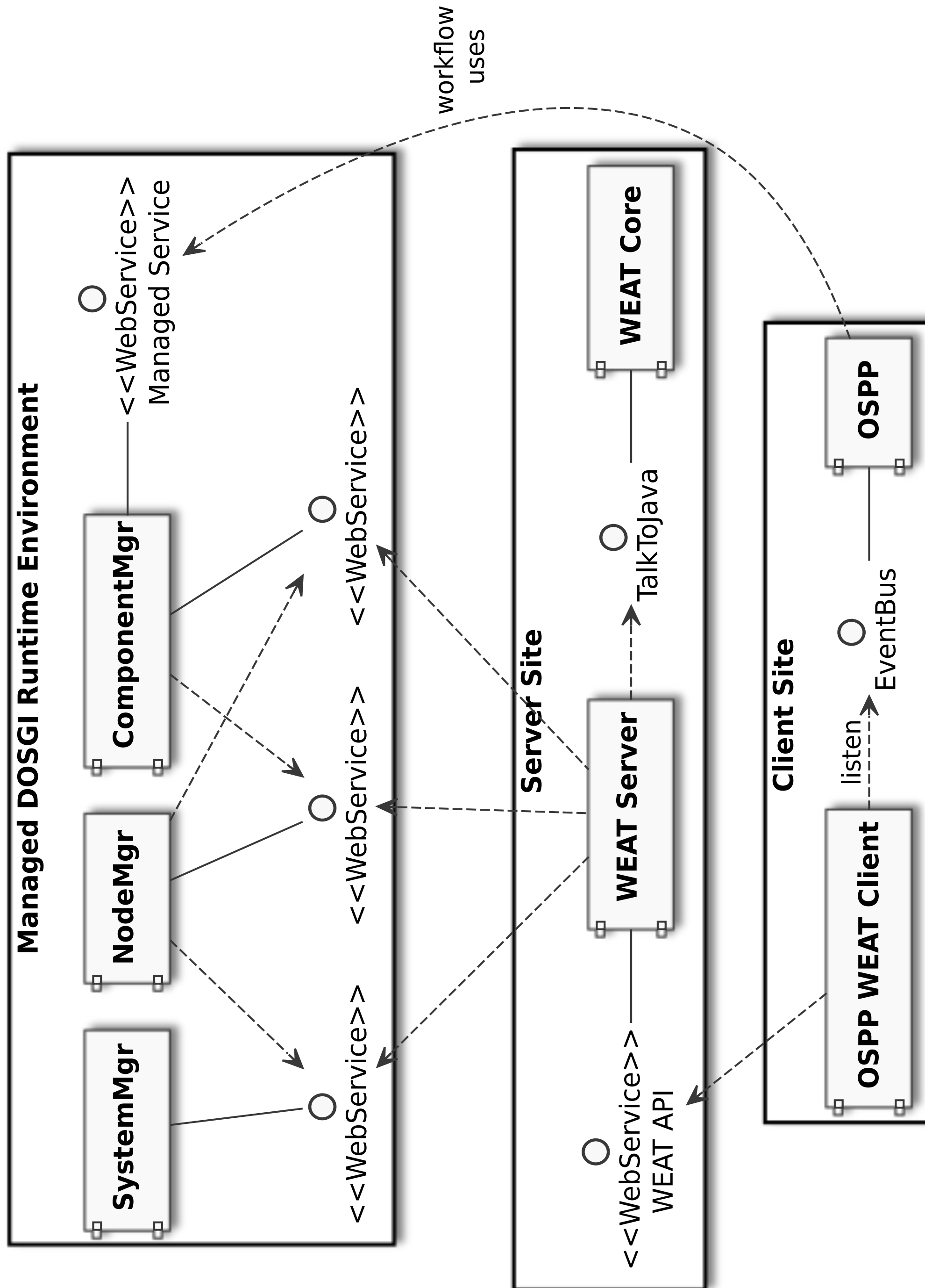
sind und sämtliche Funktionen in eigenständige Klassen kompiliert werden um sie als *Closures* verwenden zu können. Um den Zugriff zu erleichtern, wird ein Scala-Adapter-Singleton `TalkToJava` implementiert, welches eine schlanke Schnittstelle besitzt, die Java-Parameter verarbeitet.

4.6. Integration von OSPP

OSPP kann nun von WEAT entkoppelt ausgeführt werden. Das von OSPP verwendete rollenbasierte Workflowmodell ist in das WEAT-Workflowmodell zu übersetzten. Mithilfe eines synchronen Event-Bus (Klassenname `MonitorManager`) können ausgeführte Tasks WEAT-Aufrufe auslösen. Bei dem Start einer Neuen Workflowinstanz wird zunächst ein Planungsaufruf an den WEAT-Server geschickt, dieser speichert anschließend den gewonnenen Ausführungsplan. Vor Ausführung eines Tasks wird dessen Bezeicher jeweils an die WEAT-Schnittstelle gemeldet und der Server reagiert mit einer Rekonfiguration. Die gesamte Architektur wird in einer Übersicht in Abbildung 4.4 gezeigt.

Die WEAT-Benutzeroberfläche (Abbildungen B.1 und B.2) ist eine Erweiterung der OSPP-Oberfläche und somit eine Eclipse-RAP-Anwendung. Sie beinhaltet mehrere Sichten (*Views*) zur Steuerung von WEAT, zur Eingabe der Modelltexte und zur Visualisierung der WEAT-Rechenergebnisse mithilfe von Graphen.

Abbildung 4.4. WEAT-Architektur



5. Evaluation

Im folgenden Kapitel soll demonstriert werden, wie ein konkretes System modelliert und ein aus den Modellen erzeugter Konfigurationsplan durch die Planungskomponente bestimmt wird. Ziel ist es, nachzuweisen, dass diese Modellierung grundsätzlich möglich ist und die während der Planung vorausgesagten Zeit- und Energieverbräuche nur begrenzt von den tatsächlich gemessenen abweichen.

Während Komponenten- und Vertragsmodell sichere und statische Zusicherungen repräsentieren, können im Verhaltensmodell unterschiedliche Detaillierungsgrade erreicht werden. Je detaillierter das Modell ist, desto höher wird der Simulationsaufwand sein. Ist das Modell hingegen ungenauer, können die berechneten Energiemengen stark von den realen abweichen. Deshalb ist es notwendig das erzeugte Modell durch Versuche zu evaluieren und verbessern, bis eine hinreichende Aussagekraft erreicht wird.

Im anschließenden Beispiel werden nicht sämtliche vorgestellte Funktionen der obigen Modelle verwendet. Zudem ist es bisher auch noch nicht möglich, die Verbräuche der benutzten Systemressourcen in jedem Fall zu bestimmen. Einige Hersteller bieten zwar Spezifikationen ihrer Produkte mit Verbrauchsdaten an, die aber häufig nicht genügend detailliert für die Erstellung des Simulationsmodells sind. Gleiches gilt für genaue Performancedaten, die erheblichen Einfluss auf Laufzeiten der durchgeführten Operationen haben. Somit sind viele dieser Informationen nur durch Messungen zu bestimmen. Im Folgenden wird gezeigt, wie solche Messungen grundsätzlich vorgenommen werden können und wie die entsprechenden Modelle abzuleiten sind. Das durch die dargestellte Methodik erzeugte Modell ist jedoch aufgrund der bisher unzureichenden Messverfahren und der ungenügenden Herstellerangaben sowie einer zur Veranschaulichung notwendigen Vereinfachung nur ein ungenauer erster Entwurf, der in der Praxis durch mehrere Vergleichs- und Iterationszyklen (siehe Autotuning-Kreislauf 2.3) verbessert werden muss.

5.1. Anwendungsfall

Zur Veranschaulichung des Konzepts wird ein Anwendungsfall eingeführt, den Abbildung 5.1 zeigt. Der Workflow entspricht dem Kontrollfluss eines Musikerkennungsdienstes. Der Benutzer soll eine MP3-kodierte Tondatei in das System einspeisen und erwartet anschließend die Zuordnung eines Interpreten sowie des Titels des Musikstückes. Es wird davon ausgegangen, dass diese Informationen verloren gegangen sind und durch den Dienst wiederhergestellt werden müssen. Anschließend soll zusätzlich ein passendes Musikvideo heruntergeladen und dem Benutzer zur Verfügung gestellt werden. Es sind vier Schritte notwendig, um den Dienst auszuführen: Anfangs (`Decode`) wird die übermittelte Datei in ein anderes Format umgewandelt, welches durch den Dienst verarbeitet werden kann. Anschließend wird ein *Fingerprint* im gleichnamigen Task erzeugt. Der Fingerprint besteht aus einem semantischen Hashwert, mit dem zwei verschiedene Tondateien verglichen werden und deren Ähnlichkeit ermittelbar wird. Es wird also möglich mit einer durch die Ähnlichkeit implizierten Wahrscheinlichkeit die semantische Gleichheit der Inhalte beider Dateien vorauszusagen. Im nächsten Schritt `Recognize` wird auf diese Weise in einer Datenbank der ähnlichste Fingerprint bereits indizierter Tondateien gesucht. Das Ergebnis sind die in diesem Index enhaltene Metainformationen,

Abbildung 5.1. Anwendungsfall Workflow

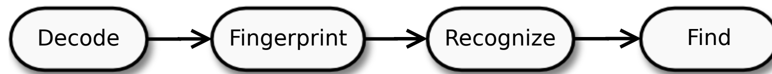
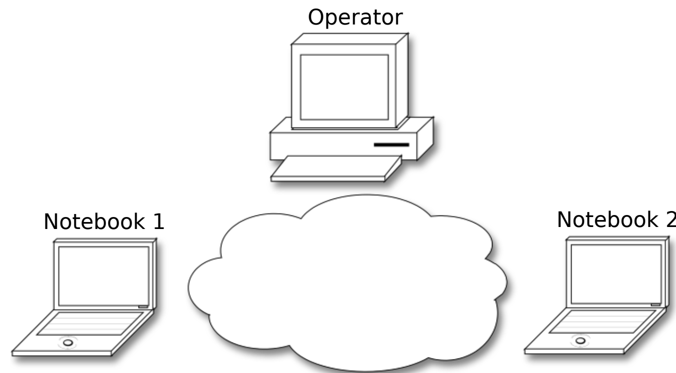


Abbildung 5.2. Versuchsaufbau



die bei der Indizierung gespeichert wurden. Diese Informationen werden anschließend (`find`) zur Suche passender Musikvideos auf einem Videoportal genutzt; das erste dort gefundene Video wird letztendlich heruntergeladen, so dass der Benutzer darauf zugreifen kann.

Der Workflow soll parallel und zeitversetzt durchgeführt werden, um die Partitionierung paralleler Abschnitte zu demonstrieren. Es werden zwei Instanzen ausgeführt: Ersterer beginnt *vier Minuten* nach dem Beginn der Planung, um dieser genügend Rechenzeit einzuräumen, die zweite Instanz *weitere fünf Sekunden später*.

5.2. Versuchsaufbau

Der Anwendungsfall wird auf zwei verschiedenen Rechenknoten durchgeführt, die eine Vergleichbarkeit der Energieeffzienz möglicher Konfigurationen der Systeme zulassen. Die Planungskomponente wird auf einen dritten Knoten ausgelagert, da die Betrachtung der Effizienz der Planungskomponente selbst im gegenwärtigen Entwicklungszustand der benutzen Algorithmen keine Aussagekraft besitzt. Es soll ausschließlich das Verhalten der für die Ausführung des Anwendungsfalls notwendigen Ressourcen und Komponenten modelliert werden, nicht das der Planungskomponente selbst. Die Plattformen sind zwei mobile Rechner. Wie im nächsten Absatz gezeigt wird, können auf diesen die verbrauchten Energiewerte genauer bestimmt werden als in stationären Systemen. Den Versuchsaufbau zeigt Abbildung 5.2. Die Rechner kommunizieren über ein Funknetzwerk, welches ebenfalls eine variable Modellierung von Verbindungsgeschwindigkeiten zulässt. Die Planungskomponente wird auf dem Knoten «Operator» betrieben und steuert durch das Netzwerk die Knoten «Notebook 1» und «Notebook 2». Bei ersterem handelt es sich um das Modell «U100» des Herstellers MSI, «Notebook 2» dagegen ist ein IBM

Lenovo «X41». Jeder dieser Rechenknoten wird als einzelner `SystemNode` repräsentiert. Die WebService-Schnittstellen werden auf dem TCP-Port 9000 publiziert, so das die Knoten die Bezeichnungen `notebook1:9000` und `notebook2:9000` tragen.

Beide Plattformen sind stark energieoptimiert. Im Versuch hat dies den Vorteil, dass der Steigerung der Leistungsaufnahme genauer gemessen werden kann, da sie einen größeren Anteil Ruheleistungsaufnahme («Grundlast») als in energieintensiveren Plattformen verursacht. Die genauen Leistungsdaten der Rechner werden im Folgenden modelliert.

5.3. Verbrauchsmessung

Befindet sich ein modernes Notebook im Batteriebetrieb, kann der Energieverbrauch durch einen in den Akkumulator integrierten Sensor gemessen werden. Dieser stellt genauere Daten zur Verfügung, als es ein zwischen Netzteil und Stromnetz geschaltetes Messgerät könnte, da bei letzterem häufig Schwankungen und Verzögerungen (beispielsweise durch die Selbstinduktion des Spannungswandlers) entstehen.

Der eingebettete *Smart Battery Sensor* (SMB) verursacht laut Herstellerangaben eine maximale Messabweichung von 10%. SMB implementiert die Schnittstelle *Smart Battery Data* und kommuniziert durch den System Management Bus (SMBus). Dieser wird von Betriebssystemen wie Windows und Linux untersützt, die auf dessen Grundlage Batterieanzeigen zur Verfügung stellen. Auch die ACPI-Implementierungen verwenden Informationen des SMB. So sind in Linuxsystemen SMB-Informationen in '/proc/acpi/-battery' abrufbar. Wird ein fester Abfragetakt festgelegt, kann so ein Energiewert für ein beliebiges Intervall bestimmt werden.

5.4. Hardwareressourcen

Beispielhaft werden für Prozessoren und Festplatten beider Rechenknoten Modelle erstellt und `ComponentMgr` implementiert. Zudem werden die Netzwerkadapter als virtuelle Ressourcen beschrieben, da deren Leistung ebenfalls Einfluss auf die Dauer der Operationen ausübt. Um die Leistungen mehrerer Ressourcen zu vergleichen, kann eine Zeit gemessen werden, in der die Ressource ein festes *Lastäquivalent* verarbeitet. Einen solcher Vergleichsworkload kann entweder einer bekannten Informationseinheit, wie beispielsweise einer Datenmenge, entsprechen oder vom Modellierer definiert sein.

Jeder Ressourcenoperation können desweiteren unterschiedliche Arten von Verhaltensmustern zugewiesen werden. Einerseits ist es möglich die Ressource über ein definiertes Zeitintervall mit einer konstanten und über das Intervall verteilten Last zu beanspruchen. Die anteilige Beanspruchung spiegelt sich in der ebenfalls anteiligen Last-Leistungsaufnahme während des Intervalls wieder. Andererseits kann ein Verhaltensmuster beschrieben werden, welches eine möglichst schnelle Verarbeitung eines gegebenen Vielfachen des gewählten Lastäquivalents verlangt.

Neben dem Verhaltensmodell werden die Leistungsaufnahmen ohne Beanspruchung sowie einige Qualitäten der Ressourcen im Vertragsmodell beschrieben. Im vorgestellten

Anwendungsfall spielen diese Informationen jedoch nur insofern eine Rolle, als mit ihnen definiert wird, in welchem Zustand eine aktive Ressource benötigt wird.

5.4.1. Prozessoren

Die Leistungsaufnahme eines Prozessors wird von den Herstellern nur ungenau beschrieben. Zwar existiert ein Maß für die maximale Leistungsaufnahme im gewöhnlichen Betrieb (*Thermal Design Power, (TDP)*, jedoch müssen solche Angaben je nach Hersteller unterschiedlich interpretiert werden. Zudem kann auf deren Grundlage keine Aussage darüber getroffen werden, wie die Leistungsaufnahme unter einer anteiligen Beanspruchung des Rechenkerns ist, oder welche Leistungsaufnahme im niedrigsten Frequenzmodus eines Prozessors eintritt.

Für Prozessoren werden in ACPI mehrere passive und aktive Zustände definiert. Da die von den Rechnenknoten zur Verfügung gestellten Dienste immer in Bereitsschaft gehalten werden müssen, werden hier ausschließlich aktive Zustände betrachtet. Zur Vereinfachung wird nur eine Regulierung zwischen dem maximalen und minimalen Zustand sowie den mit diesem verbundenen Spannungen und Frequenzen vorgenommen[1]. Für beide Rechenknoten wird ein CPU-`ComponentMgr` implementiert. Die Manager können jeweils zwischen maximaler und minimaler Taktfrequenz umschalten.

Desweiteren wird davon ausgegangen, dass die Leistungsaufnahme zwischen diesen beiden Zuständen linear ansteigt. Dies entspricht nur entfernt der Realität, da bekannt ist, dass die Spannung quadratische und die Frequenz lineare Auswirkungen auf die Leistungsaufnahme von Prozessoren haben. In der vorausgesetzten Standardeinstellung sind Spannungen Taskfrequenzen einander vom Hersteller fest zugeordnet.

Messung der Leistungsaufnahme und Prozessorleistung

Da bei oben vorgestellter Messmethode der Energieverbrauch am installierten System gemessen wird, kann meist der Beitrag einzelner Hardwareressourcen zum Energieverbrauch nicht genau unterschieden werden. Der Verbrauch des Prozessors wird als die Differenz des Systemverbrauchs unter maximaler Prozessorlast und ohne Last bestimmt. Im letzteren Fall wird zusätzlich die Festplatte aktiviert. Da das System im aktiven Zustand niemals völlig ohne Prozessorlast arbeitet, wird der niedrigste Verbrauch aber tatsächlich bei durchschnittlich 5% der Maximallast bestimmt. Während der Messung treten zudem Verbräuche im Speichersystem, System-Bus und anderen Infrastrukturressourcen auf, die in dieser vereinfachten Messung mit in das Energiemodell des Prozessors einbezogen werden. Ein höherer Detaillierungsgrad wird in Zukunft nur erreicht werden können, wenn Hersteller genauere Werte spezifizieren oder eine exakte Messung an den Ressourcen vorgenommen wird.

Die gemessenen Durchschnittswerte der beiden Prozessoren bei jeweils maximaler und minimaler Taktfrequenz zeigt Tabelle 5.1. Es zeigen sich deutliche Unterschiede zwischen dem als Netbook konzipierten «Notebook 1» und dem älteren kleinformatigen «Notebook 2». Nicht nur ist der Engierverbrauch des ersten Modells in allen Betriebszuständen

[1] In Linux-Distributionen ist dies mithilfe der *cpufreq utils* möglich

Tabelle 5.1 Messungen – Prozessoren

		«Notebook 1» MSI Wind U100		«Notebook 2» IBM Lenovo X41	
Prozessortyp		Atom N270		Pentium M 758 LV	
Thermal Design Power	[W]	2.5		7.5	
Taktfrequenz	[MHz]	800	1600	600	1500
Systemverbrauch Last 5%	[W]	11	11.5	12.8	13.3
Systemverbrauch Last 100%	[W]	12.3	13.5	15.5	20.1
Differenz	[W]	1.3	2	2.7	6.8
Verarbeitungsdauer Lastäquivalent	[s]	172	144	188	75

geringer; es konnte auch eine verminderte Verbrauchsdifferenz zwischen minimaler und maximaler Last sowie Taktfrequenz festgestellt werden. Die Differenz zwischen Minimal- und Maximalverbrauch während eines Betriebszustandes soll ein Richtwert für den im Verhaltensmodell spezifizierten zusätzlichen Lastverbrauch sein. Diese Differenz übersteigt während der Messung niemals die vom Hersteller spezifizierte TDP. Obwohl dies ein Anzeichen für die Validität der Messwerte ist, ist es durch die Einbeziehung gleichzeitig beanspruchter Ressourcen in das Modell sowie durch die unterschiedliche Interpretiertbarkeit der TDP durchaus möglich, dass während der Messungen Differenzen über dieser auftreten.

Die Leistung moderner Prozessoren kann heute kaum noch aus den verfügbaren Taktfrequenzen abgeleitet werden. Zustätzlich haben Caching, Hyperthreading, Pipe-Architekturen, Speicheranbindung und besonders echte Parallelisierung Einfluss auf die maximale Verbarbeitungsgeschwindigkeit. Auch der Vergleich zwischen unterschiedlichen Lastarten wie Fließkommzahlen- und Festwerte(Integer)-berechnungen ist kaum möglich, insbesondere wenn das Programm Cache-optimiert ist. So wurde die Leistung einer CPU bisher in Einheiten wie FLOPS oder MIPS gemessen. Da die Bestimmung dieser Werte und Übertragung auf die durch Programme verursachten Aufwände eher schwierig ist, wird ein einfacher Ansatz gewählt und mithilfe eines Algorithmus zur Berechnung der Zahl π ein Lastäquivalent erzeugt[2]. Es werden insgesamt eine Million Kommastellen in zwei Threads unter Nutzung des Gauss-Legendre-Algorithmus berechnet. Die jeweilgen Zeitverbräuche zeigt ebenfalls Tabelle 5.1.

Modellierung

Listing 5.1 zeigt die ECL'-Profile der Prozessoren. Zur Unterscheidung des Zustandes der CPUs durch abhängige Profile wird anfangs eine Charakteristik `frequency` definiert; die Ressourcen werden nur für den jeweils minimalen und maximalen Performancezustand modelliert. Die Profile definieren keine Energieverbräuche, da die Grundlast zusätzlich durch eine virtuelle Komponente `System` (Listing 5.2) beschrieben werden

[2] es wird der *System Stability Tester* eingesetzt, siehe http://systester.sourceforge.net.

Listing 5.1 ECL'-Profile der Prozessoren

```
characteristic frequency of [1/[T]]

profile CPU1Contract for CPU1{
 state max{
  provides frequency(CPU.compute) = 1.6 [GHz]
 }
 state min{
  provides frequency(CPU.compute) = 0.8 [GHz]
 }
}
profile CPU2Contract for CPU2{
 state max{
  provides frequency(CPU.compute) = 1.5 [GHz]
 }
 state min{
  provides frequency(CPU.compute) = 0.6 [GHz]
 }
}
```

Listing 5.2 Virtuelle Komponenten zur Modellierung der Grundlast

```
resource System1 implements {System}
resource System2 implements {System}
sysnode notebook1:9000 includes {System1}
sysnode notebook2:9000 includes {System2}
```

Listing 5.3 ECL'-Profile zur Modellierung der Grundlast

```
profile System1Contract for System1{
 state max{
  requires frequency(CPU.compute) = 1.6 [GHz]
  consumes power 11.5 [W]
 }
 state min{
  requires frequency(CPU.compute) = 0.8 [GHz]
  consumes power 11 [W]
 }
}
profile System2Contract for System2{
 state max{
  requires frequency(CPU.compute) = 1.5 [GHz]
  consumes power 13.3 [W]
 }
 state min{
  requires frequency(CPU.compute) = 0.6 [GHz]
  consumes power 12.8 [W]
 }
}
```

Listing 5.4 Verhaltensmuster der Prozessoren

```
1  def CPU.utilize in CPU1 = 1[]
2
3  def CPU.runtime in CPU1@max = ((CPU.amount * 0.83[s])/CPU.utilize)
4  when CPU.compute in CPU1@max run CPU.runtime consumes(power (2[W]
       * CPU.utilize))
5
6  def CPU.runtime in CPU1@min = ((CPU.amount * 1[s])/CPU.utilize)
7  when CPU.compute in CPU1@min run CPU.runtime consumes(power
       (1.3[W] * CPU.utilize))
8
9  def CPU.utilize in CPU2 = 1[]
10
11 def CPU.runtime in CPU2@max = ((CPU.amount * 0.43[s])/CPU.utilize)
12 when CPU.compute in CPU2@max run CPU.runtime consumes(power
       (6.8[W] * CPU.utilize))
13
14 def CPU.runtime in CPU2min = ((CPU.amount * 1.09[s])/CPU.utilize)
15 when CPU.compute in CPU2@min run CPU.runtime consumes(power
       (2.7[W] * CPU.utilize))
```

soll. Die Leistung unter Last hingegen wird erst im Verhaltensmodell repräsentiert. Die Transitionsgeschwindigkeit sowie der Energieverbrauch des Zustandswechsels könnte zwar statistisch bestimmt werden, spielt aber bei den hier auftretenden Abweichungen keine Rolle. In Listing 5.3 wurden die Ruheleistungsaufnahmen der Knoten modelliert. Durch die Bedingungen der jeweiligen Taktfrequenz des lokalen Prozessors wird während der Planung der logische Verknüpfung der Grundlast des Systems mit dem Zustand des Prozessors hergestellt.

Die Verhaltensmuster der Prozessoren werden in Listing 5.4 gezeigt. Das Muster `CPU.compute` beschreibt die Auslastung der CPU während einer durch die Funktion `runtime` definierten Laufzeit mit einer anteiligen Auslastung in Höhe des Parameters `utilize`, der den Standardwert «1» besitzt. Wie oben bereits erwähnt wird davon ausgegangen, dass die durch Last verursachte Leistungsaufnahme proportional zur dieser Last ist. Als Lasteinheit wurde ein Wert gewählt, der innerhalb einer Sekunde durch `CPU1` bei minimaler Taktfrequenz verarbeitet werden kann. Sie beträgt 1/172 des in Tabelle 5.1 eingesetzten Lastäquivalents. Die Verhaltensmuster können auf zwei verschiedenen Wegen parametrisiert werden: Erstens kann ein Wert für die Variable `runtime` und/oder `utilize` gesetzt werden, um eine explizte Laufzeit-beschränkte Auslastung zu simulieren. Andererseits kann anstattdessen ein Parameter `amount` übergeben werden, der durch die `runtime`-Funktion in eine Laufzeit umgerechnet wird. Wird im letzteren Fall auch noch der Variable `utilize` ein Wert zugewiesen, führt dies zu einer zusätzlichen Streckung der Verarbeitungszeit.

Tabelle 5.2 Herstellerangaben und Messungen – Festplatten

		«Notebook 1» MSI Wind U100	«Notebook 2» IBM Lenovo X41
Festplattenmodel		ATA WDC WD1600 BEVT-22ZCT0	Hitachi Travelstar HTC 426040G9AT00
Verbrauch Sleep	[W]	0.2	0.11
Verbrauch Idle	[W]	0.85	0.6
Verbrauch Read	[W]	1.6	1.5
CPU-Last Read	[%]	11	4
Verbrauch Write	[W]	1.6	1.7
CPU-Last Write	[%]	17	5
Lesen 1 Megabyte	[ms]	15	38
Schreiben 1 Megabyte	[ms]	37	53

5.4.2. Festplatten

Für die meisten Festplatten exisitieren im Gegensatz zu Prozessoren genauere Hersteller-Spezifikationen. Tabelle 5.2 zeigt diese für die 5-Volt-Betriebsmodi; ein möglicher 3-Volt-Modus wird hier nicht betrachtet. Der implementierte `ComponentMgr` kann die Festplatte in die Modi «sleep» und «idle» versetzen[3]. Um eine Reaktivierung durch Rückschreib- und Loggingzugriffe des Betriebsssystems zu verhindern, müssen diese vorher deaktiviert werden. Die Leistung der Festplatten kann ebenfalls einfacher gemessen werden, die Lasteinheit ist hier ein Megabyte. Die gemessen Daten zeigen, dass «Notebook 2» eine wesentlich performantere Festplatte besitzt. Es ist aber zu beachten, dass sowohl Rechnerinfrastruktur- als auch Dateisystemlatenzen in diese Messungen einbezogen sind. Eine Unterscheidung deren Ressourcenbeiträge ist mit diesem Messverfahren nicht möglich. Insbesondere konnte während der vermessenen Operationen auch eine zusätzliche Prozessorlast gemessen werden, die in das Modell integriert wird. Es ist außerdem zu beachten, dass hier eine Vereinfachung getroffen wurde, die eine große Ungenauigkeit des Verhaltensmusters verursacht: Die modellierten Durchsätze sind Maximalwerte, die nur bei optimalen Bedingungen und Schreib-/Lese-Blockgrößen eintreten. Andererseits können sie auch um ein Vielfaches übertroffen werden, sobald das Betriebssystem eben geschriebene oder ausgelesende Daten in einem Cache speichert.

Modellierung

Listing 5.5 zeigt die ECL'-Verträge der beiden Festplatten. Der Schreib- und Lesedurchsatz wird durch eine weitere Charakteristik `throughput` qualifiziert. Die Definition des Verhaltensmodells (Listing 5.6) wird analog zu dem der Prozessoren gestaltet. Es werden in gleicher Weise Zeit- und Datenmengen-orientierte Muster modelliert. Die parallele CPU-Last fließt durch einen Aufruf des CPU-Verhaltensmusters in das Modell ein.

[3]Tools für Linuxsysteme: *sdparm* und *hdparm*

Listing 5.5 ECL'-Profile der Festplatten

```
1  characteristic throughput of [1/[T]]
2  profile HDDContract1 for HDD1{
3   state on{
4    provides throughput(Harddisk.read) = 65 [Mbyte/s]
5    provides throughput(Harddisk.write) = 27 [Mbyte/s]
6    consumes power 0.85 [W]
7   }
8   state off{consumes power 0.2 [W]}
9  }
10 profile HDDContract2 for HDD2{
11  state on{
12   provides throughput(Harddisk.read) = 26 [Mbyte/s]
13   provides throughput(Harddisk.write) = 19 [Mbyte/s]
14   consumes power 0.6 [W]
15  }
16  state off{consumes power 0.11 [W]}
17 }
```

Listing 5.6 Verhaltensmuster der Festplatten

```
1  def Harddisk.utilize in HDD1 = 1[]
2  def Harddisk.readtime in HDD1@on =
      (((Harddisk.amount/1[Mbyte])*37 [ms])/Harddisk.utilize)
3  when Harddisk.read in HDD1@on{
4   run Harddisk.readtime consumes (power (Harddisk.utilize*0.75[W]))
5   with(CPU.utilize=0.17[] CPU.runtime=Harddisk.readtime) call
       CPU.compute
6  }
7  def Harddisk.writetime in HDD1@on =
      (((Harddisk.amount/1[Mbyte])*15 [ms])/Harddisk.utilize)
8  when Harddisk.write in HDD1@on{
9   run Harddisk.writetime consumes (power
       (Harddisk.utilize*0.75[W]))
10  with(CPU.utilize=0.11[] CPU.runtime=Harddisk.writetime) call
       CPU.compute
11 }
12 def Harddisk.utilize in HDD2 = 1[]
13 def Harddisk.readtime in HDD2@on =
      (((Harddisk.amount/1[Mbyte])*53 [ms])/Harddisk.utilize)
14 when Harddisk.read in HDD2@on{
15  run Harddisk.readtime consumes (power (Harddisk.utilize*0.9[W]))
16  with(CPU.utilize=0.05[] CPU.runtime=Harddisk.readtime) call
       CPU.compute
17 }
18 def Harddisk.writetime in HDD2@on =
      (((Harddisk.amount/1[Mbyte])*38 [ms])/Harddisk.utilize)
19 when Harddisk.write in HDD2@on{
20  run Harddisk.writetime consumes (power (Harddisk.utilize*1.1[W]))
21  with(CPU.utilize=0.04[] CPU.runtime=Harddisk.writetime) call
       CPU.compute
22 }
```

Tabelle 5.3 Messungen – Netzwerkadapter

		«Notebook 1» MSI Wind U100	«Notebook 2» IBM Lenovo X41
Systemverbrauch 100% Netwerklast	[W]	12.6	14.5
Differenz zur Grundlast	[W]	1.6	1.7
Übertragung 1 Megabyte	[ms]	2128	1695

Listing 5.7 Virtuelle Komponenten – Netzwerkkarte

```
1 resource Network1 implements {Network}
2 resource Network2 implements {Network}
3 sysnode notebook1:9000 includes {Network1}
4 sysnode notebook2:9000 includes {Network2}
```

Listing 5.8 Verhaltensmuster der Netzwerkadapter

```
1  def Network.maxbandwidth in Network1 = 0.47[Mbyte/s]
2  def Network.bandwidth in Network1 = Network.maxbandwidth
3  def Network.utilize in Network1 =
       (Network.bandwidth/Network.maxbandwidth)
4  def Network.runtime in Network1 =
       (((Network.amount/1[Mbyte])*2128[ms])/Network.utilize)
5  when Network.stream in Network1 run Network.runtime
       consumes(power (1.6[W]*Network.utilize))
6
7  def Network.maxbandwidth in Network2 = 0.59[Mbyte/s]
8  def Network.bandwidth in Network2 = Network.maxbandwidth
9  def Network.utilize in Network2 =
       (Network.bandwidth/Network.maxbandwidth)
10 def Network.runtime in Network2 =
       (((Network.amount/1[Mbyte])*1695[ms])/Network.utilize)
11 when Network.stream in Network2 run Network.runtime
       consumes(power (1.7[W]*Network.utilize))
```

Tabelle 5.4 Softwareschnittstellen und Qualitäten

Schnittstelle	Operation	Eingabe	Ausgabe	Qualität	Dimension
Decoder	decode	MP3-URL	OGG-URL	accuracy	[]
Recognizer	fingerprint	OGG-URL	Fingerprint		
Recognizer	recognize	Fingerprint	Metadaten		
VideoFinder	findVideo	Metadaten	Video-URL	videowidth	[L]

5.4.3. Netzwerkadapter

Die Netzwerkübertragung hat ebenfalls deutlichen Einfluss auf die Dauer der Softwareoperationen und ist somit zu modellieren. Jedoch sind die Netzwerkkarten der Rechenknoten nicht steuerbar, da sie erstens keine Performancezustände besitzten und zweitens im deaktivierten Zustand die Kommunikation mit der Planungskomponente des Operators verhindern würden. Dennoch erzeugen sie unter Last eine zusätzliche Leistungsaufnahme, deren Messung wie im Fall der Prozessoren angelegt wird. Tabelle 5.3 zeigt die Messergebnisse. Die Übertragungsgeschwindigkeit wird im wesentlich höherem Maße von der Verbindungsqualität als vom Modelltyp beeinflusst, so dass letzterer hier keine Rolle spielt. Es wird zudem eine stabile Verbindung mit symmetrischer Übertragungsgeschwindigkeit vorausgesetzt.

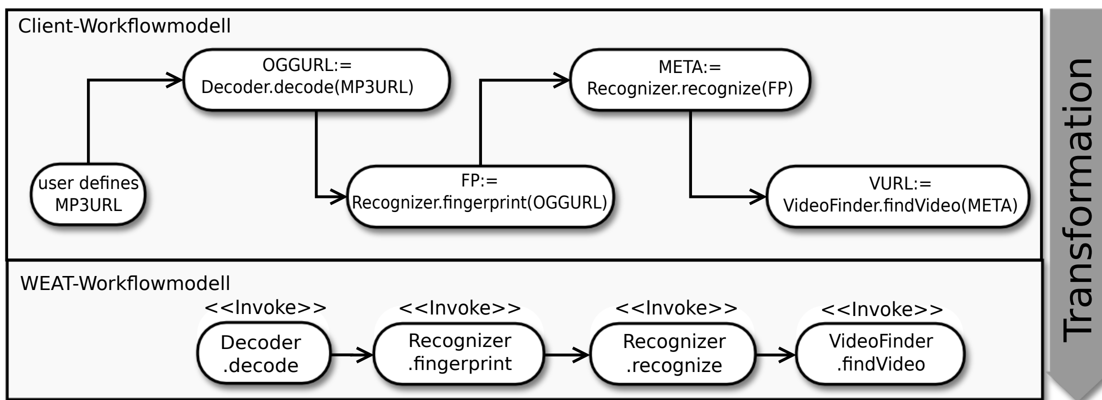
Modellierung

Für die Modellierung des Verhaltensmodells wird der gemessene Energieverbrauch ohne die oben bestimmte Ruheleistungsaufnahme beschrieben. Ein Vertragsmodell ist aufgrund der Nicht-Regulierbarkeit der Adapter nicht notwendig. Stattdessen muss ein zusätzliches virtuelles Gerät (Listing 5.7) beschrieben werden, da das verwaltete Komponentensystem auch keinen `ComponentMgr` ausführt, der die Netzwerkressource deklarieren könnte. Zudem kann der Modellierer die Funktion `bandwidth` durch eine gleichnamige Variable überschreiben und damit die Verbindungsqualität zu einen bestimmten Dienst beschreiben, die nicht höher sein kann als die Maximalbandbreite `maxbandwidth`. Die anteilige Last bestimmt in diesem Fall die Funktion `utilize` während der Simulation automatisch.

5.5. Schnittstellen und Workflowmodell

Das im Anwendungsfall vorgestellte Szenario benötigt drei Schnittstellen, die in Tabelle 5.4 gezeigt werden. Die Dienste interagieren in einer Filterkette und verarbeiten jeweils die Ausgaben des vorherigen Dienstes, die durch eine URL-Referenz oder durch direkt übergeben werden. Die `Decoder`-Schnittstelle deklariert die Operation `decode`, die eine MP3-URL entgegennimmt sowie eine URL zur erzeugten OGG-Datei zurückliefert, und eine zugehörige Qualität `accuracy` skalarer Dimension. Der Wert dieser Qualilät determiniert die Genauigkeit der dekodierten Ton-Daten, auf deren Grundlage die Musikerkennung vorgenommen werden soll. Je höher diese Qualität ist, desto größer wird die

Abbildung 5.3. Anwendungsfall – Workflowmodell



Erkennungswahrscheinlichkeit sein, die eine unmittelbare Benutzermetrik darstellt. Allerdings steigt bei zunehmender Qualität (siehe Implementierung der Softwarekomponente) auch die Rechenzeit und damit der Energieverbrauch des Dienstes. Die beiden Funktionen `fingerprint` und `recognize` zur Implementierung der Musikerkennung selbst werden durch keinerlei Charakteristiken qualifiziert. `fingerprint` verarbeitet die OGG-Datei zu einer Zeichenkette, die den Fingerprint kodiert, in `recognize` wird anschließend die Dabei mit dem ähnlichsten indizierten Fingerprint gesucht und die assoziierten Metadaten ausgegeben. Anschließend bietet die Funktion `findVideo` die Qualität `videowidth` der Längendimension. Diese entscheidet über die Auflösung des Videos[4] und führt je nach Größe zu einer unterschiedlichen Downloadzeit – diese korrespondiert ebenfalls mit einem Energieverbrauch. Die Ausgabe dieser Operation ist eine weitere URL zum heruntergeladenen Musikvideo.

Abbildung 5.3 zeigt den Workflow aus den Perspektiven des WfMS und des Planungssystems. Während im WfMS interaktive Zuweisungen (Benutzertasks) und parametrisierte Operationsaufrufe durchgeführt werden, sind im transformierten WEAT-Workflowmodell nur Informationen über die aufgerufenen Schnittstellenoperationen sowie des Kontrollflusses nach der Benutzerintaktion enthalten; dies ist der Zeitpunkt, an dem die Durchführung der Tasks bereits zugesichert werden kann.

5.6. Softwarekomponenten

Jede Schnittstelle des Anwendungsfalls wird durch eine Softwarekomponente implementiert, die auf jeweils einem Verarbeitungsknoten zur Verfügung steht, so dass in jedem Schritt des Workflows eine Auswahl im Planungsvorgang getroffen werden muss. Die Verhaltensdefinition der Softwarekomponenten beschreibt die Interaktion mit jeweils

[4] Das Seitenverhältnis wird als konstant vorrausgesetzt, so das die Breite gleichfalls die Höhe und damit die Gesamtauflösung determiniert.

anderen Softwarekomponenten (im Beispiel nicht enthalten) oder den bereits modellierten Hardwareressourcen. In letzterem Fall wird die Ausführungsgeschwindigkeit und damit die Dauer einer Softwareoperation in unterschiedlichen Zeitabschnitten meist durch eine einzelne Ressource bestimmt, insofern keine Algorithmen benutzt wurden, die explizite Wartebefehle verwenden. Die beschränkende Ressource verarbeitet damit eine definierte Last unter voller Auslastung während alle anderen gleichzeitig verwendeten Ressourcen unter Umständen Auslastungsgrade unterhalb ihrer Maximalperformance erreichen. Dies führt dazu, dass diese Ressourcen im Simulationsmodell von der beschränkenden Ressourcen getrieben werden müssen, was die Definition zusätzlicher Verhaltensmuster verlangt. Ansonsten ist bei der folgenden Modellierung der Software-Verträge zudem auf die korrekte Parametrisierung der Hardwaremuster durch Bestimmung der auf die gewählte Lasteinheit normierten und durch die Software erzeugen Lasten zu achten.

Decoder

Die Decoder-Implementierung wird auf Grundlage der Werkzeuge *lame*[5] und *sox*[6] durchgeführt. Die für die Schnittstellenoperation `decode` definierte Genauigkeits-Charakteristik wird im Vertrag in Listing 5.9 beschrieben. Beide Zustände fordern einen minimalen Schreib- und Lesedurchsatz der Festplatte von einem Megabyte, der dazu dient, Konfigurationen mit deaktivierter Festplatte im dem Rechenknoten auszuschließen, auf dem die Decoderkomponente aktiviert wird.

Listing 5.10 zeigt das Verhaltensmodell der Decoder-Komponente. Zunächst wird davon ausgegangen, dass in diesem abgeschlossenem Beispiel die Größe der MP3-Datei genau drei Megabyte beträgt. Durch die im Workflowmodell eingeführte Annotationsmöglichkeit könnte der Modellierer ein realitätsnäheres Modell erzeugen, was an dieser Stelle aber nicht umgesetzt werden soll. Im Anschluss erfolgt das Dekodieren der Datei, wie im Verhaltensmuster `decoderMP3` beschrieben. Dabei ist Last auf Netzwerkadapter, Festplatte und Prozessor zu simulieren; indes wird die Verarbeitungszeit durch die Übertragungsgeschwindigkeit des Netzwerkes beschränkt. Natürlich sollte bei einer detaillierteren Beschreibung eines Systems darauf geachtet werden, dass diese Festlegung durch unterschiedliche Konfigurationen nicht in jedem Fall eintreten muss; somit ist auch hier wieder eine Vereinfachung vorgenommen wurden. Es wird die oben definierte Funktion `Network.runtime` benutzt, um die gleichnamigen Funktionen der parallel betriebenen Ressourcen neu zu definieren. Die durch `utilize`-Parameter definierte Auslastung der Ressourcen wurde durch Messung bestimmt. Auch hier ist wieder davon auszugehen, dass die tatsächliche Auslastung in der Praxis von der letztlich verwendeten CPU determiniert wird.

Im zweiten Verarbeitungsschritt `encodeOGG` werden die Roh-Tondaten je nach gewählter Qualität `accuracy`, beziehungsweise je nach Zustand, mit unterschiedlicher Qualität in das freie Ton-Format OGG rekodiert. Die Datei-Größen sind Schätzwerte.

[5]http://lame.sourceforge.net/
[6]http://sox.sourceforge.net/

Listing 5.9 ECL'-Profil der Decoder-Komponente

```
1  characteristic accuracy of []
2
3  profile DecoderContract for DecoderImpl{
4   state slow{
5    provides accuracy(Decoder.decode) = 0.5 []
6    requires throughput(Harddisk.write) = 1 [Mbyte/s]
7   }
8   state fast{
9    provides accuracy(Decoder.decode) = 0.25 []
10   requires throughput(Harddisk.write) = 1 [Mbyte/s]
11   requires throughput(Harddisk.read) = 1 [Mbyte/s]
12  }
13 }
```

Listing 5.10 Verhaltensmuster der Decoder-Komponente

```
1  when Decoder.decodeMP3 in DecoderImpl {
2   call Network.stream
3   with(Harddisk.writetime=Network.runtime Harddisk.utilize=0.1[])
       call Harddisk.write
4   with(CPU.runtime = Network.runtime CPU.utilize=0.15[]) call
       CPU.compute
5  }
6  when Decoder.encodeOGG in DecoderImpl{
7   call Network.stream
8   with(Harddisk.readtime=Network.runtime
       Harddisk.utilize=0.1[])call Harddisk.read
9   with(CPU.runtime=Network.runtime CPU.utilize=0.3[]) call
       CPU.compute
10 }
11 when Decoder.decode in DecoderImpl@fast[
12  with(Network.amount = 3 [Mbyte]) {
13   call Network.stream
14   with(Harddisk.amount = Network.amount)call Harddisk.write
15  }
16  with(Network.amount = 70 [Mbyte]) call Decoder.decodeMP3
17  with(Network.amount = 1.25 [Mbyte]) call Decoder.encodeOGG
18 ]
19 when Decoder.decode in DecoderImpl@slow[
20  with(Network.amount = 3 [Mbyte]) {
21   call Network.stream
22   with(Harddisk.amount = Network.amount)call Harddisk.write
23  }
24  with(Network.amount = 70 [Mbyte]) call Decoder.decodeMP3
25  with(Network.amount = 2.5 [Mbyte]) call Decoder.encodeOGG
26 ]
```

Listing 5.11 Verhaltensmuster der Recognizer-Komponente

```
when Recognizer.fingerprint in RecognizerImpl[
  with(Network.amount = 2[Mbyte]) call Network.stream
 with(CPU.amount = 27.5[]) call CPU.compute
]

when Recognizer.recognizeSound in RecognizerImpl with(CPU.amount
   = 0.024[]) call CPU.compute
```

Listing 5.12 ECL'-Profil der VideoFinder-Komponente

```
characteristic videowidth of [L]

profile VideoContract for VideoFinderImpl{
    state good{
        provides videowidth(VideoFinder.findVideo) = 854 [pixel]
        requires throughput(Harddisk.write) = 1 [Mbyte/s]
    }

    state bad{
        provides videowidth(VideoFinder.findVideo) = 400 [pixel]
        requires throughput(Harddisk.write) = 1 [Mbyte/s]
    }
}
```

Recognizer

Der zentrale Komponente des Musikerkennungsdienstes wird durch die JAVA- und SOX-basierte Software JHears[7] zur Verfügung gestellt. Es wird keine ECL'-Vertragsdefinition benötigt. Die zugehörigen Verhaltensmuster dagegen zeigt Listing 5.11. Diese sind ebenfalls stark vereinfacht, unter anderem wird von einer «durchschnittlichen» Dateigröße von zwei Megabyte ausgegangen, die erst herunterzuladen ist und im Anschluss durch den Prozessor mit einer bereits gemessenen und auf auf die CPU-Lasteinheit normierten Last von 27.5 zu verarbeiten ist. Der ausgegebene Fingerprint wird der Funktion `recognizeSound` übergeben, die durch eine kurze, CPU-lastige Suche den ähnlichsten Hashwert und die mit diesem assoziierten Metadaten finden soll.

VideoFinder

Zum Suchen und Herunterladen des Musikvideos wird die Youtube-Plattform verwendet. Der größte Teil der dort verfügbaren Videos sind in den im Vertrags-Listing 5.12 definierten Auflösungen verfügbar, weshalb beispielhaft deren Existenz vorausgesetzt werden kann. Weiterhin wird wieder ein aktiver Festplattenzustand gefordert. Die Verhaltensdefinition (Listing 5.13) definiert die geschätzten Dateigrößen (die in diesem Fall tatsächlich in den meisten Fällen sehr ähnlich sind) und beschreibt anschließend die parallelen Übertragungs- und Schreibvorgänge.

[7] http://www.jhears.org

Listing 5.13 Verhaltensmuster der VideoFinder-Komponente

```
1 def VideoFinder.amount in VideoFinderImpl@good = 30 [Mbyte]
2 def VideoFinder.amount in VideoFinderImpl@bad = 15 [Mbyte]
3
4 when VideoFinder.findVideo in VideoFinderImpl@good with
     (amount=VideoFinder.amount Network.bandwidth=90[kbyte/s]){
5  call Network.stream
6  with(Harddisk.writetime=Network.runtime)call Harddisk.write
7 }
```

5.7. Berechnungs- und Messergebnisse

Es konnte bereits gezeigt werden, dass sich ein System zumindest grundlegend mit dem vorgestellten Ansatz beschreiben lässt. Aufgrund der Vereinfachungen an einigen Stellen des Simulationsmodells weichen vorhergesagte Zeit- und Energieverbräuche noch teils erheblich von den gemessenen (ebenfalls ungenauen) Werten ab. Bei häufiger Wiederholung wurde je nach Häufigkeit von nicht-betrachteten Seiteneffekten im vorgestellten Anwendungsfall ein Vorhersagefehler von bis zu etwa zwanzig Prozent gemessen. Die Genauigkeit kann durch Messung und Untersuchung des Verhaltens der Ressourcen und Komponenten und anschließender Anpassung der Modelle verbessert werden. Dazu sollte der Modellierer zuerst mit besonders abweichenden oder energieaufwändigen Systemteilen beginnen.

Es soll im Folgenden ein mögliches Berechnungsergebnis vorgestellt werden. Die Berechnungen sind nicht eindeutig (und damit auch nicht optimal), da die bisher benutzen Algorithmen zur Produktion möglicher Konfigurationspläne aufgrund ihrer hohen Rechenkomplexität zeitbeschränkt wurden. Für die parallele Ausführung des obigen Anwendungsfalls betrug die Benutzerwichtung der Charakteristiken

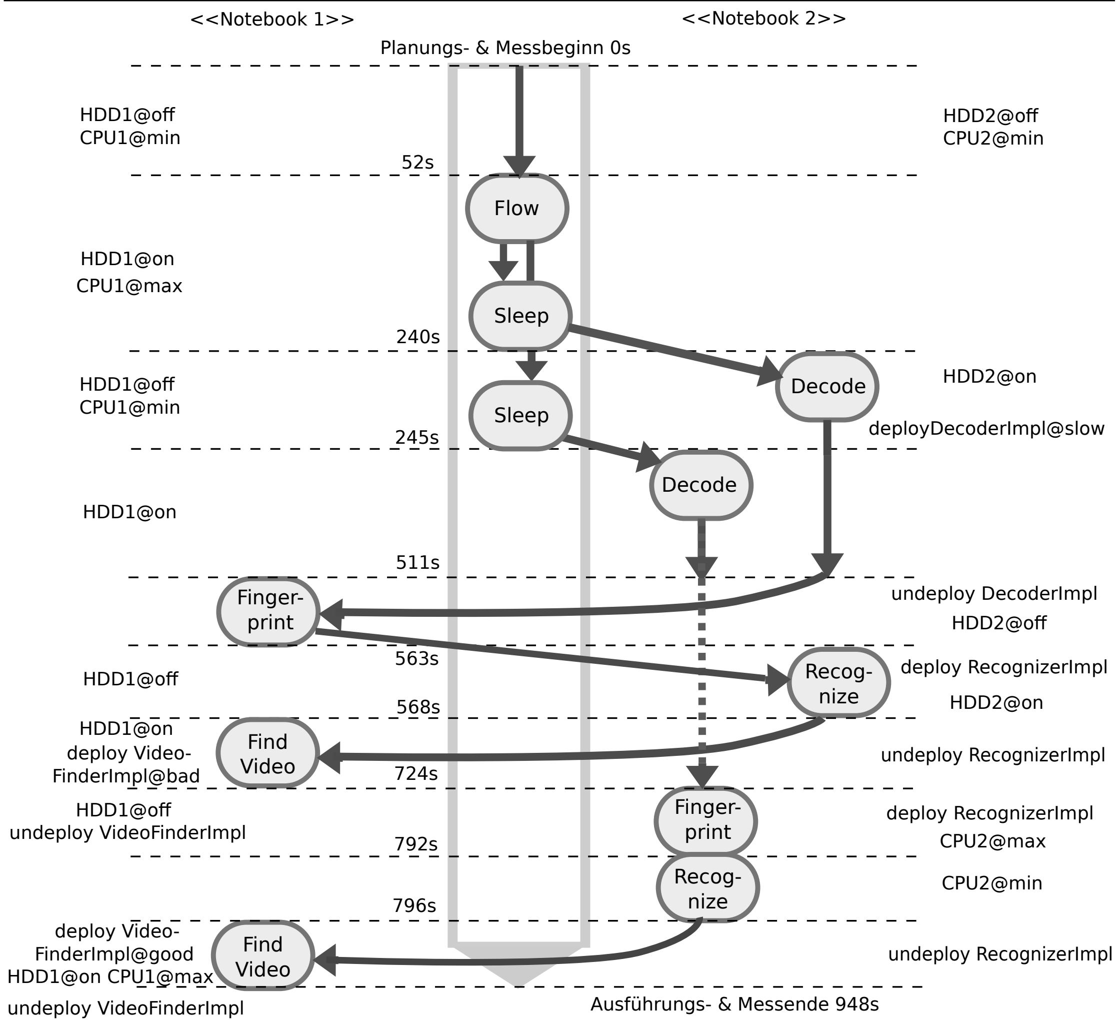
$$0.2 * \texttt{accuracy(Decoder.decode)} + 0.8 * \texttt{videowidth(VideoFinder.findVideo)}$$

.

Abbildung 5.4 zeigt sowohl den berechneten Plan, als auch die tatsächlich gemessenen Ausführungszeiten. Die Wartezeiten von 240 und 245 Sekunden der Workflows definieren die relativen Startzeitpunkte nach Beginn der Planung, weshalb auch die sekündliche Verbrauchsmessung am Zeitpunkt $t = 0$ beginnt. Da diese Messung ebenfalls durch WebServices gesteuert wird, tritt ein zusätzlicher Messfehler von bis zu einigen Sekunden auf. Nach 52 Sekunden ist die Planung abgeschlossen; das WfMS verharrt anschließend bis zum Start der Workflows. Die äußeren Spalten zeigen die jeweiligen Konfigurationsereignisse auf den beiden Rechenknoten. Die in einem Zeitintervall geplanten Rekonfigurationen werden gleichzeitig mit dem Deployment der für die enthaltenen Aufgaben notwendigen Komponenten durchgeführt. Bereits am Anfang wirkt sich die Beschränkung der Suchtiefen negativ aus – offensichtlich konnte keine alternative Variante für das anfängliche Warteintervall gefunden werden, in dem die Festplatten vollständig in den energiearmeren Zustand `sleep` versetzt worden wären. Dies ist auch der Grund für die wiederholten folgenden Zustandswechsel, die aber aufgrund der geringen Grundlast dieser Ressourcen nur zu einer ebenso geringen Messabweichung führen. Nach der

Wartezeit wird zunächst der Task `Decode` des ersten Workflows, kurz danach der des anderen parallel zum ersteren, ausgeführt. Die Parallelisierung der Tasks führt dazu dass der zuerst begonnene Task soweit behindert wird, dass die Dekodierungen gleichzeitig zum Ende kommen. Beide Decode-Tasks werden im energieaufwändigeren Zustand `slow` der Decoder-Komponente ausgeführt. Die restlichen Tasks wurden sequentiell geplant. Die Tasks des früher ausgeführten Workflows führt des System zuerst durch, so dass der andere Workflow bis zum Abschluss der ersten Videosuche warten muss (gekennzeichnet durch die unterbrochene Linie). Die Videos werden jeweils einmal in hoher und geringerer Qualität geladen. Nach Abschluss der letzten Aufgabe endet auch der Messzeitraum von $948s$ Sekunden. Im Verhältnis zu einer während der Planung vorausgesagten Gesamtdauer von 785 Sekunden wich dieser um 20.7% ab. Die Abweichung des gemessenen Energieverbrauchs von $21.19kJ$ zum geplanten Energieverbrauch von $21.7kJ$ betrug nur etwas mehr als 2%; dies sollte jedoch bei der recht hohen Zeitabweichung noch nicht als Nachweis besonderer Genauigkeit interpretiert werden. Im Rahmen der vom Hersteller des Batteriesensors angegebenen Messabweichung von bis zu 10% können die Modelle im nächsten Anpassungsschritt mit hoher Wahrscheinlichkeit die erforderliche Genauigkeit erreichen, die aber erst mit verbesserten Messgeräten validierbar wird.

Abbildung 5.4. Mögliches Berechnungsergebnis des Anwendungsfalls



6. Fazit und Ausblick

Im vorherigen Kapitel konnte gezeigt werden, dass es möglich ist, durch die eingeführten Modelle eine hinreichende Simulationsgenauigkeit zu erreichen, um den Energieverbrauch eines komplexen verteilten Systems vorherzusagen und Entscheidungen abzuleiten, die es ermöglichen, Ressourcen und Komponenten effizienzorientiert zu konfigurieren. Sowohl die statische als auch Last-verursachte Leistungsaufnahme floss in die Betrachtung des Systems über den simulierten Zeitraum ein. Die wichtigsten Voraussetzungen und deshalb Mittelpunkt des vorgestellten Ansatzes sind Vertrags- und Verhaltensmodell, die eine Verbrauch- und Qualitätsbewertung des Systems zulassen. Die Qualität kann vom Benutzer direkt oder einem Kontext-erzeugenden Klientsystem durch Wichtung gesteuert werden, je nach Einstellung hat somit eine simulierte Variante einen konkreten Effizienzwert, der den Vergleich mit anderen Varianten ermöglicht.

Damit ist das Planungssystem ein (teils vereinfachter) Prototyp des CoolSoftware-Ansatzes «Energy Auto Tuning» und zeigt die grundsätzliche Durchführbarkeit der dort herausgearbeiteten Vorgehensweise. Darüber hinaus wurde der Ansatz hinsichtlich der einer zeitlichen Betrachtung erweitert, so dass insbesondere Rekonfigurationsaufwände in den Vordergrund treten. Ein WfMS produziert dazu einen durchzuführenden, nicht-konditionalen Workflow, der die Beanspruchung des Systems über einen, durch die Simulation näher zu bestimmenden, Zeitraum beschreibt. Das System ermittelt anfangs eine Reihe von funktional möglichen Konfigurationsverläufen, indem der Workflow in kleinere Teil-Workflows (Partitionen) zerlegt wird, die jeweils eine Menge von Workflow-Tasks einschließen. Insofern ein Task ausschließlich sequentielle Nachbarn im Kontrollflussgraphen besitzt, kann er einzeln betrachtet werden, ansonsten ergeben sich eine Anzahl von Partitionen, die verschiedene parallele Ausführungen beschreiben. Während dies bereits die erste Ursache der großen Varianz der zu versuchenden Möglichkeiten ist, wird diese bei Suche nach validen Konfigurationen (Contract Negotiation) weiter vervielfacht. Diese Vielfalt ist eins der Grundprobleme die im Folgenden, teilweise unter dem Hinweis auf einen möglichen Lösungsansatz, noch einmal erläutert werden sollen.

Algorithmische Herausforderungen

Der große Varianzraum möglicher Konfigurationspläne kann durch die hier eingeführten Suchalgorithmen nur bedingt erschlossen werden. Auch eine Beschleunigung der Simulation auf Grundlage ausschließlich numerischer Algorithmen kann nur eine lineare Verbesserung herbeiführen, ebenso wie die Nutzung moderner Parallelisierung. Soweit keine Vorgehensweise gefunden wird, die eine direkte Ableitung eines Planes aus den Modellen ermöglicht, muss ein verbessertes Verfahren deshalb zukünftig eine heuristische Vorbewertung zwischen den einzelnen Transformationsschritten umsetzen. Die Heuristik kann beispielsweise auf vorherigen Planungsversuchen und beobachteter Effizienz beruhen. Womöglich können die einzelnen Verarbeitungsschritte, die schließlich sämtlich selbst Heuristiken erzeugen, auch soweit integriert werden, dass die Ineffizienz eines Suchpfades erheblich eher erkannt werden kann. Eine weitere Möglichkeit ist die Verwendung verteilten Rechnens – jeder Rechenknoten würde die Verarbeitung mindestens der ihm zugeordneten Ressourcen und Komponenten durchführen. Dadurch käme es zu einer jeweils gleichzeitigen Steigerung von Varianz und Rechengeschwindigkeit, wenn die

Kommunikationskosten begrenzt werden könnten.

Sobald die Algorithmen eine im Verhältnis zum verarbeiteten Workload effiziente Reife erreichen, sollten sie auch selbst in die Verbrauchs-Betrachtung einbezogen werden. Dabei können erneut Heuristiken zum Einsatz kommen; die Entscheidung ob ein Planungsvorgang unter Beachtung seines spezifischen Selbstverbrauchs sinnvoll ist, wird damit eine beinahe statische Betrachtung. Außerdem ist der berechnete Plan im entwickelten Ansatz teilweise bei einer erneuten Ausführung des Workflows wiederverwendbar, insofern keine zeitvarianten Seiteneffekte in die Berechnung einfließen. Negativ dagegen wirkt sich jegliche Form von Ungewissheit über zukünftige Ereignisse auf die Planung aus, sie führt zu unvorhersehbaren Änderungen im Kontrollfluss oder Aufwand – der Workflow muss deshalb nach jedem konditionalen Task neu verarbeitet werden. Möglicherweise kann dem durch einen Verzicht auf Detaillierungsgenauigkeit der Modelle entgegen gewirkt werden; dies liegt im Ermessen des Modellierers.

Herausforderungen an Modelle und Modellierer

Bis auf das Komponentenmodell mussten jegliche Informationen über Energieverbrauch und Qualität von Ressourcen und Komponenten manuell definiert werden. Die Cool-Software-Gruppe arbeitet jedoch bereits an Ansätzen, die zumindest die Energie- und Zeitverbrauchsinformationen durch Profiling automatisiert bestimmen. Ein anderer Ansatz könnte die Verwendung von statischer oder sogar dynamischer Kontrollflussanalyse sein, um das Verhaltensmodell einer Komponente zu erzeugen. Zudem stellt sich die grundsätzliche Frage, wer der Modellierer sein soll. Solang Ressourcenhersteller genaue Verbrauchsinformationen ihrer Produkte nicht veröffentlichen, werden sie auch keine Modelle dieser Art ausliefern. Komponentenentwickler dagegen müssen nur wenige Details über Interna preisgeben, um eine Vergleichbarkeit zu ermöglichen. Andererseits könnten die Betreiber energieeffizienter Plattformen, die ähnliche Ansätze wie den hier vorgestellten nutzen, auch in gemeinsamer Arbeit freie Messdaten über verbreitete Standardsoftware sammeln.

Zu diesem Zweck benötigen sie jedoch wesentlich genauere Messverfahren als die hier verwendeten. Nicht nur sollte es das Ziel sein, die Leistungsaufnahme jeder steuerbaren oder intensiv beanspruchten Ressource einzeln zu vermessen, sondern auch deren Rechenleistung vergleichbar zu machen. Dazu müssen verschiedene Arten von Workloads definiert werden, die einen Vergleich zu den Aufwänden der Komponenten erlauben. Die Aufwandsfunktionen des Verhaltensmodells sind im Übrigen wesentlich weniger rechenlastig als ein kleinteiliger Kontrollfluss, weshalb die Definition zeitvarianter Funktionen auch bei der Modellierung fokussiert werden kann.

Zurzeit sind außerdem nur wenige Seiteneffekte im Modell beschreibbar. Ein mögliches Vorgehen wäre die Unterstützung weiterer Verhaltensmustertypen für beispielsweise zeitgesteuerte oder bedingte Ereignisse; letztere würden von globalen, simulierbaren Datenänderungen ausgelöst. So könnten zeitperiodische Laständerungen oder Änderungen bei Wachstum zu verarbeitender Datenmenge in die Simulation einfließen. Solche Muster könnten ein zum Operations-Workload asynchrones Verhalten beschreiben und enthielten selbst Definitionen, wann ein Plan zu erneuern oder eine geplante Rekonfiguration

auszulösen wäre.

Herausforderungen an das Komponentensystem

Während der Verarbeitung des Workflows vergleicht das Planungssystem verschiedene Parallelisierungsverläufe von Tasks miteinander, von denen jeweils einer als optimal befunden wird. Dadurch findet aber eine zeitliche Festlegung und damit Änderung des Kontrollflusses statt. Die Ursache ist auch hier die Unsicherheit des tatsächlichen Verhaltens bei paralleler Ausführung. Um den effizienten Plan bei Ausführung des Workflows möglichst genau einzuhalten und damit die intendierte Energiemenge tatsächlich einzusparen, wird das WfMS durch Blockierung bis zum geplanten Zeitpunkt eines Tasks synchronisiert. Zur Vervollständigung dieses Ansatzes sollte aber, besonders wenn er auf andere Systeme als WfMS übertragen wird, die vollständige Ausführungskontrolle durch das Planungssystem selbst durchgeführt werden. Damit kommt es zu einer Kontrollumkehr, der Plan wird zu einem «Schedule» des geplanten Kontrollflusses. Durch die Festsetzung von Ausführungszeiten ist deshalb möglicherweise sogar eine Einbeziehung von Echtszeitsystemen vorzunehmen.

Herausforderungen an das Workflow Management System

Da bedingte Tasks und Zyklen von der Betrachtung ausgenommen wurden, verarbeitet das Planungssystem meist nicht den vollständigen Workflow, sondern nur einen begrenzten, voraussagbaren Teil-Workflow. Es ist die Aufgabe des WfMS zu entscheiden, wann eine Planung neu begonnen werden soll. Spätestens muss dies beim Verlassen des geplanten Zeitraumes geschehen, da die neuen funktionalen Abhängigkeiten des folgenden Kontrollflusses nur durch die Steuerung des Planungssystems zu erfüllen sind. Damit wird auch die Rolle der Dienstverzeichnisse (UDDI) WebService-orientierter WfMS erfüllt, eine Integration beider Komponenten (Verzeichnis und WEAT-Planung) wäre vorstellbar.

Das WfMS kann jedoch auch noch eine weitere Verbesserung bei der energieeffizienten Planung der Workflows vornehmen: Dazu kann der *gesamte* Workflow durch bestimmte Wahrscheinlichkeiten (beispielsweise auf Grundlage von Monitoring) vorhergesagt und bereits geplant werden. Die Neuplanung wäre in jedem Fall erforderlich, sobald ein vom Plan abweichendes Ereignis eintritt. Da die Ausführungspfade des Workflows meist durch Prädikate entschieden werden, kann das WfMS zudem am Zeitpunkt der Definition der für die Bedingungsprädikate notwendigen Variablen eine Vorhersage über den weiteren Verlauf treffen.

Durch die Möglichkeit der Steuerung des Verhaltensmodells mit Komplexitätsparametern (die durch Task-Annotation definiert werden), erübrigt sich zudem nicht die Frage nach der Quantifizierbarkeit von Operationsparametern. So kann bei weitem nicht für sämtliche komplexe Datenstrukturen entschieden werden, welchen Einfluss deren Werte auf die Ausführungszeit und den Energieaufwand der Operation ausüben. Damit bleiben einige Funktionen der Simulation unzugänglich.

Neben konditionalen Workflows, die in allen produktiv eingesetzten WfMS vorkommen, existieren auch Forschungsansätze, welche die Workflowstruktur erst zur Ausführungszeit

determinieren. Dazu zählt beispielsweise der OSPP-Ansatz der *reflexiven Workflows* [RBA08], bei dem Teil-Workflows in den Kontrollfluss integriert werden, um auf bestimmte, vom Regelfall abweichende, Situationen reagieren zu können. Auch dies sollte zu einer Neuplanung führen. Desweiteren sei erwähnt, dass WfMS-Ansätze, die selbst eine Dienst-Komposition während der Workflow-Ausführungszeit vornehmen (wie [RPHG10]) nicht mit dem vorgestellten Ansatz ohne weiteres kooperieren.

Herausforderungen an Hardwareressourcen und Komponenten

Neben der Präzisierung der Verbrauchsinformationen müssen die Hardwarehersteller zusätzliche aktive energiearme Zustände für Ressourcen einführen. Vor allem Geräte, die je nach Anwendungsfall nicht abgeschaltet werden können, würden mit solchen Konfigurationen viel Energie sparen. Zumeist beschränkt sich die jeweilige angebotene Qualität eines Ressourcenzustandes auf eine Performancecharakteristik. Ähnlich ist dies bei vielen Softwarekomponenten, die ebenfalls kaum unterschiedliche Zustände unterstützen. Dies sollte nachgeholt werden, so dass die Qualität der Komponenten durch Standardschnittstellen steuerbar wird.

Funktionelle und nicht-funktionelle Constraints

Durch die Verwendung der zentralisierten Kompositionssarchitektur des WfMS tritt zudem ein weiteres Problem in den Vordergrund: Parameter und Ergebniswerte von im verwalteten System durchgeführten Operationen sind entweder Werte oder Referenzen im System gespeicherter Daten. Werden größere Datenmengen verarbeitet, sind Referenzen sinnvoll, da sonst Zwischenergebnisse mehrmals zwischen den Komponenten und der WfMS - Benutzeroberfläche transportiert werden müssen. Oft spielen solche Zwischenergebnisse aber keine Rolle für den Benutzer sondern nur als Parameter anschließender Vorgänge. Doch besteht das Problem, dass bei der Übergabe einer Referenz an einen Dienst, die referenzierte Ressource verfügbar sein muss. Weil durch WEAT aber dynamische Kompositionen von Diensten erzeugt werden, kann dies nicht immer garantiert werden, da der Zugriffspunkt auf die Ressource im erzeugten Plan möglicherweise deaktiviert wird, um Energie zu sparen. Zur Lösung dieses Problems existieren zwei Möglichkeiten: Die einfachste Lösung ist die Einführung eines zentralen, persistenten Speichers, wie beispielsweise einer Datenbank. Dieser wird als zusätzliche funktionelle Abhängigkeit von Erzeuger und Verwender der Referenz modelliert. Andererseits ist auch die Verwendung von Abhängigkeitsprädikaten möglich. Dabei muss das Workflowmodell mit einer Möglichkeit ausgestattet werden, Beziehungen zwischen den komponierten Diensten herzustellen. So wäre es denkbar, beispielsweise eine Bedingung für die Gleichheit zweier gewählter Komponenten festzulegen. Diese Bedingungen könnten dann während der Contract Negotiation eingebracht werden, um den Suchraum zu beschränken.

Derartige Abhängigkeitsprädikate entsprechen funktionellen Bedingungen (Constraints). Zusätzlich sind auch nicht-funktionelle Bedingungen möglich, wie die Festlegung von Minima und Maxima einer Charakteristik. Durch die Definition eines Minimums für jede einzelne Benutzermetrik kann auch das Prinzip der Normierung durch das systemweite

Maximum einer Charakteristik abgelöst werden. Die Normierung findet dann anhand des definierten Minimums statt. Beide Verfahren benötigen jedoch Charakteristiken mit linear skalierenden Einheiten. Einheiten mit beispielsweise logarithmischen Skalen werden durch einfache Division durch einen fixen Betrag nicht korrekt normiert.

Fazit

Es sind noch eine Reihe von Fragestellungen zu beantworten, bevor ein produktives System energieeffizient adaptiert werden kann; einige dieser Fragestellungen wurden in diesem Kapitel vorgestellt. Die CoolSoftware-Gruppe wird ihre Forschung dahingehend fortsetzen, denn die Steigerung des Energierverbrauchs, wie er in der eingangs vorgestellten Studie für den vergangenden Zeitraum vorrausgesagt wurde, wird weiter fortschreiten, bis eine Kostenbarriere erreicht ist, die den Einsatz eines solchen Planungssystems zwingend erfordert.

Dabei muss keineswegs eine Berechnung für solche langen Zeiträume, die in dieser Arbeit betrachtet wurden, vorgenommen werden. Anstattdessen sind auch Lösungen für kurze Aufgaben möglich, insofern die Planungsalgorithmen effizient arbeiten. So lässt sich der Ansatz womöglich auch auf mobile Systeme übertragen; deren Batterielaufzeit würde damit noch einmal erweitert werden.

Die Planung und Steuerung von Kontrollflüssen in genau fixierten Zeiträumen durch diese Verfahren könnte eventuell auch die Verwendung von Echtzeitsystemen einschließen. Insofern aber der Detaillierungsgrad der Simulationsmodelle nicht eine (nicht näher bestimmte) Grenze überschreitet, werden deartige Systeme jedoch eher vermieden werden, da sie eine Einschränkung des Einsatzgebietes des Ansatzes bedingen.

Letztendlich wird mit dieser Art energieeffizienter Systeme ein neuer Schritt auf dem Weg zur «Green-IT» gegangen sobald er produktiv anwendbar wird. Wenn IT-Systeme geringer Last proportionale Ressourcenskalierung unterstützen, wie anfangs vorgeschlagen, lässt sich ein Großteil der verschwendeten Energie einparen, ebenso wie die mit ihr verbundenen Kosten.

A. Algorithmen zur Simulation

Listing A.1 Transformationsalgorithmus

```
1  FUNCTION transform
2  INPUT (V,E): Workflowgraph
3  OUTPUT WorkloadItem
4  VAR queue:Queue<Task> ← V in breadth-first-order
5  VAR seq:List<WorkloadItem> ← Nil
6  CONST nop: new Run(0[s],GlobalContext)
7  BEGIN
8   WHILE !queue.isEmpty DO BEGIN
9    VAR task ← queue.poll()
10   VAR next ← all successors of task in (V,E)
11   IF task is a Flow THEN BEGIN
12    (V',E') ← sub-workflow from task to corresponding Join
13    queue ← queue \ V'
14    VAR inFlow:Set<WorkloadItem> ← ∅
15    FOR n ∈ next DO BEGIN
16     (V'',E'') ← sub-workflow from n to end of (V',E')
17     inFlow ← inFlow ∪ {transform(V'',E'')}
18    END
19    seq.append(new Parallel(inFlow,GlobalContext))
20   END
21   ELSE IF task is a Sleep THEN seq ← seq :: new
        Run(task.time,GlobalContext)
22   ELSE IF task is a Invoke THEN seq ← seq :: new
        With(task.annotation,new Call(task.interface,
        task.operation,GlobalContext),GlobalContext)
23   ELSE seq ← seq :: nop
24  END
25  RETURN new Sequence(seq::nop,GlobalContext)
26 END
```

Der Algorithmus A.1 transformiert ein vom WfMS erzeugtes Workflowmodell in ein korresponierendes Verhaltensmodell, um es in gleicher Weise wie die modellierten Verhaltensmuster simulieren zu können. Die Eingabe ist der Workflowgraph, dessen Knoten (Tasks) V anfangs, in Zeile 4, in Ordnung einer Breitensuche in eine Warteschlange eingereiht werden. Danach werden die Tasks in dieser Reihenfolge verarbeitet, bis die Warteschlange leer ist. Die Verarbeitung eines Tasks ist abhängig von dessen Typ. Für einen `Flow`, Zeile 11 ff., wird zunächst der vollständige Teil-Workflow bestimmt, der bis zum Ende (`Join`) der Parallelisierung reicht. Die Tasks dieses Teil-Workflows werden auch «vorzeitig» aus der Warteschlange entfernt und sofort in einer Rekursion weiterverarbeitet. Die Rekursion Zeile 17 generiert aus jedem «Thread» der Parallelisierung einen eigenständigen Teil-Workflow und transformiert diesen anschließend seperat. Die erzeugten Ergebnisse werden dann in Zeile 19 einem `Parallel`-Objekt untergeordnet. Ein `Sleep` erzeugt in Zeile 21 ein `Run`-Objekt mit der vorgegebenen Laufzeit; aus dem `Invoke`, Zeile 22, ist die Annotation zu extrahieren und in einen `With`-Kontext mit aggegriertem `Call` umzuwandeln. Andere Tasks, die nur der Strukturerhaltung dienen, transformiert der Algorithmus in eine «NOP» (No OPeration), die aus einem zeitlosen `Run` besteht. Sämtliche erzeugten Objekte werden einer `Sequence` untergeordnet (Zeile 25), der am Ende eine weitere NOP hinzugefügt wird, die den Planungsalgorithmus anweist, in diesem (Null-Sekunden-)Zeitraum eine Rekonfiguration auf die minimale Systemvariante vorzunehmen.

Listing A.2 Schedulingalgorithmus

```
1  FUNCTION schedule
2  INPUT item: WorkloadItem
3  INPUT comp: Component
4  INPUT node: SystemNode
5  INPUT queue: Queue<(WorkloadItem,Amount<Duration>)>
6  INPUT waitFor: I → W, WorkloadItem i ↦ Set<WorkloadItem>
7  VAR wait: Set<WorkloadItem> ← ∅
8  BEGIN
9   IF item is a Call THEN BEGIN
10   VAR behavior ← operation behaviors corresponding to comp ∧
        wi.interface ∧ wi.operation
11   VAR cqueue ← corresponding queue of component associated with
        the found behavior
12   VAR ic ← behavior.creates.inContext(item.ctx)
13   cqueue.enqueue((ic,0[s]))
14   wait.append(ic)
15  END ELSE IF item is a If THEN BEGIN
16   IF item.condition.eval(item.ctx) THEN BEGIN
17    VAR ic ← item.body.inContext(item.cxt)
18    queue.enqueue((ic,0))
19    wait.append(ic)
20   END
21  END ELSE IF item is a With THEN BEGIN
22   VAR ic ← item.body.inContext(item.ctx).inContext(item)
23   queue.enqueue((ic,0))
24   wait.append(ic)
25  END ELSE IF item is a Parallel THEN BEGIN
26   FOR i ∈ item.items DO BEGIN
27    VAR ic ← i.inContext(item.cxt)
28    queue.enqueue((ic,0))
29    wait.append(ic)
30   END
31  END ELSE IF item is a Sequence THEN BEGIN
32   VAR ic ← item.first.inContext(wi.ctx)
33   VAR newseq ← item \ ic
34   queue.enqueue((ic,0))
35   queue.enqueue((newseq))
36   wait.append(ic)
37   wait.apend(newseq)
38  END
39  RETURN wait
40 END
```

Der Algorithmus A.2 ist bereits Teil der eigentlichen Simulation. Er traversiert den modellierten Kontrollfluss und generiert Warteschlangeneinträge sowie einen Abhängigkeitsgraphen, der Wartebeziehungen zwischen den erzeugten Items definiert und so die Warteschlangen synchronisiert. Eingabe ist ein Item und die Komponente sowie der Knoten, auf dem das Item durchzuführen ist. Außerdem werden auch die Warteschlange der Komponente und der bisherige Abhängigkeitsgraph `waitFor` übergeben. Dann wird jeder Item-Typ unterschiedlich behandelt: In den Zeilen 12 ff. muss beispielsweise nach dem zu einem `Call` korrespondierenden Verhaltensmuster gesucht werden. Das gefundene Item im Behavior-Feld `creates` wird dann durch Übergabe des Kontextes des aufrufenden Items von der Modellebene auf die Simulationslaufzeitebene instanziiert. Anschließend wird das erzeugte Objekt in die Warteschlange des gefundenen Implementators der `Call`-Schnittstelle eingereiht und zudem in der auszugebenden Menge `wait` gepeichert. In Zeile 15 werden Objekte der `If`-Klasse umgewandelt. In der darauf folgenden Zeile wird zunächst die Bedingung der Anweisung über dem zugeordneten Kontext ausgewertet und anschließend wieder die Umwandlung ähnlich zum `Call` umgesetzt. Ein `Parallel` (Zeile 31) führt die Umwandlung für sämtliche untergeordneten Objekte durch, die `Sequence` nur für das in der Sequenz erste Objekt. Die restlichen Objekte sind in einer neuen Sequenz unterzubringen und im nächsten Verarbeitungsschritt umzuwandeln. Die Rückgabe enthält die erzeugten Objekte.

Listing A.3 Simulationsalgorithmus

```
FUNCTION simulate
INPUT load: L → N, WorkloadItem l ↦ SystemNode n
INPUT variant:  Set<Usage>
OUTPUT (Amount<Duration>,Amount<Energy>)
VAR vtime: Amount<Duration> ← 0
VAR energy: Amount<Energy> ← 0
VAR done: Set<WorkloadItem> ← ∅
VAR doing: Set<(Component,SystemNode,Run, Amount<Duration>)> ← ∅
VAR waitFor: I → W, WorkloadItem i ↦ Set<WorkloadItem> ← ∅
VAR queues: C → Q, (Component,SystemNode) c ↦ Queue<WorkloadItem>
   q ← ∅
CONST root: Component ← new Component()
BEGIN
 FOR (l,n) ∈ load DO
  queues((root,n)) ← queues((root,n)) ∪ {(l,0)}
 END
 WHILE ∃ ((c,q)): (c,q) ∈ queues ∧ !q.isEmpty DO BEGIN
  doing ← doing \ {(c,n,r,a):(c,n,r,a) ∈ doing ∧ a < vtime}
  VAR found ← true
  WHILE found DO BEGIN
   found ← false
   FOR ((c,n),q) ∈ queues DO BEGIN
    VAR doNow ← {(w,t):(w,t) ∈ q.takeFirst(threads(c)) ∧ w is a
       Run ∧ t < vtime ∧ waitFor(w).isEmpty}
    q ← q \ doNow
    FOR ((w,t)) ∈ doNow DO BEGIN
     done ← done ∪ {w}
     IF w is a Run THEN BEGIN
      doing ← doing ∪  {(c,n) ↦ (w,w.delay.eval(w.ctx)+vtime)}
      energy += w.energy.eval(w.ctx)
     END
     ELSE BEGIN
      VAR wait ← schedule(w,c,n,q,waitFor)
      replace all occurences of w in queues by wait
     END
     found ← true
    END
   END
  END
  vtime += 1[s]
 END
 RETURN (vtime,energy)
END
```

Der Algorithmus A.3 zeigt letztendlich den Simulationsvorgang. Er wird mit einer Variante `variant` parametrisiert, welche die Systemkonfiguration im simulierten Zeitraum repräsentiert, sowie dem durchzuführenden Workload `load`, der hier als Abbildung von Items auf ein Komponente-Systemknoten-Tupel repräsentiert wird. Ziel ist die Bestimmung der verbrauchten Zeit `vtime` und Energie `energy`. Die Variablen `done` und `doing` speichern jeweils bereits abgeschlossene oder zur gegenwärten Simulationszeit aktive Items, letztere beinhaltet zudem den Ort und den Endzeitpunkt der Ausführung. `waitFor` ist ein Abhängigkeitsgraph zwischen den Items und dient zur Sicherstellung der sequentiellen Ausführung über mehrere Wartschlangen, die wiederum in der Variable `queues` zur ihren jeweiligen Komponenten oder Ressourcen zugeordnet werden. Anfangs, in Zeile 14, ordnet der Algorithmus den parametrisierten Workload in die Warteschlange einer Pseudokomponente «root» ein. Anschließend beginnt die Simulation. Im jedem Simulationsschritt ist zu überprüfen, ob noch Warteschlangen existieren, die nicht völlig entleert sind (Zeile 16). In der folgenden Zeile werden der Mengen-Variable `doing` sämtliche Items entnommen, deren eingeplante Zeit abgelaufen ist. Ab Zeile 19 folgt eine weitere Schleife, die nach sämtlichen Items sucht, die zum gegenwärtigen Simulationszeitpunkt aktiviert werden können (Zeile 22). Diese aggregiert die Variable `doNow`. Insofern es sich bei einem dieser Items im ein `Run` handelt, wird es in Zeile 27 in die Menge aktivierter Tasks `doing` eingereiht und die spezifische Energie auf die bisher ermittelte Gesamtenergie aufgeschlagen. Zur Berechnung dieser Energie und der Abschlusszeit des Items müssen die mathematischen Berechnungsvorschriften des `Run`-Objektes an diesem Punkt über dem zugeordneten Kontext ausgewertet werden. Sämtliche andere Item-Typen werden in Zeile 31 durch die Scheduling-Algorithmus transformiert. Die gefundenen Objekte dieses Aufrufs substituieren das Urprungsobjekt in dessen jeweiliger Warteschlange.

B. Benutzeroberfläche

Abbildung B.1. Benutzeroberfläche Teil I

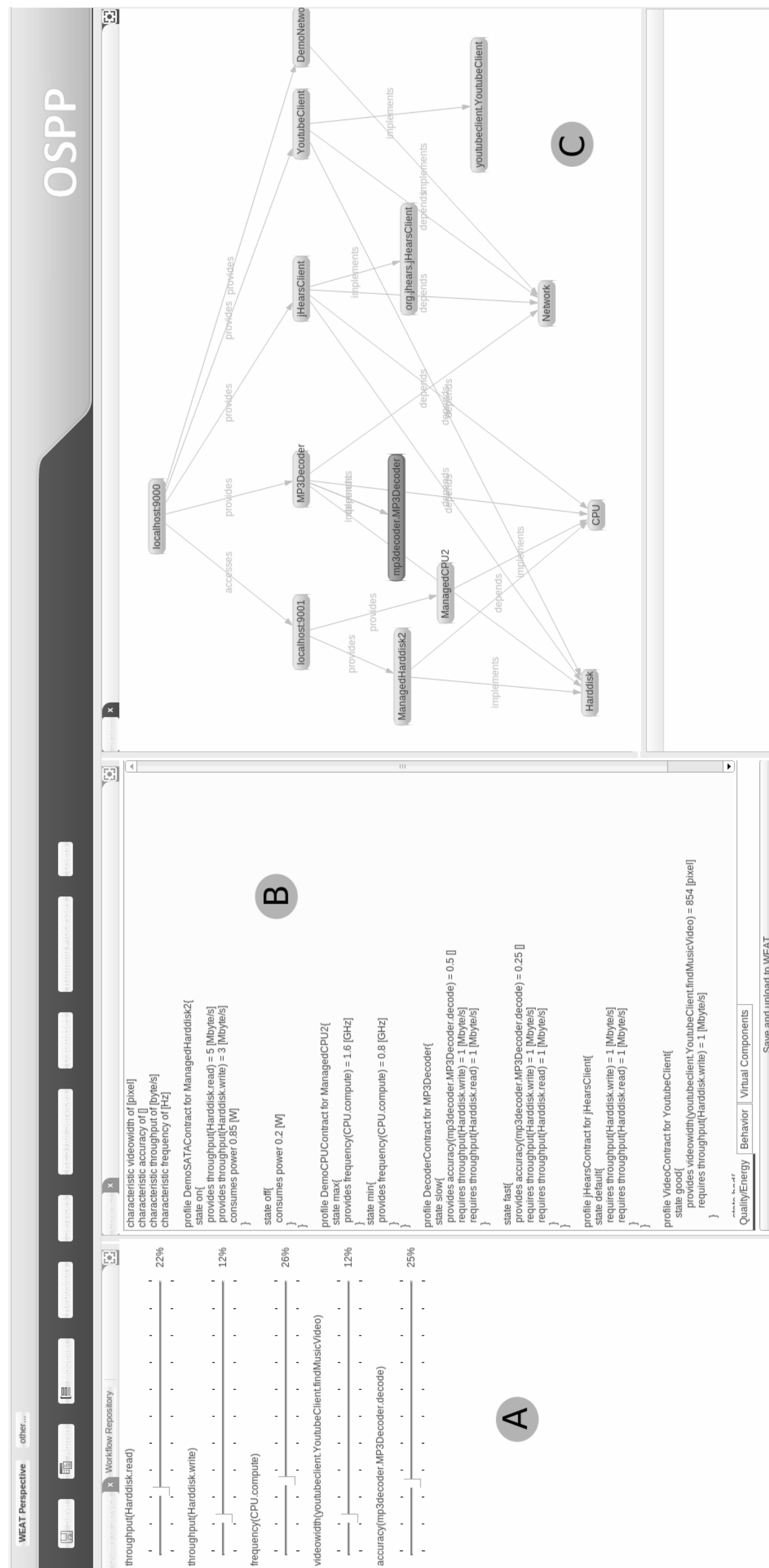


Abbildung B.2. Benutzeroberfläche Teil II

Die Abbildungen B.1 und B.2 zeigen die RAP-Oberfläche und deren Integration in OSPP. In Ansicht A können die Wichtungen der Qualitätscharakteristiken durch Schieberegler eingestellt werden. Die Regler reagieren jeweils, wenn ein einzelner verändert wird, so dass die Summe aller Gewichte immer Eins beträgt. Ansicht B dient zur textuellen Modellierung des Vertrags- und Verhaltensmodells sowie der virtuellen Ressourcen und Komponenten. C zeigt eine Übersicht des Systems mit Systemknoten, den jeweils verfügbaren Komponenten und von diesen angebotene oder benötige Schnittstellen. Ansicht D führt alle modellierten Workflows auf. Durch die Bedienelemente im unteren Bereich kann das WEAT-System deaktiviert und aktiviert werden oder der Workflow wird geplant beziehungsweise direkt gestartet. In Ansicht E wird der Konfigurationsplan dargestellt. Ein Graph repräsentiert dabei die Taskmengen anhand der Task-IDs. Wird einer der Knoten gewählt, zeigt die Ansicht weiter rechts die für diese Taskmenge gültige Konfiguration (Variante) durch einen weiteren Graph. Darunter, in F, sind die geplanten Zeit- und Energieverbräuche ablesbar. Letzendlich kann für den erzeugten Plan in Ansicht G ein Balkendiagramm angezeigt werden, in dem die jeweilige prozentuale Erfüllung der Qualitätscharakterisiken abgetragen ist.

Literaturverzeichnis

[2.108] Workflow Mangement Coalition Version 2.1a. XML Process Definition Language, Okt 2008.

[Aag01] Jan Øyvind Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, 2001.

[BH07] Luiz André Barroso and Urs Hölzle. The Case for Energy-Proportional Computing. *Computer*, 40(12):33–37, 2007.

[BR10] David J. Brown and Charles Reams. Toward Energy-efficient Computing. *Commun. ACM*, 53:50–58, March 2010.

[Coa95] Workflow Management Coalition. The Workflow Reference Model, Jan 1995.

[DDF+06] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1:223–259, December 2006.

[EPA07] EPA. EPA Report to Congress on Server and Data Center Energy Efficiency. Technical report, U.S. Environmental Protection Agency, 2007.

[FPS02] Jason Flinn, SoYoung Park, and M. Satyanarayanan. Balancing Performance, Energy, and Quality in Pervasive Computing. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, ICDCS '02, pages 217–, Washington, DC, USA, 2002. IEEE Computer Society.

[FS04] Jason Flinn and M. Satyanarayanan. Managing Battery Lifetime with Energy-Aware Adaptation. *ACM Transactions on Computer Systems (TOCS*, 22:137–179, 2004.

[FZJ08] Yunsi Fei, Lin Zhong, and Niraj K. Jha. An energy-aware framework for dynamic software management in mobile computing systems. *ACM Trans. Embed. Comput. Syst.*, 7:27:1–27:31, May 2008.

[Gar07] Matthew Garrett. Powering Down. *Queue*, 5:16–21, November 2007.

[GWS+10a] Sebastian Götz, Claas Wilke, Matthias Schmidt, Sebastian Cech, and Uwe Aßmann. Towards Energy Auto Tuning. In *Proceedings of First Annual International Conference on Green Information Technology (GREEN IT)*, 2010.

[GWS+10b] Sebastian Götz, Claas Wilke, Matthias Schmidt, Sebastian Cech, Johannes Waltsgott, and Ronny Fritzsche. THEATRE Resource Manager Interface Specification v. 1.0. Technical Report TUD-FI10-08, ISSN 1430-211X, Technische Universität Dresden, December 2010.

[HRR+08] Dirk Habich, Sebastian Richly, Andreas Ruempel, Wolfgang Buecke, and Steffen Preissler. Open Service Process Platform 2.0. In *Proceedings of the 2008 IEEE Congress on Services - Part I*, SERVICES '08, pages 152–159, Washington, DC, USA, 2008. IEEE Computer Society.

[ISO] ISO/IEC. ISO/IEC 14977:1996(E) (EBNF). http://www.cl.cam.ac.uk/ mgk25/iso-14977.pdf.

[MB01] Ruth Malan and Dana Bredemeyer. Defining Non-functional Requirements. Bredemeyer Consulting, White Paper. http://www.bredemeyer.com/papers.htm, 2001.

[MK95] Mike Mannion and Barry Keepence. SMART Requirements. *SIGSOFT Softw. Eng. Notes*, 20:42–47, April 1995.

[MS07] Mesfin Mulugeta and Alexander Schill. An approach for QoS contract negotiation in distributed component-based software. In *Proceedings of the 10th International Conference on Component-based Software Engineering*, CBSE'07, pages 90–106, Berlin, Heidelberg, 2007. Springer-Verlag.

[OAS07] OASIS. Web Services Business Process Execution Language Version 2.0, Apr 2007.

[Obj09] Object Modeling Group. Business Process Modeling Notation. `http://www.omg.org/spec/BPMN/1.2/`, January 2009.

[ONR+06] Keith J. O'hara, Ripal Nathuji, Himanshu Raj, Karsten Schwan, and Tucker Balch. Autopower: Toward energy-aware software systems for distributed mobile robots. In *In IEEE International Conference on Robotics and Automation (ICRA*, 2006.

[Pro] Java Community Process. JSR 275: Units Specification. http://www.jcp.org/en/jsr/proposalDetails?id=275.

[RBA08] Sebastian Richly, Wolfgang Buecke, and Uwe Assmann. A BDI-based Reflective Infrastructure for Dynamic Workflows. In *Proceedings of the 2008 12th Enterprise Distributed Object Computing Conference Workshops*, pages 112–119, Washington, DC, USA, 2008. IEEE Computer Society.

[RGmS10] Sebastian Richly, Sebastian Götz, Uwe Aßmann, and Sandro Schmidt. Role-based Multi-Purpose Workflow Engine Architecture. In *Proceedings of 5th International Workshop on Technologies for Context-Aware Business Process Management (TCoB 2010)*, 2010.

[RPHG10] Sebastian Richly, Georg Püschel, Dirk Habich, and Sebastian Götz. MapReduce for Scalable Neural Nets Training. In *IEEE Proceedings of Congress on Services (SERVICES 2010-I)*, 2010.

[RZ03] Simone Röttger and Steffen Zschaler. CQML+: Enhancements to CQML. In Jean-Michel Bruel, editor, *1st Intl. Workshop on Quality of Service in Component-Based Software Engineering*, pages 43–56, Toulous, France, June 2003.

[SEMM08] Chiyoung Seo, George Edwards, Sam Malek, and Nenad Medvidovic. A Framework for Estimating the Impact of a Distributed Software System's Architectural Style on itsEnergy Consumption. In *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, WICSA '08, pages 277–280, Washington, DC, USA, 2008. IEEE Computer Society.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[vBR10] J. vom Brocke and M Rosemann. *Handbook on Business Process Management: Strategic Alignment, Governance, People and Culture.* Berlin: Springer, 2010.

[VMw08] VMware. How VMware Virtualization Right-sizes IT Infrastructure to Reduce Power Consumption. Whitepaper, http://www.vmware.com/de/solutions/consolidation/green/, 2008.

[W3C00] W3C. Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/, May 2000.

[W3C01] W3C. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/2001/NOTE-wsdl-20010315, Mar 2001.

[Wes00] Douglas B. West. *Introduction to Graph Theory (2nd Edition).* Prentice Hall, August 2000.