Projektgruppe 459

Software Performance Engineering
- Entwicklung eines Modellkonverters
zur quantitativen Bewertung von
Software(modellen)

Endbericht
SoSe 2005

Ulhak Arslan, Ugur Aydin, Dacheng Chen, Zoi Choselidou,
Martin Ebers, Sadik Hamurcu, Olaf Hengesbach, Delice Jussen,
Deniz Kayar, Taher Nasched, Gulnara Ortlieb, Alexei Sacharow


UML
Hierarchical models
In
Performance
Engineering
hipe
HIT
© 2005 PG459 Universität Dortmund

Inhaltsverzeichnis

0 Einführung

Dieser Projektgruppenbericht fasst die im Wintersemester 04/05 und Sommersemester 05 am Lehrstuhl Informatik IV der Universität Dortmund von der Projektgruppe 459 (PG459) erarbeiteten Ergebnisse und Konzepte zusammen. Die Projektgruppe ist eine Spezialveranstaltung des Informatikstudiums der Universität Dortmund und ist zeitlich in zwei Semestern des Hauptstudiums angesiedelt. Sie findet ihre Begründung in der Tatsache, dass ein Informatikstudent im Laufe seines Studiums viel theoretisches Wissen erlangt, dieses aber selten praktisch umsetzt.

Der Ablauf der Projektgruppe ist so, dass der Studentengruppe, die in der Regel aus acht bis zwölf Leuten besteht, zunächst ein Problem vorgestellt wird. Anschließend bekommen sie die Gelegenheit, sich in einer Seminarphase in für die Problemlösung relevante Themengebiete einzuarbeiten. Anhand dieses Wissens muss dann selbstständig eine strukturierte Lösung erarbeitet und praktisch umgesetzt werden. Hierbei sollte erwähnt werden, dass es sich bei den erstellten Lösungen, wie auch im Falle dieser Projektgruppe, meist um Softwarelösungen handelt. Deren Umsetzung mündet also in eine Implementierung. So werden alle Phasen eines Softwareentwicklungsprozesses durchgespielt. Besonderes Augenmerk liegt auch auf dem Organisationstalent und der Teamfähigkeit der einzelnen Projektgruppenmitglieder, denn erst diese ermöglichen einen reibungslosen Ablauf des Projekts und liefern zufriedenstellende Endergebnisse.

Das Thema der PG459 ist in den Bereich des „Software Performance Engineering“ (SPE) einzuordnen und behandelt die Entwicklung eines Modellkonverters zur quantitativen Bewertung von Software- bzw. UML 2.0-Modellen. Genauer gesagt geht es darum, eine Software zu erstellen, die UML-Modelle in eine Eingabe für das Leistungsbewertungswerkzeug HIT (vgl. [Weißenberg u. a. (1999)]), welches am Lehrstuhl Informatik IV entwickelt wurde, übersetzt und darauf eine Leistungsbewertung durchführt. Der Name der Software ist „HIPE“ (Hierarchical models In Performance Engineering). UML (siehe [OMG (2005)]) bezeichnet hierbei die weitverbreitete „Unified Modelling Language“, welche der Modellierung der Struktur und des Verhaltens objektorientierter Software dient.

Die Motivation des Themas ist das Schaffen der Möglichkeit, Effizienzkriterien von Software in einem frühen Entwicklungsstadium zu bestimmen und so Mehraufwand und Kosten zu vermeiden, die durch die nachträgliche Beseitigung von Fehlern in der finalen Software auf einen Entwickler zukommen können.

In der vorherigen Projektgruppe PG438 (vgl. [PG438 (2004)]) wurde genau dieses Thema schon behandelt, mit dem Unterschied, dass anstelle von UML 2.0-Modellen UML 1.x-Modelle in eine Eingabe für HIT transformiert wurden. Eine der Hauptaufgaben dieser PG ist es also, herauszufinden, ob UML 2.0 mit seinen zahlreichen Änderungen und Erweiterungen die Beantwortung von SPE-Fragestellungen erleichtert und ob sich diese UML-Modelle nun besser

in Leistungsbewertungsmodelle umwandeln lassen.

Im ersten Kapitel wird die Realisierung eines Software Performance Engineering mit HIPE dargestellt. Einführend werden zunächst die grundlegenden Ideen des Software Performance Engineering beschrieben. Danach werden die konzeptionellen Ideen zur Bewertung von UML-Diagrammen und die konkrete Umsetzung der Bewertung dargestellt. Schließlich werden die Einschränkungen bzw. möglichen Erweiterungen dieses Konzeptes beschrieben. Dieses Kapitel wird im Folgenden als „HIPE-Paradigma“ bezeichnet. Hierbei ist zu beachten, dass das Wort Paradigma nicht im eigentlichen Sinne verwendet wird, sondern dass es das Software Engineering mit HIPE bezeichnet. Die Softwaremodelle, also u.a. die Klassendiagramme, zu diesen Ideen finden sich im darauffolgenden Kapitel „Entwurf“. Die Erläuterung, Konzepte und Methoden der implementierten Klassen befinden sich im Kapitel „Implementierung“. Die Ergebnisse einer Testphase sind im vierten Kapitel „Produkttest“ niedergeschrieben. Hier wird untersucht, ob die entwickelte Software allen möglichen Eingabeszenarien standhält und immer plausible Ergebnisse hervorbringt. Das sich hieran anschließende Kapitel „Benutzungshandbuch“ erläutert die Bedienung von HIPE. Im sechsten Kapitel „Fazit“ ziehen alle PG-Mitglieder und der PG-Betreuer ein kurzes Resümee, indem sie die während der Projektgruppe gesammelten Eindrücke und Erfahrungen in einem kurzen Text zum Ausdruck bringen. Das letzte Kapitel „Ausblick“ behandelt die Erkenntnisse des Themas und z.B. mögliche zukünftige Anknüpfungspunkte für andere Forschungsgruppen.

1 HIPE-Paradigma

Dieses Kapitel beschreibt den Kern von HIPE, nämlich das Paradigma, das zur quantitativen Analyse von UML-Diagrammen entworfen und angewandt wurde. Zuvor werden die zum Verständnis der entsprechenden Abschnitte benötigten Grundlagen und Begriffe erklärt. Dies umfasst sowohl eine kurze Einführung in das prinzipielle Vorgehen des Software Performance Engineering als auch eine grobe Beschreibung der Anforderungen, die eine erfolgreiche Anwendung von HIPE an den Benutzer stellt. Abschließend wird beschrieben, wie HIPE die ermittelten Ergebnisse an den Benutzer zurückgibt. Dieser Teil trägt die Bezeichnung "Bewertungsmodul".

1.1 Software Performance Engineering - Ideen und Ansätze

Die Entwicklung eines Software- oder Hardwaresystems beinhaltet stets implizit die Forderung nach möglichst optimaler Performance. Meist gehören explizite Forderungen an die Dauer der Berechnungen bzw. an die Antwortzeiten zur Aufgabenstellung. Dabei wird Performance allgemein definiert als Grad der Erfüllung der an das System gestellten Anforderungen unter Ausführung der eigentlichen Funktion. Das heißt, ein System mit guter Performance erledigt seine Aufgabe und respektiert außerdem die gegebenen Anforderungen wie zum Beispiel Speichernutzung, Rechengenauigkeit und/oder -geschwindigkeit. Die Einhaltung von nicht-funktionalen, d.h. performanceorientierten Anforderungen wird aber aufgrund der Komplexität der zur Verfügung stehenden Modellierungs- und Evaluationsmethoden meistens nur stiefmütterlich behandelt. Der auffälligste Grund dafür ist wohl die Weigerung der Designer, die hochspeziellen und mathematischen Formalismen für die Analyse von Warteschlangennetzwerken, Markovschen numerischen Lösungsverfahren oder Ähnliches zu erlernen. Statt dessen wird der zwar beliebte und gängige aber arbeitsintensive und risikoträchtige „fix-it-later"-Ansatz gewählt, der schlicht und ergreifend darin besteht, das System nach der Fertigstellung ausgiebig zu testen und im Falle des Versagens (bezüglich Performance und/oder Funktion) eine Notlösung zu implementieren. Meistens entsteht hierbei „unterdurchschnittlicher" Code und/oder die Notwendigkeit von teurer zusätzlicher Hardware. Hier will das Software Performance Engineering Abhilfe schaffen.

Software Performance Engineering ist eine Methode zur Konstruktion von korrekt arbeitenden Softwaresystemen, die gegebene Performance-Grenzen einhalten. Das Ziel ist die Minimierung des Risikos, dass dieses Vorhaben scheitert (ohne jedoch eine Erfolgsgarantie aussprechen zu können). Dieser Prozess sollte schon in sehr frühen Stadien der Entwicklung eingebaut werden, denn nur so kann die gesuchte Balance zwischen Systemkapazitäten und funktionalen Anforderungen über den gesamten Entwicklungszeitraum beobachtet werden. Die vielen Beispiele von Projekten, die völlig gescheitert sind und/oder deren Kosten in astronomische Höhen geschossen sind, zeigen deutlich, wie wichtig (und rentabel) es ist, sich intensiv und rechtzeitig mit entsprechenden Techniken und Modellen zu befassen und sie effektiv in der Entwicklung

anzuwenden (vgl. [Black (2000)] oder [Smith (1990)]). Von den drei bekannten Vorgehensweisen Monitoring, Antipatterns und Modelling wird in diesem Dokument (bzw. in der PG 459) nur Letzteres weiterentwickelt. Es handelt sich dabei um den aufwändigsten und umfangreichsten Ansatz. Connie U. Smith hat dabei Pionierarbeit geleistet und eine Methodik entwickelt, in jedem Entwicklungsschritt der Software deren Performance anhand von Modellen, die den aktuellen Status repräsentieren, abschätzen und somit weiterverarbeiten zu können (vgl. [Smith (1990)]). Dies beinhaltet entweder eine analytische oder numerische Lösung oder eine Simulation der Modelle. Innerhalb des HIPE-Paradigmas wird diese Aufgabe vom externen Werkzeug HIT (HIerarchical evaluation Tool) des Informatik-Lehrstuhls 4 der Universität Dortmund übernommen (vgl. [Beilner und Stewing (1987)]). Erfolgsentscheidend für die Anwendbarkeit und Aussagekraft dieser Lösungsstrategien bzw. ihrer Ergebnisse ist die Auswahl der Modellnotation. Die Design Community verlangt nach einer Möglichkeit, eine höhere Abstraktionsebene mit Notationen zu erreichen, die möglichst nah an den bereits verwendeten Designformalismen sind. Es erscheint nur logisch, dass sich mittlerweile große Anstrengungen auf die Erweiterung von UML (Unified Modelling Language) mit zeit- und performancemodellierenden Notationen konzentrieren: UML ist ein weit verbreiteter Designstandard und bietet sowohl die Zugänglichkeit für Designer als auch eine komponentenbasierte Struktur. Letztere eröffnet aufgrund ihrer Ähnlichkeit zu Hardwaresystemen deren leistungsfähige Methoden der Performanceanalyse auch dem Software Engineering. Bisher wurden bereits viele Versuche erfolgreich unternommen, UML oder dessen einzelne Bestandteile auf eine Performancebewertung abzubilden: die Fortschritte bei der direkten Simulation von einzelnen UML-Modellen, der Abbildung von Verteilungs- und Kollaborationsdiagrammen auf Warteschlangennetzwerke und der Übertragung von UML auf layered queueing networks (LQN), stochastische Petrinetze und stochastische Prozessalgebren zeigen das Potential der UML-Logik und -Notation für die Definition von Performancemodellen.

HIPE stellt einen Ansatz dar, um die Ideen des Software Performance Engineering (SPE) in den Softwareentwicklungsprozess (zumindest teilweise) zu integrieren. Es wird auf UML-Modellen gearbeitet, die den zu bewertenden Softwareentwurf darstellen. Beim SPE werden bestimmte Bedingungen an das Softwaresystem gestellt. Sie werden in Abhängigkeit von so genannten *Performance Requirements* sowie der verfügbaren Systemfunktionen und Arbeitsrate des Systems formuliert. Hier gilt es bereits, eine der Hürden des Software Performance Engineering zu überwinden, denn die Definition der Performance Requirements ist nicht zu unterschätzen und wird derzeit noch von keinem Standard oder Software-Tool unterstützt. Außerdem sind für ein im Modellierungsstadium befindliches Softwaresystem keine endgültigen Aussagen über Systemfunktionen oder Arbeitsraten der Hardware formulierbar. Dennoch werden die Angaben (wenn auch nur in Form von Schätzungen) benötigt, um ein aussagekräftiges Ergebnis erhalten zu können. HIPE ist da keine Ausnahme. Auch hier müssen direkt in der graphischen Benutzeroberfläche oder im UML-Diagramm (über Notizen, die jeweils mit einem Diagrammelement verbunden sind) Aussagen über das zu bewertende System gemacht werden, die zum Teil sehr spezifisch sind und somit (noch) nicht zum Alltag eines Software-Designers gehören. Als Beispiel seien hier Verzweigungsknoten erwähnt, die in UML-Notation zwar eindeutig formulierbar aber von HIPE nicht in dieser Form verstanden werden können. Es handelt sich letztendlich um logische (also mit einer bestimmten Semantik versehene) Ausdrücke, die erst zur Laufzeit des fertiggestellten Programms Gültigkeit besitzen. Eine Verzweigungsbedingung "`X > 0`" kann offensichtlich nur zur Laufzeit ausgewertet werden und muss daher im Diagramm zu einer Verzweigungswahrscheinlichkeit umformuliert werden (wie im Abschnitt

1.2.6.2 und Abbildung 1.5 auf S. 55 beschrieben). Es obliegt dem Benutzer, eine möglichst genaue Schätzung dieser Wahrscheinlichkeit anzugeben. Mit Hilfe der Wahrscheinlichkeitsangabe kann HIPE schließlich eine statistische Ausgabe erzeugen, die über das Performance-Niveau des Systems Aufschluss gibt. Die dafür von der PG 459 definierten Steuerungsbefehle, die der Modellierer an ein zu bewertendes UML-Modell respektive dessen Elemente annotieren muss, werden im Verlauf dieses Kapitels näher beschrieben. Die Ausgabe wird von HIPE unter Verwendung des Leistungsbewertungstools HIT erzeugt. HIT kann dabei zunächst nur die Werte der Standard-Leistungsmaße liefern (siehe Abschnitt 1.2.4.2). Sie werden von HIPE im Kontext des UML-Diagramms interpretiert und aufbereitet. HIT arbeitet dabei auf so genannten hierarchischen Systemmodellen. Sie beschreiben das zu analysierende System in Form von *Komponenten* und *Services*. Komponenten sind die *Maschinen*, die die durch die Services definierte *Last* verarbeiten. Allgemein bezeichnet man im *Last-Maschine-Konzept* die auszuführenden Aufträge als Last, die von einzelnen oder mehreren Maschinen mit bestimmten Leistungsparametern (wie z.B. Anzahl der Rechenoperationen pro Sekunde) abzuarbeiten sind. "Hierarchisch" heißt ein HIT-Modell deshalb, weil eine Komponente durch ein Subsystem verfeinert werden kann. Somit kann die Modellierung (und ebenso die Analyse) in nahezu jedem erdenklichen Detaillierungsgrad durchgeführt werden. Die *atomaren* Komponenten — also diejenigen, die nicht verfeinert werden können — heißen im HIT-Kontext *Server*. An jeder Komponente können bestimmte Leistungsmaße ermittelt werden, die schließlich zu einer Aussage über die erwartete Leistung des Gesamtsystems führen. Die Leistungsmaße werden im HIPE-Kontext als Ergebnisse der *harten Analyse* bezeichnet. Zusätzlich werden spezielle Komplexitätsmaße für objektorientierte Modelle verwendet, sie gehören zur so genannten *weichen Analyse*. In HIPE werden einige Ergebnisse der weichen Analyse zur Berechnung der Eingabeparameter für die harte Analyse verwendet, wobei das dabei verwendete Prinzip durch das Vorgehen in [Carbone und Santucci (2002)] inspiriert wurde. Sie werden als Last für eine Maschine verwendet, die anhand der Informationen im UML-Diagramm entworfen wird.

Die Aufgaben des Benutzers sind also:

- Die möglichst ausführliche Gestaltung des UML-Modells (damit für eine Analyse durch HIPE möglichst viele Informationen zur Verfügung stehen).
- Das Eintragen entsprechend vieler aussagekräftiger Annotationen für einzelne Diagrammelemente des UML-Modells.
- Die Abfrage der für ihn interessanten Leistungsdaten an entsprechend sinnvollen Stellen des Diagramms.

Obwohl beim Entwurf von HIPE versucht wurde, den Benutzer von den mathematischen und statistischen Details so gut wie möglich abzuschirmen, kann ihm dabei eine Auseinandersetzung mit einigen Details der Software-Leistungsbewertung nicht vollständig erspart bleiben.

Eine Analyse von UML-Modellen mittels HIPE im Sinne des Software Performance Engineering erfordert die Portabilität der Modelle, um sie einlesen, verarbeiten und mit den Ergebnissen versehen wieder ausgeben zu können. Diese Portabilität wird durch eine Speicherung im XMI-Format hergestellt. Wie im Zwischenbericht [PG459 (2005)] bereits dargestellt wurde, erscheint das Modellierungswerkzeug *MetaMill* für diese Aufgabe besonders geeignet. Erstellte Diagramme (bzw. das gesamte Modell) können hiermit in eine XMI-Repräsentation überführt

werden. Danach kann die XMI-Datei an HIPE übergeben und die Analyse gestartet werden. Das folgende Kapitel beschreibt die dafür vorgenommene Übersetzung der XMI-Version der eingegebenen UML-Modelle in eine Repräsentation in der von HIT verwendeten Sprache HI-SLANG sowie das Konzept der Bewertung der einzelnen UML-Konstrukte bzw. der UML-Diagramme. Die Übersetzung in die Zielsprache HI-SLANG erfolgt in bewusster Ähnlichkeit zu den Übersetzungsideen der Vorgänger-PG 438 unter Verwendung einer Darstellung aus Java-Objekten mittels JDOM (vgl. dazu [Hunter und McLaughlin (2004)] oder auch [IBM (2001)]). Erweitert wird diese eher mechanische Transformation dadurch, dass die Eingabemodelle unter Verwendung der UML in der Version 2.0 erfolgen kann (nicht nur in Version 1.4), dass bereits in Anwendungsfalldiagrammen Annotationen zur Lasterzeugung vorgenommen werden können und — die wohl umfassendste Neuerung — dass die Lastbeschreibung von Aktivitäten mit Hilfe von Objektflüssen erfolgen kann. Letzteres beinhaltet eine Analyse der Strukturen des Klassendiagramms und der Beziehungen zwischen den einzelnen Klassen. Der Benutzer erhält insgesamt drei Mittel an die Hand, um eine Lastbeschreibung für die Modellelemente (Anwendungsfälle und Aktivitäten) vorzunehmen:

- Die Verwendung von Objektflüssen in Aktivitätsdiagrammen stellt eine Bindung an ein Klassendiagramm her (also an die geplante Implementierungsstruktur), so dass dessen Informationen ebenfalls mit in die Bewertung einfließen können. Sie werden als Last interpretiert, die an den vom betrachteten Objektfluss beeinflussten Aktivitäten bei deren Abarbeitung entsteht und von der entsprechenden Maschine abgearbeitet werden muss. Die Idee bei der Berechnung dieser Last ist dem bereits erwähnten Vorgehen in [Carbone und Santucci (2002)] entliehen und wird im Abschnitt 1.2.5 näher erläutert. In einer groben Vorstellung kann die Last als Verwaltungsaufwand angesehen werden, der bei der Ausführung der mit einem Objektfluss annotierten Aktivität durch die Erzeugung und Organisation der entsprechenden Klasseninstanzen im Speicher entsteht.

- Direkte Eingaben bezüglich der Lasterzeugung in Form von Prozesserzeugungsraten reduzieren den Analyse-Effekt auf ein Minimum, denn die Angaben sind vom restlichen Modell unabhängig und verschleiern die Beziehungen zur geplanten Implementierungsstruktur. Prozesserzeugungsraten beschreiben dabei die statistische Häufigkeit, mit der eine bestimmte Last (also die einzelnen zu erledigenden Aufträge) auf eine Maschine gegeben wird. Als Beispiel sei hier die negative Exponentialverteilung erwähnt. Sie wird in HIPE mittels `negexp(x)` formuliert und beschreibt anhand des Mittelwerts $\frac{1}{x}$ einer negativ exponentialverteilten Zufallsvariablen, in welchen Zeitabständen die Tasks an der Maschine zur Bearbeitung in die dort vorhandene Warteschlange eingetragen werden. Diese und weitere Verteilungen werden im Abschnitt 1.2.2 beschrieben.

- Fehlen jegliche Eingaben zur Lastbeschreibung, so werden Standardwerte verwendet, die zwar ebenfalls vom Benutzer variiert werden können aber aufgrund ihrer modellübergreifenden Gültigkeit natürlich die Genauigkeit der Analyse extrem negativ beeinflussen. Zur Definition von Standardwerten siehe Abschnitt 1.2.4 und Kapitel 5 ("Benutzungshandbuch").

(Details zu den hier erwähnten Vorgehensweisen, Werkzeugen und Fachbegriffen sind im Zwischenbericht der Projektgruppe 459 nachzulesen, vgl. [PG459 2005].)

1.2 Bewertung von UML-Modellentwürfen

Im Folgenden wird diskutiert, welche Strategie für eine Leistungsanalyse von Softwaremodellen von der Projektgruppe verfolgt wird und wie der Nutzer zu diesem Zweck innerhalb eines UML-Modells Steuerungsbefehle an das HIPE-Tool übergeben muss. Auch die Rückgabe der Ergebnisse der Bewertung in das Modell wird im vorliegenden Abschnitt besprochen. Für die Steuerungsbefehle werden zunächst allgemeine syntaktische Konstrukte und danach für jedes Diagramm explizite und implizite Annotationen (letzteres über die von HIPE verwendete Komplexitätsmetrik von Carbone und Santucci, vgl. [Carbone und Santucci (2002)]) sowie über definierbare Standards zu vergebende Werte diskutiert.

1.2.1 Das Bewertungsszenario

Bewertungsszenarien im allgemeinen sind Zusammenstellungen von UML-Diagrammen und Annotationen von Elementen dieser Diagramme, die in ihrer Gesamtheit eine sinnvolle Leistungsbewertung ermöglichen. Beispiele für Bewertungsszenarien sind das von der PG 438 realisierte HUML-Paradigma (beschrieben in [PG438 (2004)]), das Aktivitäts- und Verteilungsdiagramme in Beziehung zueinander setzt, oder die Simulation von Statecharts, wie sie etwa in der Umgebung SimuLink (siehe [MathWorks (2005)]) realisiert wird.

Das für HIPE entwickelte Bewertungsszenario setzt Anwendungsfall-, Verteilungs-, Klassen-, und Interaktionsdiagramme (d.h. Aktivitäts- und Sequenzdiagramme) in verschiedenen Detailstufen zusammen, die, durch Annotationen erweitert, Vorhersagen über das Leistungsverhalten der beschriebenen Systeme ermöglichen.

Konzeptionell findet die Aufbereitung für eine Leistungsanalyse, die in der Erstellung eines hierarchischen Systemmodells beruht, in Anlehnung an das Konzept zur Leistungsanalyse von Hoeben (vergleiche [Hoeben (2000)]) in folgenden Schritten statt:

1. **Zerlege jeden Anwendungsfall in Methoden.**
   Es wird die Menge der Akteure und der Anwendungsfälle bestimmt, welche von diesen Akteuren angestoßen werden.

2. **Zerlege jede Methode in andere Methoden oder Hardwareressourcen.**
   Anwendungsfälle, zu denen direkt vom Nutzer Angaben gemacht wurden, können explizit auf eine Ressource abgebildet werden. Anwendungsfälle, die durch Aktivitäts- oder Sequenzdiagramme beschrieben werden, werden bis auf die Ebene der atomaren Aktions- oder Pseudozustände zerlegt.

3. **Identifiziere die Klassen, die für die Ausführung der Methoden verantwortlich sind.**
   Die im Rahmen eines solcherart beschriebenen Anwendungsfalls interagierenden Klassen werden identifiziert.

4. **Bestimme, auf welcher Hardware diese Klassen laufen. Bilde die Klassen auf die Nutzung dieser Hardware ab.**
   Über das Deployment-Diagramm wird festgestellt, auf welcher Hardware sie ausgeführt werden. Für jede Hardwarekomponente wird ein HIT-Server bereitgestellt.

5. **Bestimme die Auswirkungen von Netzwerkverbindungen.**
   Weiterhin werden auch die Übergänge zwischen Hardwarekomponenten, das heißt Assoziationen, die über die Grenzen von Komponenten im Verteilungsdiagramm hinausgehen, berücksichtigt, denn sie stellen aufgrund des üblicherweise signifikanten Kommunikationsaufwands auf Netzwerkverbindungen ebenfalls leistungsrelevante Komponenten dar. Auch sie werden als ein HIT-Server dargestellt, der angesprochen wird, falls Kommunikation über die Grenzen von Hardwarekomponenten hinweg erfolgt.

Anhand dieses Vorgehens werden aus den Elementen des zu untersuchenden UML-Modells (und deren Beziehungen zueinander) HIT-Komponenten und -Services generiert, aus denen schließlich ein hierarchisches HIT-Modell zusammengesetzt wird. Das Modell soll hierbei möglichst flach werden, das heißt Angaben, die von HIPE als atomar betrachtet werden sollen, sollen möglichst früh in der Auswertung generiert werden. Relevante Leistungsannotationen, welche in das Modell einbezogen werden sollen, können oftmals auf verschiedene Arten vorgenommen werden, entweder durch direkte Annotation eines Elements, das dadurch vom Nutzer zum atomaren Element erklärt wird, oder durch die Zusammensetzung aus Angaben zu Unterelementen (also z.B. aus Angaben an Elementen eines Aktivitätsdiagramms, das einen Anwendungsfall verfeinert). Falls weder in einer Ebene noch in Unterebenen Angaben gemacht werden, ist es darüber hinaus möglich, für manche fehlende Angaben Default-Werte einzusetzen, so daß dann immer noch eine Leistungsbewertung möglich ist. Zu beachten ist, daß eine Leistungsbewertung umso aussagekräftiger ist, je mehr Angaben vorhanden sind, denn die von HIPE gelieferten Werte stimmen umso besser mit den später tatsächlich realisierten Werten überein, je mehr Details aus einem Entwurf entwickelt werden können.

1.2.2 Lasterzeugung in UML-Modellen

Wie in der Einführung dieses Kapitels bereits beschrieben, arbeitet HIPE nach dem Last-Maschine-Prinzip. Es wird also eine Maschine mit bestimmten Leistungscharakteristika im Hinblick auf ihr Verhalten während der Verarbeitung einer bestimmten Aufgabenmenge - der Last - untersucht. Die Beschreibung beider Teile dieses Konzepts werden von HIPE aus dem UML-Diagramm ermittelt. Eine besondere Betrachtung verdient dabei die Erzeugung der Last. Wie im späteren Abschnitt 1.2.4.3 gezeigt, wird an Akteuren systemweit die Art der Last definiert. Zwei Typen sind dabei möglich: "geschlossene" und "offene" Systemlast (in HIPE werden die Last-Typen durch `closedload` bzw. `openload` angegeben, s.u.).
Für den ersten Typ gilt, dass die Anzahl der in der Maschine vorhandenen Prozesse (also die Größe der Last) während der gesamten Ausführung konstant ist. Dies wird dadurch realisiert, dass mit Hilfe einer Endlosschleife für jeden abgearbeiteten Prozess (der das System also verlässt) ein neuer erzeugt und zur Bearbeitung an das System übergeben wird. An dieser Stelle muss lediglich definiert werden, wie viele Jobs (zu jedem Zeitpunkt) im System vorhanden sein sollen (Schlüsselwort `population`) und die Zeit, die zwischen dem Ende der Bearbeitung eines Jobs und dem Stellen der nächsten Anfrage vergehen soll (`extDelay`).
Der zweite Typ der Lasterzeugung zeichnet sich dadurch aus, dass die Anzahl der im System vorhandenen Jobs nicht konstant ist, sondern beliebig variieren kann. Hier muss also lediglich die so genannte Zwischenankunftszeit angegeben werden, das heißt, in welchen Zeitabständen die Tasks am System ankommen (mittels `occurence`).
Für die beiden Parameter `occurence` und `extDelay` werden häufig die Werte einer mathematisch-stochastischen Funktion — einer so genannten *Zufallsvariablen* — eingesetzt, die eine Beschreibung der Variabilität der Prozesserzeugungsraten bzw. der Zwischenankunftszeiten

erlauben (genauer: es wird stets die Verteilung der Zufallsvariablen betrachtet). Für die Definition der Zufallsvariablen stehen folgende Möglichkeiten zur Verfügung:

- $negexp(x) \in \mathbb{R}$, $x \in \mathbb{R}^+$ — negativ exponentialverteilte Zufallsvariable mit Mittelwert $1/x$. Die Angabe $negexp(2)$ als Prozesserzeugungsrate bewirkt beispielsweise, dass durchschnittlich zwei Prozesse pro Zeiteinheit[1] erzeugt werden. Mit anderen Worten: durchschnittlich alle 0.5 Zeiteinheiten wird ein Prozess erzeugt.

- $coxg(a) \in \mathbb{R}$, $a \in \mathbb{R}^{2\times n}$ — verallgemeinert coxverteilte Zufallsvariable. Diese Funktion bietet die Möglichkeit, eine phasenbezogene Cox-Verteilung zu definieren. Das Argument a ist ein zwei-dimensionales Array, dessen zweite Zeile die Zustandsübergangswahrscheinlichkeiten für die in der ersten Zeile angegebenen Last-Werte definiert. Beispiel: der Aufruf $coxg([[0.2, 0.4, 0.5], [0.6, 0.3, 0.0]])$ repräsentiert die in Tabelle 1.1 und Bild 1.1 auf S. 15 gezeigte Cox-Verteilung. Man beachte, dass die Wahrscheinlichkeiten an den vertikalen Pfeilen im Bild stets die Komplemente der Werte der zweiten Tabellenzeile sind und somit nicht explizit angegeben werden. Ebenfalls wichtig ist die Bedingung, dass zwar die letzte Zustandübergangswahrscheinlichkeit genau 0.0 betragen muss aber alle anderen Werte echt größer als Null sein müssen.

a	1	2	3
1	0.2	0.4	0.5
2	0.6	0.3	0.0

Tabelle 1.1: Ein Eingabe-Array für den Zufallszahlengenerator coxg

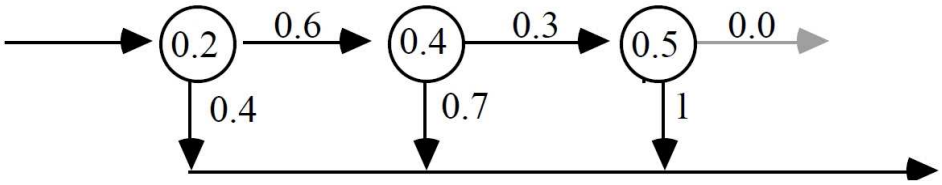

Abbildung 1.1: Die aus der Tabelle 1.1 resultierende Zustandskette mit Lastbeschreibung und Übergangswahrscheinlichkeiten

- $cox(x, y) \in \mathbb{R}$, $x \in \mathbb{R}^+$, $y \in \mathbb{R}$, $y \geq 0.1$ — coxverteilte Zufallsvariable mit Mittelwert x und Variationskoeffizient y. Mit y kann also zusätzlich die durchschnittliche Abweichung der Werte vom Mittelwert x beeinflusst werden. Innerhalb von HIT wird eine cox-verteilte Zufallsvariable in eine coxg-verteilte Zufallsvariable umgewandelt, wobei die Anzahl der Phasen und deren Parameter aus x und y bestimmt werden. Für Details dieser Umwandlung vgl. [Augustin und Büscher (1982)].

- $erlang(\lambda, k) \in \mathbb{R}$, $\lambda, k \in \mathbb{R}^+$ — Faltung von unabhängigen Exponentialverteilungen mit gleichem Mittelwert. Sind $X_1 ..., X_k$ unabhängige und alle mit dem gleichen Parameter

[1] Die Interpretation der Zeit bleibt dem Nutzer überlassen, in diesem Dokument wird eine Interpretation einer Zeiteinheit als eine Sekunde als "natürlich" nahegelegt.

λ exponentialverteilte Zufallsgrößen, so ist $Y := \sum_{i=1}^{k} X_i$ erlangverteilt mit Parametern λ und k (=Anzahl der Summanden). Die Standardabweichung der gezogenen Werte vom Mittelwert $1/y$ beträgt dabei $\frac{1}{\lambda\sqrt{k}}$.

- $normal(x, y) \in \mathbb{R}$, $x \in \mathbb{R}$, $y \in \mathbb{R}_0^+$ — Normalverteilte Zufallsvariable mit Mittelwert x und Standardabweichung y. Für diesen Verteilungstyp wird häufig auch die Bezeichnung "Gauß-Verteilung" verwendet. Deren Graphen sind dementsprechend "gaußsche Glockenkurven".

- $poisson(\lambda) \in \mathbb{N}$, $\lambda \in \mathbb{R}_0^+$ — Diskrete Wahrscheinlichkeitsverteilung (das Ergebnis einer Poisson-Ziehung ist also ein ganzzahliger Wert). Sie ordnet natürlichen Zahlen $n = 0, 1, 2, ...$ die Wahrscheinlichkeiten $P_\lambda(n) = \frac{\lambda^n}{n!}e^{-\lambda}$ zu. Die Verteilungsfunktion ist

$$F_\lambda(n) = \sum_{k=0}^{n} P_\lambda(k) = e^{-\lambda} \sum_{k=0}^{n} \frac{\lambda^k}{k!}$$

  Der Mittelwert der Poisson-Verteilung wird durch λ definiert, die Standardabweichung beträgt $\sqrt{\lambda}$.

- $randint(x, y) \in \mathbb{N}$, $x, y \in \mathbb{N}$, $x \leq y$ — Gleichverteilte diskrete Zufallsvariable auf $\{x, ..., y\} \subset \mathbb{N}$. "Gleichverteilt" bedeutet, dass jeder Wert der Definitionsmenge $\{x, ..., y\}$ mit derselben Wahrscheinlichkeit $p = \frac{1}{|\{x,...,y\}|}$ auftritt.

- $uniform(x, y) \in \mathbb{R}$, $x, y \in \mathbb{R}$, $x \leq y$ — Gleichverteilte kontinuierliche Zufallsvariable mit Definitionsbereich $[x, y] \subset \mathbb{R}$.

- $linear(a, b) \in \mathbb{R}$, $a, b \in \mathbb{R}^n$ — Zufallsvariable mit selbstdefinierter Verteilung. Die Verteilungsfunktion wird erzeugt durch Konstruktion einer linearen Interpolationsfunktion f aus den eingegebenen Stützstellen. Letztere werden in zwei gleich langen Arrays eingegeben (es muss also $lowerbound(a) = lowerbound(b)$ und $upperbound(a) = upperbound(b)$ erfüllt sein). Die Interpretation ist wie folgt: die einzelnen Array-Einträge definieren Stützstellen und Stützwerte der Interpolationsfunktion f, es gilt also $a(i) = f(b(i))$ für $lowerbound \leq i \leq upperbound$. Sie werden linear interpoliert und müssen nicht äquidistant sein. Zur Vermeidung von Fehlern des HIT-Tools (welches diese Funktion bereitstellt) sollten beide Arrays monoton steigend sein (also $a(i) \leq a(i+1)$ und $b(i) \leq b(i+1)$ für $lowerbound \leq i \leq upperbound$) und es sollte $a(lowerbound) = 0$ sowie $a(upperbound) = 1$ gelten.

- $x \in \mathbb{R}^+$ — "deterministische Zufallsvariable". Entsprechend wird an der gegebenen Stelle exakt der Wert x eingesetzt.

- $geom(x) \in \mathbb{N}, x \in \mathbb{N}$ — Geometrische Verteilung. Diese Verteilung ist vom Nutzer nicht explizit verwendbar, wird aber bei der Formulierung von Schleifen (siehe 1.2.6.2) unter Verwendung des `averagerepeats`-Schlüsselwortes zur Bestimmung der letztendlich realisierten Anzahl der Durchläufe herangezogen. Die Wahrscheinlichkeit, daß eine Schleife noch einmal durchlaufen wird, ist zu jedem Zeitpunkt $\frac{x}{x+1}$, entsprechend werden k Durchläufe mit einer Wahrscheinlichkeit von $(\frac{x}{x+1})^k$ durchgeführt. Der Parameter x der Verteilung gibt dabei die erwartete Anzahl an Durchläufen an.

Die Verteilungen haben noch zwei weitere Anwendungen, zum einen bei der Beschreibung der Fehlerrate bei Netzwerkübertragungen und zum anderen bei der direkten Angabe von Bedienzeitanforderungen einzelner Aktivitäten oder Anwendungsfälle. Ersteres beschreibt für zwei miteinander kommunizierende Komponenten des Deploymentdiagramms die zeitlichen Abstände zwischen jeweils zwei Übertragungsfehlern. Da deren Auftreten meist zufällig ist, macht die Verwendung von Zufallsvariablen Sinn. Eine Bedienzeitanforderung stellt eine Angabe über die Zeit dar, die eine Aktivität die sie ausführende Maschine ununterbrochen zur Verfügung haben muss. (Die letztendliche Antwortzeit, das heißt die tatsächliche Ausführungsdauer einer Aktivität, ergibt sich erst im Kontext des dargestellten Systems. Wartezeiten, in denen eine Aktivität nicht ausgeführt wird, und die Aufteilung der Rechenleistung einer Maschine auf mehrere Aktivitäten, führen dazu, dass die die Antwortzeit größer ist als die Bedienzeitanforderung. Die Antwortzeit einer Aktivität ist nur dann gleich der Bedienzeitanforderung, wenn keine Wartezeiten und keine Aufteilung der Rechenleistung auftreten.) Die direkte Angabe von Bedienzeitanforderungen hingegen ermöglicht es dem Benutzer, die Verfeinerung eines Anwendungsfalls oder einer Aktivität zu umgehen (also auch dessen Last nicht berechnen zu lassen) und statt dessen selbst eine Schätzung für die Dauer der Verarbeitung der entsprechenden Aufgabe angeben zu können. Dies macht zum Beispiel Sinn, falls die Modellierung eines speziellen Teils des Systems noch nicht abgeschlossen wurde und/oder dessen Komplexität bekannt ist. Die Zulassung der verschiedenen Verteilungen bei der Angabe der Bedienzeitanforderung ist dabei besonders während des Entwurfs relevant. Im allgemeinen sind die Bedienzeitanforderungen von Aktivitäten abhängig von der Eingabe, die sie bearbeiten sollen, sehr unterschiedlich. Da im Rahmen von Softwareentwürfen in UML die Darstellung aller möglichen Ausführungen einer Anwendung möglich sein soll, aber zur Zeit des Entwurfs natürlich nicht die Bedienzeitanforderungen aller Aktivitäten für alle möglichen Klassen von Eingaben bestimmt werden können, ist es sinnvoller, wenn der Nutzer auf Basis von Erfahrungswerten oder bereits vorliegenden statistischen Angaben hier eine Bedienzeitverteilung angeben kann (vergleiche dazu auch die Anmerkungen zu den deterministischen Performancemodellen auf Seite 77 im Abschnitt 1.2.9).

1.2.3 Definition von Maschinen in UML-Modellen

Die Maschinen des Last-Maschine-Paradigmas werden in HIPE durch so genannte *Komponenten* eines HIT-Leistungsmodells repräsentiert. Sie erfüllen die Aufgabe, die anstehenden Prozesse abzuarbeiten. Sie können sowohl in atomarer Form vorliegen (*Server*) als auch durch weitere Subsysteme verfeinert werden (*benutzerdefinierte Komponente*). In jedem Fall müssen zunächst ihre Leistungscharakteristika definiert werden, um sinnvolle Messungen an ihnen durchführen zu können. Für verfeinerte Komponenten geschieht dies durch die Modellierung des Subsystems, insbesondere durch die Definition der darin verwendeten Maschinen (die ebenfalls wieder aus Submodellen zusammengesetzt sein können). Aus UML-Sicht wird dies gerade durch Verfeinerung eines Anwendungsfalls oder einer Aktivität durch ein Aktivitätsdiagramm bewerkstelligt. Dieses Konstrukt wird von HIPE in ein Analogon im Leistungsmodell übersetzt. Atomare Komponenten hingegen (auch "Ressourcen" genannt) werden mit Hilfe konkreter Leistungsangaben im Verteilungsdiagramm spezifiziert. In HIPE wird zwischen *aktiven* und *passiven* Ressourcen unterschieden. Aktive Ressourcen operieren auf den eingegebenen Daten und können zum Beispiel CPUs oder Hardwarecontroller repräsentieren, passive Ressourcen sind meist Datenträger, flüchtiger Speicher (RAM etc.) oder Netzwerkverbindungen. Die Res-

sourcen werden in UML im Deploymentdiagramm definiert und in anderen Diagrammen als verwendete Hardware referenziert, beispielsweise mittels

`let cpu = <Name einer aktiven Ressource im Deploymentdiagramm>;`.

Für eine Beschreibung aktiver Ressourcen werden drei Parameter benötigt: `speed`, `dispatch` und `schedule`. Der numerische Wert der Variable `speed` bezeichnet die Verarbeitungsgeschwindigkeit der Ressource in Operationen pro Zeiteinheit (in HIPE stets Sekunden). Über `dispatch` legt der Benutzer fest, ob jeder gerade bearbeiteten Aufgabe dieselbe "volle" Rechenkraft der Hardware zur Verfügung stehen oder diese unter den Aufgaben gleichmäßig aufgeteilt werden soll (`equal` bzw. `shared`). Die Behandlung der Situation, dass mehrere Aufgaben an der Ressource zur Bearbeitung anstehen, kann mittels `schedule` gesteuert werden: soll jeder ankommende Job direkt in die Komponente übernommen (und die Hardware somit zur Parallelverabeitung befähigt) oder eine Warteschlange eingerichtet werden, die vorn gefüllt und hinten durch Entnahme einzelner Jobs bei Fertigstellung eines bearbeiteten Jobs entleert wird (`immediate` bzw. `fcfs`)? Eine dritte Möglichkeit für das Scheduling ist die zufällige Auswahl eines der in der Warteschlange vorhandenen Jobs (`random`).
Datenspeicher benötigen in ihrer Eigenschaft als passive Ressourcen eine Angabe zur Größe des verfügbaren Speichers (`size`) und der Zugriffszeit beim Lesen und Schreiben (`accesstime`). Datentransportleitungen hingegen werden über ihre Bandbreite (`speed`) und Fehlerrate (`error`) spezifiziert. Letzteres gibt an, wie viel Zeit im Durchschnitt zwischen zwei Übertragungsfehlern liegt, wie häufig also ein Datenpaket fehlerhaft am Empfänger ankommt oder beim Transport verloren geht. Zur exakten Spezifikation sei hier auf die Grammatik der HIPE-Statements in Abschnitt 1.2.4.5 verwiesen.

1.2.4 Syntax von HIPE-Statements

Das HIPE-Tool erwartet alle Eingaben zu einem Element eines UML-Modells als textuelle Angaben auf einem Notizzettel (in MetaMill als *Note* bezeichnet), der über eine Ankerkante (in MetaMill *Anchor* genannt) mit dem jeweiligen Element verbunden wird. Viele dieser Eingaben definieren direkt Belegungen für Variablen, die für eine Leistungsanalyse von HIT benötigt werden. So beschreibt beispielsweise der `speed`-Parameter die Ausführungsgeschwindigkeit einer atomaren Komponente in HIT, also einer nicht weiter verfeinerten Maschine des Last-Maschine-Konzepts. Er wird ohne weitere Behandlung oder Interpretation an die entsprechende Stelle im HI-SLANG-Sourcecode übernommen. Andere Variablen wiederum müssen interpretiert werden und dienen zur Verfeinerung des Modells oder zur Experiment-Deklaration (z.B. `diagram` oder `load`, siehe dazu unten).

Ein `.HIPE`-Block schließt auf einem Notizzettel alle Angaben ein, die in das Leistungsmodell einfließen sollen. Für jede Notiz ist dabei nur ein `.HIPE`-Block erlaubt. Es gibt drei Typen von Anweisungen:

1. Mit LET-Statements werden Variablen gesetzt, die für das Modell oder die Auswertung relevant sind.
2. GET- oder REQUIRED-Statements sind Anweisungen, die Berechnungen von Leistungsmaßen anstoßen.
3. STOP-Statements ermöglichen die Definition von Abbruchbedingungen, anhand derer die mathematische oder simulative Analyse des Modells beendet werden soll.

Einige LET-, GET- und REQUIRED-Statements werden auf den nächsten Seiten beispielhaft unter Verwendung der Notation einer Grammatik in Backus-Naur-Form (BNF) beschrieben. Die Beschreibung der STOP-Statements findet sich im darauf folgenden Abschnitt 1.2.4.4. Anschließend wird zur exakten Definition aller möglichen HIPE-Statements deren Grammatik in Backus-Naur-Form aufgeführt. Eine graphische Übersicht mit einer beispielhaften Auswahl an HIPE-Statements ist für Anwendungsfalldiagramme in Abbildung 1.2 auf S. 19, für Aktivitätsdiagramme in Abbildung 1.3 auf S. 20 und für Deployment-Diagramme in Abbildung 1.4 auf S. 21 zu finden. Man beachte, dass die einzelnen Bilder kein zusammenhängendes Beispielmodell darstellen, sondern lediglich für jeden UML-Diagrammtyp einige Möglichkeiten zur Annotation von Diagrammen aufzeigen[2].

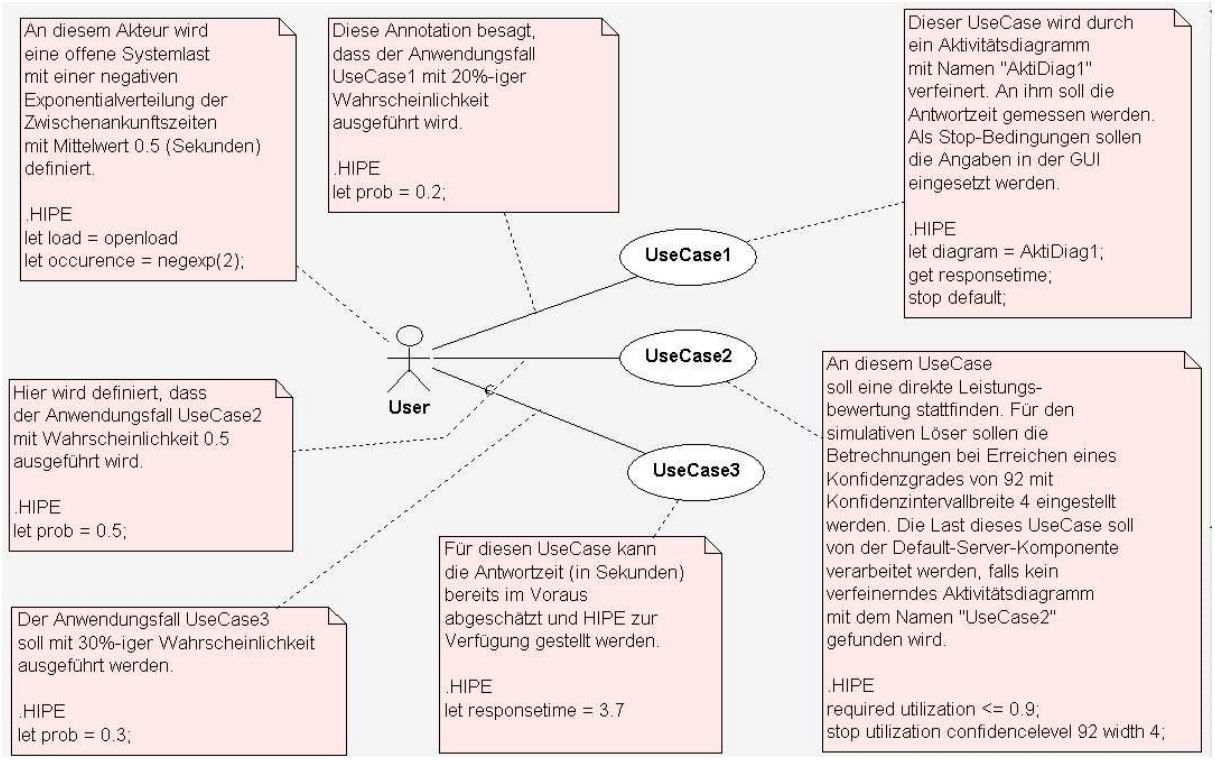

Abbildung 1.2: Beispielhafte Annotationen an Elementen eines Anwendungsfalldiagramms

1.2.4.1 LET-Statements

Mit LET-Statements werden im Rahmen des Modells Variablen gesetzt. LET-Statements besitzen folgende allgemeine Gestalt:

```
LET <name> = <value>;
```

[2]Dabei wurde in Einzelfällen aus Zeitgründen darauf verzichtet, minimale syntaktische Ungenauigkeiten zu korrigieren (zu diesem Umstand siehe auch die Bemerkungen im Produkttest).

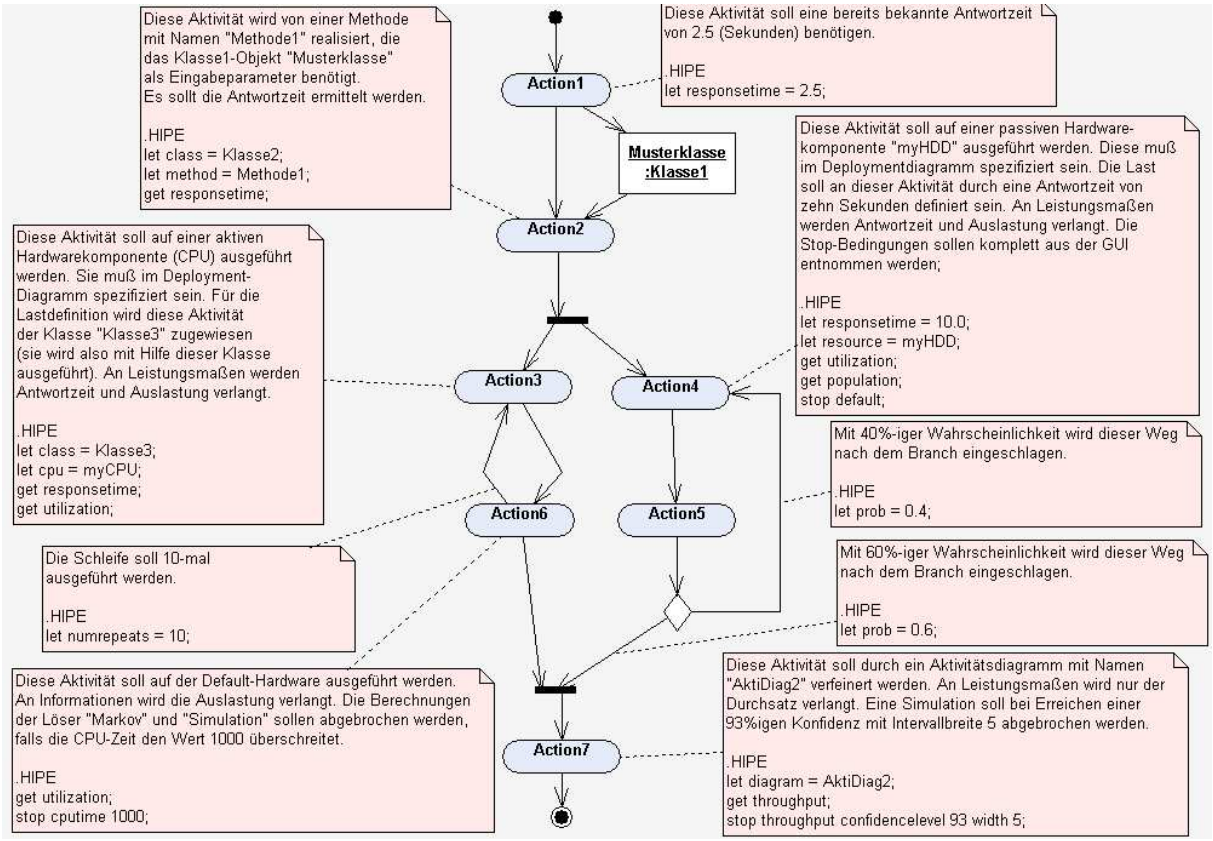

Abbildung 1.3: Beispielhafte Annotationen an Elementen eines Aktivitätsdiagramms

Dabei bezeichnet `<name>` zulässige Variablennamen und `<value>` zulässige Belegungen für diese Variablen. Bei manchen Variablen sind neben Einzelangaben auch Listen einzusetzender Werte erlaubt. Damit werden in der Regel Experimentserien (s.u.) definiert. Zur Definition einer Experimentserie werden die einzelnen Listenelemente durch das Zeichen '`&`' ("Kaufmanns-UND") getrennt. Die einzelnen Elemente einer Liste von Angaben für das `arrivaltime`-Statement, die innerhalb eines einzelnen Experiments verarbeitet werden, werden jedoch durch normale Kommata ('`,`') getrennt[3]. Numerische Werte, die eine zeitliche Angabe definieren (s.u.) werden dabei im Sinne einer natürlichen Interpretation als Spanne in Zeiteinheiten oder als Anzahl von Operationen pro Zeiteinheit aufgefaßt.

Beispiele für zulässige Angaben sind:

- `LET occurence = negexp(3);` definiert die Zwischenankunftszeit für den so annotierten Prozess als Wert einer negativ exponentialverteilten Zufallsvariablen mit Erwartungswert 1/3.

[3]Diese Trennung einzelner Elemente ist nicht mit einer Aufzählung im Sinne der Experimentserien zu verwechseln. Letztere verwendet das „Kaufmanns-UND“ als Trennzeichen.

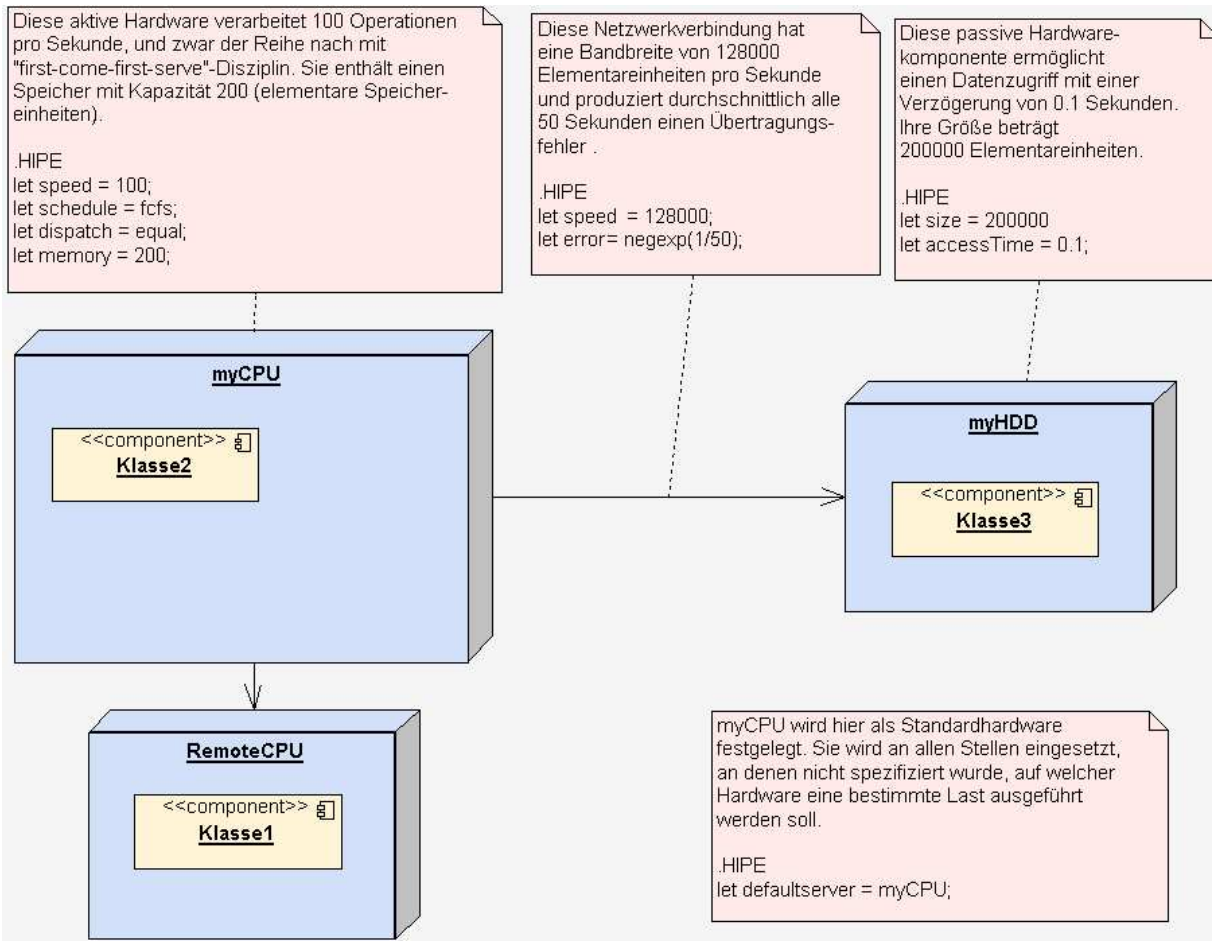

Abbildung 1.4: Beispielhafte Annotationen für ein Deploymentdiagramm

- `LET responsetime = 180;` setzt die Bedienzeitanforderung[4] der betrachteten Aktivität auf 3 Minuten (= 180 Sekunden). Die letztendliche Antwortzeit für diese Aktivität ergibt sich jedoch erst aus der Analyse des resultierenden Leistungsmodell, insbesondere natürlich in Abhängigkeit von der Leistung des für diese Aktivität zuständigen Servers.

Experimentserien[5]

LET-Statements ermöglichen die Definition von Experimentserien, also mehrere Durchläufe einer Analyse desselben Modells mit verschiedenen Werten für bestimmte Variablen. Eine Experimentserie kann an HIPE dadurch übergeben werden, dass statt eines einzigen Eintrags für `<value>` mehrere durch das Zeichen '`&`' ("Kaufmanns-UND") getrennte Belegungen für diese Variable angegeben werden, also z.B.

```
LET speed = 10 & 50 & 100;
```

[4] Man beachte die von den bei der Leistungsbewertung von Software(modellen) geltenden Standards abweichende Verwendung des Begriffs „responsetime"!

[5] Bei der Verwendung von Experimentserien sind die in Abschnitt 5.7.2 (Seite 208) beschriebenen Restriktionen zu berücksichtigen.

Hier würde die gesamte Analyse mehrfach durchgeführt, wobei jeweils dem `speed`-Parameter der so annotierten Komponente die Werte 10, 50 und 100 zugewiesen würden. Mit Hilfe von Experimentserien kann also das Verhalten eines Systems (bzw. dessen Modells) unter Variation einzelner Charakteristika untersucht werden. Analog definieren Parameterlisten an anderen Elementen des UML-Modells Experimentserien (sofern die Syntax sie dort erlaubt). Es steht dem Nutzer frei, auch mehrere Experimentserien (also Parameterlisten an mehreren verschiedenen Stellen des UML-Modells) zu definieren. Bei zu vielen Experimentserien oder "ungünstig" gewählten Werten läuft man jedoch schnell Gefahr, extrem lange auf ein Analyse-Ergebnis warten zu müssen. HIPE warnt den Benutzer daher bei Entdeckung von mehr als zwei Experimentserien und bietet den sofortigen Abbruch der Analyse an.

1.2.4.2 GET- und REQUIRED-Statements

GET-Statements werden im allgemeinen als `GET <name>;` angegeben, wobei `<name>` ein oder mehrere von HIPE per harter Analyse zu berechnende Maße angibt (also z.B. Auslastung, Population usw. — s.u.). Für jedes der angegebenen Maße liefert HIPE anschließend den Wert zurück. Zu beachten ist, daß nicht alle Maße in jedem Kontext sinnvoll zu berechnen sind. Die Korrektur einer eventuell nicht verwertbaren Eingabe bezüglich der Leistungsmaße erfolgt dabei automatisch, indem Maße, die nicht berechenbar sind, auch nicht erhoben werden.
Eine Möglichkeit, um eine qualitative Bewertung anzufordern (d.h. eine Komponente erfüllt eine Anforderung oder erfüllt sie nicht), ist das REQUIRED-Statement, das die Gestalt

`REQUIRED <name> <Relation> <value>;`
mit `<Relation> ::= <= | < | = | > | >=`

hat. Es besagt, daß von HIPE ein Leistungsmaß `<name>` für die solcherart annotierte Komponente erhoben und anschließend mit dem geforderten Wert `<value>` bezüglich der Relation `<Relation>` verglichen werden soll.

Beispiele für zulässige Angaben sind

- `GET throughput;` liefert für ein Element den Durchsatz in Prozessen pro Sekunde, die von ihm verarbeitet werden können.
- `REQUIRED utilization <= 0.9;` definiert das hiermit annotierte Element, falls sich seine Auslastung größer als 0.9 ergibt, als Flaschenhals.

Wird ein UML-Diagramm-Element mit einem GET- oder REQUIRED-Statement (und einem harten Maß) versehen, so wird an derjenigen Komponente, die im erzeugten hierarchischen Leistungsmodell dieses Element repräsentiert, das entsprechende Leistungsmaß mit Hilfe von HIT erhoben. HIT arbeitet dabei mit stochastischen und simulativen Mitteln, um das Leistungsmodell (bzw. dessen Komponenten) unter der Verarbeitung der erzeugten Last zu beobachten. Eine solche Beobachtung bezieht sich immer auf das Vorhandensein von zu erledigenden Tasks in den Komponenten (relativ zur verwendeten Zeiteinheit). Die Leistungsmaße wiederum bieten jeweils eine bestimmte Sicht auf die Daten, die die Beobachtung der Komponenten ergeben hat. Die von HIPE verwertbaren Maße sind:

- **Throughput** (Durchsatz) — Die Anzahl der die Komponente verlassenden (also fertiggestellten) Prozesse pro Zeiteinheit. Der Wert dieses Leistungsmaßes ist stets relativ

zur Menge der in der Komponente vorhandenen Prozesse (bzw. deren Ankunftsrate) zu sehen. Ein Wert ist nur dann als "hoch" (und die entsprechende Komponente somit als leistungsstark oder "schnell") zu bezeichnen, falls er die gleiche Größenordnung wie die Anzahl der vorhandenen Prozesse (bzw. ihre Ankunftsrate) hat.

- **Turnaroundtime** (Antwortzeit, Synonym: **Responsetime**) — Gesamtzeit, die ein Prozess in der Komponente verbringt, bis er abgefertigt ist. Abstrahiert man von den einzelnen Prozessen, so beschreibt beispielsweise die Turnaroundtime eines Anwendungsfalls gerade die Zeit, die die Maschine braucht, um die dort entstehende Last abzuarbeiten. Für einen Akteur beschreibt diese Größe also, wie lange man durchschnittlich auf das Ergebnis einer Anfrage an das System warten muss. Dies ist für den Anwender des zu implementierenden Systems die wohl wichtigste Größe, denn es wird bei dessen Entwicklung stets implizit gefordert, dass das Ergebnis der zu erfüllenden Funktion möglichst schnell zur Verfügung gestellt wird. Hier kann das System also bereits im Sinne des Anwenders untersucht und optimiert werden (siehe auch die Anmerkung zu Akteuren im Abschnitt 1.2.4.3).

- **Population** — Anzahl der in der Komponente anwesenden Prozesse pro Zeiteinheit.

- **Utilization** (Auslastung) — Quotient aus geleisteter Arbeit und maximal möglicher zu leistender Arbeit. Dieses Maß ist in HIT nur für Komponenten vom Typ "`Server`" (also atomare Komponenten) definiert. Das bedeutet, dass dieses Maß im UML-Diagrammen nur an solchen Elementen verlangt werden darf, die nicht weiter verfeinert werden. Mit anderen Worten: es ist nicht erlaubt, an einer Aktivität eines Aktivitätsdiagramms die Berechnung der Auslastung zu verlangen, wenn diese Aktivität durch ein weiteres Aktivitätsdiagramm noch genauer beschrieben wird.

- **Occupation** — Die Wahrscheinlichkeit, dass die betrachtete Komponente nicht leer ist. Hohe Werte dieses Leistungsmaßes deuten darauf hin, dass häufig Prozesse an der betrachteten Komponente zur Bearbeitung ankommen. In Verbindung mit einer hohen Auslastung kann dies so interpretiert werden, dass die betreffende Komponente fast an ihre Leistungsgrenze stößt. Dieses Leistungsmaß ist bei Verwendung eines analytischen Lösers (siehe Abschnitt 1.2.4.4) nicht erlaubt.

1.2.4.3 Annotationen für verschiedene UML-Diagramme

Im **Use-Case-Diagramm** muss die Arbeitslast spezifiziert werden. Dies geschieht über Annotationen am Akteur mit Hilfe der LET-Statements.
Für das offene System gilt:

```
LET load = openload;
LET occurence = <DistributionList>;
```

Für das geschlossene System gilt:

```
LET load = closedload;
LET population = <NumericalList>;
LET extDelay = <DistributionList>;
```

`extDelay` beschreibt die Verzögerungszeit zwischen der Fertigstellung des einen Auftrages und dem Beginn der Abarbeitung des nächsten. Für die Population (also die Anzahl an Aufträgen im System) wird für die geschlossene Systemlast eine Parameterliste `<NumericalList>` (mit evtl. nur einem Element) gesetzt. Hat die Liste mehr als nur ein Element, so wird dadurch — wie im vorigen Abschnitt 1.2.4.1 beschrieben — eine Experimentserie definiert.

Eine andere Möglichkeit der Last-Definition ist

```
LET load = openload;
LET arrivaltime = <ArrivalTimeList>;
```

beziehungsweise

```
LET load = closedload;
LET arrivaltime = <ArrivalTimeList>;
```

Damit kann explizit angegeben werden, zu welchen (auch und insbesondere unregelmäßigen) absoluten Zeitpunkten — im Gegensatz zu den relativen Zwischenankunftszeiten — Prozesse vom Typ des Akteurs in das Modell eintreten sollen. Zu beachten ist, dass die einzelnen Werte in der `<ArrivalTimeList>` durch Kommata getrennt werden, da hier keine Experimentserie, sondern eine Menge von Ereignissen definiert wird, die innerhalb eines Experiments auftreten.

Bei der Annotation von Akteuren ist als Besonderheit zu beachten, dass an ihnen nur die Berechnung der Antwortzeit zulässig ist (`GET responsetime;`). Dies begründet sich in der Tatsache, dass laut UML-Spezifikation die Akteure lediglich die Ausführung von Anwendungsfällen anstoßen und somit als Schnittstelle des Systems zur Umwelt (also sowohl zum menschlichen Benutzer als auch zu anderen Systemen) anzusehen sind. In der Einführung wurde bereits beschrieben, dass an diesen Schnittstellen sinnvollerweise die Antwortzeit (Turnaroundtime) als einzig interessantes Datum betrachtet wird. Dies spiegelt sich auch in der Realität wider; beispielsweise erwartet man als Kunde eines Bankautomaten lediglich die zügige (und korrekte) Bearbeitung der angeforderten Dienste und nicht etwa die geringe Auslastung der CPU.

Für die einzelnen Anwendungsfälle können spezielle Angaben über die Last angegeben werden. Falls es für den Entwickler möglich ist, die Bedienzeitanforderung für einen Anwendungsfall abzuschätzen, kann er diesen hier angeben, um den Rechenaufwand zu verringern. Das entsprechende LET-Statement lautet:

```
LET responsetime = <DistributionList>;
```

Dabei ist `<DistributionList>` eine Liste von reellen Zahlen und/oder Beschreibungen von verschieden verteilten Zufallszahlen in der ab Seite 14 angegebenen Syntax (möglicherweise mit nur einem Element).

Ein Anwendungsfall kann durch ein Aktivitätsdiagramm verfeinert werden. Der Verweis auf das entsprechende Diagramm geschieht über das Schlüsselwort `diagram`:

```
LET diagram = <String>;
```

`<String>` ist der Name des entsprechenden Aktivitätsdiagramms. Für jeden Anwendungsfall, an dem kein solches Statement gefunden wird, sucht HIPE nach einem Aktivitätsdiagramm

mit demselben Namen wie der Anwendungsfall. Bei positivem Suchergebnis wird dieses automatisch als Verfeinerung für den entsprechenden Anwendungsfall eingesetzt.

Sowohl für einen Anwendungsfall, als auch für eine Aktivität kann der Entwickler angeben, welche harten Maße gemessen werden sollen. Dies geschieht mit Hilfe von GET-Statements, also beispielsweise über das bereits gesehene

```
GET responsetime;.
```

Mittels REQUIRED-Statements kann der Entwickler die gewünschten Werte für die zu berechnende Maße angeben und diese später mit den gemessenen Werten vergleichen lassen, z.B.

```
REQUIRED responsetime < 42;.
```

Es ist sinnvoll, für die einzelnen Anwendungsfälle anzugeben, mit welcher Wahrscheinlichkeit diese ausgeführt werden. Eine solche Angabe hat die Gestalt

```
LET prob = <ProbValue>;
```

Dabei ist `<ProbValue>` eine Zahl zwischen 0 und 1.

Ein graphisches Übersichtsbeispiel mit einer Auswahl an HIPE-Statements für Anwendungsfalldiagramme ist in Abbildung 1.2 auf S. 19 zu finden.

In **Aktivitätsdiagrammen** können weitere Angaben zur Spezifizierung der Last gemacht werden. Eine Aktivität kann auf verschiedene Arten als atomar definiert werden: falls die Bedienzeitanforderung für eine Aktivität bekannt ist, kann man diese über

```
LET responsetime = <DistributionList>;
```

angeben. Ist bekannt, welche Klasse die Aktivität ausführt, kann das mittels

```
LET class = <String>;
```

angegeben werden. Dann wird anhand des Verteilungsdiagramms festgestellt, wie groß der Kommunikationsaufwand für den Transport aller durch Objektfluss dieser Klasse zwecks Ausführung der dargestellten Aktivität zugeführten Daten ist. Die Angabe

```
LET method = <String>;
```

bindet weiterhin eine Aktivität an eine Methode der bereits angegebenen Klasse. Mit Hilfe des in Abschnitt 1.2.5 dargestellten Komplexitätsmaßes wird für die Methode eine geschätzte Ausführungszeit festgestellt.

Weiterhin kann eine Aktivität mit Hardware assoziiert werden. Mit Statements der Form

```
LET cpu = <String>;
LET resource = <String>;
```

kann eine Ausführung auf einer CPU und darüber hinaus die Benutzung passiver Hardwarekomponenten angegeben werden. Diese Angaben überschreiben die Informationen, die man aus dem Verteilungsdiagramm über die Verbindung einer annotierten Klasse mit Hardwarekomponenten beziehen kann. Falls diese Angaben und das Verteilungsdiagramm fehlen, wird der Aktionszustand auf einem Default-Server ausgeführt (s.u.). Der Umfang, mit dem eine passive Ressource genutzt wird, wird mit

```
LET resourceusage = <Numerical>;
```

angegeben.

Andererseits kann die dargestellte Aktivität durch ein weiteres Aktivitätsdiagramm verfeinert werden. Mit dem Statement

```
LET diagram = <String>;
```

kann entsprechend verwiesen werden. Ist kein Diagramm als Verfeinerung und keine Annotation zur Definition des entprechenden Elements als atomares Element angegeben, dann wird automatisch ein Aktivitätsdiagramm mit dem Namen der Aktivität als Verfeinerung einzusetzen versucht. Gibt es keine solche Verfeinerung oder träte beim Einsetzen einer solchen Verfeinerung eine Rekursion auf, wird die Ausführung der betrachteten Aktivität bezüglich des Default-Servers und eines Vorgabewertes für ihre Zeitdauer bewertet.

Zur Charakterisierung von Verzweigungszuständen können folgende Angaben an den ausgehenden Kanten des Verzweigungsknotens gemacht werden: Eine Kante kann mit `LET prob = <ProbValue>` annotiert werden, wobei stets $0 <$ `<ProbValue>` ≤ 1 gelten muß.

Schleifen können über die Kantenannotationen

```
LET numrepeats = <Numerical>;
```

oder

```
LET averagerepeats = <Numerical>;
```

charakterisiert werden, wobei `numrepeats` eine absolute Anzahl und `averagerepeats` eine Anzahl als Erwartungswert einer geometrischen Verteilung angibt. Sie stehen an den Kanten, die den Schleifenkörper definieren, also üblicherweise an Rückwärtskanten. Eine graphische Übersicht mit einer Auswahl an HIPE-Statements für Aktivitätsdiagramme ist in Abbildung 1.3 auf S. 20 zu finden.

Im **Verteilungsdiagramm** können Angaben zur Beschreibung der Hardware mit Hilfe von LET-Statements gemacht werden. Für alle Ressourcen kann ein Parameter `speed` angegeben werden, für aktive Ressourcen (CPUs) darüber hinaus `dispatch` und `schedule`, für passive Ressourcen (Datenträger) `size` und `accesstime`, für Leitungen (interne und externe Verbindungen) `minpacketsize`, `maxpacketsize` und `error`.

GET-Statements stoßen im Verteilungsdiagramm die Bestimmung hardwarespezifischer Leistungsmaße an, wobei `GET <Measure>;` den Wert des Maßes `<Measure>` zurückliefert. Die Interpretation der zurückgelieferten Größe erfolgt dabei analog zu dem Schema auf Seite 22,

also wird beispielsweise der Wert zum Maß Throughput als Anzahl abgefertigter Prozesse pro Zeiteinheit interpretiert, während das Maß Turnaroundtime einen Wert in Zeiteinheiten liefert und das Maß Occupation einen dimensionslosen Wert, nämlich eine Wahrscheinlichkeit liefert. Um eine Aktion auf einer Standard-Hardwarekomponente ausführen lassen zu können, muss diese als solche definiert sein. Im Deployment-Diagramm kann für diesen Zweck ein Default-Hardware-Statement angegeben werden:

```
LET defaultServer = myCPU;
```

definiert beispielsweise die aktive Standard-Hardware als die Komponente, die im Deploymentdiagramm den Namen "myCPU" trägt. Somit ist sie für solche Diagrammelemente zuständig, deren Annotation zwar eine Erhebung von Leistungsmaßen erfordert, aber keine Hardware für ihre Ausführung spezifiziert. Man beachte, dass das Statement auf einer Notiz stehen muss, die nicht mit einem Diagrammelement des Deploymentdiagramms verbunden ist! Eine Auswahl an HIPE-Statements für Verteilungsdiagramme ist in Abbildung 1.4 auf S. 21 zusammengestellt.

1.2.4.4 Das Analyse-Backend von HIPE: HIT

Um den Sinn der verbleibenden Statements der HIPE-Grammatik verdeutlichen zu können, muss an dieser Stelle zunächst auf das im Hintergrund agierende Programm HIT eingegangen werden, welches von HIPE für die Durchführung der harten Analyse verwendet wird. Es akzeptiert das von HIPE erzeugte hierarchische Systemmodell und führt die geforderten Messungen auf diesem durch. Letztere werden in Form von so genannten *Experimenten* spezifiziert, die folgende Informationen beinhalten:

- **Evaluationsobjekte** beschreiben, an welchen Stellen des Modells bestimmte Messergebnisse ermittelt werden sollen. Mit "Stellen" sind hier die einzelnen Elemente des Leistungsmodells gemeint, das von HIPE aus dem Eingabe-UML-Modell erzeugt wird. Bis auf einige vor dem Nutzer verborgene Sonderfälle stimmen dabei die Elemente des UML-Diagramms mit denen des Leistungsmodells überein. Eine solche Ausnahme ist zum Beispiel das Konstrukt zur Repräsentation von UML-Verzweigungsknoten und -Synchronisationsknoten. Die Anfrage von Messergebnissen wird vom Nutzer mit den bereits beschriebenen `GET`- und `REQUIRED`-Statements und den aus Abschnitt 1.2.4.2 bekannten Leistungsmaßen (in HIT *Streams* genannt) an Diagrammelementen des UML-Modells vorgenommen und von HIPE in entsprechende Evaluationsobjekte transformiert. Weiter kann für jeden Ergebnistyp anhand der so genannten *Estimator* (*Schätzer*) die Qualität des Ergebnisses angegeben werden. Die verfügbaren Schätzer sind: `MEAN` (Mittelwert), `STANDARDDEVIATION` (Standardabweichung), `BOUNDS` (absolute obere und untere Grenze der Abweichung) und `CONFIDENCE LEVEL` (Zuverlässigkeitsniveau). Die ersten drei bezeichnen die gleichnamigen Eigenschaften von stochastischen Zufallsvariablen, welche den Berechnungen stets zugrunde liegen (vergleiche dazu auch Abschnitt 1.2.2). Das Zuverlässigkeitsniveau wird weiter unten genauer betrachtet. Die Angabe der gewünschten Schätzer geschieht in der graphischen Benutzeroberfläche.

- Die **Lösungsmethode** gibt an, auf welchem Weg die Ergebnisse der Systemanalyse berechnet werden sollen. HIT stellt dafür sowohl rein mathematische als auch eine simulative Methode zur Verfügung. Eine Simulation kann man sich dabei als eine "Ausführung" des Modells vorstellen, an der quasi empirisch die Leistungsmaße gemessen werden. Die mathematischen Löser arbeiten auf Warteschlangennetzen oder Zustandsautomaten

und berechnen die Leistungsmaße mit Hilfe entsprechender numerischer oder analytischer Verfahren. Da jede Simulation potentiell unendlich lange laufen kann, müssen Prämissen formuliert werden, bei deren Erfülltsein die Betrachtungen eingestellt werden sollen. Auch die mathematischen Löser können aufgrund ihrer potentiell unendlich großen Komplexität anhand bestimmter Abbruchbedingungen dazu gezwungen werden, ihre Arbeit einzustellen (s.u.). Die Angabe des gewünschten Lösers geschieht in HIPE in der Benutzeroberfläche. Ihre Namen sind der Tabelle 1.2 (Seite 28) zu entnehmen.

Löser (Schlüsselwort)	mögliche Abbruchbedingung(en) (Schlüsselwörter)	Verbindlichkeit, Definitionsmöglichkeiten
Simulation (`SIMULATIVE`)	Dauer der Simulation in Echtzeit (`CPUTIME ...`), Dauer der Simulation in Modellzeit (`MODELTIME ...`), Zuverlässigkeit der Ergebnisse (`CONFIDENCE LEVEL ... WIDTH ...`)	mindestens eine dieser Abbruchbedingungen erforderlich, pro Evaluationsobjekt definierbar
analytisch/numerisch, Algorithmus "Markov" (`ANALYTICAL_MARKOV`)	Messgenauigkeit (`ACCURACY ...`), Dauer der Simulation in Echtzeit (`CPUTIME ...`)	optional, modellweit in der GUI definiert; optional, modellweit in der GUI definiert
analytisch, Algorithmus "LIN2" (`ANALYTICAL_LIN2`)	Messgenauigkeit (`ACCURACY ...`)	optional, modellweit in der GUI definiert
analytisch, Algorithmus "DOQ4" (`ANALYTICAL_DOQ4`)	—	nicht anwendbar

Tabelle 1.2: Abbruchbedingungen für die Analyse

- Die **Abbruchkriterien** definieren die Bedingungen, die festlegen, wann die Analyse beendet werden soll. Für die verschiedenen Löser existieren unterschiedliche Möglichkeiten, einen Abbruch herbeizuführen. Die Simulation erzwingt deren Definition und ermöglicht sie für jedes einzelne Evaluationsobjekt. Für die mathematischen Löser gilt, dass `ANALYTICAL DOQ4` sämtliche Stop-Bedingungen ignoriert und `ANALYTICAL LIN2` sowie `ANALYTICAL MARKOV` sie als Option anbieten. Allerdings darf hierbei pro Experiment nur genau eine Stop-Bedingung von jedem der beiden Typen `CPUTIME` und `ACCURACY` existieren (s.u.). Das bedeutet, dass im mathematischen Fall die Stop-Bedingungen nicht für jedes Evaluationsobjekt einzeln sondern nur "modellweit" definiert werden können. Aus diesem Grund werden bei Auswahl eines der genannten mathematischen Löser sämtliche Stop-Bedingungen in Annotationen von UML-Diagrammelementen ignoriert und statt

dessen die in der Benutzeroberfläche angegebenen Werte verwendet. Tabelle 1.2 zeigt die möglichen Kombinationen. Folgende Abbruchbedingungen werden dabei verwendet:

- `CPUTIME` bezeichnet die Zeit, die die reale CPU für die Analyse des Modells aufbringen muss. Das heißt, es werden die verbrauchten Zeiteinheiten quasi in Echtzeit gemessen. Dieses Maß ist für einige Sonderfälle von der jeweils aktuellen Arbeitslast abhängig, der die CPU ausgesetzt ist. Für die Simulation gilt beispielsweise, dass die Auswertung des Leistungsmaßes `UTILIZATION` aufwendiger ist als die der anderen Leistungsmaße, also mehr CPU-Zeit verbraucht. Das Abbruchkriterium `CPUTIME X` ist genau dann erfüllt, wenn seit dem Start der Analyse mehr als `X` Zeiteinheiten vergangen sind.
- `MODELTIME` beschreibt die Zeit, die "innerhalb des Modells" voranschreitet. Sie ist (als Abbruchbedingung) nur für die Simulation zulässig. Man definiert eine Zeiteinheit in Modellzeit als verstrichen, wenn an einer atomaren Komponente des Leistungsmodells (also einem HIT-Server) eine Aufgabe zur Bearbeitung ankommt oder wenn ein Prozesserzeugungs-Statement (`CREATE ... AFTER / EVERY / AT ...`) ausgeführt wird. Die Modellzeit beginnt mit dem Wert 0.0 bei Start der Analyse. Eine Prämisse `MODELTIME X` ist erfüllt, falls `X` Zeiteinheiten in Modellzeit vergangen sind und eines der genannten Ereignisse auftritt.
- Die `ACCURACY` hat für die Lösungsmethoden `ANALYTICAL MARKOV` und `ANALYTICAL LIN2` unterschiedliche Bedeutungen. Beim Markov'schen Ansatz wird sie interpretiert als der geschätzte relative Fehler der Lösung (in Prozent). Als Stop-Bedingung für den Markov-Löser sollte für die `ACCURACY` sinnvollerweise ein Wert kleiner oder gleich 0.1 gewählt werden (obwohl auch höhere Vorgaben erlaubt sind). Die Stop-Bedingung ist erfüllt, falls der aktuelle Fehler kleiner oder gleich der Vorgabe ist. Für den Löser LIN2 definiert die Accuracy die Güte des Estimators `BOUNDS`. Je höher der Wert, desto genauer wird die Rechnung ausgeführt (und desto länger dauert die Analyse!). Es wird empfohlen, einen ganzzahligen Wert anzugeben; jeder Gleitkommawert wird kaufmännisch gerundet. Für extrem große Modelle wird die `ACCURACY` automatisch reduziert und eine Warnung ausgegeben. Die Rechnung wird gestoppt, falls die Güte des Estimators `BOUNDS` größer oder gleich der Vorgabe ist. Die `ACCURACY` kann ausschließlich in der GUI definiert werden, da sie nur als globale Stop-Bedingung eingesetzt werden kann.
- Der `CONFIDENCE LEVEL` charakterisiert die Zuverlässigkeit der erhobenen Messwerte. Er ist wie `MODELTIME` nur für die Simulation zulässig. Hier basieren sämtliche Werte der Leistungsmaße auf Schätzungen, so dass deren Nutzen direkt abhängig ist von der ebenfalls geschätzten Genauigkeit. Mit anderen Worten: es wird ein Mittelwert eines Leistungsmaßes anhand statistischer Verfahren berechnet und daran die Abweichung der einzelnen Werte gemessen. Liegt ein bestimmter Prozentsatz der Werte "in der Nähe" des Mittelwerts, so wird diese Ziffer als Confidence Level ("Zuverlässigkeitsniveau") verwendet. Die "Nähe" zum Mittelwert wird dabei durch einen weiteren Prozentwert hinter dem Schlüsselwort `WIDTH` angegeben. Für die Berechnung des Mittelwertes wird der Estimator `MEAN` verwendet. Ein Beispiel: die Angabe `CONFIDENCE LEVEL 93 WIDTH 5` als Stop-Bedingung besagt, dass die Simulation abgebrochen werden soll, falls 93 Prozent aller bisher ermittelten Werte um höchstens 5 Prozent vom Mittelwert `MEAN` abweichen. Zu beachten ist,

dass für den Confidence Level nur Werte zwischen 90 und 99 Prozent akzeptiert werden. Abweichungen nach unten oder nach oben werden automatisch korrigiert, indem der nächstliegende gültige Wert eingesetzt wird. Begründet ist diese Beschränkung in der Gefahr, entweder nutzlose Ergebnisse oder eine extrem hohe Laufzeit zu erhalten. Eine weitere Besonderheit des Confidence Levels ist, dass er für jedes Leistungsmaß gesondert definiert werden muß, das in der Notiz per `GET`- oder `REQUIRED`-Statement angefordert wird (s.u.). Dies macht deshalb Sinn, weil `ACCURACY`, `CPUTIME` oder `MODELTIME` globale Variablen sind, der Confidence Level hingegen für jedes Leistungsmaß einzeln erhoben wird.

Für jedes Diagrammelement, an dem ein `GET`- oder `REQUIRED`-Statement gefunden wird, erzeugt HIPE ein Evaluationsobjekt. Bei der Ausführung des Experiments auf dem Leistungsmodell wird die Analyse in Abhängigkeit von den jeweiligen Stop-Bedingungen beendet. Um die Informationen zur Verfügung zu stellen, die für die Generierung von gültigen Abbruchbedingungen benötigt werden, muss das `<StopCondition_Stmt>` verwendet werden. Es muss für eine simulative Lösung zwangsweise in jeder Notiz vorhanden sein, in der mindestens ein `GET`- oder `REQUIRED`-Statement steht. Ist diese Bedingung nicht erfüllt, werden die in der GUI gemachten Angaben verwendet. Im Falle mehrerer Stop-Bedingungen in einer einzigen Notiz werden die Statements durch den logischen Operator "OR" (engl. für "ODER") verknüpft. Dass heißt, nur eine der Bedingungen muss erfüllt sein, um einen Abbruch der Auswertung herbeizuführen. Dabei werden stets nur die für die angewandte Lösungsmethode gültigen Stop-Bedingungen übernommen, eventuell überflüssige Statements werden ignoriert. Als Beispiel für die verwendete Syntax betrachte man folgende Statements:

```
stop responsetime confidencelevel default width 5;
stop utilization default;
stop default;
```

Für die `responsetime` wird hier das entsprechende `GET`- oder `REQUIRED`-Statement in der Notiz vorausgesetzt. Das Schlüsselwort `default` kann verwendet werden, um die in der GUI angegebenen Belegungen einsetzen zu lassen. Dies ist sowohl für einzelne Werte (wie z.B. in den ersten beiden Statements zu sehen) als auch für komplette Stop-Bedingungen erlaubt (letztes Statement). Im zweiten Fall werden alle anderen eventuell vorhandenen Stop-Statements einer Notiz ignoriert. Das bedeutet, dass insbesondere für jedes (!) mittels `GET`- oder `REQUIRED` angeforderte Leistungsmaß das in der GUI definierte Confidencelevel und dessen Breite als Stop-Bedigung formuliert wird. Vorsicht sollte man bei der Verwendung der `ACCURACY` walten lassen, denn sie hat bei Verwendung des Markov- und des LIN2-Algorithmus' gegensätzliche Bedeutungen (s.o.). Eine Angabe von `0.075` für die `ACCURACY` in der GUI würde zwar für den Markov-Algorithmus ein sinnvolles Abbruchkriterium definieren (nämlich eine obere Fehlerschranke von 7,5%). Der LIN2-Algorithmus würde mit dieser Belegung jedoch bereits bei noch völlig unbrauchbaren Ergebnissen die Arbeit einstellen. (Die Details zur Umsetzung der Experimente in HI-SLANG werden in Abschnitt 1.2.6 beschrieben. Für weiterführende Informationen zu HIT und den hier beschriebenen Lösern und Abbruchbedingungen sei auf [Büttner u. a. (1999)] verwiesen.)

1.2.4.5 Grammatik der HIPE-Statements

Für die Annotationen gilt die folgende Grammatik in Backus-Naur-Form, wobei sämtliche Statements nicht case-sensitive sind. Man beachte, dass das Schlüsselwort `.HIPE` zur Markie-

rung des für HIPE relevanten Teils einer Annotation als erstes Zeichen einen Punkt enthält, dass Listenelemente durch das Zeichen '&' ("Kaufmanns-UND") getrennt werden und dass bestimmte Statements am Zeilenende kein Semikolon sondern einen expliziten Zeilenumbruch erwarten (diese HIPE-Statements setzen sich also aus einzelnen mit dem Schlüsselwort LET beginnenden Zeilen zusammen)! Letzterer wird im Folgenden mit '\n' kenntlich gemacht.

- Allgemeine Statements:

```
<HIPE_Block> ::= .HIPE \n {<HIPE_Stmt>; \n}

<HIPE_Stmt> ::= <UseCaseDiagram_Stmt>
              | <ActivityDiagram_Stmt>
              | <DeploymentDiagram_Stmt>
              | <StandardHardware_Stmt>
<Relation> ::= < | <= | = | >= | >
<ProbValue> ::= 0.<DigitList> | 1.0
<Numerical> ::= <DigitList> [. <DigitList>]
<DigitList> ::= <Digit>{<Digit>}
<Digit> ::= 0 | 1 | 2 | .. | 9
<Array> ::= "["<NumericalVector>"]"
<NumericalVector> ::= <Numerical> [, <Numerical>]
<NumericalList> ::= <Numerical> [& <NumericalList>]
<ArrivalTimeList> ::= <Numerical> [, <ArrivalTimeList>]
<Distribution> ::= <Numerical> | negexp(<Numerical>)
                               | cox(<Numerical>, <Numerical>)
                               | coxg("["<Array>, <Array>"]")
                               | erlang(<Numerical>, <Numerical>)
                               | normal(<Numerical>, <Numerical>)
                               | poisson(<Numerical>)
                               | randint(<Numerical>, <Numerical>)
                               | uniform(<Numerical>, <Numerical>)
                               | linear(<Array>, <Array>)
<DistributionList> ::= <Distribution> [& <DistributionList>]
<Measure> ::= responsetime | throughput
            | utilization | occupation
            | population
<Get_Stmt> ::= GET <Measure>
<Required_Stmt> ::= REQUIRED <Measure> <Relation> <Numerical>
```

- Statements für Elemente in UseCase-Diagrammen:

```
<UseCaseDiagram_Stmt> ::= <Actor_Stmt>
                        | <Connection_Stmt>
                        | <UseCase_Stmt>

<Actor_Stmt> ::= <ActorLet_Stmt>
               | <ActorGet_Stmt>
```

```
                | <ActorRequired_Stmt>
<ActorLet_Stmt> ::=  <Load_Stmt> | <StopCondition_Stmt>

<Load_Stmt> ::= <OpenLoad_Stmt> | <ClosedLoad_Stmt>
<OpenLoad_Stmt> ::= LET load = openload \n <OpenLoadParameter_Stmt>
<OpenLoadParameter_Stmt> ::= LET occurence = <DistributionList>
                           | LET arrivaltime = <ArrivalTimeList>
<ClosedLoad_Stmt> ::= LET load = closedload \n <ClosedLoadParameter_Stmt>
<ClosedLoadParameter_Stmt> ::= LET population = <NumericalList> \n
                               LET extDelay = <DistributionList>
                             | LET arrivaltime = <ArrivalTimeList>
<StopCondition_Stmt> ::= stop <Condition_Stmt>
<Condition_Stmt> ::= <SimulationStopCondition_Stmt>
                   | default
<SimulationStopCondition_Stmt> ::= <Measure> <ConfLevel_Stmt>
                                 | <SimpleStopCondition_Stmt>
<SimpleStopCondition_Stmt> ::= <Cputime_Stmt>
                             | <Modeltime_Stmt>
<ConfLevel_Stmt> ::= confidencelevel <Numerical_Stmt> width <Numerical_Stmt>
                   | default
<Cputime_Stmt> ::= cputime <Numerical_Stmt>
<Modeltime_Stmt> ::= modeltime <Numerical_Stmt>
<Numerical_Stmt> ::= <Numerical> | default

<ActorGet_Stmt> ::= GET responsetime
<ActorRequired_Stmt> ::= REQUIRED responsetime <Relation> <Numerical>

<Connection_Stmt> ::= LET Prob = <ProbValue>

<UseCase_Stmt> ::= {<UseCaseLet_Stmt> | <Get_Stmt> | <Required_Stmt>}
<UseCaseLet_Stmt> ::= <UseCaseDist_Stmt> | <UseCaseString_Stmt>
<UseCaseDist_Stmt> ::= LET <UseCaseDistParameter> = <DistributionList>
<UseCaseDistParameter> ::= responsetime
<UseCaseString_Stmt> ::= LET <UseCaseStringParameter> = <String>
<UseCaseStringParameter> ::= diagram | cpu | resource
```

- Statements für Elemente in Aktivitäts-Diagrammen:

```
<ActivityDiagram_Stmt> ::= <Activity_Stmt> | <Assoc_Stmt>

<Activity_Stmt> ::= {<ActivityLet_Stmt> | <Get_Stmt> | <Required_Stmt>}
<ActivityLet_Stmt> ::= <ActivityDist_Stmt> | <ActivityString_Stmt>
                     | <ActivityNum_Stmt>
<ActivityDist_Stmt> ::= LET <ActivityDistParameter> = <DistributionList>
<ActivityDistParameter> ::= responsetime
<ActivityString_Stmt> ::= LET <ActivityStringParameter> = <String>
```

```
<ActivityStringParameter> ::= diagram | cpu | resource | class | method
<ActivityNum_Stmt> ::= LET <ActivityNumericalParameter> = <Numerical>
<ActivityNumericalParameter> ::= resourceusage

<Assoc_Stmt> ::= <Loop_Stmt> | <Prob_Stmt>
<Loop_Stmt> ::= LET <LoopParameter> = <Numerical>
<LoopParameter> ::= numrepeats | averagerepeats
<Prob_Stmt> ::= LET prob = <ProbValue>
```

- Statements für Elemente in Deployment-Diagrammen:

```
<DeploymentDiagram_Stmt> ::= <Hardware_Stmt> | <Network_Stmt>

<Hardware_Stmt> ::= <Cpu_Stmt> | <Resource_Stmt>
<Cpu_Stmt> ::= <CpuGet_Stmt> | <CpuLet_Stmt> {\n <Cpu_Stmt>}
<CpuGet_Stmt> ::= GET <Measure>
<CpuLet_Stmt> ::= <CpuLetNum_Stmt> | <CpuLetString_Stmt>
<CpuLetNum_Stmt> ::= LET <CpuNumericalParameter> = <NumericalList>
<CpuNumericalParameter> ::= speed | memory
<CpuLetString_Stmt> ::= LET <CpuStringParameter> = <CpuStringValue>
<CpuStringParameter> ::= schedule | dispatch
<CpuStringValue> ::= <CpuStringScheduleValue> | <CpuStringDispatchValue>
<CpuStringScheduleValue> ::= immediate | fcfs | random
<CpuStringDispatchValue> ::= shared | equal

<Resource_Stmt> ::= <ResourceGet_Stmt> | <ResourceLet_Stmt> {\n <Resource_Stmt>}
<ResourceGet_Stmt> ::= GET <Measure>
<ResourceLet_Stmt> ::= LET <ResourceNumericalParameter> = <Numerical>
<ResourceNumericalParameter> ::= speed | accessTime | size

<Network_Stmt> ::= <NetworkGet_Stmt> | <NetworkLet_Stmt> {\n <Network_Stmt>}
<NetworkGet_Stmt> ::= GET <Measure>
<NetworkLet_Stmt> ::= <NetworkLetNum_Stmt> | <NetworkLetString_Stmt>

<NetworkLetNum_Stmt> ::= LET <NetworkNumericalParameter> = <Numerical>
<NetworkNumericalParameter> ::= speed | minPacketSize
                              | maxPacketSize | responsetime

<NetworkLetString_Stmt> ::= LET <NetworkStringParameter> = <DistributionList>
<NetworkStringParameter> ::= error
```

- Statements für eine diagrammweit gültige Notiz zur Definition einer aktiven Default-Hardware im Deployment-Diagramm:

```
<StandardHardware_Stmt> ::= LET defaultServer = <String>
```

1.2.5 Verwendung der weichen Analyse (Berechnung der Objektfluss-Last)

An Aktivitäten können zur Verfeinerung der Modellierung Objektflüsse annotiert werden, die zum einen Aktivitäten mit ihrem Implementierungskonzept in Verbindung setzen und zum anderen den Speicherverwaltungs- und Rechenaufwand zur Laufzeit andeuten. Letzteres ist für HIPE besonders interessant aufgrund des frühen Status der zu analysierenden Modellierung (d.h. es ist noch keine Implementierung vorhanden). Objektflüsse repräsentieren dabei Objekte, die bei der Ausführung der mit ihnen annotierten Aktivitäten manipuliert, erstellt oder verwendet, in jedem Fall also verwaltet werden müssen. Die jeweils zugrundeliegende Klasse des Klassendiagramms kann daraufhin analysiert werden, um die bei der Verwaltung entstehende Last zu approximieren. Diese dient wiederum dazu, der Aktivität bei ihrer Übersetzung nach HI-SLANG einen Wert zuzuordnen, der die Anzahl an Operationen repräsentiert, die von der angegebenen oder standardmäßig angenommenen Hardwarekomponente für die Bearbeitung der Aktivität auszuführen sind.

Im folgenden Abschnitt wird dargestellt, wie aus einem Klassendiagramm mittels der *weichen Analyse* die Komplexität von Klassen (insbesondere von Methodenaufrufen und Rückgabewerten) berechnet wird. Ausgangspunkt der Betrachtungen ist die Signatur einer Klasse, d.h. ihre Definition im Klassendiagramm. Anhand des Ansatzes in [Carbone und Santucci (2002)] zur Berechnung der erwarteten Anzahl von Codezeilen wird die Komplexität der betrachteten Klasse in Abhängigkeit der zur Verfügung stehenden Informationen abgeschätzt. Aus dem Klassendiagramm lassen sich folgende für diese Analyse relevanten Charakteristika ermitteln:

- Typ und Sichtbarkeit von Attributen
- Rückgabetyp der Methoden
- Anzahl an Methoden einer Klasse c (*numMeth(c)*) sowie deren Parameter(-Typen)
- Anzahl der mit einer Klasse c assoziierten Elemente (*numAss(c)*)

Aufgrund dieser Angaben wurde in [Carbone und Santucci (2002)] folgende Formel für die erwartete Anzahl an Codezeilen hergeleitet:

$$LOC(c) \;=\; 4 + 10CP(c)^{0,7}.$$

Dabei ist

$$\begin{aligned} CP(c) &= 2SP(c) + 3BP(c) \\ &= 2\underbrace{(numLA(c) + 3numHA(c) + 5numIA(c))}_{=SP(c)} + 3\underbrace{\left([1 + numAss(c)] \sum_{i=1}^{numMeth(c)} CM(m_i)\right)}_{=BP(c)} \end{aligned}$$

und

$$CM(m_i) \;=\; \begin{pmatrix} T & S \end{pmatrix}_i \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 5 \end{pmatrix} \begin{pmatrix} numLA(m_i) \\ numHA(m_i) \\ numIA(m_i) \end{pmatrix}$$

sowie

$$\left(\begin{array}{cc} T & S \end{array} \right)_i = \begin{cases} (1,0) & \Leftrightarrow \frac{numLA(m_i)}{numLA(m_i)+numHA(m_i)} > 0.8 \wedge numIA(m_i) = 0 \\ (0,1) & \text{sonst} \end{cases}$$

Dabei beschreiben *numLA(.)*, *numHA(.)* und *numIA(.)* für Klassen und Methoden jeweils die Quantitäten der verwendeten Attribut-Typen. Diese sind:

- *lightAttributes* (Attribute primitiven Datentyps)
- *heavyAttributes* (Attribute, deren Typen aus einer bereits getesteten bzw. optimierten Bibliothek stammen)
- *importedAttributes* (Attribute eines selbstdefinierten Typs)

In die erste Kategorie fallen beispielsweise Attribute der Typen Integer, String, Boolean, Character, Long, Double usw. (nicht jedoch deren gleichnamige Wrapper-Klassen, wie z.B. in der Java-Bibliothek existieren). Als Beispiel für die Menge der *heavyAttributes* können alle Klassen aus der Java-Standardbibliothek dienen (also Canvas, FileOutputStream, Vector, Array, usw). Schließlich werden Attribute als *importedAttributes* bezeichnet, wenn sie noch keiner Bewertung bezüglich ihrer Effizienz unterzogen wurden und somit gravierende Auswirkungen auf die Komplexität des umgebenden Programms haben können.
Die dargestellte Formel dient ursprünglich der Abschätzung der zu erwartenden Komplexität der Klasse bezüglich ihrer Codezeilen. Für das Ziel der Klassenanalyse im Rahmen von HIPE ist diese Angabe allerdings nicht von Interesse, weshalb im Folgenden nur auf die *Complexity Points* $CP(c)$ einer Klasse c und deren Bestandteile eingegangen wird.

Die Rechnung beschreibt eine Abschätzung der Klassenkomplexität als Resultat einer Summe von gewichteten Attributen- und Methodenkomplexitäten, dargestellt durch $SP(c)$ und $BP(c)$. $SP(c)$ steht dabei für *State Points* der Klasse c, $BP(c)$ bezeichnet ihre *Behavioral Points*. Für den Wert $SP(c)$ werden die Kardinalitäten der Mengen der *lightAttributes*, *heavyAttributes* und *importedAttributes* gewichtet aufsummiert. Es sei an dieser Stelle vermerkt, dass sämtliche in der Formel verwendeten Gewichte (und Korrekturfaktoren) aus [Carbone und Santucci (2002)] entnommen und dort empirisch ermittelt und getestet worden sind. $BP(c)$ setzt sich zusammen aus dem Produkt der (um 1 erhöhten) Anzahl der Assoziationen, in die die betrachtete Klasse involviert ist, und der Summe der Komplexitäten ihrer Methoden. Letzteres wird ermittelt durch die Auswahl eines Gewichtsvektors und dessen Multiplikation mit dem Vektor, der für die Parameterliste einer Methode die Kardinalitäten der drei betrachteten Attributgattungen enthält. Die Auswahl des Gewichtsvektors geschieht in Abhängigkeit der Charakterisierung der betrachteten Methode als *substantiell* $[(T,S)_i = (0,1)]$ bzw. *trivial* $[(T,S)_i = (1,0)]$. Eine Methode wird als *trivial* bezeichnet, falls in ihrer Signatur mehr als 80% der Parameter zur Menge der *lightAttributes* gehören und darin keine *importedAttributes* angegeben werden, sonst als *substantiell*.

Zur Genauigkeit der Abschätzung ist zu sagen, dass in [Carbone und Santucci (2002)] ein Beispielprojekt sowohl mit obiger Formel analysiert als auch zur Kontrolle konkret implementiert wurde. Die Abschätzung des Codeumfangs $LOC(c)$ lag hierbei um 9,5% unter der tatsächlichen Anzahl der Codezeilen. Man sieht also, dass zumindest in einzelnen Fallbeispielen die

Abschätzung (des Codeumfangs) erstaunlich nah am realen Wert liegt, wobei dieser natürlich von der Qualität der Implementierung (also von den Fähigkeiten des Programmierers) abhängig ist und somit nur selten als Referenzkriterium herangezogen werden kann.

Die mit der obigen Formel für $CP(c)$ vorgenommene Analyse einer Klasse gehört zwar zur *weichen Analyse*, ihr Resultat wird jedoch zusätzlich in der *harten Analyse* eingesetzt. Die Untersuchung wird in HIPE im Kontext des Objektflusses in Aktivitätsdiagrammen stattfinden. Das Ergebnis soll einen Anhaltspunkt für die zu erwartende Last an der entsprechenden Aktivität ergeben. Diese Last wird benötigt, um das Ergebnis der harten Analyse mittels HIT möglichst gut an eine Leistungserhebung des fertiggestellten Programms anzunähern. Um eine Beziehung zwischen den Complexity Points einer Klasse c und der von c verursachten Last herzustellen, wurde an konkreten Beispielklassen (bzw. ihren Methoden) untersucht, wie sich die *Complexity of Methods* $CM(m_i)$ zur Länge des Bytecodes der Methode m_i verhält. Unter der Annahme, dass die primitivste Anweisung im Bytecode die kleinste mögliche Anweisungslänge (8 Bit) besitzt und dass diese Anweisung genau eine Operation im Last-Maschine-Konzept repräsentiert, ergab sich für die Schätzung der von m_i erzeugten Last der Korrekturfaktor 10, d.h. $Last(m_i) = 10CM(m_i)$. Hier muss deutlich betont werden, dass dieser Faktor zwar durch sorgfältige Untersuchungen der Beispiele ermittelt wurde, aber keine universelle Gültigkeit besitzt. Es sind Fälle aufgetreten, bei denen Methoden nach der Rechnung zwar eine sehr geringe Komplexität hatten, aber dennoch in einer ungleich längeren Anweisungsfolge im Bytecode resultierten. Auch hier wurde deutlich, dass die Geschicklichkeit des Programmierers die kritischste Unbekannte in der Komplexitätsabschätzung darstellt. Außerdem sei hier erwähnt, dass nach weiteren Untersuchungen — vergleiche dazu [Antonioli und Pilz (1998)] — der Anteil an "echten" Operationen im Bytecode[6] gerade einmal 12% beträgt, auf den Rest entfällt die Repräsentation von Konstanten, Text usw. Damit lassen sich also die beschriebenen, in Einzelfällen gravierenden Abweichungen von den berechneten Komplexitätswerten erklären, denn schon eine ausufernde Verwendung von Code-Dokumentationen lässt die Länge des Bytecodes bei unveränderter Komplexität stark ansteigen.

Informationen aus Anwendungsfalldiagrammen

Eine Verfeinerung der Abschätzung kann durch Hinzunahme von Informationen aus Anwendungsfalldiagrammen geschehen. Dazu muss zunächst jedem Akteur und jedem Anwendungsfall ein Komplexitätswert zugeordnet werden. Dann kann $CP(c)$ abhängig von den Komplexitäten der mit c assoziierten Anwendungsfälle und Akteure mit einem Korrekturparameter versehen werden. Mit Hilfe der Beziehung

$$COUC(uc) = 1 + \sum_{i=1}^{numScen(uc)} numMess_i$$

wird dazu jedem Anwendungsfall uc seine Komplexität $COUC(uc)$ zugewiesen, wobei die obere Summationsgrenze $numScen(uc)$ die Anzahl der Szenarien bezeichnet, mit denen der

[6] Bytecode ist eine Sammlung von Befehlen für eine virtuelle Maschine. Bei der Compilierung eines Quelltextes mancher Programmiersprachen oder Umgebungen, wie z. B. Java, wird nicht direkt Maschinencode sondern als Zwischenprodukt zunächst Bytecode erstellt. Dieser Code ist in der Regel maschinenunabhängig und im Vergleich zum Quelltext und zu Maschinencode oft relativ kompakt. Die virtuelle Maschine führt dann dieses Zwischenergebnis aus, indem sie den Bytecode in Maschinencode für den jeweiligen Prozessor zur Laufzeit übersetzt.

Anwendungsfall assoziiert ist (z.B. Interaktionsdiagramme). $numMess_i$ ist die Anzahl aller Nachrichten zwischen allen Objekten des i-ten Szenarios. Ein Anwendungsfall ohne Szenarioassoziationen hat somit die Komplexität 1.

Für Akteure a ist die Komplexität $COA(a)$ definiert als

$$COA(a) = numAss(a) * dit(a),$$

wobei $numAss(a)$ die Anzahl aller Assoziationen mit a als Anfangs- oder Endknoten und $dit(a)$ die Tiefe von a im Vererbungsbaum[7] repräsentiert. (Ist a ein Subakteur in einer Hierarchie, so wird zu $numAss$ zusätzlich die Summe aller Assoziationen addiert, die an den Elternakteuren von a beginnen oder enden.) Schließlich sei ein Akteur a mit einer Klasse c assoziiert, falls es einen mit c assoziierten Anwendungsfall gibt, der mit a kommuniziert.

Mit Hilfe dieser Definitionen kann nun die Komplexität eines mit c assoziierten Anwendungsfalls $CUCA(c)$ und die Komplexität eines mit c assoziierten Akteurs $CAA(c)$ angegeben werden als

$$CUCA(c) = \sum_{i=1}^{numUC(c)} COUC(uc_i)$$

und

$$CAA(c) = \sum_{i=1}^{numAct(c)} COA(a_i).$$

Dabei bezeichnet $numUC(c)$ die Anzahl der mit c assoziierten Anwendungsfälle und $numAct(c)$ die Anzahl der mit c assoziierten Akteure. Aus den Werten von $CUCA(c)$ und $CAA(c)$ ergeben sich laut [Carbone und Santucci (2002)] die in Tabelle 1.3 gezeigten Korrekturfaktoren für $CP(c)$. Ist beispielsweise $CUCA(c) = 7$ und $CAA(c) = 15$, so folgt

$$CP(c)_{neu} = (1 + 0.09)CP(c)_{alt}.$$

CUCA(c)	0-3	4-6	7-9	10-13	>13
CAA(c)					
0-4	-10%	+3%	+5%	+7%	+20%
5-10	-5%	+2%	+5%	+10%	+25%
10-20	+6%	+7%	+9%	+25%	+30%

Tabelle 1.3: Anwendungsfall-Korrekturfaktoren für CP(c)

Informationen aus Interaktionsdiagrammen

Weiterhin können Informationen aus Interaktionsdiagrammen gezogen werden. $numMess(c, d_j)$ sei die Anzahl von Nachrichten, die von Instanzen der Klasse c im Interaktionsdiagramm d_j

[7] HIPE kann derzeit keine hierarchischen Anwendungsfalldiagramme verarbeiten. Siehe dazu auch Abschnitt 5.7.2, Seite 208.

gesendet oder empfangen werden. Sei $totMess(c) = \sum_{j=1}^{nid} numMess(c, d_j)$ die Summe aller im Modell des Systems von c gesendeten oder empfangenen Nachrichten, und $rsed(c)$ der prozentuale Anteil der Interaktionsdiagramme, in denen c enthalten ist. $totMess(c)$ liefert eine Angabe darüber, wie viel die Klasse c kommuniziert, und $rsed(c)$ entspricht dem Prozentsatz des gesamten Codes im System, der mit der Klasse c kommuniziert. Die Tabelle 1.4 liefert die Anpassungsfaktoren für CP(c) auf Grundlage von $totMess(c)$ und $rsed(c)$.

totMess(c)	0-3	4-6	7-9	10-13	>13
rsed(c)					
0-19%	-10%	-5%	+5%	+7%	+12%
20-49%	-5%	+2%	+5%	+10%	+13%
50-100%	+4%	+5%	+7%	+10%	+15%

Tabelle 1.4: Korrekturfaktoren für CP(c) auf Basis von Interaktionsdiagrammen

Informationen aus Zustandsdiagrammen[8]

Ein Zustandsdiagramm kann entweder mit einer Klasse c oder einer Methode von c assoziiert sein. Es ist davon auszugehen, daß das Verhalten einer Klasse oder Methode umso komplexer sein wird, je komplexer das entprechende Zustandsdiagramm ist. $numSta(std)$ gebe die Anzahl der Zustände im Zustandsdiagramm std an, $numAction(std)$ die Anzahl der Aktionen, die beim Betreten oder Verlassen von Zuständen ausgeführt werden. Sei $totSta(c)$ die Summe der $numSta(.)$-Werte über alle Zustandsdiagramme, die sich auf Elemente der Klasse c beziehen, $totAction(c)$ entsprechend die Summe der $numAction(.)$-Werte. CP(c) kann dann auf Basis der Faktoren in der Tabelle 1.5 noch einmal angepaßt werden.

totSta(c)	0-3	4-6	7-9	10-13	>13
totAction(c)					
0-5	-8%	-3%	+3%	+7%	+12%
6-10	-3%	+2%	+4%	+10%	+13%
11-20	+3%	+4%	+7%	+12%	+15%

Tabelle 1.5: Korrekturfaktoren für CP(c) auf Basis von Zustandsdiagrammen

Vererbungshierarchie im Klassendiagramm

Als weitere Korrektur sollte aufgrund der großen Bedeutung der Vererbung im objektorientierten Programmierparadigma stets $CP(superclass(c))$ als weiterer Summand zum endgültigen Wert von $CP(c)$ hinzuaddiert werden, damit die geerbte Komplexität ebenfalls in die Betrachtungen einbezogen werden kann.

Aussagekraft dieser Komplexitätsmaße

[8] Zustandsdiagramme sind derzeit nicht im HIPE-Szenario enthalten.

Die in der Anforderungsdefinition für HIPE (in [PG459 (2005)]) beschriebenen weichen Maße sind teilweise bereits in der Formel enthalten. Namentlich sind dies der WAC- und der WMC-Index. Sie wurden beschrieben als die Summe der Komplexitäten der Attribute bzw. der Methoden einer Klasse c und sind gerade die Werte SP(c) respektive BP(c). Für den WMC-Index wurde nachgewiesen, daß dieser stark mit der Speichergröße einer Klasse korrelliert ist und somit eine sinnvolle Abschätzung für die relative Größenordnung der Komplexität einer Klasse liefert (vgl. dazu [Emam u. a. (1999)]).

1.2.6 Diagramme im HIPE-Szenario

In diesem Abschnitt wird beschrieben, auf welche Art und Weise Diagramme in HIPE berücksichtigt werden, welche Angaben entsprechend in ihnen gemacht und aus ihnen gezogen werden können. Für alle in eine Bewertung einbezogenen Elemente der verwendeten Diagramme werden die möglichen Parameterwerte diskutiert, mit denen diese Elemente annotiert werden können. Schließlich wird beschrieben, in welcher Form diese Angaben in das vom Bewertungstool HIT verwendete HI-SLANG umgesetzt werden.

Im Folgenden werden die Komponenten eines syntaktisch korrekten HIT-Modells betrachtet, wie es in HI-SLANG dargestellt wird, und es wird diskutiert, auf welche Weise Elemente von UML-Modellspezifikationen durch HIPE in HI-SLANG dargestellt werden.

Ein valides HIT-Modell enthält zumindest genau ein Element vom TYPE MODEL und eine EXPERIMENT-Definition.

Definition eines MODEL-Blocks

Der MODEL-Block definiert die oberste Schicht eines HIT-Modells. Hierin wird die Arbeitslast des Modells definiert und zu den Abläufen der höchsten Abstraktionsstufe in Beziehung gesetzt. Dies geschieht, indem einerseits USED SERVICEs definiert werden, die von dem modellierten System zu bearbeitende Typen von Anfragen darstellen, und andererseits Komponenten verwiesen und beschrieben werden, die als Maschinen diese Anfragen bearbeiten sollen. Diese Komponenten können dabei Hardwareeinheiten, d.h. Prozessoren und ähnliche Ressourcen, auf denen eine Software ausgeführt wird, oder Softwarekomponenten, d.h. Abläufe darstellen, die durch einen Aufruf angestoßen werden. Komponenten im Sinne von HIT bieten selbst PROVIDEd SERVICEs an, über die sie angesprochen werden können. Mit Hilfe eines REFER-Blocks werden die USED SERVICEs mit den PROVIDEd SERVICEs in Beziehung gesetzt und dadurch definiert, welche Last von welcher Maschine bearbeitet werden soll. Der MODEL-Block hat die folgende Gestalt:

```
TYPE <Modellname> MODEL(<Parameterliste des Modells>);

  TYPE <Servicename> SERVICE (<Parameterliste der Services>);
    USE SERVICE
      <UsedServicename>(<Parameterliste des Used Service>);
      [ggf. weitere Definitionen von Used Services]
    END USE;
  BEGIN
```

```
    <Statementblock des Services>
  END TYPE <Servicename>;

  [ggf. weitere Definitionen von Services]

  COMPONENT <Komponentenname>:<Komponententyp>(<Parameterliste des Komponententyps>);

  [ggf. weitere Definitionen von Komponenten]

  REFER <Liste von Services> TO <Liste von Komponenten>
  EQUATING
    <Servicename>.<UsedServicename> TO <Komponentenname>.<ProvidedServiceName>;
    [ggf. weitere Definitionen von Referenzen]
  END REFER;

BEGIN
  <Statementblock des Modells>
  CREATE <Anzahl Prozesse> PROCESS <Prozessname> <Zeitmodifikator>;
  [ggf. weitere Arbeitslastdefinitionen]
END TYPE <Modellname>;
```

Implementierung von Aufrufen
Bei der Implementierung einzelner Used Services und Arbeitslastdefinitionen wird versucht, möglichst viele Angaben aus dem Softwaremodell selbst zu gewinnen. Wo dies nicht möglich ist, werden Default-Werte eingesetzt, die vorab vom Nutzer definiert wurden. Diese Default-Werte beziehen sich zum einen auf die Maschine, auf der Anfragen ausgeführt werden sollen, zum anderen auf den Umfang, in dem eine Maschine durch eine Anfrage in Anspruch genommen wird:

- Im Abschnitt 1.2.4.3 wurde bereits die Annotation `defaultserver` erwähnt, mit der die Default-CPU festgelegt werden kann. Kann aus der Annotation eines Anwendungsfalls oder einer Aktivität nicht ermittelt werden, auf welcher CPU die Anfrage ausgeführt werden soll, die die entsprechende Ausführung darstellt, so wird an der entsprechenden Stelle ein Aufruf der Default-CPU erstellt.
- Wird einer Aktivität keine Antwortzeit zugeordnet, so wird eine vom Nutzer vorgegebene Antwortzeit angenommen. Deren Definition erfolgt nicht innerhalb des Modells selbst, sondern als zusätzliche Einstellung im Rahmen der Konfiguration von HIPE.

Definition eines EXPERIMENT-Blocks
Das Ziel der Übersetzung des Eingabe-UML-Modells bzw. des dadurch definierten Softwaresystems ist die Durchführung einer Analyse dieses Systems mit Hilfe des Leistungsbewertungstools HIT. Dieses ermöglicht die hierarchische Modellierung des Systems und durch diverse Parameter definierte "Ausführungen" des hierarchischen Modells. Letzteres wird als "Experiment" bezeichnet und umfasst die Definition bestimmter Ausführungsparameter und der Modellkomponenten, an denen bestimmte Messungen durchgeführt werden sollen. (Eine detaillierte Beschreibung von HIT findet sich im Zwischenbericht der PG459, vgl. [PG459 (2005)].) Durch die Annotation mit GET- und REQUIRED-Statements teilt der Nutzer HIPE mit, dass

und an welchen Modellkomponenten Maße erhoben werden sollen. STOP-Statements geben an, unter welchen Bedingungen die Auswertung eines Modells abgebrochen werden soll.
Die Umsetzung dieser Angaben in HI-SLANG erfolgt in Form eines Experimentblocks. Dieser definiert, welche Maße von HIT für welche Komponenten des Modells erhoben werden und welcher Löser und welche STOP-Kriterien dabei verwendet werden sollen. Er hat die folgende Gestalt:

```
EXPERIMENT <Experimentname> METHOD <Auswertungsmethode>;
BEGIN
  EVALUATE MODEL hipemodel : <Modellname>(<Parameterliste des Modells>);
    EVALUATIONOBJECT
      <Evaluationsobjektname> VIA <Modellobjektname>;
      [ggf. weitere Definitionen von Referenzen]
    BEGIN
      MEASURE <Leistungsmaßliste> AT <Evaluationsobjektname>;
      [ggf. weitere Definitionen von Messungen und der CONTROL-Block]
    END EVALUATE;
END EXPERIMENT <Experimentname>;
```

Ein `GET`- oder `REQUIRED`-Statement führt dazu, dass ein neues Evaluationsobjekt generiert wird, an dem von HIT die gewünschten Maße erhoben werden. Soll also an einer Komponente `c`, die mittels COMPONENT oder ENCLOSED in einem Modell oder einer selbstdefinierten Komponente mit dem Namen `h` erfasst wird, eine Liste von Maßen `m1,...,mk` erhoben werden, erfolgen folgende Definitionen:

```
  EVALUATIONOBJECT
  ...
    c VIA h.c;
  ...
  MEASURE m1,...,mk AT c;
```

Definition einer selbstdefinierten Komponente mit einem COMPONENT-Block

Jede Komponente eines nicht in HIT vordefinierten Typs muss selbst definiert werden. Dazu dient ein Block vom Typ

```
TYPE <Komponentenname> COMPONENT (<Parameterliste der Komponente>)
```

der identisch zum `TYPE <Modellname> MODEL`-Block aufgebaut ist, aber zusätzlich zu den genannten Komponenten noch einen Provide-Block enthält. Der Provide-Block hat die folgende Gestalt:

```
  PROVIDE SERVICE
    <ProvidedServicename>(<Parameterliste des Provided Service>);
    [ggf. Bereitstellung weiterer Services]
  END PROVIDE;
```

Implementierung eines hierarchischen Modells

Innerhalb einer selbstdefinierten Komponente kann wieder auf vordefinierte und selbstdefinierte Komponenten verwiesen werden, wodurch ein hierarchisches Modell aufgebaut wird. Dies entspricht einerseits der Verfeinerung einer Verhaltensdefinition durch Zerlegung von Abläufen in kleinere Teilabläufe und andererseits auch der Verfeinerung einer strukturellen Definition durch eine verfeinerte Untergliederung sowohl der Software als auch der Hardware eines Systems.

MODEL- und COMPONENT-Spezifikationen können dabei gegebenenfalls auch auf Komponenten zugreifen, die an anderer Stelle innerhalb der Modellhierarchie definiert wurden, falls ein und dieselbe Komponente mehrfach verwendet werden soll. Diese Mehrfachverwendung ist zum Beispiel sinnvoll, wenn die Belastung eines Prozessors durch eine Menge darauf ausgeführter Prozesse bestimmt werden soll. Diese werden anstatt durch COMPONENT-Statements jeweils durch Statements der Form

```
ENCLOSE <Komponentenname>:<Komponententyp>;
```

verwiesen. Voraussetzung dafür ist, dass die entsprechende Komponente innerhalb des hierarchischen Modells mittels eines COMPONENT-Statements an einer Stelle eingebunden wurde, die von der Stelle aus sichtbar ist, an der er wiederverwendet werden soll. Die Sichtbarkeit einer Komponente ist dabei durch ihrer Stellung innerhalb der Hierarchie definiert.

Für jede Komponente `C` kann angegeben werden, innerhalb welcher Komponente diese mittels eines COMPONENT-Statements definiert wird. Diese Angabe kann bis hin zum MODEL fortgeschrieben werden, so daß schließlich für jede Komponente eindeutig ihre Position innerhalb des hierarchischen Modells durch einen Pfad angegeben werden kann. Dieser Pfad hat die Gestalt `M.C1.C11.[...].C`. Genauso kann die Position jeder anderen Komponente `C'` durch einen Pfad angegeben werden. Eine Komponente `C''` ist von zwei beliebig gewählten Komponenten `C` und `C'` genau dann sichtbar, wenn sie irgendwo auf dem Pfadstück definiert wird, das den Hierarchiepfaden von `C` und `C'` gemeinsam ist.

HIPE baut hierarchische Modelle entsprechend gemäß der folgenden **Hierarchieregel** auf[9]: Von mehreren Komponenten `C1,...,Ck` gemeinsam genutzte Komponenten werden an der tiefstmöglichen Stelle innerhalb der Modellhierarchie definiert, von der aus sie für alle Komponenten `C1,...,Ck` sichtbar sind.

In den folgenden Abschnitten wird erklärt, in welcher Form Elemente von um Annotationen in HIPE-Syntax erweiterten UML-Modellspezifikationen in Konstrukte innerhalb eines derart strukturierten Modells abgebildet werden.

[9] Verallgemeinernd könnte man auch einfach alle oder zumindest alle mehrfach genutzten Komponenten bereits auf der MODEL-Ebene definieren. Der verwendeten Konstruktion liegt die aus dem Performance Engineering mit AntiPatterns abgeleitete Idee zugrunde, dass so bereits aufgrund der Position einer Hardwarekomponente in einem Modell auf deren Potential geschlossen werden kann, für das System zu einem Flaschenhals zu werden.

1.2.6.1 Anwendungsfalldiagramme

Über Anwendungsfalldiagramme wird die Arbeitslast des Systems definiert, das heißt die Anzahl und Art der vom System auszuführenden Abläufe. Diese bestehen aus Akteuren, welche Anfragen an das System stellen, und mit diesen über Assoziationen verbundenen Anwendungsfällen, welche Meta-Abläufe darstellen, die zur Erfüllung der Anfragen ausgeführt werden.

Implementierung von Anwendungsfalldiagrammen

In HIPE ist grundsätzlich nur die Benutzung eines Anwendungsfalldiagramms erlaubt. Dieses bildet die oberste Ebene `MODEL` des erzeugten hierarchischen Modells, in dem einerseits die Arbeitslasten (ankommenden Prozesse) erzeugt und andererseits die Akteure dargestellt werden, welche die Eigenschaften der einzelnen erzeugten Prozesse darstellen, was ihre Anforderungen an vom System bereitgestellte Dienste angeht.

Für jeden möglichen Typ ankommender Prozesse wird ein `SERVICE` deklariert, in dessen Body Angaben über die ausgeführten Meta-Funktionalitäten des Systems gemacht werden, welche im Anwendungsfalldiagramm als Anwendungsfälle dargestellt sind. Im Body des Modells selbst werden mittels `CREATE`-Statements die Arbeitslasten erzeugt. Im folgenden soll erklärt werden, auf welche Weise der Nutzer Angaben über diese Prozesse machen kann, und wie diese genau von HIPE in HI-SLANG umgesetzt werden.

Arbeitslasten

Mit Hilfe der Akteure wird die Arbeitslast des Systems definiert, das heißt es wird die im System befindliche Population von Prozessen modelliert. Die Arbeitslast kann offen (`openload`) oder geschlossen (`closedload`) sein. Offen bedeutet, daß Prozesse in das System eintreten, ihre Anfrage ausführen und die maximale Population gleichzeitig im System befindlicher Prozesse im allgemeinen unbegrenzt sein kann (um die Restriktionen der analytischen Löser einhalten zu können, begrenzt HIPE faktisch die Anzahl an Prozessen), während bei einer geschlossenen Arbeitslast die Anzahl von Prozessen konstant bleibt.

Als Eigenschaft einer offenen Arbeitslast ist die Zwischenankunftszeit der einzelnen Anfragen (`occurence`) anzugeben, also die Zeit, die nach der Ankunft einer Anfrage vergeht, bis die nächste Anfrage an das System gestellt wird. Die Eigenschaften einer geschlossenen Arbeitslast sind die konstante Population des Systems (`population`) und die Zeit, die zwischen dem Ende der Bearbeitung einer Anfrage und dem Stellen der nachfolgenden Anfrage vergehen soll (`extdelay`). Für offene wie geschlossene Arbeitslasten kann alternativ ein Parameter `arrivaltime` angegeben werden, der angibt, zu welchen Zeitpunkten eine Anfrage gestellt werden soll. Der Nutzer gibt in diesem Fall eine durch Kommata getrennte, aufsteigend geordnete Liste von Zeitpunkten $a_1, ..., a_n$ an, zu denen Prozesse vom Typ des annotierten Akteurs in das System eintreten sollen. Bei der folgenden Beschreibung der Statements werden teilweise Pseudovariablen verwendet, um anzudeuten, dass die entsprechende Angabe aus einer UML-Element-Notiz unverändert in den HISLANG-Sourcecode übernommen wird.

Die Anweisungsfolge

```
LET load = openload;
```

```
LET occurence = o;
```

die als Annotation des Akteurs *oload* angebracht wird, wird nach HI-SLANG als

```
CREATE 1 PROCESS oload EVERY o;
```

übersetzt. Die Zeichenkette, die im `let`-Statement in der Notiz anstelle des `o` steht, wird dabei unverändert in den HI-SLANG-Code übernommen.
Die Anweisungsfolge

```
LET load = closedload;
 LET population = p;
  LET extdelay = e;
```

des Akteurs *cload* wird als

```
CREATE p PROCESS cload;
```

übersetzt (die Verarbeitung des `extdelay` erfolgt im Body des Services *cload*, s.u.). Eine Anweisungsfolge

```
   LET load = openload;
LET arrivaltime = a1,a2,...;
```

bzw.

```
   LET load = closedload;
LET arrivaltime = a1,a2,...;
```

wird auf eine Anweisungsfolge

```
CREATE 1 PROCESS load AT a1;
CREATE 1 PROCESS load AT a2;
            ...;
```

abgebildet.

Für jede verarbeitete Arbeitslastdefinition, das heißt für jeden Akteur, wird ein Service erzeugt, der das Verhalten dieses Akteurs beschreibt. Zur Konstruktion des Bodys dieses Services werden Angaben über die vom Akteur ausgeführten Anwendungsfälle verwendet. Stellt der Prozess *cload* dabei eine geschlossene Arbeitslast dar, so dürfen Prozesse dieses Typs das System nicht mehr verlassen. Entsprechend wird der Body des Services *cload* durch

```
LOOP
 <alter Body von cload>
 spend(e);
END LOOP;
```

ersetzt. Dabei stellt `spend(e);` eine Verzögerung des nächsten Aufrufs eines Anwendungsfalls um e Zeiteinheiten entsprechend `LET extdelay = e;` dar.

Anwendungsfälle

Anwendungsfälle stellen die Funktionalitäten oder Meta-Methoden dar, welche das System ausführt. Der Nutzer kann für jeden Anwendungsfall entweder direkt im Anwendungsfalldiagramm Angaben machen, oder auf ein Interaktions-Diagramm verweisen, das den Anwendungsfall darstellt. (HIPE betrachtet allerdings nur Aktivitätsdiagramme.) Mögliche Angaben sind entsprechend die angenommene Bedienzeitanforderung einer Anfrage (`responsetime`) und der Verweis auf das ausarbeitende Diagramm (`diagram`). Zu beachten ist, dass falls direkte Angaben vorhanden sind, HIPE diese Angaben verwendet, statt das verwiesene Diagramm zu untersuchen. Falls alle Angaben fehlen, setzt HIPE ein eventuell vorhandenes Aktivitäts- oder Sequenzdiagramm ein, das den gleichen Namen wie der Anwendungsfall trägt.

Mit dem `responsetime`-Statement kann in HIPE die Bedienzeitanforderung eines Anwendungsfalls in Zeiteinheiten direkt angegeben werden. In den von HIPE konstruierten HIT-Modellen wird jedoch die Arbeitslastanforderung an einen Server als Anzahl von Operationen angegeben, die dieser ausführen soll. Entsprechend muß die angegebene `responsetime` zunächst auf eine Anzahl von Elementaroperationen dieses Servers umgesetzt werden. Ist die Geschwindigkeit eines Servers als s Operationen pro Zeiteinheit gegeben und die Ausführungsdauer als r Zeiteinheiten, so ergibt sich für die entsprechende Aktivität eine Arbeitslastanforderung von $r \cdot s$ Operationen.

Ein Anwendungsfall *uc1*, der mit einer `LET responsetime = r`-Annotation versehen ist, wird in HI-SLANG als `COMPONENT <Server von uc1>: server;` dargestellt. Welcher Server für `uc1` eingesetzt wird, ist davon abhängig, welche aktive Ressource dem Anwendungsfall mit `let cpu = <ServerName>;` zugeordnet wird. Findet eine solche Zuordnung nicht statt, so wird eine Ausführung aus dem Default-Server angenommen. Der Body des Services, mit dem der Anwendungsfall *uc1* assoziiert ist, wird anschließend um das Statement

```
uc1(r*<Geschwindigkeit des Servers von uc1>);
```

ergänzt.

Ein Anwendungsfall *uc2*, dem mit `LET diagram = Diagramm` ein Anwendungsfall- oder Sequenzdiagramm zugeordnet ist, wird als eine nutzerdefinierte Komponente `ADiagramm_1` vom Typ `ADiagramm` dargestellt. Die Beschreibung dieses Typs ergibt sich aus der Übersetzung des entsprechenden Diagramms mit Namen *Diagramm*. Der Body des mit *uc2* assoziierten Services wird um das Statement `uc2;` ergänzt.

Einfügungen in den Code finden in beiden Fällen bei den Used Services statt, wo der Aufruf des atomaren Anwendungsfalls *uc1* und des durch das Aktivitätsdiagramm weiter ausgearbeiteten Anwendungsfalls *uc2* in der Form

```
uc1(amount: REAL DEFAULT r*<Geschwindigkeit des Servers von uc1>);
uc2;
```

stattfindet. Weiterhin werden beide Aufrufe von Used Services im Refer-Teil mittels

```
<Name des Akteurs>.uc1 WITH <Name des Servers von uc1>.request;
<Name des Akteurs>.uc2 WITH ADiagramm_1.request;
```

an die ausführende CPU bzw. an die ausführende selbstdefinierte Komponente angebunden. Der Aufruf der Used Services erfolgt im Body des Services, der den momentan betrachteten

Akteur darstellt, entsprechend der Ausführwahrscheinlichkeiten der entsprechenden Aktivitäten. Im folgenden Abschnitt wird das entsprechende Konstrukt beschrieben und dann ein Beispiel für das Aussehen eines Bodys.

Ausführwahrscheinlichkeiten
Assoziationen setzen Akteure mit Anwendungsfällen in Beziehung. Die Annotation einer Kante liefert die Angabe, mit welcher Wahrscheinlichkeit (`prob`) ein Akteur des adjazenten Typs den über die Kante verbundenen Anwendungsfall aufruft. (Optional kann an einer Kante die Annotation fehlen. Dieser wird dann die zu 1 fehlende Restwahrscheinlichkeit zugeordnet, die nicht bereits durch die Wahrscheinlichkeiten der anderen abgedeckt ist.) Wird an keiner der von einem Akteur abgehenden Kanten eine Wahrscheinlichkeitsannotation gefunden, wird angenommen, daß der Akteur jeden der mit ihm assoziierten Anwendungsfälle mit der gleichen Wahrscheinlichkeit aufruft, also eine Gleichverteilung über alle alternativen Ausführungen.

Falls die Assoziation, die einen Akteur *load* mit einem Anwendungsfall *uc* verbindet, die Annotation `LET prob = p` besitzt, findet sich im Body von *load* die Anweisung

`BRANCH PROB p: uc... END BRANCH;`.

Alle anderen annotierten Anwendungsfälle werden ebenfalls mit ihren Wahrscheinlichkeiten innerhalb dieses `BRANCH`-Blocks eingetragen. Zu beachten ist, daß die Summe aller Wahrscheinlichkeiten höchstens 1 betragen darf.

Wird beispielsweise der Anwendungsfall *uc1* aus dem vorigen Abschnitt mit einer Wahrscheinlichkeit von 0.4 ausgeführt und der Anwendungsfall *uc2* mit einer Wahrscheinlichkeit von 0.6, so hat der Body die Gestalt

```
BRANCH
  PROB 0.4: uc1;
  PROB 0.6: uc2;
END BRANCH;
```

Beschränkungen der Anwendungsfalldiagramme

Zu bemerken ist, daß Anwendungsfälle allein nur eine nicht sehr aussagekräftige Bewertung zulassen, da sie das Verhalten nicht detailliert genug darstellen und nicht an ein bestimmtes System gebunden sind und damit keine Aussage über die Fähigkeit dieses Systems gemacht werden kann, ein Ausführungsszenario zu erfüllen. Um solche Aussagen machen zu können, ist die Betrachtung weiterer Diagramme notwendig.

Da insbesondere das Verhalten der modellierten Anwendung betrachtet werden soll, sind natürlich Diagramme interessant, mit denen dieses detaillierter dargestellt werden kann. Die Einarbeitung in die für das Projekt relevanten Themen (vgl. [PG459 (2005)]) ergab, dass die nächstniedrigere Abstraktionsstufe bei der Darstellung des Verhaltens eines Systems gerade durch Interaktionsdiagramme, das heißt Aktivitäts- und Sequenzdiagramme, gebildet wird. Informationen aus Diagrammen, die statische Eigenschaften der Anwendung und des sie ausführenden Systems darstellen, können bei der Bewertung der Performance höchstens verwendet werden, um in Interaktionsdiagrammen fehlende Angaben über die Speicherkomplexität der

Software und leistungsrelevante Eigenschaften der Hardware zu gewinnen. Erstere Angaben werden aus Klassendiagrammen, letztere aus Verteilungsdiagrammen gewonnen[10]. Im Folgenden wird die von HIPE durchgeführte Umsetzung von Interaktionsdiagrammen dargestellt und in diesem Kontext auch beschrieben, welche Informationen aus der statischen Struktur der Anwendung und des Systems bei der Erstellung des Leistungsmodells berücksichtigt werden.

1.2.6.2 Interaktionsdiagramme

Interaktionsdiagramme, also Aktivitätsdiagramme und Sequenzdiagramme, stellen Ausführungsszenarien des Systems dar, das heißt Methoden oder Meta-Methoden, welche vom System ausgeführt werden, um bestimmte Funktionalitäten zu realisieren.

Aktivitätsdiagramme stellen diese Ausführungsszenarien in verschiedenen Abstraktionsstufen dar; Abläufe können somit hierarchisch modelliert werden, so daß ausgeführte Aktionen in weiteren Aktivitätsdiagrammen weiter zerlegt werden können. Weiterhin können Aktivitätsdiagramme mehrere alternative Abläufe in einem Diagramm darstellen, so dass der Gehalt an Informationen über das Verhalten einer Anwendung selbst höher ist als bei Sequenzdiagrammen. Aus diesen Gründen werden im Rahmen von HIPE Aktivitätsdiagramme umgesetzt und die Sequenzdiagramme nicht weiter betrachtet.

Von HIPE verarbeitbare Aktivitätsdiagramme bestehen aus genau einem Anfangsknoten (*Initial node*), der den Anfang aller dargestellter Abläufe markiert, eine beliebiger Anzahl von Aktivitäten, die ausgeführte Abläufe in Form von atomaren Aktivitäten oder in anderen Diagrammen genauer spezifizierten Aktivitätsabläufen enthalten, Entscheidungsknoten, die getroffene Entscheidungen darstellen, Aufteilungsknoten (*Fork nodes*), mit denen nebenläufige Bereiche modelliert werden können, und beliebig vielen Fluss- bzw. Aktivitätsendknoten (*Flow final* bzw. *Activity final*), die Abschlüsse einzelner im Diagramm dargestellter Abläufe markieren. Diese einzelnen Knoten werden gemäß der darzustellenden Abläufe über Kontrollflusskanten miteinander verbunden. Zusätzlich werden mit *Object*-Elementen Objekte dargestellt, die mittels Objektflusskanten an Aktivitäten angebunden werden, um Objektflüsse (Erzeugung und Verarbeitung von Objekten durch diese Aktivitäten) darzustellen.

Sequenzdiagramme stellen einen Spezialfall der Aktivitätsdiagramme dar. Sie können zum aktuellen Zeitpunkt von HIPE nicht verarbeitet werden.

Implementierung von Aktivitätsdiagrammen

Ein Interaktionsdiagramm oder Diagrammteil wird von HIPE in HI-SLANG in eine selbstdefinierte Komponente verpackt, die einen einzelnen Service *request* besitzt, der als Provided Service dieser Komponente nach außen erscheint, so dass die von dieser Komponente dargestellte Funktionalität im Modell benutzt werden kann. Das Diagramm selbst oder der relevante Block wird als Body dieses Services *request* in der Form eines hierarchischen Warteschlangennetzes kodiert. Sequentielle Abfolgen und Verzweigungsknoten werden in der HI-SLANG-typischen

[10] Ein Beispiel für eine solche Informationsgewinnung ist die Umsetzung des oben genannten "let cpuStatements. Dabei wird die Geschwindigkeit der entsprechenden CPU aus der Annotation des korrespondierenden Knotens im Verteilungsdiagramm ermittelt.

Struktur der `OPEN_CHAIN` kodiert, mit der typischerweise Warteschlangennetze beschrieben werden, in denen zum Beispiel implementierungsuntypisch sich überschneidende, d.h. nicht wohlverschachtelte Sprünge ausgeführt werden können. Elemente, die auf diese Weise nicht kodiert werden können, nämlich nebenläufige Bereiche und Schleifen, werden im Sinne der `OPEN_CHAIN` als einzelnes Element dargestellt, dessen Innenleben in einer selbstdefinierten Komponente ausgearbeitet wird.

Das nachfolgende Codestück stellt einen Anweisungsblock dar, der eine `OPEN_CHAIN` enthält.

```
OPEN_CHAIN
  QNODE node_name
    PROB p1: node_name1
    PROB p2: node_name2
    ...
    PROB pn: node_namen
    ELSE   : node_name(n+1)
  QNODE node_name1
    ...
END OPEN_CHAIN;
```

Jeder QNODE `node_name` stellt im Sinne des HIPE-Paradigmas eine Aktion auf gleicher Ebene dar, kann also sowohl eine atomare Aktivität, die Ausführung eines verwiesenen Aktivitätsdiagramms, die Benutzung einer passiven Ressource oder Leitung oder einen Verweis auf einen nebenläufigen Bereich oder eine Schleife darstellen. `node_name` stellt einen Used Service des Services dar, dessen Body die `OPEN_CHAIN` enthält, also die Benutzung einer CPU oder den Aufruf einer selbstdefinierten Komponente in einer tieferliegenden Hierarchieebene.

Innerhalb einer `OPEN_CHAIN` müssen die Namen dieser Used Services eindeutig vergeben werden. Der Nutzer sollte gleichzeitig aber nicht dazu verpflichtet werden, stets darauf zu achten, daß innerhalb eines Diagramms allen Knoten eindeutige Namen (oder überhaupt Namen) zugeordnet sind. Von HIPE aus findet deshalb eine interne Namensvergabe statt, das heißt den einzelnen in der `OPEN_CHAIN` dargestellten Elementen eines Aktivitätsdiagramms wird ein Name zugewiesen, der innerhalb der selbstdefinierten Komponente, die das Diagramm darstellt, garantiert eindeutig ist. Ein eindeutiger Name im Sinne von HIPE besteht aus dem Namen der benutzten Komponente und einer Ordnungsnummer, die angibt, die wievielte Benutzung der verwiesenen Komponente im aktuellen Diagramm der Used Service darstellt.

Als Name einer durch eine atomare Aktivität benutzten Komponente wird der Name der CPU gesetzt, auf der diese Aktivität ausgeführt wird. Findet innerhalb der Annotation einer Aktivität ein Verweis auf ein anderes Aktivitätsdiagramm statt, so ist der Komponentenname der Name des Aktivitätsdiagramms. Werden nebenläufige Bereiche oder Schleifen von HIPE aus selbständig in selbstdefinierte Komponenten verpackt, werden diesen Namen zugeordnet, die sich aus Kennzeichnungen der Knoten ergeben, die den entsprechenden Bereich charakterisieren. Der interne Name für die fünfte Aktivität, die auf der CPU *myCPU* ausgeführt wird, ist entsprechend also `myCPU_5`, für den dritten Aufruf eines Aktivitätsdiagramms *Diagramm* entsprechend `ADiagramm_3`.

Aktionen

In Aktivitätsdiagrammen werden auszuführende Aktionen als Aktionszustände dargestellt. Aktionszustände können entweder atomar sein, also elementare Aktionen definieren, deren Bedienzeitanforderung (`responsetime`) direkt angegeben wird, oder durch ein Aktivitäts- oder Sequenzdiagramm (`diagram`) weiter ausgearbeitet werden. Aktionen können weiterhin durch direkte Annotation mittels `class` an die für die Ausführung dieser Aktionen verantwortlichen Klassen, zusätzlich mittels `method` an Methoden dieser Klasse gebunden werden. Mit einem `cpu`- oder `resource`-Statement kann für die aktuelle Aktivität die Hardware angegeben werden, auf der diese Aktivität ausgeführt bzw. die bei der Ausführung dieser Aktivität benutzt wird. Zur Berechnung der Laufzeit wird von HIPE nicht nur der Aktionszustand selbst, sondern auch die Kommunikation betrachtet, die aufgrund der von der durch ihn dargestellten Methode oder Meta-Methode konsumierten Eingaben und aufgrund der gelieferten Rückgaben entsteht.

Für eine aktive Ressource (z.B. CPU, siehe Seite 59) bedeutet dies, daß diese mit dem Ausführungsaufwand der Aktion belastet wird (im Sinne der angegebenen Komplexitätsmetrik ist dies gerade $CM(m)$, falls es sich dabei um eine Methode handelt, oder die auf die Geschwindigkeit der CPU skalierte Bedienzeitanforderung `responsetime`), während einer passiven Ressource oder Leitung der entstehende Kommunikationsaufwand zugeordnet wird (falls ein Objekt der Klasse c transportiert wird, ist dies $CP(c)$ bzw. der im Klassendiagramm für die Klasse c angegebene Aufwand, ansonsten die Paketgrößen für Anforderungen und Rückgaben).

Wir betrachten für eine Aktivität *act* verschiedene Möglichkeiten, ihre Bewertung zu spezifizieren. Die eingangs genannten Statements gliedern sich in drei Kategorien, abhängig von ihrer Verwendbarkeit.

- Mit den Statements `cpu` und `class` wird die für die Ausführung einer Aktion verantwortliche CPU festgelegt. Mit `cpu` wird direkt festgelegt, auf welcher CPU die dargestellte Aktion ausgeführt werden soll, während eine Zuordnung mit `class` dazu führt, dass der Aktion die CPU zugeordnet wird, der im Verteilungsdiagramm die angegebene Klasse zugeordnet ist (für nähere Angaben zur in HIPE geforderten Darstellung siehe Seite 62). `cpu` und `class` schließen sich gegenseitig aus, das heißt es kann in der Annotation eines Aktionszustands nur eines der beiden Statements auftauchen.
- Mit den Statements `responsetime` und `method` wird die Ausführungsdauer einer Aktion bestimmt. Mit `responsetime` wird eine direkte Setzung der Bedienzeitanforderung in Zeiteinheiten vorgenommen, während mit `method` der Name einer Methode angegeben wird, die im Rahmen der dargestellten Aktion ausgeführt werden soll. Die Angabe von `method` macht nur Sinn, falls bereits mit `class` die ausgeführte Klasse angegeben wurde. Auch die Statements `responsetime` und `method` sind gegenseitig ausschließend.
- Mit dem Statement `diagram` wird angegeben, daß der Aktionszustand die Ausführung eines Ablaufs darstellt, der durch ein weiteres Aktivititätsdiagramm genauer spezifiziert wird. Wird `diagram` angegeben, so darf kein anderes vorgenanntes Statement verwendet werden.

Für nicht annotierte Aktionszustände wird versucht, aus Angaben, die entweder vom Nutzer innerhalb des Modells oder als allgemeine Default-Werte gemacht werden, Informationen über

die mutmaßlichen Charakteristika eines Aktionszustands zu gewinnen. Sei ein betrachteter Aktionszustand mit *aktiname* beschriftet.

- Falls ein Aktivitätsdiagramm mit dem gleichen Namen *aktiname* existiert, wird angenommen, der Aktionszustand stelle einen Verweis auf eine Ausführung der in diesem Aktivitätsdiagramm dargestellten Funktionalität dar.
- Falls es kein Aktivitätsdiagramm mit dem Namen *aktiname* gibt, wird die Aktivität als atomare Aktivität dargestellt, die auf der Default-CPU mit Default-Ausführungsdauer abläuft.

Die Tabelle 1.6 stellt dar, welche Möglichkeiten zur Darstellung einer Aktion *name* zur Verfügung stehen und welche Umsetzung dabei gewählt wird. Es wird jeweils ausgewiesen, welchen Namen `name` die Komponente trägt, auf der die Aktion ausgeführt wird, und in welchem Umfang *amount* diese benutzt wird. Bei der Implementierung wird im Used-Service-Teil entsprechend mit

```
<name>_<Numerical>(amount: REAL DEFAULT <amount>)
```

der entsprechende Used Service deklariert und im Refer-Teil mit

```
request.<name>_<Numerical> WITH <name>.request
```

an eine atomare oder selbstdefinierte Komponente gebunden.

Die Geschwindigkeit der in der Tabelle erwähnten CPU `defCPU` sei als `sp(defCPU)` gegeben, die Geschwindigkeiten der anderen CPUs analog. `cpu(myClass)` bezeichne die CPU, welche der Klasse `myClass` im Verteilungsdiagramm zugeordnet ist. Die vorher festgesetzte Default-Antwortzeit für eine Aktivität sei `drt`. Die Angabe `CM(myClass.myMethod)` bezeichnet die Komplexität der Methode *myMethod* der Klasse *myClass* gemäß der Metrik für die Methodenkomplexität (vergleiche Seite 34). `defCPU` ist die Default-CPU, die entweder vom Nutzer im Verteilungsdiagramm entsprechend angegeben (vergleiche Seite 27) oder, falls ein Verteilungsdiagramm nicht vorhanden ist, von HIPE automatisch aus Default-Werten erzeugt wird.

Festlegung der ausführenden CPU Ausführungsdauer	keine, Diagramm *name* nicht vorhanden	keine, Diagramm *name* vorhanden	`let cpu = myCPU`	`let class = myClass`
keine, `let diagram = name` nicht vorh.	`defCPU` `drt * sp(defCPU)`	`Aname` -	`myCPU` `drt * sp(myCPU)`	`cpu(myClass)` `drt * sp(myClass)`
keine, `let diagram = name` vorhanden	nicht erlaubt	`Aname` -	nicht erlaubt	nicht erlaubt
`let responsetime = r`	`defCPU` `r * sp(defCPU)`	`defCPU` `r * sp(defCPU)`	`myCPU` `r * sp(myCPU)`	`cpu(myClass)` `r * sp(cpu(myClass)`
`let method = myMethod`	nicht erlaubt	nicht erlaubt	nicht erlaubt	`cpu(myClass)` `CM(myClass.myMethod)`

Tabelle 1.6: Umsetzung eines Aktionszustands in Abhängigkeit von seiner Annotation

Wenn eine Aktivität einer CPU *cpu* zugeordnet wurde, wird die Anzahl *actamount* von Elementaroperationen berechnet, die nötig sind, damit die CPU diese Aktivität ausführen kann, und gemäß der in Tabelle 1.6 angegebenen Umsetzungsvorschrift ein Used Service

```
cpu_<Numerical>(amount: REAL DEFAULT actamount)
```

deklariert, der die Ausführung dieser Aktivität darstellt. Dieser wird im Refer-Teil mittels

```
request.cpu_<Numerical> WITH cpu.request
```

an die entsprechende CPU gebunden. Findet eine Zuordnung zu einem Aktivitätsdiagramm *diag* statt, wird mit

```
Adiag_<Numerical>
```

ein Used Service erzeugt, dem keine Ausführungsdauer zugeordnet ist. Der Provided Service *request* der selbstdefinierten Komponente, die das Aktivitätsdiagramm *diag* darstellt, wird im Refer-Teil mittels

```
request.Adiag_<Numerical> WITH Adiag.request
```

an diesen Used Service gebunden.

Mit dem `resource`-Schlüsselwort kann der Nutzer weiterhin auf eine passive Ressource (z.B. Festplatte) verweisen, die für die Ausführung der dargestellten Aktion benutzt wird. Mit `resourceusage` wird weiterhin angegeben, in welchem Umfang in Elementareinheiten die Ressource in Anspruch genommen wird (z.B. wie viele Elementareinheiten an Daten auf die Festplatte geschrieben werden). Für weitere Informationen über passive Ressourcen siehe Seite 59.

Für die Benutzung einer Ressource *res* im Umfang *useamount* wird ein Used Service

```
res_<Numerical>(amount: REAL DEFAULT useamount)
```

deklariert. Dieser Used Service wird im Refer-Teil mittels

```
request.res_<Numerical> WITH res.request
```

an die selbstdefinierte Komponente gebunden, welche die passive Ressource darstellt.

In der `OPEN_CHAIN` wird eine entweder durch Bindung an eine CPU oder ein Aktivitätsdiagramm dargestellte Aktivität, während derer eine passive Ressource benutzt wird, als Sequenz von zwei QNODEs dargestellt. Der erzeugte Code hat die Gestalt

```
...
QNODE <Darstellung der Aktivität>_<Numerical>
  PROB 1.0: res_<Numerical>;
QNODE res_<Numerical>;
  PROB 1.0: <Nachfolger dieser Aktivität>;
...
```

Kommunikation mittels Kontroll- und Objektflüssen

In UML 2 wird die Kommunikation in Aktivitätsdiagrammen in Form von Kontrollflüssen und Objektflüssen dargestellt. Der Objektfluss stellt im allgemeinen die Übergabe einer Objektinstanz zwischen Elementen zur Laufzeit dar (vgl. [OMG (2005)]), so dass damit Aufrufe und

Rückgaben parametrisierter Aktionen modelliert werden können. Während Kontrollflüsse in der Regel nur einen geringen Kommunikationsaufwand aufwerfen, sind besonders Objektflüsse performancerelevant.

Der Objektfluss besitzt im Sinne des HIPE-Paradigmas eine weitergehende Semantik. Dabei wird zunächst der Aufwand betrachtet, der mit der Instanziierung eines Objektes verbunden ist, der eigentliche Transport von Objekten über im Verteilungsdiagramm modellierte Leitungen (siehe Seite 60) und der ggf. auftretende Aufwand für die Speicherung der transportierten Objekte auf dem Zielrechner. Im folgenden werden diese einzelnen Punkte kurz diskutiert.

1. Speicherverwaltung beim instanziierenden System.
   In vielen Rechensystemen erfordert die Generierung eines Objektes (sowohl in der Form der Instanziierung als auch des Transports dieses Objektes zum fraglichen System) einen weiteren Aufwand, der dadurch entsteht, dass zunächst sichergestellt werden muß, dass überhaupt genug Speicherplatz zur Verfügung steht, um das Objekt abzuspeichern. Dieser Aufwand besteht zum Beispiel in der Durchführung einer Garbage Collection.

   Ein zentrales Problem bei der Betrachtung des Objektflusses besteht nun in der Bestimmung dieses Aufwands. Eine mögliche Strategie ist die einfache Festsetzung, daß ein bestimmter fester Anteil der CPU-Zeit für die Speicherverwaltung reserviert wird. Der potentielle Nachteil dieser Strategie ist jedoch, daß hier nicht garantiert werden kann, daß bei der Instanziierung eines Objektes auch genügend Speicherplatz zur Verfügung steht, wenn der entsprechende Anteil zu klein gewählt wird, oder unter Umständen viel Laufzeit verschwendet wird, weil er zu groß gewählt wird.

   Als Möglichkeit zur Berechnung des tatsächlich notwendigen Aufwands bietet HIPE die so genannte "Jamaica-Semantik" an. Sie basiert auf einer Aufwandsschätzung von Fridtjof Siebert (in [Siebert (2000)]), wie viele Elementareinheiten im Rahmen der Speicherverwaltung betrachtet werden müssen, damit danach garantiert genügend Speicher für die Instanziierung bzw. Abspeicherung eines Objektes einer bestimmten Komplexität K zur Verfügung steht, und möglichst wenig Laufzeit für die Garbage Collection verschwendet wird. Ist M die Gesamtmenge des verfügbaren Speichers, so ergibt sich der Aufwand für die zusätzlich zur Instanziierung durchzuführende Speicherverwaltung in Abhängigkeit von dem prozentualen Anteil $k = K/M$ der Größe des instanziierten Objektes an der Gesamtgröße des verfügbaren Speichers als

   $$P(K,M) = \frac{K}{(1-k) \cdot e^{(1-k)\cdot e^{-k}-1-k}}$$

   Die Gesamtmenge an Speicher, die Programmen auf der entsprechenden CPU zur Verfügung steht, wird durch Annotation der entsprechenden CPU im Deployment-Diagramm mittels `let memory = M;` festgelegt, wobei M ein numerischer Wert in Elementareinheiten im Sinne der Complexity Points (wie im Abschnitt 1.2.5 auf Seite 34 dargestellt) ist. Wird kein `let memory`-Statement angegeben, wird auf die Betrachtung der Garbage Collection verzichtet. Dieser Verzicht kann auf verschiedene Arten interpretiert werden, z.B. daß der Speicher der entsprechenden CPU so groß ist, daß der zusätzliche Aufwand nicht performancerelevant ist, oder dass eine Software modelliert wird, dessen spätere

Arbeitsumgebung ein System ohne Garbage Collection ist.

Soll sie jedoch dargestellt werden, und wird das betrachtete Objekt auf der CPU *myCPU* erzeugt, so wird für die Speicherverwaltung ein Used Service

```
<interner Name für die Speicherverwaltung> (amount: REAL DEFAULT P(k))
```

erzeugt, der im Refer-Teil mittels

```
request.<interner Name für die Speicherverwaltung> WITH myCPU.request
```

an die ausführende CPU gebunden wird.

2. Instanziierung eines Objektes.
   Der Objektfluss, der einen Aktionszustand in einem Aktivitätsdiagramm verläßt, stellt die Instanziierung eines Objektes dar. Die Instanziierung selbst verursacht einen Aufwand an Rechenoperationen auf der CPU, die diesem Aktionszustand zugeordnet ist. Dieser Aufwand wird auf Basis der Komplexität des Objektes in Complexity Points abgeschätzt.

   Die Instanziierung eines Objekts der Klasse *Class* durch die CPU *myCPU* wird entsprechend als Used Service

   ```
   <interner Name für die Instanziierung> (amount: REAL DEFAULT CP(Class))
   ```

   dargestellt, der im Refer-Teil mittels

   ```
   request.<interner Name für die Instanziierung> WITH myCPU.request
   ```

   an die CPU gebunden wird.

   In bestimmten Systemen werden zur Laufzeit keine Objekte instanziiert. Diese Eigenschaft läßt sich darstellen, indem der Nutzer bei der Modellierung von Aktivitätsdiagrammen auf abgehende Objektflüsse verzichtet.

3. Transport von Daten über Leitungen.
   Mit der Metrik von Carbone und Santucci (siehe Abschnitt 1.2.5) kann die Menge an transportierten Daten als Summe der Komplexitäten der transportierten Objekte abgeschätzt und entsprechend auf die Bandbreite, minimale und maximale Paketgröße der benutzten Leitungen umgerechnet werden.

   Welche Daten eingetragen werden oder für die verwendete Leitung überhaupt relevant sind, wird allgemein und auch an Beispielen auf Seite 60 diskutiert. Sind für eine Leitung *conn* eine minimale Paketgröße *connmin* und eine maximale Paketgröße *connmax* festgelegt, und ist K die Komplexität des übertragenen Objektes, so wird dieses in $\lfloor K/connmax \rfloor$ Pakete der Größe *connmax* und ein weiteres Paket zerlegt, das die Größe $connrest = max(connmin, K - (connmax \cdot \lfloor (K/connmax) \rfloor)$ besitzt.

   Die letztendliche Used Service-Definition ist also

```
<[interner Name für die Verwendung der Leitung conn für den]
 Transport des Paketes #1>(amount: REAL DEFAULT connmax);
...
<[interner Name für die Verwendung der Leitung conn für den]
Transport des Paketes #n>(amount: REAL DEFAULT connrest);
```

wobei die einzelnen Services im Refer-Teil mittels

```
request.<Transport Paket #i> WITH conn.request
```

an die transportierende Leitung gebunden werden.

4. Speicherverwaltung beim empfangenden System.
   Entsprechend der Speichermenge, die der CPU zugeordnet ist, die den Objektfluss empfängt, wird die Berechnung für den Aufwand der Speicherverwaltung, die notwendig ist, um das Objekt dort abzuspeichern, auf der empfangenden CPU nochmals durchgeführt, in diesem Falle natürlich bezogen auf die `let memory`-Angabe dieser CPU, und im Refer-Teil mittels

   ```
   request.<interner Name für die Speicherverwaltung> WITH empfCPU.request
   ```

   gebunden.

Ein kompletter Umgang mit dem Objektfluss wird schließlich in der `OPEN_CHAIN` als Abfolge von Aktivitäten dargestellt, die zum einen die CPU belasten, die der Aktivität zugeordnet ist, von der der Objektfluss ausgeht, zum anderen die CPU, die auf das Ziel des Objektflusses bezogen ist.

```
QNODE <[interner Name für die] Ausführung der Quellaktivität>
  PROB 1.0: <Speicherverwaltung der instanziierenden CPU>;
QNODE <Speicherverwaltung der instanziierenden CPU>
  PROB 1.0: <Instanziierung des Objekts>;
QNODE <Instanziierung des Objekts>
  PROB 1.0: <Transport Paket #1>;
QNODE <Transport Paket #1>
  PROB 1.0: ...
...
QNODE <Transport Paket #n>
  PROB 1.0: <Speicherverwaltung der empfangenden CPU>;
QNODE <Speicherverwaltung der empfangenden CPU>
  PROB 1.0: <Ausführung der Zielaktivität>;
QNODE <Ausführung der Zielaktivität>
  ...
```

Damit ein Objektfluss in dieser Weise betrachtet werden kann, muß der Nutzer in MetaMill ein Objekt, das im Rahmen einer Aktivität erzeugt werden soll, mittels einer von der Aktivität abgehenden ObjectFlow-Kante an sie binden. Falls entsprechend ein Objekt durch eine Aktivität konsumiert werden soll, muß eine zu dieser Aktivität hinführende ObjectFlow-Kante

erzeugt werden.

Entscheidungen

Verzweigungszustände reflektieren verschiedene Möglichkeiten, die Ausführung eines Ablaufs fortzusetzen. In der Regel handelt es sich dabei um ja/nein-Entscheidungen oder Zerlegungen von Wertebereichen. Um auf ihnen eine Leistungsbewertung durchzuführen, müssen die einzelnen Zweige bzw. der Verzweigungszustand selbst mit Wahrscheinlichkeiten (mittels des `prob`-Parameters) annotiert werden. Der Nutzer muß entsprechend die im Softwareentwurf gemachten, in der Regel natürlichsprachlichen oder programmiersprachlichen Angaben, die von HIPE semantisch nicht interpretiert werden können, in Wahrscheinlichkeiten umrechnen. Zum Beispiel ist eine Entscheidung à la

```
IF (x > 0) {DoJobNo1();} ELSE {DoJobNo2();},
```

die im UML-Aktivitätsdiagramm durch einen Entscheidungsknoten mit zwei alternativen Nachfolgeaktivitäten `DoJobNo1` und `DoJobNo2` realisiert wird, vom Benutzer in die (evtl. auch nur grob geschätzten) Wahrscheinlichkeiten der beiden Kontrollfluss-Möglichkeiten zu übersetzen, vgl. Abbildung 1.5 auf S. 55.

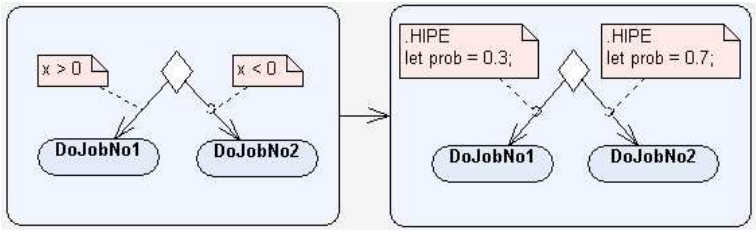

Abbildung 1.5: Übersetzung bzw. Angabe von Verzweigungswahrscheinlichkeiten

Die Wahrscheinlichkeitsannotationen für die einzelnen Alternativen a1,...,ak werden in HI-SLANG im Rahmen des `OPEN_CHAIN`-Statements dargestellt als

```
QNODE <Branchnode>
   PROB p1: <Endknoten der mit a1 annotierten Kante>; ...
   PROB pk: <Endknoten der mit ak annotierten Kante>;
```

Über die automatische Erzeugung von Blöcken

Eine Einschränkung des Vorgängerprojektes HUML war, daß der Nutzer nebenläufige und mehrfach auszuführende Anweisungsfolgen selbst in einem eigenen Aktivitätsdiagramm darstellen und vom Hauptdiagramm aus auf dieses Aktivitätsdiagramm verweisen musste (vgl. [PG438 (2004)]. Diese Formulierung ist kontraintuitiv, da sie von der üblichen Praxis abweicht, mehrfach auszuführende Teilabläufe, deren Komplexität nicht zu groß ist, auf der gleichen Ebene darzustellen.

HIPE erlaubt es dem Nutzer hingegen, solche Abläufe intuitiv auf der gleichen Ebene zu definieren. Die zu ihnen korrespondierenden Anweisungsblöcke können in HI-SLANG allerdings nicht auf der gleichen Ebene dargestellt werden wie atomare Aktivitäten oder Aufrufe von

Aktivitätsdiagrammen. Sie müssen daher von HIPE eigenständig in selbstdefinierte Komponenten umgewandelt werden, die entsprechend an der Stelle aufgerufen werden, an der (ohne Betrachtung dieser besonderen Komponenten) ansonsten der Anfangsknoten des jeweiligen Blocks aufgerufen worden wäre. Entsprechend werden Kontrollflussübergänge, die von anderen Knoten aus auf den Anfangsknoten verweisen, als Verweise auf den durch diesen Knoten definierten Block interpretiert. Die eigentliche Implementierung der dargestellten Blöcke findet dann in diesen selbstdefinierten Komponenten statt. Im folgenden wird diskutiert, wie HIPE nebenläufige und mehrfach auszuführende Anweisungsfolgen in HI-SLANG umwandelt.

Nachteil dieses Vorgehens ist natürlich, daß bestimmte Abläufe nicht dargestellt werden können oder dürfen, wie

- Kontrollübergänge von außerhalb an Elemente, die sich innerhalb eines dieser Bereiche befinden, aber nicht mit dem Anfangsknoten identisch sind,
- das Verlassen solcher Bereiche zu anderen Knoten als dem Fußknoten,
- Kontrollflussübergänge zwischen als nebenläufig gekennzeichneten Pfaden oder
- zwischenzeitlich erfolgende Synchronisationen zwischen einzelnen nebenläufigen Pfaden

Nebenläufigkeiten

Die Angabe `<Used Service zum kkk>` stellt im Folgenden eine Abkürzung dafür dar, dass der Name des Used Service eingesetzt wird, der mittels eines REFER-Statements mit dem Provided Service der Komponente verbunden wird, die dieses Element oder diesen Block darstellt.

Mit Hilfe von Aufteilungsknoten (in MetaMill *fork nodes* genannt) werden konkurrente Abläufe modelliert, d.h. Abläufe, die gleichzeitig stattfinden und zusätzlich an Vereinigungsknoten (*join nodes*) miteinander synchronisiert werden sollen. Diese werden von HIPE dekodiert und in eine äquivalente Konstruktion in HI-SLANG umgewandelt. Dazu sind keine weiteren Annotationen notwendig.

Sollen Pfade a1,...,ak parallel ausgeführt werden, wird an der entsprechenden Stelle eine selbstdefinierte Komponente mit einem einzelnen Provided Service eingesetzt, dessen Body ein

```
CONCURRENT
  <Used Service zum Pfad a1>;
TO
  ...
TO
  <Used Service zum Pfad ak>;
END CONCURRENT;
```

Block ist. Jeder der Pfade a1,...,ak wird wiederum in einer selbstdefinierten Komponente als `OPEN_CHAIN` dargestellt.

Als Vereinigungsknoten können im Rahmen von HIPE sowohl Aktivitäten und Entscheidungsknoten als auch wiederum ein Aufteilungsknoten verwendet werden. Der erste Knoten eines der

drei Typen, der alle durch den Aufteilungsknoten aufgespannten Pfade wieder zu einem Pfad zusammenfasst, wird von HIPE als Vereinigungsknoten interpretiert und bildet entsprechend den Abschluß des nebenläufigen Bereichs. Als Nachfolger der Komponente, die den nebenläufigen Bereich implementiert, wird im Sinne der `OPEN_CHAIN` auf der aktuellen Ebene in dem Fall, dass ein Aktivitätsknoten eingesetzt wird, dieser Aktivitätsknoten eingesetzt. Für den Fall, dass ein Entscheidungs- oder Aufteilungsknoten verwendet wird, wird dessen einzelner Nachfolger eingesetzt. Bei der Verwendung dieser Knoten ist also zu beachten, daß diese dann nicht wieder selbst eine Teilung oder Entscheidung darstellen können.

Die `OPEN_CHAIN`, die das Diagramm darstellt, in dessen Verlauf ein nebenläufiger Bereich modelliert wird, enthält anschließend die Anweisungsfolge

```
\% Ggf. gibt es Knoten, von denen aus Kontrollflussübergänge
\% auf den Anfangsknoten des nebenläufigen Bereiches verweisen.
\% Diese werden hier dargestellt.
QNODE <Used Service zum Vorgänger des CONCURRENT-Blocks>
  PROB ...: <Used Service zum CONCURRENT-Block>;
\% ----
...
QNODE <Used Service zum CONCURRENT-Block>
  PROB 1.0: <Nachfolger des CONCURRENT-Blocks>;
QNODE <Used Service zum Nachfolger des CONCURRENT-Blocks>
  ...
```

Mehrfachausführungen

Gegebenenfalls besteht die Möglichkeit, eine Kette von Aktionen mehrfach auszuführen, das heißt in einem Ablauf zu einem vorher bereits besuchten Zustand zurückzukehren. Eine Schleife wird durch eine vom Ende bis zum Beginn des mehrfach auszuführenden Bereiches zurückweisende Kante markiert.

Die Möglichkeit der stochastischen Beschreibung von Schleifen ergibt sich bereits aus den obigen Ausführungen über die Beschreibung von Entscheidungen. Wird aus einem Entscheidungsknoten heraus rückwärts verwiesen, so wird die Wahrscheinlichkeit des nächsten Schleifendurchlaufs mittels `prob` an der entsprechenden Kante vorgegeben. Ein Verweis mit Wahrscheinlichkeit `prob = p1` auf einen früheren Zustand a1 wird im Sinne der `OPEN_CHAIN` kodiert als

```
QNODE <Branchnode>
  PROB p1: a1;
```

Im Rahmen dieser Beschreibung kann aber nicht angegeben werden, wie viele Male die Schleife durchlaufen wird, sondern nur die Wahrscheinlichkeit, diese Schleife k-mal zu durchlaufen, als p_1^k abgeschätzt werden.

Soll die Anzahl der Durchläufe dieser Kette hingegen fest vorgegeben werden und nicht stochastisch bestimmt werden, kann durch Annotation vorgegeben werden, wie oft diese Schleife durchlaufen werden soll. Das Schlüsselwort `numrepeats` gibt dabei eine exakte Anzahl an.

Zu beachten ist, dass für bestimmte Löser die Anzahl der Wiederholungen einer Schleife auf diese Weise nicht vorgegeben werden kann. In diesem Falle findet von HIPE aus eine Umsetzung statt (siehe Kapitel 1.2.8). Diese kann aber auch vom Nutzer mit dem Schlüsselwort `averagerepeats` selbst gegeben werden, wobei der Parameter hier der Erwartungswert einer geometrisch verteilten Anzahl an Wiederholungen ist (zur Beschreibung der geometrischen Verteilung siehe 1.2.2). Dabei wird eine natürliche Semantik verwendet, indem der Nutzer die Gesamtanzahl an Wiederholungen angibt, das heißt für eine Angabe `numrepeats = r` jede Aktion innerhalb des Schleifenrumpfes r-mal (für `averagerepeats = r` erwartet r-mal) durchgeführt wird.

Zur Herstellung einer Repräsentation in HI-SLANG wird zunächst die mehrfach auszuführende Anweisungsfolge in einer selbstdefinierten Komponente gekapselt, die einen Teilbereich des Aktivitätsdiagramms darstellt und entsprechend selbst als `OPEN_CHAIN` kodiert wird. Deren Name bildet den Aufruf der mehrfach auszuführenden Anweisungsfolge.

Ein `LET numrepeats = r`-Statement wird von HIPE dann in eine weitere selbstdefinierte Komponente umgewandelt, die einen Provided Service besitzt, dessen Body eine Schleife

```
FOR i := 0 STEP 1 UNTIL r LOOP
  <Used Service zur mehrfach auszuführenden Anweisungsfolge>;
END LOOP;
```

enthält, darin wird also die `OPEN_CHAIN` aufgerufen, die eine Iteration der mehrfach auszuführenden Anweisungsfolge darstellt. Desgleichen wird `LET averagerepeats = r` zu

```
AVERAGE r TIMES LOOP
  <Used Service zur mehrfach auszuführenden Anweisungsfolge>;
END LOOP;
```

umgewandelt. Die Komponente, die die Schleife darstellt, erscheint analog zu der Komponente, die nebenläufige Ausführungen beschreibt, in der `OPEN_CHAIN` als

```
...
QNODE <Used Service zum LOOP-Block>
  PROB 1.0: <Used Service zum Nachfolger des LOOP-Blocks>;
QNODE <Used Service zum Nachfolger des LOOP-Blocks>
  ...
```

Als Nachfolger des LOOP-Blocks wird dabei immer der einzelne Knoten ausgewählt, der dem Fußknoten der Schleife nachfolgend angeordnet ist.

1.2.6.3 Verteilungsdiagramme

Verteilungsdiagramme liefern Angaben über die Hardware, auf der die entworfene Software laufen soll, indem sie die benutzten Ressourcen und eine Abbildung der Klassen der Software auf diese Ressourcen darstellen. Im Sinne von HIPE und MetaMill bestehen Verteilungsdiagramme aus Hardwareknoten (*Nodes*), die Prozessoren und passive Ressourcen darstellen, und Assoziationen, die Leitungen darstellen. Jedem dieser Elemente kann durch eine Annotation eine Anzahl von Parametern zugeordnet werden. Weiterhin benötigt werden eine nicht an

ein Diagrammelement gebundene Annotation, mit der der Default-Server festgelegt wird, und Komponenten (*Components*), die innerhalb der Hardwareknoten platziert werden, wobei jede Komponente, die innerhalb eines Hardwareknotens erscheint, den Namen einer Klasse trägt, die auf der durch diesen Hardwareknoten dargestellten CPU ausgeführt und/oder abgespeichert wird.

Implementierung von Verteilungsdiagrammen

Die verschiedenen in einem Verteilungsdiagramm erzeugten Hardwarekomponenten werden in HI-SLANG im Falle von CPUs als flache Komponenten dargestellt, die modellweit als `ENCLOSED` Komponenten immer dann benutzt werden, wenn ihnen mittels `let cpu`-Statements oder weil es sich bei ihnen um die Default-CPU handelt, Aktivitäten zugeordnet werden. Passive Ressourcen und Leitungen werden als selbstdefinierte Komponenten dargestellt, die ebenfalls `ENCLOSED` benutzt werden. Diese selbstdefinierten Komponenten enthalten jeweils einen flachen Server, der für die Ausführung, weiterhin werden darin aber Eigenschaften implementiert, die bestimmte Charakteristika der jeweiligen Hardwarekomponenten darstellen, nämlich im Falle von passiven Ressourcen eine Verzögerung in der Größe der Zugriffszeit und im Falle von Leitungen einen Fehlerdienst, der das Auftreten von Übertragungsfehlern darstellt. Im folgenden wird auf die dargestellten Eigenschaften näher eingegangen.

Ressourcen

Alle Hardwarekomponenten und ihre Assoziationen untereinander stellen im Sinne von HIPE Ressourcen dar, deren Eigenschaften im Zuge der Leistungsbewertung berücksichtigt werden. Die Parameter der Ressourcen werden durch Annotation dieser Diagrammelemente eingestellt.

Ressourcen besitzen eine Geschwindigkeit (`speed`), welche als Verarbeitungskapazität in elementaren Operationen (Rechengeschwindigkeit, Bandbreite) pro Sekunde formuliert wird. Darüber hinaus können für einzelne Typen von Ressourcen spezielle Parameter vergeben werden.

- Prozessoren sind Ressourcen ohne die Fähigkeit zur Parallelverarbeitung, die zu jedem Zeitpunkt nur einen Prozess bedienen können. Um mehrere Prozesse bedienen zu können, wird ein Prozessor in Zeitscheiben, das heißt seine Rechenleistung auf die einzelnen anhängigen Prozesse verteilt. Typische Parameter zur Darstellung der verwendeten Scheduling-Disziplin sind `schedule` und `dispatch`. Für `let schedule`-Statements sind die Werte `immediate`, `fcfs` oder `random` zulässig, für `let dispatch`-Statements die Werte `shared` und `equal` (zur Interpretation der einzelnen Werte siehe Abschnitt 1.2.3 auf Seite 17). Diese Werte für eine CPU werden vom Nutzer entsprechend der intendierten Verarbeitungsstrategie gesetzt. Die Tabelle 1.7 bildet Parametersetzungen für die zulässigen Strategien bzw. Eigenschaften ab, welche die CPU haben soll (vergleiche [Tanenbaum und Woodhull (1997)] und [Weißenberg u. a. (1999)]) Zum Beispiel kann eine Round Robin-Strategie durch Setzung der Parameter `schedule = immediate` und `dispatch = shared` realisiert werden. Eine andere Möglichkeit ist etwa, dass eine im Verteilungsdiagramm als CPU dargestellte Komponente den Ausführungsort von Nutzerdialogen darstellt.

Disziplin/Eigenschaft	`schedule`	`dispatch`
Round Robin Scheduling	`immediate`	`shared`
modelliert Nutzerdialog	`immediate`	`equal`
modelliert E/A-Gerät	`fcfs`	`equal`

Tabelle 1.7: Beispiele für die Konfiguration von Prozessoren

- Passive Ressourcen sind Hardwarekomponenten, die in Beziehung zu Softwarekomponenten stehen, aber nicht direkt an deren Ausführung beteiligt sind, beispielsweise Speichermedien wie Cachespeicher, RAM oder Festplatten. Typische Parameter sind die Größe (`size`), die mittlere Zugriffszeit (`accesstime`) und die Datenübertragungsrate (`speed`). Passive Ressourcen werden als Server mit den vom Nutzer nicht veränderbaren Parametern `schedule = immediate` und `dispatch = equal` realisiert[11].

- Leitungen sind spezielle passive Ressourcen, über die Daten zwischen Hardwarekomponenten transportiert werden können. Sie simulieren Netzwerkverbindungen oder Verbindungsprotokolle. Typische Parameter sind die Bandbreite (`speed`) (in Elementareinheiten, typischerweise in Byte, pro Sekunde) und die minimale (`minpacketsize`) und die maximale (`maxpacketsize`) Paketgröße, die übertragen werden können (siehe dazu [Piastowski (2003)]). Die Parameter für die minimalen und maximalen Paketgrößen werden bei der Implementierung des Objektflusses in Aktivitätsdiagrammen verwendet, um größere Objekte in mehrere Pakete aufzuteilen (siehe Seite 51).

  Auf Leitungen treten oft Übertragungsfehler auf, die zu Paketverlusten führen. Mit dem Parameter `error` kann ein Fehlermodell vorgegeben werden, um für Leitungen Übertragungsfehler oder Paketverluste zu simulieren, die Neuübertragungen von Daten erfordern und somit den Durchsatz begrenzen. Mit diesem Parameter kann der Zeitabstand zwischen dem Auftreten zweier Fehler auf der simulierten Leitung angegeben werden. Durch Einsetzen verschiedener Verteilungen können Eigenschaften wie etwa die positive Korrelation von Fehlern auf physikalischen Medien[12] (vgl. [Xylomenos (1999)]) abgebildet werden.

  Durch Setzung der verschiedenen Parameter kann der Nutzer die simulierten Leitungen so gut wie möglich an die in der späteren Einsatzumgebung vorhandene oder geplante Hardware anpassen. Die Tabelle 1.8 gibt einige Beispiele für Parametersetzungen an (vergleiche z.B. [Xylomenos (1999)] oder [Joines u. a. (2002)]), mit denen bestimmte Leitungen oder Protokolle eingestellt werden können. Es wird zum Beispiel auf die Parameter für die Paketgröße verzichtet, falls eine leitungsvermittelte Verbindung modelliert werden soll. Weiterhin gibt es teilweise verschiedene alternative Modellierungen. Ist zum Beispiel bekannt, dass eine Leitung mit bestimmten Werten von `speed = s` und

[11] Hier wird eine Ressource so betrachtet, daß die interne Verarbeitungslogik beliebig viele Anfragen zwischenspeichern kann und diese ggf. umsortiert, um einen möglichst effektiven Zugriff zu realisieren. Eine andere mögliche Interpretation wäre ein `fcfs`-Schedule, das andeutet, daß z.B. eine Festplatte nur eine Anfrage gleichzeitig bearbeiten kann.

[12] Das heißt, dass Fehler auf Leitungen, die in der Regel durch die Hardware bedingt sind, tendentiell gehäuft vorkommen.

`error = e` eine effektive Übertragungsgeschwindigkeit `F(s,e)` besitzt, so kann stattdessen auch direkt `speed = F(s,e)` gesetzt und das `error`-Statement entfernt werden (vgl. [Joines u. a. (2002)]).

In der Tabelle sei s die Geschwindigkeit der entsprechenden Verbindung, und - bedeutet, dass das Schlüsselwort in dem Falle in der Annotation nicht angegeben werden muss. Die Angaben zur Geschwindigkeit erfolgen in Byte pro Sekunde, Paketgrößen werden in Byte aufgeführt. Angegebene Geschwindigkeiten sind also entsprechend umzurechnen. Der Wert von `error` wird als vorgegebener Zeitabstand von Fehlern in Sekunden interpretiert. Dabei soll der Wert von error leicht zu berechnen sein. Jeder Fehler beansprucht eine Elementareinheit. Hat die Leitung also eine Bandbreite von s Elementareinheiten pro Sekunde und soll sie zu einem Prozentsatz $0 \leq p \leq 1$ fehlerhaft arbeiten, so müssen pro Sekunde $p \cdot s$ Fehler auftreten, dies führt zu einem Wert für `error` von $1/(p \cdot s)$.

Leitungstyp	`speed`	`minpacketsize`	`maxpacketsize`	`error`
interner Bus	s	-	-	-
Ethernet-Leitung (1)	s	40	1500	$1/(0.4 \cdot s)$
Ethernet-Leitung (2)	$0.6 \cdot s$	40	1500	-
WLAN-Leitung (1)	s	1400	1400	$negexp(0.0155 \cdot s)$
WLAN-Leitung (2)	s	1400	1400	$1/(0.023 \cdot s)$
Verbindung per TCP/IP	s	576	576	-

Tabelle 1.8: Beispiele für die Konfiguration von Leitungen

Eine CPU *cpu* mit einer Geschwindigkeit von s Operationen pro Sekunde, einer zugeordneten Warteschlangendisziplin *sch* und einer Abfertigungsdisziplin *dis* wird von HIPE auf eine einzigartige und entsprechend modellweit verwendete flache Komponente

```
cpu: server(LET schedule := sch, LET dispatch := dis(LET speed := s))
```

abgebildet. Modellweit verwendet werden kann die Komponente, indem sie in der obersten Hierarchieebene, in der sie entweder verwendet wird, oder falls sie in verschiedenen Pfaden der Hierarchie verwendet wird, in der Modellebene, die Oberebene aller dieser Hierarchiepfade ist, einmal als `COMPONENT cpu: server...` deklariert und an allen anderen Stellen, wo sie verwendet wird, mittels `ENCLOSE cpu:server...` auf die einmal definierte Komponente verwiesen wird.

Eine passive Ressource *pres* mit Zugriffsgeschwindigkeit t und Zugriffszeit a wird auf eine modellweit verwendete selbstdefinierte Komponente vom Typ *resource* abgebildet, die genauso wie die CPU zunächst einmal mittels `COMPONENT` deklariert und danach mit `ENCLOSE` verwiesen wird. Diese Komponente enthält ihrerseits einen HIT-Server mit der Geschwindigkeit t, wobei einem Aufruf dieses HIT-Servers ein Aufruf `spend(a);` vorgeschaltet ist, der die Latenzzeit der Ressource abbildet. Der Body des Services *request* einer selbstdefinierten Komponente des Typs *resource* enthält also die Aufrufe

```
spend(a);
req(amount);
```

In der Modellebene, in der sie verwendet wird, wird die selbstdefinierte Komponente der Ressource mittels des Used Service

```
<Interner Name dieser Benutzung der Ressource>(amount: REAL DEFAULT resourceusage);
```

in das Warteschlangennetz eingefügt und im Refer-Teil mittels

```
request.<Interner Name dieser Benutzung der Ressource> WITH pres.request
```

angebunden.

Leitungen mit Bandbreite w werden analog zu Ressourcen auf eine selbstdefinierte Komponente vom Typ *Endknoten1_Endknoten2* abgebildet, wobei *Endknoten1* und *Endknoten2* die Namen der CPUs sind, die die Leitung im Verteilungsdiagramm miteinander verbindet. Diese Komponente enthält einen Priority Server *Endknoten1_Endknoten2_server* mit der Disziplin *preemptive repeat* und der Geschwindigkeit w und zur Abbildung des Fehlermodells einen nach außen nicht angebotenen Dienst *errorService*, der gemäß des gewählten Fehlermodells Übertragungsfehler auf der Leitung simuliert, die Neuübertragungen erzwingen. Sowohl der nach außen weitergereichte Service *request* als auch *errorService* besitzen jeweils einen Used Service, die im Refer-Teil der Komponente mit

```
request.req WITH Endknoten1_Endknoten2_server.request;
errorService.error WITH Endknoten1_Endknoten2_server.request;
```

an den Priority Server gebunden werden. Das Auftauchen von Fehlern wird im Anweisungsblock der Komponente durch `CREATE 1 PROCESS errorService EVERY E;` dargestellt, wobei E der Wert aus dem `let error = E`-Statement zu der Leitung im Verteilungsdiagramm ist. Die Komponente wird modellweit verwendet und entsprechend genauso angebunden wie andere Komponenten, die Ressourcen darstellen.

Zuordnung von Klassen zu Ressourcen

Um entscheiden zu können, ob bei der Darstellung des Objektflusses in Aktivitätsdiagrammen (siehe Seite 51) Leitungen benutzt werden, sind Angaben über die Orte zu machen, an denen Objekte gespeichert werden. Die Angabe über den Ort, an dem ein Objekt gespeichert wird, bestimmt darüber, ob bei der Darstellung der Verwaltung eines eben erzeugten Objektes die Benutzung einer Leitung dargestellt wird, die dieses Objekt von der CPU, an der es erzeugt wird, zu der CPU transportiert, an der es gespeichert wird. Aus HIPE-Sicht muss also jede Klasse, von der in einem Aktivitätsdiagramm ein Objekt instanziiert oder konsumiert wird, im Verteilungsdiagramm einer CPU zugeordnet werden. Dies geschieht in MetaMill durch Erzeugung einer Komponente (*Component*), innerhalb eines Knotens (*Node*), der eine CPU darstellt.

1.2.6.4 Klassendiagramme

Klassendiagramme liefern Angaben über die zu erwartende Größe von Objekten, die entweder vom Nutzer mittels `size` direkt selbst angegeben werden kann oder von HIPE auf Basis der schon gemachten Angaben im Entwurfsmodell selbst mit Hilfe der Complexity-Point-Metrik

von Carbone und Santucci (vergleiche Abschnitt 1.2.5) berechnet wird.

Klassendiagramme werden nicht direkt in HI-SLANG umgesetzt, sondern nur Angaben über die Komplexität der einzelnen Klassen gemäß der Complexity Points-Metrik generiert, mit denen der Aufwand bestimmt wird, der bei der Realisierung von Objektflüssen entsteht.
Diese Angaben können überlagert werden, indem der Nutzer mit `LET size = s` als Annotation einer Klasse *Class* eine Größe in Elementareinheiten im Sinne der Complexity Points-Metrik vorgibt. Entsprechend wird für $CP(Class)$ der Wert s eingesetzt.

1.2.6.5 Zustandsdiagramme

Zustandsdiagramme stellen Zustände von Klassen bzw. Methoden und die Bedingungen dar, unter denen diese Klassen bzw. Methoden Zustandsübergänge durchführen. Aus Sicht der Complexity Points-Metrik läßt die Komplexität eines Zustandsdiagramms, also die Anzahl der Zustände und der Übergänge zwischen ihnen, Rückschlüsse auf die Komplexität der dargestellten Klasse oder Methode zu.

Sie werden von HIPE im Rahmen des HIPE-Paradigmas selbst nicht bewertet, mit einer Klasse oder einer Methode assoziierte Zustandsdiagramme können aber in der weichen Analyse nach [Carbone und Santucci (2002)] verwendet werden, um durch andere Teilauswertungen gemachte Aussagen über die zu erwartende Komplexität von Objekten bei Objektflüssen in ihrer Qualität zu verbessern.

1.2.7 Darstellungskonzept

Das Darstellungskonzept von HIPE dient im Wesentlichen zur textuellen und grafischen Veranschaulichung der Ergebnisse aus der harten und weichen Analyse. Eine grafische Darstellung von Daten macht aber nur dann einen Sinn, wenn es gelingt, numerische Relationen derart in visuelle Variablen umzusetzen, dass die Beziehung dieser Variablen durch Wahrnehmung einfacher, d.h. mit geringerem geistigen Aufwand zu erfassen sind als ohne, wobei jedoch keine oder nur unwesentliche Abstriche an der Genauigkeit der zielrelevanten Relationen in Kauf zu nehmen ist. Die Hauptmerkmale des Darstellungskonzepts sind im Folgenden:

- Die textuelle und grafische Darstellung der Ergebnisse aus der harten und weichen Analyse, wobei die Datenreihen der harten Analyse aus dem in HIT erzeugten DUMP-File ausgelesen werden. Die Datenreihen der weichen Analyse werden zur Laufzeit von HIPE erzeugt und ausgelesen.
- Für die textuelle Darstellung werden die Ergebnisse in einer Baumstruktur abgelegt. Dieser Form der Aufbewahrung erlaubt aufgrund seines hierarchischen Aufbaus eine übersichtliche und einfache Bedienung. Zu den Ergebnissen zählen alle existierenden Diagramme. Darüber hinaus werden zu den jeweiligen Diagrammen die verfügbaren Evaluationsobjekte und deren Maße und dazugehörigen Schätzer angezeigt. Siehe Abbildung 1.6 auf S. 64. Die grafische Präsentation der Ergebnisse erfolgt in Form von zweidimensionalen Säulen- und Punktdiagrammen, wobei letztere für die Ergebnisse der weichen Analyse dient.

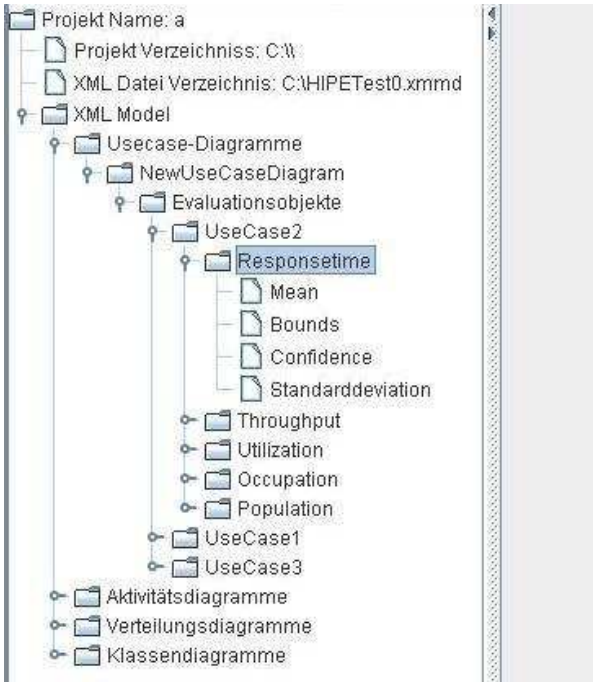

Abbildung 1.6: Baumstruktur

Wenn man mehrere Ergebnisse grafisch darstellen will, so kann man für jede Datenreihe ein gesondertes Diagramm oder alle Datenreihen in einem einzigen Diagramm präsentieren. Für

die harte Analyse werden die Ergebnisse einer Datenreihe gesondert in einem Säulendiagramm dargestellt. Bei der weichen Analyse erfolgt die gesamte Ergebnisdarstellung in einem einzigen Punktdiagramm. Beide Grafiktypen erfüllen im Übrigen von den Wahrnehmungsbedingungen her ein sehr hohes Genauigkeitsniveau.

Eine Datenreihe dient dabei als Struktur zum Speichern der Ergebnisse aus der harten bzw. weichen Analyse. Diese sind die Schätzer *Mean*, *Bounds*, *Confidence Level* und *Standard Deviation* — je nach Auswahl der Analysevorgaben. Die Schätzer gelten für alle Leistungsmaße *Population*, *Throughput*, *Responsetime* und *Utilization*, die wiederum bei Verfügbarkeit für alle existierenden Evaluationsobjekte gelten. Im Folgenden werden die Diagramme an einem exemplarischen Experiment mit einer Serienlänge von 3 für die jeweiligen Schätzer dargestellt.

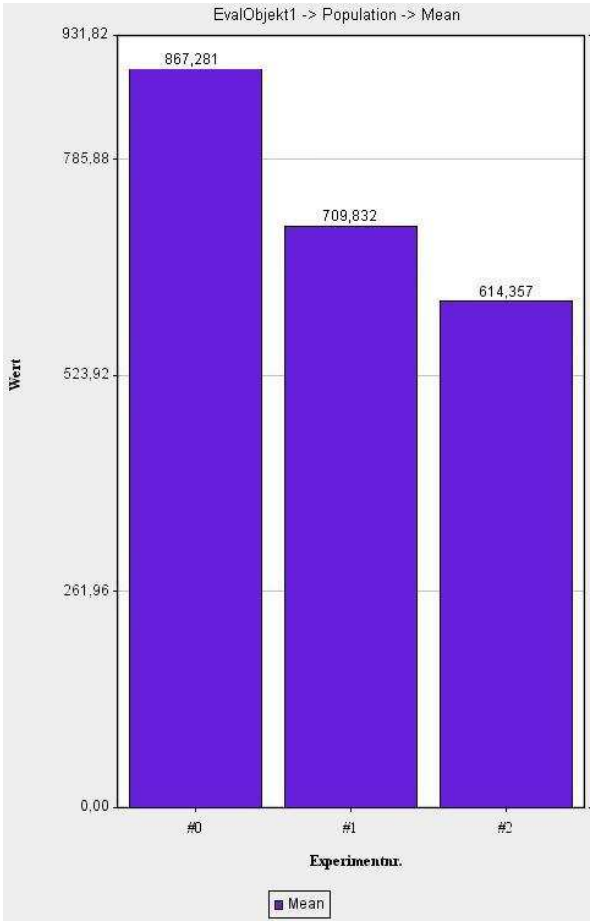

Abbildung 1.7: Visualisierung des Mittelwerts Mean der Population

Das Diagramm und seine Elemente (Säulen, Skalen, Raster, Rahmen) wurden wie folgt aufgebaut. Die Überschrift des Diagramms zeigt für alle Schätzer an, welcher Wert, bezogen auf das ausgewählte Leistungsmaß und bezogen auf das ausgewählte Evaluationsobjekt selektiert wurde. Auf der Y-Achse des zweidimensionalen Säulendiagramms werden die Schätzerwerte aufgetragen. Die X-Achse gibt bei einem Experiment bzw. bei einer Experimentserie das jewei-

lige Experiment an. Die Legende zur Identifizierung der Säulen wird unterhalb des Diagramms dargestellt. Die berechneten Werte werden dabei direkt oberhalb der Säulen angezeigt. Abbildung 1.7 auf S. 65 zeigt das so aufgebaute Diagramm exemplarisch für den Schätzer Mean des Leistungsmaßes Population.

Die Rahmenstruktur für die grafische Darstellung der Schätzer *Bounds*, *Confidence Level*, *Standard Deviation* und *Mean* ist identisch. Bei der Darstellung der Ergebisse für den Schätzer *Bounds* wird aber den drei berechneten Werten *Mean*, *Upperbound* und *Lowerbound* jeweils eine Säule zugeordnet. Diese Art eines Diagrammtyps nennt man *ClusteredBarChart*. Siehe Abbildung 1.8 auf S. 66.

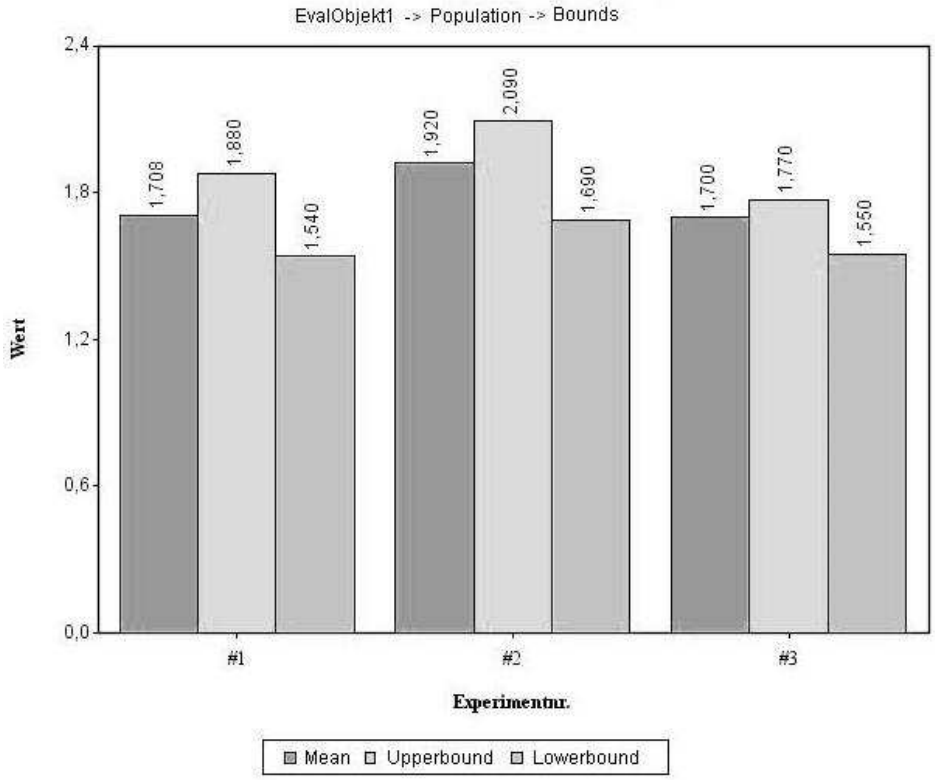

Abbildung 1.8: Visualisierung des Schätzers Bounds

Bei dem Schätzer *Confidence Level* wird ebenfalls ein *ClusteredBarChart* verwendet, der pro Experiment jeweils den *Mean*-Wert, die absolute untere Schranke des Konfidenzintervalls (C_Lowerbound) und die absolute obere Schranke des Konfidenzintervalls (C_Upperbound) anzeigt. Siehe Abbildung 1.9 auf S. 67. .

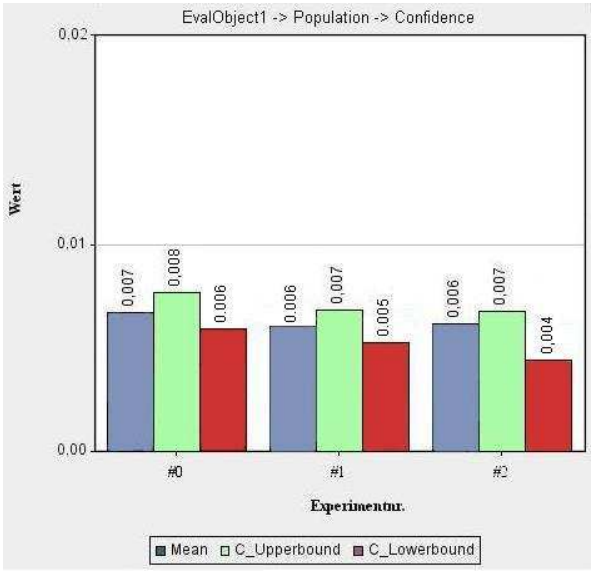

Abbildung 1.9: Visualisierung des Konfidenzintervalls

Beim Schätzer *Standarddeviation* wird ebenfalls ein *ClusteredBarChart* verwendet, das wiederum pro Experiment jeweils den *Mean*-Wert und die *Standard Deviation* anzeigt. Siehe Abbildung 1.10 auf S. 68.

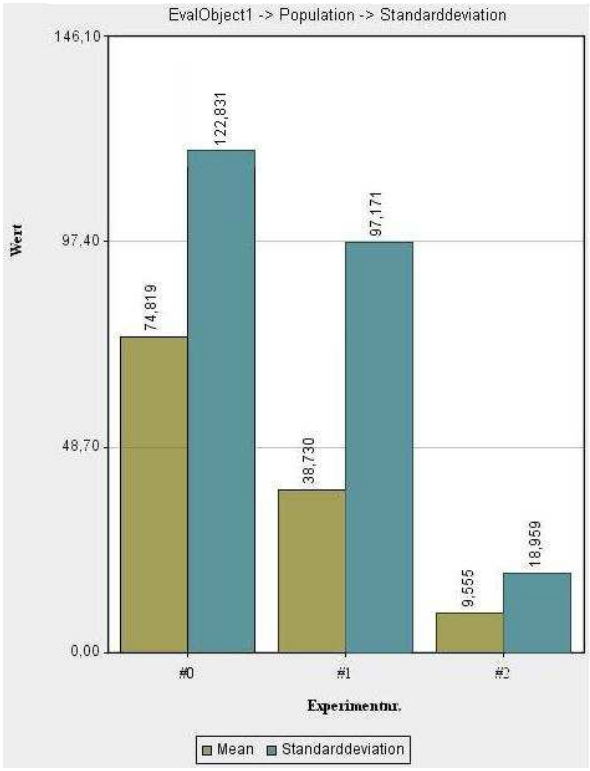

Abbildung 1.10: Visualisierung der Standardabweichung

Die grafische Darstellung der Ergebnisse der weichen Analyse werden allesamt in einem einzigen Punktdiagramm dargestellt. Dabei werden auf der X-Achse die verwendeten weichen Maße angezeigt. Diese sind DIP, NOC, NOT, RFC, MIF, AIF, MHF, AHF, PF, COF, LOC und CP. Die Abkürzungen stehen für folgende objektorientierte Leistungsmaße:

- DIP = Depth of Inheritance
- NOC = Number of Children
- NOT = Number of Types
- MIF = Method Inheritance Factor
- AIF = Attribute Inheritance Factor
- MHF = Method Hiding Factor
- AHF = Attribute Hiding Factor
- PF = Polymorphismusfaktor
- COF = Kopplungsfaktor

Auf der Y-Achse befinden sich für alle existierenden Klassen des Klassendiagramms die berechneten Werte der Leistungsmaße. Die vorhandenen Ergebnisse werden dabei von farblich

abgestimmten rautenförmigen Geometrien dargestellt. Die Abbildung 1.11 auf S. 69 stellt die Werte für ein Klassendiagramm mit 2 Klassen dar.

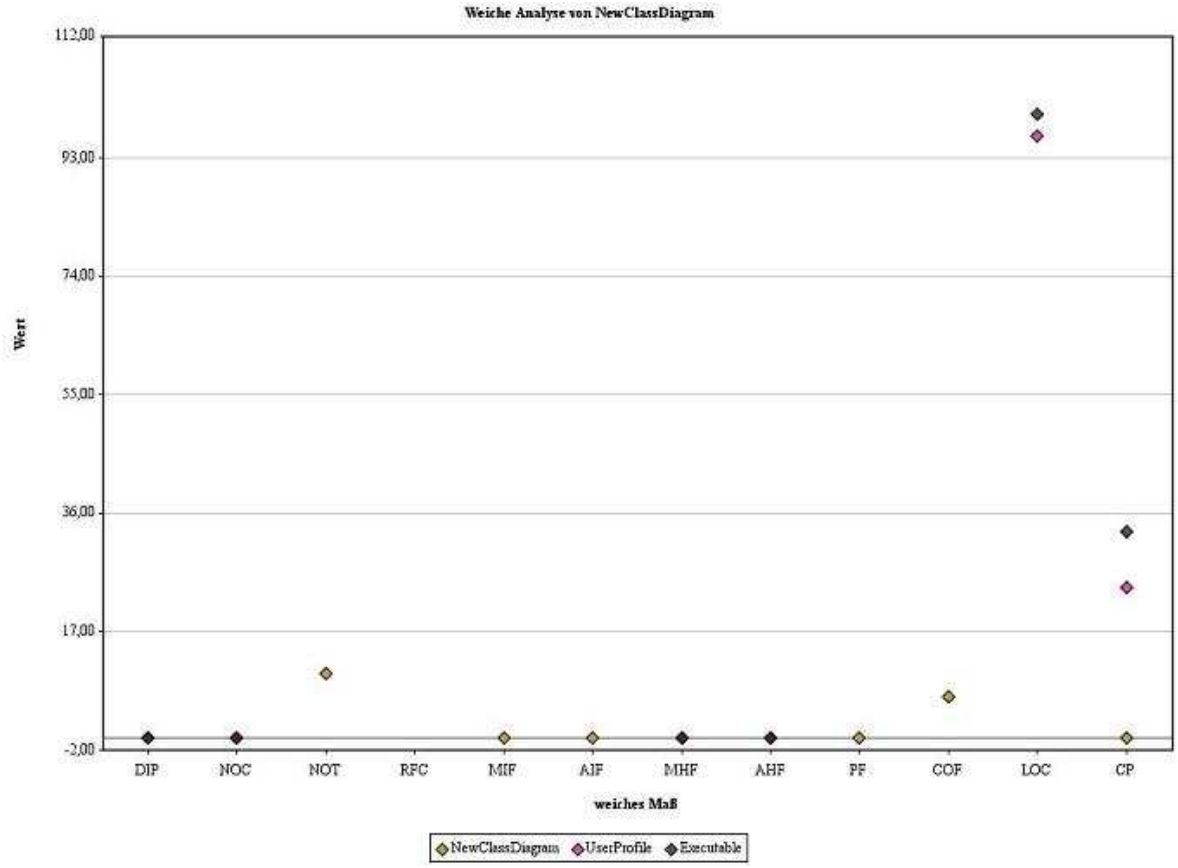

Abbildung 1.11: Visualisierung der Ergebnisse der weichen Analyse

Neben der grafischen Ausgabe der Ergebnisse ist eine weitere Funktion des Darstellungskonzepts das Erzeugen einer Ergebnisdatei. Bei dieser handelt es sich um eine Kopie der Eingabe-XMI-Datei, mit dem Unterschied, dass die REQUIRED- und GET-Statements durch die von HIPE generierten Ergebnisse ersetzt werden.

Für eine Experimentserienlänge von 4 und folgendes GET-Statement

```
GET utilization;
```

würde die Ergebniszeile folgendermaßen aussehen:

```
utilization = 0.05, 0.1, 0.15, 0.2;
```

An den Positionen der Ergebnisse hinter dem „=“-Symbol erkennt man das Experiment, für welches das Ergebnis berechnet wurde. Die 0.05 ist das Ergebnis des ersten Experiments, die 0.1 das des zweiten, usw.

Handelt es sich bei dem Statement um ein REQUIRED-Statement

```
REQUIRED utilization > 0.1;
```

dann findet man folgende Zeilen in der Ausgabedatei:

```
utilization = 0.05 und das Requiredstatement ist nicht erfüllt für Modelparameter:
                   Occurence am Modellelement User = 1;
utilization = 0.1  und das Requiredstatement ist nicht erfüllt für Modelparameter:
                   Occurence am Modellelement User = 2;
utilization = 0.15 und das Requiredstatement ist erfüllt für Modelparameter:
                   Occurence am Modellelement User = 3;
utilization = 0.2  und das Requiredstatement ist erfüllt für Modelparameter:
                   Occurence am Modellelement User = 4;
```

Wie man sieht ist der variable Modelparameter die Zwischenankunftszeit am Modellelement „User“, welcher nacheinander die Werte 1, 2, 3 und 4 annimmt. Dem Benutzer von HIPE wird an dieser Stelle also klar, dass erst für Zwischenankunftszeiten größer 2 die Auslastung am betrachteten Modellelement Werte größer als 0.1 annimmt.

Die Ausgabedatei wird im selben Verzeichnis gespeichert, in dem sich die Eingabedatei befindet. Der Name der neuen Datei setzt sich sich aus dem der Ursprungsdatei und dem Suffix „-HIPE-Result“ zusammen. Heißt die Eingabe-XMI-Datei beispielsweise „Test.xmmd“ so erhält man nach der Analyse eine Datei mit folgender Bezeichnung „Test-HIPE-Result.xmmd“.

1.2.8 Einschränkungen von Lösern

Zu beachten ist, dass nicht unter allen Lösern, die der Nutzer auswählen kann, alle vom Nutzer gewünschten Modelleigenschaften vollständig berücksichtigt werden können. Andererseits soll der Nutzer grundsätzlich dazu in die Lage versetzt werden, ein lauffähiges Modell zu erzeugen, das unter dem von ihm gewählten Löser auswertbar ist, auch wenn dieser für die gemäß der Abschnitte 1.2.6.1-1.2.6.5 durchgeführte Umsetzung des eingegebenen UML-Modells zunächst nicht anwendbar ist. Zu diesem Zwecke werden bestimmte Anpassungen vorgenommen, die Empfehlungen aus dem HI-SLANG Reference Manual [Büttner u. a. (1999)] folgen oder zumindest von der dort entnommenen Darstellung der Eigenschaften der verschiedenen Löser motiviert sind. Diese verändern allerdings teilweise die Semantik des Modells, weshalb ihre Verwendung nicht grundsätzlich empfohlen wird. Der Nutzer wird entsprechend darauf hingewiesen, wenn solche Anpassungen vorgenommen wurden. In manchen Fällen kann allerdings gar keine Anpassung vorgenommen werden.

In folgenden Situationen werden Anpassungen vorgenommen:

- Soll ein Modell mit Hilfe eines analytischen Lösers ausgewertet werden, werden `arrivaltime`-Anweisungen in Abhängigkeit von der Anzahl der darin angegebenen Werte und des Typs der zu definierenden Arbeitslast verschieden umgesetzt.
  - Folgt dem `arrivaltime`-Schlüsselwort nur die Angabe *eines einzigen* Zeitpunktes, so wird die Arbeitslastdefinition auf eine `CREATE 1 PROCESS ...`-Anweisung abgebildet.
  - Für bestimmte Analysen, zum Beispiel der Überprüfung der Belastungsgrenze eines Systems, kann es wünschenswert sein, dafür eine geschlossene Arbeitslast zu

definieren, zu der in gewissen Abständen immer wieder Prozesse dazukommen. Mit einem analytischen Löser kann eine solche Analyse nicht durchgeführt werden, da das untersuchte Modell keinen stationären Zustand erreicht, ein analytischer Löser aber die Existenz eines solchen voraussetzt (vgl. [Büttner u. a. (1999)], Seite 271). Eine geschlossene Arbeitslast wird entsprechend auf ein Statement `CREATE n PROCESS ...` abgebildet, falls dem `arrivaltime`-Schlüsselwort *eine Liste* von n Zeitpunkten folgt. (Um die genannte Analyse dennoch durchzuführen, empfiehlt sich die Definition einer Experimentserie über die Population des entsprechenden Akteurs.)

– Bei der Definition einer offenen Arbeitslast wird aus *einer Liste* von Zeitpunkten eine `CREATE 1 PROCESS ... EVERY E`-Anweisung erzeugt. Dabei ergibt sich das angegebene Zeitintervall E als Durchschnitt der Zeitintervalle zwischen den einzelnen vom Nutzer angegebenen Ankunftszeitpunkten $a_1, ..., a_n$, also

$$E = \frac{\sum_{i=1}^{n}(a_i - a_{i-1})}{n}$$

mit dem Hilfswert $a_0 = 0$.

- `FOR`-Schleifen, die über `numrepeats` beschrieben werden, können mit analytischen Lösern nicht verwendet werden. HIPE bildet solche Schleifen auf `AVERAGE` ab, falls die Löser MARKOV, DOQ4 oder LIN2 verwendet werden sollen. Diese Anpassung ist allerdings kritisch, da in der Praxis die Anzahl von Iterationen einer Schleife nicht geometrisch verteilt, sondern entweder – im Rahmen einer FOR-Schleife – deterministisch oder – bei einer datenabhängigen WHILE-Schleife – nichtdeterministisch ist. Im allgemeinen wird aber bei WHILE-Schleifen die Wahrscheinlichkeit, eine weitere Iteration durchzuführen, abnehmen, während diese Wahrscheinlichkeit bei einer geometrischen Verteilung von Iteration zu Iteration zunächst zunehmen wird, bis der Wert von `averagerepeats` erreicht ist. Eine `AVERAGE`-Schleife kann dann zwar den Mean-Wert der Antwortzeit solcher Schleifen vorhersagen, nicht aber die Verteilung der Antwortzeiten (vgl. [Jonkers (1995)]).

- Nicht in jeder Modellwelt sind alle möglichen Verteilungen für Zufallsvariablen zulässig, deswegen werden ggf. an numerischen Werten, die vom Nutzer eingegeben werden, Anpassungen vorgenommen.

  – In den analytischen Modellwelten sind keine "deterministischen Verteilungen", das heißt konstante Werte, zur Angabe der Bedienzeitanforderungen von Used Services erlaubt. Diese werden entsprechend durch die zulässige Verteilung `cox` mit gleichem Mittelwert und geringer Varianz approximiert. Soll also ein deterministischer Wert x angegeben werden, so wird dieser zu einem Wert `cox(x,0.32)` umgesetzt. Damit folgt die Gruppe einer im HI-SLANG Reference Manual gegebenen Empfehlung ([Büttner u. a. (1999)], Seite 270). Die Anzahl der Phasen der von HIT dort eingesetzten coxg-Verteilung ergibt sich gemäß [Augustin und Büscher (1982)] als $\lceil 1/0.32^2 \rceil = 10$. An der gegebenen Stelle wird also eine Verteilung mit 10 Phasen und Mittelwert x eingesetzt. Durch diese Anpassung kann der Determinismus auch unter MARKOV annähernd repräsentiert werden, allerdings steigt die Komplexität des Modells und damit die aufzuwendende Rechenzeit.

- Analytische Löser unterstützen in `PROB`-Statements in bestimmten Fällen keine Wahrscheinlichkeitsangaben von 0.0 oder 1.0. In anderen Fällen werden diese zwar unterstützt, dies ist allerdings von der Struktur der definierten Kette und von verschiedenen internen Eigenschaften abhängig (vgl. [Büttner u. a. (1999)], Seite 263). Im Zuge von Experimenten mit HIT während der Entwicklung von HIPE hat sich diese Einschränkung in seltenen Fällen in den von HIPE konstruierten `OPEN_CHAINs` unter dem Löser MARKOV manifestiert. Deswegen werden bei Verwendung des Lösers MARKOV an Stellen in `OPEN_CHAINs` Wahrscheinlichkeitsangaben von 1.0 durch die geringfügig kleinere Angabe 0.999999 ersetzt[13].
- Unter analytischen Lösern müssen die Zwischenankunftszeiten bei offenen Arbeitslasten `negexp`-verteilt sein. `occurence`-Statements, bei denen die Zwischenankunftszeiten deterministisch als x oder cox-verteilt mit Mittelwert x Zeiteinheiten angegeben werden, werden zu `negexp(1/x)` umgesetzt.
- Unter MARKOV kann die Scheduling-Disziplin FCFS nicht verwendet werden. Eine Approximation, die von den Machern von HIT selbst nahegelegt wird, ist die Änderung dieser Angabe auf RANDOM [Weißenberg u. a. (1999)].
- Unter LIN2 muss der Zeitbedarf für jedes Element eines Diagramms, sofern eine CPU mit `fcfs` oder `random`-Scheduling verwendet wird, `negexp`-verteilt mit der gleichen Rate sein, also den gleichen Erwartungswert haben.

Darüber hinaus gibt es Situationen, in denen eine Anpassung nicht möglich ist. Es wird entsprechend empfohlen, dafür einen anderen Löser zu verwenden, der nicht der entsprechenden Restriktion betroffen ist.

- `CONCURRENT`-Bereiche können in analytischen Lösern nicht verwendet werden. Es wurde kein Konstrukt implementiert, um diese Bereiche anderweitig zu behandeln. Soll also zur Lösung ein analytisches Verfahren verwendet werden, darf es im Diagramm keine nebenläufigen Bereiche geben.
- Unter LIN2 können keine Leitungen verwendet werden, da dieser Löser die Art und Weise, in der Leitungen in HIPE realisiert sind (*prioserver*), nicht unterstützt.
- Eine weitere Einschränkung von LIN2 (und auch im Sinne der Logik) ist es, dass Parameter von selbstdefinierten Komponenten, d.h. Eigenschaften, die auf tieferen Modellebenen verwendet werden, unter diesem Löser keiner Verteilung folgen dürfen. Soll LIN2 verwendet werden, so ist also darauf zu achten, dass aufgrund dessen die Realisierung von Experimentserien nur sehr eingeschränkt möglich ist.

1.2.9 Verbesserungsmöglichkeiten der Analyse

In diesem Abschnitt werden Ansätze zur weitergehenden Beschäftigung mit dem Themenbereich angegeben. Diese beziehen sich auf eine bessere Ausnutzung der Darstellungsmöglichkeiten der UML, von HIT oder auf weitere Analysemöglichkeiten. Bezüglich möglicher Änderungen oder Erweiterungen der Implementierung von HIPE beachte man zusätzlich Abschnitt

[13] Dies stellt eigentlich eine "Überoptimierung" dar, da diese Abänderung eigentlich nur in Fällen notwendig wäre, in denen mehrere Möglichkeiten einer Fortsetzung bestehen. Dann darf keiner der Ausführungen eine Wahrscheinlichkeit von 1 bzw. 0 zugeordnet werden. Diese Einschränkung ist unseres Wissens nicht im HISLANG Reference Manual beschrieben. Im Kapitel 4.1.1 ab der Seite 123 wird darauf noch einmal Bezug genommen.

5.7.2 im Handbuch. Darin werden diejenigen Aspekte des bestehenden Paradigmas aufgezählt, die von der Gruppe nicht praktisch umgesetzt wurden. Begründungen dafür können sein,

- dass jene Eigenschaften auch über die Wunschkriterien des Pflichtenheftes hinausgehen,
- als Wunschkriterien so komplex sind, dass eine Realisierung im Rahmen der vorhandenen Zeit von den Mitgliedern nicht geleistet werden konnte,
- oder von der Gruppe nicht zu verantwortende Restriktionen die Umsetzung verhindert haben.

Die einzelnen Aspekte überlappen teilweise, so dass im Zuge einer Erweiterung und Verbesserung von HIPE auch darüber nachgedacht werden kann, dazu in Beziehung stehende Aspekte ebenfalls zu betrachten. Aus den Ausführungen dürfte sich weiterhin ergeben, dass auch diese Sammlung von Möglichkeiten der Bewertung und Optimierung von Softwaremodellen nicht als erschöpfend zu verstehen ist.

Bei der Analyse verwendete Interaktionsdiagramme

Es wurde bereits mehrfach erwähnt, dass alle Interaktionsdiagramme, das heißt Aktivitäts- und Sequenzdiagramme, bewertet werden könnten. In HIPE wurde jedoch nur die Auswertung der aussagekräftigeren Aktivitätsdiagramme implementiert. Von der Implementierung der Sequenzdiagramme wurde abgesehen, da erstens in UML 2.0 die Sequenzdiagramme nicht mehr als Spezialfall der Aktivitätsdiagramme zu betrachten sind (vgl. [Jeckle (2004)]) und sich zweitens auch ihre Darstellung in XMI erheblich von der Darstellung der Aktivitätsdiagramme unterscheidet.

Verwendung von Aggregationen

Da bei der Definition von Experimentserien das Kreuzprodukt über alle möglichen Parametersetzungen gebildet wird, steigt der Analyseaufwand exponentiell. Um die Laufzeit (auch für Auswertungen ohne Experimentserie) möglichst gering zu halten, wäre es sinnvoll, extensiv das von HIT angebotene `AGGREGATE`-Statement zu nutzen, um Bereiche, innerhalb derer kein Maß erfaßt werden soll, vorab ein einziges Mal zu analysieren. Dabei würde eine Aggregation erzeugt werden, eine Komponente, die nach außen hin dasselbe Verhalten zeigt wie die ursprüngliche Komponente oder Modellhierarchie. Diese Komponente besitzt aber keine Innensicht mehr. Falls also innerhalb einer Komponente oder Modellhierarchie eine Experimentserie definiert wird, ist eine Aggregation nicht möglich, da das nach außen dargestellte Verhalten der Komponente durch die im Zuge dieser Experimentserie zu variierenden Werte bestimmt wird. Gegebenenfalls müsste dafür die Struktur des hierarchischen Modells an die zu erhebenden Maße angepasst werden, das heißt neben der logischen Blockung durch `LOOP`- und `CONCURRENT`-Bereiche würde noch eine Blockung dahingehend vorgenommen werden, dass Objekte, an denen keine Messung vorgenommen werden soll, in selbstdefinierte Komponenten eingeordnet werden, während Objekte, an denen zu messen ist, möglichst allein stehen sollten, bzw. die Hierarchie möglichst flach gehalten werden sollte.

Einsatz des Load Filterung

Das Load Filtering von HIT erlaubt es, für eine Komponente dediziert festzustellen, wie stark bestimmte Aufrufe sich auf diese auswirken. HIPE übersetzt Ausführungen von Aktivitäten auf Aufrufe von Komponenten. Mit Hilfe des Load Filtering könnte also zum Beispiel festgestellt werden, welche Auswirkungen auf die Performance des Systems der Aufruf einer bestimmten Methode innerhalb eines bestimmten Kontextes hat.

Benutzung und Semantik von Experimentserien

Bei der Benutzung von Experimentserien gibt es Einschränkungen, die aus Eigenschaften von HIT und der Konstruktion von Experimentserien in HIPE resultieren. Einerseits gibt es unter der gewählten Implementierung der Aktivitätsdiagramme als `OPEN_CHAIN`s die Einschränkung, dass innerhalb einer Hierarchie, die vom obersten Aktivitätsdiagramm (also dem, das einen Anwendungsfall verfeinert) aufgerufen wird, nur jeweils eine Experimentserie erlaubt ist. Andererseits wird, falls innerhalb eines Modells die Werte für mehrere Eigenschaften variiert werden sollen, das Kreuzprodukt über alle möglichen Setzungen der einzelnen Eigenschaften gebildet. Beides sind Eigenschaften, die vom Nutzer nicht unbedingt erwünscht sind. Gegebenenfalls sollen z.B. auf einer tieferen Ebene mehrere Werte variiert, im Rahmen einer Experimentserie nur das Kreuzprodukt über die innerhalb eines Diagramms oder Blockes zu variierenden Werte gebildet oder bestimmte Werte für einzelne Eigenschaften nur unter der Bedingung eingesetzt werden, dass andere Eigenschaften auf bestimmte Werte gesetzt sind.

Darstellung der Rekursion bei Aktivitätsdiagrammen

Mit `let diagram` bzw. durch Freilassen der Annotation bei gleichzeitiger Anwesenheit eines Aktivitätsdiagramms mit gleichem Namen kann ein Aktivitätsdiagramm verwiesen werden, womit bei der Umsetzung Rekursionen auftreten können. Diese können irrtümlich auftreten, aber auch vom Nutzer intendiert sein. Da HIT rekursive Modelle nicht analysieren kann, wird in HIPE deren Konstruktion verhindert, indem der zweite Aufruf einer verwiesenen Komponente innerhalb eines Hierarchiepfades nicht erlaubt und dieser zweite Aufruf auf eine atomare Komponente abgebildet wird. Rekursive Strukturen können also zum Abschlusszeitpunkt der Projektgruppe nicht modelliert werden. Techniken zur Entrekursivierung könnten eingesetzt werden, um rekursive Modelle in von HIT analysierbare hierarchische Modelle umzusetzen, und rekursiven Aufrufen Eigenschaften wie die Rekursionstiefe zugeordnet werden.

Auswirkungen von Polymorphismen und Sichtbarkeiten

In Interaktionsdiagrammen, d.h. in Aktivitäts- und Sequenzdiagrammen, können zum Beispiel abstrakte Klassen als handelnde bzw. im Rahmen von Objektfluss erzeugte oder konsumierte Entitäten erscheinen, wobei in der Realität entsprechend eine abgeleitete Klasse die dargestellte Funktionalität erfüllen soll. Es kann in einem Softwareentwurf mehrere abgeleitete Klassen geben, die sich performancerelevant unterscheiden. Das wird von HIPE nicht berücksichtigt. Eine Bewertung in HIPE erfolgt bisher unter der Annahme, dass ein Objekt der in einem zu übersetzenden Diagramm eingesetzten Klasse auch die handelnde bzw. behandelte Entität ist. Weiterhin wird von HIPE nicht die Sichtbarkeit von Objekten oder Methoden überprüft, die - insbesondere im Hinblick auf die Vererbung - ebenfalls eine Eigenschaft ist, die die Bewertung eines Softwareentwurfs beeinflusst (vgl. jeweils [Hoeben (2000)]).

Modellierung paralleler Systeme

Multiprozessorsysteme und Parallelverarbeitung in Programmen sind in den letzten Jahrzehnten immer wichtiger geworden, um Anwendungen mit hohen Erfordernissen an die Rechenleistung zu realisieren. Multiprozessorsysteme besitzen eine Anzahl gleichartiger Prozessoren, auf denen Software parallel ausgeführt werden kann. Dabei wird die Leistung der Prozessoren nicht direkt weitergegeben, sondern es treten Kommunikationsoverheads auf und es werden nicht-parallelisierbare Codebereiche ausgeführt, so dass die letztendlich realisierte Laufzeit der Software auf n Prozessoren signifikant größer ist als $1/n$ der Laufzeit auf einem Einprozessorsystem. Im Sinne des Performance Engineering und auch unter ökonomischen Gesichtspunkten ist besonders das Maß interessant, um das sich durch die Benutzung eines derartigen Rechnersystems die Performance einer Anwendung beschleunigt (vgl. [Clement und Quinn (1993)]). Das Gespann HIT/HIPE ist zur Modellierung und Bewertung solcher paralleler Abläufe allerdings nur teilweise geeignet. Verschiedene Aspekte müssten eingehend behandelt werden, damit eine Betrachtung paralleler Systeme möglich ist.

Syntax und Semantik paralleler Abläufe. Parallele Systeme unterscheiden sich von sequentiellen dadurch, dass die Komplexität möglicher Interaktionen zwischen Tasks exponentiell steigt und sich entsprechend komplexere Programmodelle ergeben, die analysiert werden müssen (vgl. [Jonkers (1995)],[Grove (2003)]). In der im Abschnitt 1.2.6.2 ab Seite 56 beschriebenen Form zur Darstellung von Nebenläufigkeiten können nur sogenannte fork-join-Programme dargestellt werden, das heißt Programme mit einander abwechselnden seriellen und parallelen Bereichen, wobei die verschiedenen parallelen Tasks voneinander unabhängig ablaufen und nur abschließend einmal miteinander synchronisieren (vgl. [Adve und Vernon (2004)]). Dies stellt eine Einschränkung dar, da in vielen Fällen parallel ablaufende Tasks auch zwischenzeitlich miteinander interagieren, indem sie etwa Nachrichten versenden oder gemeinsame Ressourcen nutzen, wodurch sie sich gegenseitig in ihrer Ausführung beeinflussen. Diese Interaktionen und Beeinflussungen sind performancerelevant (vgl. [Jonkers (1995)]), müssen also erfasst werden, damit eine adäquate Vorhersage der Performance möglich ist. Es müssen folglich eine Syntax und Semantik definiert werden, mit der in von HIPE zu verarbeitenden Softwaremodellen solche Interaktionen dargestellt werden können.

Auswertung nebenläufiger Programme. Gegen eine Auswertung mittels Simulation sprechen unter Umständen die zu erwartende Laufzeit oder ihr primär empirischer Charakter, der im gegebenen Fall die Möglichkeiten zur Interpretation der Ergebnisse einschränken kann. Entsprechend wird eine analytische Auswertung vorgezogen (vgl. [Leung (1988)],[Clement und Quinn (1993)]). Nebenläufige Bereiche können in HIT jedoch nur mit dem simulativen Löser ausgewertet werden, ein analytisches Modell fehlt. In der Literatur finden sich verschiedene Typen von analytischen Modellen, auf deren Grundlage versucht wird, Aussagen über die zu erwartende Performance paralleler Programme abzuleiten. Die bisher vorgeschlagenen Modelle sind aber auf verschiedene Arten eingeschränkt, so dass sie nur beschränkt anwendbar sind (vgl. [Adve und Vernon (2004)]). Pezzè et al. kommentieren, dass die inhärente Komplexität nebenläufiger Programme kaum erwarten lasse, dass ein einzelnes Modell eine für alle möglichen Programme akzeptable Kombination von Aufwand und Genauigkeit darstellen wird (vgl. [Pezzè u. a. (1995)].

Statische Modelle, z.B. [Pezzè u. a. (1995)], können dabei helfen, bestimmte formale Aspekte paralleler Programme zu überprüfen, etwa die Freiheit von Deadlocks und das Ausbleiben unerlaubter Interaktionen zwischen Tasks. Diese Analysen sind leicht automatisierbar, da keine Annotationen seitens des Nutzers notwendig sind. Sie sind allerdings auf wenige Aspekte beschränkt, wobei auch für das Eintreten dieser Eigenschaften nur das Vorhandensein notwendiger, nicht aber hinreichender Bedingungen überprüft werden kann. Weiterhin können die performancerelevanten Kosten, die durch die Belegung von Ressourcen entstehen, nur schlecht abgeschätzt werden (vgl. [Jonkers (1995)]).

Qualitative bzw. parametrische Analysemodelle sind die einfachsten Modelle zur Betrachtung der Performance paralleler Systeme. Amdahl's Law (vgl. [Amdahl (1967)]) und Erweiterungen davon (z.B. [Clement und Quinn (1993)]) erlauben eine Vorhersage der letztendlich realisierten Parallelität. Dazu müssen zum Beispiel der nicht-parallelisierbare Anteil der Software und die Auswirkungen der Architektur des Mehrprozessorsystems bestimmt werden. Diese Modelle liefern allerdings nur qualitative Aussagen über die zu erwartende Performance und Skalierbarkeit von Programmen, nicht aber detailliertere Angaben etwa über kritische Stellen der Anwendung, wie schlecht parallelisierbare Abschnitte, die Auswirkung der Zuordnung der einzelnen Aktivitäten auf CPUs oder die Auswirkung der Belegung von Ressourcen (vgl. [Adve und Vernon (2004)]).

In quantitativen Analysemodellen wird versucht, die performancerelevanten Aspekte zu identifizieren und in soweit darzustellen, dass eine adäquate Vorhersage der Performance möglich wird und die Quellen von Performanceproblemen identifiziert werden können. Diese Modelle können einerseits anhand der durch sie bewertbaren Programme und andererseits gemäß ihrer Annahmen über die Arbeitslast, die Antwortzeiten der beteiligten Aktivitäten und die von der ausführenden Hardware verwendeten Scheduling-Disziplinen unterschieden werden. Je komplexer die Anforderungen sind, was einerseits die zugelassenen Parametersetzungen in Bezug auf die genannten Aspekte und andererseits die Größe des Modells angeht, das bewertet werden soll, desto problematischer wird aber die Auswertung. Die bestehenden Modelle sind auf eine oder mehrere Arten aus der folgenden Liste eingeschränkt (vgl. [Adve und Vernon (2004)]):

- Exponentielle Komplexität. Modelle, die Scheduling, Zwischenankunftszeiten und Ausführungsdauern genau abbilden können, zum Beispiel auf Basis von Markovketten, sind so komplex, dass sie nicht zur Auswertung von Programmen praxisrelevanter Größe benutzt werden können.

- Beschränkung der Anwendbarkeit auf bestimmte Typen von Programmen. Zum Beispiel nehmen Modelle für fork/join-Programme voneinander unabhängige Tasks an, so dass Kommunikationskosten zwischen Tasks nicht betrachtet werden können, und die durch sie generierte Abschätzungen der Leistungsmaße für allgemeine Programme stark fehlerhaft sein können.

- Beschränkung auf eine geringe Anzahl an Scheduling-Disziplinen und/oder bestimmte Verteilungen von Zwischenankunftszeiten von Anfragen an das System und/oder Bedienzeitverteilungen der einzelnen Aktivitäten. Je nachdem, wie gut die Annahmen punktuell mit den tatsächlichen Gegebenheiten übereinstimmen, führen solche einschränkenden

Annahmen zu über das Modell betrachtet sehr inkonsistenten Abweichungen der geschätzten und der in der lauffähigen Anwendung schließlich realisierten Leistungswerte.

Üblicherweise müssen bei der Entscheidung für ein Analysemodell also verschiedene Tradeoffs betrachtet werden, zum Beispiel zwischen der angestrebten Genauigkeit und der in Anspruch genommenen Rechenzeit-/Speicherkomplexität oder zwischen einer bestimmten erleichternden Einschränkung und einer daraus folgenden anderen Einschränkung für die Analyse.

Als Beispiel für die komplexen Auswirkungen einer konkreten Entwurfsentscheidung seien die so genannten deterministischen Modelle angeführt: Die Modelle von Adve und Vernon (vgl. [Adve und Vernon (2004)]) und Jonkers (vgl. [Jonkers (1995)]) erlauben für Bedienzeitanforderungen der Aktivitäten nur mehr deterministische Werte, während stochastische Größen nur noch zur Darstellung von Aktivitäten des Rechnersystems und auch dort nur sehr eingeschränkt verwendet werden. Diese Modelle sind effizient lösbar und liefern auch für komplexe Programme genaue Aussagen. Für ihre Benutzung müssen allerdings nichtdeterministische Effekte eliminiert werden, wie sie etwa durch Verzweigungen im Kontrollfluss von Programmen entstehen. Die Analyse erfolgt also auf Basis von Durchläufen, die jeweils nur einen einzigen Ablauf ohne Verzweigungen darstellen, wobei jeder Durchlauf das Verhalten der Anwendung auf einer Klasse von Eingaben repräsentiert. Diese müssen dann gewichtet werden. Wegen der Vielzahl der möglichen Klassen von Eingaben eines komplexen Programms ist dies ggf. unpraktikabel (vgl. [Mason (2002)]). Aus dieser Problematik ergibt sich der weitere Kritikpunkt, dass die Ausführungsdauern aller Aktivitäten in einer Software für jede Klasse möglicher Eingaben einzeln angegeben oder erschlossen werden müssten. Dies stellt eine zusätzliche Erschwernis dar, da solche Daten dann üblicherweise noch nicht verfügbar sind. Als Alternative kann im Rahmen eines "semiempirischen" Verfahrens (vgl. [Xu u. a. (1996)]) das Verhalten der Anwendung der Anwendung für alle Durchläufe anhand von Angaben über ihr Verhalten über einige wenige Durchläufe extrapoliert werden. Dieses Vorgehen kann allerdings zu großen Fehlern bei der Abschätzung führen. Die Angabe einer Verteilung für die Antwortzeiten, deren Parameter zum Beispiel anhand von Erfahrungswerten geschätzt werden können, wäre also wesentlich einfacher und i.a. weniger fehlerbehaftet. Deterministische Modelle können weiterhin nichtdeterministischer Effekte bzw. die Auswirkungen von Veränderungen nichtdeterministischer Größen nicht vorhersagen, z.B. den Effekt der Übertragung der Anwendung auf ein Zielsystem mit anderen Leistungsparametern (vgl. [Adve und Vernon (2004)]) oder die sich in Message-Passing-Systemen ergebende nichtdeterministische Anordnung von Nachrichten (vgl. [Grove (2003)]). Besonders der letztere Faktor kann immer noch nur simulativ erfasst werden.

Bewertung des Zusammenspiels von Anwendung und Rechnersystem. In Rechnersystemen können im Zusammenhang mit der Parallelarbeit aus verschiedenen Gründen Performanceprobleme auftreten. Es seien beispielhaft fünf konkrete Probleme genannt, in denen das Zusammenspiel von Anwendungen und vom System zur Verfügung gestellten Ressourcen eine Rolle spielt:

- Leichtgewichtige Prozesse, so genannte Threads, werden bei der parallelen Programmierung benutzt, um eine gegenüber Prozessen "feinere" Nebenläufigkeit zu modellieren. Dabei werden mehrere Threads einem Prozess zugeordnet und erhalten folglich auch Ressourcen gemeinsam, wodurch sie sich gegenseitig blockieren können (vgl. [Tanenbaum und Woodhull (1997)] und [Oxenrider (2005)]).

- In parallelen Systemen ist ein gängiges Optimierungsziel eine hohe Auslastung der bereitgestellten Ressourcen. Ist der Task-Scheduler des betrachteten Systems in der Lage, Leerlaufzeiten, die etwa eintreten, wenn alle Tasks einer aktuellen Periode bereits abgelaufen sind, durch Vorziehen von Ausführungen zu überbrücken (vgl. [Espinosa u. a. (1998)]), oder können Abläufe im Programm so formuliert werden, dass Leerlaufzeiten nicht entstehen, so verbessert sich dadurch die Auslastung des parallelen Systems.
- Eine Steigerung der Auslastung führt aber nicht immer zu einer Verbesserung der Antwortzeit. Insbesondere ist eine Zerlegung eines zu bewertenden Algorithmus gesucht, so dass sich gleichzeitig zu einer hohen Auslastung eine möglichst gute Beschleunigung ergibt (vgl. [Vrsalovic u. a. (1988)]).
- Zugriffe auf den Speicher und Berechnungen können ggf. überlappt, d.h. parallelisiert werden, wodurch sich Performancegewinne ergeben können. Um – unter der Annahme, dass die Architektur zum Beispiel nichtblockierende Speicherzugriffe und geeignete Scheduling-Strategien bereitstellt – das Potential für diese Parallelisierungen zu erschließen, muss z.B. überprüft werden, ob der Softwareentwurf geeignet strukturiert ist (vgl. [Cameron und Sun (2003)]).
- In der Regel wird beim Software Performance Engineering angenommen, dass die in Entwicklung befindliche Anwendung das Rechnersystem allein zur Verfügung hat oder zumindest doch durch den Scheduler die Beeinflussung durch andere Anwendungen minimiert wird. Dies führt ggf. zu einer Fehleinschätzung der Synchronisationskosten (vgl. [Adve und Vernon (2004)]). Eine Spezifikation der bereits auf einem System bestehenden Arbeitslast ist allerdings abhängig von der Zielplattform vielleicht nicht einmal möglich (vgl. [Horgan u. a. (1995)]).

Betrachtung von Realzeitsystemen

Nicht in allen Fällen sind stochastische Aussagen über die Erfüllung von Performancezielen einer Software ausreichend. In sicherheitskritischen Systemen werden vielmehr Garantien dafür gefordert, dass z.B. Deadlines eingehalten werden, da die bei Verletzung dieser Deadlines auftretenden Konsequenzen unter Umständen nicht zu akzeptieren sind (vgl. [Marwedel (2003a)]). Eine Adaption wurde bereits durchgeführt, indem in HIPE für die Betrachtung der Instanziierung von Objekten die Jamaica-Semantik verwendet wird, die eine Kalkulation des Maximalaufwands für die Speicherverwaltung ermöglicht. Es wäre zu fragen, welche weiteren Möglichkeiten es noch gibt, die dargestellten Lösungsansätze für den gewählten Problemkreis der stochastischen Leistungsbewertung auf die neue Situation zu adaptieren und zu erweitern. Dazu gehören zum Beispiel die Ermittlung der worst-case Ausführungszeit von (Meta-)Aktivitäten, der Belegung von Ressourcen durch Prozesse und der Belegung von Speichern durch Daten. Weiter ist zu fragen, von welcher Qualität und Sicherheit die gemachten Abschätzungen sind. Sehr sichere, aber auch gleichzeitig sehr konservative Schätzungen gehen davon aus, dass die maximalen Kosten, die durch Ressourcenbelegung, Kommunikation etc. entstehen, bis zu zwei Größenordnungen über den mittleren Kosten liegen (vgl. [Grove (2003)]). Dadurch wird aber im allgemeinen Fall die Performance der bewerteten Anwendung zu schlecht eingeschätzt.

Simulation passiver Ressourcen

Verschiedene weitergehende und in bestimmten Anwendungen performancerelevante Eigenschaften passiver Ressourcen werden in HIPE nicht betrachtet:

- Die Belegung von Speichern (`memory`) durch bereits vorhandene Objekte hat eine Auswirkung auf den für neue Objekte zur Verfügung stehenden Speicherplatz. Werden also Daten in einem Speicher abgelegt, muss diese Belegung bei künftigen Zugriffen auf diesen Speicher berücksichtigt werden.
- Festplatten und ähnliche Datenspeicher können an sie gerichtete Aufträge gemäß verschiedener Scheduling-Disziplinen abarbeiten. Diese und die Verteilung von Daten auf den Datenträgern ist ggf. performancerelevant (vgl. [Baumgartl (2004)]).
- RAMs können unterschiedlich strukturiert, etwa als Ein- oder Mehrportspeicher ausgeführt sein (vgl. [Marwedel (2003b)].
- In Speicherhierarchien greifen verschiedene passive Ressourcen (etwa First und Second Level Cache, RAM und Festplatte) ineinander. Eigenschaften wie die Speicherverwaltung der Maschine und des Betriebssystems, auf der eine Software abläuft (vgl. [Marwedel (2003b)]), die Hit-Rate (vgl. [Brown (1995)], [Chung u. a. (1994)]) oder verschiedene Austauschstrategien für Inhalte der Speicher, die die Hierarchie ausmachen (vgl. [Marwedel (2003b)]), wirken sich jeweils wesentlich auf die Zugriffszeiten auf Daten aus.
- Die Simulation von Leitungen kann um Scheduling-Disziplinen ergänzt werden. In Abhängigkeit vom verwendeten Protokoll oder Netztyp können Leitungen zum Beispiel durch unterschiedliche Kombinationen von `schedule`- und `dispatch`-Parametern dargestellt werden (vgl. [Martinka (1995)]).

Darstellung von Netzwerktopologien

Eine Einschränkung der Implementierung von HIPE ist die Notwendigkeit, logische Leitungen zu modellieren, das heißt zwischen allen CPUs, zwischen denen eine Kommunikation (hier nur durch Objektflüsse realisiert) erfolgen soll, muss im Verteilungsdiagramm eine Leitung existieren. Dies ist, abhängig von der in der realen Umgebung verwendeten Netzwerktopologie, im allgemeinen nicht der Fall. Stattdessen werden transportierte Objekte über die physikalischen Leitungen im Netzwerk transportiert, wobei die Zwischenkomponenten für das Routing zuständig sind. Dabei müssen verschiedene weitere performancerelevante Eigenschaften des Netzwerks berücksichtigt werden, z.B. verschiedene mögliche Routing-Strategien oder Protokolle, die beim Datentransport verwendet werden. Auf Basis dieser Angaben kann dann ein hierarchisches Modell erstellt werden (vgl. [Jonkers (1995)]), das passend in das schon bestehende hierarchische Modell der Software eingegliedert werden kann.

Komplexitätsmaße für Methoden und Klassen

Zur Abschätzung der Laufzeitkomplexität von Methoden wird die $CM(m)$-Metrik von Carbone und Santucci verwendet, für die Größe von Klassen im Speicher die $CP(c)$-Metrik. Die ab der Seite 36 dargestellten Anpassungen dieser Maße, mit denen die Einbindung von Klassen in Abläufe des entworfenen Softwaresystems und ihre internen Abläufe in die Komplexitätsabschätzungen mit einbezogen werden können, wurden nicht in HIPE implementiert. Werden diese umgesetzt, können die letztendlichen Komplexitäten der einzelnen Klassen bis auf 10%

genau abgeschätzt werden. Weiterhin können möglicherweise bessere bzw. für andere Analysen geeignetere Abschätzungen entwickelt werden, indem verschiedene selbst vorgenommene und aus der Literatur entnommene empirische Auswertungen miteinander kombiniert werden.

Integrative Betrachtung verschiedener Qualitätsmaßstäbe für Software

Insbesondere interessant ist die gemeinsame Betrachtung verschiedener Qualitätsmaßstäbe für Software. In der Praxis existieren zum Beispiel Trade-offs zwischen der Performance und der Zuverlässigkeit (Reliability) von Software (vgl. [Goseva-Popstojanova und Trivedi (2000)]). Solche Trade-offs bestehen allerdings auch in ökonomischer Hinsicht, da die Optimierung der Performance einer Anwendung ggf. an anderen Stellen in einer Anwendung ansetzt als die Optimierung der Zuverlässigkeit: Aus Performance-Sicht mögen sich die kritischen Stellen für die Performance einer Anwendung als die Codeblöcke definieren, die am häufigsten ausgeführt werden (vgl. [Oxenrider (2005)]), die kritischen Stellen für die Zuverlässigkeit aber als die Codeblöcke, die nur selten ausgeführt werden (vgl. [Horgan u. a. (1995)]). Ziel eines "Performability Engineering", das Performance und Reliability Engineering miteinander kombiniert, ist eine sinnvolle Balance zwischen diesen beiden verschiedenen Qualitätszielen, ggf. auch unter den Kostenzielen. Diese Betrachtung kann um weitere Maßstäbe wie den Energieverbrauch oder um Teilbereiche wie die Genauigkeit von Berechnungen erweitert werden, wobei weitere Trade-offs berücksichtigt werden müssen.

Modellierung der Zuverlässigkeit von Software

Als Beispiel für einen weiteren Qualitätsmaßstab von Software wird die Zuverlässigkeit genannt. Diese ist ein Maß für die Erfüllung gestellter Anforderungen an ein System. Dazu gehört die Performance, d.h. die Erfüllung bestimmter Leistungsziele, aber auch die Fehlerfreiheit des Systems, gemessen daran, wie wahrscheinlich es ist, dass ein System unter einer vorgegebenen Arbeitslast von seiner spezifizierten Funktionalität abweicht. Die Fehlerfreiheit hängt maßgeblich von der Anzahl und Intensität der Fehler im System ab. In der Literatur werden verschiedene Modelle diskutiert, wie anhand von Angaben über eine Software die Fehler in dieser Software erschlossen werden können. Dabei können einerseits nach Beendigung der Entwicklungsphase Angaben über auftretende Fehler gesammelt, andererseits aber auch schon während der Entwicklungsphase Angaben aus dem Entwurf der Software und Eigenschaften des Entwicklungsprozesses betrachtet werden.

Ein exemplarisches Vorgehen, mit dem die Komplexität und Fehlerfreiheit einer Software schon im Entwurfsstadium abgeschätzt werden kann, besteht aus drei Stufen.

1. *Ermittlung der Größe einer Software.* Zunächst wird die Größe einer Software in einer nachvollziehbaren und auch für die Modellierung der Zuverlässigkeit geeigneten Komplexitätsmetrik erzeugt. Die in das HIPE-System bereits integrierte Komplexitätsmetrik von Carbone und Santucci liefert eine Abschätzung der Komplexität einer Implementation in Lines-of-Code (LOC), das heißt in effektiven Anweisungen.

2. *Ermittlung der Fehleranzahl bzw. -dichte.* Im zweiten Schritt werden aus den Angaben über die Komplexität der Software bzw. einzelner Komponenten Informationen über die Anzahl potentieller Fehler darin erzeugt. Aus der Empirie sind zum Beispiel Schätzwerte

für die Anzahl von Fehlern pro Codezeile bekannt [Fenton und Neil (September/October 1999)].

3. *Ermittlung der Zuverlässigkeit.* Aus den Angaben über die Fehleranzahl bzw. -dichte werden Abschätzungen entwickelt, wie wahrscheinlich das Auftreten eines Fehlers ist, der zum Versagen eines Systems führt. Auch für diesen Schritt gibt es empirisch ermittelte Formeln, mit denen die Fehlerrate z.B. in Abhängigkeit von der Fehlerdichte, der Geschwindigkeit und der Systemlaufzeit abgeschätzt werden kann.

Wird ein solches oder anderes Analysemodell implementiert, muss natürlich überprüft werden, ob die Schätzungen über die Zuverlässigkeit, die dieses Modell liefert, qualitativ mit den von der Software letztlich realisierten Zuverlässigkeit übereinstimmen.

Das HIPE-System müsste zur Realisierung einer Zuverlässigkeitsanalyse weiterhin um Möglichkeiten erweitert werden, die mutmaßliche Zuverlässigkeit eines Systems darzustellen. Das oben dargestellte Modell geht davon aus, dass aus der Komplexität von Ausführungen die Wahrscheinlichkeit gefolgert werden kann, dass darin Fehler auftreten. Mit dieser Wahrscheinlichkeit würde also aus einer konkreten Aktivität in einen Fehlerzustand übergegangen. Die Gesamtzuverlässigkeit R des Systems ergibt sich als $R = 1 - \pi_{Err}$, wobei π_{Err} die Wahrscheinlichkeit ist, dass im System während der Ausführung ein Fehler auftritt, der zu seinem Versagen führt.

Betrachtung programmiersprachlicher Elemente
- Reverse Engineering und Referenzimplementierungen

In UML werden nur selten wirklich detailliert alle Algorithmen einer Software modelliert. Dies ist auch kaum möglich, da die UML nicht formal genug definiert[14] ist und die Möglichkeit zur Einbindung programmiersprachlicher Elemente fehlt. In manchen Fällen kann allerdings eine Betrachtung von Software auf der Ebene der Algorithmen interessant sein, insbesondere beim Reverse Engineering, also bei der Analyse vorliegender Software (vgl. [Oxenrider (2005)]), und während der Entwicklung von Software, wenn eventuell vorliegende Referenzimplementierungen auf ihre Adaptierbarkeit für eine bestimmte Einsatzumgebung bewertet werden sollen (vgl. [Marwedel (2003a)]). Dafür einsetzbar ist zum Beispiel die sogenannte CPS-Transformation. Diese ist eine Methode, mit der Programmcode als Kontrollflussgraph ähnlich den Aktivitätsdiagrammen dargestellt werden kann (vgl. [Mason (2002)]). Dabei wird jeder Codeblock, der ohne den Aufruf einer anderen Methode auskommt, als eine Aktivität dargestellt. Jede Aktivität stellt einen Zustand einer Markovkette dar. Diese Darstellung ermöglicht verschiedene Analysen:

- Beim Reverse Engineering kann dem Nutzer die problematische und ggf. fehlerbehaftete Aufgabe abgenommen werden, die Bedienzeitanforderungen der einzelnen ausgeführten Aktivitäten anzugeben.
- Aus Angaben über die einzelnen Aktivitäten und die möglichen Abläufe des dargestellten

[14] HIPE kann zum Beispiel nur eine kleine Teilmenge der möglichen UML-Diagramme umsetzen, die bestimmten Aufbauregeln folgen müssen. Die UML selbst ist freier definiert, um Entwicklern den Entwurf von Software zu erleichtern, da diese sich über die genaue Semantik bestimmter Abläufe keine Gedanken machen müssen.

Algorithmus können durch kontrollflußbasierte Analysen die erwartete und auch die worst-case Laufzeit eines Programms hergeleitet werden.

- Zum Zwecke einer Zuverlässigkeitsbewertung der Software kann den einzelnen Aktivitäten eine Komplexität und damit eine Fehlerwahrscheinlichkeit zugeordnet werden, die sich aus der Analyse des Programmcodes ergibt.
- Profiling-Techniken ermöglichen eine Identifikation der gemäß der betrachteten Qualitätsmaße kritischen Stellen im Algorithmus.

Benutzung verschiedener Performancetools

Im Zwischenbericht war im Rahmen der Anforderungsdefinition bereits angeregt worden, das HIPE-Tool so offen zu gestalten, dass es prinzipiell auch als Frontend für andere Performancetools dienen kann. Denkbar sind Umsetzungen in das Performance Model Interchange Format [Smith und Llado (2004)] oder in die Sprache des SHARPE-Tools [Sahner und Trivedi (1992)]. Die Benutzung des SHARPE-Tools ist dabei besonders interessant, da auf verschiedenen Modellebenen verschiedene Modellwelten und Analysetechniken zum Einsatz kommen können. Es kann also ggf. für jedes Experiment, zum Beispiel jede zu erfassende Größe, eine adäquate Darstellung ausgewählt werden. Besonders sinnvoll ist dies auch, falls verschiedene Qualitätsmaßstäbe von Software betrachtet werden sollen.

Betrachtung von Antipatterns

In HIPE wird von den drei möglichen Vorgehensweisen des Performance Engineering nur der Bereich des Modelling betrachtet. Auch möglich ist die Betrachtung von Antipatterns, das heißt der Versuch, in Softwareentwürfen Entwurfsmuster zu entdecken, die möglicherweise eine negative Wirkung auf die Performance der so realisierten Systeme haben. Beispiele für Antipatterns sind die "Gott"-Klasse, die einen Großteil der Arbeit eines Systems erledigt, oder die "Rampe", die symbolisiert, dass die Antwortzeit einer Aktivität mit zunehmender Laufzeit des Systems immer größer wird. Weitere Beispiele für Antipatterns finden sich in [Smith und Williams (2000)], [Smith und Williams (2002)] und [Smith und Williams (2003)].

Optimierung von Softwareentwürfen

Im Sinne des Software Performance Engineering zieht eine schlechte Bewertung der Performanceeigenschaften einer Software idealerweise konkrete Designentscheidungen nach sich, die dazu führen sollen, dass sich die Performance verbessert. Allerdings ist in vielen Fällen nicht ersichtlich, welche Optimierungen einen positiven Effekt haben (vgl. [Oxenrider (2005)]) oder überhaupt praxisgerecht sind (vgl. [Marwedel (2003b)]). Manchmal sind sogar intuitiv gesehen dysfunktionale Eigenschaften zu treffen, etwa die Bedienzeitanforderung einer Aktivität zu erhöhen oder die Parallelität von Abläufen – und damit den Kommunikationsoverhead – zu verringern (vgl. [Espinosa u. a. (1998)]). Im Zuge einer vom Nutzer zu wählenden Optimierungsstrategie kann der Raum der möglichen Entwürfe auf sinnvolle Designentscheidungen überprüft werden, die einen positiven Effekt auf die Performance der Software haben. Es seien hier nur drei Beispiele möglicher Optimierungsstrategien genannt:

- Abhängig von der Struktur einer Software und der Hardware, auf der diese abläuft, gibt es zum Beispiel im Rahmen des sogenannten "Bottlenecking" verschiedene Methoden zur

Behandlung von Flaschenhälsen (vgl. [Tregunno (2003)]). Methoden des "Bottlenecking" sind etwa die Bereitstellung mehrerer Kopien eines Tasks, der einen Flaschenhals des Systems darstellt, oder eine bessere Verteilung von Daten im System.

- Smith und Williams schlagen in ihren Papers über Antipatterns (s.o.) auch Möglichkeiten zu deren Behebung vor. Ist die Semantik von UML-Diagrammen genügend eingeschränkt und hinreichend gut definiert, könnten zum Beispiel automatisch Primitive ergänzt werden, die die Entfernung nicht mehr benötigter Objekte aus einer Liste darstellen.
- Die Bedienzeitanforderungen, die vom Nutzer mittels `responsetime`-Statements angegeben werden, können auch im Sinne einer geforderten Antwortzeit verstanden werden. Aus einer Bibliothek von Hardwarekomponenten kann das kostengünstigste System zusammengestellt werden, in dem diese Antwortzeiten eingehalten werden (vgl. [Henkel u. a. (1993)]).

1.3 Zusammenfassung

In diesem Kapitel wurden zunächst die Ideen beschrieben, die dem Software Performance Engineering zugrundeliegen. Es wurde das Konzept dargestellt, das HIPE zugrundeliegt. Dieses beschreibt, wie Softwaremodelle, die in Form von syntaktisch und semantisch noch sehr eingeschränkten UML-Diagrammen dargestellt werden, um performancerelevante Angaben erweitert, in ein von HIT bewertbares Leistungsmodell transformiert und die Ergebnisse von Bewertungen schließlich dargestellt werden können. Dabei ggf. notwendige Anpassungen, um Modelle auch mit Hilfe analytischer Löser bewerten zu können, sind insbesondere Beschränkungen auf bestimmte Verteilungen für Zwischenankunftszeiten von Anfragen an das System und für Bedienzeitanforderungen von Aktivitäten. Abschließend wurden Aspekte dargestellt, die weiterhin im Rahmen des Software Performance Engineering betrachtet werden können, um potentiell interessante Aspekte von Anwendungen und Systemen besser zu erschließen.

Mit dem dargestellten Paradigma wurde die obere Grenze der Funktionalität festgelegt, die mit dem Projekt HIPE realisiert werden sollte. In den folgenden Kapiteln wird dargestellt, in welcher Form und in welchem Umfang dies geschah.

2 Entwurf

Die Projektgruppe entwickelte in der ersten Phase des Projektes im Sommersemester 2005 einen Entwurf für *HIPE*. Anschließend übernahmen die Gruppenmitglieder, die sich mit den verschiedenen Software-Modulen beschäftigten, deren Implementierung. Der Entwurf dient dazu, einen ersten Überblick über die weitreichenden Softwareanforderungen von *HIPE* zu verschaffen. Während der Entwurfsphase wurden unterschiedliche UML-Diagramme entwickelt und schriftlich dokumentiert. Sie wurden in der Entwicklungsumgebung *Eclipse* mit Hilfe des Zusatzmoduls *EclipseUML* der Firma Omondo (siehe [OMONDO (2002)]) modelliert. Die Dokumentation des Entwurfs ist Gegenstand dieses Kapitels.

Der Entwurf besteht aus folgenden Abschnitten:

- Grobkonzept — Hier wird das Grundgerüst von HIPE sowie die Umwandlung von UML nach JDOM über XMI beschrieben.
- Client/Server — Thema dieses Abschnitts ist das Konzept und das Klassendiagramm des Kommunikationsmoduls von HIPE.
- Beschreibung der Anwendungsfälle — Dieses Unterkapitel enthält die Dokumentation der Anwendungsfälle für HIPE.
- Klassendiagramme — Beinhaltet die übrigen Konzepte und Klassendiagramme.
- Plausibilitätskontrolle — Mit Hilfe von Aktivitäsdiagrammen wird hier die Vorgehensweise der Plausibilitätskontrolle von HIPE beschrieben.
- Weiche Analyse — Enthält die Modellierung und Beschreibung des Konzepts der weichen Analyse.

2.1 Grobkonzept

Nun wird das Grobkonzept von HIPE vorgestellt. Auf alle hier erwähnten Teile von HIPE wird im weiteren Verlauf des Kapitels näher eingegangen. An dieser Stelle soll nur deren Zusammenspiel erklärt werden. Die Abbildung 2.1 auf S. 86 zeigt das Grobkonzept von HIPE, zu dem nun Stellung genommen wird. HIPE erwartet als Eingabe XMI-Dateien. Diese XMI-Dateien werden in ein JDOM-Document-Objekt eingelesen bzw. aus diesem heraus in eine Datei zurückgeschrieben. Die alleinige Repräsentation innerhalb des JDOM ist allerdings für eine Analyse nicht aussagekräftig genug, weshalb eine weitere Datenstruktur, die UML-Repräsentation, benötigt wird. Die UML-Repräsentation wird dann von HIPE analysiert und aus den daraus resultierenden Daten wird ein HI-SLANG-Dokument erzeugt. Dieses dient im anschließenden Aufruf des HIT-Programms als Übergabeparameter. Dazu muss das Dokument über ein Client-Server-Modul erst auf den Rechner gebracht werden, auf dem eine HIT-Installation verfügbar ist. HIT startet daraufhin seine Analyse und erzeugt eine dazugehörige Ausgabe, die

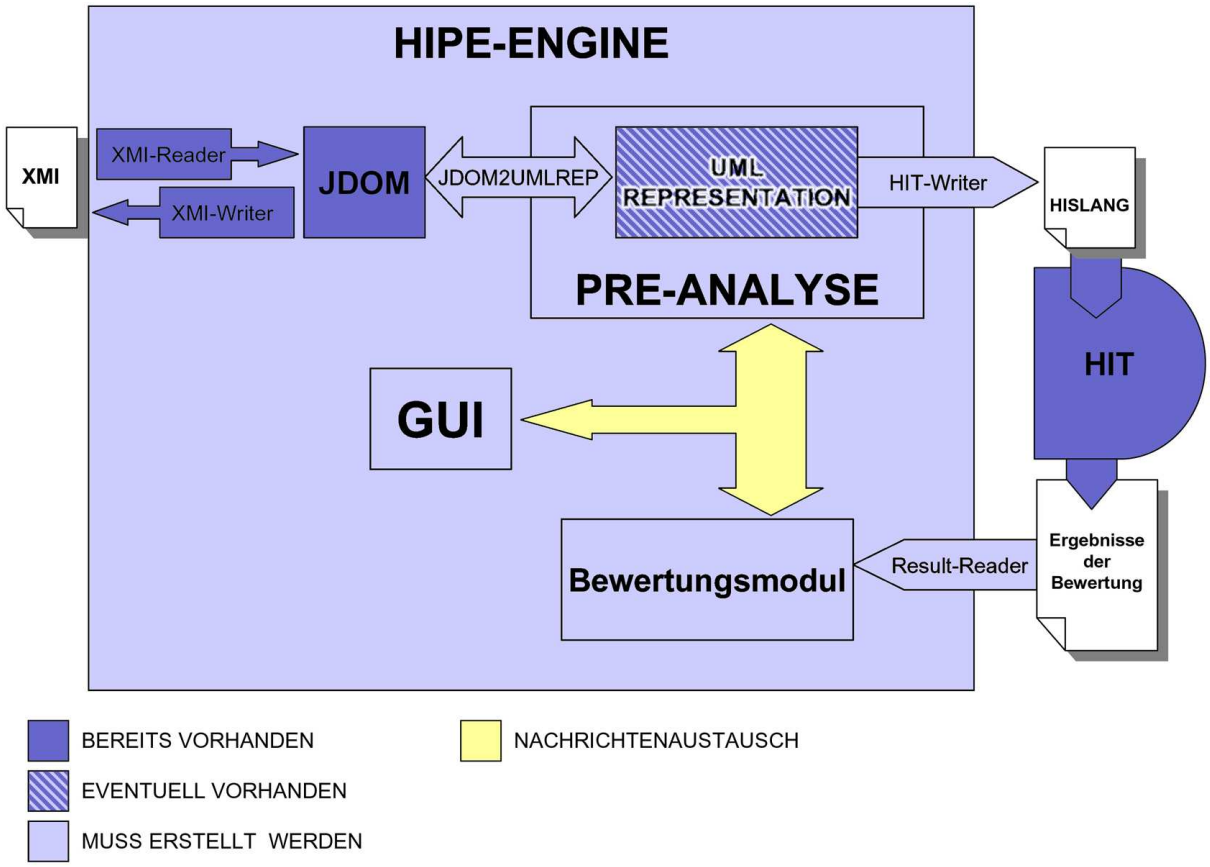

Abbildung 2.1: Das Grobkonzept von HIPE

durch das Client-Server-System von HIPE eingelesen wird. Danach nimmt das Bewertungsmodul seine Arbeit auf und gibt die berechneten Werte graphisch in Form von zweidimensionalen Säulendiagrammen wieder. Die Ergebnisse können nun z.B. durch die GUI angezeigt werden oder aber auch über die UML-Repräsentation und JDOM wieder in das ursprüngliche UML-Modell zurückgeschrieben werden. Dort kann man die Ergebnisse dann direkt am Diagramm ablesen.

2.1.1 XMI- und JDOM-Repräsentation von UML-Modellen

HIPE benötigt als Eingabe ein UML-Modell. Dieses UML-Modell kann nicht von HIPE selbst erstellt werden. Dies geschieht mit der Anwendung METAMILL, welche die UML-Daten im XMI-Format abspeichert (siehe [MetaMill Software (2004)]). XMI definiert ein Austauschformat auf XML-Basis für Metadaten. Im Rahmen der PG wird XMI in der Version 1.2 verwendet. So lassen sich UML-Beschreibungen mittels XMI auf standardisierte Weise exportieren und mit anderen UML-Werkzeugen austauschen. XML ist eine Metasprache und definiert, wie Daten strukturiert in Textdateien gespeichert werden. Es wird zunächst näher hierauf eingegangen um anschliessend wieder auf XMI und dessen Verarbeitung durch HIPE zurückzukommen. XML-Datenstrukturen sind anwendungs- und plattformunabhängig. XML ist als Metasprache erweiterbar und ist ebenso wie HTML und XHTML eine Teilmenge von SGML.

XML 1.0 wurde Anfang 1998 vom W3C als Standard verabschiedet und hat seitdem eine sehr große Verbreitung und Akzeptanz seitens der Industrie erfahren (vgl. [W3.org (2004)]). Das folgende Beispiel ist ein wohlgeformtes XML-Fragment:

```
<?xml version="1.0"?>
<Dialog>
  <Adam Emotion="heftig">ich liebe dich!</Adam>
  <Eva Emotion="heftig">ich dich auch!</Eva>
</Dialog>
```

Wohlgeformt bedeutet in diesem Zusammenhang nicht unbedingt auch inhaltlich korrekt, sondern nur, dass einige wichtige Regeln im Umgang mit XML eingehalten wurden. In Bezug auf die von HIPE zu verarbeitenden XMI-Dateien, denen auch XML zugrunde liegt, muss man ausserdem noch eine Menge anderer Faktoren berücksichtigen. Da in XMI-Dateien UML-Diagramme gespeichert werden sollen, müssen UML-spezifische Besonderheiten beachtet werden. Beispielsweise muss eine Assoziation stets einen Start- und einen Endpunkt besitzen. Es macht an dieser Stelle wenig Sinn, auf den genauen XMI-Standard einzugehen, da sich keines der von uns geprüften Programme — einschliesslich METAMILL — wirklich daran hält. Viele erweitern den Standard um eigene Tags, andere setzen ihn nicht voll um. Jede XMI-Datei hat einen *Header-* und einen *Content-Tag*, wobei der Header nur allgemeine Informationen über die Datei enthält. Der wirklich wichtige Bereich ist der Content-Tag. METAMILL nutzt die Flexibilität von XMI voll aus und erweitert die vorhandene Struktur noch um einen *Extensions-Tag*. Im Content-Block werden die einzelnen Elemente der UML-Diagramme gespeichert. Die Extensions hingegen enthalten ganze Diagramme mit ihren Elementen, wobei bei den Elementen immer auf den Content-Tag verwiesen wird. Mit folgendem Beispiel soll dieser Sachverhalt näher erläutert werden. In Abbildung 2.2 wurde ein einfaches Use-Case-Diagramm konstruiert,

Abbildung 2.2: Das UseCase-Diagramm zum XMI-Quellcode in Abschnitt 2.1.1

welches nur aus einem Actor und einem Use-Case besteht. Hier nun der dazugehörige und auf das Wesentliche gekürzte XMI-Code:

```
<?xml version="1.0" encoding="UTF-8"?>
<XMI xmlns:UML="http://omg.org/UML2.0" xmi.version="1.2">
  <XMI.header>
    <XMI.documentation>
      <XMI.exporter>Metamill</XMI.exporter>
      <XMI.exporterVersion>4.0 (build 691)</XMI.exporterVersion>
      <XMI.shortDescription>Metamill Model (new_model)</XMI.shortDescription>
    </XMI.documentation>
    <XMI.metamodel xmi.name="UML" xmi.version="2.0"/>
  </XMI.header>
  <XMI.content>
    <UML:Model name="new_model" xmi.id="mm:9a2b16ce-e69a">
```

```
    <UML:Namespace.ownedElement>
      <UML:UseCase xmi.id="mm:aa42fbc7-e69a" name="UseCase1" visibility="public">
        <UML:UseCase.extensionPoint></UML:UseCase.extensionPoint>
      </UML:UseCase>
      <UML:Actor xmi.id="mm:b038cf57-e69a" name="Actor1" visibility="public"
       stereotype="actor"/>
      <UML:Association xmi.id="mm:b6c74bbd-e69a" visibility="public">
        <UML:Association.connection>
          <UML:AssociationEnd xmi.id="mm:b6c74bbb-e69a" type="mm:b038cf57-e69a"
           visibility="public"/>
          <UML:AssociationEnd xmi.id="mm:b6c74bbc-e69a" type="mm:aa42fbc7-e69a"
           visibility="public"/>
        </UML:Association.connection>
      </UML:Association>
    </UML:Namespace.ownedElement>
  </UML:Model>
</XMI.content>
<XMI.extensions xmi.extender="Metamill">
  <UML:Diagrams>
    <UML:Diagram name="NewUseCaseDiagram" xmi.id="mm:a7e08da7-e69a"
         kind="Use Case Diagram" owner="mm:9a2b16ce-e69a">
      <UML:Diagram.elements>
        <UML:DiagramElement vcode="VUseCase" xmi.id="mm:aa42fbc5-e69a"
             modelElement="mm:aa42fbc7-e69a">
          <UML:DiagramElement.taggedValue>
            <UML:TaggedValue tag="name" value="UseCase1"/>
          </UML:DiagramElement.taggedValue>
        </UML:DiagramElement>
        <UML:DiagramElement vcode="VActor" xmi.id="mm:b038cf55-e69a"
             modelElement="mm:b038cf57-e69a">
          <UML:DiagramElement.taggedValue>
            <UML:TaggedValue tag="name" value="Actor1"/>
          </UML:DiagramElement.taggedValue>
        </UML:DiagramElement>
      </UML:Diagram.elements>
      <UML:Diagram.lines>
        <UML:DiagramLine vcode="VRelationship" rstype="RSAssociation"
             xmi.id="mm:b6c74bb6-e69a" modelElement="mm:b6c74bbd-e69a">
          <UML:DiagramLine.taggedValue>
            <UML:TaggedValue tag="sourceId" value="mm:b038cf55-e69a"/>
            <UML:TaggedValue tag="targetId" value="mm:aa42fbc5-e69a"/>
          </UML:DiagramLine.taggedValue>
        </UML:DiagramLine>
      </UML:Diagram.lines>
    </UML:Diagram>
  </UML:Diagrams>
  <UML:ImportedElements>
```

```
      <UML:ElementStub xmi.id="actor"/>
    </UML:ImportedElements>
  </XMI.extensions>
</XMI>
```

Dieses noch sehr einfach gehaltene Beispiel verdeutlicht vor allem, dass es ziemlich umständlich werden kann, wenn man versucht, diesen Output auf die herkömmliche Weise von oben nach unten zu parsen. Dabei sollte man sich vor Augen halten, dass es im Normalfall sehr viele Diagramme mit sehr vielen Elementen sein können, die es einzulesen gilt. Diese können zusätzlich auch noch untereinander verbunden sein, so dass ein organisiertes Auslesen nur mit extrem viel Aufwand bewerkstelligt werden kann. Es ist also wünschenswert, die gesamte XMI-Datei im Speicher zu halten. Das vom W3C-Konsortium entwickelte DOM (Document-Object-Model) bietet sich hier als eine von den vielen möglichen Repräsentationen an. DOM in Verbindung mit dem Java-eigenem SAX-Parser wären eine Möglichkeit, diese Datei einzulesen und im Speicher präsent zu halten. Eine wesentlich bessere, da komplett auf Java-Entwickler zugeschnittene Version von DOM ist JDOM. JDOM kann XMI-Dateien einlesen, auf Wohlgeformtheit prüfen, erweitern und speichern. Auch JDOM erstellt einen Dokumentenbaum, der dann wie jeder Baum anhand der Vater-Sohn-Beziehung seiner Knoten traversiert werden kann. Bevor HIPE ein UML-Modell analysiert, muss es erst eine Plausibilitätskontrolle durchlaufen. JDOM kann solche Plausibilitätskontrollen nicht durchführen. Das einzige, was JDOM im Zusammenhang mit HIPE leistet, ist die Konstruktion einer JDOM-Document-Repräsentation der XMI-Datei im Speicher und die Überprüfung auf Wohlgeformtheit. Letzteres betrifft aber nur generelle Regeln zur Syntax. Diese Regeln besagen zum Beispiel, dass alle geöffneten Tags auch geschlossen werden und dass Attribute eindeutig sein müssen. Eine semantische Überprüfung, wie etwa in der Plausibilitätskontrolle gefordert, muss dann durch HIPE selbst erfolgen.

Bei der Benutzung von JDOM arbeitet man hauptsächlich mit zwei Klassen:

- Document — Das ist das Wurzelobjekt (und somit der Einstiegspunkt zur Navigation) des JDOM-Baumes, das das Wurzelelement (root), den Dokumenttyp und Verarbeitungsanweisungen enthält.
- Element — Diese Klasse repräsentiert die Informationen für Elemente (beispielsweise das Elternelement (*parent*), den Elementnamen (*name*) und eine Liste der Attribute (*attributes*)).

Nun folgen ein paar Beispiele, die die Benutzung von JDOM innerhalb von HIPE veranschaulichen sollen. Über `Document.getRootElement()` kann man auf das Wurzelelement zugreifen, mittels `Element.getChild("NameDesKindes")` auf das in `Element` enthaltene, mit der angegebenen Zeichenfolge benannte Unterelement.

```
Element root=uml.getRootElement();
Element extenderKnoten = root.getChild("XMI.extensions");
```

Der Zugriff auf die Gesamtheit aller Unterelemente erfordert ein anderes Vorgehen:

```
  List DiagrammListe=ExtenderKnoten.getChildren("Diagram",XMLUML);
  ListIterator Durchlauf=DiagrammListe.listIterator();
    while (Durchlauf.hasNext()){
```

```
    Element aktuell = (Element)Durchlauf.next();
      if (aktuell.getAttributeValue("kind").
                              equalsIgnoreCase("class diagram")){
      System.out.println("Klassendiagramm gefunden:" +
                              aktuell.getAttributeValue("name"));
      classDiagramList.add(new ClassDiagram(aktuell));
    }
  }
```

Das Quelltextbeispiel zeigt in einer auf das Wesentliche gekürzten Form, wie das Auffinden von Klassendiagrammen implementiert ist. Für Java-Verhältnisse ist das bereits ein sehr komfortabler Zugriff auf eine XML-Datei, aus Sicht der von HIPE durchzuführenden und teilweise recht komplexen Analysen jedoch noch viel zu umständlich und fehleranfällig. Deswegen wurde innerhalb von HIPE eine Datenstruktur geschaffen, die es erlaubt, einfach und bequem auf die UML-Elemente zuzugreifen.

2.1.2 Client/Server

Um die vom HIPE-Tool generierten HI-SLANG-Dateien ausführen zu können, müssen diese zunächst zu einer Solaris-Maschine übertragen werden. Das ist notwendig, da das HIT-Tool momentan nur für Solaris Rechner verfügbar ist. Um diesen Transfer nicht von Hand vornehmen zu müssen, wird auf einer entsprechend ausgerüsteten Maschine ein Server-Modul gestartet. Dieses soll HI-SLANG-Dateien annehmen und sie anschließend an das HIT-Tool bei dessen Ausführung übergeben. Die Ergebnisse können dann vom Klienten angefordert werden. Der momentane Entwurf des Client/Server-Pakets sieht vor, nur die Ergebnisse der harten Analyse zurückzugeben. Denkbar wäre auch, die komplette Projektdatei zu übertragen. Dadurch könnten die Ergebnisse einer Analyse auch auf anderen Klienten, die die Projektdatei nicht besitzen, betrachtet werden.
Der Server ist als Dämon realisiert, der als Hintergrundprozess auf einer Solaris-Maschine ausgeführt wird. Als Parameter bekommt er einen Port übergeben, an dem er auf einkommende Verbindungen horcht. Weiterhin wird ein Arbeitsverzeichnis angegeben, in diesem sollen die übertragenen Dateien und die Ergebnisse der HIT-Analyse gespeichert werden. Nach dem Starten des Servers soll zunächst das Arbeitsverzeichnis nach Dateien durchsucht werden, die noch nicht analysiert wurden. So ist sichergestellt, dass im Falle eines Neustarts des Servers bereits übertragene Dateien noch analysiert werden. Der Server soll dann am angegebenen Port auf einkommende Verbindungen warten, für jede neue Verbindung soll ein neuer Socket generiert werden. Dieser ist dann für die Kommunikation mit dem Klienten verantwortlich. Dadurch wird es möglich, mehrere Klienten gleichzeitig zu bedienen.

Vom Server werden folgende Aufgaben realisiert:

1. **Empfangen einer Datei vom Klienten**
   Die generierten HI-SLANG-Dokumente werden vom Klienten zum Server geschickt und dort im Arbeitsverzeichnis gespeichert. Dabei wird der Dateiname, unter dem die Datei auf dem Server abgespeichert werden soll, vom Server festgelegt. Falls die Übertragung erfolgreich durchgeführt wurde, erhält der Klient eine Bestätigung, um mögliche Fehlerfälle abfangen zu können. Weiterhin wird er darüber informiert, unter welchem Namen

die Datei abgelegt wurde. Anhand dieses Namens kann die Datei später identifiziert werden. Ebenso sind die Ergebnisse der Analyse mit diesem Namen verknüpft und abrufbar. Nachdem die Datei erfolgreich übertragen wurde, wird sie auf dem Server in eine Warteschlange eingereiht. Dies ist notwendig, damit keine Konflikte bei der gleichzeitigen Bedienung mehrerer Klienten auftreten bzw. die Rechenlast durch parallel ausgeführte Analysen zu groß wird. Ein zusätzlicher Thread sorgt dafür, dass die Dateien aus der Warteschlange nach der FCFS-Regel abgearbeitet werden. Dabei muss darauf geachtet werden, dass der Zugriff auf die Warteschlange synchronisiert wird. Nachdem die HIT-Analyse beendet ist, wird das Ergebnis im Arbeits-Ordner gespeichert. Dieses beinhaltet zwei Ergebnis-Dateien, nämlich die von HIT produzierte Dump-Datei und die Ausgabe, die normalerweise auf `stdout` geschrieben wird. Damit die Größe des Arbeitsordners nicht zu schnell anwächst, werden die übrigen während der Analyse erzeugten temporären Dateien gelöscht. Alle angemeldeten Klienten werden über den aktuellen Status der auf dem Server übertragenen Dateien informiert.

2. **Holen einer Liste der analysierten Dateien**
   Am Server kann eine Liste der bereits analysierten Dateien in dessen Arbeitsverzeichnis angefordert werden. Dafür wird vom Klienten ein Kommando an den Server geschickt, das den Vorgang initiiert. Der Server beginnt dann mit dem Übertragen der Liste. Nach erfolgreicher Übertragung wird vom Server eine Bestätigung an den Klienten geschickt, damit der Klient im Falle eines Serverausfalls nicht blockiert. Der Klient wartet nach der Initiierung auf eine positive Bestätigung, sollte diese nicht innerhalb von 20 Sekunden vom Server geschickt worden sein, bricht die Operation mit einer Fehlermeldung ab.

3. **Holen eines Ergebnisses vom Server (Dumpfile)**
   Um sich das Ergebnis einer Analyse vom Server zu holen, wird ähnlich verfahren wie beim Übertragen einer Liste. Der Klient initiiert eine Anfrage und wartet 20 Sekunden auf eine Antwort vom Server. Hierbei wird das Ergebnis beim Klienten allerdings nur im Speicher gehalten, eine Speicherung findet erst später in der Projektdatei statt.

4. **Abfrage des Status einer zum Server geschickten Datei**
   Es ist jederzeit möglich, den Status einer Datei, die zum Server geschickt wurde, abzufragen. Dabei wird zwischen folgenden Zuständen unterschieden:

   a) `InQueue`: Die HI-SLANG-Datei ist in der Warteschlange des Servers.
   b) `CurrentlyAnalyzed`: Die HI-SLANG-Datei wird gerade von HIT analysiert.
   c) `ResultAvailable`: Die HI-SLANG-Datei ist fertig analysiert, die Ergebnisse können abgefragt werden.
   d) `Error`: Der Status der HI-SLANG-Datei ist unbekannt (eine entsprechende Datei bzw. das Dumpfile existiert nicht).

   Diese Abfrage geschieht aktiv. Eine passive Benachrichtigung der Klienten ist ebenfalls möglich. Das heißt, sie werden, solange sie am Server angemeldet sind, über Statusänderungen der übertragenen HI-SLANG-Dateien benachrichtigt. Dabei wird der Dateiname der Datei und der aktuelle Status übertragen.
   Für den aktuellen Status können folgende Zustände angenommen werden:

   a) `PutIntoQueue`: Die HI-SLANG-Datei ist in der Warteschlange des Servers eingereiht worden.

b) `StartedHitAnalysis`: Die HI-SLANG-Datei wird gerade von HIT analysiert.

c) `AnalysisDone`: Die HI-SLANG-Datei ist fertig analysiert, die Ergebnisse können abgefragt werden.

Um diese Aufgaben zu erfüllen, wurden Kommandos spezifiziert, mit denen die Kommunikation zwischen dem Klienten und dem Server stattfindet. Ein Kommando besteht dabei aus zwei Teilen, dem eigentlichen Befehl und einen Parameter. Durch den Parameter wird es möglich, zusätzliche Informationen wie Dateinamen zu übertragen.

- `StartFileTransfer`
  Initiiert einen FileTransfer, der Server wird angewiesen, eine Datei zu öffnen. Als Parameter kann ein Dateiname angegeben werden. Ist der Parameter nicht vorhanden, wird der Dateiname vom Server generiert.

- `FileData`
  Senden von Textstrings, als Parameter wird der zu übertragende Text übergeben. Dieser wird zeilenweise in die zuvor geöffnete Textdatei geschrieben.

- `EndFileTransfer`
  Beendet den Filetransfer, die Datei wird auf dem Server geschlossen und in die Warteschlange eingereiht.

- `SendFileName`
  Informiert den Klienten über den initiierten Dateinamen. Der Dateiname wird als Parameter übergeben.

- `StartListTransfer`
  Initiiert einen Listentransfer; die empfangenen Strings werden in einer Liste gespeichert. Die Liste wird beim Empfang dieses Befehls als neues Objekt angelegt.

- `ListData`
  Senden von Textstrings, als Parameter wird der zu übertragende Text übergeben. Der String wird als neues Element in die zuvor erzeugte Liste eingefügt.

- `EndListTransfer`
  Beendet den Listentransfer.

- `RequestResultList`
  Weist den Server an, eine Liste der auf dem Server liegenden Ergebnisse an den Klienten zu schicken.

- `RequestResult`
  Weist den Server an, ein Ergebnis zum Klienten zu schicken (dump-file).

- `RequestHitOutput`
  Weist den Server an, die Ausgabe einer HIT Analyse zu schicken (stdout file).

- `NotifyCommandOk`
  Kommando zur Bestätigung, dass Operationen ohne Fehler ausgeführt wurden. Beispielsweise sendet der Server, nachdem er aufgefordert wurde, eine neue Datei anzulegen und alles fehlerfrei ausgeführt wurde, ein `NotifyCommandOk`.

- `RequestFileStatus`
  Weist den Server an, den Status einer Datei abzufragen.
- `FileStatus`
  Wird benutzt, um den Status einer Datei zum Klienten zu übertragen.
- `ProjectStatusChanged`
  Der Projektstatus hat sich geändert, als Parameter wird der Dateiname und der Status (InQueue, Analyzed, Finished) übertragen.

Das Klassendiagram des Client/Server-Moduls ist in Abbildung 8.3 auf S. 243 zu sehen.

2.2 Beschreibung der Anwendungsfälle

Im Folgenden werden alle Anwendungsfälle einzeln dokumentiert. Die Abbildung 2.3 auf S. 93 zeigt das Anwendungsfalldiagramm und die daraus resultierenden Anwendungsfälle für HIPE.

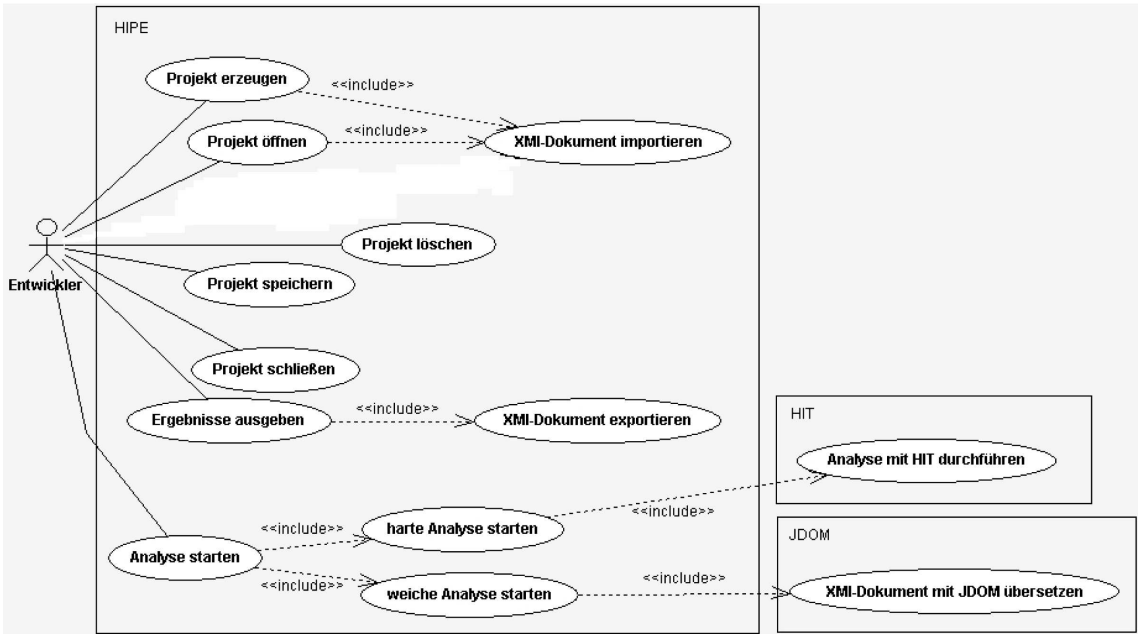

Abbildung 2.3: Anwendungsfalldiagramm für HIPE

2.2.1 Projekt erzeugen

Kurzbeschreibung: Ein Projekt dient innerhalb von HIPE dazu, alle relevanten Daten einer Analyse unter einem einzigen Namen zusammenzufassen (und in einer entsprechend benannten Datei zu speichern). Relevante Daten sind dabei beispielsweise die Einstellungen für die Kommunikation mit dem Server-Modul, die von HIT im Zuge einer Analyse erzeugte Ausgabe und die verwendeten Default-Werte aus der GUI. Eine Projektdatei wird nach ihrer Erzeugung im vom Benutzer angegebenen Verzeichnis als `<Projektname>.hpr` gespeichert. Beim

Erzeugen eines HIPE-Projektes wird die zum Projekt gehörende XMI-Datei importiert und die Plausibilitätskontrolle durchgeführt.
Vorbedingungen: Es ist noch kein Projekt geöffnet. Der Anwender startet die Software und wählt den Menüpunkt „Projekt erzeugen“ aus.
Nachbedingungen: Im Workspace wurde ein Projekt mit dem vom Anwender gegebenen Namen erzeugt. Die XMI Datei wurde auf Fehlerfreiheit und Vollständigkeit überprüft (Plausibilitätskontrolle) und im Projekt importiert.

Fehlersituation:

1. Es existiert ein Projekt mit dem gleichen Namen.
2. Der Projektname enthält Sonderzeichen.
3. Der Projektname enthält keine Zeichen.
4. Die XMI Datei ist nicht vollständig oder ist fehlerhaft.

Nachzustand im Fehlerfall:

1. Es wird kein Projekt erzeugt, eine neue Eingabe wird erwartet.
2. Es wird kein Projekt erzeugt, eine neue Eingabe wird erwartet.
3. Es wird kein Projekt erzeugt, eine neue Eingabe wird erwartet.
4. Fehlermeldung wurde ausgegeben, kein Projekt erzeugt.

Akteure: Anwender

Standardablauf:

1. Der Menüpunkt „Projekt erzeugen“ wird gewählt.
2. Projektverzeichnis wird gewählt.
3. Name des Projekts wird angegeben.
4. Anzeige des Dialogs zur Auswahl der zu importierenden XMI Datei.
5. Die XMI Datei wird vom Benutzer angegeben.
6. Die Plausibilitätskontrolle wird durchgeführt.
7. Plausibilitätskontrolle liefert negatives Ergebnis, gehe nach Anzeige einer Fehlermeldung zu 4.
8. Plausibilitätskontrolle liefert positives Ergebnis, das Projekt ist erzeugt.

2.2.2 Projekt öffnen

Kurzbeschreibung: Der Anwender wählt ein Projektverzeichnis und öffnet ein bestehendes Projekt. Anschließend wird eine Plausibilitätskontrolle durchgeführt. Das Projekt wird geöffnet.
Vorbedingungen: Es ist kein Projekt geöffnet. Fokus des Programms ist im Hauptmenü.
Nachbedingungen: Das Projekt ist geöffnet.
Fehlersituation: Die angegebene Datei ist kein HIPE Projekt, die Plausibilitätskontrolle gibt kein Ergebnis zurück.
Nachzustand im Fehlerfall: Fehlermeldung, der Anwender wird dazu aufgefordert, ein anderes Projekt auszuwählen.
Akteure: Anwender

Standardablauf:

1. Der Menüpunkt „Projekt öffnen“ wird gewählt.
2. Der Anwender wählt ein Projektverzeichnis.
3. Wenn die angegebene Datei kein HIPE Projekt ist gehe zu 2.
4. Die Plausibilitätskontrolle wird durchgeführt.
5. Plausibilitätskontrolle liefert negatives Ergebnis, Anfrage an den Benutzer, welche XMI Datei importiert werden soll.
6. Plausibilitätskontrolle liefert positives Ergebnis, das gewählte Projekt wird geöffnet.

2.2.3 XMI - Dokument importieren

Kurzbeschreibung: Die zu einem Projekt zugehörige XMI-Datei wird aus einem Verzeichnis importiert und dem Projekt zugeordnet. Es folgt die Plausibilitätskontrolle, in der das XMI-Dokument auf Vollständigkeit und Fehlerfreiheit überprüft wird.
Vorbedingungen: Ein XMI-Dokument wird in den Anwendungsfällen „Projekt erzeugen“, „Projekt öffnen“ und „Projekt aktualisieren“ verarbeitet bzw. erstellt. Der Anwender wählt einen der drei oben erwähnten Menüpunkte aus.
Nachbedingungen: Das XMI-Dokument ist im Projekt eingebunden (wird im Projektordner gespeichert).

Fehlersituation:

1. Beim angegebenen Dokument handelt es sich nicht um eine XMI-Datei.
2. Plausibilitätskontrolle liefert negatives Ergebnis.

Nachzustand im Fehlerfall: Neue Eingabe wird erwartet, kein XMI-import durchgeführt.

Akteure: Anwender

Standardablauf: Wird in den Anwendungsfällen „Projekt erzeugen“, „Projekt öffnen“ und „Projekt aktualisieren“, wiederholt beschrieben.

1. Anzeige eines Dialogs zur Auswahl der zu importierenden XMI-Datei.
2. Angabe des Verzeichnisses durch den Benutzer.
3. Anwählen der betreffenden Datei durch den Benutzer.
4. Bestätigung der Auswahl.

2.2.4 Projekt speichern

Kurzbeschreibung: Erfolgt insbesondere nach „Projekt erzeugen“ und nach „Analyse durchführen“. Dabei wird das ganze Projekt und alle dazugehörende Dateien gespeichert. Anschließend gibt es die Möglichkeit, das Projekt zu schließen, die Analyse zu starten, oder das Projekt zu löschen.
Vorbedingungen: Ein Projekt ist geöffnet.
Nachbedingungen: Das Projekt und alle assoziierten Dateien sind in einem ausgewählten Verzeichnis gespeichert.
Fehlersituation: I/O-Exception, Projekt kann nicht gespeichert werden.
Nachzustand im Fehlerfall: Projekt ist im vorigen Zustand, es ist keine Speicherung erfolgt. Der Speichervorgang muss abgebrochen werden.
Akteure: Anwender
Standardablauf: Menüpunkt „Projekt speichern“ wird angewählt und das Projekt wird gespeichert.

2.2.5 Projekt löschen

Kurzbeschreibung: Der Anwender wählt ein Projektverzeichnis und löscht ein bestehendes Projekt.
Vorbedingungen: Der Anwender wählt den Menüpunkt „Projekt löschen“. mindestens ein Projekt ist vorhanden.
Nachbedingungen: Das Projekt und alle dazu gehörende Dateien werden gelöscht.

Fehlersituation:

1. I/O-Exception, Projekt kann nicht gelöscht werden.
2. Das Projekt ist geöffnet.

Nachzustand im Fehlerfall:

1. Aktualisierungsvorgang muss abgebrochen werden.
2. Der Anwender wird dazu aufgefordert, das Projekt zu schließen.

Akteure: Anwender

Standardablauf:

1. Der Menüpunkt „Projekt löschen“ wird gewählt.
2. Der Anwender wählt das entsprechenden Projektverzeichnis.

3. Es wird kontrolliert, ob das Projekt geöffnet ist. Falls nicht, weiter mit 5.
4. Der Anwender wird dazu aufgefordert, das Projekt zu schließen.
5. Projekt wird gelöscht.

2.2.6 Projekt schließen

Kurzbeschreibung: Der Anwender beendet seine Arbeit und schließt das aktuell geführte Projekt. Falls dieses noch nicht gespeichert wurde, wird der Anwender gefragt, ob das Projekt gespeichert werden soll.
Vorbedingungen: Der Anwender wählt den Menüpunkt „Projekt schließen“.
Nachbedingungen: Das Projekt ist geschlossen. Anschließend kann ein neues Projekt geöffnet werden.

Fehlersituation:

1. Das Projekt wurde noch nicht gespeichert.
2. I/O-Exception, Projekt kann nicht gespeichert werden.

Nachzustand im Fehlerfall:

1. Der Anwender wird dazu aufgefordert, das Projekt abzuspeichern.
2. Speichervorgang muss abgebrochen werden.

Akteure: Anwender

Standardablauf:

1. Der Menüpunkt „Projekt schließen“ wird gewählt.
2. Es wird überprüft, ob das Projekt gespeichert ist.
3. Wenn ja, gehe zu 6.
4. Wenn nicht, wird der Anwender dazu aufgefordert, das Projekt abzuspeichern. Wird diese Aufforderung mit ”nein” beantwortet, weiter mit 6.
5. Das Projekt wird gespeichert.
6. Das Projekt wird geschlossen.

2.2.7 Analyse starten

Kurzbeschreibung: Der Anwender startet die Analyse, um eine Bewertung durchzuführen. Als nächstes werden die weiche Analyse und die harte Analyse durchgeführt. Nach der Analyse werden die Ergebnisse in der Projektdatei gespeichert.
Vorbedingungen: Der Anwender wählt den Menüpunkt „Analyse starten“.
Nachbedingungen: Der XMI-Datei wurde analysiert und die Ergebnisse im Projekt gespeichert.

Fehlersituation:

1. I/O-Exception, Analyse kann nicht durchgeführt werden.

Nachzustand im Fehlerfall:

1. Analyse muss abgebrochen werden.

Akteure: Anwender, HIPE, JDOM, HIT

Standardablauf:

1. Der Menüpunkt „Analyse starten“ wird ausgewählt.
2. Die XMI-Datei wird mittels JDOM in einen Java-Baum übersetzt.
3. Die daraus resultierende Datei wird in HIPE importiert.
4. Die weiche Analyse wird durchgeführt (Anwendungsfall "weiche Analyse starten").
5. Ergebnisse werden in der Projektdatei gespeichert.
6. Die Eingabedaten werden zu HI-SLANG-Sourcecode übersetzt und an HIT übergeben.
7. Die harte Analyse wird durchgeführt (Anwendungsfall "harte Analyse starten").
8. Die Ergebnisse der harten Analyse werden gespeichert.

2.2.8 Weiche Analyse starten

Kurzbeschreibung: In der weichen Analyse werden die weichen Leistungsmaße berechnet. Das XMI-Dokument wird mit JDOM in einen Baum übersetzt und HIPE übergeben. Dann werden die weichen Leistungsmaße berechnet und gespeichert.
Vorbedingungen: Der Menüpunkt „Analyse starten“ wurde gewählt, das Eingabe-UML-Modell liegt in einer Baumstruktur aus Java-Objekten vor.
Nachbedingungen: Die Analyse wurde durchgeführt und die Ergebnisse gespeichert.
Fehlersituation: I/O-Exception, Analyse kann nicht durchgeführt werden.
Nachzustand im Fehlerfall: Analyse muss abgebrochen werden.
Akteure: Anwender

Standardablauf:

1. Der Menüpunkt „Analyse starten“ wird ausgewählt.
2. Die XMI-Datei wird nach JDOM übergeben und in einen Java-Baum übersetzt.
3. Weiche Analyse wird durchgeführt.
4. Ergebnisse werden in der Projektdatei gespeichert.

2.2.9 Harte Analyse starten

Kurzbeschreibung: Die harte Analyse, also die Bestimmung der HIT-spezifischen (harten) Leistungsmaße, findet statt. Die Projektdatei wird in HI-SLANG übersetzt, anschließend nach HIT übergeben und die Analyse mit HIT durchgeführt.
Vorbedingungen: Die weiche Analyse ist erfolgreich durchgeführt worden.
Nachbedingungen: Die Bestimmung der HIT-spezifischen Leistungsmaße wurde durchgeführt und gespeichert.
Fehlersituation: I/O-Exception, Analyse kann nicht durchgeführt werden.
Nachzustand im Fehlerfall: Analyse muss abgebrochen werden.
Akteure: Anwender

Standardablauf:

1. Die weiche Analyse wird durchgeführt.
2. Die Eingabedaten werden in HI-SLANG übersetzt und nach HIT übergeben.
3. HIT wird gestartet und die harten Leistungsmaße ermittelt.
4. Die Ergebnisse der harten Analyse werden gespeichert.

2.2.10 XMI-Dokument in JDOM übersetzten

Kurzbeschreibung: Das XMI-Dokument wird nach JDOM exportiert und in eine Java-Baumstruktur übersetzt um für die nachfolgende weiche Analyse zur Verfügung zu stehen.
Vorbedingungen: Der Menüpunkt „Analyse starten“ wurde gewählt.
Nachbedingungen: Die erstellte Baumstruktur ist im Speicher verfügbar.
Fehlersituation: I/O-Exception, Analyse kann nicht durchgeführt werden.
Nachzustand im Fehlerfall: Analyse muss abgebrochen werden.

Akteure: JDOM

Standardablauf:

1. Der Menüpunkt „Analyse starten“ wurde gewählt.
2. JDOM wird gestartet.
3. Die XMI-Datei wird an JDOM übergeben und übersetzt.
4. Die daraus resultierende Baumstruktur wird im Speicher gehalten.

2.2.11 Analyse mit HIT durchführen

Kurzbeschreibung: Das Programm HIT wird gestartet. Die Bewertung mittels HIT wird durchgeführt. Die Ergebnisse liegen danach in einem Dumpfile vor.
Vorbedingungen: Die weiche Analyse wurde erfolgreich durchgeführt und die Eingabedaten wurden fehlerfrei in HI-SLANG-Sourcecode übersetzt. HIT wurde mit dem erzeugten Code als Eingabe gestartet.
Nachbedingungen: Die Ergebnisse der Analyse durch HIT liegen in Form eines Dumpfiles

vor.
Fehlersituation: I/O-Exception, Analyse kann nicht durchgeführt werden.
Nachzustand im Fehlerfall: Analyse muss abgebrochen werden.

Akteure: HIT, HIPE.

Standardablauf:

1. HIT wird gestartet und die harten Leistungsmaße ermittelt.
2. Die Ergebnisse aus der harten Analyse werden Projekt gespeichert.

2.2.12 Ergebnisse ausgeben

Kurzbeschreibung: Die Ergebnisse aus der weichen und harten Analyse werden für den Entwickler aufbereitet und ausgegeben. Dies geschieht sowohl in Form einer Tabelle in der Log-Datei als auch graphisch. Zusätzlich werden sie im XMI-Dokument an die entsprechenden Stellen zurückgeschrieben und sind somit auch im ursprünglichen UML-Diagramm sichtbar. Für jedes einzelne UML-Diagrammelement sind die Ergebnisse in der graphischen Benutzeroberfläche im Strukturbaum auswählbar.
Vorbedingungen: Die Analyse wurde erfolgreich durchgeführt, im Strukturbaum sind die Einträge bezüglich der Leistungsmaße verfügbar.
Nachbedingungen: Dem Anwender liegen die Ergebnisse der Analyse in geeigneter Darstellung vor.
Fehlersituation:

1. Die harte Analyse ist fehlerhaft verlaufen.
2. I/O-Exception, die Ergebnisse können nicht ausgegeben werden.

Nachzustand im Fehlerfall:

1. Es kann kein Ergebnis verglichen und ausgegeben werden.
2. Der Vorgang muss abgebrochen werden.

Akteure: Anwender, HIPE

Standardablauf:

1. Im Strukturbaum wird ein Eintrag bezüglich eines Leistungsmaßes ausgewählt.
2. Die Ergebnisse werden aufbereitet und ausgegeben (Aufgabe des Bewertungsmoduls).

2.2.13 XMI-Dokument exportieren

Kurzbeschreibung: Die Ergebnisse der Analyse sollen im UML-Diagramm angezeigt werden. Das XMI-Dokument muss für diesen Zweck aus HIPE exportiert werden. Danach kann das UML-Tool gestartet und die XMI-Datei geöffnet werden.
Vorbedingungen: Die Analyse wurde erfolgreich durchgeführt.

Nachbedingungen: Die Ergebnisse der Analyse sind in der XMI-Datei eingetragen. Die Datei kann im UML-Tool geöffnet werden.

Fehlersituation:

1. I/O-Exception, die Datei kann nicht exportiert und im UML-Tool angezeigt werden.

Nachzustand im Fehlerfall:

1. Vorgang wird abgebrochen.

Akteure: Anwender, HIPE.

Standardablauf:

1. Die Ergebnisse der harten Analyse werden an die entsprechenden Stellen der XMI-Datei geschrieben.
2. Das UML-Tool wird gestartet.
3. Die XMI-Datei wird vom UML-Tool importiert und das UML-Diagramm inklusive der Ergebnisse geöffnet.

2.3 Klassendiagramme

Ausgehend von HIPE´s Use-Case-Diagramm wurde mit dem Erstellen des Klassendiagramms begonnen. Die Abbildung 8.1 auf S. 241 zeigt das Haupt-Klassendiagramm. Die Klasse Project bildet das Gerüst für die Anwendungsfälle *Projekt erstellen*, *schliessen* und *öffnen*. Aus ihr wird später das Objekt erzeugt, welches dann den Projektraum von HIPE darstellen wird. Die Klasse XMIDocument ist ein *Container* für JDOM, welches wiederum verwendet wird, um die XMI-Datei strukturiert in den Speicher einzulesen.

Die Struktur der Daten innerhalb einer XMI1.2-Datei führt dazu, dass JDOM, welches diese lediglich eins zu eins auf eine Repräsentation aus Java-Objekten abbildet, nicht ausreicht, um Analysen durchzuführen. Struktur bekommt das Ganze erst durch die Klasse *Modell*. *Modell* enthält vier Listen, welche die vier bewertbaren Diagrammtypen *Klassen-*, *Verteilungs-*, *Use-Case-* und *Aktivitätsdiagramm* in sich aufnehmen können. An diesem Punkt muss man erklären, dass eine Struktur geschaffen wurde, anhand derer es möglich ist, sowohl schnellen als auch flexiblen und einfachen Zugriff auf die JDOM-Repräsentation zu erlangen. Die Struktur baut sich um die Klasse *ModelElement* auf (siehe Abbildung 8.2 auf S. 242). Sie enthält eine Referenz auf das zu dem jeweiligen Objekt gehörende JDOM-Element. Da alle anderen Klassen der Struktur auch von *ModelElement* erben, besitzen alle einen Verweis auf ihr zugehöriges Element. Direkte Nachfolger von *ModelElement* sind *DiagramElement* und *Diagram*. *Diagram* ist der Vater der vier bewertbaren Hauptdiagramme. Die Diagramme verfügen über Listen, gefüllt mit ihren jeweiligen *DiagramElements*. *DiagramElement* ist von *ModelElement* abgeleitet und wird letztendlich in verschiedenen Ausprägungen verwendet, beispielsweise in Form einer Instanz der Klasse *UseCase*. Eine Instanz der Klasse *ClassDiagram* repräsentiert entsprechend ein komplettes Klassendiagramm und verfügt z.B. über eine Liste *classesList*. In ihr werden *Class*-Objekte gespeichert. Diese wiederum erben von *DiagramElement*. Sie besitzen

daher wie alle *DiagramElement*-Derivate eine *ConnectionList*, in der sämtliche Assoziationen jeder Art gespeichert werden. Die Assoziationen erben alle vom Objekt *Connection*, welches auch ein Kind von *DiagramElement* ist. Dies können im Beispiel des Klassendiagramms die *Generalisation*, die *Aggregation*, die *Composition* oder die *Association* sein. Bei den Use-Case-Elementen wären das z.B. *Extend* und *Include*. Diese Struktur gewährleistet, dass man den JDOM-Baum nur einmal parsen muss, nämlich um die Struktur zu erzeugen. Danach hat man auf sämtliche relevanten UML-Elemente wahlfreien, schnellen Zugriff. Und zwar so, wie der Entwickler dies später braucht, um seine Analyse-Klasse möglichst kompakt und übersichtlich zu gestalten.

Der Rest des Programms besteht aus der gerade genannten Analyse-Klasse *Analyzer*, der Bewertungs-Klasse *EvaluationEngine*, der Klasse, die Verbindung mit dem Server aufnimmt, genannt *TcpClientSocket*, sowie dem Hauptprogramm *HIPE*. Der *Analyzer* enthält die Methoden, die notwendig sind, um die Plausibilitätsprüfung durchzuführen und die weiche bzw. harte Analyse anzustoßen. Die harte Analyse muss zwangsläufig mit dem *TcpClientSocket* kooperieren, da sie den erzeugten HI-SLANG-Code auf den Rechner mit der HIT-Installation transportieren muss. Der *TcpClientSocket* nimmt nach Vollendung der Analyse die Ergebnisse vom Server entgegen und füllt damit ein *Result*-Objekt, welches dann in der *ResultList* des Projektes landet. Im Anschluss daran kann die *EvaluationEngine* die Ergebnisse anwendergerecht aufbereiten.

2.4 Plausibilitätskontrolle

Nach dem Einlesen der XMI-Datei, wird als erstes die Plausibilitätskontrolle durchgeführt. Sie dient dazu, frühzeitig grobe Design-Fehler zu erkennen und dem Softwaremodellierer zu melden. Sie sorgt dafür, dass nur bewertbare Modelle betrachtet werden. Die Plausibilitätskontrolle wird grob in folgende Kontroll-Prozesse unterteilt:

- Als erstes werden alle vier Diagrammlisten durchsucht. Wenn eine Diagrammliste leer ist, wird eine Fehlermeldung ausgegeben. Hintergrund dieses Kriteriums ist die Tatsache, dass HIPE nur dann aussagekräftige Ergebnisse erzielen kann, wenn alle im Szenario betrachteten Diagramme vorhanden sind (siehe auch Abschnitt 1.2.1).

- Als nächstes wird der Objektfluss im Aktivitätsdiagramm geprüft (siehe Abbildung 2.4 auf S. 103). Die Liste der Aktivitäten wird durchsucht, um alle Verbindungen mit Objektflüssen zu überprüfen. Wenn ein Objektfluss nicht mit einer Aktivität, sondern mit einem Entscheidungs- oder Startknoten verbunden ist, wird ein Fehler ausgegeben. Weiterhin wird die Verbindung zwischen zwei Aktivitäten geprüft. Wenn diese Verbindung nur durch einen Objektfluss verwirklicht ist, kann man das Aktivitätsdiagramm nicht parsen und es wird wieder eine Fehlermeldung ausgegeben.

- Weiter muss der Objektfluss (wie in Abbildung 2.5 auf S. 103 gezeigt) auf Konsistenz mit dem Verteilungsdiagramm geprüft werden. Es wird eine Fehlermeldung erzeugt, wenn die Klasse, die zu einem im Objektfluss auftauchenden Objekt gehört, nicht im Verteilungsdiagramm gefunden wird.

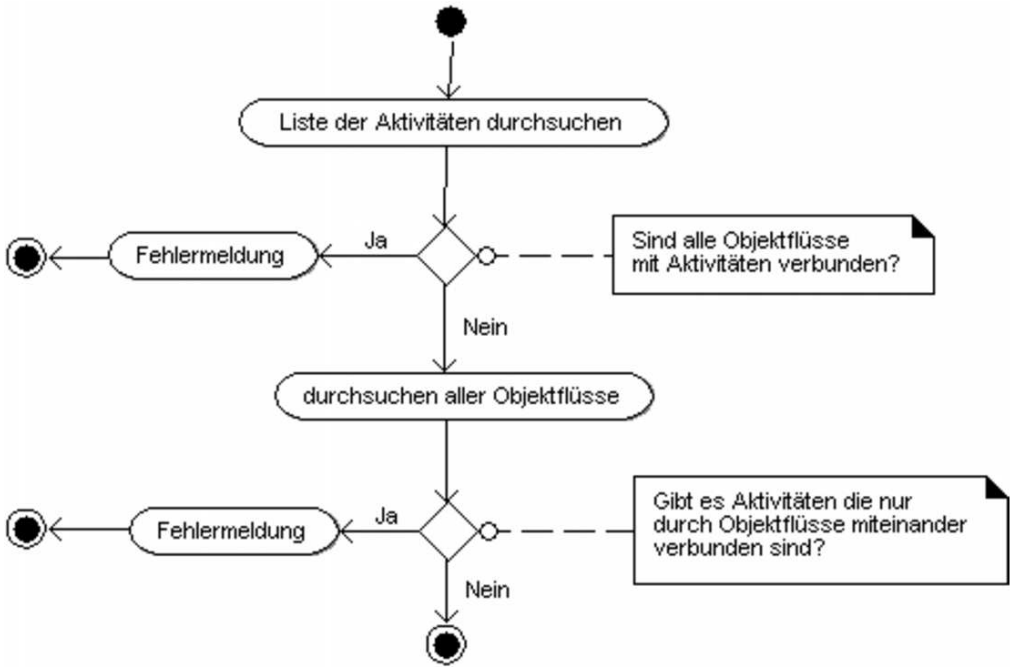

Abbildung 2.4: Prüfung des Objektflusses

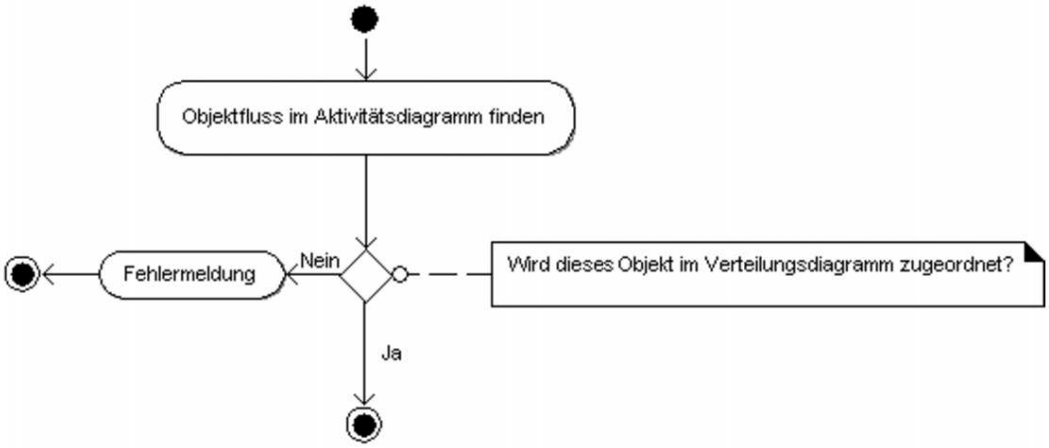

Abbildung 2.5: Die Objektfluss-Verifizierung

- Abbildung 2.6 auf S. 104 zeigt den nächsten Schritt der Plausibilitätsprüfung. Hier wird nach verbotenen Sprüngen in Aktivitätsdiagrammen gesucht, also zum Beispiel eine Verbindung zwischen zwei Aktivitäten, von denen sich eine innerhalb einer Sequenz befindet, die parallel zu einer weiteren Aktivitätssequenz verläuft. (Im Paradigma Unterkapitel 1.2.6.2 wurde ein solches Konstrukt als Concurrent-Bereich beschrieben.)

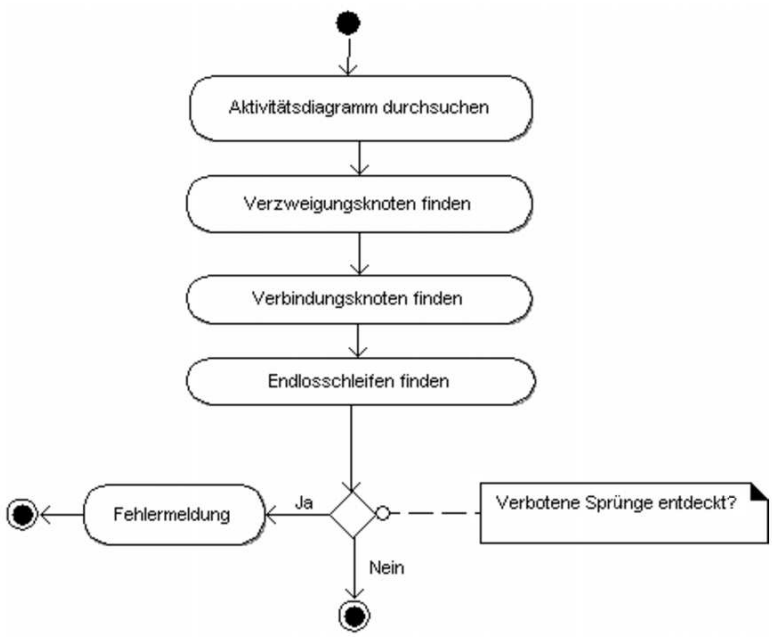

Abbildung 2.6: Suche nach verbotenen Sprüngen in Aktivitätsdiagrammen

2.5 Weiche Analyse

Nach der Plausibilitätskontrolle wird die weiche Analyse (siehe auch Abschnitt 1.2.5) durchgeführt. Sie dient dazu, die für die harte Analyse benötigten Größen (insbesondere den WAC- und WMC-Index) zu ermitteln. Es werden die folgenden weichen Leistungsmaße nacheinander untersucht:

- DEPTH OF INHERITANCE (DIP) (= Anzahl Elternklassen)
  Je tiefer eine Klasse im Vererbungsbaum angesiedelt ist, desto mehr Methoden erbt sie, d.h. sie wird komplexer. Der empfohlene Wertebereich ist [0,5].

- NUMBER OF CHILDREN (NOC) (= Anzahl direkter Kindklassen)
  Je mehr Kinder eine Klasse hat, desto größer ist ihr potentieller Einfluss auf das Systemverhalten. Empfehlung: NOC $\in [0, 10]$.

- NUMBER OF TYPES (ebenfalls NOT) (= Anzahl Klassen und Interfaces)
  Je größer der NOT-Index, desto größer ist der potentielle Nachrichtenverkehr zwischen den Klassen eines Programms. Dieses Leistungsmaß ist analog für abstrakte, exportierte und konkrete Klassen etc. definiert. Empfehlung: NOT $\in [0, 80]$.

- Vererbungskriterien für Klassenhierarchien bzw. -bibliotheken

  - METHOD INHERITANCE FACTOR (MIF)
    Gibt das Verhältnis der Anzahl vererbter Methoden zur Anzahl aller Methoden an. Empfohlener Wertebereich ist]0,25;0,37[
  - ATTRIBUTE INHERITANCE FACTOR (AIF)
    Gibt das Verhältnis der Anzahl vererbter Attribute zur Anzahl aller Attribute an.

- Information Hiding für Klassen
  - METHOD HIDING FACTOR (MHF)
    Gibt das Verhältnis der Anzahl versteckter (nicht nach außen sichtbarer) Methoden zur Anzahl aller Methoden der betrachteten Klasse an.
  - ATTRIBUTE HIDING FACTOR (AHF)
    Gibt das analoge Verhältnis bezüglich der Klassenattribute an.
- POLYMORPHISMUSFAKTOR (PF)

$$PF = \sum \frac{\sum M_0(C_i)}{M_d(C_i)DC(C_i)}$$

  Dabei ist $M_d(C_i)$ die Gesamtanzahl an Methoden, $DC(C_i)$ die Anzahl aller Subklassen und $M_0(C_i)$ die Anzahl derjenigen Methoden einer Klasse C_i, die in mindestens einer ihrer Subklassen überschrieben werden und in C_i neu definiert werden. Für $M_0(C_i)$ kommen also nur diejenigen Methoden in Betracht, die C_i nicht von einer Elternklasse geerbt hat.
- KOPPLUNGSFAKTOR (COF)

$$COF = \sum \frac{\sum isClient(C_i, C_j)}{TC^2 - TC}$$

  Dabei bestimmt die Funktion isClient die jeweilige Beziehungsform und TC steht für die Gesamtzahl aller Klassen. Empfehlung: $COF < 0,52$.
- ERMITTLUNG DER LAST
  Abschließend werden für einzelne Klassen c deren *Complexity Points* berechnet, die genaue Beschreibung findet man in Kapitel 1.2.5.

3 Implementierung

Die Aufgaben der Implementierung von HIPE wurden mit Hilfe einer Paketstruktur organisiert. Es wurden die fünf Java-Pakete `clientServer`, `concept`, `gui`, `paradigma` und `umlInterface` angelegt, die die unterschiedlichen Funktionen von HIPE zur Verfügung stellen. Unter `gui` sind dabei alle Klassen der Benutzeroberfläche zusammengefasst. Sie wurden von dem für das Projekt verwendeten GUI-Builder *Jigloo* (siehe [Kinnersley (2005)]) automatisch erzeugt und werden nicht weiter erläutert. Es wurden lediglich die Teile der einzelnen ActionListener bearbeitet, um die Bedienungsabläufe zu erstellen und um die GUI an die Klassen des `concept`-Pakets anzubinden. Zusätzlich wurden Pakete von Drittanbietern eingebunden. Namentlich sind dies die Pakete des oben genannten JDOM-Projekts sowie die des Open-Source-Pakets JCharts. Letzteres ist eine Klassenbibliothek für Java zur Darstellung von statistischen Graphen und Tabellen. Die vier verbleibenden Pakete der Java-Klassen für HIPE werden im Folgenden näher beschrieben. Man beachte, dass diese Beschreibung nur Anhaltspunkte für die Ideen der Programmierung bietet. Deren Details sind in den entprechenden JavaDocs nachzulesen.

3.1 Paket `clientServer`:

Die im Paket `clientServer` angesiedelten Klassen realisieren die Kommunikation zwischen dem Rechner, auf dem ein HIPE-Client läuft, und dem System, das die HIT-Executable beheimatet.

- `TCPSocket`
  Die Klasse `TCPSocket` enthält alle grundlegenden Funktionalitäten, die sowohl beim Klienten als auch beim Server vorhanden sein müssen. Dazu zählen das Senden und Empfangen von Nachrichten über einen Socket. `TCPClientSocket` und `TCPServerSocket` erben von dieser Klasse.

- `TCPClientSocket`
  Die Klasse `TCPClientSocket` bildet die Schnittstelle nach außen. Sie enthält alle wichtigen Methoden, die benötigt werden, um Anfragen an den Server zu stellen.

- `TCPServerSocket`
  Die Klasse `TCPServerSocket` realisiert einen Server-Socket. Für jeden Klienten, der eine Verbindung herstellt, wird ein neuer Server-Socket erzeugt, der diesen bedient.

- `HislangFileAnalyser`
  Die Klasse `HislangFileAnalyser` verwaltet die HI-SLANG-Files, die vom Klienten zum Server geschickt wurden. Die empfangenen Dateien werden in eine Warteschlange eingereiht und der Reihe nach (FCFS) abgearbeitet.

- `ClientMessageReceiver`
  Die abstrakte Klasse `ClientMessageReceiver` muss implementiert werden, um über

den Status einer Analyse am Server informiert zu werden. Sie informiert über den Status und den Namen der HI-SLANG-Datei. Nachdem man ein ClientMessageReceiver Objekt erzeugt hat, muss dieses noch beim TcpClientSocket registiert werden. Dieses geschieht über die Methode TcpClientSocket.addClientMessageReceiver.

- `HIPEServer`
  Die Klasse `HIPEServer` enthält die `main`-Methode des Servers.

3.2 Paket `concept`:

Das Paket `concept` enthält die Klassen, die als Schnittstelle zwischen der Benutzeroberfläche und dem eigentlichen Programmkern dienen. Sie stellen eine grobe Sicht auf das Konzept dar.

- `Analyzer`
  In der Klasse `Analyzer` sind sämtliche Methoden enthalten, die Analysevorgänge anstoßen. Dadurch ist sie ein wichtiger Einsprungpunkt für alle Analysen. Diese Klasse dient zum Anstoßen der Plausibilitätskontrolle sowie der weichen und harten Analyse. Die Ergebnisse der weichen Analyse werden in die Klasse `softMetrics` eingetragen. Die gesamte Logik der weichen Analyse befindet sich in `Analyzer`.

- `Capsul`
  `Capsul` ist die Persistenz-Container-Klasse von HIPE. Es ist das Objekt, welches von HIPE gespeichert oder geladen wird und sämtliche Informationen bereitstellt, die benötigt werden, um daraus dann ein `Project`-Objekt zu erzeugen. Es enthält die String-Variablen `name`, `filename`, `xmifilename` sowie Objekte aus der Klasse `Datastorage` des `paradigma`-Packages. Diese Variablen entsprechen den gleichnamigen in der Klasse `Project`. Capsul enthält keine Methoden. Die Methoden, die mit Capsul-Objekten arbeiten, sind loadProject und saveProject und befinden sich in der Klasse `HIPE`.

- `EvaluationEngine`
  Diese Klasse ist für die grafische Ausgabe von HIPE zuständig. Sie kapselt Methoden zum einen für die weiche Analyse und zum anderen für das Auslesen der von der harten Analyse generierten Ergebnisdatei, für das Zurückschreiben der Ergebnisse in die XMI-Datei des Projekts und zur grafischen Darstellung der Ergebnisse in Form eines Säulendiagrammes und Liniendiagrammes in der GUI. Bei der von HIT erzeugten Ergebnisdatei handelt es sich um ein sogenanntes Dumpfile. Diese ist so formatiert, dass sie leicht maschinell ausgelesen werden kann. Die ausgelesenen Daten werden in einer Liste von Instanzen der Klasse `EvalObject` abgelegt.

  Die grafische Ausgabe der Ergebnisse wird mit Hilfe der Open-Source Bibliothek *JCharts* realisiert, die für die Implementierung verschiedene Diagrammarten anbietet. Die *JCharts*-Bibliothek in der Version 0.7.5, die der *General Public License* unterliegt, erwies sich hierbei als geeignet. *JCharts* ist eine zu 100% auf Java basierende Open-Source-Bibliothek, die für die grafische Darstellung von Diagrammarten via Swing Applikationen, Java Servlet Pages (JSP) und Java Servlets entwickelt wurde. Das erste Release von *JCharts* erfolgte im Dezember 2000. Sie wurde seitdem erweitert und verfügt über eine Vielzahl von Diagrammklassen. Für die Visualisierung der Datenreihen bzw. Ergebnisse wurden wie im Bewertungskonzept schon angedeutet, die Diagrammarten Säulendiagramm und

Liniendiagramm verwendet, obwohl für die Darstellung von Datenreihen für die harte Analyse auch StockCharts sehr sinnvoll gewesen wären. Der Grund für diese Entscheidung liegt darin, dass JCharts für StockCharts eine eingeschränkte Formatierung des Diagramms erlaubt, z.B. ist es bei der Klasse StockCharts nicht möglich, die berechneten Werte für die jeweiligen Datenreihen (Datasets) anzuzeigen.

Für die Implementierung der Säulendiagramme für die harte Analyse werden die Klassen *BarChart* und *ClusteredBarChart* aus der *JCharts* Bibliothek verwendet. Für die weiche Analyse wird die Klasse *LineChart* benutzt. Die Formatierung der Diagramme, wie z.B. der Titel des Diagramms, werden grundsätzlich mit der Klasse DataSeries realisiert. Dies gilt für alle Diagrammarten.

```
DataSeries dataSeries = new DataSeries(
xAxisLabels, xAxisTitle, yAxisTitle, title );
```

Ein weiterer Aspekt bei der Diagrammerzeugung ist die Festlegung der speziellen Attribute, wie beispielsweise die Form der Linien, für das jeweilige Diagramm. Hierzu liefern die Properties-Klassen für die Diagramme die nötigen Methoden. Für das BarChart sieht die Implementierung wie Folgt aus:

```
BarChartProperties barChartProperties= new BarChartProperties();
```

Für die weiteren verwendeten Diagrammtypen lauten die Klassen *BarChartProperties* und *LineChartProperties*. Gleichermaßen gibt es eine Properties-Klasse *ChartProperties* für die Rahmenstruktur des Diagramms selbst. Diese dient z.B. dazu, die Schriftart oder die Position der Texte anzupassen.

Als nächstes wird das *Dataset* angelegt. Ein Dataset bezeichnet dabei eine Struktur bzw. einen Container, der zum Speichern aller für das Diagramm relevanten Komponenten dient. Der Quellcode sieht am Beispiel des BarCharts wie Folgt aus:

```
AxisChartDataSet axisChartDataSet= new AxisChartDataSet(
Ergebnissdaten, LegendLabels, Paints,ChartType.BAR, barChartProperties );
```

Ein letzter, sehr entscheidender Schritt ist die Erzeugung des Diagramms selbst. Dazu muss eine Instanz des jeweiligen Diagrammtyps, in diesem Fall von der Klasse AxisChart, erzeugt werden. Siehe Folgenden Code:

```
AxisChart axisChart= new AxisChart(dataSeries, chartProperties,
axisProperties, legendProperties, AxisCharts.width, AxisCharts.height );
```

- `EvalObject`
  In Objekten dieser Klasse EvalObject werden die Ergebnisse der erzeugten HIT-Evalutionsobjekte gespeichert. Für jedes möglicherweise geforderte Leistungsmaß gibt es einen Vektor, in dem für jedes Experiment eine Hashtable angelegt wird, die die ausgelesenen Ergebnisse enthält. Ausserdem enthält `EvalObject` zweidimensionale Arrays für jedes Ergebnis, welche zur Darstellung der Diagramme erforderlich sind.

- `HIPE`
  Diese Klasse bildet den Einsprungspunkt für die Ausführung von HIPE. Es enthält somit die `main`-Methode, sowie eine Referenz auf eine Instanz der Klasse `Project`, bietet Methoden zum Laden, Speichern und Erzeugen dieser `Project`-Objekte und startet die GUI.

- `Project`
  Das `Project`-Objekt ist der Dreh- und Angelpunkt beinahe aller Aktionen in HIPE. Alles, was miteinander interagieren muss, ist hier versammelt. Eine Instanz dieser Klasse enthält Name, Speicherort, UML-Modell und `XMIDocument` zum Projekt sowie ein `Analyzer`-Objekt zum Anstoßen der Analysevorgänge.

- `softMetrics`
  Die Klasse `softMetrics` ist eine Containerklasse ohne Logik. In ihr werden die Werte gespeichert, die bei der weichen Analyse ermittelt werden. Diese Werte sind: AIF, COF, DIP, MHF, MIF, NOC, NOT, PF, RFC, LOC, CP und AHF. Wofür diese Abkürzungen stehen, wurde bereits im Abschnitt 2.5 auf Seite 104 (weiche Analyse) beschrieben. Daher wird an dieser Stelle nicht mehr darauf eingegangen. Diese Variablen werden anfangs mit "-1" initialisiert. Da die weiche Analyse nur auf Klassen und Klassendiagrammen ausgeführt wird, befinden sich auch nur in `hipe.umlInterface.Class` und in `hipe.umlInterface.ClassDiagram` `softMetrics`-Objekte. So ist jedem `Class`- bzw. `ClassDiagram`-Objekt sein jeweiliges Ergebnis aus der weichen Analyse zugeordnet.

- `XMIDocument`
  `XMIDocument` ist ein sich selbst verwaltender Container für Objekte vom Typ `org.jdom.Document`. Jedes `Project`-Objekt hat einen Link auf sein `XMIDocument`-Objekt und umgekehrt.

3.3 Paket `paradigma`:

Das `paradigma`-Paket fasst alle Klassen zusammen, die zur Realisierung des im Abschnitt 1 (*HIPE-Paradigma*) beschriebenen Vorgehens zur Analyse von UML-Diagrammen nötig sind. Bei der Gestaltung der Struktur dieses Pakets wurde die Grammatik von HI-SLANG (nachzulesen in [Büttner u. a. (1999)]) zugrunde gelegt. Für jedes darin enthaltene "größere" Sprachkonstrukt wurde eine Wrapper-Klasse entworfen, die den entsprechenden Codeblock repräsentiert. Beispielsweise kann eine Instanz der Klasse `Concurrent` einen HI-SLANG-CONCURRENT-Block erzeugen. Ein HI-SLANG-Programm (bzw. dessen Sourcecode) wird zunächst durch eine Klassenhierarchie dieser Klassen dargestellt, beginnend mit einer Instanz der Klasse `HislangFile`. Für sie wird anschließend die Methode zur eigentlichen Codeerzeugung aufgerufen, die sich über die entsprechenden Aufrufe in den einzelnen Klassen der Hierarchy sukzessiv die Codestücke zusammenstellt.

- `Component`
  Die Klasse `Component` modelliert einen COMPONENT-Block in HI-SLANG-Sourcecode für eine einfache Komponente. Dabei werden alle notwendigen Parameter wie der Name der Komponente, Geschwindigkeit, Schedule und Dispatch- Disziplinen aus den UML-Diagrammen rausgesucht. Die Klasse enthält auch die Methode zum Setzen eines Defaultservers.

- `Concurrent`
  Ein Concurrent-Block kapselt eine Menge von Prozessen, die unter Synchronisationsbedingungen nebenläufig ablaufen sollen. Die Darstellung der Synchronisationsbedingung, dass die Ausführung erst dann fortgeführt werden darf, wenn alle diese Prozesse abgeschlossen sind, stellt eine Erweiterung der geschichteten Warteschlangennetze dar. Die Klasse `Concurrent` stellt das entsprechende Konstrukt in HI-SLANG bereit, das zur Beschreibung eines solcherart nebenläufigen Bereiches dient. Die `generateHierarchy()`-Methode stößt die Erzeugung der einzelnen miteinander zu synchronisierenden Blöcke von Anweisungen an und generiert den HI-SLANG-Code zur Darstellung der Synchronisation.

- `DataStorage`
  Diese Klasse dient zur Speicherung aller erzeugten HIT-Komponenten und anderer Daten, die für die Übersetzung des Eingabe-UML-Diagramms in eine von HIT analysierbare Version in HI-SLANG relevant sind. Sie steht jeder Klasse dieses Pakets zur Verfügung und vereinfacht die Datenhaltung. Weiter enthält die einzige von dieser Klasse vorhandene Instanz diverse Libraries zur Rückübersetzung der intern verwendeten Variablen und Komponentennamen in die Namen der assoziierten UML-Elemente. Damit wird dem Bewertungsmodul der Zugriff auf die UML-Modellelemente ermöglicht, um die ermittelten Ergebnisse in das Modell zurückschreiben zu können.

- `ElementGetCouple`
  In einer Instanz dieser Klasse werden die für die Rückübersetzung der HIT-Ergebnisse benötigten Informationen eines Get-Statements einer Diagrammelement-Notiz gespeichert. Diese Informationen sind die EvaluationsID des betrachteten Diagrammelements und der geforderte Parameter des Statements. Die EvaluationsID ist dabei die modellweit eindeutige Zeichenkette eines jeden Diagrammelements. Mit Hilfe der entsprechenden Methode in der Klasse `Modell` (des Pakets `umlInterface`) kann anhand der EvaluationsID das zugehörige Diagrammelement gefunden werden. Für das Statement

  ```
  get responsetime
  ```

  ist der geforderte Parameter gegeben durch die Integer-Konstante

  ```
  Parser.STATEMENT_RESPTIME.
  ```

- `ElementRequiredQuadruple`
  In einer Instanz dieser Klasse werden die für die Rückübersetzung der Ergebnisse einer HIT-Analyse benötigten Informationen eines Required-Statements einer Diagrammelement-Notiz gespeichert. Diese Informationen sind die EvaluationsID des betrachteten Diagrammelements, die mathematische Relation des Required-Statements und der numerische Wert des Statements. Für das Statement

  ```
  required responsetime < 10
  ```

  ist die Relation beispielsweise gegeben durch die Integer-Konstante

  ```
  Parser.RELATION_LESS,
  ```

der numerische Wert ist 10.

- `Evaluation`
  Ein Objekt dieser Klasse erzeugt den HI-SLANG-Code zur Definition der HI-SLANG-Evaluationsobjekte und der daran zu ermittelnden Leistungsmaße. Es benötigt dazu Paare aus UML-Diagrammelementen und deren Hierarchiebeschreibung in Punktnotation (z.B. model.cpu). Aus diesen Informationen wird dann jeweils ein Auswertungs-Statement in HI-SLANG-Code erzeugt. Für Experimentserien wird außerdem eine Liste von Parametern gehalten, die beim „Aufruf"-Statement des Modells übergeben werden müssen (beispielsweise wird bei

  ```
  EVALUATE MODEL model:example(arrival_rate);
  ```

  eine Experimentserie durch die Verwendung des Parameters `arrival_rate` repräsentiert, der zuvor mit den im UML-Diagramm angegebenen Werten belegt wird).

- `EvaluationReference`
  Diese Klasse dient als Datenstruktur für die am UML-Diagramm annotierten GET- und REQUIRED-Statements. Sie werden in HI-SLANG im EVALUATION-Block aufgeführt. Es werden die zu messenden Streams, das vermessene Objekt (das Objekt, an dem die GET-/REQUIRED-Statements annotiert sind) sowie die Position dieses Objekts in der Hierarchie des HIT-Modells in Form eines Strings gespeichert.

- `Experiment`
  Stellt die Objekt-Hierarchie zur Erzeugung des `EXPERIMENT`-Abschnitts des HISLANG-Sourcecodes für das eingegebene UML-Diagramm her und generiert den entsprechenden Code. Hier wird das auf dem hierarchischen Modell auszuführende Experiment (bzw. die Experimentserien) definiert, das von HIT schließlich durchgeführt werden soll. Die bei der Erzeugung der Modelldefinition entdeckten Experimentseriendeklarationen werden hier in Form von Variablen (für die Verwendung im HI-SLANG-Code) repräsentiert. Diese werden von den entsprechenden Objekten der Modelldefinitionshierarchie (deren Erzeugung mit einer Instanz der Klasse `ModelDef` startet) mittels einer dafür zur Verfügung gestellten Methode übergeben und gespeichert. Diese Methode bietet dabei die Möglichkeit, einen Zufallsnamen beliebiger Länge erzeugen zu lassen und als Name für die Variable zu verwenden. Im Fall von zwei oder mehr Variablen wird bei der Codegenerierung vor dem exponentiell wachsenden Aufwand gewarnt, so dass eine unverhältnismäßige Laufzeit vermieden werden kann. Man beachte, dass die Erzeugung der Objekthierarchie und die Codegenerierung in getrennten Abläufen realisiert sind. Das `Experiment`- (und insbesondere das `Evaluation`-) Objekt muss bei der Erzeugung der Modelldefinitionshierarchie stets verfügbar sein, damit allen Objekten in dieser Hierarchie der Zugriff auf oben genannte Methoden eröffnet ist. Deshalb wird die beim `Experiment`-Objekt startende Hierarchie vor dem Start der Modelldefinitionshierarchie und unabhängig von ihr erzeugt.

- `HislangFile`
  Diese Klasse stellt den Einsprungspunkt für die Klassen-Hierarchie dar, die die Konstruktion der HI-SLANG-Ausgabedatei anstoßen soll. Letztere enthält zwei Blöcke, die Modellbeschreibung und die Definition der darauf auszuführenden Experimente. In der Klas-

senhierarchie werden diese Blöcke von Instanzen der Klassen `ModelDef` und `Experiment` erzeugt.

- `Loop`
  Ein Loop-Block enthält eine Menge von Aktivitäten, die mehrfach ausgeführt werden sollen. Die Angabe einer Anzahl von Ausführungen eines solchen Bereichs ist eine Erweiterung des hierarchischen Warteschlangennetzes. Die Klasse Loop stellt in ihrer `generateHierarchy()`-Methode die Implementierung eines Loop-Blocks bereit. Sie erzeugt dazu auf der aktuellen Ebene des Modells den `StatementBlock` eines Services, in dem auf ein in einer Komponente dargestelltes Warteschlangennetz verwiesen wird, das diesen Block implementiert.

- `ModelDef`
  Ein Objekt dieser Klasse generiert den HI-SLANG-Code der Beschreibung des Leistungsmodells. Dieser kann einen oder mehrere Services, einfache oder auch selbstdefinierte Komponenten und einen Refer-Block enthalten. Jeder Akteur aus dem Anwendungsfalldiagramm stellt dabei einen Service dar. Der HI-SLANG-Code für einen Service-Block wird in der Klasse `Service` generiert. Jeder Anwendungsfall stellt eine selbstdefinierte Komponente dar, falls er durch ein Aktivitätsdiagramm verfeinert wird. Analog werden die Aktivitäten im Aktivitätsdiagramm übersetzt. Der entsprechende HI-SLANG-Code wird durch die Klasse `SelfDefComponent` generiert. Die nicht verfeinerten Anwendungsfälle oder Aktivitäten werden auf einer einfachen Komponente oder einer Ressource ausgeführt. Der entsprechende Code wird in den Klassen `Component` und `Resource` erzeugt. In dem Refer-Block werden die Services mit den Komponenten assoziiert, auf denen sie ausgeführt werden. Dieser Block wird durch die Klasse `Refer` erzeugt.

- `NetWorkConnection`
  Die Klasse `NetWorkConnection` modelliert einen NetWorkConnection-Block für eine Netzwerkleitung. Eine Netzwerkleitung wird in HI-SLANG als eine selbstdefinierte Komponente modelliert. Die notwendigen Parameter werden aus den UML-Diagrammen rausgesucht.

- `Refer`
  Ein Objekt der Klasse `Refer` erzeugt einen Refer-Block für das Modell oder eine selbstdefinierte Komponente. In dem Refer-Block werden die einzelnen Use-Services mit den Komponenten assoziiert, auf denen diese ausgefürt werden. Die Komponenten können die Instanzen folgender Klassen sein: `Component, SelfDefComponent, Resource, NetWorkConnection`.

- `Resource`
  Ein Objeckt der Klasse `Resource` generiert den HI-SLANG-Code für eine passive Ressource. Diese wird als eine selbstdefinierte Komponente modelliert. Neben dem regulären Use-Service, der den Kontakt zur ausführenden Hardwarekomponente herstellt, enthält der Service-Block einen weiteren Use-Service „spend“, der die Verzögerung beim Zugriff simuliert. Die Zugriffszeit und die Geschwindigkeit der Ressource werden aus dem Deploymentdiagramm rausgesucht.

- `SelfDefComponent`
  Diese Klasse dient dazu, eine selbstdefinierte Komponente im Sinne von HI-SLANG

zu generieren. Falls ein Usecase im gegebenen Usecasediagramm oder eine Aktivität in einem verfeinernden Aktivitätsdiagramm durch ein weiteres Aktivitätsdiagramm verfeinert wird, werden die verfeinernden Aktivitätsdiagramme in selbstdefinierte Komponenten übersetzt.

- `Service`
  Die Klasse `Service` modelliert einen Service-Block eines HIT-Modells oder auch einer selbstdefinierten Komponente. Dabei wird ein Use-Service-Block und ein Statement-Block generiert.

- `StatementBlock`
  Die Klasse `StatementBlock` sorgt für die Implementierung der Arbeitslast im Statementblock des Modells und der wichtigen Statementblöcke der einzelnen Services, die die einzelnen Ebenen des Modells darstellen. Die Konstruktion beginnt auf der obersten Ebene an einem Use-Case-Diagramm, dessen einzelne Akteure natürliche oder synthetische Nutzer des Systems darstellen, die Anfragen an dieses herantragen. Jeder Akteur kann eine Anzahl von Anwendungsfällen aufrufen, die jeweils selbst durch Annotationen atomar beschrieben oder durch Aktivitätsdiagramme weiter spezifiziert werden. Ein Aktivitätsdiagramm enthält Aktivitäten, die entweder atomar sein oder selbst wieder auf ein Aktivitätsdiagramm verweisen können. Die Klasse `StatementBlock` enthält für Use-Case-Diagramme, Akteure und betrachtete Abschnitte von Aktivitätsdiagrammen jeweils eine Methode `generateHierarchy()`. Die Hierarchiemethode für Use-Case-Diagramme erstellt aus der Beschreibung der einzelnen Akteure dieser Diagramme die Arbeitslast des Modells. Die Methoden, die sich auf Akteure und Aktivitätsdiagramme beziehen, fügen die auf der aktuellen Modellebene dargestellten Zugriffe auf Komponenten (also Anwendungsfälle, Aktivitäten, Blöcke und Objektflüsse) zu einem Warteschlangennetz zusammen und stoßen bei Vorhandensein weiterführender Angaben (also falls ein Diagramm vorhanden ist, das eine Aktivität genauer spezifiziert) oder falls im betrachteten Aktivitätsdiagramm ein Loop- oder Concurrent-Block auftritt, die Erstellung tieferer Modellebenen an, die einzelne Knoten dieses Netzes verfeinern.

- `Use`
  Die Klasse `Use` modelliert einen Use-Service-Block. Diese Klasse enthält eine innere Klasse `UseService`, die einen Use-Service representiert. Die Objekte der Klasse `UseService` werden in einer Liste gespeichert. Diese kann über eine entsprechende Methode gefüllt werden.

3.4 Paket `umlInterface`:

Im `umlInterface`-Paket sind die Klassen vereinigt, die die Schnittstelle zum JDOM-Paket bilden. Mit ihrer Hilfe kann die von JDOM erstellte Struktur der eingelesenen XMI-Datei organisiert und algorithmisch durchlaufen werden. Sie liefern den Inhalt des XMI-Dokuments in Form von Listen (also Objekten, auf denen Java operieren kann) und stellen somit Referenzen zwischen UML-Diagrammelementen und den entsprechenden von HIPE verwendeten Java-Objekten her. Einsprungpunkt ist die Klasse `Modell`. Die anderen Klassen dienen hauptsächlich als Wrapper für ihre namensgleichen UML-Äquivalente. Alle Wrapperklassen erben von der Klasse `ModelElement`. Da die meisten UML-Elemente, bis auf wenige Unterschiede,

dieselben Attribute haben, ist ModelElement auch mächtig geraten. Auf ModelElement und Modell wird nun stärker eingegangen und bei den anderen Wrapperklassen wird nur genannt, was sie in ihrer Implementierung von ModelElement unterscheidet. Zu beachten ist dabei, dass die einzelnen Wrapperklassen noch sehr viel Logik enthalten, deren Kenntniss bei ihrer Benutzung allerdings nicht erforderlich ist. Diese Methoden sind nicht public deklariert, da sie automatisch aufgerufen werden, wenn der Benutzer auf das Modell zugreifen möchte. Als Programmierer greift man einfach auf die Attribute und Listenelemente zu und muss sich um die Instanziierung der einzelnen Elemente nicht kümmern. Wenn er zum Beispiel wissen möchte, mit welchen anderen Modell-Elementen ein bestimmtes Modell-Element verbunden ist, ruft er lediglich die getConnectionCount()-Methode auf und kann anhand der zurückgelieferten Zahl mit getConnection(Zahl) jede beliebige Verbindung zu einem anderen Objekt aufrufen. Über dieselbe Methode kann man auch auf die Kommentare, die sogenannten Notes, zugreifen. Je nachdem, um welches UML-Element es sich handelt, wird eine andere Wrapperklasse benutzt. Ein Use-Case wird durch ein Objekt der Klasse UseCase abgebildet und das Diagram-Objekt, in dem es enthalten ist, durch die Klasse UseCaseDiagram. Das Diagram-Objekt selber ist wiederum in der UseCaseDiagramList-Liste innerhalb der Klasse Modell zu finden.

- ModelElement
  Die Klasse ModelElement steht für jedes UML-Element innerhalb eines UML-Modells. Jedes UML-Element hat einen Namen und es besteht bei jedem UML-Element die Möglichkeit, einen Kommentar einzufügen. Über ModelElement ist auch sichergestellt, dass man die Ergebnisse auch wieder ins ursprüngliche Modell zurückschreiben kann. Ausserdem kann jedes UML-Element mit jedem anderem verbunden sein. Jedes UML-Element besitzt eine Evaluations-ID, die es innerhalb von HIPE eindeutig kennzeichnet. Dies gilt nicht immer für den Namen des UML-Elementes. Da alle anderen Wrapperklassen von ModelElement erben, wird im weiteren Verlauf nur noch auf die Unterschiede eingegangen.

- Modell
  Die Klasse Modell ist die nach außen hin wichtigste Klasse innerhalb des umlInterface-Pakets. Sie enthält folgende vier private Listen:

  – activityDiagramList
  – useCaseDiagramList
  – deploymentDiagramList
  – classDiagramList

  Diese Listen enthalten die jeweiligen Objekte der jeweiligen Diagram-Klasse. Die Diagramme ihrerseits enthalten wiederum Listen, die die jeweiligen Objekte der Diagram-Element-Klasse in sich aufnehmen. Alle diese Listen sind bereits gefüllt, wenn ein Modellobjekt mit dem nichtleeren Constructor erzeugt wurde. Der Zugriff auf die Diagramme sowie auf ihre Unterelemente geschieht ausschließlich über die entsprechend benannten get- oder in Einzelfällen auch set-Methoden. Siehe dazu auch die Beschreibung der Klasse ActivityDiagram.

- Activity
  Die Klasse Activity ist eine Klasse, die Aktivitätsdiagrammelemente kapselt. Sie hat außer ihren ererbten Attributen und Methoden noch die Attribute Typ und ParentClass.

`Typ` ist ein privates Attribut vom Typ `String`. Über seine Zugriffsmethoden ist es möglich, herauszufinden, um welche Art von Aktivität es sich bei dieser handelt. Da Aktivitätsdiagrammelemente auch Objekte sein können, gibt es zudem noch das private Attribut `ParentClass` vom Typ `Class`. `Typ` kann eine Menge verschiedener Werte annehmen und deutet stets darauf, ob es sich bei dieser Activity um eine Aktivität, einen Knotenpunkt, eine Verzweigung, ein Objekt oder eine Aktion handelt. Mögliche Werte sind:

- `VActivity`
- `VControlNode`
- `VFork`
- `VObject`
- `VAction`
- `VInitial`
- `VActifinal`
- `VDecision`

- `ActivityDiagramm`
  Die Klasse `AktivityDiagramm` ist eine Klasse, die Aktivitätsdiagramme kapselt. Sie besteht außer den geerbten noch aus folgenden, von außen nicht zugreifbaren Attributen:

  - `InitialNodeList`: Eine Liste gefüllt mit den Startpunkten dieses Diagramms
  - `Activities`: Eine Liste gefüllt mit allen Elementen dieses Diagramms

  Diese Listen werden über private Methoden gefüllt. Der Zugriff gestaltet sich über die Methoden:

  - `getActivitiesCount()`
  - `getActivity(index)`
  - `getInitialNodesCount()`
  - `getInitialNode(index)`

  Hier ein Beispiel für den Zugriff auf die Activities-Liste:

```
int i;
for (i = 0,i < getActivitiesCount(),i++)
 {
 Activity ac=(Activity) getActivity(i);
 System.out.println(Ac.getName());
}
```

- `Actor`
  Die Klasse Actor ist eine Klasse, die Use-Case-Diagrammelemente kapselt. Sie hat außer ihren ererbten Attributen und Methoden keine weiteren.

- `Attribute`
  Die Klasse `Attribute` ist eine Klasse, die Attribute von Klassen kapselt. Analog zu echten Attributen in UML-Diagrammen hat sie außer ihren bereits ererbten Attributen noch die Attribute `Typ` und `Visibility`. `Typ` kann zum Beispiel `String` oder `Integer` sein, wohingegen bei `Visibility` Werte wie `public`, `private`, `package` oder `protected` möglich sind. Der Zugriff auf diese Attribute geschieht über die jeweiligen Zugriffsmethoden. Wie bei fast allen Attributen im Package `umlInterface` kann man von außen nur lesend auf die Attribute zugreifen.

- `Class`
  Die Klasse `Class` ist eine Klasse, die UML-Klassen kapselt. Analog zu echten Klassen in UML-Diagrammen hat sie zusätzlich zu ihren bereits ererbten Attributen noch Attribut- und Methodenlisten sowie ein `SoftMetric`-Attribut.

  - `attributeList`: Liste aller Attributen der Klasse.
  - `methodeList`: Liste aller Methoden der Klasse.
  - `SoftMetric`: Objekt, welches die weichen Maße der Klasse kapselt.

  Auf diese Listen kann über die bekannten Zugriffsmethoden (also die standardmäßig benannten `get`-Methoden) zugegriffen werden. Siehe dazu auch die Beschreibung der Klasse `ActivityDiagram`.

- `ClassDiagram`
  Die Klasse `ClassDiagram` ist eine Klasse, die Klassendiagramme kapselt. Sie besteht außer den geerbten Attributen noch aus der von außen nicht zugreifbaren Liste `classList`, die alle Startpunkte dieses Diagramms enthält. Sie wird über die private Methode `fillClassList()` gefüllt. Der Zugriff gestaltet sich über die Methoden:

  - `getClassesCount()`
  - `getClass(index)`

  Siehe dazu auch die Beschreibung der Klasse `ActivityDiagram`.

- `Connection`
  Die Klasse `Connection` ist eine Klasse, die Verbindungen jeder Art zwischen den Diagrammelementen kapselt. Jede Verbindung hat einen Startpunkt (*source*) und einen Endpunkt (*target*). Ausserdem hat jede Verbindung einen Typ, da es ja sehr viele unterschiedliche Verbindungen gibt. Typ kann dabei folgende Werte annhemen:

  - `VUndefined`
  - `VDependency`
  - `VInclude`
  - `VExtend`
  - `VAssociation`
  - `VAggregation`
  - `VComposition`
  - `VGeneralization`

  - `VRealization`
  - `VControlFLow`
  - `VObjectFLow`

  Einige Eigenheiten der XMI-Dateibehandlung durch METAMILL machen es hier auch notwendig, Verbindungen innerhalb von Aktivitätsdiagrammen getrennt zu behandeln. Daher gibt es zum Füllen der `Connection` zwei verschiedene Methoden, die je nach Bedarf von dem entsprechenden Konstruktor aufgerufen werden.

  - `fillConnectionActivityCase()`: Wird nur bei Aktivitätsdiagrammen benutzt.
  - `fillConnectionList()`: Wird bei allen anderen benutzt.

- `DeploymentDiagram`
  Die Klasse `DeploymentDiagram` ist eine Klasse, die Verteilungsdiagramme kapselt. Sie besteht außer den geerbten Attributen noch aus der von außen nicht zugreifbaren Liste `node`, die alle `Node`-Objekte, die zu diesem Diagramm gehören, enthält. Diese Liste wird über die private Methode `fillNodeList()` gefüllt. Der Zugriff gestaltet sich über die Methoden:

  - `getNodeCount()`
  - `getNode(index)`
  - `getNode(name)`

  Siehe dazu auch die Beschreibung der Klasse `ActivityDiagram`.

- `DiagramElement`
  Die Klasse `DiagramElement` ist eine Klasse, die Diagrammelemente kapselt. Sie enthält außer den geerbten noch die folgenden, selbsterklärenden Methoden:

  - `getOutgoingAssosiations()`
  - `getIncomingAssosiations()`

- `Method`
  Die Klasse `Method` ist eine Klasse, die Methoden von Klassen kapselt. Analog zu echten Methoden in UML-Diagrammen hat sie die Attribute `returnType` und `Visibility`, sowie eine `parameterList`. `returnTyp` beschreibt den Typ ihres Rückgabewertes und kann zum Beispiel `String` oder `Integer` sein, wohingegen bei der Sichtbarkeitsbeschreibung `Visibility` Werte wie `public`, `private`, `package` oder `protected` möglich sind. Der Zugriff auf diese Attribute geschieht über die jeweiligen Zugriffsmethoden. Wie bei fast allen Attributen im Package `umlInterface` kann man von außen nur lesend auf die Attribute zugreifen. Weiter enthält diese Klasse noch die Methode `fillParameterList-AndReturnType()`, die die Parameter der Methode und ihren Rückgabewert anhand der Daten im eingegebenen UML-Diagramm füllt.

- `Node`
  Die Klasse `Node` ist eine Klasse, die Verteilungsdiagramm-Elemente kapselt. Sie besteht aus den Attributen `parentNode` (Referenz auf den Elternknoten) und `sonNode` (Nachfolger) sowie der Methode `fillNode()`, mit der die Attributen der Node eingefügt werden.

Die Attribute `parentNode` und `sonNode` deuten bereits an, dass sich diese Objekte beliebig schachteln lassen, wie es in Verteilungsdiagrammen auch vorgesehen ist. Ansonsten verfügen `Node`-Objekte über ihre von `DiagramElement` ererbten Attribute und Methoden.

- `Parameter`
  Die Klasse `Parameter` ist eine Klasse, welche die Parameter von Klassen kapselt. Parameter haben immer einen Typ und einen Namen, so auch diese Klasse.

- `UseCase`
  Die Klasse `UseCase` ist eine Klasse, die Use-Cases kapselt. Sie hat außer ihren ererbten Attributen und Methoden keine weiteren.

- `UseCaseDiagram`
  Die Klasse `UseCaseDiagram` ist eine Klasse, die UseCase-Diagramme kapselt. Sie besteht neben den geerbten noch aus folgenden, von außen nicht zugreifbaren Attributen:

  – `useCaseList`: Die Liste aller Anwendungsfaelle in diesem Anwendungsfalldiagramm.
  – `actorList`: Die Liste aller Akteure des Anwendungsfalldiagramms.
  – `combinedList`: Die Liste aller Verbindungen des Anwendungsfalldiagramms.

  Diese Listen werden über private Methoden gefüllt. Der Zugriff gestaltet sich über die Methoden:

  – `getUseCaseCount()`
  – `getUseCase(index)`
  – `getActorCount()`
  – `getActor(index)`

  Auf diese Listen kann über die bekannten Zugriffsmethoden (also die standardmäßig benannten `get`-Methoden) zugegriffen werden. Siehe dazu auch die Beschreibung der Klasse `ActivityDiagram`.

3.5 Änderungen gegenüber Kaptitel 2

Grundsätzlich folgte die Implementierung den Ideen des im Kapitel 2 gezeigten Konzepts. In Einzelfällen ergaben sich jedoch nicht vorhersehbare programmiertechnische Situationen, die kleinere Umstrukturierungen und Ergänzungen erforderlich machten. Beispielsweise gab es im ursprünglichen Entwurf des `paradigma`-Pakets auch eine `Branch`-Klasse, die den gleichnamigen Block in HI-SLANG repräsentieren sollte. Sie ist entfallen, da ihre Funktionalität besser "lokal" realisiert werden konnte. In jedem Paket wurden außerdem je nach Notwendigkeit Klassen ergänzt, die bei der Erzeugung der gewünschten Logik behilflich sind, also nur "Mittel zum Zweck" sind. Sie tauchen daher nicht im Entwurf auf.

4 Produkttest

Aufgrund der hohen Komplexität der Klassen des Paradigma-Pakets und des daraus resultierenden Aufwands der Treiberprogrammierung und des gleichermaßen beengten Zeitrahmens der Projektgruppe musste der Klassentest entfallen. Lediglich ein "optimistischer" Produkttest wurde durchgeführt. Seine Ergebnisse werden im Folgenden präsentiert. Es werden Tests aus fünf verschiedenen Kategorien durchgeführt, die jeweils unterschiedliche Aspekte des HIPE betonen.

- *Korrektheit der Implementierung.* Einige schon relativ komplexe Beispiele, in denen alle grundsätzlichen Fähigkeiten von HIPE im Umgang mit den verschiedenen Diagrammtypen vorkommen, dienen der Überprüfung, ob HIPE zumindest die implementierten Teile des HIPE-Paradigmas korrekt umsetzt. Korrektheit bedeutet für die einzelnen Teiltests, dass jeweils alle Angaben bereitgestellt werden, die an einer gegebenen Stelle notwendig sind, um ein im Sinne des HIPE-Paradigmas an dieser Stelle vollständiges und mit HIT bewertbares Modell zu erzeugen. Die wichtigen Teiltests beziehen sich zum einen auf die erfolgreiche Umsetzung von Diagrammen für den Löser SIMULATIVE und zum anderen auf Aspekte die Anpassungen, die HIPE vornimmt, damit Modelle auch auf anderen Lösern ausgewertet werden können.

- *Korrektheit der Berechnungen.* An den Diagrammen aus der vorigen Kategorie kann ohne größeren Aufwand die Korrektheit der von HIT auf dem durch HIPE umgesetzten Modellen höchstens tendentiell oder qualitativ erfasst werden. Für einen "harten" Test erscheint diese Vorgehensweise zu grob. Daher werden zum Test auf korrekte Berechnung vereinfachte Modelle verwendet, deren Eigenschaften noch gut manuell erfasst werden können oder für die die erwarteten Ergebnisse unter bestimmten Parametersätzen bereits bekannt sind. Beispiele für solche Modelle stammen etwa aus dem Modellierungspraktikum [PG459 (2005)].

- *Test der weichen Analyse.* HIPE kann neben den Maßen, die HIT liefern kann, im Rahmen der weichen Analyse weitere Maße berechnen. Deren korrekte Berechnung wird gesondert anhand eines Klassendiagramms überprüft.

- *Verhalten von HIPE im Umgang mit großen Modellen.* Es soll schließlich getestet werden, wie sich HIPE anhand eines großen Beispiels verhält. Es reizt die Möglichkeiten von HIPE voll aus, was die verwendbaren Diagrammelemente angeht, und soll der Überprüfung dienen, ob HIPE größere Strukturen, in denen zum Beispiel viele Komponenten modellweit (in der Terminologie von HISLANG "ENCLOSEd") verwendet werden, korrekt handhaben kann. An diesem Modell wird allerdings nur ein eingeschränkter Test vorgenommen, was der Kürze der zum Testen zur Verfügung stehenden Zeit geschuldet ist.

- *Test der Fehlerbehandlung.* In HIPE wurde eine zweistufige Fehlerbehandlung implementiert. Einerseits werden Situationen erkannt, in denen Anpassungen am Modell vor-

genommen werden müssen, und den Nutzer davon informiert, dass sich die Semantik eines angepassten Modells möglicherweise von der von ihm intendierten Semantik unterscheidet. Andererseits sollen Modelle erkannt werden, die HIPE nicht umsetzen kann, weil entweder fehlende Angaben nicht durch andere Angaben ergänzt bzw. ersetzt werden können, oder bestimmte Konstrukte unter der vom Nutzer gewählten Konfiguration weder eine Bewertung erlauben noch dafür eine Anpassung existiert.

Dabei darf man sich den Test nicht als erschöpfend vorstellen. Das wäre bei einem Produkt dieser Größe auch kaum möglich. Es wurde aber der Nachweis versucht, dass HIPE zumindest die grundlegenden Anforderungen erfüllt und grundsätzlich auch Fehlersituationen erkannt werden. Tatsächlich wurden aber auch bei diesem noch eher eingeschränkten Test noch einige Fehler im Produkt festgestellt. An den entsprechenden Stellen wird darauf noch einmal Bezug genommen.

Für den Test wurden in HIPE verschiedene Default-Werte bzw. Parameter für die verschiedenen Löser vorgegeben, die bei der Generierung von Aufrufen, Komponenten und Experimenten verwendet werden, falls in den jeweils untersuchten Modellen entsprechende Angaben fehlen. Die im Folgenden präsentierten Default-Werte stellen eine willkürliche und nicht auf das untersuchte Modell abgestimmte Zusammenstellung dar. Zur Wahl der Werte für den Konfidenzlevel (CONFIDENCE LEVEL) und die Konfidenzintervallbreite (CONFIDENCE INTERVAL WIDTH) ist zu sagen, dass deren Belegung mit 99 bzw. 1 die strengste Abbruchbedingung für eine simulative Analyse darstellen. Diese Belegung folgt jedoch wie die anderen Werte keiner besonderen Motivation. Die Werte im einzelnen:

- Default-Wert für Dispatch (Eigenschaft der Default-CPU): `DISPATCH_EQUAL`
- Default-Wert für Schedule (Eigenschaft der Default-CPU): `SCHEDULE_IMMEDIATE`
- Default-Wert für ExtDelay (Eigenschaft geschlossener Arbeitslasten): 3 Zeiteinheiten
- Default-Wert für Responsetime (Eigenschaft atomarer Aufrufe): 1 Zeiteinheit
- Default-Wert für Speed (Eigenschaft der Default-CPU): 1000 Anweisungen/Zeiteinheit
- Parameter POPULATION LIMIT für den Löser ANALYTICAL_MARKOV: 200
- Parameter CPUTIME für den Löser ANALYTICAL_MARKOV: 120
- Parameter ACCURACY für den Löser ANALYTICAL_MARKOV: 0.1
- Parameter CPUTIME für den Löser SIMULATIVE: 120
- Parameter MODELTIME für den Löser SIMULATIVE: 10000
- Parameter CONFIDENCE LEVEL für den Löser SIMULATIVE: 99
- Parameter (CONFIDENCE INTERVAL) WIDTH für den Löser SIMULATIVE: 1

4.1 Tests von Use-Case-Diagrammen

Im Folgenden wird zunächst die Korrektheit der Umsetzung von Use-Case-Diagrammen überprüft. Dabei werden im einzelnen zunächst die verschiedenen gemäß des Paradigmas möglichen Umsetzungen überprüft. Danach werden reine Use-Case-Modelle erstellt, die von HIPE in einfache Warteschlangenmodelle umgesetzt werden, um für diesen noch nachvollziehbaren Fall die Korrektheit der von HIPE angewandten Transformationen in Bezug auf die berechneten Ergebnisse zu kontrollieren. Beispielhaft werden ein M/M/1-Modell und ein Modell mit zwei parallelgeschalteten M/M/1-Systemen ("M/M/1||M/M/1") betrachtet. Dabei werden auch die automatischen Anpassungen überprüft, die HIPE vornimmt, um ein mit Hilfe eines analytischen Lösers bewertbares Modell zu erzeugen.

4.1.1 Test mit einem Use-Case-Diagramm

Es wird ein Use-Case-Diagramm *UseCaseTest* erstellt, das grundsätzlich eine Betrachtung aller im Kapitel 1.2.6.1 ab Seite 43 beschriebenen Umsetzungen dieses Diagrammtyps ermöglicht. Es werden also die Definition der Arbeitslast und die von HIPE vorgenommenen Anpassungen überprüft, die eine Bewertung des Modells mit einem analytischen Löser erlauben. Als Referenz für die verschiedenen vorgenommenen Anpassungen dient dabei das Kapitel 1.2.8 ab Seite 70.

Dazu werden zur Definition der Arbeitslast vier Akteure erstellt, die mit jeweils einer der möglichen vollständigen Annotationen für die Arbeitslast versehen werden. Dabei wird getestet, ob die Definition der Arbeitslast unter den verschiedenen Lösern korrekt erfolgt, das heißt die Restriktion berücksichtigt wird, daß unter einer analytischen Lösungsmethode mittels der Annotation `LET arrivaltime = ...;` angegebenen einzelnen Ankunftszeitpunkte in einen Wert gemäß des Schlüsselwortes `occurence` umgerechnet werden.

Jedem der Akteure wird weiterhin eine Menge von Anwendungsfällen zugeordnet. Damit wird getestet, ob die diesen Anwendungsfällen zugeordneten Aufrufwahrscheinlichkeiten und die im Paradigma beschriebenen Transformationen für die in den Anwendungsfällen dargestellten Aktivitäten korrekt umgesetzt werden. Zusätzlich zum Use-Case-Diagramm wird noch ein Aktivitätsdiagramm erstellt, auf das von HIPE aus automatisch bzw. durch Annotation von Anwendungsfällen verwiesen werden kann. Das im Test diskutierte Anwendungsfalldiagramm findet sich in Abbildung 4.1 auf S. 124.

Erwartetes Verhalten
Der Akteur `OO` stellt eine offene Arbeitslast dar, es wird alle drei Zeiteinheiten mittels `CREATE 1 PROCESS OO EVERY 3;` ein Prozess des Typs `OO` erzeugt. Wird der Löser MARKOV ausgewählt, wird das `CREATE`-Statement zu `CREATE 1 PROCESS OO LIMIT 200 EVERY negexp(1/3);` erweitert. Bei der Zuweisung der Aufrufwahrscheinlichkeiten wird die Methode der Gleichverteilung angewandt: `OO` ruft entsprechend mit einer Wahrscheinlichkeit von 50% den Anwendungsfall `A1` auf - wobei automatisch erkannt wird, dass ein Aktivitätsdiagramm `A1` vorhanden ist - und mit einer Wahrscheinlichkeit von 50% den Anwendungsfall `A2` auf, in dem explizit durch `LET diagram = A1;` ebenfalls auf das Aktivitätsdiagramm `A1` verwiesen wird.

Der Akteur `OA` stellt eine offene Arbeitslast dar, wobei zu den durch `let arrivaltime = 1,2,`

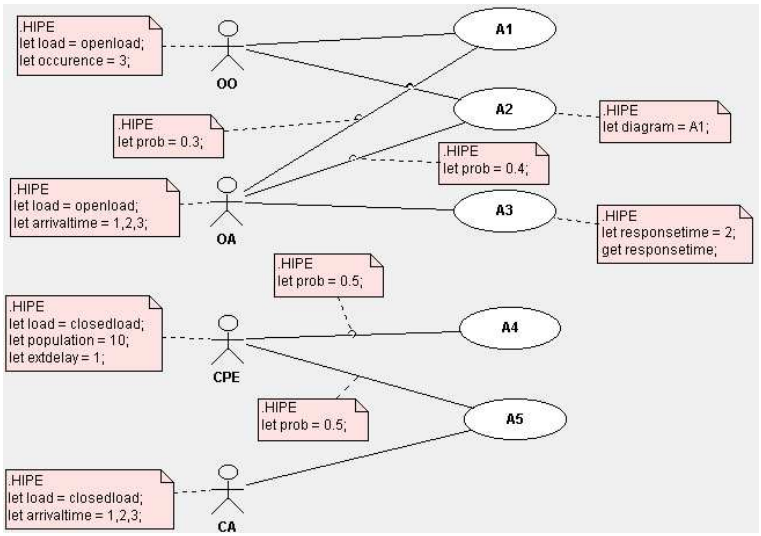

Abbildung 4.1: Use-Case-Diagramm *UseCaseTest*

`3;` gegebenen Zeitpunkten jeweils ein Prozess des Typs `OA` erzeugt wird. Bei Auswahl einer analytischen Lösungsmethode wird dieses Statement in das Statement `EVERY negexp(1/1)` umgesetzt und bei der Auswahl von MARKOV der Term `LIMIT 200` ergänzt. Bei der Zuweisung der Aufrufwahrscheinlichkeiten wird die Differenzmethode angewandt: `OA` ruft mit einer Wahrscheinlichkeit von 30% den Anwendungsfall `A1`, mit 40% den Anwendungsfall `A2` auf. Für den einzigen verwiesenen und nicht annotierten Anwendungsfall `A3` wird die Wahrscheinlichkeit $1 - (0.3 + 0.4) = 0.3$ eingesetzt. `A1` und `A2` werden als Aufrufe des Aktivitätsdiagramms `A1` codiert, `A3` als Aufruf der von HIPE aus implizit vorgegebenen Default-CPU, der zwei Zeiteinheiten erfordert.

Der Akteur `CPE` stellt eine geschlossene Arbeitslast dar. Es werden 10 Prozesse des Typs `CPE` erzeugt, und es wird für `CPE` eine Verzögerung zwischen der Fertigstellung einer Anfrage und der Stellung der nächsten Anfrage eingesetzt, die eine Zeiteinheit beträgt. Bei der Zuweisung der Aufrufwahrscheinlichkeiten wird die Methode der vollständigen Annotation angewandt, entsprechend werden die Anwendungsfälle `A4` und `A5` jeweils mit einer Wahrscheinlichkeit von 50% aufgerufen. Beide Anwendungsfälle stellen Aufrufe der Default-CPU dar, die jeweils die Default-Dauer ausmachen.

Der Akteur `CA` stellt eine geschlossene Arbeitslast dar, wobei zu den durch `let arrivaltime = 1,2,3;` vorgegebenen Zeitpunkten jeweils ein Prozess des Typs `CA` erzeugt wird. Bei Auswahl einer analytischen Methode wird dieses Statement darauf abgebildet, dass insgesamt drei Prozesse des Typs `CA` erzeugt werden. Der Prozess `CA` ruft mit einer Wahrscheinlichkeit von 100% den einzigen Anwendungsfall `A5` auf. Als Wert für die externe Verzögerung wird der Vorgabewert, in diesem Fall 3 Zeiteinheiten, eingesetzt.

Es werden folgende weitere Anpassungen für analytische Löser durchgeführt: Da MARKOV Determinismen (Übergänge mit Wahrscheinlichkeit 100%) nicht unterstützt, wird für solche Übergänge (zum Beispiel also für den Aufruf des Anwendungsfalls `A5` durch den Prozess `CA`)

eine geringfügig kleinere Wahrscheinlichkeit von 99.9999% eingesetzt. Weiterhin werden in Aufrufen von atomaren Aktivitäten (z.B. der Aufruf von `A3` durch den Service `OA`) deterministische Ausführungszeiten auf cox-verteilte Ausführungszeiten abgebildet, der genannte Aufruf entsprechend von einem Umfang von deterministisch 2000 Einheiten auf cox(2000,0.32) Einheiten.

Da in dem getesteten Modell kein Verteilungsdiagramm enthalten ist, in dem eine CPU definiert wird, erzeugt HIPE automatisch eine Default-CPU mit den Eigenschaften speed = 1000, dispatch = equal und schedule = immediate.

Der Modellname ist dabei identisch mit dem Namen des Projekts, dem dieses Modell in HIPE zugeordnet ist. Für diese Betrachtung wurde der Projektname identisch zu dem des Use-Case-Diagramms, also als `UseCaseTest` gewählt. Für von HIPE erzeugte (nicht explizit im Eingabemodell vorhandene) Komponenten werden Zufallsnamen erzeugt, die dann an den entsprechenden Stellen im HI-SLANG-Quellcode eingesetzt werden. Die Namen der Used Services ergeben sich wie folgt: Falls der korrespondierende Anwendungsfall atomar ist, d.h. kein Aktivitätsdiagramm mit dem Namen dieses Anwendungsfalls gefunden und auch kein Aktivitätsdiagramm explizit verwiesen wird, so wird der Name aus dem Namen des Anwendungsfalls gebildet. Falls ein Aktivitätsdiagramm mit dem Namen des Anwendungsfalls gefunden oder explizit ein Aktivitätsdiagramm verwiesen wird, besteht der Name des korrespondierten Used Services aus dem Präfix `A` (für „Aktivitätsdiagramm“), dem Namen des Aktivitätsdiagramms und einem Nummerierungs-Postfix, das während der Konstruktion angibt, wie viele Used Services in diesem Use-Case-Diagramm inklusive dem aktuell betrachteten bereits auf dieses Aktivitätsdiagramm zugreifen (siehe dazu auch Tabelle 1.6).

Realisiertes Verhalten

Im Folgenden werden zum einen der Code des Modells, in dem die einzelnen Prozesse erzeugt werden und zum anderen die Bodies der einzelnen Servicetypen mit dem erwarteten Verhalten von HIPE verglichen, dies jeweils unter der Auswahl des simulativen wie eines analytischen Lösers.

```
% Code des Modells, Löser "SIMULATIVE":
-->001 %ANALYZER
-->002 %PARM=UPDATES
-->003 %END
-->004 TYPE UseCaseTest MODEL;
-->005   TYPE OO SERVICE;
-->006     USE
-->007       SERVICE
-->008         AA1_1;
-->009         AA1_2;
-->010     END USE;
-->011     BEGIN
-->012       BRANCH
-->013       PROB 0.5: AA1_1;
-->014       PROB 0.5: AA1_2;
-->015       END BRANCH;
-->016   END TYPE OO;
-->017   TYPE OA SERVICE;
-->018     USE
-->019       SERVICE
-->020         A3 (AMOUNT : REAL);
-->021         AA1_3;
-->022         AA1_4;
-->023     END USE;
-->024     BEGIN
-->025       BRANCH
-->026       PROB 0.30000000000000004: A3(2000.0);
-->027       PROB 0.3: AA1_3;
-->028       PROB 0.4: AA1_4;
```

```
-->029       END BRANCH;
-->030    END TYPE OA;
-->031    TYPE CPE SERVICE;
-->032      USE
-->033        SERVICE
-->034          A4;
-->035          A5;
-->036      END USE;
-->037      BEGIN
-->038        LOOP
-->039        BRANCH
-->040        PROB 0.5: A4(1.0*1000);
-->041        PROB 0.5: A5(1.0*1000);
-->042        END BRANCH;
-->043        spend(1);
-->044        END LOOP;
-->045    END TYPE CPE;
-->046    TYPE CA SERVICE;
-->047      USE
-->048        SERVICE
-->049          A5;
-->050      END USE;
-->051      BEGIN
-->052        LOOP
-->053        BRANCH
-->054        PROB 1.0: A5(1.0*1000);
-->055        END BRANCH;
-->056        spend(3.0);
-->057        END LOOP;
-->058    END TYPE CA;
-->059    TYPE AA1 COMPONENT;
% Die Struktur des Aktivitätsdiagramms wird in diesem Test nicht betrachtet
-->078    END TYPE AA1;
-->079    COMPONENT dgmlri : server (LET schedule := immediate,LET dispatch := equal(LET speed := 1000));
-->080    COMPONENT AA1_1 : AA1;
-->081    REFER OO,OA,CPE,CA TO dgmlri,AA1_1 EQUATING
-->082      OO.AA1_1 WITH AA1_1.request;
-->083      OO.AA1_2 WITH AA1_1.request;
-->084      OA.A3 WITH dgmlri.request;
-->085      OA.AA1_3 WITH AA1_1.request;
-->086      OA.AA1_4 WITH AA1_1.request;
-->087      CPE.A4 WITH dgmlri.request;
-->088      CPE.A5 WITH dgmlri.request;
-->089      CA.A5 WITH dgmlri.request;
-->090    END REFER;
-->091  BEGIN
-->092    CREATE 1 PROCESS OO EVERY 3;
-->093    CREATE 1 PROCESS OA AT 1;
-->094    CREATE 1 PROCESS OA AT 2;
-->095    CREATE 1 PROCESS OA AT 3;
-->096    CREATE 10 PROCESS CPE;
-->097    CREATE 1 PROCESS CA AT 1;
-->098    CREATE 1 PROCESS CA AT 2;
-->099    CREATE 1 PROCESS CA AT 3;
-->100  END TYPE UseCaseTest;
-->101  EXPERIMENT UseCaseTest_exp METHOD SIMULATIVE;
% Die Struktur des Experiments wird in diesem Test nicht betrachtet
```

Als Default-CPU wird ein HIT-Server mit dem Zufallsnamen `dgmlri` erzeugt, der eine Geschwindigkeit von 1000 Anweisungen pro Zeiteinheit, die Dispatch-Eigenschaft equal und die Schedule-Eigenschaft immediate besitzt.

Vom Akteur `OO` wird jeweils mit einer Wahrscheinlichkeit von 50% einer der beiden Used Services aufgerufen, von denen der erste den Aufruf des implizit (durch Namensgleichheit) und der zweite den Aufruf des explizit (über `LET diagram`) verwiesenen Aktivitätsdiagramms *A1* darstellt.

Der Akteur `OA` ruft mit 30%-iger Wahrscheinlichkeit den Anwendungsfall `A1` auf, mit 40%-iger Wahrscheinlichkeit den Anwendungsfall `A2`, wobei `A1` und `A2` implizit bzw. explizit dem Aktivitätsdiagramm *A1* verbunden sind. Der Anwendungsfall `A3` wird mit einer Wahrscheinlichkeit von 30% aufgerufen, wobei dieser ein atomarer Aufruf ist, der auf die Default-CPU abgebildet wird.

Der Akteur `CPE` ruft jeweils mit der gleichen Wahrscheinlichkeit von 50% einen der beiden An-

wendungsfälle `A4` und `A5` auf, wobei beide durch Aufrufe atomarer Abläufe im Umfang der als Default-Wert vorgegebenen Ausführungsdauer eines Aufrufs multipliziert mit der Geschwindigkeit der CPU dargestellt werden. Zwischen dem Abschluss einer Anfrage und der Stellung der nächsten tritt eine Verzögerung von einer Zeiteinheit auf.

Der Akteur `CA` ruft mit einer Wahrscheinlichkeit von 100% den Anwendungsfall `A5` auf, wobei der Aufruf wiederum auf die Default-CPU abgebildet wird, die für die als Default-Dauer angegebene Zeitdauer in Anspruch genommen wird.

Im Body des Modells werden die einzelnen Prozesse erzeugt: alle drei Zeiteinheiten ein Prozess des Typs `OO`, zu den Zeitpunkten 1,2 und 3 jeweils ein Prozess des Typs `OA`, 10 Prozesse des Typs `CPE` und zu den Zeitpunkten 1,2 und 3 jeweils ein Prozess des Typs `CA`.

Wir betrachten weiterhin den Code, der den Body des Modells und den Header der Experimentdefinition bei der Auswahl des analytischen Lösers MARKOV darstellt:

```
% Code des Modells, Löser "ANALYTICAL MARKOV":
-->001 %ANALYZER
-->002 %PARM=UPDATES
-->003 %END
-->004 TYPE UseCaseTest MODEL;
-->005   TYPE OO SERVICE;
-->006     USE
-->007       SERVICE
-->008         AA1_1;
-->009         AA1_2;
-->010     END USE;
-->011     BEGIN
-->012       BRANCH
-->013       PROB 0.5: AA1_1;
-->014       PROB 0.5: AA1_2;
-->015       END BRANCH;
-->016   END TYPE OO;
-->017   TYPE OA SERVICE;
-->018     USE
-->019       SERVICE
-->020         A3 (AMOUNT : REAL);
-->021         AA1_3;
-->022         AA1_4;
-->023     END USE;
-->024     BEGIN
-->025       BRANCH
-->026       PROB 0.30000000000000004: A3(cox(2000.0,0.32));
-->027       PROB 0.3: AA1_3;
-->028       PROB 0.4: AA1_4;
-->029       END BRANCH;
-->030   END TYPE OA;
-->031   TYPE CPE SERVICE;
-->032     USE
-->033       SERVICE
-->034         A4 (AMOUNT : REAL);
-->035         A5 (AMOUNT : REAL);
-->036     END USE;
-->037     BEGIN
-->038       LOOP
-->039       BRANCH
-->040       PROB 0.5: A4(cox(1.0*1000,0.32));
-->041       PROB 0.5: A5(cox(1.0*1000,0.32));
-->042       END BRANCH;
-->043       spend(cox(1,0.32));
-->044       END LOOP;
-->045   END TYPE CPE;
-->046   TYPE CA SERVICE;
-->047     USE
-->048       SERVICE
-->049         A5 (AMOUNT : REAL);
-->050     END USE;
-->051     BEGIN
-->052       LOOP
-->053       BRANCH
-->054       PROB 0.999999: A5(cox(1.0*1000,0.32));
-->055       END BRANCH;
-->056       spend(cox(3.0,0.32));
-->057       END LOOP;
-->058   END TYPE CA;
-->059   TYPE AA1 COMPONENT;
```

```
% ...
-->078   END TYPE AA1;
-->079   COMPONENT bwrtqs : server (LET schedule := immediate,LET dispatch := equal(LET speed := 1000));
-->080   COMPONENT AA1_1 : AA1;
-->081   REFER OO,OA,CPE,CA TO bwrtqs,AA1_1 EQUATING
-->082     OO.AA1_1 WITH AA1_1.request;
-->083     OO.AA1_2 WITH AA1_1.request;
-->084     OA.A3 WITH bwrtqs.request;
-->085     OA.AA1_3 WITH AA1_1.request;
-->086     OA.AA1_4 WITH AA1_1.request;
-->087     CPE.A4 WITH bwrtqs.request;
-->088     CPE.A5 WITH bwrtqs.request;
-->089     CA.A5 WITH bwrtqs.request;
-->090   END REFER;
-->091 BEGIN
-->092   CREATE 1 PROCESS OO LIMIT 200 EVERY negexp(1/3);
-->093   CREATE 1 PROCESS OA LIMIT 200 EVERY negexp(1/1.0);
-->094   CREATE 10 PROCESS CPE;
-->095   CREATE 3 PROCESS CA;
-->096 END TYPE UseCaseTest;
-->097 EXPERIMENT UseCaseTest_exp METHOD ANALYTICAL "MARKOV";
% ...
```

Aufrufe, die zuvor mit einer Wahrscheinlichkeit von 100% durchgeführt wurden, geschehen jetzt zur Vermeidung HIT-spezifischer Besonderheiten bei Determinismen in bedingten Verzweigungen unter Verwendung analytischer Löser mit einer etwas geringeren Wahrscheinlichkeit von 99.9999%. Aufrufe in deterministischem Umfang werden in Aufrufe mit cox-verteiltem Umfang umgewandelt. Da das Modell mit dem Löser MARKOV untersucht werden soll, wird unter der Maßgabe, dass sich nur maximal 200 Prozesse dieses Typs im System aufhalten dürfen, alle negexp(1/3) Zeiteinheiten ein Prozess des Typs `OO` erzeugt. Für den Prozess `OA` wird neben der Einführung des Populationslimits auch die Umrechnung der `arrivaltime`-Statements vorgenommen. Die Zwischenankunftszeiten sind entsprechend negexp-verteilt mit einem Erwartungswert von einer Zeiteinheit. Weiterhin werden zehn Prozesse des Typs `CPE` und drei Prozesse des Typs `CA` erzeugt.

Das von HIPE in diesem Test realisierte Verhalten entspricht also dem erwarteten Verhalten.

Nachtrag des Korrektors: *In einem – allerdings vergleichsweise geringfügigen – Punkt war allerdings das erwartete Verhalten falsch. Im HI-SLANG Reference Manual wurde erwähnt, dass Wahrscheinlichkeitsangaben in Modellen, die mit analytischen Lösern ausgewertet werden sollen, unter gewissen Voraussetzungen nicht 0.0 oder 1.0 sein dürfen (vgl. [Büttner u. a. (1999)], Seite 263). Welche Voraussetzungen dies sind, wurde unseres Wissens dort aber nicht erwähnt. Mit verschiedenen Versuchen wurde ermittelt, dass sich diese Einschränkung nur manifestiert, wenn der Löser MARKOV ausgewählt wird. Die entsprechende HIT-Fehlermeldung schien bei jedem während der Implementierung zur Festlegung der eventuell notwendigen Anpassungen benutzten Testmodell aufzutreten. Entsprechend wurde die im Abschnitt 1.2.8 dargestellte Anpassung implementiert, dass Wahrscheinlichkeitsangaben von 1.0 generell auf 0.999999 umgesetzt werden, wenn der Löser MARKOV verwendet wird. Dies ist im Sinne von HIT aber nicht erforderlich, wie sich erst nach dem Abschluss der Arbeiten an HIPE ergab. Vielmehr wäre diese Anpassung nur vonnöten, wenn mehrere Alternativen implementiert werden, weil keiner der Alternativen eine Wahrscheinlichkeit von 0.0 oder 1.0 zugeordnet werden darf. Also stellt diese Anpassung eigentlich eine "Überanpassung" dar. Wie die Rechenbeispiele in den folgenden Abschnitten zeigen werden, verändern sich die Werte für die erhobenen Performancemaße dadurch aber nicht erheblich. In Absprache mit dem Projektgruppenleiter wurde die Anpassung in HIPE nicht mehr abgeändert.*

4.1.2 Test des M/M/1-Modells

Es sei nochmals vermerkt, dass es sich bei den Angaben, die mittels `let responsetime` gemacht werden, grundsätzlich um Bedienzeit*anforderungen* handelt. Es wird damit also angegeben, wie lange der so annotierte Anwendungsfall oder die Aktivität exklusiv und ununterbrochen auf der entsprechenden Maschine ausgeführt werden müsste, wobei sich zu diesem Zeitpunkt keine anderen Anforderungen in Bearbeitung befinden oder auf Bearbeitung warten. Die letztendliche Antwortzeit der Aktivität ergibt sich aber erst unter der tatsächlichen Arbeitslast und ist in vielen Fällen ungleich der Bedienzeitanforderung. Dieses Faktum wird im Folgenden noch einmal durch die Berechnung an zwei einfachen Warteschlangenmodellen verdeutlicht.

Das M/M/1-Modell stellt das erste Rechenbeispiel für Anwendungsfalldiagramme dar. Dieses M/M/1-System wurde bereits im Zwischenbericht der Projektgruppe [PG459 (2005)] diskutiert. An dem System kommen in der Sekunde 1000 Pakete an, welche jeweils eine Bedienzeit von durchschnittlich 200 Operationen benötigen. Diese sollen gemäß der Scheduling-Disziplin RANDOM abgearbeitet werden. Die Bediengeschwindigkeit des Systems ist 400.000 Operationen pro Sekunde.

Für einen entsprechenden HIPE-Durchlauf wurde ein UML-Modell *MM1Test* (siehe Abbildung 4.2 auf S. 130) entwickelt, welches das beschriebene M/M/1-System repräsentiert. Die Hardware wurde mit dem `speed`-Parameter auf die verlangten 400000 Operationen pro Sekunde skaliert, indem der Wert für die Geschwindigkeit der Default-CPU auf 400000 gesetzt wurde, und die Bedienzeitanforderung des Anwendungsfalls *BedienEinheit* auf durchschnittlich 1/2000 Sekunden gesetzt. Dies ergibt sich aus den für die Abarbeitung jedes Pakets benötigten 200 Operationen und der Leistung der Hardware von 400000 Operationen pro Sekunde. Am Akteur wird eine offene Arbeitslast mit einer Prozesserzeugungsrate von durchschnittlich 1000 Prozessen pro Sekunde erzeugt. Analog zu den Rechnungen im Zwischenbericht ergibt sich daraus für die Antwortzeit einer Anfrage:

$$E[r] = \frac{\left(\frac{1}{Bedienrate}\right)}{(1 - Auslastung)} = \frac{\left(\frac{1}{2000}\right)}{\left(\frac{2000}{2000} - \frac{1000}{2000}\right)} = \frac{1}{1000}\ sec.$$

Erwartetes Verhalten
Es wird eine Default-CPU mit einer Geschwindigkeit von 400000 Operationen pro Sekunde erzeugt. Deren Wert für Schedule wird wie gefordert auf RANDOM gesetzt. Da beim Wert für Dispatch nicht vom oben angenommenen Default-Wert abgewichen werden soll, wird EQUAL eingesetzt. Die Zwischenankunftszeit von an die Default-CPU gestellten Anfragen ist negativ exponentialverteilt mit Mittelwert 1/1000 Sekunde. Die Bedienzeitanforderung von im Mittel 1/2000 Sekunde pro Anfrage wird auf Operationen umgerechnet. Damit ergibt sich für eine Anfrage eine Bedienzeitanforderung von im Mittel 400000 * 1/2000 = 200 Operationen. Diese ist negativ exponentialverteilt, die Angabe lautet also negexp(1/200).

Je nach dem ausgewählten Löser werden die für diesen relevanten STOP-Bedingungen gesetzt. Das heißt, dass für den Löser ANALYTICAL_MARKOV ein Abbruch nach 120 Sekunden Rechenzeit oder bei einer erzielten Wert für ACCURACY von 0.1 erfolgt, für den Löser SIMULATIVE nach 120 Sekunden Rechenzeit, 10000 Sekunden Modellzeit oder falls der wahre Mittelwert für ein Leistungsmaß mit 99%-iger Wahrscheinlichkeit in einem vom während der

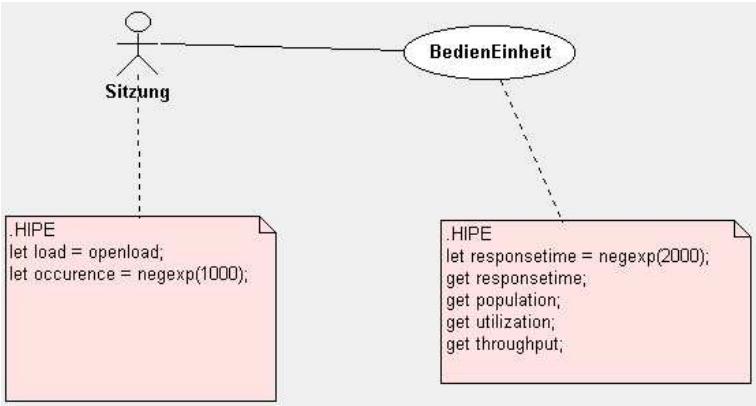

Abbildung 4.2: Use-Case-Diagramm *MM1Test* für das Rechenbeispiel mit einem M/M/1-System

Simulation berechneten Mittelwert nach unten und nach oben jeweils um 1 abweichenden Intervall liegt (in HI-SLANG ausgedrückt als `CONFIDENCE LEVEL 99 WIDTH 1`).

Als erwartete Ergebnisse für die am atomaren Anwendungsfall – und damit im Sinne des HIPE-Paradigmas direkt an der ausführenden Default-CPU – zu erhebenden Leistungsmaße werden die Maße aus einem Vergleichsmodell zugrunde gelegt. Dieses wurde in HITGraphic angelegt (siehe Abbildung 4.3 auf S. 131) und einer Analyse mittels HIT unterzogen. Die Ergebnisse dieser Analyse sind in der Tabelle 4.1 aufgeführt.

	ANALYTICAL "MARKOV"	SIMULATIVE
Population	1.000000	0.991549
Throughput	1000	999.014526
Turnaroundtime	0.001000	0.0009933
Utilization	0.500000	0.498267

Tabelle 4.1: Die Ergebnisse für das M/M/1-Modell (HIT)

Realisiertes Verhalten
HIPE erzeugt mit dieser Eingabe folgenden HI-SLANG-Code für das Lösungsverfahren MARKOV (die für MARKOV nicht zulässigen Schätzer *Standard Deviation*, *Confidence Level* und *Bounds* werden dabei von HIT ignoriert):

```
-->001 %ANALYZER
-->002 %PARM=UPDATES
-->003 %END
-->004 TYPE MM1Test MODEL;
-->005   TYPE Sitzung SERVICE;
-->006     USE
-->007       SERVICE
-->008         BedienEinheit (AMOUNT : REAL);
-->009     END USE;
-->010     BEGIN
-->011       BRANCH
-->012       PROB 1.0: BedienEinheit(negexp(1/200));
-->013       END BRANCH;
-->014   END TYPE Sitzung;
-->015   COMPONENT qkrihr : server (LET schedule := random,LET dispatch := equal(LET speed := 400000));
```

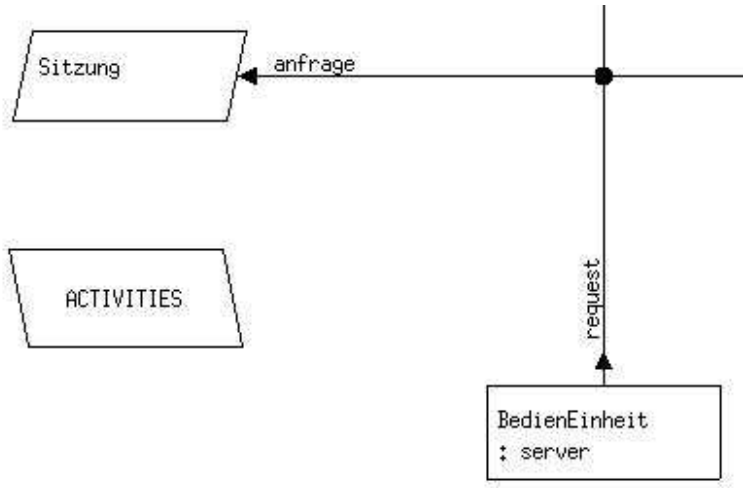

Abbildung 4.3: M/M/1–Modell in HITGraphic

```
-->016   REFER Sitzung TO qkrihr EQUATING
-->017     Sitzung.BedienEinheit WITH qkrihr.request;
-->018   END REFER;
-->019 BEGIN
-->020   CREATE 1 PROCESS Sitzung LIMIT 200 EVERY negexp(1000);
-->021 END TYPE MM1Test;
-->022 EXPERIMENT MM1Test_exp METHOD ANALYTICAL "MARKOV";
-->023 BEGIN
-->024   EVALUATE
-->025     MODEL hipemodel : MM1Test;
-->026     EVALUATIONOBJECT
-->027       x0_eva VIA hipemodel.qkrihr
-->028     DEFAULT
-->029       ESTIMATOR MEAN, STANDARDDEVIATION, CONFIDENCE LEVEL 97, BOUNDS
-->030       OUTPUT  TABLE "TABLE",
-->031               DUMPFILE "DUMP";
-->032   BEGIN
-->033     MEASURE TURNAROUNDTIME, POPULATION, UTILIZATION, THROUGHPUT
-->034       AT x0_eva;
-->035     CONTROL
-->036       STOP CPUTIME 120
-->037       OR   ACCURACY 0.1;
-->038   END EVALUATE;
-->039 END EXPERIMENT MM1Test_exp;
```

Es wird eine Komponente mit zufälligem Namen erzeugt, deren Geschwindigkeit 400000 Operationen pro Sekunde beträgt. Die realisierte Scheduling-Strategie ist RANDOM, der Wert für Dispatch ist EQUAL. Die Zwischenankunftszeit von Anforderungen an diese Komponente ist negativ exponentialverteilt mit einem Mittelwert von 1/1000 Sekunde, und die Bedienzeitanforderung jeder Anfrage beträgt im Mittel 200 Operationen.

Gemäß der STOP-Statements werden die Berechnungen eingestellt, falls die CPU-Zeit den Wert 120 über- oder der prozentuale Fehler (ACCURACY) den Wert 0.1 (= 10%) unterschreitet. Man beachte weiter das Limit bei der Prozesserzeugung, das als Anpassung für den Löser MARKOV gesetzt wurde.

Auch ein Lauf mit dem simulativen Lösungsverfahren wurde durchgeführt. Dabei wurde folgender Code erzeugt:

```
-->001 %ANALYZER
-->002 %PARM=UPDATES
-->003 %END
```

```
-->004 TYPE MM1Test MODEL;
-->005   TYPE Sitzung SERVICE;
-->006     USE
-->007       SERVICE
-->008         BedienEinheit (AMOUNT : REAL);
-->009     END USE;
-->010     BEGIN
-->011       BRANCH
-->012       PROB 1.0: BedienEinheit(negexp(1/200));
-->013       END BRANCH;
-->014   END TYPE Sitzung;
-->015   COMPONENT rtwhkk : server (LET schedule := immediate,LET dispatch := equal(LET speed := 400000));
-->016   REFER Sitzung TO rtwhkk EQUATING
-->017     Sitzung.BedienEinheit WITH rtwhkk.request;
-->018   END REFER;
-->019 BEGIN
-->020   CREATE 1 PROCESS Sitzung EVERY negexp(1000);
-->021 END TYPE MM1Test;
-->022 EXPERIMENT MM1Test_exp METHOD SIMULATIVE;
-->023 BEGIN
-->024   EVALUATE
-->025     MODEL hipemodel : MM1Test;
-->026     EVALUATIONOBJECT
-->027       x0_eva VIA hipemodel.rtwhkk
-->028     DEFAULT
-->029       ESTIMATOR MEAN, STANDARDDEVIATION, CONFIDENCE LEVEL 97, BOUNDS
-->030       OUTPUT  TABLE "TABLE",
-->031               DUMPFILE "DUMP";
-->032   BEGIN
-->033     MEASURE TURNAROUNDTIME, POPULATION, UTILIZATION, THROUGHPUT
-->034       AT x0_eva;
-->035     CONTROL
-->036       AT x0_eva
-->037         STOP CPUTIME 120
-->038         OR   MODELTIME 10000
-->039         OR   CONFIDENCE LEVEL 99 WIDTH 1
-->040              MEASURE TURNAROUNDTIME
-->041         OR   CONFIDENCE LEVEL 99 WIDTH 1
-->042              MEASURE POPULATION
-->043         OR   CONFIDENCE LEVEL 99 WIDTH 1
-->044              MEASURE UTILIZATION
-->045         OR   CONFIDENCE LEVEL 99 WIDTH 1
-->046              MEASURE THROUGHPUT;
-->047   END EVALUATE;
-->048 END EXPERIMENT MM1Test_exp;
```

Die STOP-Statements sind so gewählt, dass die Simulation entweder nach 120 Sekunden Berechnungszeit, 10000 Sekunden Modellzeit, oder wenn für ein Leistungsmaß die geforderte Konfidenz erreicht wird, abbricht.

Sowohl der Code, den HIPE für die beiden Verfahren erzeugt, als auch die Ergebnisse der harten Analyse (siehe Tabelle 4.2) sind mit denen des Modellierungspraktikums (Tabelle 4.1) vergleichbar. Die simulativen Werte der Tabelle sind dabei immer als die Werte des Estimators *Mean* zu verstehen.

	ANALYTICAL "MARKOV"	SIMULATIVE
Population	1.000000	0.970750
Throughput	1000	996.006573
Turnaroundtime	0.001000	0.000975
Utilization	0.500000	0.493277

Tabelle 4.2: Die Ergebnisse für das M/M/1-Modell (HIPE)

Das Verhalten von HIPE auf dem M/M/1-Modell entspricht somit dem erwarteten Verhalten.

4.1.3 Test des M/M/1||M/M/1-Modells

Das M/M/1||M/M/1-Modell ist ein weiteres Rechenbeispiel auf Basis einfacher Wartesysteme. Dieses wurde ebenfalls im Modellierungspraktikum behandelt. An einem System aus zwei identischen parallelen M/M/1-Teilsystemen kommen Pakete mit einer Ankunftsrate $\lambda = 1000$ Pakete/sec an. Jedes Paket verlangt eine Bedienung von im Mittel 200 Operationen. Die Bediengeschwindigkeit jedes Teilsystems sei 200.000 Operationen/sec, deren Bedienstrategie ist FIFO (bzw. FCFS). Durch eine Analyse mit HIT sollten die mittlere Auslastung, die mittlere Warteschlangenlänge, die mittlere Durchlaufzeit und der mittlere Durchsatz ermittelt werden und zwar für die beiden M/M/1-Teilsysteme sowie für das Gesamtsystem.

Das entsprechende UML-Modell *MM1ParaTest* zu diesem System besteht aus einem Use-Case-Diagramm, welches zwei Anwendungsfälle und einen Akteur enthält, und einem Verteilungsdiagramm. Am Akteur wird eine offene Arbeitslast erzeugt, wobei pro Sekunde eintausend Anfragen an das System herangetragen werden, die dann jeweils zur Hälfte von einer der beiden Bedieneinheiten ausgeführt werden. Die Bedienzeitanforderung für eine Anfrage ist im Mittel 1/1000 Sekunde. Im Verteilungsdiagramm für das Modell wird die Geschwindigkeit der beiden Bediener jeweils auf 200000 Operationen pro Sekunde, die Bedienstrategie auf FCFS gesetzt. Aus dem Zwischenbericht folgen somit für die erwartete Antwortzeit für das Gesamt- und die Einzelsysteme:

$$E[r]_{einzel} = \frac{Population_{einzel}}{Durchsatz_{einzel}} = \frac{1}{500}\ sec.$$

$$E[r]_{gesamt} = \frac{Population_{gesamt}}{Durchsatz_{gesamt}} = \frac{2}{1000} = \frac{1}{500}\ sec.$$

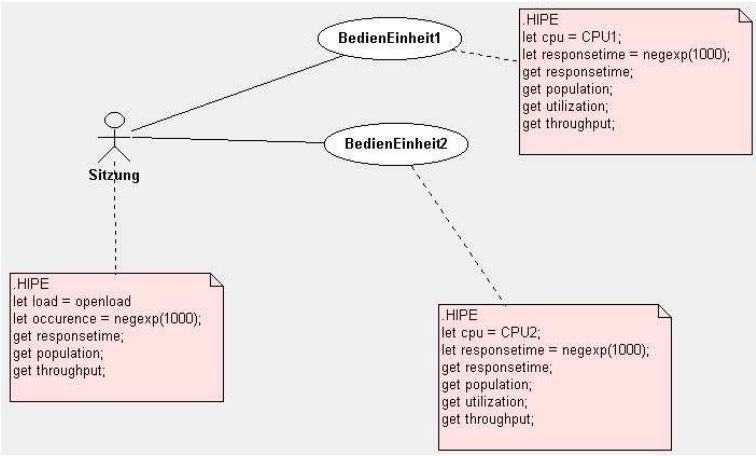

Abbildung 4.4: Use-Case-Diagramm *MM1ParaTest* des M/M/1||M/M/1-Systems

Erwartetes Verhalten
Es werden gemäß der Angaben im Verteilungsdiagramm zwei Server mit den Namen `CPU1` und `CPU2` erzeugt, deren Geschwindigkeit 200000 Operationen pro Sekunde beträgt. Als Scheduling-Disziplin wird zunächst FCFS eingesetzt, als Dispatch-Disziplin der im Modell nicht veränderte Default-Wert EQUAL. Die Zwischenankunftszeit für an das System gestellte Anfragen

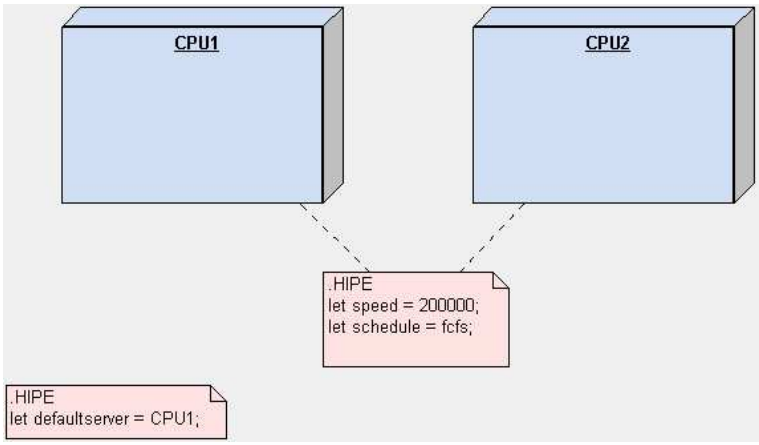

Abbildung 4.5: Verteilungsdiagramm zum Modell des M/M/1||M/M/1-Systems

ist negativ exponentialverteilt mit Mittelwert 1/1000 Sekunde. Eine Anfrage wird mit jeweils 50%-iger Wahrscheinlichkeit einem der beiden Server zugeordnet, wo sie eine Bedienzeitanforderung von im Mittel 200000 * 1/1000 = 200 Operationen stellt. Diese Angabe ist negativ exponentialverteilt, die Anforderung wird im Modell also als negexp(1/200) dargestellt.

Anhand des in HITGraphic erstellten Modells des M/M/1||M/M/1-Systems, das in Abbildung 4.6 dargestellt wird, wurden jeweils die Ergebnisse für einen einzelnen Bediener und für das Gesamtsystem mit Hilfe der beiden HIT-Löser ANALYTICAL "MARKOV' und SIMULATIVE berechnet. Die Ergebnisse dieser Berechnungen finden sich in den Tabellen 4.3 und 4.4.

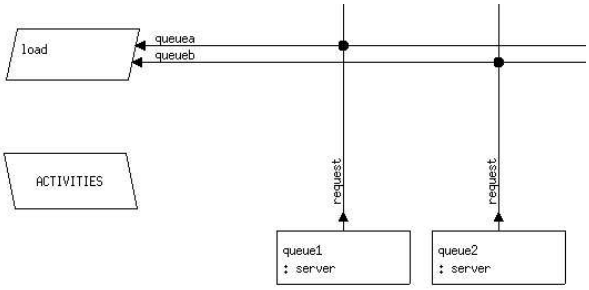

Abbildung 4.6: Leistungsmodell des M/M/1||M/M/1-Systems in HITGraphic

Für die Erzeugung des Codes ist weiterhin zu bemerken, dass für analytische Löser das Verfahren FCFS nicht verwendet werden kann. HIPE ändert selbständig den Parameter `schedule` für die betreffenden Komponenten auf `random`.

Realisiertes Verhalten

Zunächst wird der von HIPE erzeugte Code für das simulative Lösungsverfahren betrachtet.

```
-->001 %ANALYZER
-->002 %PARM=UPDATES
```

Einzelsystem	Population	Throughput	Turnaroundtime	Utilization
Mean	0.984332	497.110085	0.001980	0.495804
Stdev	1.388167	0.002015	0.001967	0.499982
Con 95%	± 2.83%	± 0.86%	± 2.46%	± 1.21%
Gesamtsystem	Population	Throughput	Turnaroundtime	
Mean	1.977333	998.572255	0.001980	
Stdev	1.967741	0.000998	0.001975	
Con 95%	± 2.00%	± 0.61%	± 1.71%	

Tabelle 4.3: Ergebnisse für das Einzel- und Gesamtsystem mit SIMULATIVE (HIT)

Mean	Population	Throughput	Turnaroundtime	Utilization
Einzelsystem	1.006277	500.383736	0.002011	0.500384
Gesamtsystem	2.012553	1000.767472	0.002011	

Tabelle 4.4: Ergebnisse für das Einzel- und Gesamtsystem mit ANALYTICAL "MARKOV" (HIT)

```
-->003 %END
-->004 TYPE MM1ParaTest MODEL;
-->005   TYPE Sitzung SERVICE;
-->006     USE
-->007       SERVICE
-->008         BedienEinheit2 (AMOUNT : REAL);
-->009         BedienEinheit1 (AMOUNT : REAL);
-->010     END USE;
-->011     BEGIN
-->012       BRANCH
-->013       PROB 0.5: BedienEinheit2(negexp(0.0050));
-->014       PROB 0.5: BedienEinheit1(negexp(0.0050));
-->015       END BRANCH;
-->016   END TYPE Sitzung;
-->017   COMPONENT CPU2 : server (LET schedule := fcfs,LET dispatch := equal(LET speed := 200000));
-->018   COMPONENT CPU1 : server (LET schedule := fcfs,LET dispatch := equal(LET speed := 200000));
-->019   REFER Sitzung TO CPU2,CPU1 EQUATING
-->020     Sitzung.BedienEinheit2 WITH CPU2.request;
-->021     Sitzung.BedienEinheit1 WITH CPU1.request;
-->022   END REFER;
-->023 BEGIN
-->024   CREATE 1 PROCESS Sitzung EVERY negexp(1000);
-->025 END TYPE MM1ParaTest;
-->026 EXPERIMENT MM1ParaTest_exp METHOD SIMULATIVE;
-->027 BEGIN
-->028   EVALUATE
-->029     MODEL hipemodel : MM1ParaTest;
-->030     EVALUATIONOBJECT
-->031       x0_eva VIA hipemodel,
-->032       x1_eva VIA hipemodel.CPU2,
-->033       x2_eva VIA hipemodel.CPU1
-->034     DEFAULT
-->035       ESTIMATOR MEAN, STANDARDDEVIATION, BOUNDS
-->036       OUTPUT  TABLE "TABLE",
-->037               DUMPFILE "DUMP";
-->038   BEGIN
-->039     MEASURE TURNAROUNDTIME, POPULATION, THROUGHPUT
-->040       AT x0_eva;
-->041     MEASURE TURNAROUNDTIME, POPULATION, UTILIZATION, THROUGHPUT
-->042       AT x1_eva;
-->043     MEASURE TURNAROUNDTIME, POPULATION, UTILIZATION, THROUGHPUT
-->044       AT x2_eva;
-->045     CONTROL
-->046       AT x0_eva
-->047         STOP CPUTIME 120
-->048         OR   MODELTIME 10000
-->049         OR   CONFIDENCE LEVEL 99 WIDTH 1
-->050              MEASURE TURNAROUNDTIME
-->051         OR   CONFIDENCE LEVEL 99 WIDTH 1
-->052              MEASURE POPULATION
```

```
-->053         OR   CONFIDENCE LEVEL 99 WIDTH 1
-->054              MEASURE THROUGHPUT
-->055       AT x1_eva
-->056         STOP CPUTIME 120
-->057         OR   MODELTIME 10000
-->058         OR   CONFIDENCE LEVEL 99 WIDTH 1
-->059              MEASURE TURNAROUNDTIME
-->060         OR   CONFIDENCE LEVEL 99 WIDTH 1
-->061              MEASURE POPULATION
-->062         OR   CONFIDENCE LEVEL 99 WIDTH 1
-->063              MEASURE UTILIZATION
-->064         OR   CONFIDENCE LEVEL 99 WIDTH 1
-->065              MEASURE THROUGHPUT
-->066       AT x2_eva
-->067         STOP CPUTIME 120
-->068         OR   MODELTIME 10000
-->069         OR   CONFIDENCE LEVEL 99 WIDTH 1
-->070              MEASURE TURNAROUNDTIME
-->071         OR   CONFIDENCE LEVEL 99 WIDTH 1
-->072              MEASURE POPULATION
-->073         OR   CONFIDENCE LEVEL 99 WIDTH 1
-->074              MEASURE UTILIZATION
-->075         OR   CONFIDENCE LEVEL 99 WIDTH 1
-->076              MEASURE THROUGHPUT;
-->077    END EVALUATE;
-->078 END EXPERIMENT MM1ParaTest_exp;
```

Es werden zwei Server `CPU1` und `CPU2` erzeugt, deren Geschwindigkeit jeweils 200000 Operationen pro Sekunde beträgt. Die Scheduling-Disziplin ist FCFS, die Dispatch-Disziplin ist EQUAL. Mit einem negativ exponentialverteilten Abstand von im Mittel 1/1000 Sekunde werden an das System Anfragen gestellt. Diese werden mit jeweils 50% an einen der beiden Server weitergeleitet. Der Umfang jeder Bedienanfrage ist negativ exponentialverteilt mit im Mittel $1/0.0050 = 200$ Operationen.

Der Code, den HIPE für die Benutzung des Verfahrens ANALYTICAL "MARKOV" generiert, sieht folgendermaßen aus:

```
-->001 %ANALYZER
-->002 %PARM=UPDATES
-->003 %END
-->004 TYPE MM1ParaTest MODEL;
-->005    TYPE Sitzung SERVICE;
-->006      USE
-->007        SERVICE
-->008          BedienEinheit2 (AMOUNT : REAL);
-->009          BedienEinheit1 (AMOUNT : REAL);
-->010      END USE;
-->011      BEGIN
-->012        BRANCH
-->013        PROB 0.5: BedienEinheit2(negexp(0.0050));
-->014        PROB 0.5: BedienEinheit1(negexp(0.0050));
-->015        END BRANCH;
-->016    END TYPE Sitzung;
-->017    COMPONENT CPU2 : server (LET schedule := random,LET dispatch := equal(LET speed := 200000));
-->018    COMPONENT CPU1 : server (LET schedule := random,LET dispatch := equal(LET speed := 200000));
-->019    REFER Sitzung TO CPU2,CPU1 EQUATING
-->020      Sitzung.BedienEinheit2 WITH CPU2.request;
-->021      Sitzung.BedienEinheit1 WITH CPU1.request;
-->022    END REFER;
-->023 BEGIN
-->024    CREATE 1 PROCESS Sitzung LIMIT 200 EVERY negexp(1000);
-->025 END TYPE MM1ParaTest;
-->026 EXPERIMENT MM1ParaTest_exp METHOD ANALYTICAL "MARKOV";
-->027 BEGIN
-->028    EVALUATE
-->029      MODEL hipemodel : MM1ParaTest;
-->030      EVALUATIONOBJECT
-->031        x0_eva VIA hipemodel,
-->032        x1_eva VIA hipemodel.CPU2,
-->033        x2_eva VIA hipemodel.CPU1
-->034      DEFAULT
-->035        ESTIMATOR MEAN, STANDARDDEVIATION, BOUNDS
-->036        OUTPUT  TABLE "TABLE",
-->037                DUMPFILE "DUMP";
-->038    BEGIN
-->039      MEASURE TURNAROUNDTIME, POPULATION, THROUGHPUT
-->040        AT x0_eva;
-->041      MEASURE TURNAROUNDTIME, POPULATION, UTILIZATION, THROUGHPUT
-->042        AT x1_eva;
-->043      MEASURE TURNAROUNDTIME, POPULATION, UTILIZATION, THROUGHPUT
```

```
-->044       AT x2_eva;
-->045     CONTROL
-->046       STOP CPUTIME 120
-->047       OR   ACCURACY 0.1;
-->048   END EVALUATE;
-->049 END EXPERIMENT MM1ParaTest_exp;
```

Festzustellen ist, dass für die Komponenten, die im simulativen Löser auf FCFS-Scheduling konfiguriert werden, in der Modellbeschreibung für den analytischen Löser die Scheduling-Disziplin RANDOM eingesetzt wurde.

Die Ergebnisse des HIPE-Durchlaufs mit diesen Eingaben für den simulativen bzw. analytisch-numerischen Löser sind in den Tabellen 4.5 und 4.6 zu finden. Auch hier sind die von HIT und HIPE gelieferten Ergebnisse quasi identisch.

Einzelsystem	Population	Throughput	Turnaroundtime	Utilization
Mean	0.951938	491.249294	0.001938	0.488134
Stdev	1.346303	0.002041	0.001909	0.499859
Con 95%	± 8.74%	± 2.80%	± 7.67%	± 3.89%
Gesamtsystem	Population	Throughput	Turnaroundtime	
Mean	1.973886	999.899186	0.001974	
Stdev	1.975966	0.000991	0.001961	
Con 95%	± 6.45%	± 1.99%	± 5.34%	

Tabelle 4.5: Ergebnisse für das Einzel- und Gesamtsystem mit SIMULATIVE (HIPE)

Mean	Population	Throughput	Turnaroundtime	Utilization
Einzelsystem	1.005467	500.43546	0.001889	0.500231
Gesamtsystem	2.013123	1000.755893	0.002003	

Tabelle 4.6: Ergebnisse für das Einzel- und Gesamtsystem mit ANALYTICAL "MARKOV" (HIPE)

4.2 Tests mit Aktivitätsdiagrammen

Im Folgenden wird mit verschiedenen Aktivitätsdiagrammen getestet, ob HIPE die verschiedenen im HIPE-Paradigma für eine Leistungsbewertung betrachteten Elemente korrekt umsetzt. Es wird zunächst ein einfaches Diagramm betrachtet, das verschiedene Aktivitäten enthält, die umgesetzt werden sollen. In weiteren Diagrammen wird getestet, wie Verzweigungen, Objektflüsse, Concurrent- und Loop-Blöcke umgesetzt werden. Die beiden letzten Umsetzungen sind besonders interessant, da HIPE dabei automatisch selbstdefinierte Komponenten erzeugt.

Das Anwendungsfalldiagramm *ActivityTest* zum betrachteten Testmodell findet sich in Abbildung 4.7 auf S. 138. Dargestellt wird eine offene Arbeitslast mit einer Prozesserzeugungsrate von 1 Prozess pro Zeiteinheit. Mit dem entsprechenden Akteur sind die verschiedenen Anwendungsfälle assoziiert, von denen jeder auf ein Aktivitätsdiagramm verweist. Mit dem daran

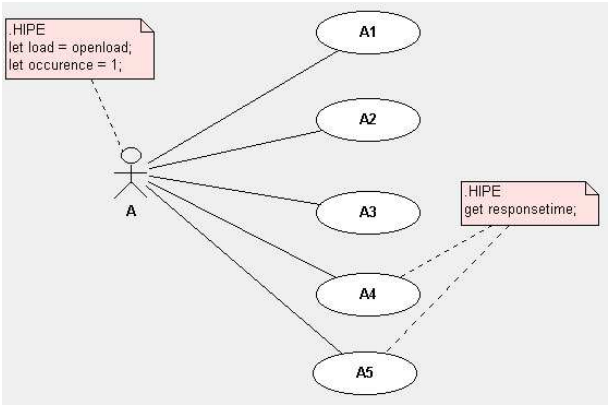

Abbildung 4.7: Anwendungsfalldiagramm *ActivityTest* zum Testmodell für Aktivitätsdiagramme

vermerkten `get`-Statement wird angezeigt, dass die mittlere Durchlaufzeit eines Aufrufs dieses Anwendungsfalls ermittelt werden soll. Diese wird im Folgenden nur in den Rechenbeispielen in den Abschnitten 4.2.4, 4.2.5 und 4.2.6 betrachtet, da dort die betrachteten Aktivitätsdiagramme einfach genug ist, um die von der harten Analyse gelieferten Ergebnisse manuell zu überprüfen bzw. bereits Messwerte aus anderen Implementierungen des gleichen Modells existieren, mit denen die mittels HIT erhobenen Maße verglichen werden können.

Weiterhin wird, soweit im jeweiligen Unterkapitel nicht explizit angegeben, in diesem Kapitel ein Verteilungsdiagramm verwendet, das in der Abbildung 4.8 auf S. 138 zu finden ist. Dieses stellt zwei CPUs und die Leitung dar, die diese verbindet.

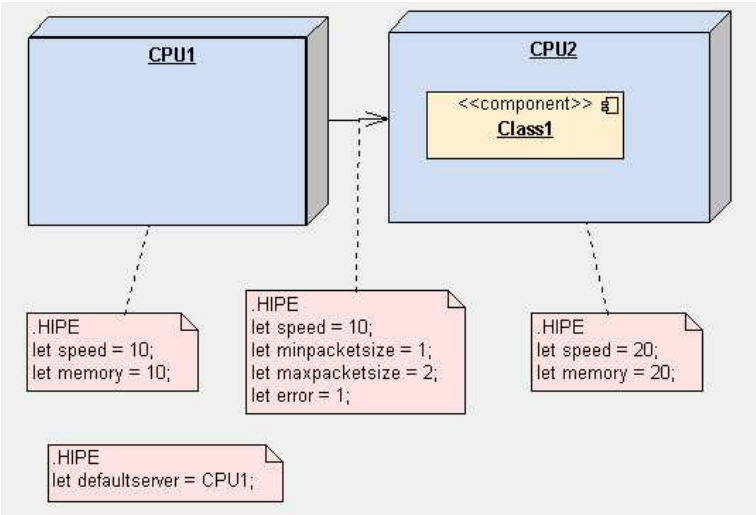

Abbildung 4.8: Das Verteilungsdiagramm zum Test mit Aktivitätsdiagrammen

Erwartetes Verhalten

Zunächst wird die Umsetzung des Anwendungsfalldiagramms im Kontext der anderen im Modell verwendeten Diagramme betrachtet. Werden zusätzlich zum Anwendungsfalldiagramm noch ein Verteilungsdiagramm und ein bis mehrere Aktivitätsdiagramme definiert, so wirkt sich bei der Erzeugung von Komponenten die Hierarchieregel (vgl. Seite 42) aus. Das bedeutet, dass Komponenten nicht mehr unbedingt auf der MODEL-Ebene, sondern im allgemeinen auf der niedrigsten Hierarchieebene definiert werden, die für alle Komponenten, in denen von diesen bereitgestellte Dienste (provided services) zugegriffen werden, gemeinsam sichtbar ist. Es erscheinen zunächst jeweils die Verweise auf die Diagramme *A1* bis *A5*. Diese Aktivitätsdiagramme werden in den Kapiteln 4.2.1 bis 4.2.5 behandelt. Die Namen der Komponenten sind gemäß der Namensregel für von Anwendungsfalldiagrammen aus verwiesene Aktivitätsdiagramme (siehe die Beschreibungen auf Seite 45 und auf Seite 125) gleich `AA1` bis `AA5`.

Die Aktivitätsdiagramme *A1* bis *A4* sind dabei so konstruiert, dass dort stets alle im Verteilungsdiagramm definierten CPUs (*CPU1* und *CPU2*) verwendet werden. Deshalb wird in diesem Beispiel erwartet, dass die Komponenten, die diese darstellen, auf der MODEL-Ebene erscheinen. Die Namen dieser Komponenten sind dabei identisch mit denen, die im Verteilungsdiagramm verwendet werden.

Die im Verteilungsdiagramm zusätzlich modellierte Leitung zwischen *CPU1* und *CPU2* wird nur im Diagramm *A3* verwendet, das dem Test der korrekten Implementierung des Objektflusses – siehe Kapitel 4.2.3 – dient. Deshalb erscheint diese Leitung nicht auf der hier dargestellten MODEL-Ebene, sondern auf der Ebene der selbstdefinierten Komponente, die das Diagramm *A3* darstellt. Die `let memory`-Annotationen der CPUs und die Charakteristika der Leitung spielen ebenfalls nur im Kontext der Modellierung des Objektflusses eine Rolle und werden daher auf der Ebene des Anwendungsfalldiagramms nicht codiert.

Zusätzlich wird im Rahmen eines EXPERIMENT-Blocks angegeben, dass an den Komponenten `AA4` und `AA5` jeweils die Antwortzeit, das heisst die Dauer eines Durchlaufs des jeweiligen Diagramms, zu erheben ist.

Realisiertes Verhalten

Im Folgenden werden die Abschnitte im von HIPE für das Gesamtmodell erzeugten HISLANG-Quellcode betrachtet, die im Zusammenhang mit dem umzusetzenden Anwendungsfalldiagramm relevant sind.

```
-->001 %ANALYZER
-->002 %PARM=UPDATES
-->003 %END
-->004 %COPY "PRIOSERVER"
-->005 TYPE ActivityTest MODEL;
-->006   TYPE A SERVICE;
-->007     USE
-->008       SERVICE
-->009         AA1_1;
-->010         AA2_1;
-->011         AA3_1;
-->012         AA4_1;
-->013         AA5_1;
-->014     END USE;
-->015     BEGIN
-->016       BRANCH
-->017       PROB 0.2: AA1_1;
-->018       PROB 0.2: AA2_1;
-->019       PROB 0.2: AA3_1;
-->020       PROB 0.2: AA4_1;
-->021       PROB 0.2: AA5_1;
-->022       END BRANCH;
-->023   END TYPE A;
```

```
% Codierung der einzelnen Aktivitätsdiagramme und exklusiv dort
% benutzten Komponenten wird in den folgenden Abschnitten betrachtet
-->370   COMPONENT CPU1 : server (LET schedule := immediate,LET dispatch := equal(LET speed := 10));
-->371   COMPONENT CPU2 : server (LET schedule := immediate,LET dispatch := equal(LET speed := 20));
-->372   COMPONENT AA1_1 : AA1;
-->373   COMPONENT AA2_1 : AA2;
-->374   COMPONENT AA3_1 : AA3;
-->375   COMPONENT AA4_1 : AA4;
-->376   COMPONENT AA5_1 : AA5;
-->377   REFER A TO AA1_1,AA2_1,AA3_1,AA4_1,AA5_1 EQUATING
-->378     A.AA1_1 WITH AA1_1.request;
-->379     A.AA2_1 WITH AA2_1.request;
-->380     A.AA3_1 WITH AA3_1.request;
-->381     A.AA4_1 WITH AA4_1.request;
-->382     A.AA5_1 WITH AA5_1.request;
-->383   END REFER;
-->384 BEGIN
-->385   CREATE 1 PROCESS A EVERY 1;
-->386 END TYPE ActivityTest;
-->387 EXPERIMENT ActivityTest_exp METHOD SIMULATIVE;
-->388 BEGIN
-->389   EVALUATE
-->390     MODEL hipemodel : ActivityTest;
-->391     EVALUATIONOBJECT
-->392       x0_eva VIA hipemodel.AA4_1,
-->393       x1_eva VIA hipemodel.AA5_1
-->394     DEFAULT
-->395       ESTIMATOR MEAN, STANDARDDEVIATION, BOUNDS
-->396       OUTPUT  TABLE "TABLE",
-->397               DUMPFILE "DUMP";
-->398   BEGIN
-->399     MEASURE TURNAROUNDTIME
-->400       AT x0_eva;
-->401     MEASURE TURNAROUNDTIME
-->402       AT x1_eva;
-->403     CONTROL
-->404       AT x0_eva
-->405         STOP CPUTIME 120
-->406         OR   MODELTIME 10000
-->407         OR   CONFIDENCE LEVEL 99 WIDTH 1
-->408              MEASURE TURNAROUNDTIME
-->409       AT x1_eva
-->410         STOP CPUTIME 120
-->411         OR   MODELTIME 10000
-->412         OR   CONFIDENCE LEVEL 99 WIDTH 1
-->413              MEASURE TURNAROUNDTIME;
-->414   END EVALUATE;
-->415 END EXPERIMENT ActivityTest_exp;
```

Im MODEL-Block mit dem Namen ActivityTest wird zunächst das Anwendungsfalldiagramm codiert: Es werden Used Services `AA1` bis `AA5` definiert, die über den REFER-Block mit selbstdefinierten Komponenten des jeweils gleichen Namens assoziiert sind. Diese werden im Rahmen einer Anfrage mit jeweils 20%-iger Wahrscheinlichkeit aufgerufen. Im EXPERIMENT-Block wird verlangt, dass an den Komponenten `AA4` und `AA5` jeweils der Wert für TURNAROUNDTIME erhoben wird. Weiterhin werden zwei Server `CPU1` und `CPU2` definiert.

Im Sinne des erwarteten Verhaltens arbeitet HIPE also an dieser Stelle korrekt.

4.2.1 Test mit Aktivitäten

Das getestete Aktivitätsdiagramm *A1* findet sich in Abbildung 4.9 auf S. 141. Es enthält jeweils eine Aktivität für jede der möglichen Annotationen, mit der Aktivitäten versehen werden können. Ferner wird ein Verteilungsdiagramm erzeugt, das zwei CPUs enthält, um die Zuweisung einer Aktivität zu einer CPU zu testen. Dieses Diagramm findet sich in Abbildung 4.8 auf S. 138.

Zu bemerken ist, dass eine gleichzeitige Verwendung von `LET class = ...` und `LET method = ...` in diesem Aktivitätsdiagramm nicht vorkommt, da diese in der aktuellen Version von HIPE nicht implementiert wurde.

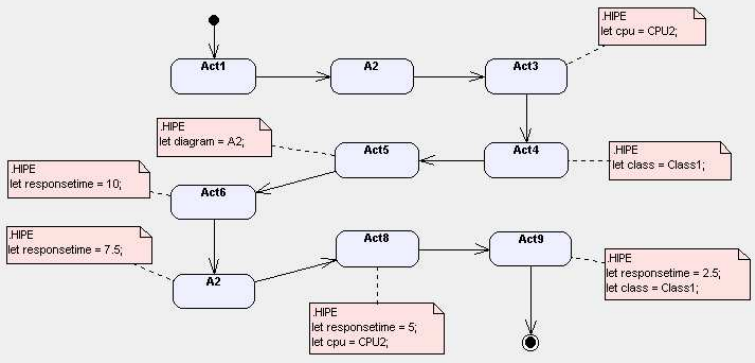

Abbildung 4.9: Das im Test verwendete Aktivitätsdiagramm *A1*

Erwartetes Verhalten

Analog zur Betrachtung der Namensregeln für Anwendungsfalldiagramme werden im Folgenden die Namensregeln für Aktivitätsdiagramme betrachtet. Diese folgen der Darstellung in Tabelle 1.6 und der dazugehörigen Beschreibung ab Seite 49. Grundsätzlich kann eine Aktivität auf drei verschiedene Arten einer Maschine zugeordnet werden, wobei jeweils mehrere Möglichkeiten bestehen, einen Verweis der entsprechenden Art zu generieren. Davon, welche der Gegebenheiten realisiert wird, wird die Namensvergabe des Used Service abhängig gemacht, der die Ausführung dieser Aktivität darstellt.

- Die CPU, auf der sie ausgeführt werden soll, kann explizit angegeben werden. Der Name des die Ausführung der Aktivität darstellenden Used Service setzt sich dann aus dem Namen dieser explizit verwiesenen CPU und einem Nummerierungssuffix zusammen. Das Nummerierungssuffix eines Used Service gibt dann jeweils an, wieviele Aktivitäten während des Diagrammdurchlaufs bisher auf dieser CPU ausgeführt werden, und das Maximum der Nummerierungssuffizes für eine CPU schließlich die Gesamtanzahl aller Aktivitäten, die auf dieser CPU ausgeführt werden.
- Es besteht die Möglichkeit, implizit oder explizit auf ein weiteres Aktivitätsdiagramm zu verweisen. Dieses wird als eine selbstdefinierte Komponente betrachtet, deren Name sich aus dem Präfix `A` für "Aktivitätsdiagramm", dem Namen des Aktivitätsdiagramms und einem Nummerierungssuffix zusammensetzt, der ebenso interpretiert wird wie der Nummerierungssuffix bei Verweisen auf eine CPU.
- Wenn beide andere Möglichkeiten nicht realisiert werden können, wird eine Ausführung auf der im Verteilungsdiagramm unter der Annotation `defaultserver` verwiesenen CPU angenommen. Deren Name ergibt sich aus dem Namen der im Verteilungsdiagramm unter der Annotation `defaultserver` verwiesenen Komponente und einem entsprechenden Nummerierungssuffix.

Weiterhin muss der Umfang bestimmt werden, in dem die auszuführende Aktivität die angegebene Maschine belastet, wobei es auch hier verschiedene Möglichkeiten gibt.

- Einem Verweis auf ein Aktivitätsdiagram wird kein Umfang zugeordnet, dieser ergibt sich im Zuge der Analyse mit HIT durch die Auswertung der selbstdefinierten Komponente, die das entsprechende Aktivitätsdiagramm darstellt.

- Bei einer atomaren Aktivität, das heisst einer Aktivität, die auf eine CPU abgebildet wird, wird versucht, möglichst viele Angaben aus dem Modell selbst zu extrahieren. Für fehlende Angaben werden Default-Werte eingesetzt. Eine vollständige Angabe besteht aus einer `responsetime`-Angabe r (in Sekunden) für die Aktivität und einer `speed`-Angabe s (in Operationen pro Sekunde) für die ausführende CPU. Der Bedienzeitumfang wird in Operationen angegeben und ergibt sich als $r \cdot s$.

Gemäß der in der Tabelle 1.6 angegebenen Umsetzungsvorschriften soll folgende Umsetzung der einzelnen Aktivitäten stattfinden:

- Die Aktivität `Act1` wird auf einen Bedienaufruf der Default-Dauer an die Default-CPU `CPU1` umgesetzt. Der Umfang der Bedienzeitanforderung beträgt entsprechend $1 \cdot 10 = 10$ Einheiten. Die entsprechende Definition des Used Service ist
`CPU1_1 (AMOUNT: REAL DEFAULT 10.0).`

- Die Aktivität `A2` stellt einen Aufruf des existierenden Aktivitätsdiagramms *A2* dar. Es ergibt sich der Used Service `A2_1`.

- Die Aktivität `Act3` stellt einen Aufruf der CPU `CPU2` mit Default-Dauer dar, entsprechend hat diese Anfrage einen Umfang von $1 \cdot 20 = 20$ Einheiten. Die Definition lautet folglich `CPU2_1 (AMOUNT: REAL DEFAULT 20.0).`

- Das Gleiche gilt für die Aktivität `Act4`, die auf die CPU `CPU2` abgebildet wird, weil die Klasse *Class1* im Deploymentdiagramm 4.8 dieser CPU zugeordnet ist. Der Umfang der Anfrage beträgt entsprechend $1 \cdot 20 = 20$ Einheiten. Die Definition lautet `CPU2_2 (AMOUNT: REAL DEFAULT 20.0).`

- Die Aktivität `Act5` stellt einen weiteren Aufruf des Diagramms *A2* dar. Sie wird durch den Used Service `A2_2` dargestellt.

- `Act6` wird auf der Default-CPU `CPU1` ausgeführt, der Umfang der Anforderung ist $10 \cdot 10 = 100$ Einheiten. Der entsprechende Used Service lautet
`CPU1_2 (AMOUNT: REAL DEFAULT 100.0).`

- Die darauf folgende Aktivität `A2` wird mittels des `LET responsetime`-Statements auf `CPU1` abgebildet, der Umfang der Anforderung beträgt 75 Einheiten. Entsprechend ergibt sich der Used Service `CPU1_3 (AMOUNT: REAL DEFAULT 75.0).`

- Die Aktivität `Act8` wird auf eine Anforderung im Umfang von $5 \cdot 20 = 100$ Einheiten an `CPU2` abgebildet. Die Used-Service-Definition lautet dann
`CPU2_3 (AMOUNT: REAL DEFAULT 100.0).`

- Schließlich wird die Aktivität `Act9` auf eine weitere Anforderung an `CPU2` im Umfang von $2.5 \cdot 20 = 50$ Einheiten abgebildet. Die Anforderung wird dann auf die Used-Service-Definition `CPU2_4 (AMOUNT: REAL DEFAULT 50.0)` abgebildet.

Alle diese einzelnen Aufrufe werden im Rahmen einer `OPEN_CHAIN` in der angegebenen Reihenfolge aufeinanderfolgend ausgeführt. Die selbstdefinierte Komponente `AA2_1`, die das Aktivitätsdiagramm *A2* darstellt, und die Komponenten, die die CPUs *CPU1* und *CPU2* darstellen, werden ENCLOSEd eingebunden, da sie im Rahmen der Hierarchieregel von HIPE bereits auf

der MODEL-Ebene als COMPONENTs definiert wurden.

Realisiertes Verhalten
Es wird im folgenden das von HIPE produzierte Listing untersucht, das die Umsetzung des Aktivitätsdiagramms enthält.

```
-->024   TYPE AA1 COMPONENT;
-->025     PROVIDE
-->026       SERVICE
-->027         request;
-->028     END PROVIDE;
-->029     TYPE request SERVICE;
-->030       USE
-->031         SERVICE
-->032           CPU1_1 (AMOUNT : REAL DEFAULT 10.0);  % Act1
-->033           AA2_1;                                % A2
-->034           CPU2_1 (AMOUNT : REAL DEFAULT 20.0);  % Act3
-->035           CPU2_2 (AMOUNT : REAL DEFAULT 20.0);  % Act4
-->036           AA2_2;                                % Act5
-->037           CPU1_2 (AMOUNT : REAL DEFAULT 100.0); % Act6
-->038           CPU1_3 (AMOUNT : REAL DEFAULT 75.0);  % A2, mit responsetime überlagert
-->039           CPU2_3 (AMOUNT : REAL DEFAULT 100.0); % Act8
-->040           CPU2_4 (AMOUNT : REAL DEFAULT 50.0);  % Act9
-->041       END USE;
-->042       BEGIN
-->043         OPEN_CHAIN
-->044         QNODE CPU1_1
-->045         PROB 1.0: AA2_1;
-->046         QNODE AA2_1
-->047         PROB 1.0: CPU2_1;
-->048         QNODE CPU2_1
-->049         PROB 1.0: CPU2_2;
-->050         QNODE CPU2_2
-->051         PROB 1.0: AA2_2;
-->052         QNODE AA2_2
-->053         PROB 1.0: CPU1_2;
-->054         QNODE CPU1_2
-->055         PROB 1.0: CPU1_3;
-->056         QNODE CPU1_3
-->057         PROB 1.0: CPU2_3;
-->058         QNODE CPU2_3
-->059         PROB 1.0: CPU2_4;
-->060         QNODE CPU2_4
-->061         END OPEN_CHAIN;
-->062     END TYPE request;
-->063     ENCLOSE CPU1 : server;
-->064     ENCLOSE CPU2 : server;
-->065     ENCLOSE AA2_1 : AA2;
-->066     REFER request TO CPU1,CPU2,AA2_1 EQUATING
-->067     request.CPU1_1 WITH CPU1.request;
-->068     request.AA2_1 WITH AA2_1.request;
-->069     request.CPU2_1 WITH CPU2.request;
-->070     request.CPU2_2 WITH CPU2.request;
-->071     request.AA2_2 WITH AA2_1.request;
-->072     request.CPU1_2 WITH CPU1.request;
-->073     request.CPU1_3 WITH CPU1.request;
-->074     request.CPU2_3 WITH CPU2.request;
-->075     request.CPU2_4 WITH CPU2.request;
-->076     END REFER;
-->077   END TYPE AA1;
```

Zunächst wird die `OPEN_CHAIN` betrachtet, die die Abfolge der einzelnen Aktivitäten darstellt. Aus deren Betrachtung erschließt sich, dass die einzelnen Aktivitäten in einer einzelnen Folge dargestellt werden. Die Aktivität `Act1` wird auf eine Benutzung von `CPU1` im Umfang von 10 Einheiten abgebildet. `A2` stellt einen expliziten Aufruf des Aktivitätsdiagramms *A2* dar. `Act3` und `Act4` werden auf `CPU2` im Umfang von 20 Einheiten ausgeführt. `Act5` stellt einen weiteren Aufruf des Aktivitätsdiagramms *A2* dar. `Act6` stellt einen Aufruf von `CPU1` im Umfang von 100 Einheiten dar, die mit der Annotation `let responsetime` versehene Aktivität `A2` einen Aufruf von `CPU1` im Umfang von 75 Einheiten. Schließlich werden `Act8` und `Act9` auf Ausführungen auf `CPU2` im Umfang von 100 bzw. 50 Einheiten abgebildet.

Die Komponente `AA2_1`, die das Aktivitätsdiagramm darstellt, und die Komponenten `CPU1` und `CPU2`, die die gleichnamigen CPUs aus dem Verteilungsdiagramm darstellen, werden als

ENCLOSEd Komponenten definiert.

Das von HIPE in diesem Falle realisierte Verhalten entspricht folglich dem erwarteten.

4.2.2 Test mit Branches

Das in Abbildung 4.10 auf S. 144 dargestellte Aktivitätsdiagramm *A2* enthält drei Verzweigungsknoten, von denen jeweils einer nach einer der drei gemäß des HIPE-Paradigmas zulässigen Annotationsmethoden beschriftet wurde.

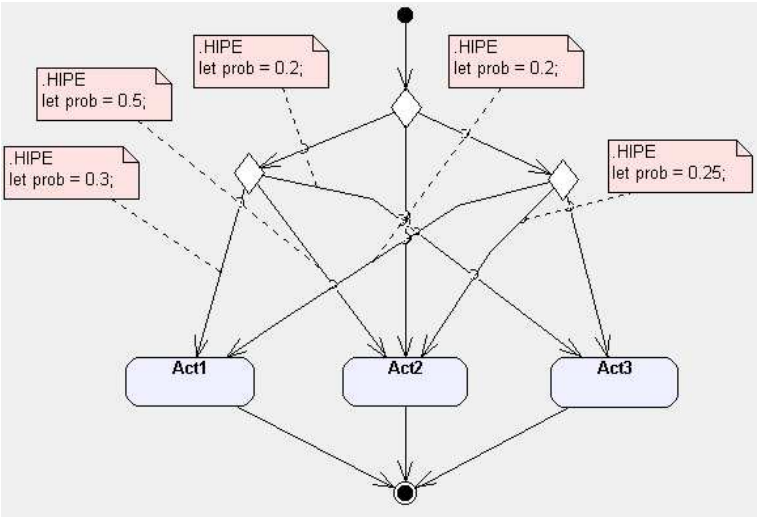

Abbildung 4.10: Diagramm *A2* zum Test mit Branches

Um das Ergebnis dieses Tests nachzuvollziehen, ist es wichtig zu wissen, dass die Umsetzung von Aktivitätsdiagrammen, in denen einer oder mehrere Knoten mehrere Nachfolger haben, von der Reihenfolge abhängig ist, in der während der Modellierung in MetaMill von diesen Knoten aus die Kanten gezogen wurden. Dies bezieht sich nicht auf die Korrektheit – das entstehende Diagramm wird ebenfalls korrekt umgesetzt –, wohl aber auf die Vergabe der Namen. Das Aktivitätsdiagramm *A2* wurde so konstruiert, dass zunächst der oberste Entscheidungsknoten mit den beiden anderen Entscheidungsknoten und danach zunächst der oberste Entscheidungsknoten mit der Aktivität *Act2*, dann der linke Entscheidungsknoten zuerst mit *Act1*, danach mit *Act2* und schließlich mit *Act3* verbunden wurde. Für die Namensvergabe ist die Verbindungsreihenfolge vom rechten Entscheidungsknoten aus nicht mehr entscheidend, da sich dadurch nur noch die Reihenfolge verändert, in der die einzelnen möglichen Nachfolger dieses Knotens im Rahmen seiner `QNODE`-Spezifikation erscheinen.

Erwartetes Verhalten
Die einzelnen Entscheidungen im Aktivitätsdiagramm werden selbst als Aktivitäten dargestellt, die auf die Default-CPU (das heisst `CPU1`) zugreifen, wobei der Umfang auf 0.000001 Operationen festgesetzt wird. Die Namensvergabe für die einzelnen Aktivitäten erfolgt im

Einklang mit den Namensregeln für die Umsetzung von Aktivitäten (siehe Tabelle 1.6 und die Darstellung auf Seite 141).

- Dem obersten Verzweigungsknoten werden gemäß der Methode der Gleichverteilung mit einer Wahrscheinlichkeit von jeweils einem Drittel einer der beiden anderen Entscheidungsknoten und die Aktivität *Act2* zugewiesen. Dieser Knoten wird auf die Default-CPU `CPU1` abgebildet. Da er die erste im Diagramm ausgeführte Aktivität darstellt, erhält der Used Service, der seine Ausführung darstellt, den Namen `CPU1_1`.
- Dem linken Verzweigungsknoten wird gemäß der Methode der vollständigen Annotation mit einer Wahrscheinlichkeit von 30% die Aktivität *Act1*, mit 50% *Act2* und mit 20% *Act3* zugewiesen. Der Verzweigungsknoten wird ebenfalls auf die Default-CPU `CPU1` abgebildet. Der Name des entsprechenden Used Service ist `CPU1_2`.
- Dem rechten Verzweigungsknoten wird gemäß der Methode der Restwahrscheinlichkeit mit einer Wahrscheinlichkeit von 20% *Act1*, mit 25% *Act2* und mit 55% *Act3* zugewiesen. Der Name des Used Service ist `CPU1_3`.
- Die Aktivität *Act2* wird auf eine Ausführung der Default-Dauer von 1 Sekunde auf der Default-CPU `CPU1` umgesetzt, der Umfang der Bedienanforderung ist entsprechend $1 \cdot 10 = 10$ Operationen. Der Name des entsprechenden Used Service ist `CPU1_4`.
- Die Aktivität *Act1* erhält ebenfalls die gleichen Default-Eigenschaften, der Used Service heißt `CPU1_5`.
- Die Aktivität *Act3* wird durch den Aufruf des Used Service `CPU1_6` dargestellt.

Realisiertes Verhalten
Wir betrachten den von HIPE für dieses Aktivitätsdiagramm erzeugten Code. Wichtig ist die `OPEN_CHAIN`, die die Struktur des Aktivitätsdiagramms *A2* implementiert.

```
-->078   TYPE AA2 COMPONENT;
-->079     PROVIDE
-->080       SERVICE
-->081         request;
-->082     END PROVIDE;
-->083     TYPE request SERVICE;
-->084       USE
-->085         SERVICE
-->086           CPU1_1 (AMOUNT : REAL DEFAULT 0.000001);
-->087           CPU1_2 (AMOUNT : REAL DEFAULT 0.000001);
-->088           CPU1_3 (AMOUNT : REAL DEFAULT 0.000001);
-->089           CPU1_4 (AMOUNT : REAL DEFAULT 10.0);
-->090           CPU1_5 (AMOUNT : REAL DEFAULT 10.0);
-->091           CPU1_6 (AMOUNT : REAL DEFAULT 10.0);
-->092       END USE;
-->093       BEGIN
-->094         OPEN_CHAIN
-->095         QNODE CPU1_1 % oberster Entscheidungsknoten
-->096         PROB 0.3333333333333333: CPU1_2;
-->097         PROB 0.3333333333333333: CPU1_3;
-->098         PROB 0.3333333333333333: CPU1_4;
-->099         QNODE CPU1_2 % linker Entscheidungsknoten
-->100         PROB 0.3: CPU1_5;
-->101         PROB 0.5: CPU1_4;
-->102         PROB 0.2: CPU1_6;
-->103         QNODE CPU1_3 % rechter Entscheidungsknoten
-->104         PROB 0.55: CPU1_6;
-->105         PROB 0.25: CPU1_4;
-->106         PROB 0.2: CPU1_5;
-->107         QNODE CPU1_4 % Act2
-->108         QNODE CPU1_5 % Act1
-->109         QNODE CPU1_6 % Act3
-->110         END OPEN_CHAIN;
-->111     END TYPE request;
-->112     ENCLOSE CPU1 : server;
```

```
-->113     REFER request TO CPU1 EQUATING
-->114     request.CPU1_1 WITH CPU1.request;
-->115     request.CPU1_2 WITH CPU1.request;
-->116     request.CPU1_3 WITH CPU1.request;
-->117     request.CPU1_4 WITH CPU1.request;
-->118     request.CPU1_5 WITH CPU1.request;
-->119     request.CPU1_6 WITH CPU1.request;
-->120     END REFER;
-->121   END TYPE AA2;
```

Der QNODE `CPU1_1` stellt den obersten Entscheidungsknoten dar. Gemäß der Gleichverteilung werden die drei Knoten `CPU1_2`, `CPU1_3` und `CPU1_4` angesteuert. Der QNODE `CPU1_2` stellt den linken Entscheidungsknoten dar. Alle davon abgehenden Kanten sind annotiert, entsprechend wird die Annotation übertragen, so daß auf den Knoten `CPU1_5` mit Wahrscheinlichkeit 30%, auf den Knoten `CPU1_4` mit 50% und auf `CPU1_6` mit 20% verwiesen wird. Beim rechten Entscheidungsknoten sind alle bis auf eine Kante annotiert, die Annotation der Kanten zu `CPU1_4` (25%) und `CPU1_5` (20%) werden direkt aus dem Diagramm übernommen, die Übergangswahrscheinlichkeit zu `CPU1_6` (55%) ist die zu 100% dann noch fehlende Wahrscheinlichkeit.

Daraus folgt, dass das von HIPE realisierte Verhalten dem erwarteten entspricht.

Nachtrag des Korrektors: *Die von HIPE durchgeführte Umsetzung von Entscheidungsknoten hilft, die Struktur von Diagrammen im erzeugten Code nachzuvollziehen. Andererseits haben diese Knoten aber keine wesentliche Relevanz für die Performance, so dass eine Abänderung der Transformation überdacht werden kann, die diese zwar weiterhin berücksichtigt, um die Eintritts- und Übergangswahrscheinlichkeiten zwischen den einzelnen Aktivitäten korrekt zu ermitteln, aber sie nicht separat als* `QNODE`*s codiert.*

4.2.3 Test mit Objektfluss

Das Objektflussmodell besteht aus einem Aktivitätsdiagramm und einem trivialen Klassendiagramm. Das in Abbildung 4.11 auf S. 147 abgebildete Aktivitätsdiagramm *A3* enthält eine Aktivität, die ein Objekt der Klasse *Class1* instanziiert und eine Aktivität, die dieses Objekt als Argument annimmt. Zum Test mit dem Objektfluss wird ein Klassendiagramm (*K1*) verwendet, worin einem Objekt der dargestellten Klasse *Class1* eine gewisse Komplexität zugewiesen wird. Diese Komplexität errechnet sich mit Hilfe der Ergebnisse der weichen Analyse und spiegelt sich in der Definition der Last beim Aufruf des entsprechenden Services wider. Das Klassendiagramm findet sich in Abbildung 4.12 auf S. 147.

Beide bei der Durchführung des Aktivitätsdiagramms ablaufende Aktivitäten werden auf der Default-CPU `CPU1` ausgeführt, das Objekt selbst aber – wie in Abbildung 4.8 auf S. 138 zu sehen – auf der `CPU2` gespeichert.

Erwartetes Verhalten
Die Komplexität eines Objektes der Klasse *Class1* wird anhand des Klassendiagramms in Abbildung 4.12 auf S. 147 ermittelt. Sie ergibt sich gemäß der verwendeten Komplexitätsmetrik von Carbone und Santucci (siehe Abschnitt 1.2.5) aus einer gewichteten Summe ihrer Attribute und Methoden.

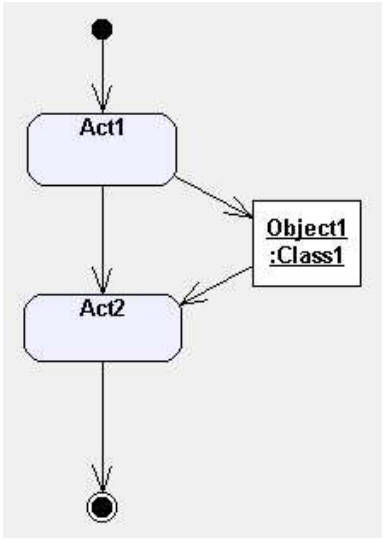

Abbildung 4.11: Aktivitätsdiagramm *A3* zum Test mit Objektfluss

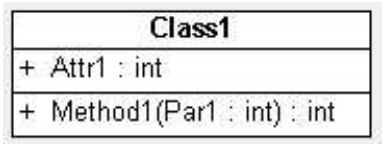

Abbildung 4.12: Klassendiagramm *K1* zum Test mit Objektfluss

Die *State Points* der Klasse *Class1* ergeben sich als gewichtete Summe ihrer Attribute. *Attr1* ist ein Attribut vom Typ *int*, das heisst ein sogenanntes *lightAttribute*. Da *Class1* keine weiteren Attribute hat, ergibt sich $numLA(Class1) = 1, numHA(Class1) = 0, numIA(Class1) = 0$, und demnach

$$SP(Class1) = 1 \cdot 1 + 3 \cdot 0 + 5 \cdot 0 = 1$$

Die *Behavioral Points* der Klasse *Class1* ergeben sich als Summe der Komplexitäten ihrer Methoden. Da *Class1* nur eine Methode besitzt, ist dieser Wert mit der Komplexität dieser Methode identisch. *Method1* ist eine Methode, die ein leichtes Attribut *Par1* vom Typ *int* als Argument und ein leichtes Attribut vom Typ *int* als Rückgabewert besitzt. Da *Method1* keine weiteren Attribute hat, ergibt sich $numLA(Method1) = 2$, $numHA(Method1) = 0$, $numIA(Method1) = 0$. Diese Methode ist offensichtlich eine *triviale Methode*, da es keine *heavy* oder *imported attributes* gibt. Entsprechend ergeben sich die Komplexität der Methode und die *Behavioral Points* der Klasse *Class1* zu

$$BP(Class1) = CM(Method1) = 1 \cdot 2 + 2 \cdot 0 + 3 \cdot 0 = 2$$

Als Wert der *Complexity Points* der Klasse ergibt sich dann

$$CP(Class1) = 2 \cdot SP(Class1) + 3 \cdot BP(Class1) = 2 \cdot 1 + 3 \cdot 2 = 8$$

Jedem Objekt dieser Klasse wird entsprechend eine Komplexität von 8 Speichereinheiten zugeordnet.

Der Ablauf im Kontext des Objektflusses entspricht der Darstellung im Kapitel 1.2.6.2 ab der Seite 51.

- Zunächst wird die Aktivität *Act1* auf *CPU1* ausgeführt.
- Durch die von der Aktivität *Act1* abgehende Objektflusskante, die diese mit einem Objekt der Klasse *Class1* verbindet, wird dargestellt, dass ein Objekt dieser Klasse instanziiert werden soll. Dazu muss zunächst im Rahmen einer Garbage Collection und anderen Speicherverwaltungsoperationen Speicherplatz auf der CPU *CPU1* bereitgestellt werden. Der Umfang der Speicherverwaltung ergibt sich gemäß der von HIPE verwendeten Jamaica-Semantik (siehe Seite 52) als Funktion der Komplexität der Klasse *Class1* und der Größe des auf der CPU *CPU1* zur Verfügung stehenden Speichers. Als Komplexität wird der Wert von CP(Class1) eingesetzt, als Speichergröße die im Verteilungsdiagramm in Abbildung 4.8 auf S. 138 angegebene Annotation für `memory` von *CPU1*. Gemäß der Formel von Siebert ergibt sich damit ein Aufwand von

$$P(8,10) = \frac{8}{(1-8/10)\cdot e^{(1-8/10)\cdot e^{-8/10}-1-8/10}} = \frac{8}{0.0361683} = 221.1881421$$

- Danach wird die Instanziierung des Objektes der Klasse *Class1* durchgeführt. Diese hat einen Umfang von 8 Einheiten.
- In der Folge findet der Transport des Objektes über die Leitung zwischen `CPU1` und `CPU2` statt. Dafür sind die im o.g. Verteilungsdiagramm für die Leitung angegebenen Charakteristika `speed = 10`, `minpacketsize = 1` und `maxpacketsize = 2` relevant. Dies definiert, dass kein Aufruf dieser Leitung größer als 2 Einheiten sein darf, aber mindestens 1 Einheit groß sein muss. Ein Objekt der Klasse *Class1*, das eine Komplexität von 8 Einheiten hat, muss also in vier Pakete zerlegt werden, von denen jedes einen Umfang von 2 Einheiten hat.
- Beim empfangenden System `CPU2` wird eine weitere Speicherverwaltung im Umfang von

$$P(8,20) = \frac{8}{(1-8/20)\cdot e^{(1-8/20)\cdot e^{-8/20}-1-8/20}} = \frac{8}{0.2212120} = 36.1643974$$

  Einheiten ausgeführt.
- Die Aktivität *Act2* konsumiert das eben erzeugte Objekt der Klasse *Class1*. Damit diese ausgeführt werden kann, muss entsprechend das Objekt von seinem Speicherort `CPU2` wieder über die selbe Leitung zum Ausführungsort `CPU1` der Aktivität transportiert werden. Das Objekt wird wiederum in vier Paketen á 2 Einheiten übertragen.
- Beim empfangenden System `CPU1` wird eine weitere Speicherverwaltung im Umfang von $P(8,10) = 221.1881421$ Einheiten (siehe oben) durchgeführt.
- Abschließend wird die Aktivität *Act2* ausgeführt.

Die Leitung zwischen den beiden CPUs wird als eine selbstdefinierte Komponente dargestellt, die einen von außen zugänglichen Service *request* und einen nur intern verwendeten Service *errorService* besitzt, von dem im Abstand von einer Zeiteinheit jeweils ein Exemplar erzeugt wird. Beide Services greifen über einen Used Service auf eine flache Komponente vom Typ *prioserver* zu, dessen Scheduling-Disziplin *preemptive repeat* ist. Da diese Leitung nur innerhalb des Aktivitätsdiagramms *A3* verwendet wird, wird sie innerhalb der selbstdefinierten Komponente vom Typ `AA3` als COMPONENT definiert.

Realisiertes Verhalten

Es wird nunmehr die Codierung des Aktivitätsdiagramms betrachtet, die von HIPE (unter Verwendung der Nameskonvention aus Tabelle 1.6) erzeugt wird.

```
-->122   TYPE AA3 COMPONENT;
-->123     PROVIDE
-->124       SERVICE
-->125         request;
-->126     END PROVIDE;
-->127     TYPE request SERVICE;
-->128       USE
-->129         SERVICE
-->130           CPU1_1 (AMOUNT : REAL DEFAULT 10.0);
-->131           CPU1_2 (AMOUNT : REAL DEFAULT 5.0);
-->132           CPU1_3 (AMOUNT : REAL DEFAULT 33.09293326744951);
-->133           CPU1_CPU2_1 (AMOUNT : REAL DEFAULT 2.0);
-->134           CPU1_CPU2_2 (AMOUNT : REAL DEFAULT 2.0);
-->135           CPU1_CPU2_3 (AMOUNT : REAL DEFAULT 1.0);
-->136           CPU2_1 (AMOUNT : REAL DEFAULT 12.9749347168455855);
-->137           CPU1_CPU2_4 (AMOUNT : REAL DEFAULT 2.0);
-->138           CPU1_CPU2_5 (AMOUNT : REAL DEFAULT 2.0);
-->139           CPU1_CPU2_6 (AMOUNT : REAL DEFAULT 1.0);
-->140           CPU1_4 (AMOUNT : REAL DEFAULT 33.09293326744951);
-->141           CPU1_5 (AMOUNT : REAL DEFAULT 10.0);
-->142       END USE;
-->143       BEGIN
-->144         OPEN_CHAIN
-->145         QNODE CPU1_1
-->146         PROB 1.0: CPU1_3;
-->147         QNODE CPU1_3
-->148         PROB 1.0: CPU1_2;
-->149         QNODE CPU1_2
-->150         PROB 1.0: CPU1_CPU2_1;
-->151         QNODE CPU1_CPU2_1
-->152         PROB 1.0: CPU1_CPU2_2;
-->153         QNODE CPU1_CPU2_2
-->154         PROB 1.0: CPU1_CPU2_3;
-->155         QNODE CPU1_CPU2_3
-->156         PROB 1.0: CPU2_1;
-->157         QNODE CPU2_1
-->158         PROB 1.0: CPU1_CPU2_4;
-->159         QNODE CPU1_CPU2_4
-->160         PROB 1.0: CPU1_CPU2_5;
-->161         QNODE CPU1_CPU2_5
-->162         PROB 1.0: CPU1_CPU2_6;
-->163         QNODE CPU1_CPU2_6
-->164         PROB 1.0: CPU1_4;
-->165         QNODE CPU1_4
-->166         PROB 1.0: CPU1_5;
-->167         QNODE CPU1_5
-->168         END OPEN_CHAIN;
-->169     END TYPE request;
-->170     TYPE CPU1_CPU2 COMPONENT;
-->171     PROVIDE
-->172       SERVICE
-->173         request(AMOUNT :REAL);
-->174     END PROVIDE;
-->175     TYPE request SERVICE(AMOUNT :REAL);
-->176      USE
-->177       SERVICE
-->178        req (AMOUNT : REAL; PRIO: INTEGER DEFAULT 1);
-->179      END USE;
-->180     BEGIN
-->181      req(AMOUNT);
-->182     END TYPE request;
-->183     TYPE errorService SERVICE;
-->184      USE
-->185       SERVICE
-->186        error (AMOUNT : REAL; PRIO : INTEGER DEFAULT 0);
-->187      END USE;
-->188     BEGIN
-->189      error(1);
```

```
-->190     END TYPE errorService;
-->191     COMPONENT CPU1_CPU2_server : prioserver (LET schedule := prioprep,LET dispatch := shared(LET
-->192           speed
-->193           :=
-->194           10));
-->195     REFER request, errorService TO CPU1_CPU2_server EQUATING
-->196      request.req WITH CPU1_CPU2_server.request;
-->197      errorService.error WITH CPU1_CPU2_server.request;
-->198     END REFER;
-->199     BEGIN
-->200      CREATE 1 PROCESS errorService EVERY 1;
-->201     END TYPE CPU1_CPU2;
-->202     ENCLOSE CPU1 : server;
-->203     ENCLOSE CPU2 : server;
-->204     COMPONENT CPU1_CPU2_1 : CPU1_CPU2;
-->205     REFER request TO CPU1,CPU2,CPU1_CPU2_1 EQUATING
-->206     request.CPU1_1 WITH CPU1.request;
-->207     request.CPU1_2 WITH CPU1.request;
-->208     request.CPU1_3 WITH CPU1.request;
-->209     request.CPU1_CPU2_1 WITH CPU1_CPU2_1.request;
-->210     request.CPU1_CPU2_2 WITH CPU1_CPU2_1.request;
-->211     request.CPU1_CPU2_3 WITH CPU1_CPU2_1.request;
-->212     request.CPU2_1 WITH CPU2.request;
-->213     request.CPU1_CPU2_4 WITH CPU1_CPU2_1.request;
-->214     request.CPU1_CPU2_5 WITH CPU1_CPU2_1.request;
-->215     request.CPU1_CPU2_6 WITH CPU1_CPU2_1.request;
-->216     request.CPU1_4 WITH CPU1.request;
-->217     request.CPU1_5 WITH CPU1.request;
-->218     END REFER;
-->219   END TYPE AA3;
```

Die Used Services `CPU1_1` und `CPU1_5` stellen die Aufrufe der beiden Aktivitäten *Act1* und *Act2* dar. `CPU1_2` stellt die Instanziierung eines Objektes der Klasse *Class1* mit der Komplexität von 8 Einheiten und `CPU1_3` die dazu benötigte Speicherverwaltung im Umfang von $P(8, 10) = 221.1881421$ Einheiten dar. Die Used Services `CPU1_CPU2_1` ... `CPU1_CPU2_3` stellen den Transport der drei Pakete über die Leitung von *CPU1* nach *CPU2* dar, `CPU2_1` den Aufwand von $P(8, 20) = 36.1643974$ Einheiten für die Speicherverwaltung, damit das transportierte Objekt auf *CPU2* abgespeichert werden kann. Den Transport des Objektes von *CPU2* nach *CPU1* stellen die Used Services `CPU1_CPU2_4` ... `CPU1_CPU2_6` dar. `CPU1_4` beschreibt schließlich die Speicherverwaltung, die *CPU1* ausführen muss, damit das Objekt dort gespeichert werden kann, wiederum im Umfang von $P(8, 10) = 221.1881421$ Einheiten.

Die Leitung zwischen den beiden CPUs ist eine Komponente vom Typ `CPU1_CPU2` mit dem Provided Service *request* und einem nur intern zugänglichen Service *errorService*, die beide auf eine atomare Komponente vom Typ *prioserver* zugreifen, deren Scheduling-Disziplin *preemptive repeat* ist. Im Abstand von einer Zeiteinheit wird ein Prozess vom Typ *errorService* erzeugt.

Entsprechend folgt, dass das von HIPE realisierte Verhalten *nicht* dem erwarteten Verhalten entspricht.

4.2.4 Test mit Concurrent

Das in der Abbildung 4.13 auf S. 151 dargestellte Aktivitätsdiagramm *A4* enthält einen nebenläufigen Bereich, in dem zwei Prozesse parallel auf den beiden im Verteilungsdiagramm in Abbildung 4.8 auf S. 138 dargestellten CPUs ablaufen und anschließend miteinander synchronisieren. Man beachte, dass es einen deutlichen Unterschied zwischen einem Concurrent-Modell („Fork/Join-Programm“, vgl. [Adve und Vernon (2004)]) wie dem dargestellten und dem in den Abschnitten 4.1.3 und 4.2.6 in zwei verschiedenen Realisierungen gesehenen Parallel-System gibt: in einem Concurrent-Bereich werden entsprechend der Ankunftsrate im Fork-Knoten auf *jedem* Pfad Anfragen erstellt. In einem Parallel-System wird dem gegenüber die Anfrage nur auf *einem* der möglichen Wege weitergeleitet.

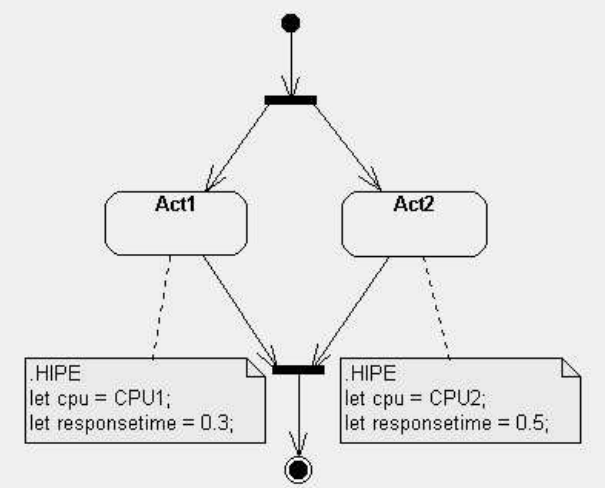

Abbildung 4.13: Aktivitätsdiagramm *A4* zum Test mit Concurrent

Erwartetes Verhalten
Der Concurrent-Block, der im Diagramm enthalten ist, wird in eine selbstdefinierte Komponente umgewandelt, die von der Ebene der selbstdefinierten Komponente aus aufgerufen wird, die das Diagramm darstellt. Der Name der Komponente, die den Concurrent-Block selbst darstellt, ergibt sich dabei aus den internen Namen des fork- und des join-Knotens, die bei der Erstellung der Repräsentation innerhalb von HIPE vergeben werden, als
`C<interner Name des fork>_<interner Name des join>`
wobei das Präfix "C" für eine von HIPE selbsttätig erzeugte Komponente steht.

Der Name der Komponente, die einen Pfad darstellt, ergibt sich aus dem internen Namen des ersten Knotens, der vom fork-Knoten aus aus diesem Pfad erreicht wird, und dem internen Namen des join-Knotens, als
`C<interner Name des ersten Pfadknotens>_<interner Name des join>S`
wobei das Suffix "S" zur Unterscheidung von der Komponente dient, die den ganzen Block darstellt. Die internen Namen der Komponenten können vorab nicht bestimmt werden, da sie sich aus den Referenzen der Objekte ergeben, die im Rahmen von HIPE erstellt werden, um das entsprechende Aktivitätsdiagramm zu repräsentieren, und damit letztlich "zufällig" sind. Der Name jedes Used Service, der eine Anfrage an eine dieser Komponenten darstellt, besteht analog zu der bisher verwendeten Namenskonvention für Used Services aus dem Namen der angesprochenen Komponente und einem Nummerierungssuffix.

Um im Zuge der Diskussion des erwarteten Verhaltens die von HIPE erstellten Komponenten voneinander zu unterscheiden, ordnen wir der Komponente, die den Concurrent-Block selbst darstellt, den Namen `CF_J_1` und den Komponenten, die die beiden Pfade darstellen, die Namen `CAct1_JS_1` und `CAct2_JS_1` zu. Innerhalb der Komponente vom Typ `AA4`, die das Aktivitätsdiagramm *A4* darstellt, wird zunächst ein Used Service aufgerufen, der mit dem Service *request* der Komponente `CF_J` verbunden ist. Dieser Service enthält einen `CONCURRENT`-Block, der die Aufrufe der beiden selbstdefinierten Komponenten, die jeweils einen der beiden nebenläufigen Pfade darstellen, nebenläufig anordnet. Die entsprechenden Used Services tragen die

Namen `CAct1_JS_1` und `CAct2_JS_1` und sind jeweils mit den provided services *request* der gleichnamigen selbstdefinierten Komponenten verbunden. Die Komponente `CAct1_JS_1` stellt den ersten Pfad dar. Dieser enthält nur eine Aktivität *Act1*, die auf einen Aufruf der CPU `CPU1` im Umfang von $0.3 \cdot 10 = 3$ Einheiten abgebildet wird. Die Komponente `CAct2_JS_1` stellt den zweiten Pfad dar, der eine Aktivität *Act2* enthält, die auf einen Aufruf der CPU `CPU2` im Umfang von $0.5 \cdot 20 = 10$ Einheiten umgesetzt wird.

Im Use-Case-Diagramm des Modells wird an dem Anwendungsfallknoten, der auf das betrachtete Aktivitätsdiagramm verweist, die mittlere Antwortzeit eines Aufrufes gemessen. Die Parameter des Modells wurden so einfach gewählt, dass die Werte einfach überprüft werden können. Zum Beispiel wird keine der beiden CPUs durch eine andere Anfrage belastet, während das aktuelle Diagramm bearbeitet wird, so dass für ankommende Anfragen keine Wartezeit besteht. Die Antwortzeit eines Aufrufs des Aktivitätsdiagramms ist abhängig von den Antwortzeiten der beiden CPUs auf die Bedienanfragen, die im Rahmen der Pfade an diese gestellt werden. Da die Aktivitäten auf den beiden Pfaden auf verschiedenen CPUs ausgeführt werden, ergibt sie sich genauer als das Maximum der Antwortzeiten der beiden Aktivitäten. Zu bemerken ist, dass im allgemeinen die Antwortzeit von Concurrent-Modellen nicht auf derart einfache Weise oder in vielen Fällen überhaupt exakt berechnet werden kann.

Die vorliegende D/D/1-Queue wird für die Berechnung gemäß der Regeln für G/G/1-Queues (vgl. [Jain (1991)]) analysiert, die auch für den Spezialfall jeweils deterministischer Ankunfts- und Bedienzeiten anwendbar sind (vgl. [Willig (1999)]). λ_k sei die Anzahl an Anfragen, die pro Sekunde an der CPU k ankommen, μ_k die Anzahl der während einer Zeiteinheit durch die CPU k abgefertigten Anfragen, und T_k sei die Bedienzeitanforderung der in diesem Modell einzigen Anfrage, die der CPU k gestellt wird. Die mittlere Anzahl an Ankünften an beiden CPUs ist gegeben durch die Angabe der Arbeitslast, nämlich als

$$\lambda = \lambda_1 = \lambda_2 = 1 \text{ Anfrage/sec}$$

Die mittlere Anzahl an Anfragen des jeweils angegebenen Umfangs, die von der CPU k pro Zeiteinheit verarbeitet werden können, ergibt sich als

$$\mu_k = \frac{1}{T_k}$$

Für die beiden CPUs gilt entsprechend $\mu_1 = 1/(3/10) = 3.\overline{3}$ und $\mu_2 = 1/(10/20) = 2$.

Die mittlere Population der CPU k ergibt sich aus der Anzahl der Anfragen, die sich im Mittel in ihrer Warteschlange befinden, und der Anzahl der Anfragen, die sich im Mittel in Bedienung befinden. Die mittlere Anzahl in Bearbeitung befindlicher Anfragen auf der CPU k ergibt sich als

$$E[n]_k = \frac{\lambda_k}{\mu_k}$$

Für die beiden CPUs ergibt sich $E[n]_1 = 1/3.\overline{3} = 0.3$ und $E[n]_2 = 1/2 = 0.5$. Der Wert von $E[n]_k$ ist für die betrachteten Queues jeweils gleich der Auslastung ρ_k. Da also für beide CPUs $\rho_k < 1$ ist, ergeben sich keine Wartezeiten (vgl. auch [Vastola (1996)]), und die Anzahl der wartenden Anfragen ist jeweils gleich 0.

Die erwartete Antwortzeit ergibt sich also exklusiv aus der Bearbeitungszeit der Anfragen auf den CPUs zu

$$E[t]_k = \frac{E[n]_k}{\lambda_k} = \frac{E[n]_k}{\lambda}$$

Für die CPU 1 gilt also $E[t]_1 = 0.3/1 = 0.3$ sec und für die CPU 2 ist $E[t]_2 = 0.5/1 = 0.5$ sec.

Da der Durchlauf erst beendet wird, wenn beide Pfade durchlaufen sind, ergibt sich die Antwortzeit für den gesamten Bereich als Maximum der beiden Werte, der erwartete Wert für TURNAROUNDTIME ist also 0.5 Sekunden.

Realisiertes Verhalten
Das Listing bildet die Codierung des Aktivitätsdiagramms in HISLANG ab.

```
-->220   TYPE AA4 COMPONENT;
-->221     PROVIDE
-->222       SERVICE
-->223         request;
-->224     END PROVIDE;
-->225     TYPE request SERVICE;
-->226       USE
-->227         SERVICE
-->228           Cb66cc_f84386C_1;
-->229       END USE;
-->230       BEGIN
-->231         OPEN_CHAIN
-->232         QNODE Cb66cc_f84386C_1
-->233         END OPEN_CHAIN;
-->234     END TYPE request;
-->235     TYPE Cb66cc_f84386C COMPONENT;
-->236       PROVIDE
-->237         SERVICE
-->238           request;
-->239       END PROVIDE;
-->240       TYPE request SERVICE;
-->241         USE
-->242           SERVICE
-->243             C1e4457d_f84386S_1;
-->244             C18e2b22_f84386S_1;
-->245         END USE;
-->246         BEGIN
-->247           CONCURRENT
-->248           C1e4457d_f84386S_1;
-->249           TO
-->250           C18e2b22_f84386S_1;
-->251           END CONCURRENT;
-->252       END TYPE request;
-->253       TYPE C1e4457d_f84386S COMPONENT;
-->254         PROVIDE
-->255           SERVICE
-->256             request;
-->257         END PROVIDE;
-->258         TYPE request SERVICE;
-->259           USE
-->260             SERVICE
-->261               CPU1_1 (AMOUNT : REAL DEFAULT 3.0);
-->262           END USE;
-->263           BEGIN
-->264             OPEN_CHAIN
-->265             QNODE CPU1_1
-->266             END OPEN_CHAIN;
-->267         END TYPE request;
-->268         ENCLOSE CPU1 : server;
-->269         REFER request TO CPU1 EQUATING
-->270     request.CPU1_1 WITH CPU1.request;
-->271         END REFER;
-->272       END TYPE C1e4457d_f84386S;
-->273       TYPE C18e2b22_f84386S COMPONENT;
-->274         PROVIDE
-->275           SERVICE
-->276             request;
-->277         END PROVIDE;
-->278         TYPE request SERVICE;
-->279           USE
-->280             SERVICE
-->281               CPU2_1 (AMOUNT : REAL DEFAULT 10.0);
-->282           END USE;
-->283           BEGIN
-->284             OPEN_CHAIN
-->285             QNODE CPU2_1
```

```
-->286            END OPEN_CHAIN;
-->287         END TYPE request;
-->288         ENCLOSE CPU2 : server;
-->289         REFER request TO CPU2 EQUATING
-->290      request.CPU2_1 WITH CPU2.request;
-->291         END REFER;
-->292       END TYPE C18e2b22_f84386S;
-->293       COMPONENT C1e4457d_f84386S_1 : C1e4457d_f84386S;
-->294       COMPONENT C18e2b22_f84386S_1 : C18e2b22_f84386S;
-->295       REFER request TO C1e4457d_f84386S_1,C18e2b22_f84386S_1 EQUATING
-->296      request.C1e4457d_f84386S_1 WITH C1e4457d_f84386S_1.request;
-->297      request.C18e2b22_f84386S_1 WITH C18e2b22_f84386S_1.request;
-->298       END REFER;
-->299      END TYPE Cb66cc_f84386C;
-->300      COMPONENT Cb66cc_f84386C_1 : Cb66cc_f84386C;
-->301      REFER request TO Cb66cc_f84386C_1 EQUATING
-->302      request.Cb66cc_f84386C_1 WITH Cb66cc_f84386C_1.request;
-->303      END REFER;
-->304    END TYPE AA4;
```

Der fork-Knoten erhielt bei dieser Übersetzung des Modells den internen Namen `b66cc`, der join-Knoten den Namen `f84386`. Der interne Name von *Act1* ist `1e4457d`, der von *Act2* ist `18e2b22`. Entsprechend ergibt sich der Name der Komponente, die den Concurrent-Block darstellt, zu `Cb66cc_f84386_1`, und die Komponenten, die die einzelnen Pfade darstellen, heißen `C1e4457d_f84386S_1` und `C18e2b22_f84386S_1`.

Der Used Service `Cb66cc_f84386C_1`, der im Service `request` des Komponententyps `AA4` aufgerufen wird, stellt den Aufruf der selbstdefinierten Komponente `Cb66cc_f84386C_1` dar. Im Body des einzigen Service `request` dieser Komponente werden mittels des `CONCURRENT`-Statements die beiden durch die Aufrufe dargestellten Pfade `C1e4457d_f84386S_1` und `C18e2b22_f84386S_1` als nebenläufig gekennzeichnet. Der Typ `C1e4457d_f84386S` implementiert einen einzelnen Aufruf der CPU `CPU1` im Umfang von 3 Einheiten, `C18e2b22_f84386S` hingegen einen Aufruf der CPU `CPU2` im Umfang von 10 Einheiten.

Der Messwert für die mittlere Antwortzeit ergibt sich aus der Messung des Wertes Responsetime am Anwendungsfall. Die von HIT zurückgelieferten Ergebnisse werden in der Tabelle 4.7 dargestellt. Zu bemerken ist, dass die Messwerte für die einzelnen CPUs nicht erhoben wurden, da diese gemäß der Art, wie HIPE atomare Aufrufe umsetzt, für das betrachtete Aktivitätsdiagramm *A4* keine Aussagekraft hätten.

Turnaroundtime	SIMULATIVE
Mean	0.5
Stdev	0.0
Con 95%	± 0.00%

Tabelle 4.7: Berechnungsergebnisse zum Concurrent-Diagramm

Als Wert für die mittlere Antwortzeit (Messung des Wertes Responsetime am Anwendungsfall) ergeben sich 0.5 Sekunden. Das realisierte Verhalten entspricht also dem erwarteten.

4.2.5 Test mit Loop

Das in Abbildung 4.14 auf S. 155 dargestellte Aktivitätsdiagramm *A5* enthält einen Bereich, der im Rahmen einer Schleife mehrfach ausgeführt werden soll. Gleichzeitig wird an dem Anwendungsfall im Use-Case-Diagramm, der auf das Aktivitätsdiagramm verweist, eine Messung

der Antwortzeit durchgeführt.

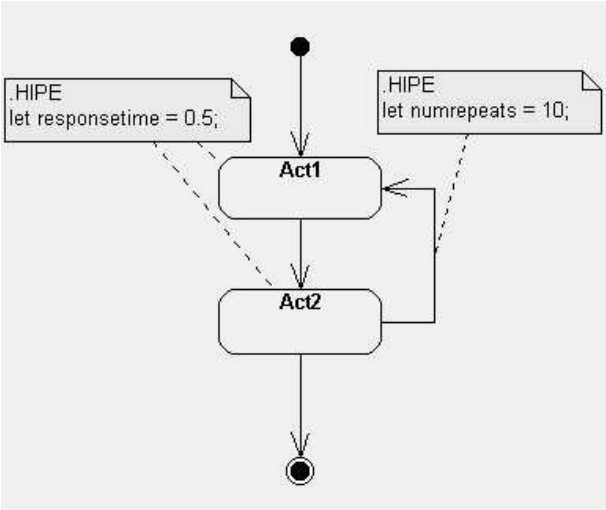

Abbildung 4.14: Aktivitätsdiagramm *A5* zum Test mit Loop

Erwartetes Verhalten
Der Loop-Block, der im Diagramm enthalten ist, wird in eine selbstdefinierte Komponente umgewandelt, die von der Ebene der Komponente, die das Diagramm darstellt, aufgerufen wird. Der Name dieser Komponente wird aus Grundlage der internen Namen der Knoten gebildet, die die erste und die letzte innerhalb des Blocks auszuführende Aktivität darstellen, in diesem Falle also aus den internen Namen der Knoten *Act1* und *Act2*. Er ergibt sich als
`C<interner Name von Act1>_<interner Name von Act2>L_1`
wobei "C" für eine von HIPE selbsttätig erzeugte Komponente und "L" für eine Komponente steht, die eine Schleife darstellt.

Der Service `request` dieser aufgerufenen Komponente (der analog zu den gleichnamigen Services der HIT-Standardkomponenten eingesetzt wird) enthält ein `LOOP`-Statement, das den Aufruf der Komponente kapselt, die den einmaligen Durchlauf der mehrfach zu wiederholenden Aktivitäten darstellt. Der Name dieser Komponente ergibt sich aus den internen Namen der Knoten, die die erste und die letzte innerhalb des Blocks auszuführende Aktivität darstellen, als
`C<interner Name von Act1>_<interner Name von Act2>S_1`
wobei "S" zur Unterscheidung von der Komponente dient, die den Loop-Block selbst darstellt. Die "S"-Komponente enthält die zwei Aktivitäten, die jeweils eine Bedienanforderung an die CPU `CPU1` im Umfang von $0.5 \cdot 10 = 5$ Einheiten darstellen. In der "L"-Komponente findet sich eine FOR-Anweisung, die angibt, dass die Schleife zehnmal wiederholt wird.

Die Antwortzeit des Diagramms ergibt sich als Summe der Antwortzeiten der beiden ausgeführten Aktivitäten, multipliziert mit der Anzahl der Schleifendurchläufe. Die Antwortzeit für einen Schleifendurchlauf wird analog zu der Berechnung im Kapitel 4.2.4 ermittelt.

Die Ankunftsrate der Anfragen ist $\lambda = 1$ Anfrage/sec. Der Gesamtumfang der Anfragen, die während einer Iteration der Schleife an die CPU 1 gestellt werden, ist $0.5 + 0.5 = 1$ sec. Demnach können also pro Sekunde $\mu = 1/1 = 1$ Iterationen ausgeführt werden.

Als Auslastung ergibt sich

$$\rho = \frac{\lambda}{\mu} = \frac{1}{1} = 1$$

Ein D/D/1-System funktioniert auch bei dieser Auslastung stabil (vgl. [Spaniol und Güneş (2001)]). Die mittlere Population der CPU 1 ergibt sich aus den bedienten Anfragen als

$$E[n] = \frac{\lambda}{\mu} = 1$$

Die erwartete Antwortzeit einer Iteration ist dann

$$E[t] = \frac{E[n]}{\lambda} = 1 \text{ sec}$$

Um die Gesamtantwortzeit einer Ausführung des Diagramms zu erhalten, muss dieser Wert nun noch mit der Anzahl der Iterationen multipliziert werden, die durchgeführt werden sollen. Dann ergibt sich die erwartete TURNAROUNDTIME als $10 \cdot E[t] = 10 \cdot 1 = 10$ sec.

Realisiertes Verhalten

Im folgenden wird die Codierung des Aktivitätsdiagramms in HISLANG betrachtet.

```
-->305   TYPE AA5 COMPONENT;
-->306     PROVIDE
-->307       SERVICE
-->308         request;
-->309     END PROVIDE;
-->310     TYPE request SERVICE;
-->311       USE
-->312         SERVICE
-->313           C18a7efd_16cd7d5L_1;
-->314       END USE;
-->315       BEGIN
-->316         OPEN_CHAIN
-->317         QNODE C18a7efd_16cd7d5L_1
-->318         END OPEN_CHAIN;
-->319     END TYPE request;
-->320     TYPE C18a7efd_16cd7d5L COMPONENT;
-->321       PROVIDE
-->322         SERVICE
-->323           request;
-->324       END PROVIDE;
-->325       TYPE request SERVICE;
-->326         USE
-->327           SERVICE
-->328             C18a7efd_16cd7d5S_1;
-->329         END USE;
-->330       VARIABLE nbltr3nqyp2c:REAL;
-->331         BEGIN
-->332           FOR nbltr3nqyp2c := 1 STEP 1 UNTIL 10 LOOP
-->333           C18a7efd_16cd7d5S_1;
-->334           END LOOP;
-->335       END TYPE request;
-->336       TYPE C18a7efd_16cd7d5S COMPONENT;
-->337         PROVIDE
-->338           SERVICE
-->339             request;
-->340         END PROVIDE;
-->341         TYPE request SERVICE;
-->342           USE
-->343             SERVICE
-->344               CPU1_1 (AMOUNT : REAL DEFAULT 5.0);
-->345               CPU1_2 (AMOUNT : REAL DEFAULT 5.0);
-->346           END USE;
-->347           BEGIN
-->348             OPEN_CHAIN
-->349             QNODE CPU1_1
-->350             PROB 1.0: CPU1_2;
```

```
-->351              QNODE CPU1_2
-->352              END OPEN_CHAIN;
-->353          END TYPE request;
-->354          ENCLOSE CPU1 : server;
-->355          REFER request TO CPU1 EQUATING
-->356      request.CPU1_1 WITH CPU1.request;
-->357      request.CPU1_2 WITH CPU1.request;
-->358          END REFER;
-->359        END TYPE C18a7efd_16cd7d5S;
-->360        COMPONENT C18a7efd_16cd7d5S_1 : C18a7efd_16cd7d5S;
-->361        REFER request TO C18a7efd_16cd7d5S_1 EQUATING
-->362      request.C18a7efd_16cd7d5S_1 WITH C18a7efd_16cd7d5S_1.request;
-->363        END REFER;
-->364      END TYPE C18a7efd_16cd7d5L;
-->365      COMPONENT C18a7efd_16cd7d5L_1 : C18a7efd_16cd7d5L;
-->366      REFER request TO C18a7efd_16cd7d5L_1 EQUATING
-->367      request.C18a7efd_16cd7d5L_1 WITH C18a7efd_16cd7d5L_1.request;
-->368      END REFER;
-->369    END TYPE AA5;
```

In der HIPE-Repräsentation erhält die Aktivität *Act1* den internen Namen `18a7efd` und die Aktivität *Act2* den internen Namen `16cd7d5`. Die Komponente, die den Loop-Block kapselt, trägt entsprechend den Namen `C18a7efd_16cd7d5L_1`, und die Komponente, die einen Durchlauf der Schleife kapselt, den Namen `C18a7efd_16cd7d5S_1`.

Im Body des Services `request` im Komponententyp `AA5` wird der Used Service mit dem Namen `C18a7efd_16cd7d5L_1` aufgerufen, der auf die "L"-Komponente `C18a7efd_16cd7d5L_1` verweist. Der Body von deren Service `request` enthält ein `LOOP`-Statement, das die im Aktivitätsdiagramm angegebene 10-fach zu durchlaufende Schleife darstellt. Der Used Service `C18a7efd_16cd7d5S_1` stellt den Aufruf der "S"-Komponente `C18a7efd_16cd7d5S_1` dar, die die Kette der mehrfach zu wiederholenden Aktivitäten darstellt. Diese bestehen in zwei aufeinander folgenden Aufrufen an die CPU `CPU1`, die jeweils den Umfang von 5 Einheiten besitzen.

In der Tabelle 4.8 werden die Ergebnisse der Auswertung am Anwendungsfall angegeben, der auf das Aktivitätsdiagramm *A5* verweist. Auch hier wurden nicht die Antwortzeiten der einzelnen Aktivitäten erhoben, da diese für das betrachtete Aktivitätsdiagramm *A5* keine Aussagekraft hätten.

Turnaroundtime	SIMULATIVE
Mean	10.0
Stdev	0.0
Con 95%	± 0.00%

Tabelle 4.8: Berechnungsergebnisse zum Loop-Diagramm

Als Wert für die mittlere Antwortzeit für den Diagrammdurchlauf ergeben sich 10 Sekunden. Daraus folgt, dass das realisierte Verhalten dem erwarteten entspricht.

4.2.6 Rechenbeispiel mit Aktivitäten und Branches

Im folgenden soll die Korrektheit der Umsetzung anhand einer Beurteilung der qualitativen Richtigkeit der Berechnungsergebnisse aufgezeigt werden. Die in diesem Kapitel bereits betrachteten Aktivitätsdiagramme sind für diesen Zweck zu komplex, daher wurde ein weiteres einfaches Aktivitätsdiagramm entworfen. Dieses Diagramm adaptiert das M/M/1||M/M/1-Modell aus dem Abschnitt 4.1.3, das dort in Form eines Use-Case-Diagramms realisiert wurde.

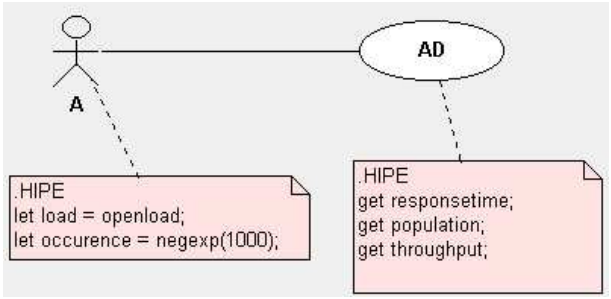

Abbildung 4.15: Anwendungsfalldiagramm zum Rechenbeispiel für Aktivitätsdiagramme

Das Anwendungsfalldiagramm in Abbildung 4.15 auf S. 158 verweist auf ein Aktivitätsdiagramm, wobei die mittlere Antwortzeit, Population und Durchsatz durch HIT erhoben werden sollen. Da einerseits HIT nur die Auslastung flacher Komponenten messen kann, andererseits Aktivitätsdiagramme aber auf selbstdefinierte Komponenten abgebildet werden, kann die Auslastung hier nicht erhoben werden.

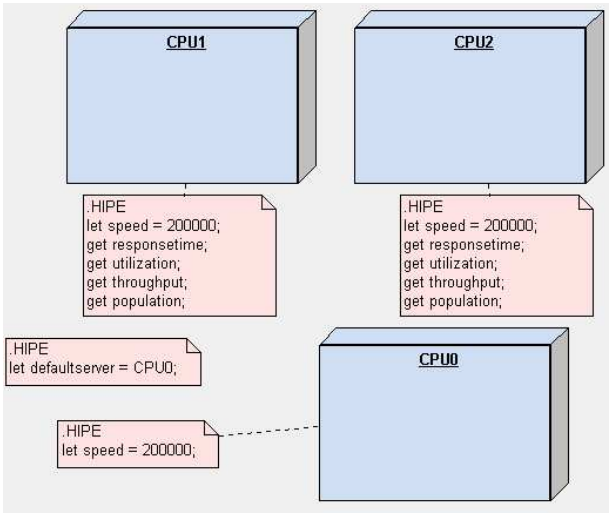

Abbildung 4.16: Verteilungsdiagramm zum Rechenbeispiel für Aktivitätsdiagramme

Das Verteilungsdiagramm in Abbildung 4.16 auf S. 158 stellt ein System mit insgesamt drei CPUs *CPU0*, *CPU1* und *CPU2* dar, wobei die CPU mit dem Namen *CPU0* zur Default-CPU erklärt wird. Die einzelnen CPUs haben jeweils eine Geschwindigkeit von 200000 Anweisungen pro Sekunde. An *CPU1* und *CPU2* sollen weiterhin die im Rahmen der harten Analyse erhebbaren Maße Turnaroundtime, Utilization, Throughput und Population erhoben werden.

Zu der Einführung und Verwendung von *CPU0* ist anzumerken, dass Entscheidungsknoten stets auf Aufrufe der Default-CPU abgebildet werden. HIT errechnet Antwortzeiten durch Bildung von Mittelwerten über alle Anfragen an einen Bediener. Werden Aufrufe von Entscheidungsknoten auf einen Bediener abgebildet, der auch andere Anfragen bearbeiten muss,

können dadurch bei der Berechnung von Messwerten an diesem Bediener Verfälschungen auftreten, das heißt, Messwerte entsprechen eventuell nicht den Erwartungen. Nach der Art, wie HIPE in der aktuellen Version Aufrufe umsetzt (siehe Tabelle 1.6, Seite 50 und die Ausführungen auf Seite 141), ist es somit sinnvoll, nur zum Zwecke der Übernahme von Entscheidungen eine weitere schnelle CPU einzuführen, die als Default-CPU markiert wird. Messungen sowohl an den anderen CPUs als auch an Aktivitäten in den Aktivitätsdiagrammen oder Anwendungsfällen in den Anwendungsfalldiagrammen liefern dann die erwarteten Ergebnisse.

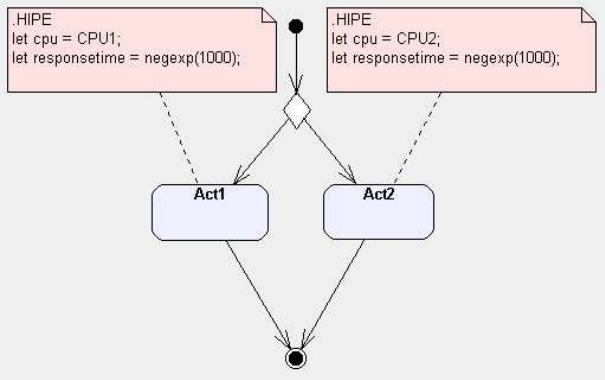

Abbildung 4.17: Aktivitätsdiagramm zum Rechenbeispiel für Aktivitätsdiagramme

Das Aktivitätsdiagramm wird in Abbildung 4.17 auf S. 159 dargestellt. Es gibt einen Entscheidungsknoten, der entsprechend der obigen Anmerkung auf die Default-CPU abgebildet wird, und zwei Aktivitäten, die jeweils auf eine der beiden übrigen CPUs *CPU1* und *CPU2* abgebildet werden. Die Bedienzeitanforderungen der Aktivitäten sind negativ exponentialverteilt mit einem Erwartungswert von 1/1000 Zeiteinheit. Es sei noch einmal analog zur Anmerkung auf der Seite 129 darauf hingewiesen, dass sich die letztendlich realisierten Antwortzeiten der Aktivitäten erst unter der Betrachtung der tatsächlichen Arbeitslast auf den Komponenten ergeben, die diese Aktivitäten ausführen. Diese können sich von den Bedienzeitanforderungen unterscheiden.

Erwartetes Verhalten
Wie bereits angemerkt ist das Modell eine Adaption des M/M/1||M/M/1-Modells. Die erwarteten Ergebnisse entsprechen den Messergebnissen aus dem Kapitel 4.1.3. Sie werden für jeden Messort noch einmal in der Tabelle 4.9 aufgeführt. Zu beachten ist, dass wegen der Art der Umsetzung von Modellen durch HIPE die Auslastung (Utilization) am Gesamtsystem nicht gemessen werden kann.

Mean	Population	Throughput	Turnaroundtime	Utilization
Einzelsystem (Messort CPUx)	1.0	500.0	0.002	0.5
Gesamtsystem (Messort AD)	2.0	1000.0	0.002	

Tabelle 4.9: Vergleichswerte zum Rechenbeispiel für Aktivitätsdiagramme

Realisiertes Verhalten
Das betrachtete Modell wird in HIPE eingespeist und die Ergebnisse betrachtet, die von der harten Analyse geliefert werden. Die Tabelle 4.10 stellt die Ergebnisse dar, die mit HIT unter dem Löser `SIMULATIVE` ermittelt wurden.

CPU1	Population	Throughput	Turnaroundtime	Utilization
Mean	0.987976	503.716813	0.001962	0.499056
Stdev	1.363691	0.001975	0.002432	0.499999
Con 95%	± 7.06%	± 2.61%	± 5.86%	± 3.25%
CPU2	Population	Throughput	Turnaroundtime	Utilization
Mean	1.036863	503.407442	0.002060	0.504519
Stdev	1.491727	0.001972	0.002701	0.499980
Con 95%	± 8.49%	± 2.39%	± 7.43%	± 3.20%
AD	Population	Throughput	Turnaroundtime	
Mean	1.999762	1009.261399	0.002011	
Stdev	2.028849	0.000988	0.002570	
Con 95%	± 6.03%	± 1.88%	± 4.64%	

Tabelle 4.10: Ergebnisse zum Rechenbeispiel für Aktivitätsdiagramme

Die errechneten Ergebnisse stimmen im Rahmen der Wahrscheinlichkeit mit den erwarteten Ergebnissen überein. Daraus folgt, dass HIPE das erwartete Verhalten zeigt.

Nachtrag des Korrektors: *In den verschiedenen Rechenbeispielen wurde mehrfach die Art, wie HIPE Aktivitäten umsetzt, als Ursache dafür festgestellt, dass bestimmte Messungen keine Aussagekraft haben. HIPE setzt atomare Aktivitäten direkt auf Aufrufe der CPU um. Dadurch bezieht sich die Erhebung eines Leistungsmaßes an einer atomaren Aktivität sich nicht allein auf diese Aktivität, liefert also nicht den Mittelwert dieses Leistungsmaßes über alle Aufrufe dieser Aktivität, sondern den Wert über alle Aufrufe dieser CPU. Da die Messung für eine einzelne Aktivität ggf. interessant ist, um kritische Stellen besser beurteilen zu können, wird der Verbesserungsvorschlag gemacht, die Abbildung atomarer Aktivitäten so abzuändern, dass dort auch wirklich nur Aufrufe der entsprechenden Aktivität bewertet werden. In Absprache mit dem Projektgruppenleiter wurde die dafür notwendige Änderung nicht mehr implementiert.*

4.3 Test der weichen Analyse

Durch Hinzufügen eines Klassendiagramms zum M/M/1-Modell im Abschnitt 4.1.2 kann die weiche Analyse getestet werden. Das Klassendiagramm ist in Abbildung 4.18 auf S. 161 dargestellt. Die weiche Analyse liefert für dieses Diagramm folgende Ausgabe auf der Konsole, wobei die Syntax der Auflistung dem Schema

`---- <weiches Maß> <Klassen- bzw. Diagrammname> <berechneter Wert des Maßes>`

folgt.

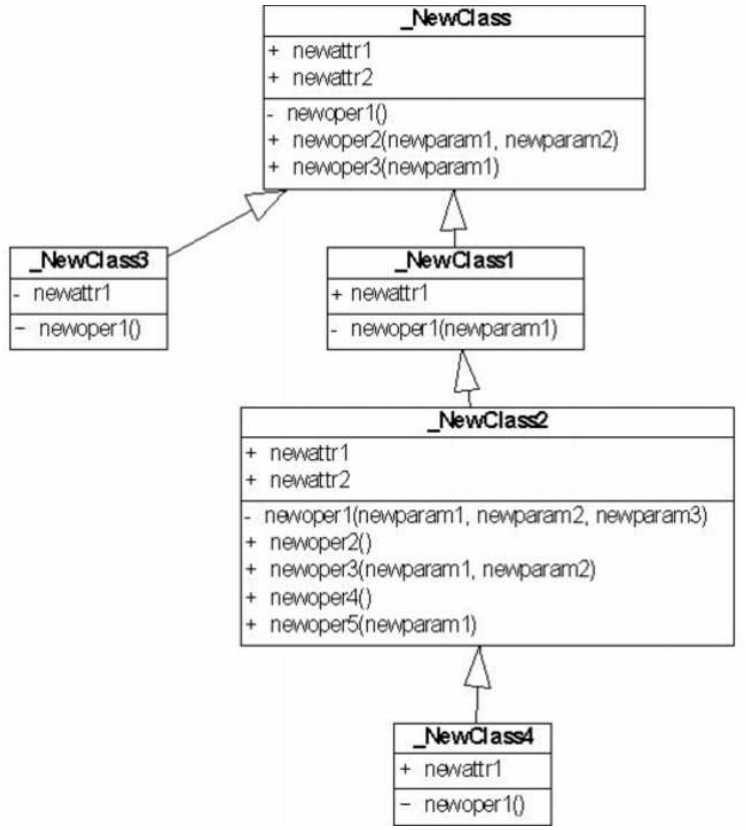

Abbildung 4.18: Das Klassendiagramm des M/M/1-Modells

```
&&&&&&&&&&&& Weiche Analyse gestartet &&&&&&&&&&&&
----DIP _NewClass 0
----DIP _NewClass1 1
----DIP _NewClass2 2
----DIP _NewClass3 1
----DIP _NewClass4 3
----NOC _NewClass 2
----NOC _NewClass1 1
----NOC _NewClass2 1
----NOC _NewClass3 0
----NOC _NewClass4 0
----NOT Klassendiagramm 5
----MIF Klassendiagramm 0.5217391304347826
----AIF Klassendiagramm 0.5
----MHF_NewClass 0.3333333333333333
----MHF_NewClass1 1.0
----MHF_NewClass2 0.2
----MHF_NewClass3 0.0
----MHF_NewClass4 0.0
----AHF_NewClass 0.0
```

```
----AHF_NewClass1 1.0
----AHF_NewClass2 0.0
----AHF_NewClass3 1.0
----AHF_NewClass4 0.0
----PF Klassendiagramm 0.95
----COF Klassendiagramm 0.35
----CP _NewClass 39.0
----LOC _NewClass 133.94070685374635
----CP _NewClass1 15.0
----LOC _NewClass1 70.56775051475124
----CP _NewClass2 66.0
----LOC _NewClass2 191.7935859204229
----CP _NewClass3 6.0
----LOC _NewClass3 39.05144086407193
----CP _NewClass4 6.0
----LOC _NewClass4 39.05144086407193
&&&&&&&&&&&& Weiche Analyse beendet &&&&&&&&&&&&&
```

Um diese Ergebnisse zu überprüfen, wurden die Werte nochmals per Hand nachgerechnet:

Nachtrag des Korrektors: *Die folgenden Kontroll-Rechnungen zeigen, dass die weiche Analyse den Test* ***nicht besteht****. Hier wurden offensichtlich die Formeln nicht korrekt interpretiert. Insbesondere wurde die Vererbungshierarchie des Klassendiagramms bei einigen weichen Maßen (z.B. MIF, AIF, MHF, AHF, ...) außer Acht gelassen. Aufgrund des allgegenwärtigen Zeit- und Arbeitskraftmangels konnte dieses Manko nicht mehr behoben werden, so dass die Ergebnisse einer weichen Analyse mit HIPE* ***nicht verwertbar*** *sind. Dies betrifft jedoch nicht die Ergebnisse der harten Analyse. Sie sind nicht von den falsch implementierten weichen Leistungsmaßen abhängig (s.u.).*

- Tabelle 4.11 zeigt die Resultate der Rechnungen bezüglich des DIP-Index (*Depth of Inheritance*). Er beschreibt die Tiefe einer Klasse im Vererbungsbaum, also die Anzahl an Elternklassen.

Klasse	DIP
_NewClass	0
_NewClass1	1
_NewClass2	2
_NewClass3	1
_NewClass4	3

Tabelle 4.11: Weiches Maß DIP

- In Tabelle 4.12 sind die manuell ermittelten Ergebnisse bezüglich des NOC-Index (*Number of Children*) aufgelistet. Er beziffert die Anzahl von Subklassen einer Klasse.
- Die Gesamtanzahl an Klassen ist 5 (NOT, *Number of Types*).

Klasse	NOC
_NewClass	2
_NewClass1	1
_NewClass2	1
_NewClass3	0
_NewClass4	0

Tabelle 4.12: Weiches Maß NOC

- Tabelle 4.13 beschreibt das Vorkommen an geerbten und lokal definierten Methoden für die einzelnen Klassen. Das diagrammweit ermittelte Verhältnis der Anzahl geererbter Methoden zur Anzahl aller Methoden wird damit durch den MIF-Index definiert (*Method Inheritance Factor*):

$$MIF = \frac{Anzahl\ geerbter\ Methoden}{Anzahl\ aller\ Methoden} = \frac{19}{19+11} = 0.63$$

Klasse	geerbte Methoden	lokal definierte Methoden
_NewClass	0	3
_NewClass1	3	1
_NewClass2	4	5
_NewClass3	3	1
_NewClass4	9	1
Summe	19	11

Tabelle 4.13: Weiches Maß MIF

- Analog zum MIF beschreibt der AIF (*Attribute Inheritance Factor*) das diagrammweite Verhältnis der Anzahl geererbter Attribute zur Anzahl aller Attribute. Die dafür benötigten Kardinalitäten der Mengen der geerbten und lokal definierten Attribute sind für die einzelnen Klassen in Tabelle 4.14 dargestellt. Der AIF berechnet sich somit zu:

$$AIF = \frac{Anzahl\ geerbter\ Attribute}{Anzahl\ aller\ Attribute} = \frac{12}{12+7} = 0.63$$

- Der MHF (*Method Hiding Factor*) beschreibt pro Klasse das Verhältnis der Anzahl versteckter Methoden zur Anzahl aller Methoden. Für die im Testmodell vorhandenen Klassen sind die einzelnen MHF-Werte der Tabelle 4.15 zu entnehmen.

- Der AHF (*Attribute Hiding Factor*) beschreibt analog zum MHF pro Klasse das Verhältnis der Anzahl versteckter Attribute zur Anzahl aller Attribute (siehe Tabelle 4.16).

- Der Polymorphismusfaktor für das Klassendiagramm wurde anhand der Formel in Abschnitt 2.5 berechnet. Tabelle 4.17 enthält die damit errechneten Werte, wobei die Spalte "Zwischenergebnis" den für jede Klasse zu berechnenden Summanden des Polymorphismusfaktors angibt.

Klasse	geerbte Attribute	lokal definierte Attribute
_NewClass	0	2
_NewClass1	2	1
_NewClass2	3	2
_NewClass3	2	1
_NewClass4	5	1
Summe	12	7

Tabelle 4.14: Weiches Maß AIF

Klassen	versteckte Methoden	Methoden insgesamt	MHF
_NewClass	1	3	0.33
_NewClass1	2	4	0.5
_NewClass2	3	9	0.33
_NewClass3	1	1	1.0
_NewClass4	1	1	1.0

Tabelle 4.15: Weiches Maß MHF

Klassen	versteckte Attribute	Attribute insgesamt	AHF
_NewClass	0	2	0
_NewClass1	1	2	0.5
_NewClass2	0	2	0
_NewClass3	1	2	0.5
_NewClass4	0	2	0

Tabelle 4.16: Weiches Maß AHF

Klassen	überschriebene Methoden	Methoden insgesamt	Subklassen	Zwischenergebnis
_NewClass	1	3	4	0.83
_NewClass1	0	4	2	0
_NewClass2	0	9	1	0
_NewClass3	0	3	0	0
_NewClass4	0	9	0	0
PF =				0.83

Tabelle 4.17: Weiches Maß PF

- Analog zum Polymorphismusfaktor wurde der von HIPE gelieferte Kopplungsfaktor (COF) anhand der Formel im Abschnitt 2.5 kontrolliert (die Beziehungen zwischen den Klassen bezüglich der *isClient*-Funktion sind in Tabelle 4.18 abgebildet). Dabei steht das Kürzel TC für die Gesamtzahl aller Klassen.

Nachtrag des Korrektors: *Hier wurde die Semantik der Funktion isClient missverstanden. Statt einer Vererbungsbeziehung — wie bei der Implementierung angenommen — drückt diese Funktion das Vorhandensein einer Assoziation zwischen zwei Klassen aus. Das Test-Diagramm ist für letztere allerdings ungeeignet, da darin keine Assoziationen enthalten sind. Der Kopplungsfaktor für dieses Beispieldiagramm ist demnach gleich null.*

$$COF = \sum \frac{\sum isClient(C_i, C_j)}{TC^2 - TC} = \frac{0}{5^2 - 5} = 0$$

isClient	_NewClass	_NewClass1	_NewClass2	_NewClass3	_NewClass4
_NewClass	-	0	0	0	0
_NewClass1	-	-	0	0	0
_NewClass2	-	-	-	0	0
_NewClass3	-	-	-	-	0
_NewClass4	-	-	-	-	-

Tabelle 4.18: Weiches Maß COF

- Die *Complexity Points CP* und der (für eine Leistungsanalyse irrelevante) *Lines Of Code*-Index *LOC* wurden anhand zweier Beispielklassen (_NewClass und _NewClass2) stichprobenartig verifiziert. Für beide Klassen wurden nach dem in Kapitel 1.2.5 beschriebenen Vorgehen die Complexity Points berechnet und zusätzlich für jede Klasse die erwartete Anzahl an Quellcode $LOC(c)$ mit folgender Formel ermittelt:

$$LOC(c) = 4 + 10CP(c)^{0.7}$$

Nachtrag des Korrektors: *Die folgenden Berechnungen (insbesondere die Tabellen 4.19 und 4.20) wurden unter der idealisierenden Annahme erstellt, dass sämtliche im Diagramm verwendeten Attribute und Methoden-Parameter vom Typ „Heavy Attributes“ sind. Es wurde offensichtlich darauf verzichtet, die Attribut- und Parametertypen individuell zu spezifizieren. Unter dieser Voraussetzung sind die Ergebnisse korrekt.*

  - Die Berechnung der Methodenkomplexität für die Klasse _NewClass ist in Tabelle 4.19 zu sehen. $CP(\_NewClass)$ und $LOC(\_NewClass)$ lassen sich somit wie folgt ermitteln:

$$\begin{aligned}
BP(_NewClass) &= (1 + numAss(_NewClass)) * summeCM \\
&= 1 * 9 \\
&= 9 \\
SP(_NewClass) &= numLA(_NewClass) + 3 * numHA(_NewClass) + \\
&\quad 5 * numIA(_NewClass) \\
&= 0 + 3 * 2 + 0 \\
&= 6 \\
CP(_NewClass) &= 2 * SP(_NewClass) + 3 * BP(_NewClass)
\end{aligned}$$

$$
\begin{aligned}
&= 2 * 6 + 3 * 9 \\
&= 39 \\
LOC(\_NewClass) &= 4 + 10 * 39^{0.7} \\
&= 133.94
\end{aligned}
$$

	numLA	numHA	numIA	(T,S)	CM
newoper1	0	0	0	(0,1)	0
newoper2	0	2	0	(0,1)	6
newoper3	0	1	0	(0,1)	3
Summe CM					9

Tabelle 4.19: Berechnung von CM für NewClass

– Die Berechnung der Methodenkomplexität für die Klasse _NewClass2 ist in Tabelle 4.20 skizziert. $CP(\_NewClass2)$ und $LOC(\_NewClass2)$ ergeben sich damit zu:

$$
\begin{aligned}
BP(\_NewClass2) &= (1 + numAss(\_NewClass2)) * summeCM \\
&= 1 * 18 \\
&= 18 \\
SP(\_NewClass2) &= numLA(\_NewClass2) + 3 * numHA(\_NewClass2) + \\
&\quad 5 * numIA(\_NewClass2) \\
&= 0 + 3 * 2 + 0 \\
&= 6 \\
CP\_NewClass2) &= 2 * SP(\_NewClass2) + 3 * BP(\_NewClass2) \\
&= 2 * 6 + 3 * 18 \\
&= 66 \\
LOC\_NewClass2) &= 4 + 10 * 66^{0.7} \\
&= 191.79
\end{aligned}
$$

4.4 Test des Bankomat-Modells

Das Bankomat-Modell ist ein sehr komplexes Modell, das zu dem Zweck erstellt wurde, die Funktionen von HIPE auszureizen. Anhand dieses Modells soll insbesondere die Fähigkeit von HIPE demonstriert werden, von HIT akzeptierte Modellhierarchien aufzubauen, das heißt, modellweit mehrfach verwendete Komponenten korrekt in der Modellhierarchie zu platzieren. Das Modell besteht aus einem komplexen Klassendiagramm, einem Use-Case-Diagram, einem Deploymentdiagramm und mehreren Aktivitätsdiagrammen, die die einzelnen Use-Cases

	numLA	numHA	numIA	(T,S)	CM
newoper1	0	3	0	(0,1)	9
newoper2	0	0	0	(0,1)	0
newoper3	0	2	0	(0,1)	6
newoper4	0	0	0	(0,1)	0
newoper5	0	1	0	(0,1)	3
Summe CM					18

Tabelle 4.20: Berechnung von CM für NewClass2

verfeinern. Das Use-Case-Diagramm enthält die vier Anwendungsfälle des Modells (Abbildung 4.19 auf S. 167). Der Akteur Kunde hat die Möglichkeit, entweder seinen Kontostand zu überprüfen, eine Überweisung zu tätigen oder Geld abzuheben. Wenn der Kunde Geld abheben möchte, kommt es darauf an, ob er an dem Bankomaten seiner Hausbank oder aber an einem Fremdautomaten Geld abhebt. Dementsprechend gibt es für die Tätigkeit "Geld abheben" zwei verschiedene Anwendungsfälle. Das Deployment diagram (Abbildung 4.20 auf S. 168) enthält

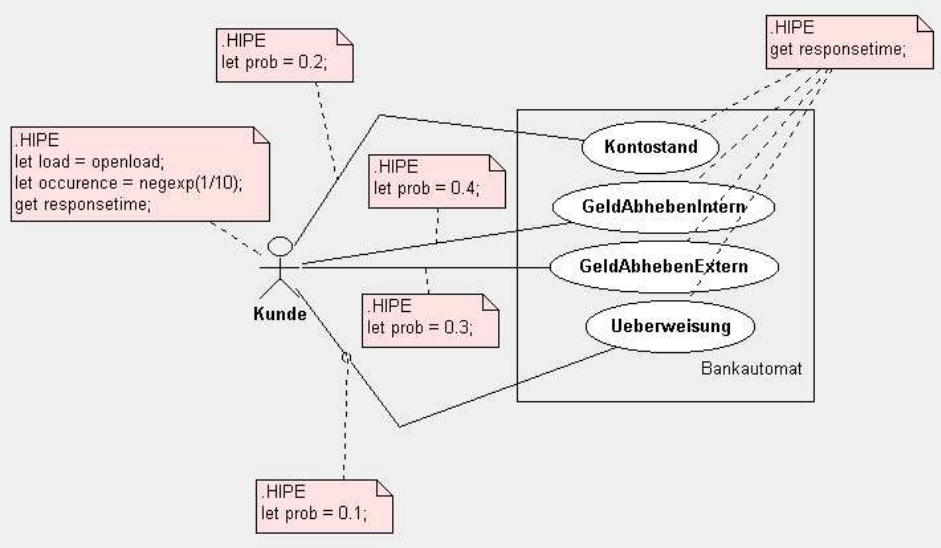

Abbildung 4.19: Bankomat: Anwendungsfälle

die verschiedenen Serverkomponenten und ihre Annotationen. Die meisten Klassen, aus denen das Bankomat-Modell besteht, sind in der Hardware-Komponente *Bankomat* zu finden, werden also auf dieser ausgeführt. Bis auf den Anwendungsfall *Geldabhebenextern* benötigen alle Anwendungsfälle zu ihrer Ausführung den *Bankomat* und den *Bankserver* (Lokaler Server der Hausbank). Diejenigen Klassen des Klassendiagramms (Abbildung 4.21 auf S. 169), die in den Aktivitäten zur Ausführung kommen, haben jeweils einen Bezug zu einer Komponente aus dem Deploymentdiagramm. Die Anwendungsfälle werden in gleichnahmigen Aktivitätsdiagrammen verfeinert (Abbildungen 4.22, 4.23, 4.24 und 4.25). Diese Aktivitätsdiagramme wiederum enthalten Aktivitäten, die ihrerseits in gleichnamigen Aktivitätsdiagrammen ver-

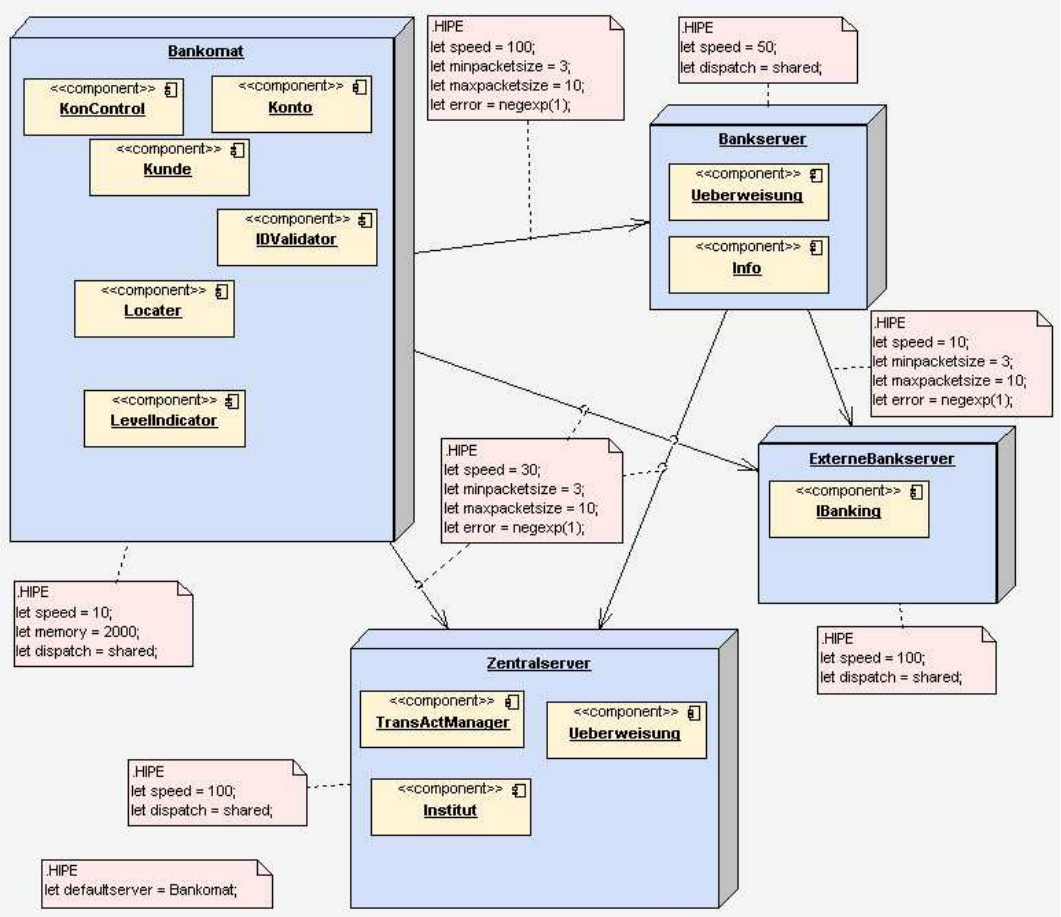

Abbildung 4.20: Bankomat: Deploymentdiagramm

feinert werden (Abbildungen 4.27, 4.28, 4.29 und 4.30). Was die Komplexität und somit den Vergleich zu realen UML-Diagrammen vergrössert, ist die Tatsache, dass die Aktivitäten nicht nur verschachtelt sind, sondern auch häufig wiederverwendet werden. Ein gutes Beispiel dafür ist die Aktivität *Ueberpruefen* (Abbildung 4.26 auf S. 172). Bevor der Kunde irgendeine Aktion durchführen kann, muss erst überprüft werden, ob er ein Kunde der Bank ist, an dessen Bankomaten er seine Karte eingegeben hat (Abbildung 4.29 auf S. 174). Ausserdem muss geprüft werden, ob bezüglich des Kunden oder des Kontos Sperrungen ausgesprochen wurden (Abbildung 4.27 auf S. 173). Eine letzte Prüfung stellt sicher, dass der angeforderte Betrag nicht zu hoch bzw. zu niedrig ist. Das Überprüfen des Kunden findet dabei in einer Unteraktivität von *Kontopruefen* statt (Abbildung 4.28 auf S. 173). Andere gute Beispiele für die genannte Verschachtelung sind die Aktivitäten *KontoSperren* (Abbildung 4.30 auf S. 174) und *Abbruch*. Ausserdem sind Konstruktionen wie Schleifen und Parallelität in die eine oder andere Aktivität eingebaut, um herauszufinden, ob sie von HIPE korrekt in die HI-SLANG-Äquivalente (`LOOP`- und `CONCURRENT`-Block) übersetzt werden. Hervorzuheben ist auch die Verwendung von Objektflüssen in den Aktivitätsdiagrammen. Durch sie machen die weichen Maße, die vor der harten Analyse ermittelt werden, erst richtig Sinn. Wie bereits erwähnt reicht es nicht, diese nur in dem jeweiligen Aktivitätsdiagramm über Objektflüsse einzubinden. Zusätzlich müssen

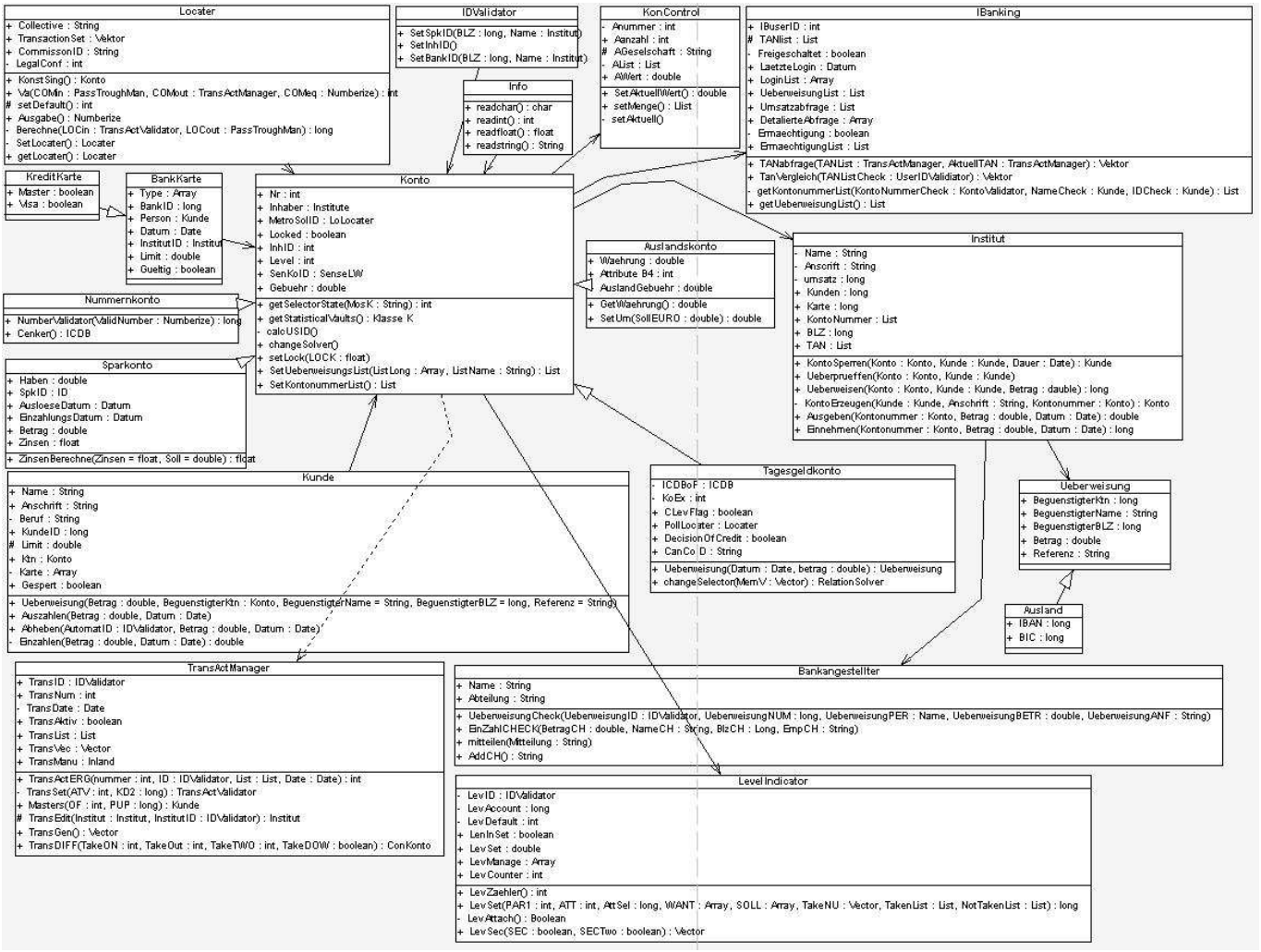

Abbildung 4.21: Bankomat: Klassendiagramm

diese Objekte bzw. ihre Klassen im jeweiligen Deploymentdiagramm als Unterkomponente derjenigen Komponente definiert werden, auf der sie auch ausgeführt werden.

Gemäß der Regeln, die in HIT für die Erstellung von Hierarchien gelten, muss eine Komponente, die von mehreren anderen Komponenten innerhalb des Modells zugegriffen (in HI-SLANG ENCLOSEd) wird, innerhalb des hierarchischen Modells so platziert werden, dass sie von allen diesen Komponenten aus sichtbar ist. Dies ist der Fall, wenn sie in einer Komponente des Modells als COMPONENT definiert wird, die auf dem Stück der Pfade von den einzelnen Komponenten hin zur Wurzel liegt, das diese alle gemeinsam haben. Von den selbstdefinierten Komponenten `M.C1.C11` und `M.C1.C12` aus sind also alle Komponenten gemeinsam sichtbar, die in den Komponenten `M.C1` und `M`, nicht aber solche, die in der Komponente `M.C2` deklariert werden. Im folgenden soll überprüft werden, ob HIPE hierarchische Modelle im Sinne von HIT korrekt implementiert.

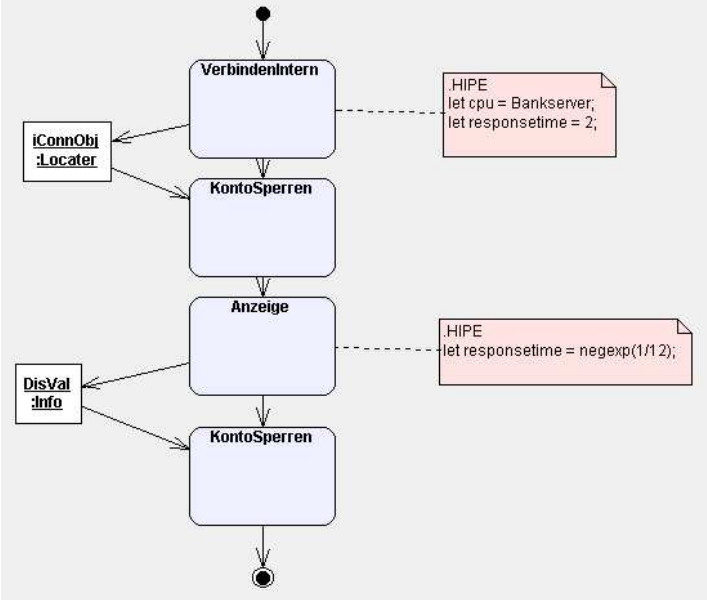

Abbildung 4.22: Bankomat: Verfeinerung des Anwendungsfalls *Kontostand*

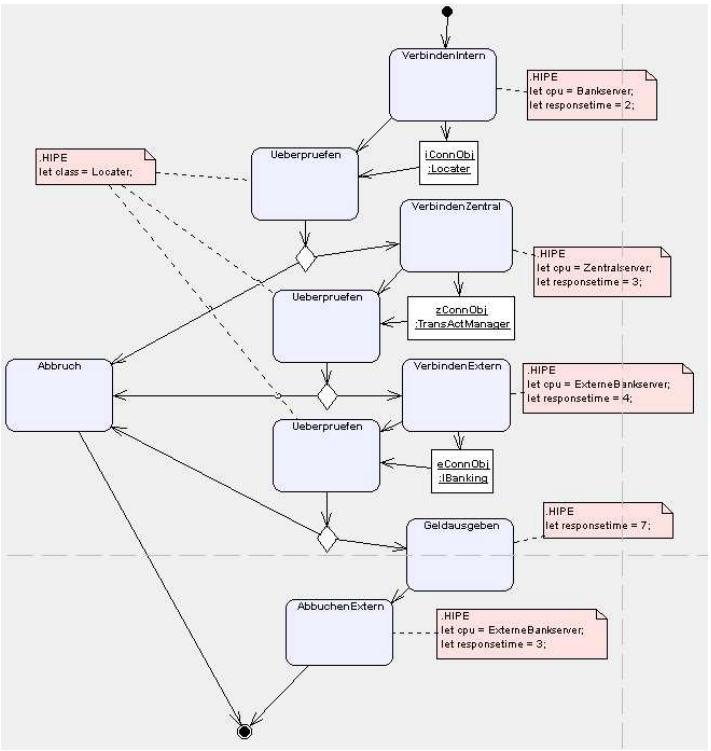

Abbildung 4.23: Bankomat: Verfeinerung des Anwendungsfalls *GeldAbhebenExtern*

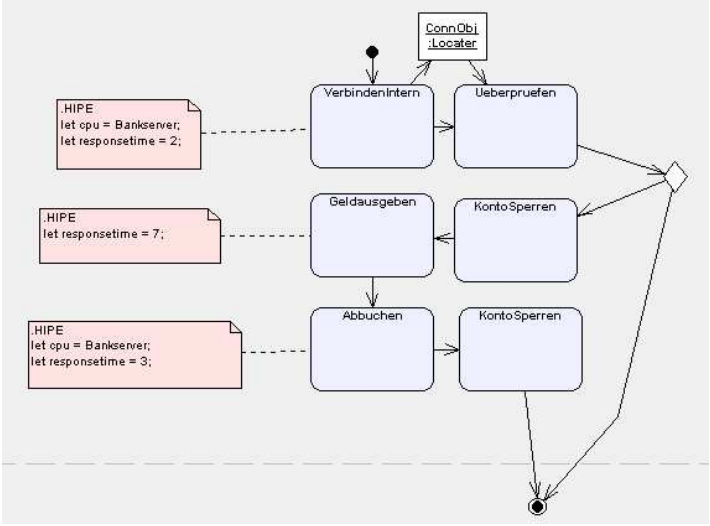

Abbildung 4.24: Bankomat: Verfeinerung des Anwendungsfalls *GeldAbhebenIntern*

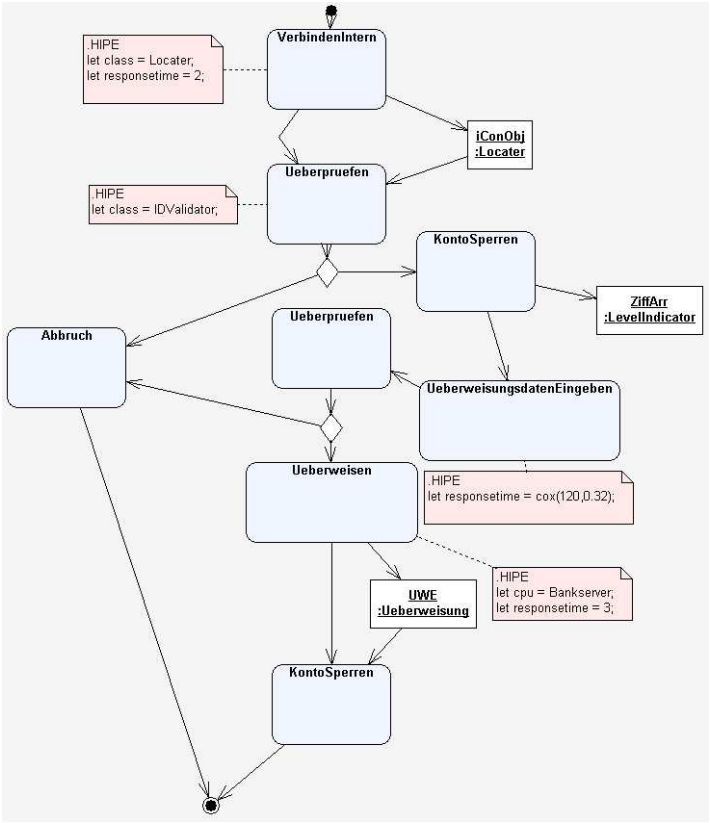

Abbildung 4.25: Bankomat: Verfeinerung des Anwendungsfalls *Ueberweisung*

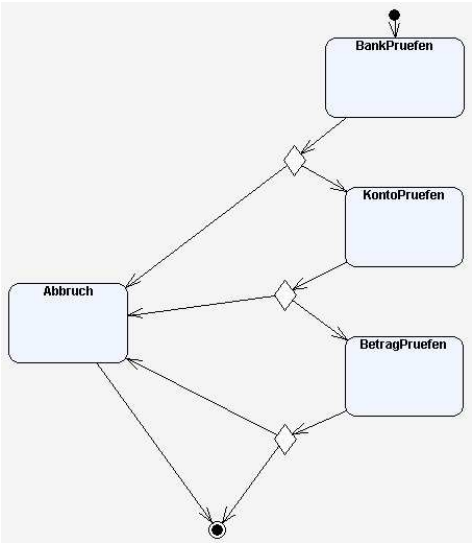

Abbildung 4.26: Bankomat: Verfeinerung der Aktivität *UeberPruefen*

Erwartetes Verhalten
Das Modell ist zu komplex, um alle Komponenten, die dabei erzeugt und von mehreren anderen Komponenten aus zugegriffen werden, wirklich zu verfolgen. Deshalb beschränkt sich der Test auf eine einzelne gemeinsam genutzte Komponente.

Als Beispielkomponente wird hier die Leitung zwischen den CPUs *Bankomat* und *Bankserver* gewählt, die in den Aktivitätsdiagrammen *GeldAbhebenExtern*, *GeldAbhebenIntern*, *Kontostand* und *Ueberweisung* zum Transport von Objekten des Typs *Ueberweisung* genutzt wird. Da die Leitung entsprechend von allen vier Aktivitätsdiagrammen benutzt wird, die die Anwendungsfälle definieren, muss sie gemäß der Hierarchieregel innerhalb des MODEL definiert werden, das das Use-Case-Diagramm darstellt.

Realisiertes Verhalten
Das erzeugte Modell hat mehr als 2000 Zeilen Code, daher stellt sich die Einschränkung auf eine Beispielkomponente als richtig heraus. In der von HIPE erzeugten hierarchischen Struktur des Modells wird für die ausgesuchte Beispielkomponente festgestellt, wo sich die Komponenten in der Hierarchie befinden, die auf sie zugreifen und wo die Beispielkomponente selbst erscheint.

Der folgende Codeabschnitt stellt die relevanten Teile der MODEL-Komponente und der selbstdefinierten Komponenten des erzeugten hierarchischen Modells dar, in denen die oben genannten Aktivitätsdiagramme dargestellt werden:

```
-->005 TYPE ProjektTest MODEL;
-->006   TYPE Kunde SERVICE;
-->007     USE
-->008       SERVICE
-->009         AKontostand_1;
-->010         AGeldAbhebenIntern_1;
-->011         AUeberweisung_1;
-->012         AGeldAbhebenExtern_1;
-->013     END USE;
-->014     BEGIN
```

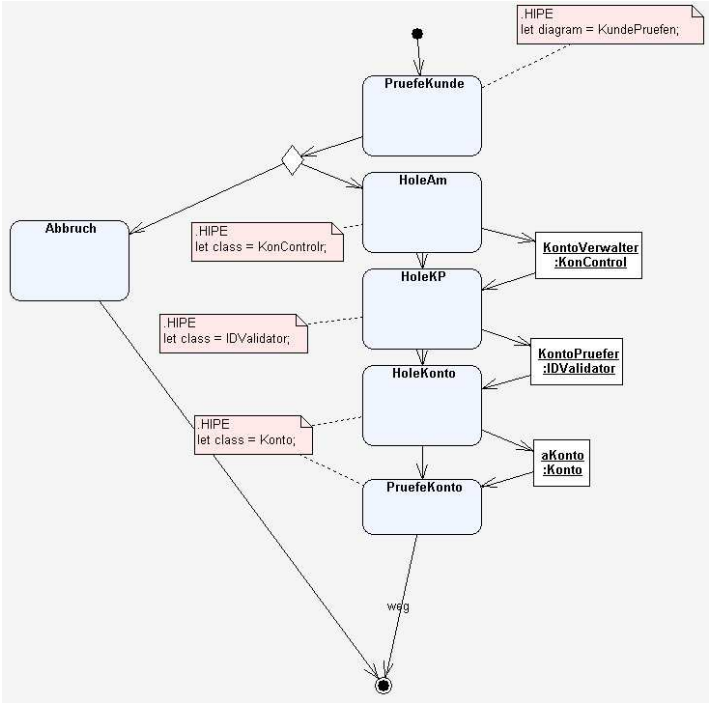

Abbildung 4.27: Bankomat: Verfeinerung der Aktivität *KontoPruefen*

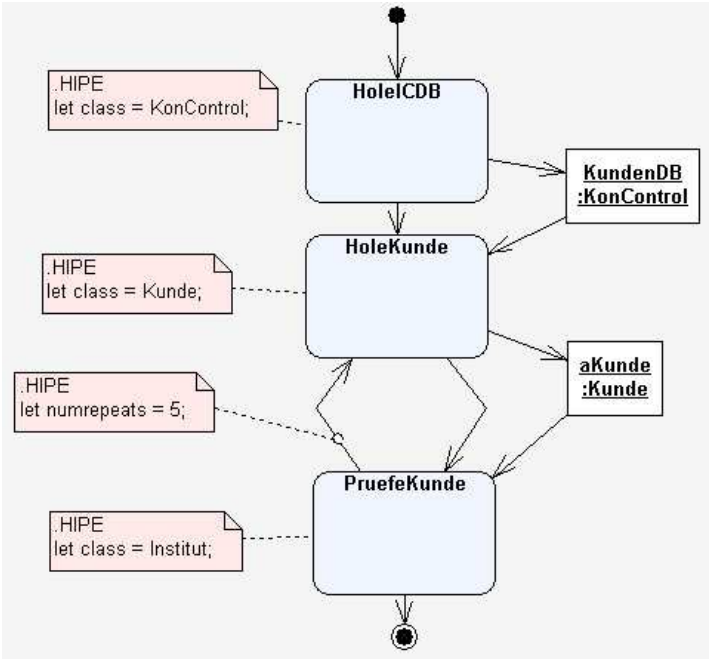

Abbildung 4.28: Bankomat: Verfeinerung der Aktivität *KundePruefen*

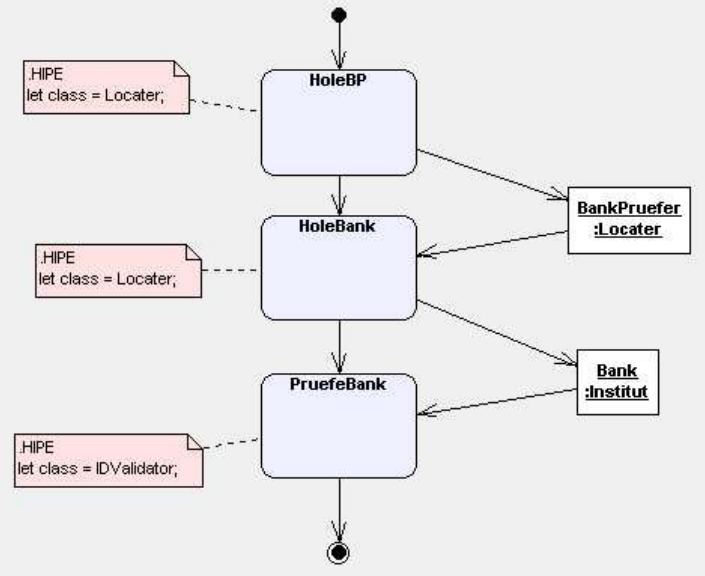

Abbildung 4.29: Bankomat: Verfeinerung der Aktivität *BankPruefen*

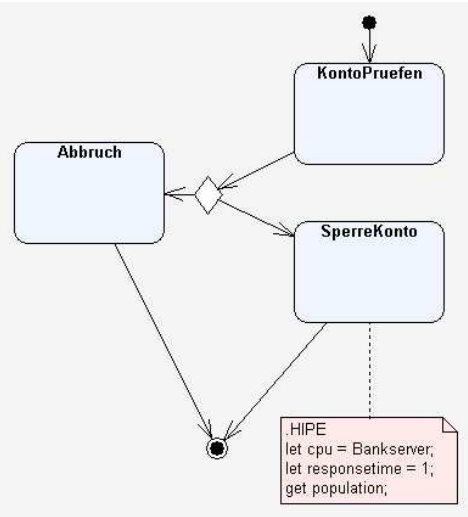

Abbildung 4.30: Bankomat: Verfeinerung der Aktivität *KontoSperren*

```
-->015       BRANCH
-->016       PROB 0.2: AKontostand_1;
-->017       PROB 0.4: AGeldAbhebenIntern_1;
-->018       PROB 0.1: AUeberweisung_1;
-->019       PROB 0.3: AGeldAbhebenExtern_1;
-->020       END BRANCH;
-->021    END TYPE Kunde;
-->022    TYPE AKontostand COMPONENT;
% Innerhalb der Komponente vom Typ AKontostand wird
% auf die Komponente vom Typ Bankomat_Bankserver verwiesen:
-->108     ENCLOSE Bankomat_Bankserver_1 : Bankomat_Bankserver;
% Diese wird auf die Definition von AKontostand folgend
% definiert:
-->136    END TYPE AKontostand;
-->137    TYPE Bankomat_Bankserver COMPONENT;
% ...
```

```
-->169   END TYPE Bankomat_Bankserver;
% ...
% Das Diagramm GeldAbhebenIntern verwendet ebenfalls
% die Komponente vom Typ Bankomat_Bankserver modellweit,
% ebenso Ueberweisung und GeldAbhebenExtern:
-->718   TYPE AGeldAbhebenIntern COMPONENT;
% ...
-->798     ENCLOSE Bankomat_Bankserver_1 : Bankomat_Bankserver;
% ...
-->826   END TYPE AGeldAbhebenIntern;
% ...
-->1613   TYPE AUeberweisung COMPONENT;
% ...
-->1688     ENCLOSE Bankomat_Bankserver_1 : Bankomat_Bankserver;
% ...
-->1712   END TYPE AUeberweisung;
-->1713   TYPE AGeldAbhebenExtern COMPONENT;
% ...
-->1972     ENCLOSE Bankomat_Bankserver_1 : Bankomat_Bankserver;
% ...
-->2058   END TYPE AGeldAbhebenExtern;
% ...
% Die modellweit verwendete Komponente vom Typ Bankomat_\
% Bankserver wird im MODEL definiert:
-->2067   COMPONENT Bankomat_Bankserver_1 : Bankomat_Bankserver;
% ...
-->2087 END TYPE ProjektTest;
% ...
```

Das Aktivitätsdiagramm *Kontostand* wird in die Hierarchie als Komponente `ProjektTest.AKontostand_1` vom Typ `AKontostand` eingeordnet, *GeldAbhebenIntern* als Komponente `ProjektTest.AGeldAbhebenIntern_1` vom Typ `AGeldabhebenIntern` und die beiden anderen Aktivitätsdiagramme analog. Die Pfade zur Wurzel des Modells, die von diesen Komponenten ausgehen, haben nur das Hierarchieelement `ProjektTest` gemeinsam, das in diesem Fall gerade die MODEL-Komponente des hierarchischen Modells bezeichnet.

Entsprechend des angegebenen Listings wird die gemeinsam benutzte (ENCLOSEd) Leitung, auf die von den einzelnen selbstdefinierten Komponenten aus zugegriffen wird, gerade in der Komponente `ProjektTest` vom Typ MODEL definiert.

Nachtrag des Korrektors: *Obwohl sich das Bankomat-Beispiel hervorragend für einen ausgiebigen „Stress-Test“ eignen würde und noch viele weitere interessante Betrachtungen darauf möglich wären, müssen diese Möglichkeiten auf Grund des bereits erwähnten Mangels an Zeit und Arbeitskräften entfallen. Der Informationsgehalt dieses Abschnitts beschränkt sich somit auf eine stichprobenartige Kontrolle der korrekten Umsetzung einzelner Bestandteile eines UML-Modells, das eine realitätsnahe und praxisgerechte Größe aufweist. Dass das vom betrachteten UML-Modell tatsächlich repräsentierte System niemals in dieser Form konzipiert werden würde, steht dabei außer Frage.*

4.5 Test der Fehlerbehandlung von HIPE

Wegen der Kürze der Zeit konnte kein Test vorgenommen werden, in dem alle kritischen Fehler und Warnungen, die HIPE erzeugen kann, wirklich überprüft wurden. Es wird stattdessen versucht, anhand einfacher Anwendungsfall- bzw. Aktivitätsdiagramme jeweils beispielhaft eine der möglichen Warnungen bzw. einen kritischen Fehler zu erzeugen, um die Reaktion von HIPE darauf darzustellen.

4.5.1 Test der Behandlung von Warnungen in HIPE

Warnungen treten auf, wenn HIPE auf Angaben stößt, die entweder unvollständig sind, aber automatisch durch Einsetzen von Default-Werten vervollständigt werden können, oder wenn bestimmte Gegebenheiten der Analyse, wie der vom Nutzer ausgewählte Löser, gewisse Anpassungen notwendig machen. Als Folge der Behandlung dieser Situationen kann das Verhalten des Modells danach von dem vom Nutzer intendierten Verhalten abweichen, aber eine Analyse ist weiterhin möglich.

Ein Beispiel für eine Warnung ist eine Meldung über die Anpassung eines Wertes, die vorgenommen wird, falls in HIPE ein analytischer Löser ausgewählt wird und der Wert nicht bestimmten Verteilungen folgt. Im dargestellten Anwendungsfalldiagramm ist dies an zwei Stellen der Fall, da zum einen am einzigen Akteur der Wert von `occurence` als 1 und der Wert von `responsetime` beim einzigen Anwendungsfall als 3 angegeben ist.

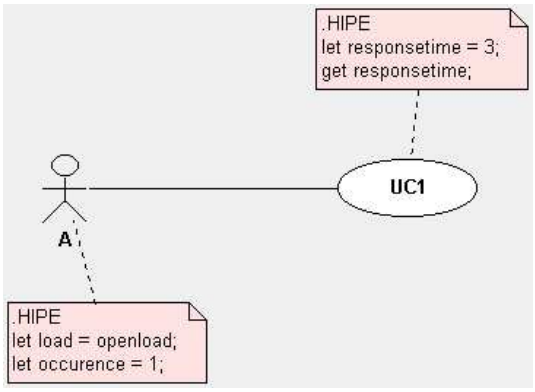

Abbildung 4.31: Anwendungsfalldiagramm zum Test der Fehlerbehandlung von HIPE

Erwartetes Verhalten
Es erscheint ein Dialogfeld, in dem die folgenden Warnmeldungen erscheinen, die auf die von HIT vorgenommene Anpassung hindeutet:

```
Angabe 3 bei Verteilung für Löser ANALYTICAL nicht zulässig,
setze cox(3.0,0.32)
```

Diese Warnung bezieht sich auf die Angabe der Antwortzeit des einzigen Anwendungsfalls. Mit der Anpassung wird der HIT-internen Einschränkung Rechnung getragen, dass der Umfang eines atomaren Aufrufs nur negativ exponential-, cox- oder coxg-verteilt sein darf. Die Anpassung mittels cox-Verteilung wurde im der Implementierung gewählt, weil sie am ehesten die Intention des Nutzers abbildet, eine Anfrage mit einer Ausführungsdauer von *genau* drei Zeiteinheiten zu darzustellen.

```
Angabe 3000.0 bei Verteilung für Löser ANALYTICAL nicht zulässig,
setze cox(3000.0,0.32)
```

Diese Angabe bezieht sich auf die Größe der durch den Anwendungsfall dargestellten Anfrage in Einheiten auf der Default-CPU, die 1000 Operationen pro Zeiteinheit ausführen kann. Insgesamt ergibt sich also ein Umfang der Anfrage von 3000 Einheiten, bzw. unter der Anpassung auf die cox-Verteilung von cox(3000.0,0.32).

```
Angabe 1 bei Zwischenankunftszeiten für Löser ANALYTICAL nicht zulässig,
setze negexp(1/1)
```

Entsprechend wird der Wert für die Zwischenankunftszeit von Prozessen des Typs `A` auf `negexp(1/1)` gesetzt, was angibt, dass die neue angepasste Zwischenankunftszeit negativ exponentialverteilt mit einem Erwartungswert von einer Zeiteinheit ist.

Nach Bestätigung der Meldung durch Anklicken des OK-Buttons wird die Analyse fortgesetzt.

Realisiertes Verhalten
Es erscheint das in der Abbildung 4.32 auf S. 177 dargestellte Dialogfeld.

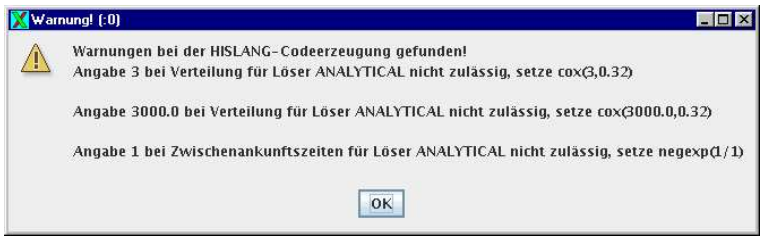

Abbildung 4.32: Von HIPE mit dem Testdiagramm erzeugte Warnmeldungen

Im Code werden die durch die Warnungen angemerkten Anpassungen entsprechend berücksichtigt:

```
-->001 %ANALYZER
-->002 %PARM=UPDATES
-->003 %END
-->004 TYPE HipeTest MODEL;
-->005  TYPE A SERVICE;
-->006    USE
-->007      SERVICE
-->008        UC (AMOUNT : REAL);
-->009    END USE;
-->010    BEGIN
-->011      BRANCH
-->012      PROB 0.999999: UC(cox(3000.0,0.32));
-->013      END BRANCH;
-->014  END TYPE A;
-->015  COMPONENT cexvdb : server (LET schedule := random,LET dispatch := equal(LET speed := 1000));
-->016  REFER A TO cexvdb EQUATING
-->017    A.UC WITH cexvdb.request;
-->018  END REFER;
-->019 BEGIN
-->020  CREATE 1 PROCESS A LIMIT 10 EVERY negexp(1/1);
-->021 END TYPE HipeTest;
```

4.5.2 Test der Behandlung kritischer Fehler in HIPE

Kritische Fehler treten auf, wenn wichtige Angaben fehlen oder unvollständig sind, aber nicht durch Einsetzen von Default-Werten vervollständigt werden können, oder von HIPE nicht entschieden werden kann, wie ein vorliegender Wert angepasst werden kann, um eine Analyse zu ermöglichen. Das Auftreten solcher Fehler führt dazu, dass das betrachtete Modell nicht

analysiert werden kann.

Die Betrachtung des in Abbildung 4.13 auf S. 151 dargestellten Aktivitätsdiagramms führt zu einem kritischen Fehler, falls in HIPE ein analytischer Löser gewählt wird.

Erwartetes Verhalten
Es erscheint ein Dialogfeld, das die folgende Fehlermeldung enthält:

```
CONCURRENT-Block kann mit Löser ANALYTICAL nicht ausgewertet werden
```

Es tritt ein kritischer Fehler auf, wenn versucht wird, einen Concurrent-Block mit einem analytischen Löser auszuwerten. Da dieser Vorgang von HIT nicht unterstützt wird, bricht HIPE die Umsetzung ab, falls eine solche Situation auftritt. Nach Bestätigung des Dialogfeldes findet entsprechend keine Erhebung von Ergebnissen mittels HIT statt.

Realisiertes Verhalten
Das in Abbildung 4.33 auf S. 178 dargestellte Dialogfeld erscheint. Es wird keine HIT-Analyse durchgeführt.

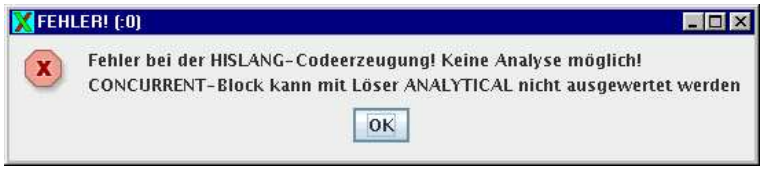

Abbildung 4.33: Von HIPE mit dem Testdiagramm erzeugte kritische Fehlermeldung

5 Benutzungshandbuch

Dieses Benutzungshandbuch erklärt die Verwendung des von der PG459 entwickelten Leistungsbewertungswerkzeug HIPE (Hierarchical Models in Performance Engineering).

5.1 Aufgabe des Programmsystems

Das Werkzeug HIPE dient zur Leistungsbewertung von Softwaremodellen, die in UML 2.0 spezifiziert sind. HIPE ist in den Bereich des „Software Performance Engineering“ (SPE) einzuordnen und hat die Aufgabe, UML-Modelle in eine Eingabe für das Leistungsbewertungswerkzeug HIT, welches am Lehrstuhl Informatik IV der Universität Dortmund entwickelt wurde, zu übersetzen und darauf eine Leistungsbewertung durchzuführen. Daher wird ein Einsatz im Umfeld des Softwaredesigns bzw. der Softwareerstellung als sinnvoll erachtet. Denkbar ist auch eine nachträgliche Analyse von Modellen bereits realisierter Software, um nicht erkannte Performance-Engpässe aufzuspüren. HIPE geht im wesentlichen in folgenden Schritten vor:

- **Eingabe:**
  Der Entwickler kann die vorhandenen Softwareentwürfe durch das UML-Tool MetaMill[1] als XMI-Datei exportieren und in einem geeigneten Verzeichnis ablegen. Diese XMI-Datei wird von HIPE importiert und für die weitere Verarbeitung vorbereitet.

  Für den Einsatz von HIPE werden Anwendungs-, Aktivitäts-, Klassen-, und Verteilungsdiagramme benötigt. Die Eingaben für HIPE zu einem Element des UML-Diagramms erfolgen als textuelle Angaben auf einem Notizzettel (in MetaMill als *Note* bezeichnet), der über eine Ankerkante (in MetaMill *Anchor* genannt) mit dem jeweiligen Element verbunden wird. Eine genauere Beschreibung der Syntax der von HIPE benötigten Eingaben ist im Kapitel 1 (HIPE-Paradigma) zu finden. Das Fehlen einiger Diagramme führt zu einer Fehlermeldung, in der ersichtlich ist, welche Diagramme für eine sinnvolle Bewertung noch benötigt werden.
- **Analyse:**
  Die Leistungsbewertung mit HIPE besteht in der Durchführung der „weichen“ und „harten Analyse“. Der erste Schritt, die „weiche Analyse“, ist alleinige Aufgabe von HIPE. Der zweite Schritt, die „harte Analyse“, wird mit Hilfe von HIPE lediglich vorbereitet, die eigentliche Analyse führt das externe Tool HIT durch.
- **Bewertung und Ausgabe:**
  Die Ergebnisse werden textuell innerhalb der ursprünglichen UML-Diagramme

[1] Da für das Speichern von UML-Diagrammen im XMI-Format keine einheitliche Standards gelten, gibt es keine Garantie, dass HIPE korrekt läuft und korrekte Ausgaben erzeugt, wenn XMI-Dateien eingegeben werden, die mit anderen UML-Tools erzeugt wurden.

dargestellt. Außerdem werden die Ergebnisse nach der Analyse in Form von Diagrammen und Tabellen dargestellt.

HIPE bietet die Möglichkeit, Experimentreihen durchzuführen, das heißt, der Entwickler kann Parameter, wie beispielsweise die Prozessorgeschwindigkeit einer Maschine oder die Größe einzelner Lasten, variieren lassen, um dadurch festzustellen, welche Parameterkonfiguration für die gegebenen Softwareentwürfe die besten Ergebnisse liefert oder ob Flaschenhälse vorliegen. Die grafische Ausgabe der Ergebnisse wurde im Hinblick auf die Auswertung der Experimentserien entworfen.

HIPE erlaubt eine anschauliche Darstellung der Ergebnisse nach der Leistungsanalyse, so dass die Möglichkeit besteht, verschiedene Entwurfsvarianten miteinander zu vergleichen. Anschließend kann der Entwickler entscheiden, welche Entwurfsvariante die besten Resultate liefert, und kann auf Grundlage eines optimierten Software-Entwurfs die Implementierungsphase initiieren.

5.1.1 Leseanleitung

Dieses Handbuch ist gegliedert in sieben Unterabschnitte.

- Dieser erste Abschnitt 5.1 bildet die „Aufgabe des Programmsystems“. Es wurde bereits ein grober Überblick über die Möglichkeiten von HIPE und eine kurze Beschreibung des Programm-Konzeptes gegeben. Die vorliegende Übersicht über den Rest des Handbuches bildet den Abschluss dieses Abschnitts.

- Der zweite Abschnitt, „Zielgruppe“ (Kapitel 5.2, Seite 181), befasst sich mit der erwarteten Benutzergruppe und welche Vorkenntnisse HIPE voraussetzt.

- Der dritte Abschnitt befasst sich mit der „Programmbeschreibung“ (Kapitel 5.3, Seite 182). Hier wird ein weiterer, detaillierter Überblick über die Fähigkeiten dieses Softwarepaketes sowie eine Beschreibung seiner Komponenten gegeben.

- Der vierte Abschnitt erläutert, welche „ergonomischen Eigenschaften“ (Kapitel 5.4, Seite 195) der Programmgestaltung zugrunde gelegt und aus welche Gründen die verwendete graphische Schnittstelle ausgewählt wurde.

- Der fünfte Abschnitt enthält eine „Beispielanwendung“ (Kapitel 5.5, Seite 195). Dieser Abschnitt dient dazu, den Anwender mit dem Umgang des Systems anhand eines Beispiels vertraut zu machen.

- Der sechste Abschnitt besteht aus möglichen „Hilfen/Fehlermeldungen“ (Kapitel 5.6, Seite 204) und den dazugehörigen Problemlösungen.

- Der siebte Abschnitt enthält alle „Besonderheiten“ (Kapitel 5.7, Seite 207), die nicht zum Inhalt eines der anderen Kapitel zugeordnet werden konnten.

Zu allen Beschreibungen und Beispielen existieren Abbildungen, die es dem Anwender erleichtern, das Beschriebene nachzuvollziehen. Bei der Ausführung von HIPE werden dieselben Graphiken als Bildschirmausgabe zu sehen sein.

5.2 Zielgruppe

Das Programm HIPE richtet sich an qualifiziertes Fachpersonal aus dem Bereich Softwaredesign. Der sichere Umgang mit Rechner wird deshalb erwartet. Es werden außerdem fundierte Kenntnisse im Bereich Softwaremodellierung, bzw. Softwaredesign vorausgesetzt.

5.2.1 Installation

Der produktive Einsatz des Tools erfordert eine bestimmte Grundkonfiguration des Systems, auf dem es ausgeführt werden soll. Die Software- und Hardware-Aspekte dieser Konfiguration werden im Kapitel 5.7.1 („Systemanforderungen“, Seite 207) beschrieben. Das Programm liegt als JAR-Archiv mit dem Namen `HIPE_fat.jar` vor. Dieses Archiv enthält den Maschinencode von HIPE sowie sämtliche Pakete der verwendeten externen Werkzeuge (JDOM und JCHarts) und ist direkt ausführbar[2]. Falls aufgrund einer Fehlkonfiguration des Systems das direkte Ausführen des Archivs nicht funktionieren sollte, kann HIPE über die DOS-Box von Windows („Start“ -> „Ausführen“ -> „cmd“ eintippen) nach einem Wechsel in den Ordner, wo `HIPE_fat.jar` liegt, mittels

```
java -jar HIPE_fat.jar
```

gestartet werden. Das Server-Modul von HIPE muss, wie in Abschnitt 5.7.1 beschrieben, auf einer gesonderten Maschine gestartet werden. Die dafür notwendigen Schritte sind unter dem Punkt 5.3.3.11 (Seite 193) zu finden.

Zu beachten ist, dass alle Bilder (`Elem_JPanel_Hintergrund.jpg`, `Error.gif`, `Splash_intro.gif` und `Start.gif` im Verzeichnis `hipe/gui`) und die Datei `properties.pro` (im Verzeichnis `hipe/concept`) für eine vollständige Funktionalität aus dem `HIPE_fat.jar`-Archiv unter Verwendung der Pfadangaben entpackt werden sollten (`HIPE_fat.jar` ist jedoch auch ohne diese Maßnahme lauffähig). Eine vollständige Installation sieht somit wie folgt aus:

```
(root)
  |
  +-- hipe
  |    |
  |    +-- concept
  |    |      |
  |    |      \-- properties.pro
  |    |
  |    \-- gui
  |         |
  |         +-- Elem_JPanel_Hintergrund.jpg
  |         |
  |         +-- Error.gif
  |         |
```

[2] Man beachte, dass die Pakete der externen Werkzeuge nicht „eingekauft“ wurden und somit nicht mit HIPE weitergegeben werden dürfen! Die allgemeinen Copyright-Bestimmungen sind zu beachten! Lediglich zur Ausführung innerhalb eines Forschungs- und Lehrauftrages ist es gestattet, das -Archiv zu verwenden. Für Anwendungen außerhalb eines solchen Kontextes müssen die externen Pakete gesondert akquiriert und lizensiert werden!

```
|          +-- Splash_intro.gif
|          |
|          \-- Start.gif
|
\-- HIPE_fat.jar
```

Zur einfacheren Installation wurde diese Ordnerstruktur mitsamt aller Dateien (inklusive des `HIPE_fat.jar`-Archivs) in Form des ZIP-Archivs „HIPE.ZIP“ archiviert. Zur Erzeugung der oben beschriebenen vollständigen Installation genügt es somit, lediglich dieses Archiv unter Verwendung der Option „Pfadangaben verwenden“ (oder so ähnlich, je nach verwendetem ZIP-Tool) zu entpacken.

5.3 Programmbeschreibung

Im folgenden Abschnitt werden die verschiedenen Programmfunktionen detailliert beschrieben und Anleitungen zu den unterschiedlichen Programmfunktionen gegeben.

5.3.1 Abbildung der Dialogstruktur

Die Abbildungen 5.1 auf S. 183 und 5.2 auf S. 184 zeigen das Diagramm der Dialogstruktur für HIPE. Die äußeren Zustandsfelder beschreiben die durch Klick auf den jeweiligen Menü- und Untermenüpunkt erreichbaren Funktionen des Programms. Die weiteren Zustände beschreiben die Aktionen auf den Übergängen zwischen den einzelnen Dialogen.

5.3.2 Erläuterungen

- Das Hauptfenster ist in vier Bereiche aufgeteilt: ganz oben die Hauptmenüleiste, über die man mit einem einfachen Klick auf alle Bereiche und Funktionen zugreifen kann, links ein Bereich, in dem die Projektdatei als Baum dargestellt wird, rechts eine Fläche, in der alle Dialogfenster erscheinen und unten das Textfeld, in dem die Ausgaben des Programms aufgelistet werden, sowie die Statuszeile. Die unterschiedlichen Dialogfenster und Funktionen werden nachfolgend noch detaillierter vorgestellt.

- Ein HIPE-Projekt besteht aus einer einzigen Datei, die die Endung `.hpr` besitzt. Die Projektdatei enthält alle für eine Analyse nötigen Informationen. Um ein HIPE-Projekt zu erzeugen, wird eine XMI-Datei benötigt. Diese enthält die UML-Diagramme des Softwareentwurfes, der analysiert werden soll. Während der Erzeugung eines neuen HIPE-Projektes werden alle nötigen Informationen aus der XMI-Datei ausgelesen und in die HIPE-Projektdatei geschrieben. Nach einer Analyse werden die Ergebnisse in die Projektdatei zurückgeschrieben und über die Baumansicht der Projektdatei dem Benutzer zugänglich gemacht. Bei dieser besonderen Form der Dateiansicht werden die einzelnen Bestandteile der Projektdatei in einem Baum dargestellt.

- Man beendet das Programm, indem man über den Menüpunkt „Projekt“ an der Hauptmenüleiste den Untermenüpunkt „Beenden“ mit einem Klick anwählt.

- Neu erzeugte Projekte sind nicht gesichert, solange sie nicht explizit gespeichert wurden. Ein neu erzeugtes Projekt sollte erst gespeichert werden, bevor weiter daran gearbeitet

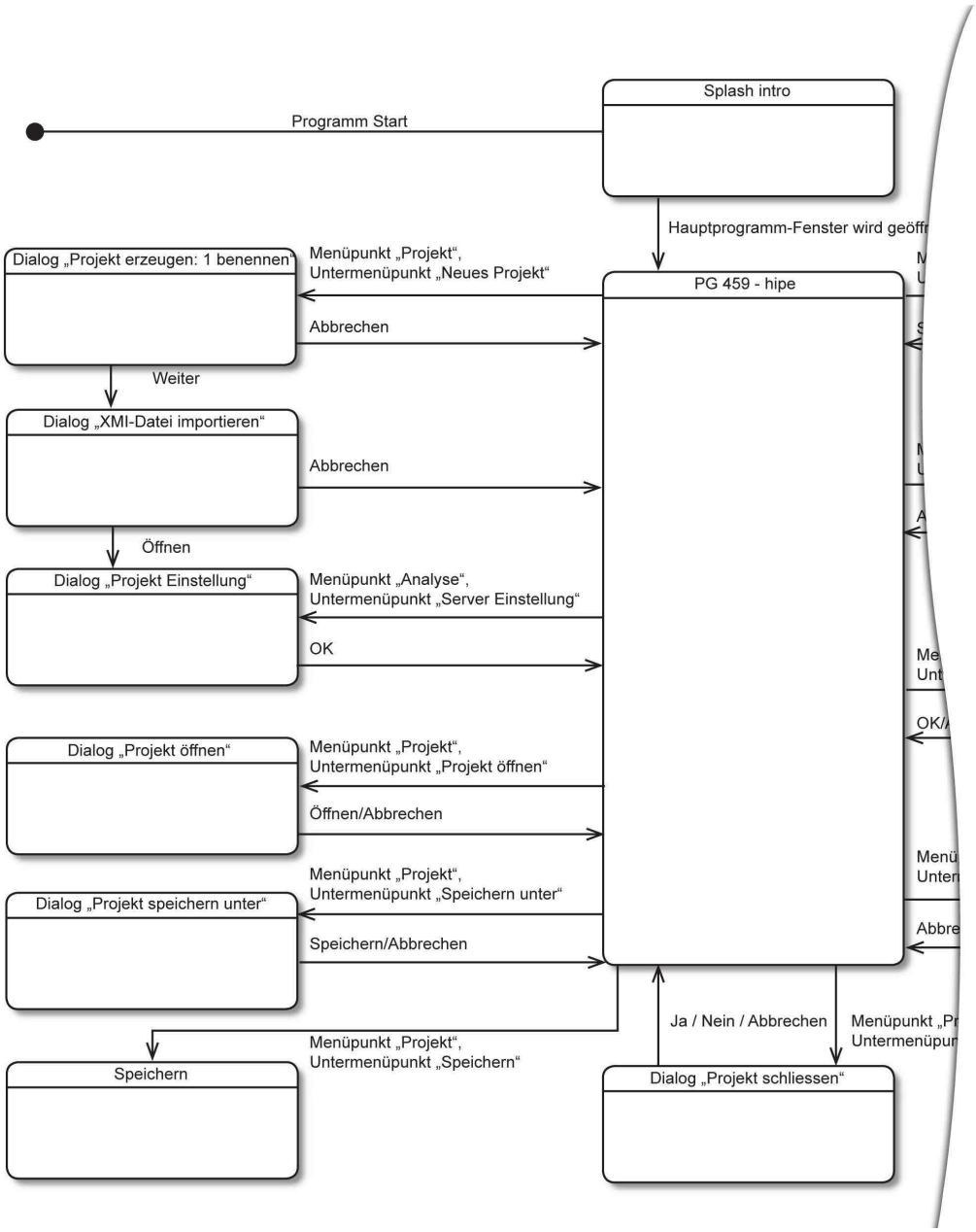

Abbildung 5.1: Erster Teil des Dialogstrukturdiagramms für HIPE. Zweiter Teil siehe Abbildung 5.2 auf S. 184

wird. Andernfalls ist die Gefahr groß, dass alle vorgenommenen Analysen und Bewertungen verloren gehen. Es wird ausserdem empfohlen, in regelmäßigen Abständen manuell zu sichern.

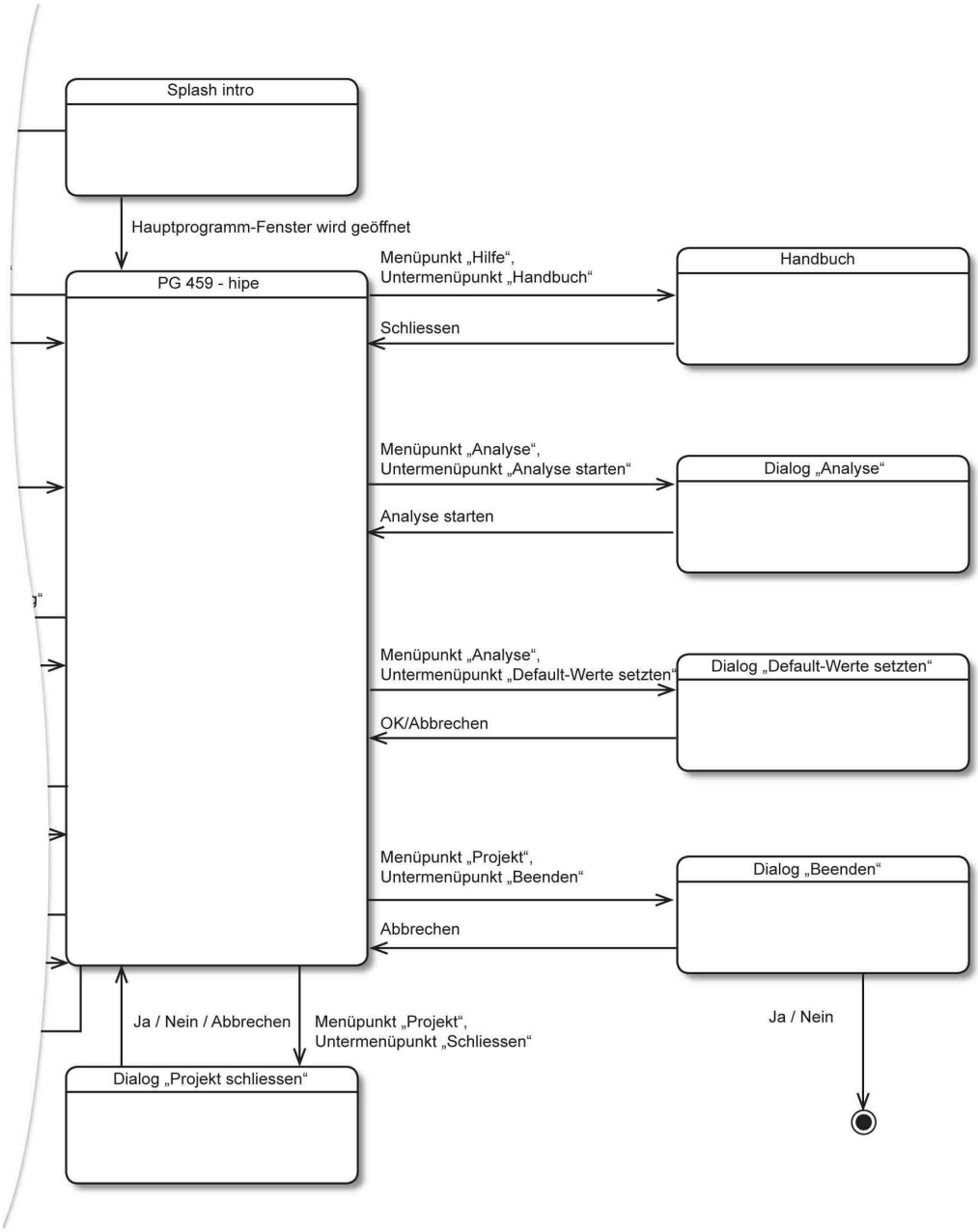

Abbildung 5.2: Zweiter Teil des Dialogstrukturdiagramms für HIPE. Erster Teil siehe Abbildung 5.1 auf S. 183

5.3.3 HIPE-Hauptfenster

Über das Hauptfenster von HIPE (siehe Abbildung 5.3 auf S. 185) erhält man Zugriff auf die drei Hauptmenüpunkte „Projekt“, „Analyse“ und „Hilfe“. Die Menüliste von „Projekt“ (siehe Abbildung 5.4 auf S. 186) enthält die Untermenüs „Neues Projekt“, „Projekt öffnen“, „Schliessen“ „Speichern“, „Speichern unter“ und „Beenden“. Die Menüliste „Analyse“ enthält die Untermenüpunkte „Analyse starten“, „Default-Werte setzen“ und „Server Einstellung“.

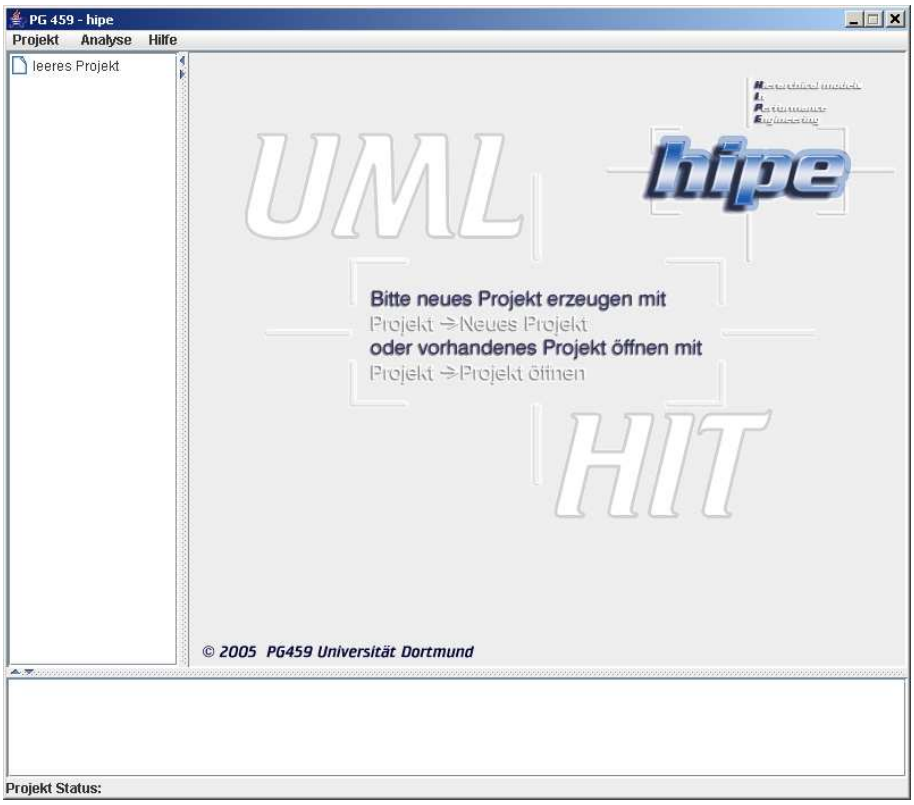

Abbildung 5.3: Das Hauptfenster von HIPE

Die Menüliste „Handbuch“ enthält lediglich die Auswahl „Handbuch“. Im linken Feld des Hauptfensters wird die Projektstruktur dargestellt, die nach dem Programm-Start den Eintrag „leeres Projekt“ enthält. Wird ein neues Projekt erzeugt, oder ein schon vorhandenes Projekt geöffnet, erscheint in diesem Feld die detaillierte Struktur des Projektes in Form eines Verzeichnisbaums. In diesen Verzeichnisbaum werden die einzelnen UML-Modelle als Zweige dargestellt. Außerdem werden die Ergebnisse der weichen Analyse für jedes Modell angegeben. Das rechte Feld ist anfangs leer. Erst wenn man einen Menüpunkt gewählt hat, erscheinen im rechten Feld weitere Funktionen und Dialoge. Die zwei Felder werden durch eine Slidebar (mit variabler Position) getrennt.

5.3.3.1 Neues Projekt erzeugen

Ein neues Projekt erzeugt man mit einem Klick auf den Hauptmenüleisten-Eintrag „Projekt“ und anschließender Auswahl des Untermenüpunkts „Neues Projekt“.
Im rechten Feld des Hauptfensters öffnet sich der Dialog „Projekt erzeugen: 1. benennen“ (siehe Abbildung 5.5 auf S. 186), in dem man als erstes das neue Projekt benennen und das Verzeichnis für das neue Projekt wählen kann. Den Namen gibt man im editierbaren Textfeld „Dateiname“ an, nachdem man mit einen Klick im Textfeld den Cursor gesetzt hat. Als nächstes wählt man im darunter befindlichen Feld das Projektverzeichnis. Mit „Suchen in“

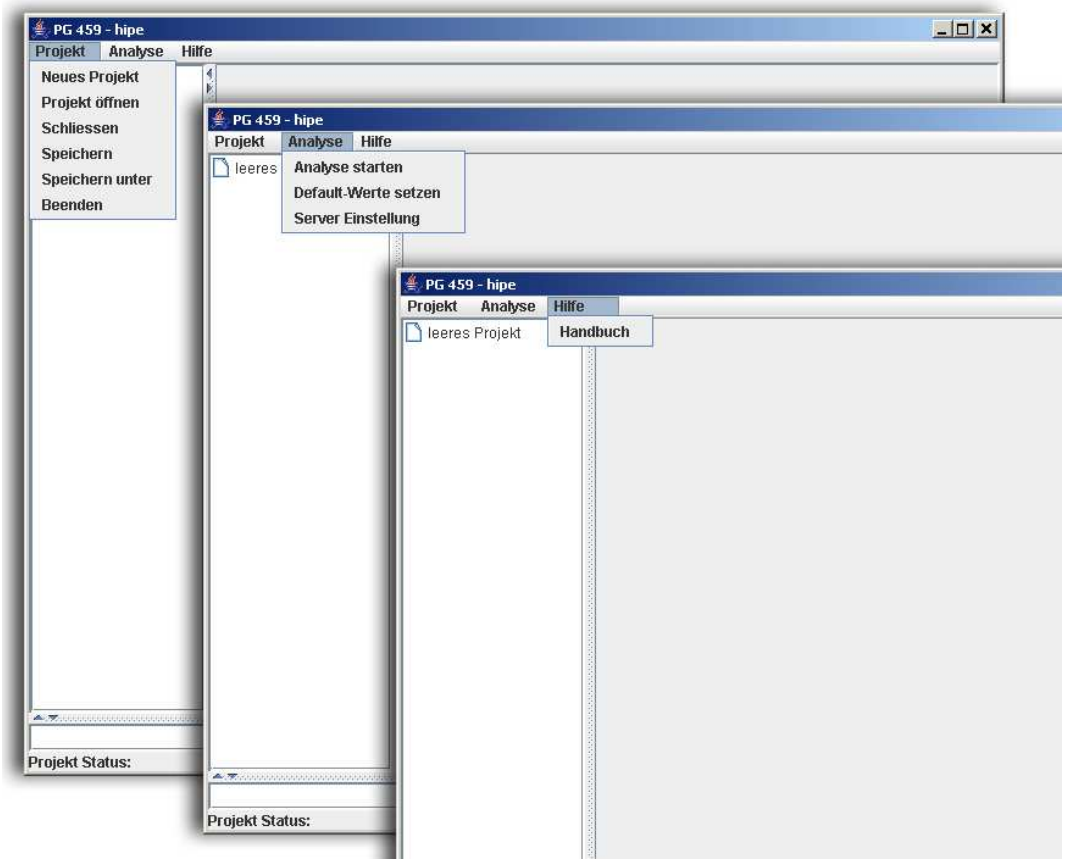

Abbildung 5.4: Die Hauptmenüleiste mit den Einträgen „Projekt“, „Analyse“ und „Hilfe“

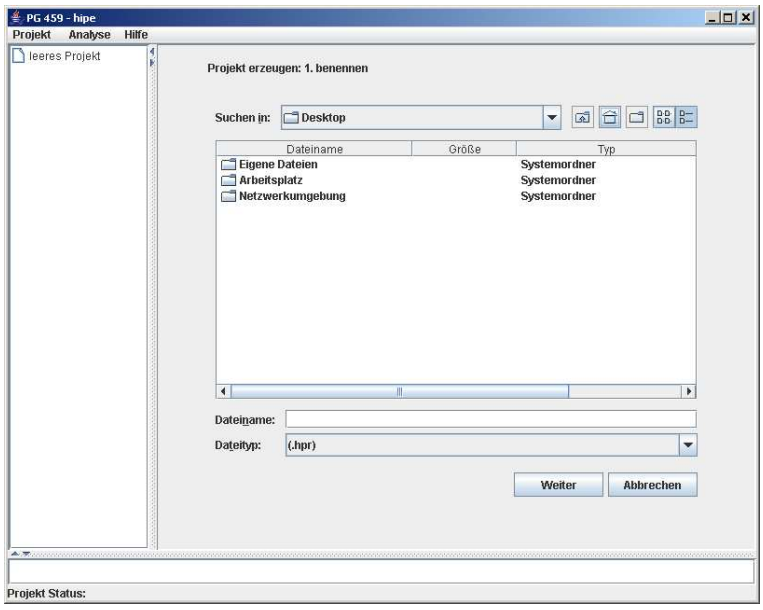

Abbildung 5.5: Erzeugen eines neuen Projektes in HIPE; Erster Schritt: Projekt benennen

wählt man das gewünschte Verzeichnis, in dem darunter liegenden Feld wird der Inhalt des Verzeichnis angezeigt. Wenn das gewünschte Verzeichnis gewählt ist, kann man mit *Weiter* zum nächsten Schritt „XMI-importieren“ übergehen, oder den Vorgang mit einen Klick auf „Abbrechen“ beenden und zum Hauptfenster zurückkehren. Mit einen Klick auf „Weiter“ erscheint das nächste Dialogfenster „XMI-Datei importieren“ (siehe Abbildung 5.6 auf S. 187),

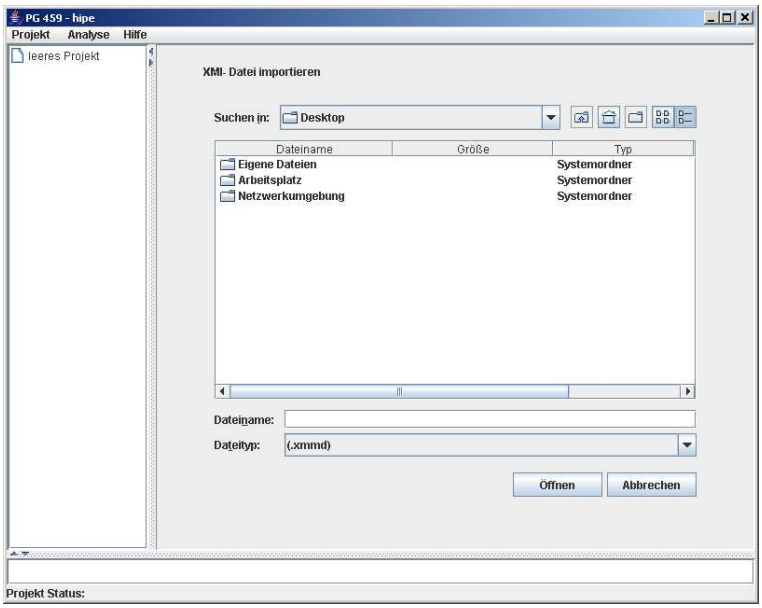

Abbildung 5.6: Erzeugen eines neuen Projektes in HIPE: 2.Schritt Importieren der XMI-Datei

in dem der Benutzer die „XMI-Datei“ mit der Endung `.xmmd` angeben muss. Man schließt den Vorgang mit einen Klick auf „OK“ ab. Anschließend öffnet sich der Dialog „Server Einstellung“ (siehe Abbildung 5.7 auf S. 187), in dem man in den editierbaren Textfeldern mit den

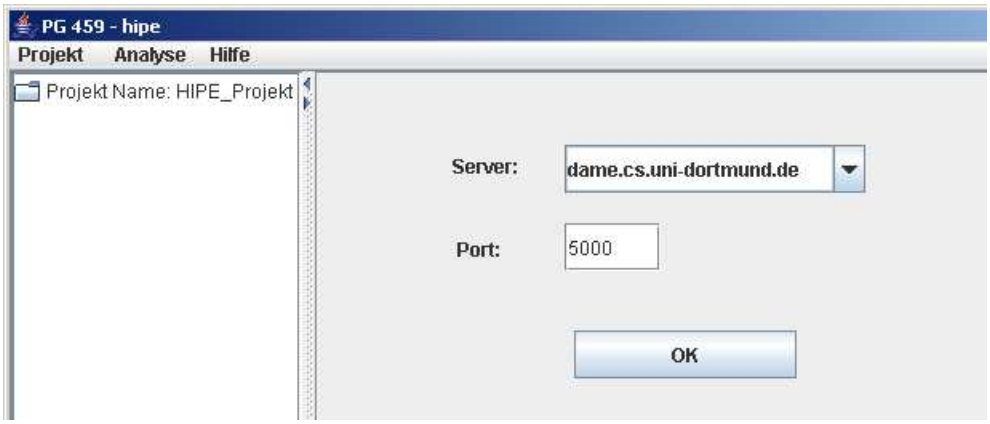

Abbildung 5.7: Menüpunkt „Server Einstellung“

Beschriftungen „Server“ und „Port“ die nötigen Eingaben machen muss. Den Vorgang schließt

man mit einen Klick auf „OK“ und kehrt zum Hauptfenster zurück.

5.3.3.2 Projekt öffnen

Um ein Projekt zu öffnen, wählt man über den Menüpunkt „Projekt“ der Hauptmenüleiste den Untermenüpunkt „Projekt öffnen“ mit einem Klick. Im rechten Feld des Hauptfensters öffnet sich der Dialog „Projekt öffnen“ (siehe Abbildung 5.8 auf S. 188).“ In dem Dialog befindet

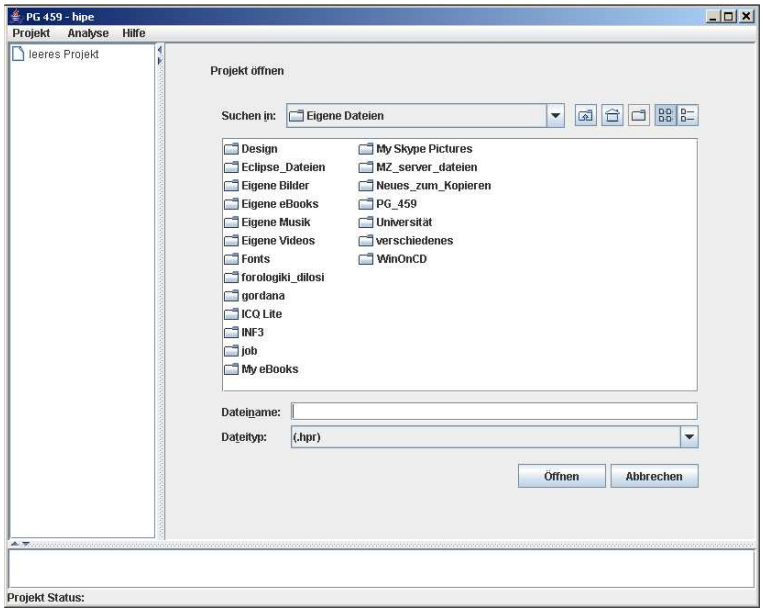

Abbildung 5.8: Dialog „Projekt öffnen

sich ein Datei-Explorer, in dem man das Projektverzeichnis und die gewünschte Projektdatei wählen kann. Nach dem man das gewünschte Projekt gewählt hat, bestätigt man die Auswahl mit einem Klick auf „Öffnen“.

5.3.3.3 Projekt schließen

Ein Projekt wird geschlossen, indem man über den Menüpunkt „Projekt“ der Hauptmenüleiste den Untermenüpunkt „Schließen“ mit einem Klick auswählt. Im rechten Feld des Hauptfensters öffnet sich der Dialog „Änderungen am Projekt speichern?“ (siehe Abbildung 5.9 auf S. 189). Wählt man hier „Ja“, werden die Änderungen innerhalb des Projekts abgespeichert und das Projekt wird geschlossen. Wählt man hier „Nein“, wird das Projekt ohne Übernahme der Änderungen geschlossen. Wählt man hier „Abbrechen“, so wird der Vorgang abgebrochen.

5.3.3.4 Projekt speichern

Ein Projekt wird gespeichert, indem man über den Menüpunkt „Projekt“ der Hauptmenüleiste den Untermenüpunkt „Speichern“ mit einem Klick auswählt.

Abbildung 5.9: Menüpunkt Dialog „Projekt schließen“ und „Beenden“

5.3.3.5 Projekt speichern unter

Ein Projekt wird gespeichert, indem man über den Menüpunkt „Projekt“ der Hauptmenüleiste den Untermenüpunkt „Speichern unter“ mit einem Klick auswählt. Im rechten Feld des Hauptfensters öffnet sich der Dialog „Projekt speichern unter“ (siehe Abbildung 5.10 auf S. 189), in dem man das Verzeichnis, in dem das Projekt gespeichert werden soll, angeben muß.

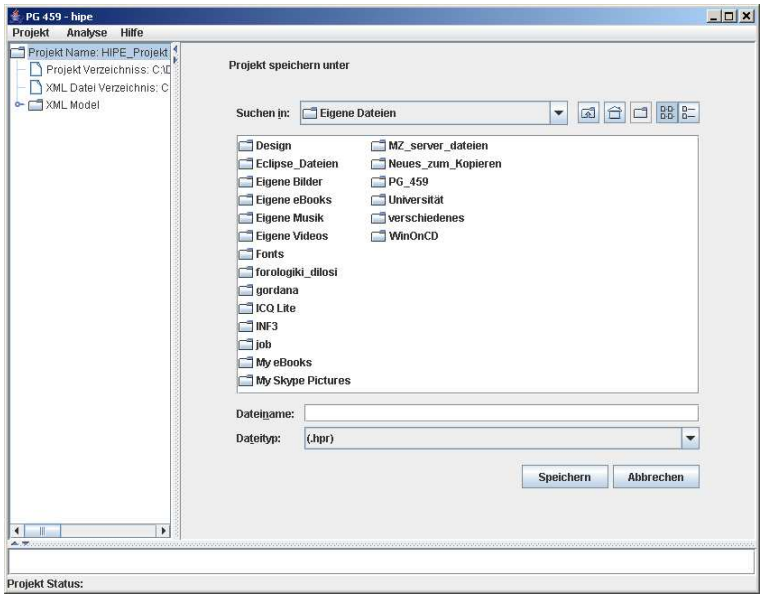

Abbildung 5.10: Menüpunkt „Projekt speichern unter“

5.3.3.6 Beenden

Um HIPE zu beenden wählt man in der Hauptmenüleiste mit einen Klick den Menüpunkt „Projekt“ und anschließend den Untermenüpunkt „Beenden“ aus. Im rechten Feld des Hauptfensters öffnet sich der Dialog „Änderungen am Projekt speichern?“ (siehe Abbildung 5.9 auf S. 189), in den man die Wahl treffen kann zwischen „Ja“, um mögliche Änderungen an dem

Projekt zu speichern, „Nein“, um die Änderungen zu verwerfen und „Abbrechen“, um den Vorgang abzubrechen.

5.3.3.7 Analyse starten

Um ein Projekt zu analysieren, wählt man in der Hauptmenüleiste mit einen Klick den Menüpunkt „Analyse“ und anschließend den Untermenüpunkt „Analyse starten“ aus. Im rechten Feld des Hauptfensters öffnet sich der Dialog „Analyse Starten“. Der Dialog ist in zwei Felder geteilt. Das obere Feld mit der Überschrift „Solver“ enthält einen Pull-Down-Menü mit dem man zwischen den Lösern SIMULATIVE, ANALYTICAL_DOQ4, ANALYTICAL_LIN2 und ANALYTICAL_MARKOV wählen kann (siehe Abbildung 5.11 auf S. 190).

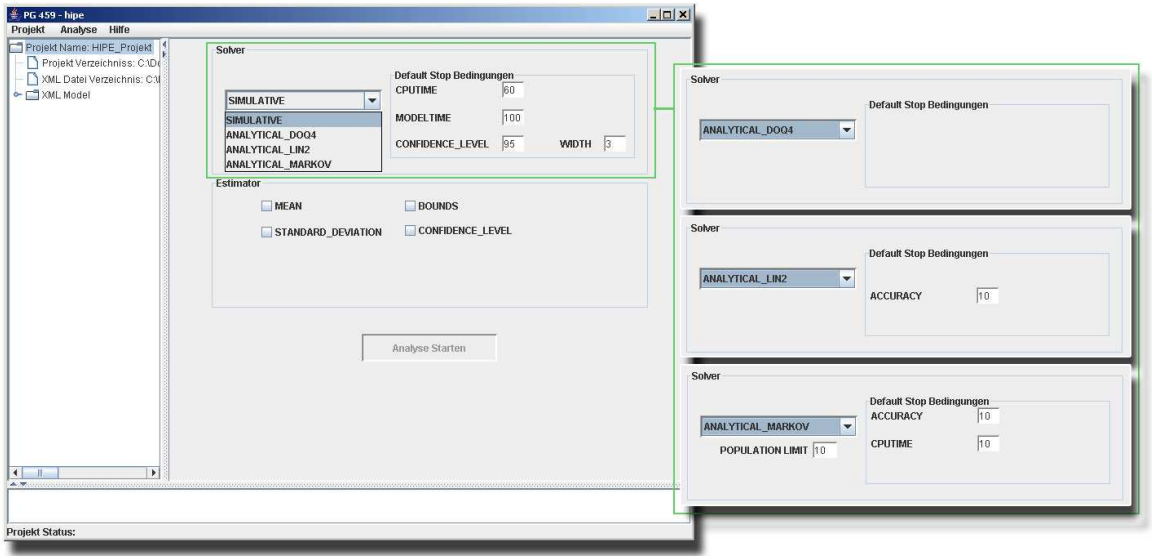

Abbildung 5.11: Dialogfenster der Analyse: Auswahl der Löser

Je nach Art des Lösers gilt es, unterschiedliche Parameter zu spezifizieren. Für SIMULATIVE sind dies CPUTIME und MODELTIME, ANALYTICAL_LIN2 benötigt den Parameter ACCURACY, für ANALYTICAL_MARKOV sind es ACCURACY, POPULATION LIMIT sowie CPUTIME und ANALYTICAL_DOQ4 enthält keine weiteren Parameter.
Im unteren Feld des Dialogs befindet sich das Feld „Estimator“ in dem man eine Auswahl treffen kann zwischen MEAN, STANDARD_DEVIATION, BOUNDS und CONFIDENCE_LEVEL.
Wenn man seine Auswahl des Löser und der Estimator getroffen hat, klickt man auf „Analyse Starten“[3].

Bewertung
Nachdem man den Button „Analyse starten“ gedrückt hat, passiert folgendes: Als erstes wird die Analyse bezüglich der weichen Leistungsmaße durchgeführt. Dieser Vorgang findet innerhalb von HIPE statt. Nachdem die weiche Analyse abgeschlossen ist, werden die Ergebnisse

[3]Um ein Abbrechen der Analyse zu verhindern wurde der Button „Abbrechen“ bewusst ausgelassen. In Ausnahmefällen kann man die Analyse dennoch abbrechen, in dem man einmal in das Ausgabefenster des Hauptfensters klickt und anschließend die Taste F9 drückt

in eine für das Evaluationstool HIT geeignete Eingabe umgewandelt und mit Hilfe des HIPE-Servers nach HIT übergeben. In diesem Vorgang initiiert das HIPE-Server-Modul die Analyse bezüglich der harten Leistungsmaße mittels HIT. Im Anschluß werden die Ergebnisse der Analyse vom Server an HIPE zurückgeliefert. Wenn die Analyse fehlerfrei durchgeführt wurde, erscheint in der Statusleiste des Hauptfensters die Meldung „Ergebnisse sind verfügbar". Die Ergebnisse sind dann über die Baumansicht abrufbar. Mit einem Klick auf die Zweige der einzelnen UML-Diagramme ruft man die jeweiligen Ergebnisse ab. Die Visualisierung der Ergebnisse erfolgt in Form einer grafischen Darstellung als zweidimensionale Säulendiagramme, die im rechten Feld des Hauptfensters aufgebaut werden. Eine detaillierte Beschreibung der einzelnen Diagrammtypen wird im Kapitel 1.2.7 (Seite 64) des vorliegenden Endberichts dargestellt. Die graphische Darstellung der Resultate der weichen und harten Analyse unterscheiden sich dadurch, dass die Säulen der Werte der weichen Analyse horizontal abgebildet werden, während die Säulen der einzelnen Ergebnisse der harten Analyse vertikal aufgebaut sind.

5.3.3.8 Baumansicht

Im Folgenden wird die Darstellung und Funktionsweise der Baumansicht erläutert. Die Baumansicht der Projektdatei erlaubt die Anzeige der Ergebnisse der Analyse. Im Baum werden die Bestandteile der Projektdatei als Knoten und Blätter dargestellt. Man unterscheidet dabei Knoten, die weitere Unterelemente haben können, und Blätter, die keine Unterelemente haben. Blätter sind die letzten Elemente in einem Ast.
Der Benutzer kann über den Baum direkt die einzelnen Evaluationsobjekte auswählen, analysieren und bewerten (siehe Abbildung 5.12 auf S. 192). Der Projektname bildet die Baumwurzel und wird mit einen Ordner-Symbol und dem Projektnamen als Label dargestellt. Mit einen Doppelklick auf den Projektordner erscheint die nächste Hierarchie-Ebene. Unter dem Projekt-Ordner werden das Projektverzeichnis und das Verzeichnis der XMI-Datei angegeben. Die XMI-Datei wird genau darunter als Ordner mit dem Label „XMI-Model" dargestellt.
Mit einen Doppelklick auf den abgebildete Ordner kann man vorhandene Zweige ein- und ausblenden. So kann man mit einen Doppelklick alle in der XMI-Datei vorhandenen UML-Diagramme sehen. Sie sind nach der Art des UML-Diagramms sortiert. Unterhalb eines jeden UML-Diagramms wird mit einem Doppelklick die Liste der erzeugten Evaluations-Objekte des betreffenden Diagramms angezeigt.
Mit einen Doppelklick auf den Ordner mit dem Label „Klassendiagramme" erscheinen alle in der XMI-Datei vorhandenen UML-Klassendiagramme und die zum jeweiligen Klassendiagramm zugehörigen Klassen und die weichen Leistungsmaße des gesamten Klassendiagramms (siehe Abbildung 5.13 auf S. 193). Mit einem Klick auf das Griff-Symbol vor einer Klasse werden die weichen Leistungmaße der Klasse angezeigt.

5.3.3.9 Default-Werte setzen

Bevor man ein Projekt analysiert, können diesbezüglich noch Default-Werte gesetzt werden. Wenn keine Änderungen vorgenommen werden, führt HIPE die Analyse mit den programmspezifischen Default-Werten aus. Der Anwender kann diese Werte nach seinen Wünschen modifizieren. Um die Default-Werte zu ändern, wählt man in der Hauptmenüleiste mit einem Klick den Menüpunkt „Analyse" und anschließend den Untermenüpunkt „Default Werte setzen" (siehe Abbildung 5.14 auf S. 194) aus. Es öffnet sich der Dialog „Default-Werte setzen". Im

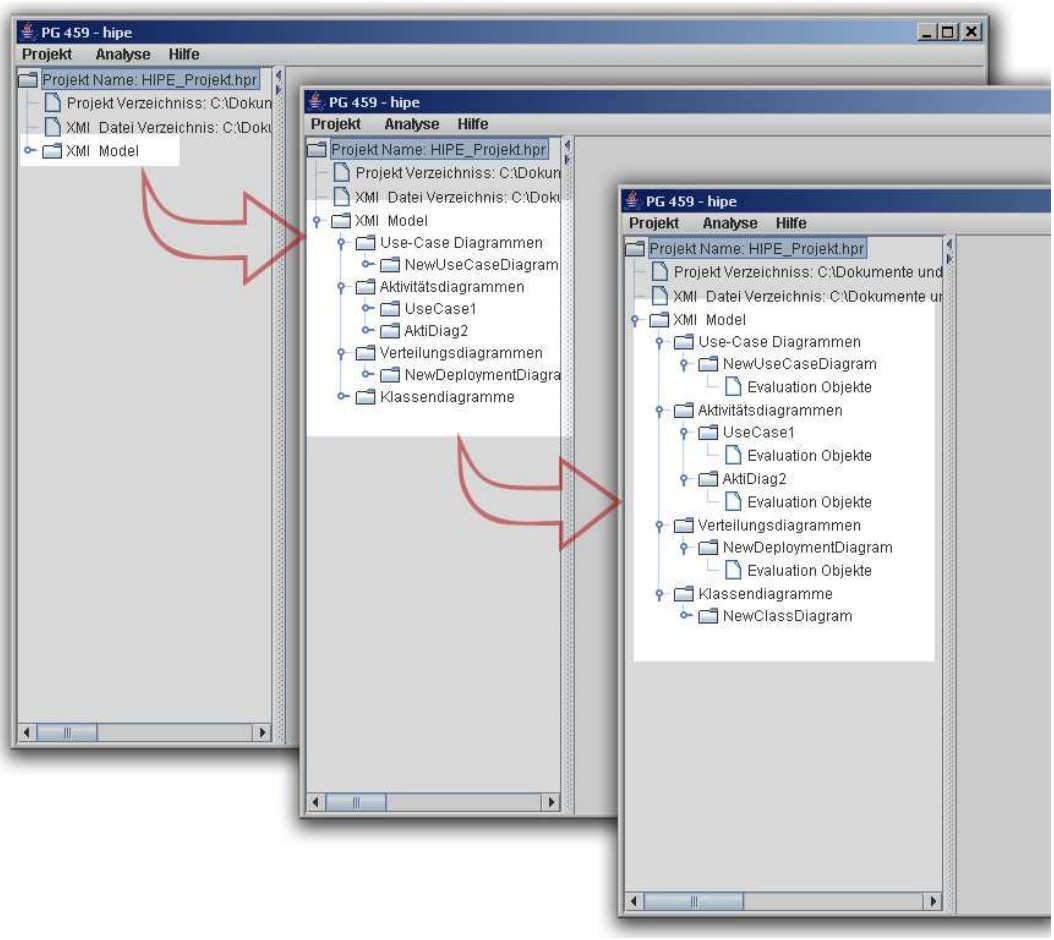

Abbildung 5.12: Die Baumansicht: Abbildung der in der XMI-Datei enthaltenen UML-Diagrammen und der Evaluationsobjekten aus den einzelnen UML-Diagrammen im Projektdatei-Baum

oberen Teil des Dialoges befinden sich die zwei Pull-Down-Menüs „Dispatch“ und „Schedule“. Bei „Dispatch“ kann die Wahl zwischen DISPATCH_EQUAL und DISPATCH_SHARED, bei „Schedule“ zwischen FCFS und IMMEDIATE getroffen werden. Darunter gibt es die Eingabefelder „ExtDelay“, „Population“, „Resptime“ und „Speed“, indem man ganzzahlige Einträge vornehmen kann. Die genauen Einstellungsmöglichkeiten werden im Kapitel 5.5 („Beispielanwendung“, Seite 195) genauer beschrieben.

5.3.3.10 Server Einstellung

Die harte Analyse wird mit Hilfe von HIPE lediglich vorbereitet, die eigentliche Analyse führt das externe Tool HIT durch. Um die harte Analyse durchführen zu können, müssen folglich die Daten von HIPE auf eine *Solaris*-Maschine übertragen werden, auf der HIT installiert ist. Über den Dialog „Server Einstellung“ gibt man den Server und den Port an, damit die Datenübertragung erfolgen kann. Der Dialog öffnet sich direkt nach der Erzeugung eines neuen Projekts.

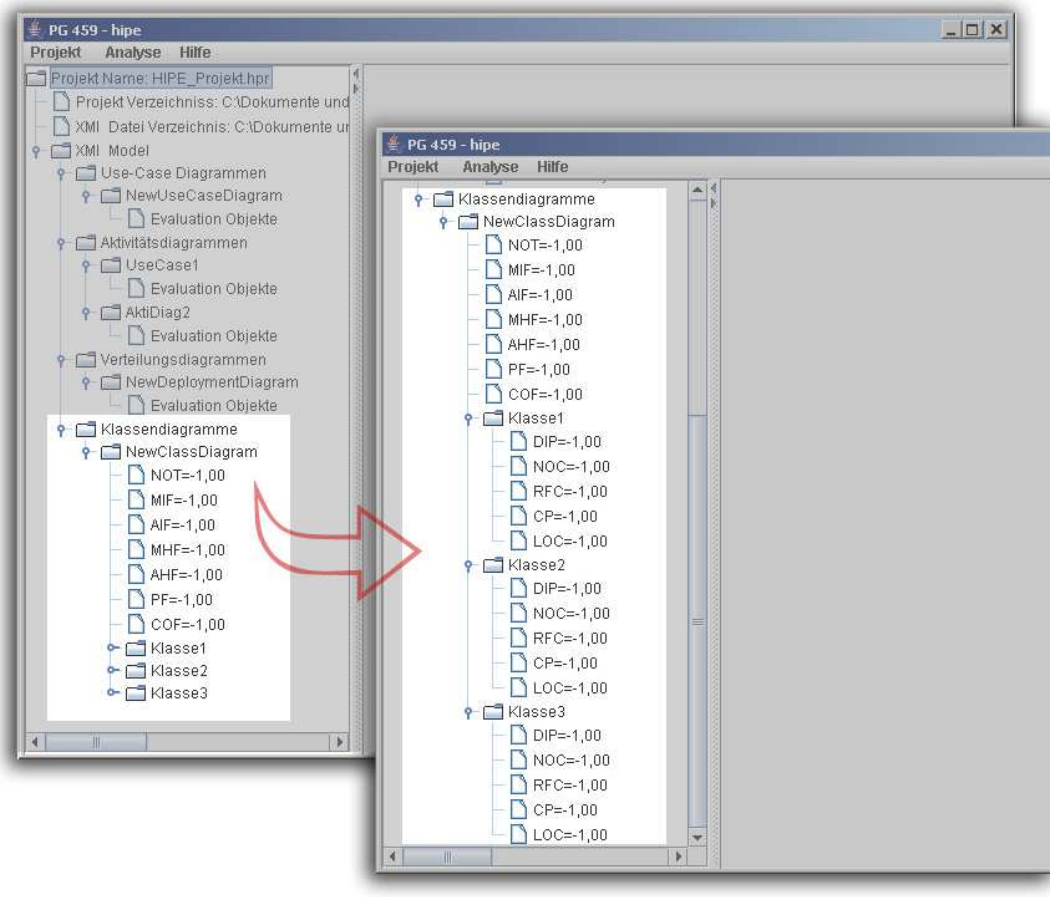

Abbildung 5.13: Abbildung der weichen Leistungsmaßen für das Klassendiagramm und für die einzelnen Klassen des Klassendiagramms im Projektverzeichnis-Baum

Man kann die Einstellungen jederzeit über den entsprechend benannten Untermenüpunkt des Menüs „Analyse“ ändern (siehe Abbildung 5.7 auf S. 187). Im entsprechenden Dialog werden die nötigen Eingaben über die beiden Felder „Server“ und „Port“ dem Programm mitgeteilt. Den Vorgang schließt man mit einen Klick auf „OK“ ab und kehrt zum Hauptfenster zurück.

5.3.3.11 Server starten

Das Server-Modul von HIPE befindet sich im ausführbaren Java-Archiv HIPEServer.jar. Es sollte unabhängig von HIPE auf einem Rechner mit läuffähiger HIT-Installation ausgeführt werden und dient dazu, die von HIPE in HI-SLANG kompilierten UML-Modelle über das Netzwerk anzunehmen und deren Analyse mit HIT zu starten (siehe dazu auch Abschnitt 7.3.3 des Zwischenberichts [PG459 (2005)], „Produktschnittstellen“). Der HIPE-Server kann durch das Kommando

```
java -jar HIPEServer.jar Port ArbeitsVerzeichnis HitVerzeichnis [-d]
```

gestartet werden. Dabei besteht die Möglichkeit, den Server als Vordergrund- bzw. Hintergrundprozess auszuführen. Wenn der Server im Vordergrund ausgeführt wird, kann er durch

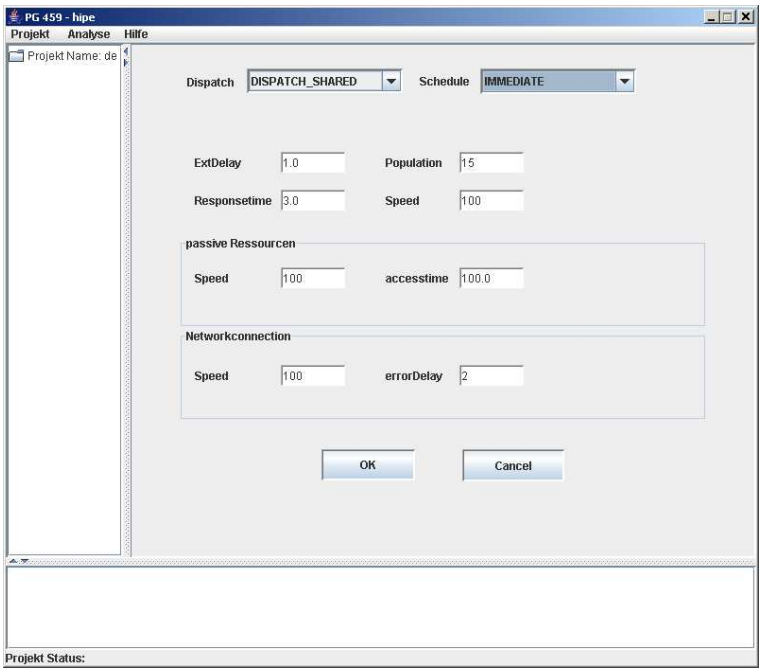

Abbildung 5.14: Menüpunkt Default-Werte setzten

das Drücken der `Enter`-Taste beendet werden. Bei der Ausführung als Hintergrundprozess ist die Beendigung nur über das Kommando

```
pkill java
```

möglich. Im Folgenden werden die Eingabeparameter des HIPE-Servers beschrieben.

- `Port`:
  Gibt den Port an, auf dem der Server auf einkommende Verbindung horcht. Dieser muss, neben den Rechnernamen, den Klienten bekannt gemacht werden, wenn diese eine Verbindung aufbauen wollen.

- `Arbeitsverzeichnis`:
  Definiert das Arbeitsverzeichnis des Servers; in diesem Ordner werden die zum Server übertragenen Dateien und deren Auswertungen gespeichert. Diese werden nicht gelöscht, so dass es möglich ist, die Ergebnisse früherer Analysen abzurufen. Da es prinzipiell möglich ist, auf einem Rechner mehrere HIPE-Server auszuführen, ist darauf zu achten, dass jeder Server sein eigenes Arbeitsverzeichnis erhält. In diesem werden neben den zuvor genannten Dateien noch andere für den Server wichtige Einstellungen gespeichert.

- `HitVerzeichnis`:
  Gibt den Ordner an, in dem sich die ausführbare HIT-Datei befindet.

- `-d`: Optionaler Parameter, er muss mit angegeben werden, wenn der Server als Hintergrundprozess gestartet werden soll.

Um einen Server auf Port 5000, dem Arbeitsverzeichnis /home/Nasched/HipeServerWD und dem Pfad zu HIT /app/unido-i04pub/HIT.exp zu starten, muss also folgendes Kommando eingegeben werden:

```
java -jar HIPEServer.jar 5000 /home/Nasched/HipeServerWD /app/uni
                          do-i04pub/HIT.exp
```

Als Hintergrundprozess ist folgendes Kommando nötig:

```
nohup java -jar HIPEServer.jar 5000 /home/Nasched
  /HipeServerWD /app/unido-i04pub/HIT.exp -d &
```

5.3.3.12 Hilfe

Als Hilfe steht dem Benutzer das vorliegende Handbuch zur Verfügung. Es ist über den Menüpunkt „Hilfe“ und den Untermenüpunkt „Handbuch“ zu erreichen (siehe Abbildung 5.4 auf S. 186).

5.4 Ergonomische Eigenschaften

Die HIPE-Benutzeroberfläche richtet sich an Software-Modellierer und somit an erfahrene Rechner-Nutzer. Bei der Gestaltung des Programms und insbesondere der Benutzeroberfläche wurde großer Wert auf die einfache und schnelle Durchführung der benötigten Funktionen gelegt. Dem Benutzer wird ermöglicht, alle Aufgaben direkt über passend benannte Menüpunkte an der Hauptmenüleiste zu erreichen, wo alle wichtigen Funktionen gebündelt dargestellt werden. Mit jedem Mausklick auf die entsprechenden Untermenüpunkte öffnen sich Dialoge, die die gewünschte Aktion unterstützen.

5.5 Beispielanwendung

Im Folgenden wird die Funktionsweise von HIPE an einem Beispiel erklärt, wobei zu betonen ist, dass der Schwerpunkt auf dem prinzipiellen Vorgehen liegt, nicht auf der Realitätsnähe und/oder der Relevanz des Modells.

Es sollen einige Abläufe eines Internet-Cafés modelliert werden. Durchschnittlich 80 Prozent der Kunden wollen ausschließlich surfen, der Rest besucht das Café, um ein PC-Spiel zu spielen. Das Café beschränkt jedoch die Spielzeit stets auf eine halbe Stunde, um einer ”Verstopfung” des Cafés mit Dauerspielern vorzubeugen und um den Server, der sämtliche Terminals bedient, nicht zu überlasten. Die Kunden können an den Internet-Terminals ihre heruntergeladenen Daten auf eine CD brennen, was auch rege genutzt wird. Jeder Kunde benötigt ein Mitgliederprofil des Internet-Cafés und muss sich zum Surfen per Login dem System bekannt machen. Es wird angenommen, dass im Durchschnitt alle zwei Minuten ein Kunde das Lokal betritt. Um eine eventuelle Aufrüstung der Ausstattung abschätzen zu können, soll die Benutzung der Internet-Terminals (inklusive der durchschnittlichen Anzahl an Kunden) näher untersucht werden. Wie bereits erwähnt, dienen alle Terminals nur als Ausgabe-Gerät für den Server, der sämtliche Aufgaben sowohl der Spiele- als auch der Internet-Terminals abarbeitet.

5.5.1 Erstellen des Eingabe-UML-Diagramms

Um das System bewerten zu können, muss dieses in Form eines UML-Diagramm modelliert und mit Leistungsparametern versehen werden. Mit *MetaMill*[4] wird ein System modelliert, das drei Hardwarekomponenten (`myCPU`, `CDBrenner` und `remoteRepository`), ein kleines Klassendiagramm bestehend aus zwei Klassen `UserProfile` und `Executable`, ein UseCase-Diagramm und zwei dazugehörige Aktivitätsdiagramme enthält. Das entsprechende Deployment-Diagramm ist in Abbildung 5.15 auf S. 198 zu sehen, Abbildung 5.16 zeigt das Klassendiagramm, Abbildung 5.17 enthält das Anwendungsfalldiagramm und die Aktivitätsdiagramme sind in den Abbildungen 5.18 und 5.19 abgebildet.

Die Hardwarekomponente `CDBrenner` ist offensichtlich das für das CD-Brennen verwendete Gerät, `myCPU` symbolisiert die Recheneinheit, die die Internet-Terminals bedient und das `remoteRepository` stellt die Benutzerprofile zur Verfügung. Dementsprechend ist die Klasse `UserProfile` im Deploymentdiagramm dem `remoteRepository` zugeordnet. Die Klasse `Executable` stellt die ausführbare Datei des Webbrowsers dar. Der Akteur soll mit 80-prozentiger Wahrscheinlichkeit den Anwendungsfall `Internet` und ansonsten den Anwendungsfall `PCSpiel` anstoßen. Sowohl die Aktivitäten des UseCase `PCSpiel` als auch die der Internet-Terminals sollen auf der Standardhardware ausgeführt werden, die wiederum im Deploymentdiagramm mit dem `defaultServer`-Statement auf `myCPU` festgelegt wurde[5]. Die Notiz am Anwendungsfall `PCSpiel` muss also keine Statements bezüglich der verwendeten Hardware enthalten. Die am UseCase anfallende Bearbeitungsdauer ist bereits bekannt (nämlich die fürs Spielen erlaubten 30 Minuten, die wohl jeder Spieler voll ausnutzen wird), so dass der für die Bearbeitung der Last eingesetzte Server mittels `let responsetime = 1800` gerade so skaliert wird, dass jeder an ihm ankommende Prozess durchschnittlich 1800 Zeiteinheiten (= 30 Minuten) für seine Verarbeitung beanspruchen soll.

Der Anwendungsfall `Internet` wird statt dessen durch das gleichnamige Aktivitätsdiagramm verfeinert. An ihm sind die Antwortzeit und die durchschnittliche Anzahl an Prozessen von Interesse, denn diese Werte stehen für die durchschnittliche Verweildauer eines Kunden an einem Internet-Teminal bzw. die durchschnittliche Anzahl an Kunden, die pro Zeiteinheit ein Terminal belegen[6].

Das verfeinernde Aktivitätsdiagramm mit Namen `Internet` enthält drei Aktivitäten, von denen zwei einen Objektfluss (`Benutzerprofil` vom Typ `UserProfile`) einschließen. Bis auf `CD_brennen` wird keine Aktivität dieses Diagramms verfeinert. Die Verfeinerung besteht lediglich aus dem ca. eine halbe Minute (30 Sekunden) dauernden Zusammenstellen der Dateiliste und dem eigentlichen Brennvorgang, der im Durchschnitt zehn Minuten (= 600 Sekunden) in Anspruch nimmt. Die Logik der Aktivität `Browser_oeffnen` soll durch die Methode

[4] Bei der Entwicklung von HIPE wurde das UML-Tool *Metamill* für die Erzeugung der UML-Dateien als Standard festgelegt, da *Metamill* die Modellierungssprache UML in der Version 2.0 unterstützt und die UML-Modelle komplett als eine einzige XMI-Datei exportiert.

[5] Dies ist also der Rechner, der sowohl die Aufgaben für die Internet-Terminals als auch die der Spiel-Terminals übernehmen soll. Um dem entstehenden Leistungsbedarf gerecht werden zu können, ist er als System mit paralleler Prozessverarbeitung konzipiert, d.h. für ihn gilt `schedule = immediate` und `dispatch = equal`.

[6] Da die hier betrachtete Zeiteinheit stets eine Sekunde ist, müssen die Ergebnisse der Analyse im Anschluss auf eine sinnvolle Größe (z.B. Stunde) skaliert werden.

`executableMeth1` der Klasse `Executable` realisiert werden.

Für die Spezifikation der Hardware wurden im Deploymentdiagramm weitgehend selbsterklärende Parameter angegeben:

- `myCPU` ist eine aktive Hardwarekomponente mit der Fähigkeit zur Parallelverarbeitung von Prozessen und einer Leistung von 20 Millionen Operationen pro Sekunde. Sie erhält Zugriff auf 256MB Speicher. Mit `let schedule = immediate` und `let dispatch = equal` wird festgelegt, dass die ankommenden Aufgaben sofort zur Bearbeitung freigegeben werden sollen und jede der Aufgaben mit voller Rechenleistung der Hardwarekomponente bearbeitet wird.
- `remoteRepository` ist eine passive Hardware-Ressource (nicht-flüchtiger Speicher) der Größe 768MB mit einer Leistung von 100000 Operationen pro Sekunde (als Approximation eines internen Hardware-Controllers) und der Zugriffzeit von 0.1 Sekunden.
- Die Netzwerkverbindung, die beide Hardwarekomponenten verknüpft, hat eine Bandbreite von 128kbps mit einer Fehlerrate von durchschnittlich "1/60 Fehler pro Sekunde" (also ca. eine Fehlübertragung pro Minute). Bei der Übertragung der Daten gilt, dass die minimale Paketgröße 4 elementare Speichereinheiten (also z.B. 4 Byte) beträgt. Ist der zu versendende Datensatz größer als 1024 Einheiten, soll er in entsprechende Teilpakete aufgesplittet werden.
- Der `CDBrenner` ist die Hardware-Einheit, die für die Aktivität `brennen` zuständig ist. Da über die Hardware-Eigenschaften dieses Geräts keine näheren Informationen vorliegen, wird seine Geschwindigkeit mit einer elementaren Operation pro Zeiteinheit angegeben. (Auf diese Angabe hätte auch verzichtet werden können. In diesem Fall würde der Wert aus dem `speed`-Feld des GUI-Dialogs zur Definition der Default-Werte eingesetzt werden.)

Für die Analyse wurden folgende Messpunkte definiert:

- Am Akteur des Anwendungsfalldiagramms soll die Antwortzeit (also die durchschnittliche Verweildauer eines Kunden im Internetcafé) ermittelt werden.
- Am Anwendungsfall `Internet` wird die Antwortzeit und der Durchsatz (an Kunden) verlangt.
- An der Aktivität `Browser_oeffnen` wird die Antwortzeit gesucht.
- An der Hardwarekomponente `myCPU` ist die Auslastung von Interesse.

5.5.2 Analyse durch HIPE

Im nächsten Schritt wird die Leistungsanalyse mit HIPE und HIT durchgeführt. Zunächst wird HIPE gestartet und das Projekt für unser Anwendungsbeispiel „HipeBeispiel“ erzeugt (siehe Abbildung 5.20 auf S. 200). Es wird definiert, wo das Projekt gespeichert werden soll und welchen Namen es tragen soll. Hier ist zu beachten, dass Pfad und Name keine Umlaute und Leerzeichen enthalten dürfen (vgl. Abschnitt 5.7 über Besonderheiten bei der Benutzung von HIPE, Seite 207).

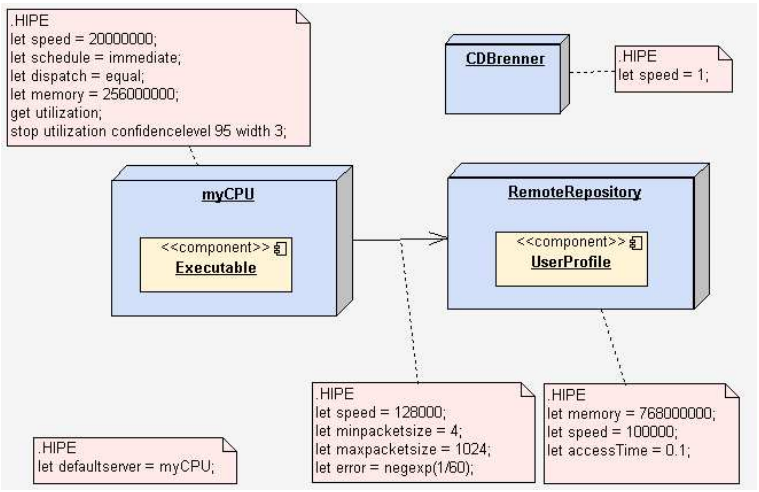

Abbildung 5.15: Das Deployment-Diagramm für das Internet-Café-Beispiel

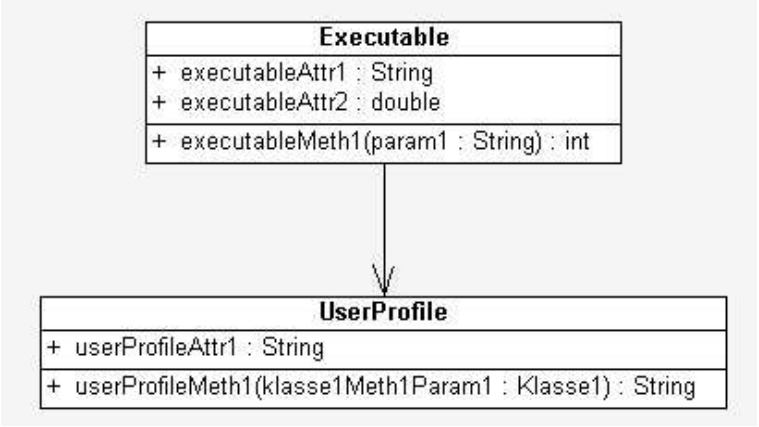

Abbildung 5.16: Das Klassen-Diagramm für das Internet-Café-Beispiel

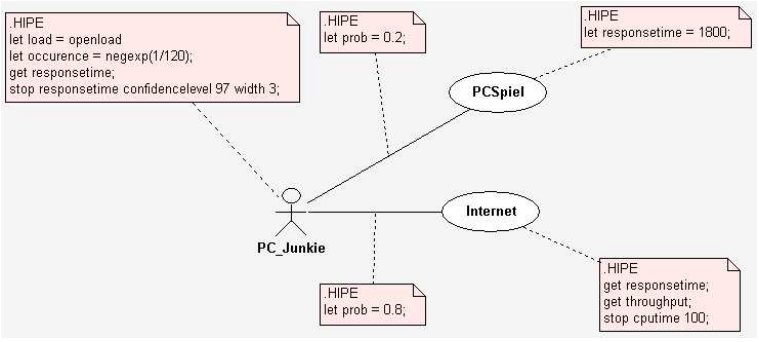

Abbildung 5.17: Das Anwendungsfall-Diagramm für das Internet-Café-Beispiel

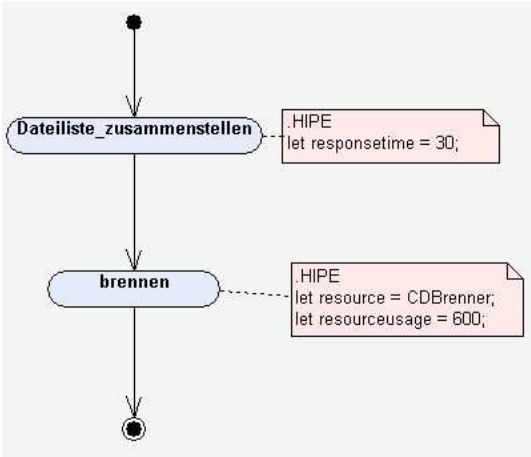

Abbildung 5.18: Das Aktivitätsdiagramm "AktiDiag2" des Internet-Café-Beispiels

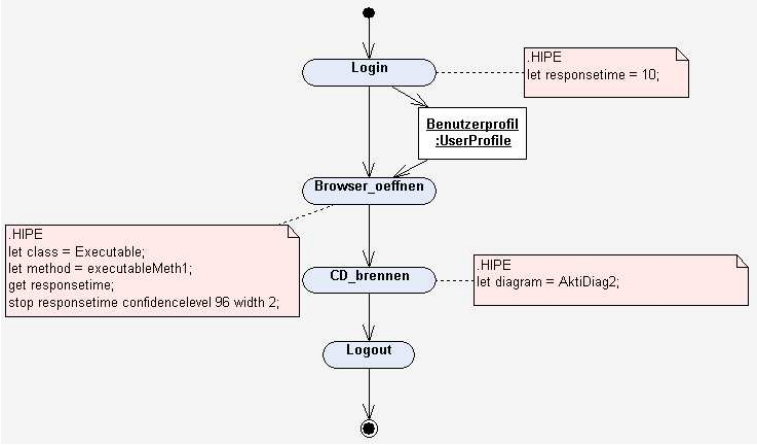

Abbildung 5.19: Das Aktivitätsdiagramm "Internet" des Internet-Café-Beispiels

Nach dem wir den Pfad und Namen eingegeben haben, klicken wir auf „Weiter“ und gelangen zum Dialogfenster „XMI-Dokument importieren“, wo wir die mit Metamill exportierte XMI Datei für den Import angeben (siehe Abbildung 5.21 auf S. 200).
Als dritten Schritt der Projekt-Erzeugung müssen wir Server und Port für die Durchführung der Analyse einstellen (siehe Abbildung 5.22 auf S. 201). Auf dem Server *dame.cs.uni-dortmund.de* steht das HIPE-Server-Modul am Port 5000 zur Verfügung. Durch Bestätigung mit „OK“ haben wir unser Beispiel-Projekt vollständig erzeugt.
Nach dem Erzeugen des Projekts müssen wir noch die Default-Parameter für die Analyse einstellen. Abbildung 5.24 auf S. 203 zeigt den dazugehörigen Dialog[7]. Sie werden bei der Generierung des hierarchischen Leistungsmodells an jenen Stellen eingesetzt, an denen keine entsprechenden Angaben im UML-Diagramm gefunden werden. Würde beispielsweise die Notiz im Deploymentdiagramm des Anwendungsbeispiels fehlen, die besagt, dass die

[7] Die in Abbildung 5.24 auf S. 203 enthaltenen Werte sind die beim Öffnen des Dialogs als Initialisierung durch HIPE eingesetzten Angaben. Sie sollten vor einer Analyse geprüft und gegebenenfalls angepasst werden!

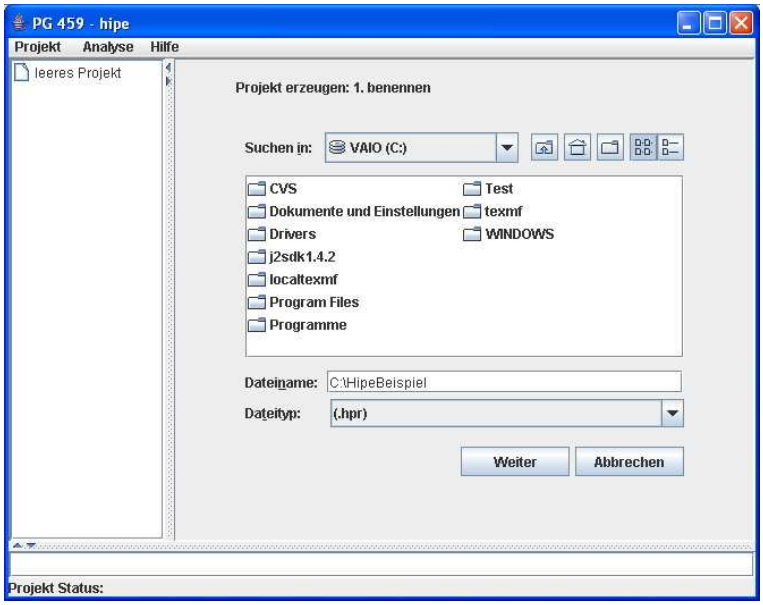

Abbildung 5.20: Beispiel-Projekt erzeugen

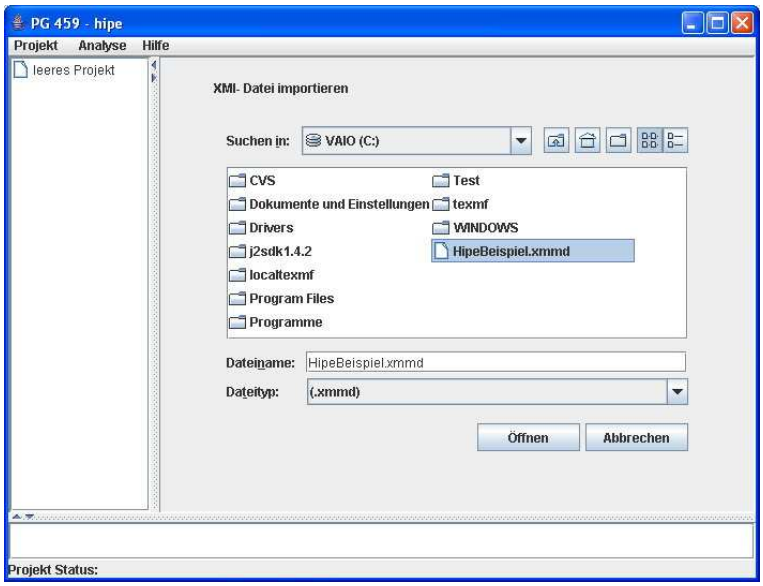

Abbildung 5.21: Beispiel-XMI-Datei Importieren

standardmäßig einzusetzende Hardwarekomponente die Komponente mit Namen `myCPU` ist (`let defaultserver = myCPU`), so könnte für die Aktivität `logout` im Aktivitätsdiagramm `Internet` nicht festgestellt werden, auf welcher Hardware sie ausgeführt werden soll. In diesem Fall würde die benötigte Hardwarekomponente von HIPE in Form eines HIT-Servers definiert und in das Leistungsmodell integriert werden, für dessen Parameter `dispatch`, `schedule` und

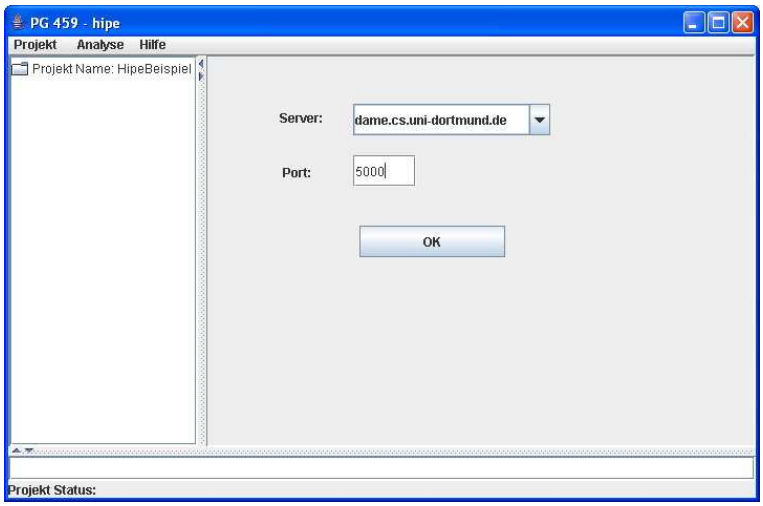

Abbildung 5.22: Server und Port für das Beispiel-Projekt wählen

`speed` die Werte aus der GUI eingesetzt würden.
Folgende Parameter können für solche Situationen in der GUI gesetzt werden:

- Für atomare Modellkomponenten:
  - `dispatch`: Die Auswahl von „DISPATCH_EQUAL“ besagt, dass die hiermit parametrisierte atomare Komponente die an ihr ankommenden Prozesse mit ihrer vollen Rechenleistung abarbeiten soll. „DISPATCH_SHARED“ besagt statt dessen, dass ihre Leistung gleichmäßig auf alle aktuell zu bedienenden Prozesse aufzuteilen ist.
  - `schedule`: Für die Scheduling-Disziplin stehen die Belegungen „FCFS“ und „IMMEDIATE“ zur Verfügung. Ersteres definiert für den so beschriebenen HIT-Server eine Warteschlange für ankommende Prozesse, die nach dem FIFO-Prinzip abgearbeitet wird. „IMMEDIATE“ verhindert die Einrichtung einer solchen Warteschlange, jeder ankommende Prozess wird umgehend bedient.
  - `speed`: Definiert die Anzahl an Operationen pro elementarer Zeiteinheit, die die Standard-Hardware ausführen kann.
- Für die Lastbeschreibung in UseCase- und Aktivitätsdiagrammen:
  - `extDelay`: Legt für eine geschlossene Arbeitslast die Zeit fest, die zwischen dem Ende der Bearbeitung eines Jobs und der Ankunft des nachfolgenden vergehen soll, falls keine andersartige Aussage gefunden wird.
  - `population`: Legt analog für eine geschlossene Arbeitslast die Population fest, falls keine andersartige Aussage gefunden wird.
  - `responsetime`: Legt die standardmäßig einzusetzende Bedienzeitanforderung einer Aktivität oder eines Anwendungsfalls fest.
- Für (passive) Ressourcen:

  - `speed`: Definiert analog zum gleichnamigen Parameter bei atomaren Komponenten die standardmäßig angenommene Leistung einer passiven Ressource[8].
  - `accesstime`: Hiermit wird die Zugriffszeit für passive Ressourcen für solche Fälle spezifiziert, in denen keine analoge Aussage im Deploymentdiagramm gefunden wird.

- Für NetWorkConnections:
  - `speed`: Analog zum *speed*-Parameter bei atomaren Komponenten und passiven Ressourcen wird hier die Geschwindigkeit für solche Netzwerkverbindungen festgelegt, an denen keine anderslautende Aussage gefunden wird. Die Einheit ist elementare Dateneinheit pro Sekunde.
  - `error`: Mit diesem Parameter kann der standardmäßig anzunehmende Zeitabstand zwischen dem Auftreten zweier Fehler auf der simulierten Leitung angegeben werden. Er wird überall dort eingesetzt, wo zwar eine Netzwerkverbindung definiert, aber eine entsprechende Angabe nicht gemacht wurde.

Schließlich müssen vor dem letztendlichen Starten der Berechnungen noch die Ausführungs-Parameter der Analyse definiert werden. Dies geschieht nach Auswahl der Option „Analyse starten“ im Menü „Analyse“ im daraufhin geöffneten Dialogfeld. Hier stehen folgende Optionen zur Verfügung:

- Für die einzelnen Lösungsmethoden (*Solver*) sind die anzuwendenden Stop-Bedingungen anzugeben. Sie definieren (wie in Abschnitt 1.2.4.4 auf Seite 27 beschrieben) die Konditionen, deren Erfülltsein einen Abbruch der Analyse bewirken. Es werden jeweils nur die gültigen Stop-Bedingungen angeboten. Man beachte, dass die im UML-Diagramm gefundenen Stop-Statements die hier eingetragenen Werte überschreiben können.

- Unter „Estimators“ muss vor Beginn der Analyse die gewünschte Auswahl an Schätzern zusammengestellt werden. Auch sie wurden bereits in Abschnitt 1.2.4.4 erklärt.

(Die theoretischen Details der aufgezählten Optionen sind im Kapitel 1 ab Seite 9 nachzulesen.)

Nach Auswahl mindestens eines Estimators ist der Knopf „Analyse starten“ aktiviert und die Analyse kann angestoßen werden. Für jedes einzelne Diagramm stehen bei erfolgeicher Umwandlung der Eingabe in HI-SLANG-Code und Ausführung von HIT die Werte jedes Evaluationsobjekts, das für die Auswertung des Leistungsmodells erzeugt wurde, im linken Teil des Hauptfensters (dem Projektbaum) zur Verfügung und kann durch einfaches Anklicken angezeigt werden (kein Doppelklick!). Dabei werden Diagramme erzeugt, die für die ausgewählten Löser und Schätzer die Ergebnisse der Analyse darstellen (für Analysen ohne Experimentserien enthalten sie natürlich für die meisten Leistungsmaße lediglich einen einzigen vertikalen Balken). Nun kann der Benutzer mit den gelieferten Informationen Rückschlüsse auf die Qualität des eingegebenen UML- bzw. Softwaremodells ziehen und — falls nötig — entsprechende Änderungen daran vornehmen.

[8] Dies ist notwendig, da auch passive Ressourcen als HIT-Server realisiert werden und somit diesen Parameter zur korrekten Definition benötigen. Als Vorstellung für diesen Parameter kann die Geschwindigkeit des internen Hardware-Controllers dienen.

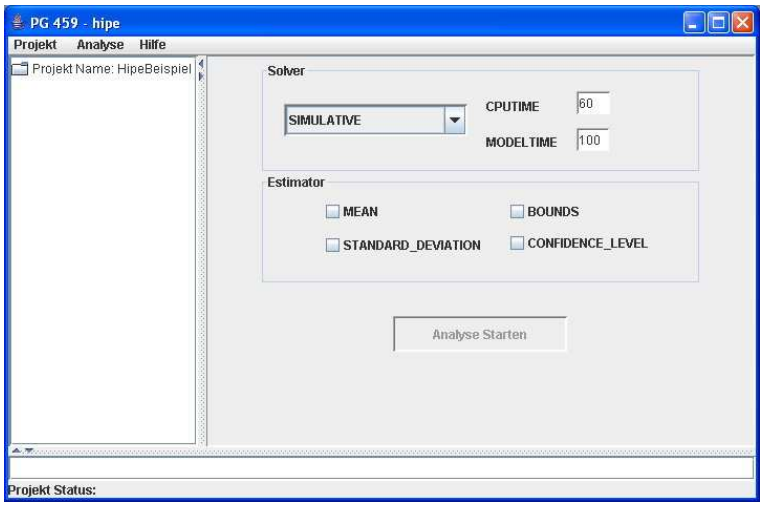

Abbildung 5.23: Analyse des Beispiel-Projekts starten

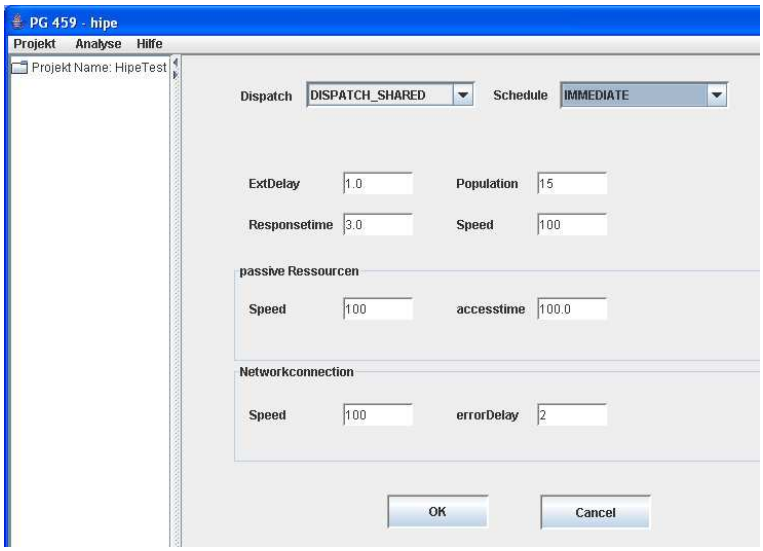

Abbildung 5.24: Dialogfeld zur Eingabe der Default-Werte

Für das vorgestellte Beispiel sind die Ergebnisse einer simulativen Erhebung des Estimators MEAN für das Leistungsmaß Responsetime für den Anwendungsfall `Internet`, für den Akteur `PC_Junkie` und für die Aktivität `Browser_oeffnen` in den Abbildungen 5.25, 5.26 sowie 5.27 auf S. 205 zu sehen. Die textuelle Ausgabe[9] für die UTILIZATION der Hardwarekomponente `myCPU` ergab einen Mittelwert von 0.33 bei einer Konfidenz von 95 Prozent (Breite des Konfidenzintervalls: 5.57%). Nach einer eventuellen Umrechnung der Daten von "pro Sekunde" in "pro Stunde" erkennt man beispielsweise für den Anwendungsfall `Internet`, dass ein Kunde

[9] Die Ergebnisse des Meßpunkts `myCPU` waren zum Zeitpunkt der Erstellung dieses Anwendungsbeispiels noch nicht verfügbar. Die Messungen an Elementen des Verteilungsdiagramms wurden erst kurz vor der Fertigstellung des Projekts implementiert.

durchschnittlich mindestens 12,4 Minuten am Internet-Terminal verbringt. Dieser Wert beinhaltet noch nicht die Zeit, die der Kunde tatsächlich online verbringt(!), sondern lediglich die Zeit, die er für die Archivierung der Daten benötigt. Die reine Online-Zeit schwankt von Kunde zu Kunde unverhältnismäßig stark, so dass deren Betrachtung im Modell nur mit stark verfälschenden Nebenwirkungen einbezogen werden könnte. Somit sind andere Werte der Analyse wichtiger und interessanter, beispielsweise die Auslastung von `myCPU`. Sie zeigt, dass das System die anstehenden Aufgaben ohne Weiteres verkraftet und somit auch einem höheren Andrang an Kunden ausgesetzt werden könnte (z.B. ein neuer Kunde pro Minute). Für dieses Modell ergibt die Analyse also keine negativ-Ergebnisse; das zugrundeliegende System benötigt keine Veränderungen.

Jedoch sind die in Abbildung 5.29 auf S. 206 gezeigten Werte für die Antwortzeit am Akteur des Anwendungsfalldiagramms (Schätzer Konfidenzlevel), die in Form einer Experimentserie ermittelt wurden, auch ohne ein solches negativ-Ergebnis durchaus interessant. Für die Definition der Experimentserie wurde die Dauer variiert, die ein Kunde braucht, um die Liste der auf CD zu brennenden Dateien zusammenzustellen. Genauer wurde die Analyse anhand von fünf Vorgaben für die Antwortzeit an der Aktivität `Dateiliste_zusammenstellen` durchgeführt. Statt des einen Wertes `30` (Sekunden) wird dieser Aktivität nun die Folge `600 & 300 & 60 & 30 & 10` zugewiesen (Abbildung 5.28). Das Resultat in Abbildung 5.29 zeigt, dass die Annahme von 30 Sekunden für das Zusammenstellen der Dateiliste bereits das optimale Ergebnis bezüglich der durchschnittlichen Antwortzeit des Gesamtsystems darstellt. Man beachte die Legende: die jeweils linke Ergebnis-Säule der Experimente stellt den Mittelwert dar, die mittlere und rechte Säule repräsentieren die obere respektive untere Grenze des Konfidenzintervalls als Absolutwert. Der geringfügige Anstieg der Antwortzeit beim letzten Experiment[10] und der Unterschied dieses Wertes zu dem in Abbildung 5.26 gezeigten Resultat lässt sich dabei durch die stochastischen Schwankungen bei der (erneuten) simulativen Auswertung des Leistungsmodells erklären. Der Anstieg der MEAN-Werte in negativer Richtung der x-Achse lässt eine (sub-)lineare Steigung der Antwortzeit des Gesamtsystems für Belegungen des Antwortzeit der Aktivität `Dateiliste_zusammenstellen` größer oder gleich 60 Sekunden erahnen. Damit kann als Ergebnis für das untersuchte Verhältnis zwischen der Komplexität der Aktivität `Dateiliste_zusammenstellen` und dem Verhalten des Gesamtsystems eine gute Skalierung festgehalten werden.

5.6 Hilfen/Fehlermeldungen

In diesen Abschnitt werden möglicherweise auftretende Fehlermeldungen erklärt und Lösungsmöglichkeiten bereitgestellt. Als kleine Hilfe und Anleitung erscheint beim Programmstart im rechten Feld die Aufforderung, ein Projekt zu erzeugen oder ein vorhandenes Projekt zu öffnen (Abbildung 5.3 auf S. 185).

[10] Die Zählung der Experimente beginnt wie in der Informatik üblich bei 0.

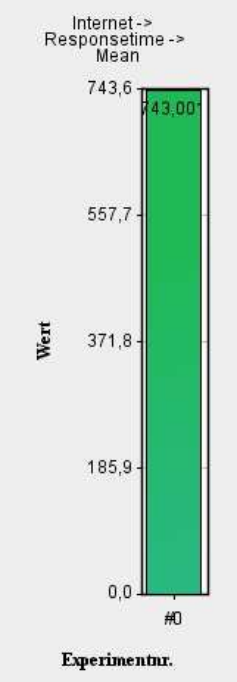

Abbildung 5.25: `Internet`

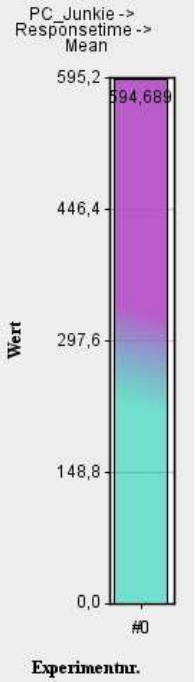

Abbildung 5.26: `PC_Junkie`

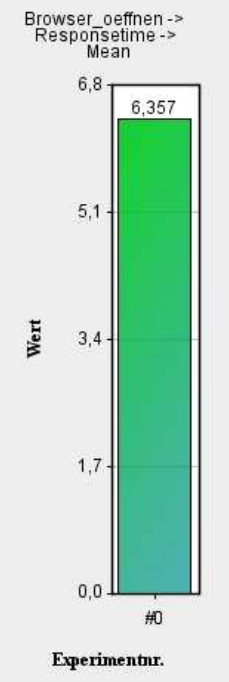

Abbildung 5.27: `Browser_oeffnen`

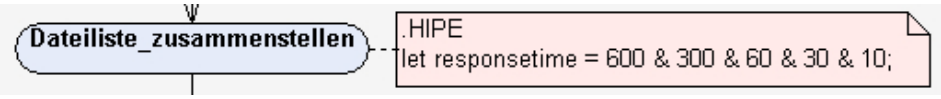

Abbildung 5.28: Belegung der Aktivität `Dateiliste_zusammenstellen` des Anwendungsbeispiels mit einer Experimentserie

5.6.1 Fehlermeldungen

Es gibt zwei Arten von Fehlermeldungen, die von Bedienungsfehlern oder Fehlern im UML-Modell herrühren. Weiterhin sind Fehler, die im UML-Modell auftreten können, in Warnungen und kritische Fehler untergliedert.

- Warnungen treten auf, wenn HIPE auf Angaben stößt, die entweder unvollständig sind, aber automatisch durch Einsetzen von Default-Werten vervollständigt werden können, oder wenn bestimmte Gegebenheiten der Analyse, wie der vom Nutzer ausgewählte Löser, gewisse Anpassungen notwendig machen. Als Folge der Behandlung dieser Situationen kann das Verhalten des Modells danach von dem vom Nutzer intendierten Verhalten abweichen.
- Kritische Fehler treten auf, wenn wichtige Angaben fehlen oder unvollständig sind, aber nicht durch Einsetzen von Default-Werten vervollständigt werden können, oder von HIPE nicht entschieden werden kann, welche Angabe in einer konkreten Situation berücksichtigt werden soll. Das Auftreten solcher Fehler führt dazu, dass das betrachtete Modell nicht analysiert werden kann.

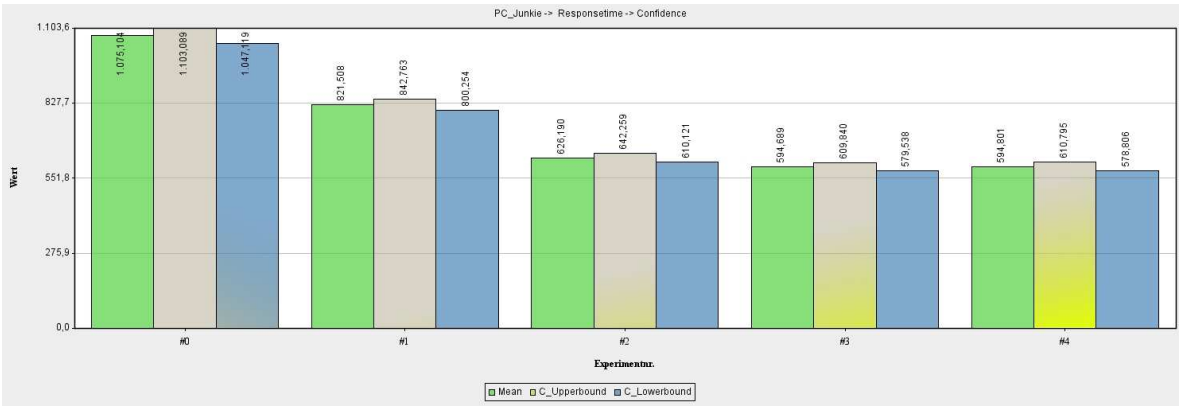

Abbildung 5.29: Resultat einer Experimentserie für das Anwendungsbeispiel

Alle Fehlermeldungen werden in textueller Form in dem im unteren Teil des Hauptfensters befindlichen Ausgabefenster ausgegeben.

Fehlermeldungen, die von Bedienungsfehlern herrühren		
Bedingung	**Fehlermeldung**	**Lösung**
Untermenüpunkt Analyse starten gewählt Untermenüpunkt Default Werte setzen gewählt Untermenüpunkt Server Einstellung gewählt	Error: kein Projekt vorhanden	Ein Projekt erzeugen oder öffnen

Fehlermeldungen, die von Fehlern im UML-Modell herrühren:

1. Summe der Übergangswahrscheinlichkeiten am Akteur <Akteur> größer als 1.
2. Unzulässiges Format der Wahrscheinlichkeitsangaben am Akteur <Akteur>.
3. Vom Nutzer angegebenes Aktivitätsdiagramm <Aktivitätsdiagramm> nicht vorhanden.
4. Verwiesenes Aktivitätsdiagramm <Aktivitätsdiagramm> existiert nicht.
5. Übergangsannotationen vom Knoten <Diagrammelement> illegal:
   Summe der Wahrscheinlichkeiten größer als 1. x Kanten annotiert und y Kanten unannotiert.
6. Dem Akteur <Akteur> sind keine Anwendungsfälle zugeordnet.
7. Keine Arbeitslast am Akteur <Akteur> definiert.
8. Gemeinsame Definition individueller Ankunftszeiten (ARRIVALTIME) und Intervallen (OCCURENCE) am Akteur <Akteur>.
9. Gemeinsame Definition individueller Ankunftszeiten, der Population und Verzögerungszeit am Akteur <Akteur>.
10. In der Experimentserie definierte Zwischenankunftszeiten folgen nicht der gleichen Verteilung.
11. Concurrent-Bereich <Concurrent-Block> nicht terminiert.

12. Doppelte Deklaration von ausgeführter Klasse und ausführender CPU am Knoten <Diagrammelement>.
13. Klasse mit dem Namen <Klasse> existiert nicht.
14. Klasse <Klasse> keiner Hardware im Deployment-Diagramm zugeordnet.
15. Knoten <Diagrammelement> existiert nicht.
16. Klasse <Klasse> nicht im Deployment-Diagramm vorhanden.
17. Objekt kann nicht instanziiert werden, da Speicher von <Hardwarekomponente> nicht ausreicht.
18. Gemeinsame Definition von FOR- und AVERAGE-Loop bei Diagrammelement <Diagrammelement>.
19. Keine MEASURE-Statements vorhanden!
20. Keine Maße an <Diagrammelement> vorhanden!
21. User-Abbruch wegen zu vieler Experimentserien!
22. Exception!
23. Kein Projektname oder keine Lösungsmethode gefunden!

5.7 Besonderheiten

Bei der Erstellung der UML-Modelle in Metamill müssen die drei gelben Ordner `UseCaseView`, `DesignView` und `ImplView` gelöscht werden(!), damit die Analyse von HIPE korrekt durchgeführt werden kann. Die Namen der Diagrammelemente dürfen keine Umlaute oder Leerzeichen enthalten. Ebenso darf für Elementnamen keine Zeichenfolge vergeben werden, die bereits in HIT für dessen Sprach- bzw. Modell-Elemente reserviert ist (z.B. „Server“).

5.7.1 Systemanforderungen

HIPE wurde unter Verwendung des UML-Modellierungstools *EclipseUML* der Firma *OMONDO* entwickelt und getestet. Für die UML-Modelle wird die Verwendung des UML-Modellierungstools *MetaMill* empfohlen. Die Funktionsfähigkeit mit anderen UML-Tools wird nicht garantiert. Weiterhin werden eine Java Virtual Machine ab Version 1.5 (bzw. 5.0) für die fehlerfreie Programmausführung und ein mit Solaris betriebener Rechner mit einer lauffähigen Version des Leistungsbewertungswerkzeugs HIT vorausgesetzt.
Die allgemeinen Hardwarevoraussetzungen entsprechen denen der virtuellen Java-Maschine für das ausgewählte Betriebssystem. Zusätzlich wird eine Anbindung an ein Netzwerk (bzw. das Internet) benötigt, über das ein Solaris-System mit lauffähiger HIT-Installation erreicht werden kann. HIPE kann auf einer Windows-Umgebung eingesetzt werden.

Um das Benutzungshandbuch zu öffnen wird das Programm *Acrobat Reader* benötigt. Des Weiteren müssen die entsprechenden Umgebungsvariablen in der Systemsteuerung gesetzt

werden. Das erfolgt in Windows über den Dialog „Systemeigenschaften“ (zu erreichen über „Start - Einstellungen - Systemsteuerung - System“). Im Dialogfenster wählt man dann den Karteireiter „Erweitert“ und schließlich das ganz unten befindliche Feld „Umgebungsvariablen“. Es öffnet sich ein weiteres Dialogfenster mit zwei Feldern, oben die Benutzervariablen, darunter die Systemvariablen. Unter den Systemvariablen wählt man die Variable „Path“ und klickt auf „Bearbeiten“. Es öffnet sich ein weiteres Dialogfenster „Systemvariable bearbeiten“, in dem man im Textfeld „Wert der Variablen“ den Pfad mit dem Speicherort des Acrobat Reader ergänzen muss. **WICHTIG:** Der schon existierende Pfad im Textfeld **darf nicht gelöscht werden**, er wird nur mit dem Pfad des Acrobat Readers — beginnend mit einem ”**;**” — ergänzt. Um den Vorgang abzuschließen, bestätigt man alle Dialoge mit „OK“ und verlässt die Systemsteuerung.

5.7.2 Restriktionen von HIPE

Zusätzlich zu den in Kapitel 1.2.9 beschriebenen Aspekten an nicht enthaltenen Funktionen sollten folgende Punkte bei der Anwendung von HIPE beachtet werden:

- An UML-Features werden derzeit ausschließlich diejenigen bei der Analyse berücksichtigt, die im Kapitel 1 ausdrücklich erwähnt werden. Die Menge der nicht betrachteten Konstrukte umfasst (aber ist nicht beschränkt auf):
  - *Include*- und *extends*-Beziehungen in Anwendungsfalldiagrammen
  - *Artefakte* in Deploymentdiagrammen
  - *Interfaces* in Klassen- und Deploymentdiagrammen
  - *Senden und Empfangen von Signalen* in Aktivitätsdiagrammen
  - Mehr als einen *flow finalizer* für Kontrollflüsse in Aktivitätsdiagrammen
  - *Interrupt flows* in Aktivitätsdiagrammen
  - *Regionsdefinitionen und Swimlanes* in Aktivitätsdiagrammen
  - *System boundaries* in Anwendungsfalldiagrammen
  - *Realization/Provide*-Assoziationen
- HIPE erwartet im übergebenen UML-Modell die Existenz eines einzigen UseCase-Diagramms sowie eines einzigen Deployment-Diagramms. Dies bedeutet insbesondere, dass ein UseCase nur durch ein Aktivitätsdiagramm verfeinert werden darf, nicht durch ein weiteres UseCase-Diagramm.
- Anwendungsfälle dürfen im Anwendungsfalldiagramm nicht gleichzeitig mit zwei Akteuren gekoppelt werden, die eine Arbeitslast unterschiedlichen Typs, oder falls alle Akteure geschlossene Arbeitslasten beschreiben, mit verschiedenen Werten für `extDelay`, definieren. Eine Methodik, die solche Konflikte erkennt und ggf. neue Services erzeugt, wurde nicht implementiert.
- In Abweichung zum Kapitel 1 können Aktivitäten nicht mit der Kombination *class/method* annotiert werden. Die Funktionalität, mit der die Komplexität einer Methode abgefragt werden kann, wurde nicht implementiert.
- Im Zuge jeder Aktivität kann (mittels `resource` und `resourceusage`) nur auf die Benutzung einer passiven Ressource verwiesen werden.

- Für das UML-Konstrukt der Wiederholungen ("Loop") gilt die Einschränkung, dass ein Diagramm-Element das Ziel höchstens einer Kante sein darf, die einen Loop definiert. Eine Methodik zur Sortierung von Schleifen nach "Größe", um eine Wohlverschachtelung herzustellen, ist zwar grundsätzlich möglich, wurde aber nicht implementiert.

- Bei der Modellierung von Netzwerkleitungen sind "logische Leitungen" zu modellieren, das heißt zwischen allen CPUs, zwischen denen eine Kommunikation (hier nur durch Objektflüsse realisiert) erfolgen soll, muss im Verteilungsdiagramm eine Leitung existieren.

- Unter den Lösern DOQ4 und LIN2 sind Netzwerkverbindungen nicht verwendbar, da LIN2 die Scheduling-Disziplin `prioprep` nicht unterstützt und für DOQ4 alle Anfragen identisch negexp-verteilt sein und den gleichen Bedienumfang haben müssen.

- Die Scheduling-Prozedur `random` für atomare Hardwarekomponenten kann in der HIPE-GUI nicht als Default gewählt werden.

- In Modellen, die mit analytischen Lösern ausgewertet sollen, dürfen keine nebenläufigen Bereiche vorkommen.

- Die im Kapitel 1 entworfenen Experimentserien konnten aufgrund einiger Eigenheiten von HIT nicht in der beschriebenen Form implementiert werden:

  - Die Definition von Experimentserien über Prozesserzeugungsraten ist nur zulässig, wenn die einzelnen vorgegebenen Verteilungen von Zwischenankunftszeiten alle vom selben Typ sind (also alle "deterministisch", cox-, normal-, oder negativ exponentialverteilt), und wenn über die ganze Serie nur derjenige Parameter variiert wird, der den Erwartungswert darstellt. Das heißt, dass die bei den Cox- und Normalverteilungen für die Beschreibung der Varianz zuständigen Parameter über alle Elemente der Serie konstant bleiben müssen.
  - Die Experimentserien für die Population und die Geschwindigkeiten `speed` von CPUs wurden nicht implementiert. Die "Listen" von Populationen und CPU-Geschwindigkeiten dürfen also nur eine einzige Angabe enthalten.
  - Weiterhin ist die Verwendung von Experimentserien ab der dritten Modellebene des UML-Modells auf höchstens eine Experimentserie pro Diagramm beschränkt. Das heißt, dass nur im UseCase-Diagramm und in denjenigen Aktivitätsdiagrammen, die einen UseCase verfeinern, mehrere Experimentserien pro Diagramm definiert werden dürfen. Für alle weiteren Aktivitätsdiagramme gilt die Beschränkung auf höchstens eine Experimentserie pro Diagramm.

- Die Werte der Zufallszahlengeneratoren (negexp, cox, etc. — vgl. Abschnitt 1.2.2) hängen essentiell von der Belegung einer HIT-internen Variablen ab. Diese behält im Verlauf einer Analyse durch HIPE stets ihren standardmäßig vorgegebenen Wert. Dies reduziert effektiv die Generierung einer Zufallszahl mit den beschriebenen Methoden zum Table-Lookup. Mehrfache Aufrufe derselben Analyse liefern somit dieselben Ergebnisse; ein HIPE-Durchlauf ist stets reproduzierbar. Die Möglichkeit, die interne Variable benutzerdefiniert oder automatisch mit sinnvollen Werten belegen zu lassen, ist nicht in HIPE enthalten.

- HIT stellt die so genannte *Aggregation* zur Verfügung, mit der Teile des Systems, deren Verhalten bezüglich der Leistungsmaße bekannt und/oder momentan uninteressant ist, von der Analyse ausgeschlossen werden können. Sie können dann als eine Blackbox angesehen werden und somit lediglich als "Konstante" in die Analyse einfließen. Dieses mächtige Werkzeug, mit dem die Dauer der Analyse für große Modelle deutlich reduziert werden kann, steht in HIPE nicht zur Verfügung.

- Insgesamt bietet das in diesem Dokument manifestierte Vorgehen nur einen äußerst beschränkten Blick auf die Möglichkeiten der Leistungsbewertung mit HIT. Das Werkzeug verfügt über weitaus mehr Konstrukte zur Bewertung eines Leistungsmodells als ein UML-Diagramm für die Konstruktion eines solchen Modells an Vorlagen liefern kann. Als Beispiele seien hier die Verwendung von *Load Filtering Hierarchies*, die Definition eigener Leistungsmaße (*Streams*) und die acht weiteren Standard-Komponenten zur Beschreibung einer Maschine im Leitungsmodell genannt. Die Beschreibung dieser "Bausteine" einer Leistungsbewertung durch HIT erfordern statt dessen direkte Benutzereingaben, so dass die Analyse nur noch wenige Rückschlüsse auf die Qualität des Eingabe-UML-Diagramms zulassen würde.

- Die zuvor angestrebte Möglichkeit, durch farbige Akzente im UML-Diagramm eine Interpretation der Leistungsmaße in anschaulicher Form an den Benutzer zurückzuliefern, musste entfallen. Es ist nicht möglich, eine von MetaMill erzeugte XMI-Datei mit Farbinformationen für einzelne Diagrammelemente zu erweitern. Lediglich die Änderung bereits vorhandener Farbinformationen ist möglich. Es ist jedoch unzumutbar, jedem Element, an dem eventuell ein Ergebnis zurückgeliefert wird, vom Benutzer vor der Übergabe an HIPE einen Farbwert zuordnen zu lassen.

6 Fazit der PG Mitglieder

Dieses Kapitel enthält die persönlichen Stellungnahmen der PG-Mitglieder und des Betreuers zur PG. Sie stellen die Meinungen dar, die sich die Beteiligten über die PG im Laufe der PG-Semester gemacht haben und unterliegen keinerlei redaktioneller Prüfung, weder in inhaltlicher noch in sprachlicher Form.

Jürgen Mäter

Die PG 438 (HUML) hat bereits im Wintersemester 2003/04 und im Sommersemester 2004 an einem ähnlichem Projekt gezeigt, dass es prinzipiell möglich ist, in UML spezifizierte Softwareentwürfe in HIT-Leistungsmodelle, unter Einsatz compilergenerierender Techniken, zu transformieren und zu analysieren. Das entwickelte Konzept sowie das Produkt, HUML (HIT und UML), beschränkt sich auf UML 1.4 Aktivitäts- und Deployment-Diagramme. Es lag daher auf der Hand, in einer anschliessenden Projektgruppe Ähnliches auch für UML 2.0 und weitere Diagrammtypen zu konzipieren und zu entwickeln.
Als Veranstalter der Projektgruppe war mir von vornherein klar, dass ich keine Kenntnisse aus dem Bereich Leistungsbewertung von Rechensystemen von den potentiellen Projektgruppenmitgliedern erwarten konnte. Deshalb habe ich ein einführendes Kompaktseminar derart gestaltet, dass ein erster grober Überblick in dieses Gebiet erarbeitet werden konnte. Aus organisatorischen und terminlichen Gründen einiger Projektmitglieder musste das Seminar vor Ort stattfinden. Es hat sich hierbei mal wieder herausgestellt, dass universitätsinterne, einführende Seminare zwar zum Einstieg in unbekannten Stoff dienen können, aber nicht sonderlich zum Kennenlernen der Gruppenmitglieder untereinander beitragen. Auch was in neudeutsch als „corporate identity“ verstanden wird, hat sich während der gesamten Projektdauer nicht eingestellt.
Ein Lehrziel, das in einer Projektgruppe vermittelt werden soll, bildet die Organisation, Leitung und Koordinierung von Projekten, also das Projektmanagement. Die Meilensteinvorgaben der beiden Projektmanagern wurden jedoch zum grössten Teil von den Projektgruppenmitgliedern ignoriert, sodass es letztendlich zu erheblichen Terminüberschreitungen gekommen ist.
Als fortführende Einarbeitung, d.h. zur Vertiefung der im Kompaktseminar erworbenen Kenntnisse, folgte ein Modellierungspraktikum. Während der erste Teil des Praktikums so ausgelegt war, dass anhand einfacher Modelle die Grundlagen der Modellierung von Rechensystemen erlernt werden sollten, war im zweiten Teil die Anwendung der im ersten Teil erworbenen Kenntnisse und Fähigkeiten an ausgewählten größeren Modellen zu beweisen. Es hat sich bereits hier gezeigt, dass die Projektgruppenmitglieder wohl dieses Gebiet unterschätzt haben und sie ziemlich blauäugig an die Aufgabenstellungen gegangen sind. Selten wurde vertiefende Literatur "freiwillig" durchgearbeitet, so dass das Praktikum nur mit vielen Hilfestellungen absolviert werden konnte.
Vorbereitend für die sich anschliessende Analysephase sollte, in Form eines zweiten Seminars,

eine Vertiefung in die Themenbereiche UML 2.0, HIT und das HUML-Paradigma der Vorgängerprojektgruppe erfolgen. In den Projektsitzungen vor dem zweiten Seminar wurden vom Projektmanagement einschliesslich mir leider nur Fortschritts- und keine Inhaltskontrollen der Seminarthemen vorgenommen. Es stellte sich erst während des Seminars heraus, dass die UML-Gruppe nur einführende Literatur verwendet hatte und nicht die von mir bereitgestellte. So wurde das Thema Metamodellierung, MDA (Model Driven Architecture)-Techniken und entsprechende Transformationen völlig ausgelassen. Die HIT-Gruppe war hauptsächlich mit der Übersetzung der englischsprachlichen Dokumente befasst, sodass auch hier die eigentlich erforderlichen HIT-Kenntnisse nicht wesentlich vertieft werden konnten. Das HUML-Paradigma wurde zwar untersucht, aber man wollte ja selbst etwas viel besseres entwickeln. Nach dem Seminar war es leider zu spät, gerade die o.a. ausgelassenen Themen nachträglich zu behandeln, sodass mit dem augenblicklichen Wissensstand die Analysephase gestartet wurde.
Es wurde eine Anforderungsdefinition sowie ein Pflichtenheft erstellt, wobei allerdings seitens der Projektgruppenmitglieder leider nur Minimalanforderungen angestrebt wurden, weshalb das Gesamtkonzept, das HIPE-Paradigma, erst im zweiten Projektgruppensemester fertiggestellt werden konnte. Die Gruppe hat sich hierbei sehr schwer getan, da sie offensichtlich wissenschaftliches Arbeiten noch nicht gewohnt war und sich insbesondere mit dem Projektgruppenthema nicht identifizieren konnte.
Bei der Erstellung des Zwischenberichtes, obwohl frühzeitig initiiert, ergaben sich erhebliche Kommunikations- und Organisationsprobleme, d.h. dass Teile nicht rechtzeitig fertig wurden, bzw. deren Qualität zu wünschen übrig ließ. Es weckte den Anschein, dass sorgfältiges Arbeiten nicht gewohnt war und sich zu sehr auf Korrekturen meinerseits verlassen wurde. Es fehlte an Einsicht, dass mindestens ein Projektgruppenmitglied den Zwischenbericht im Ganzen inhaltlich überwachen sollte. Es wurden so von den Untergruppen nicht wechselseitig konsistente Kapitel geschrieben, die erst durch viel redaktionelle Arbeiten annähernd konsistent gemacht werden mussten. Erschwerend kam hinzu, dass ca. 70 Prozent der PG-Mitglieder der deutschen Sprache nicht ganz mächtig waren, so dass für die restlichen 30 Prozent enorme Korrekturarbeiten angefallen sind.
Die im zweiten Projektsemester begonnene Entwurfsphase verlief sehr schleppend, da die Projektgruppenmitglieder offensichtlich Angst vor einer umfangreichen Implementierung hatten, so dass erst von einem minimalen Funktionsumfang des zu entwickelnden Programmes ausgegangen wurde. Außerdem zeigte sich, dass nicht alle Projektgruppenmitglieder das zuvor entwickelte Konzept voll verstanden hatten. Es entstand dadurch ein weiterer Zeitverzug gegenüber den Planungsvorgaben.
Mit der Implementierung wurde bereits während der Entwurfsphase begonnen, was zur Folge hatte, dass der anfangs noch inkonsistente Entwurf die Arbeiten an dem Prototyp wesentlich verzögert hat. Dies hat meines Erachtens folgende Gründe, zum einen wurden weder MDA- noch compilergenerierende Techniken eingesetzt, weshalb die Paradigma-Gruppe die wesentlichsten und anspruchvollsten Arbeiten zu leisten hatte und sich noch zusätzlich um die Transformation kümmern musste, andererseit fehlte es an Sachverständnis und Programmierpraxis. Ausserdem wurde zuviel Zeit aufgebracht, um eine geeignete Entwicklungsumgebung auf den Notebooks der PG-Mitglieder festzulegen. Die isolierte Entwicklung der Programmmodule hat, aufgrund erheblicher Kommunikationsprobleme, zu Inkonsistenzen und damit automatisch zum weiteren Zeitverzug geführt.
Das Produkt HIPE hat den prototypischen Testeinsatz nicht ganz bestanden. Die Projektgruppe hat aber gezeigt, daß es möglich ist, ausgewählte UML-Konstrukte für eine Leistungsbewertung mittels HIT umzusetzen.

Der Endbericht zeigte anfangs die gleichen Probleme auf wie der Zwischenbericht. Erst nach einer radikalen Umstrukturierung und mehreren Überarbeitungen konnte er in der vorliegenden Form akzeptiert werden.
Zur Gruppenarbeit möchte ich nur ein paar globale Bemerkungen machen, da ich in den Untergruppensitzungen nicht eingebunden war. Die Ergebnisse der Untergruppen waren zum Teil von sehr unterschiedlicher Qualität. Es hat sich jedoch mal wieder gezeigt, dass die Untergruppen nicht über ihren Tellerrand hinaus und von den Produkten der anderen Untergruppen nur bis zu den gemeinsam festgelegten Schnittstellen geschaut haben, also nicht das Gesamtprodukt vor Augen hatten. Als Veranstalter muss ich mir ausserdem vorwerfen lassen, dass ich nach dem ersten PG-Semester nicht mindestens 5 PG-Mitglieder von der weiteren PG-Arbeit ausgeschlossen habe, da sie offensichtlich von dem PG-Thema überfordert waren und teilweise erhebliche Verständnisschwierigkeiten hatten. Zum Glück gab es auch weniger anspruchsvolle Tätigkeiten, die ihnen zugewiesen werden konnten.
Die Projektgruppe hat somit das Minimalziel erreicht. Ich hoffe, dass die Teilnehmerinnen und Teilnehmer für ihr weiteres Studium, insbesondere für ihre Diplomarbeiten, etwas über wissenschaftliches Arbeiten gelernt haben.
Ich wünsche allen für die Zukunft alles Gute und dass sie ihr Studium gut abschließen.

Ulhak Arslan

Die Projektgruppe war eine äußerst lehrreiche Erfahrung, wobei ich mir persönlich gewünscht hätte am Ende sagen zu können "Das war ein schönes und erfolgreiches Jahr, das bleibt mir in guter Erinnerung. Nun ja, es bleibt mir in Erinnerung und das Resultat ist zufriedenstellend. Ich hätte mir aber wirklich sehr gewünscht, abgesehen vom Thema und den daraus entstandenen Problemen, das wir in der Gruppe miteinander besser zu Recht gekommen wären. Irgendwie war der Wurm drin, was meiner Meinung nach damit zu tun haben kann, dass der Bildungsweg der meisten Gruppenmitglieder doch sehr unterschiedlich war. Vor allem nicht zu vergessen, die Kommunikationsschwierigkeiten, die die ganze Zeit präsent waren. Durch diese Spannungen, die wie ein Schatten über der PG lagen, war es selten möglich, eine angenehme und entspannte Arbeitsatmosphäre entstehen zu lassen. Diese und andere Faktoren führten bei wenigen PG-Mitgliedern unterm Strich dazu, dass sie demotiviert wurden. Durch diesen Produktivitätsverlust innerhalb der Gruppe war ein Outstanding-Resultat nach zwei Semestern kaum zu erwarten. Es schlich sich fortlaufend ein „free-riding-Effekt“ ein.

Eine weitere Schwierigkeit war meiner Meinung nach die von Anfang wenig vorhandene Transparenz bei den Kriterien zur Bewertung von Gruppenleistungen und der zu erreichenden Ziele. Zwar wurden am Anfang des ersten Semesters Aufgaben verteilt, wie die Seminarphase oder das Modellierungspraktikum, welche in meinen Augen eher eigenständige personenbezogene Übungen waren und weniger Gruppenarbeiten. Diese und andere Aufgaben wurden zum größten Teil in Kleingruppen gelöst, wobei der Informationsaustausch zwischen den Gruppen nicht immer optimal verlief. Zu Anfang des ersten Semesters wurden zwar zwei Gruppenteilnehmer für die Projektleitung ernannt, unter anderem meine Wenigkeit, die unter anderem die PG-Organisation geführt haben, aber diese Herausforderung führte in Anbetracht der schwierigen Situation des Öfteren zu Problemen. Ich hätte mir in dieser Frage persönlich mehr Unterstüt-

zung vom Gruppenleiter gewünscht.

Außerdem hätte ich es besser gefunden, wenn innerhalb der Gruppe Verhaltensweisen wie Disziplin, Pünktlichkeit oder Zuverlässigkeit noch ausgeprägter vorhanden gewesen wären, es wäre z.B. besser gewesen, vom ersten Tag an Störfaktoren wie „Handy Klingeln“ in Form eines Strafkatalogs unter Strafe zu stellen. Last but not least, die vorhandenen Sprachdefizite von einigen PG-Mitgliedern. Aufgrund solcher Erfahrungen würde ich persönlich als Gruppenleiter für zukünftige Projektgruppen mindestens genauso viel Wert auf Sprachkompentenzen legen wie auf fachliche Kompetenzen.

Abschließend möchte ich mich noch herzlichst bei unserem Gruppenleiter Jürgen bedanken, für sein Verständnis und seine endlose Geduld.

Ugur Aydin

Projektgruppe klingt eigentlich harmlos, ist es aber nicht. So, wie bei vielen anderen auch, gab es neun andere PG´s, in die ich lieber wollte, als in diese. Ich glaube, in unserer PG gab es ohnehin nur einen, der „freiwilling“ bzw. in erster Priorität an dieser PG teilnahm und der hat sich wohl vertan. Dazu kommt erschwerend hinzu, dass viele Teilnehmer der deutschen Sprache nicht ganz mächtig sind. Ein grosser Nachteil, wenn man bedenkt, wieviel Arbeit allein das dokumentieren mit sich bringt. Viele Teilnehmer waren auch gänzlich unerfahren, was die Softwareentwicklung betrifft. Ausserdem gab sich unser Betreuer auch keine grosse Mühe mit uns. Jedenfalls hatte ich diesen Eindruck. Sehr oft bekam man gesagt, was man falsch gemacht hatte, aber Hinweise, wie es richtig geht, kamen im Vorfeld kaum und es scheint jetzt so, als ob das absichtlich gemacht wurde. Wahrscheinlich, damit wir uns selbst darum kümmern oder vielleicht, weil wir einige Fehler machen sollen/müssen, um den gewünschten Lerneffekt zu steigern/erzielen. Falls es bei der PG darum ging, möglichst viele schwerwiegende Fehler zu machen, um dann daraus zu lernen, war sie (für mich und viele andere) ein zweifelloser Erfolg. Zu meinen Fehlern und was ich daraus gelernt habe, komm ich am Ende meines Fazits. Ich bin mir sicher, dass alle Teilnehmer viel aus dieser PG mitnehmen werden. Zu den obengenannten Problemen gesellten sich schon bald selbstgemachte. Diese finden aber auch in den Fazits der anderen Teilnehmer (Jürgens scheint mir sehr interessant) genügend Erwähnung und ich will sie hier nicht noch einmal aufzählen. Es gab für mich aber auch positive Erfahrungen. Es gab oft Situationen, die mich in meinen fachlichen und manchmal auch menschlichen Qualitäten bestätigten.
Was habe ich gelernt? Neben sehr viel UML, XML, TEX und JAVA/ECLIPSE habe ich an vielen Beispielsituation, in denen ich auch mal die Hauptrolle spielen durfte, gelernt, wie man sich in einer Gruppe von Gleichgestellten besser nicht verhält.
Ich wünsche allen Teilnehmern viel Glück und Erfolg im weiteren Leben!

> „ Wir, die guten Willens sind, geführt von Ahnungslosen, versuchen, für die Undankbaren das Unmögliche zu vollbringen. Wir haben so viel mit so wenig so lange versucht, dass wir jetzt qualifiziert sind, fast alles mit nichts zu bewerkstelligen. “

Dacheng Chen

Erste will ich mich alle unsere Gruppenmitglieder bedanken, besonders bei Olaf, als Gruppemanager hat er sehr viel getan für unsere Gruppe, z. B. bei der Organisation vom Projektverlauf, bei der Korrekturen vom Endbericht, bei dem Debuggern vom HIPE-Programm, usw..
Bei dem zweiter Phase von unser PG hat sehr schwer voran gelaufen, das Problem ist so, dass wir bei dem Konzeptentwicklung nicht die Grupperessourcen betrachtet, eigentlich bei unsere Gruppe verfügt nicht so viel Kapazität wie unsere Konzept erfordern sollen. Von diesem Grund haben wir weiter paar Monaten gesessen, um ganz Projekt fertig machen zu können. Es ist eigentlich sehr schwere, dass die Gruppenmitglieder, die nicht vorher kennt, die Ressourcen zu schätzen.
Als letzter Satz wünsche ich unsere Gruppenmitglieder einem sehr erfolgreichen Studium.

Zoi Choselidou

Bevor ich mit Aussagen wie: *Das letzte Jahr war sehr lehrreich* glänze, oder dieses Fazit dazu benutze, mit dem letzten Jahr *abzuschließen*, wollte ich lieber nichts schreiben. Persönlich wollte ich nicht in meinen Fazit Kritik üben, aber nach all dem was meine Kollegen geschrieben haben, sehe ich mich gezwungen etwas zu erwidern. Ich möchte ehrlich sein. Das letzte Jahr war sehr stressig. Vieles hätte anders laufen können und vieles ist gut gelaufen.

> „Rache ist eine Handlung, die man begehen möchte, wenn und weil man *machtlos* ist. Sobald aber dieses Gefühl des Unvermögens beseitigt wird, schwindet auch der Wunsch nach Rache“[1].

Beinahe jedes Mitglied hat aktiv am Projektthema gearbeitet. Jetzt zum Abschluss unserer Arbeit wurde für mich deutlich, was wir alle gemeinsam geleistet haben, trotz der meiner Meinung nach, eher schlechten Zusammenarbeit.

Die Kommunikation zwischen uns allen ließ viel zu Wünschen übrig. Viele Probleme wären mit ein Paar direkten Worten und Aussprachen aus der Welt geschafft worden.

Bereitschaft Aufgaben und Probleme eigenständig und eigenverantwortlich zu lösen zeigten wenige (oder zu spät), denn es bestand immer die Hoffnung, dass sich andere um die unerledigten Aufgaben kümmern und dass man sich hinter einen der „fleißigen“ verstecken könnte.

Sehr interessant ist auch die Variante, bei der Arbeit die Vorgaben und die Meinung der Mehrheit zu ignorieren und einfach das zu tun was man selbst für richtig hält, so dass wertvolle Arbeitszeit mit der anschließende Korrektur und endlosen Diskussionen verloren ging. Pünktlichkeit und Disziplin... Manch einer hat das von anderen gefordert und erwartet, konnte sich aber wohl nicht selbst daran halten...

Interessant ist auch über die Deutschkenntnisse und Eignung zum Studium anderer die Nase zu rümpfen, während man selber das Glück hatte in Deutschland geboren zu werden und nicht weiß, was es bedeutet in eine andere Sprache als in der Muttersprache zu studieren.

Viele beschweren sich im Fazit, über ein Paar Mitglieder, die nicht viel beigetragen haben.
Wenn man diesen Leuten von Anfang an gesagt hätte, wo sie stehen - nämlich dass sie nicht sehr viel Arbeitseinsatz zeigten und dass sie mit den Problemen und Schwierigkeiten allein fertig werden müssen - würde es für sie nur zwei Möglichkeiten geben: sich Mühe zu geben, oder weiterhin nichts zu tun. Bei der ersten Möglichkeit wären sie weiter gekommen, in der PG, im Studium. Bei der zweiten Möglichkeit, wären sie raus geflogen, schon im ersten Semester der PG.
Stattdessen, wurde viel Energie verschwendet, um ihnen bei ihren Aufgaben zu helfen, und bei den Sitzungen kein böses Wort gesagt. So lernten die „Schwachen“ nichts und wurden als Ballast durch die PG-Zeit getragen und die „Helfer“ lästerten im nach hinein. Für mich wurde

[1] Aus *Anleitung zum Unglücklichsein* [Watzlawick (1983)]

übrigens damit der Beweis erbracht, dass auch Männer intrigant hinter dem Rücken anderer sticheln ;-)

Jetzt, wo alles vorbei ist, ist es mehr als ungerecht diesen Personen die Scheine nicht mehr zu gönnen - denn wir sind alle selbst schuld.

Die Bereitschaft sich in etwas neues einzuarbeiten war öfters nicht vorhanden. Aufgaben wurden hin und her geschoben, unter der Begründung, dass man selbst nicht weiter kam und deshalb ein jemand diese Aufgaben übernehmen sollte, der sich mit der Thematik und Problematik auskennt. Am besten war es aber, ganz lange so zu tun, als beschäftigt man sich mit der Lösung der übernommenen Aufgabe, um anschließend, - lange nachdem etliche Fristen ohne Ergebnis verstrichen waren - das offensichtliche zuzugeben und die Abgabe der vorher mit Stolz übernommenen Aufgabe in die Wege zu leiten.

So konnten zwei-drei Monate PG locker überbrückt werden...*ROFL*

Das funktionierte auch umgekehrt gut: sich im Vordergrund zu stellen, Aufgaben übernehmen und geschickt an Andere verteilen, es aber so aussehen lassen, als würde man die Hauptlast der Arbeit tragen.

Das sind aber, aus Organisationstechnischen und -psychologischen Gründen nicht wünschenswerten Arbeitsgruppenleiter Qualitäten (und damit meine ich nicht Olaf oder Jürgen, eher die, die es gerne wären). Denn ein Gruppenleiter sollte die kompetenteste Person der Gruppe sein, nicht die Person, die meiste heiße Luft von sich gibt. Ein guter Gruppenleiter mus die meiste Arbeit leisten, dafür „darf“ er auch Arbeit verteilen.

Nicht das „schlechte“ Projektmanagement, vielmehr diese Missstände führten dazu, dass Meilensteine nicht eingehalten wurden und Deadlines einfach verschoben wurden, denn die Zeitplanung wurde eigentlich nie ernst genommen.

Bei der Projektgruppenarbeit handelt es sich, um eine „Lehrveranstaltung“. Demnach hab ich das „Recht“ Dinge nicht zu verstehen, oder Wissenslücken aufzuweisen, Fehler zu machen und das immer wieder. Letztlich entwickelt eine Gruppe eine Eigendynamik, die nicht vorhersagbar ist.

Martin Ebers

> „Omnia fieri possent.“ (Alles kann passieren.)
> *(Seneca, "Ad Lucilium Epistulae Morales")*

Diese PG habe ich nicht im Irrtum zu meiner ersten Wahl gemacht, sondern einem randomisierten Algorithmus vertraut: Nachdem ich alle bis auf drei PGs gemäß meines Interesses hatte ordnen können und die drei letzten für mich gleich (sehr) interessant waren, habe ich kurzerhand eine Münze geworfen und die letzten Einträge damit so sortiert, dass schließlich

Jürgens PG die höchste Priorität erhielt. Leistungsbewertung (oder Bewertung von Software im allgemeinen) ist tatsächlich ein ganz interessantes Thema :).

> „If we knew what it was we were doing, it would not be called research, would it?"
> *(Albert Einstein)*

Allerdings war es ein Nachteil, dass wir alle zu Beginn so gut wie keine Erfahrung mit der Leistungsbewertung von Software hatten. In früheren Vorlesungen hatte sicher jeder von uns mal gehört, dass gute Performance einer Software wichtig ist, und auch dass es effiziente und weniger effiziente Algorithmen gibt[2], vielleicht auch noch in etwa, wie Markovketten oder ereignisdiskrete Simulation funktionieren. Es gehört sicher nicht zu den Zielsetzungen einführender Vorlesungen, all zu sehr ins Detail zu gehen. Entsprechend war es das auch schon. Das Kompaktseminar, für das sich Jeder in ein Thema eingearbeitet hat, das für unsere spätere Arbeit möglicherweise wichtig werden würde, und das daran anschließende Modellierungspraktikum waren eine ganz gute Idee, aber sicher auch nicht ausreichend, um einen entsprechenden Zugang zum Thema zu finden. Entsprechend wussten wir auch bei der Vorbereitung unseres zweiten Seminars bei unserer Einarbeitung noch nicht genau, wohin die Reise gehen würde, so dass - wie Jürgen schon anmerkte - die Einarbeitungen dann doch nicht zielgerichtet genug waren.

> „Wer in [dieser] Arbeit ein eklektizistisches Sammelsurium erblickt, täuscht sich nicht. Der Grundgedanke, den [wir] dabei allerdings bemüht war[en] herauszuarbeiten, ist die Frage nach einer Lebenseinstellung oder Weltanschauung, die uns motivieren könnte, am Leben zu bleiben."
> *(Klaus Weyell, "Das Untier und seine Verantwortung")*

Zu Beginn des zweiten Semesters hatten wir beschlossen, auf den Einsatz eines Compilergenerators zu verzichten. Nachdem das Konzept erst einmal soweit "stand", erschien der Einsatz compilergenerierender Techniken als zu großer Aufwand für die relativ kleine Aufgabe, Annotationen in UML-Diagrammen zu parsen. Es gibt natürlich auch die große Sicht, eine Art "UML-Grammatik" zu definieren, so dass ein Graph relativ automatisiert hätte geparst werden können (vgl. [Petriu und Shen (2002)]). Solche Umsetzungen hätten aber nur unter Einschränkungen funktioniert. Deswegen wurde letztlich ein "Brute-Force-Ansatz" gewählt, mit dem Diagramme DFS-mäßig durchsucht werden (vgl. auch den SPT-Algorithmus in [Petriu (2001)]), wobei aber die wegen der Erfordernisse der Umsetzung nach HI-SLANG zahlreichen, teilweise erst sehr spät entdeckten Sonderfälle leicht eingepasst werden konnten.

Ein "Sammelsurium" ist dieses Projekt aber nicht nur wegen der zahlreichen Sonderfälle, sondern auch wegen vieler - wie wir fanden, teilweise ganz guter - Ideen, die wir gewissermaßen "am Wegrand" aufgetan haben, was etwa die Messung der Komplexität von Klassen anging.

[2] Man kann darum streiten, in wiefern in HIPE wirklich Algorithmen beurteilt werden könnten, da zum Beispiel kaum jemand in UML einen Algorithmus genügend detailliert "zeichnen" würde, so dass wir es stets mit einem Abstraktum zu tun haben, oder auch eine Angabe der Laufzeit eines Algorithmus ja eigentlich nicht im Sinne einer geforderten Bedienzeit erfolgt, sondern als Laufzeitordnung in Abhängigkeit von der Eingabegröße...

In manchen Ecken ging es auch zu sehr ins Detail, während an anderen Stellen ein bisschen mehr abstrahiert und damit allgemeiner gearbeitet hätte werden können.

> „Wenn [uns] das gelingt, so wäre für [uns] diese Einstellung nicht zuletzt von pädagogischer Relevanz, denn gibt es einen philosophischen Impfstoff gegen Gleichgültigkeit, so sollte dieser nach Möglichkeit schon ab der Grundschule verabreicht werden."
> *(Klaus Weyell, "Das Untier und seine Verantwortung")*

Gerade diese Erfahrungen offenbaren entsprechend eine gewisse Notwendigkeit, das Thema der Bewertung von Software auch in der Lehre stärker zu motivieren. Der Umfang der Aufgabe war auch schon ohne die notwendige Einarbeitung in das Thema "Performance Engineering" immens. Da ist sicher die eine oder andere Lehrveranstaltung zum Thema von Vorteil.

> „Das sogenannte *Fatum Stoicum* war nicht so etwas Düsteres, wie man es auszumalen pflegt; es lenkte die Menschen nicht von der Sorge um ihre Geschäfte ab, sondern strebte danach, ihnen mit Bezug auf die zukünftigen Ereignisse das Bewußtsein der Ruhe zu verleihen, vermöge der Erwägung der Notwendigkeit, die all unsere Sorgen und Kummer unnütz macht."
> *(Gottfried Wilhelm Leibniz, "Die Theodizee")*

Ich empfand es im übrigen tatsächlich als positiv, dass Jürgen uns im ersten Semester praktisch jeden Fehler erst einmal hat machen lassen, was etwa die Zeitplanung oder die dann doch streckenweise ziemlich laxe Fortschrittskontrolle anging, durch die wir im ersten Semester viel Zeit verloren haben, die dann am Ende nachgearbeitet werden musste. Allerdings war die wohl zugrundeliegende Einschätzung, dass wir daraus lernen würden, streckenweise ein bisschen zu optimistisch. Denn wir haben manch einen Fehler auch zweimal gemacht.

Im Nachhinein würde ich entsprechend sagen, dass an vielen Stellen die Zeitplanung und die Zusammenarbeit - besonders über Grenzen von Teilgruppen hinweg - hätten besser sein können. Grundsätzlich waren wir wohl alle bemüht, über den eigenen Aufgaben den Blick für das Ganze nicht zu verlieren. In der Tat hatten die Leute, die die meiste Programmierarbeit hatten, auch die meisten Bezüge über die Grenze ihres Paketes hinaus. Bei unserer besonderen Zusammensetzung war darüber hinaus die Schnittmenge mit denjenigen, die besonders viel Text zu verantworten hatten, relativ groß. Resultat war also eine Vielfachbelastung mancher Gruppenmitglieder, die sich wiederum stark auf das Zeitbudget ausgewirkt hat.

Trotz etwaiger Schwächen im Deutschen oder beim Programmieren haben sich aber alle Gruppenmitglieder gut eingebracht. Ich persönlich hätte mich gar nicht getraut oder wäre auch nicht fähig gewesen, ein Projekt in einer für mich fremden Sprache - ausgenommen Englisch - durchzuführen. Von etwaigen zusätzlichen Doppel- oder Dreifachbelastungen durch Studium, Beruf und/oder Familie gar nicht zu sprechen. Letztlich muss sich kein Gruppenmitglied darum ängstigen, dass es im Rahmen seiner Möglichkeiten zu wenig gearbeitet hätte.

> „It's a poor sort of memory which only works backwards."
> *(Lewis Carroll, "Alice in Wonderland")*

Ich möchte auch noch einen kleinen Kommentar zu Jürgens Rhetorik abgeben: Manchmal hat mich Jürgens Wortwahl schon in Angst versetzt. Mit dem Blick des erfahrenen Entwicklers hat er uns schon in der Entwurfsphase immer wieder die Auswirkung eines kleineren bis größeren Klopses für das Gesamtprojekt vor Augen geführt, wenn sich dieser bis zum endgültigen Produkt so durchzöge. Allerdings erschien mir diese Bewertung nicht immer konsistent zu sein. So führte einmal das Fehlen einer if-Anweisung, mit der in HIPE eine löserbedingte Anpassung vorgenommen wurde, dazu, dass "der gesamte Code total falsch" war - eine Bedienzeitanforderung im Umfang von `cox(cox(x,0.32),0.32)` Einheiten macht tatsächlich nicht wirklich Sinn, erfordert aber nur eine kleine Änderung im Quelltext. Jürgen machte noch zu zwei, drei anderen Gelegenheiten derartige gruselige Bemerkungen. Eben weil das so kleine Änderungen waren, frage ich mich natürlich - und ich bin da ausdrücklich "mit-gemeint" -, warum wir diese Änderungen nicht schon vorher gemacht hatten. Weitere Änderungsvorschläge und Tips, die von Jürgen nicht in dieser irritierenden Art kommentiert wurden, die im Nachhinein betrachtet ein, zwei andere Ecken und Kanten ausgeglichen hätten, fielen demgegenüber seinerzeit leider ein wenig unter den Tisch. Da waren wir also schon etwas "abgestumpft".

> „Never spend more than a year on anything."
> *(Donald Knuth, in: Donald Knuth et al., "Mathematical Writing")*

Insbesondere möchte ich mich aber bei Jürgen für seine Geduld bedanken, die wir in diesen knapp drei Semestern sicher ganz schön belastet haben.

Und Zoi, ein Letztes noch: Ich habe tatsächlich eine ganze Menge gelernt ;)...

> „...exitus alterius malus gradus est futuri."
> (Eines Übels Ende ist des nächsten Anfang.)
> *(Olympia Fulvia Morata an Celio Secondo Curione, 25. Juli 1554, in: "Orationes, Dialogi, Epistolae, Carmina")*

...Im Mittel möchte ich aber behaupten, dass uns das Projekt wenigstens in Bezug darauf einigermaßen gelungen ist.

Sadik Hamurcu

Ich hätte mich gefreut, wenn ich das PG Thema bekommen hätte, die mit dem höheren Priorität. Software Performance Engineering ist meine 9. Wahl gewesen. Es war aber trotzdem ein sehr Interessantes Thema. Wir hätten bisschen mehr Zeit bekommen können oder das Thema bisschen eingeschränkt bearbeiten müssen. So hätten wir viel erfolgreiche unsere PG abgeschlossen. Ich bin mir sicher dass diese PG eine sehr gute Erfahrung für spätere Berufsleben geworden ist. Ich danke allen fürs mit machen.

Olaf Hengesbach

Ein Zitat von Alfred Polgar[3] besagt:

[3] 1873-1955, östr. Schriftsteller u. Kritiker

> „Arbeit ist das, was man tut, damit man es eines Tages nicht mehr zu tun braucht."

So oder so ähnlich muss leider auch mein persönliches Fazit ausfallen, wenn ich an die PG-Arbeit denke. Ohne Zweifel ist die PG als Lehrveranstaltung eine sinnvolle Einrichtung, um erste Einblicke und praktische Erfahrungen bezüglich der alltäglichen Aufgaben und Probleme eines Informatikers im Berufsleben zu vermitteln. Für den Einzelfall dieser PG bleibt jedoch ein äußerst fader Beigeschmack. Nach einem vollen Jahr(!) PG-Arbeit ist der Umstand offensichtlich, dass das Thema der PG eine erheblich strengere Kontrolle der Aufnahmevoraussetzungen der Teilnehmer erfordert hätte, damit es sinnvoll im zeitlichen Rahmen hätte bearbeitet werden können. Da hilft es auch nicht, ein Einführungsseminar und ein Modellierungspraktikum zu veranstalten, wenn es als eine gängige Praxis angesehen wird, dass die darin formulierten Aufgaben fast kommentarlos gestellt werden, tatsächlich aber ohne Rücksprache mit dem Betreuer kaum zu lösen sind. Meine ganz persönliche Erfahrung war, dass sich trotz der entsprechenden Hingabe bei der Bearbeitung der Aufgaben das Verständnis des zugrundeliegenden Sachverhalts — wie es für die anschließenden Phasen des Paradigma- und Prototyp-Entwurfs benötigt worden wäre — nicht einstellen wollte. Ich wusste ja schließlich noch nicht, worum es in der PG wirklich gehen würde. Man bearbeitet also die Aufgaben soweit, bis man ein ähnlich richtig aussehendes Ergebnis wie das der Mehrheit der übrigen PG-Mitglieder vorliegen hat und geht zum nächsten Punkt über. Selbst wenn dieses Vorgehen und das Erreichen eines solchen minimalen Wissensstandes beim Modellierungspraktikum noch akzeptabel ist, so scheitert man doch spätestens, wenn sich die Gruppe in Eigenregie neue Aufgaben stellen soll und in die Phase der Erstellung eines Paradigmas und eines Konzepts für einen Software-Prototypen einsteigt. Ohne eine klare Vorgabe, wie die Arbeit der PG im Detail auszusehen hatte, sollte ein PG-internes Projektmanagement die Aufgaben der — aus meiner Sicht — zeitlich und fachlich völlig überforderten Gruppe delegieren, wobei anscheinend wieder kollektiv (ich inklusive) das „nur-die-aktuelle-Aufgabe-hinter-sich-bringen"-Schema angewandt wurde. Nicht zuletzt aus Zeitmangel war dies schließlich häufig die einzige Möglichkeit, mit den gestellten Aufgaben abzuschließen und im Entwicklungsprozess voranzukommen. Dies mündete dann zwangsweise in eine schlichtweg als chaotisch zu bezeichnende Implementierungsphase, in der sich die PG zwar große Ziele gesetzt hat, diese aufgrund der mangelhaften Kenntnisse, der quasi nicht vorhandenen Motivation und der schlechten Organisation aber nur halbherzig umsetzen konnte.

Natürlich bin ich als Mitglied des Projektmanagement-Teams mitverantwortlich für die miserable Situation, jedoch bleibt in einer Gruppe aus gleichberechtigten Mitgliedern als Mittel zum Zweck nur die Möglichkeit, Deadlines zu setzen und verbal auf deren Einhaltung zu bestehen, so dass der Verlauf der Arbeit nur besprochen, aber nicht im eigentlichen Sinn kontrolliert oder gar forciert werden kann. Kommt dann noch der Umstand hinzu, dass die Definition von „angemessenem Arbeitsaufwand" extrem unterschiedliche Interpretationen erfährt, bleibt eine unaufhaltsame Frustrationssteigerung nicht aus.

Als theoretischer Richtwert sind in der Diplomprüfungsordnung Kerninformatik (2001) für eine Projektgruppe acht Semesterwochenstunden angelegt. Auch wenn es wohl kaum ein sinnvolles Projekt gibt, dass in dieser Zeit mit bemerkenswerten Ergebnissen bearbeitet werden kann, so war die für diese Projektgruppe erforderliche Zeit für viele Mitglieder unmöglich in einen akzeptablen Rahmen zu pressen und ging teilweise — auch in der vorlesungsfreien Zeit — über

das Dreifache der Vorgabe hinaus(!). So zielt meine Kritik denn auch hauptsächlich auf das aus den Fugen geratene Kosten/Nutzen-Verhältnis bzw. das nicht vorhandene Gleichgewicht zwischen der Komplexität des PG-Themas und den zur Verfügung stehenden Fähigkeiten bzw. „Arbeitskräften". Eine Konsequenz aus diesem Problem war, dass sowohl im ersten als auch (und insbesondere) im zweiten PG-Semester die Leistungsbewertung von Software viel zu selten im Mittelpunkt des Interesses gestanden hat. Die PG musste sich jedes noch so kleinen aber ungleich mehr zeitraubenden (und meistens weit abseits des eigentlichen PG-Themas liegenden) Problems in Eigenregie annehmen und ist blind und zielstrebig in jede Bärenfalle getappt, die der Software-Entwicklungsprozess und das PG-Thema bietet. Interessanterweise wurde die Vorgänger-PG nach eigenen Angaben von denselben oder zumindest vergleichbaren Problemen geplagt. Von einer effektiven (geschweigedenn weiterentwickelten) Betreuung kann somit nach meiner Auffassung keine Rede sein. Auch das Argument, im wahren Berufsleben ginge es ähnlich zu, hat hier keine Gültigkeit. Im wahren Leben wäre jeder Vorgesetzte völlig überfordert, wenn er die Leistungen seiner derart inkompetenten Untergebenen vor seinem Chef rechtfertigen müsste. Als logische Ursache kommen dafür meist nur zwei Dinge in Frage: schlechte Auswahl der Mitarbeiter und/oder schlechte Führung. Bereits eine ausführliche Erklärung, wie ein Zwischen- oder Endbericht auszusehen hat, hätte wahre Wunder gewirkt und unzählige Korrekturen (sowohl am Inhalt als auch am Layout und der Struktur) wären vermeidbar gewesen. Zum Zeitpunkt der Erstellung dieses Fazits sind bereits mehrere Monate seit Ende des zweiten Semesters vergangen, in denen fast ausschließlich der Endbericht für die endgültige Abnahme aufbereitet wurde. Die so entstehende Verzögerung bei der Ausgabe des PG-Scheins (es wurde von Dezember gesprochen!) hat für viele PG-Mitglieder empfindliche Konsequenzen, unabhängig von ihrem Arbeitspensum.

— Update: Als hoffentlich letzte PG-bezogene Handlung muss ich meinem Fazit noch das heutige Datum hinzufügen: 1.3.2006. In vier Wochen wird das dritte Semester seit Beginn der PG vergangen sein. Gerade habe ich meine letzten Korrekturen am Endbericht abgeschlossen. *Frustration* ist mein persönliches Wort der Jahre 2005 und 2006. Wohl denen, die von diesen Erfahrungen verschont geblieben sind! —

Somit wird mir die PG leider nur als reines Hindernis auf dem Weg zum Diplom und als abschreckendes Beispiel für die Aufgaben und Verantwortlichkeiten im Projektmanagement in Erinnerung bleiben. Nur die vielen neuen sozialen Kontakte, die ich dank der PG knüpfen konnte, werden mir hoffentlich erhalten und ganz sicher positiv im Gedächtnis bleiben.

Taher Nasched

Also... Wir haben nun Ende November und ich schreibe mein Zitat. Ich denke diese Aussage alleine sollte den meisten bereits zu denken geben, eigentlich sollte die PG seit Juli beendet sein. Nun ja, aber verwunderlich ist das eigentlich nicht. Aber fangen wir von vorne an.
Anfangs noch höchstmotiviert und gespannt der Dinge, die da kommen sollten, begann vor ca. 18 Monaten unsere Projektgruppe, mit dem Ziel, ein Softwaremodell in einer frühen Entwicklungsphase zu bewerten. Um überhaupt einen Einstieg in das Thema zu bekommen, wurden von den Projektmitgliedern Vorträge vorbereitet und in einer zweitägigen Seminarphase präsentiert. Da wir darin natürlich alle sehr ungeübt waren, hat diese Seminarphase eigentlich nicht wirklich zu der Lösung unseres Problems beigetragen. Hier hat sich bereits gezeigt, dass

viele Teilnehmer wohl nicht die Motivation hatten, sich richtig in das Thema einzuarbeiten und somit versucht haben, etwas zu erklären, was sie selber gar nicht richtig verstanden hatten. Normalerweise sollte diese Seminarphase auch dazu dienen, sich in der Gruppe näher kennen zu lernen, idealerweise bei einem gemeinsamen Aufenthalt in einer Jugendherberge. Leider konnte so etwas bei uns nicht durchgeführt werden, da sich einige Gruppenmitglieder dagegen ausgesprochen hatten. Dieses führte dazu, dass die Seminarphase weder inhaltlich noch in sozialer Weise ein Erfolg war. Allerdings hat man dabei gelernt, sofern man sich Mühe gegeben hat, einen Vortrag vorzubereiten und diesen dann einer Gruppe von Leuten zu präsentieren.
Direkt im Anschluss begannen wir mit dem Modellierungspraktikum, wir bildeten Untergruppen, in denen Aufgaben gelöst werden sollten. Das einführende Praktikum ging noch relativ schnell und einfach voran, dass anschließende Lösen der Aufgaben stellte sich dann jedoch als sehr schwierig heraus. Dies lag nicht an den Aufgaben, sondern an der mangelnden Unterstützung des Betreuers bzw. der meiner Meinung nach nicht vorhandenen Einsicht, dass man sich nicht eben mal in einer Woche in ein so komplexes Werkzeug wie HIT einarbeiten kann. Allerdings muss man Jürgen zugute heißen, dass er, wenn man zu ihm gegangen ist, sich immer Zeit genommen hat, um einem zu helfen. Allerdings hatte man dabei häufig das Gefühl, als ob er einem Vorwürfe machen würde, so als ob man sich gar nicht mit der Sache beschäftigt hätte, was bei vielen Teilnehmern dazu geführt hat, dass sie anstatt zu fragen lieber alleine versuchten, das Problem zu lösen. Mit Aussagen wie „das steht doch im Handbuch“ war mir jedenfalls meistens nicht geholfen. Ich kann mich noch sehr gut daran erinnern, wie wir mehrere Stunden verzweifelt versucht haben, den „seed“ zu verändern. Gut, da stand was im Hislang Manual, aber wie zum Teufel macht man das dann in HIT??? Wenn man alleine für solche „Rumsuchereien“ in einer Gruppe von 12 Leuten mehrere Stunden braucht, muss irgendetwas falsch gelaufen sein. Ach ja... bereits hier zeigte sich, dass einige Gruppenmitglieder keine Motivation hatten, sich überhaupt mit dem Tool zu beschäftigen. Mit Worten wie „das verstehe ich nicht“ wurde sich herausgeredet und man saß alleine da, letztendlich arbeitete man mit Leuten aus anderen Untergruppen zusammen um überhaupt eine Lösung hinzubekommen. Denn das war letztendlich das, was wichtig war. Niemanden (Jürgen) interessierte es, wie die Lösung zustande gekommen war. Hauptsache, sie war da. Selbst wenn es bei der Präsentation offensichtlich war, dass der Vortragende gar keine Ahnung hatte, was er da vorträgt, hatte dieses keine ernsthaften Konsequenzen.
Anschließend sollten größere Vorträge vorbereitet werden, dafür hatten wir mehrere Wochen Zeit, die allerdings nicht wirklich effektiv genutzt wurden. Eine Zusammenarbeit fand eigentlich nur in der Abstimmung der Literatur statt. Kontrolliert wurde hier ebenfalls nichts. Um nicht ganz unvorbereitet zu sein, bestand ich darauf, dass wir uns 2 Tage vor der Präsentation dieses Zwischenergebnisses treffen sollten, um wenigstens einmal gemeinsam alles durchzugehen. Nur soviel... Von den 4 Leuten kamen 3, davon einer mit einem halben Vortrag und dem anderen musste man beinahe alles erklären, was er auf seinen Folien hatte, denn er hatte nichts davon verstanden.
Die anschließend zu schreibende Zusammenfassung für den Zwischenbericht lief ähnlich chaotisch, diejenigen mit den schlechtesten Deutschkenntnissen waren erstaunlicherweise die, die als erstes fertig waren.
Nun gut, so verlief das erste Semester bis zum Ende weiter und wir machten halt eben irgendwas, was wir vielleicht auch im zweiten Semester implementieren wollten. Allerdings hatte noch keiner ein wirkliches Konzept ausgearbeitet, noch waren wir in irgendeiner Weise konkret geworden. Es hieß immer, wir nehmen UML, das konvertieren wir nach XMI, dann nach HISLANG, dann bewerten wir das ganze und zeigen das Ergebnis an.

Ein Gruppenmitglied hat sich sehr viel Arbeit gemacht und versucht, weiche Maße zu bestimmen und dafür sehr viel Recherche betrieben. Nach seinem Vortrag hat er Jürgen gefragt, ob die getroffene Auswahl sinnvoll wäre. Woraufhin von Jürgen nur ein „Ich hab keine Ahnung von den weichen Maßen, hab ich noch nie gesehen“ kam... Jürgen hat uns eh viel zu lange einfach machen lassen, ohne irgendwie in das Geschehen einzugreifen. Aber hinterher beklagen, wenn's zu spät ist. So wurde beispielsweise bemängelt, dass wir keine Compilergeneratoren benutzt haben, denn schließlich hatten wir ja extra einen Vortrag dazu gehört. Allerdings hat er außer Acht gelassen, dass wir mehrmals in der Sitzung erwähnt hatten, dass wir keine Compilergeneratoren einsetzen wollen. Vielleicht wäre das besser gewesen, aber warum hat er uns das nicht vorher gesagt? Letztendlich war selbst den Vortragenden zum Thema nicht klar, wie wir bei den Bewertungen, die wir durchführen wollten, einen Compilergenerator einsetzen könnten. Und das lag meiner Meinung nach nicht daran, dass er nicht verstanden hatte, worum es geht, sondern wirklich daran, dass sie für unsere Zwecke nicht geeignet sind. Irgendwie hatten wir sowieso das Gefühl, dass das ganze nicht wirklich gut durchdacht war. Besonders prägend fand ich das Beispiel, das Jürgen uns gegeben hat, damit wir die minimale Funktionalität unseres Entwurfs testen konnten. Mit diesem Beispiel hatte er wohl der Vorgänger-PG gezeigt, dass sie die Aufgabe nicht korrekt gelöst hatten. Ich hab überlegt und überlegt bis mir letztendlich bei einer Diskussion mit einem anderen PG-Teilnehmer des Rätsels Lösung kam. Das, was Jürgen laut dieses Beispiels bewertet haben wollte, war nicht das Softwaremodel einer zu implementierenden Software, sondern eine Bewertung des Metamodels (also des Problems, für dessen Lösung die Software modelliert wurde). Hatten wir also über 8 Monate in die falsche Richtung gearbeitet? Nach Rücksprache mit Jürgen stellte sich dann heraus, dass es sich wohl um einen „Schnellschuss“ gehandelt habe und das Beispiel falsch wäre. Aber das Beispiel gab es doch bereits seit über einem Jahr... Schnellschuss ???
So kam es also, dass alles immer chaotischer wurde und die Arbeit letztendlich nur noch von wenigen Leuten gemacht wurde. Viele fühlten sich nicht zuständig oder redeten sich mit dem bekannten „das verstehe ich nicht“ raus. Unserer Projektmanager fühlte sich total überfordert. Aufgaben, die er verteilte, wurden oft nicht bearbeitet und Deadlines fanden sowieso keine Beachtung. Dieses führte soweit, dass er kurz davor stand, das Management an eine andere Person abzugeben. Noch heute bereut er, dass er es damals nicht getan hat.
Das einzige, was viele Leute doch noch dazu gebracht hat, etwas zu tun, war, wenn Jürgen in das Geschehen eingegriffen hat, was er leider viel zu selten getan hat. Oft haben wir uns gefragt, wozu wir überhaupt einen Gruppenleiter haben, wenn sich seine hauptsächliche Arbeit darauf beschränkt, die Berichte auf Fehler zu durchsuchen. Letztendlich ist er der einzige, der ein Druckmittel in der Hand hält, um die Leute zum Arbeiten zu bewegen. Nun ja, so kam es, dass wir, ohne ein vernünftiges Konzept entwickelt zu haben, ins zweite Semester starteten. Die offiziellen Treffen fanden nun nur noch einmal die Woche statt, die inoffiziellen dreimal. Da diese treffen keinerlei Kontrolle von Jürgen unterlagen, saß man meistens nur in kleiner Runde von 3-4 Leuten da. Selbst für eine Gruppe von 12 Informatikern wäre die gestellte Aufgabe schwer zu lösen gewesen, in dieser kleinen Gruppe allerdings ein Ding der Unmöglichkeit. Problematisch auch deshalb, da eigentlich keiner der Gruppenmitglieder ausreichend Programmiererfahrung hatte. Jürgen meinte zwar, dass das jeder können müsste, man hätte ja schließlich SOPRA gemacht. Allerdings müsste es sich eigentlich mittlerweile auch bis zu den Lehrenden herumgesprochen haben, dass es immer nur einige wenige sind, die programmieren und die anderen sitzen daneben und gucken Löcher in die Luft. Also kurz gesagt, die Anforderungen waren viel zu hoch und die Aufgabe für die Menge an Leuten, die wirklich gearbeitet haben, unlösbar.

Irgendwie haben wir es aber hinbekommen, etwas auf die Beine zu stellen, das ein UML-Diagramm als Eingabe bekommt und auch eine sinnvolle Bewertung durchführt. Ich denke mal, dass wir bei den Bedingungen, unter denen wir arbeiten mussten, wirklich stolz sein können, dass wir das geschafft haben. Im Grunde genommen ist es leider egal, denn es interessiert wirklich niemanden, was wir da gemacht haben. Die Krönung fand ich persönlich, dass ein Professor noch vor dem Ende unserer Abschlusspräsentation den Raum verlassen hat. Gut, die Präsentation war vielleicht nicht gerade ein Glanzstück, aber es steckt von einigen Leiten sehr viel Arbeit in diesem Projekt und dem sollte man doch ein wenig Respekt entgegenbringen.
Einige wenige haben wirklich Hervorragendes geleistet, diesen Leuten möchte ich hiermit meinen Dank und Anerkennung aussprechen. Unter diesen Bedingungen hatte ich ehrlich gesagt nicht mehr daran geglaubt, dass wir überhaupt noch etwas Sinnvolles hinbekommen, aber wir haben es geschafft. Ich werde mich noch lange an die Zeit im Pavillon erinnern, wo viel gestritten, diskutiert und gelacht wurde.

Special thx to...
Olaf „Seit beginn der PG hat das leben sowieso keinen Sinn mehr“
Martin „Im Grunde genommen...“
Alexei „Ich muss das noch Debuggen“
Zoi „Jürgen sagt der Endbericht läßt sich nicht übersetzen“

Delice Jussen

Entscheidend hier für war aber wohl auch, dass das Leistungsniveau der Teilnehmer unterschiedlich ist. Auf Grund der Unterschiedlichkeit hat die Zeitplanung in der zweite Semester nicht gut funktioniert. Die Zuteilung von Rollen war in den meisten Fällen sinnvoll. Ich habe während der Projektzeit viel Neues gelernt, sowohl die Kenntnisse in HIT, die Erstellung der GUI mit Eclipse, als auch die Art und Weise in Team zu arbeiten. Ich bedanke mich bei allen, die mir bei Problemen geholfen haben und wünsche allen ein erfolgreiches Zukunft.

Deniz Kayar

Ja, auch ich habe einiges gelernt....
Neben handwerklichen Dingen wie Latex, Java, und Eclipse steht eine Erkenntnis besonders im Vordergrund. Nämlich jene, dass das Konzept „Projektgruppe“, wie es in vorliegender Form von den Verantwortlichen präsentiert wird, reine Schikane ist und zum Scheitern verurteilt ist. Wenn man schon dazu gezwungen ist, ein Jahr lang mit irgendwelchen Leuten, von denen einige möglicherweise schon mit einem Volkshochschulkurs endlos überfordert wären, irgendein Thema zu erforschen, welches man eher weniger priorisiert hat und dessen Sinn äußerst fragwürdig ist, dann sollte man für diese Qual bezahlt werden und nicht zu knapp.
Wenn ich im nachhinein von Mitstudenten höre, dass in ihre PG, welche von mir favorisiert wurde, sogar Leute ohne Vordiplom aufgenommen wurden, dann platzt mir die Hutschnur. Auch gibt mir die Tatsache zu denken, dass diese unsere PG solch eine hohen Prozentsatz an Ausländern bzw. nicht deutschstämmigen aufweist. Gerade mal zwei von zwölf Mitgliedern ha-

ben die von den anderen Projektgruppenbetreuern anscheinend schwer bevorzugten deutschen Wurzeln. Anders kann ich mir nicht erklären, dass ich und meine Projektgruppenmitglieder wie ein Stück Müll in diese äußerst unattraktive Projektgruppe abgeladen wurden.
Nun könnte man mir ja vorwerfen, dass ich auf die nächste Projektgruppenperiode hätte warten können. Ich hätte mich also erneut der Willkür von irgendwelchen frustrieten - zumindest erscheinen mir viele der hier in Dortmund herumgeisternden Gestalten so - Informatikern aussetzen sollen und möglicherweise erneut zu meiner Ernüchterung feststellen sollen, dass sich nichts geändert hat? Ich hätte meine Studium unnötigerweise um ein weiters Semester verlängern sollen? Nein Danke! Es könnte ja auch der Verdacht aufkommen ich sei für ansprechendere Projektgruppen nicht ausreichend qualifiziert gewesen. Und wenn schon! Ich finde, dass ein Informatikfachbereich, der mit einer in Deutschland einmaligen „Spezialität", nämlich der Projektgruppe, wirbt, dazu verpflichtet ist, jedem Studenten eine möglichst erträgliche PG zuzuteilen.
Herzlichen Dank.

Gulnara Ortlieb

Diese PG war meine 6. Priorität. Aber es ist egal, in welche PG du bist. Weil es überall viel gelernt wird: im Team arbeiten, mit einander kommunizieren, sich an die Leute anpassen. Ich habe mich während der PG-Zeit gute Leute kennen gelernt. An allen möchte ich Danke sagen. Besonders an Ulhak, Olaf, Martin und Dacheng. Und auch eine Zitate dafür: „Wer fragt ist ein Narr für fünf Minuten, wer nicht fragt - für ewig."

Alexei Sacharow

Schmeiß ein kleines Kind ins kalte Wasser und sieh zu wie es lernt zu schwimmen. Ungefähr so kann man die PG als Lehrnveranstaltung beschreiben. Ich hätte etwas mehr Unterstützung von Jürgens Seite erwartet. Denn wir waren die meiste Zeit total überfordert. Das Thema der PG war auch zu mächtig und zu kompliziert für zwei Semester. So war auch die Planung zu optimistisch. Die Seminarphase und Modelierungspraktikum könnten kompakter ausfallen, dafür hätte man für die Implementierung mehr Zeit einteilen sollen. Aber so lernt man das und das nächste mal würden wir das bestimmt auch viel besser machen.

7 Ausblick

Das in diesem Dokument beschriebene Konzept des Leistungsbewertungstools HIPE ist in vielerlei Hinsicht unvollständig. Sowohl im Paradigma selbst als auch bei der Programmgestaltung sowie der Weiterverarbeitung und Ausgabe der Ergebnisse bieten sich vielfältige Verbesserungen und Erweiterungen an. Die vielen erstrebenswerten Möglichkeiten für einen Ausbau des Paradigmas wurden bereits in Abschnitt 1.2.9 (Seite 72) und im Handbuch (Abschnitt 5.7.2, Seite 208) beschrieben. Für die GUI lassen sich recht einfach ergonomische und kosmetische Verbesserungen finden. So bietet sich beispielsweise eine "Wizard"-ähnliche Benutzerführung mit entsprechend ausführlichen textuellen und graphischen Erklärungen und Beispielen an, um den Einstieg ins Software Performance Engineering zu erleichtern. Weiter wäre eine umfangreiche, graphische Möglichkeit des Vergleichs mehrerer Analysen bzw. Experimentserien wünschenswert, um den Verlauf der Verbesserungen (oder Verschlechterungen) am UML-Diagramm verfolgen zu können. Dies würde ebenso eine Ergänzung des Gesamtkonzepts um eine Datenbank erfordern, wie sie im ersten Ansatz vorgesehen war (siehe dazu den Zwischenbericht dieser Projektgruppe [PG459 (2005)]). Ebenfalls in diesem Ansatz enthalten war die Idee, sämtliche Modelle im *Performance Model Interchange Format* (PMIF, siehe [Smith und Llado (2004)]) zu verwalten. Die dadurch sichergestellte globale Transportabilität würde den Austausch mit anderen Leistungsbewertungs-Tools verfügbar machen.

Literaturverzeichnis

[Adve und Vernon 2004] Adve, Vikram S. ; Vernon, Mary K.: Parallel Program Performance Prediction Using Deterministic Task Graph Analysis. In: *ACM Trans. on Computer Systems (TOCS)* 22 (2004), Nr. 1, S. 94–136. – URL `http://www.cs.uiuc.edu/ vadve/Papers/detmodel.tocsfinal.pdf`

[Amdahl 1967] Amdahl, Gene: Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In: *AFIPS Conference Proceedings* 30 (1967), S. 483–485

[Antonioli und Pilz 1998] Antonioli, Denis ; Pilz, Markus: Analysis of the Java Class File Format / University of Zurich. 1998. – Forschungsbericht

[Augustin und Büscher 1982] Augustin, Reinhard ; Büscher, Klaus-Jürgen: Characteristics of the COX-Distribution. In: *ACM SIGMETRICS Performance Evaluation Review* 12 (1982), Nr. 1, S. 22–32. – URL `http://portal.acm.org/citation.cfm?id=1041818.1041821`

[Baumgartl 2004] Baumgartl, Robert: *Echtzeitsysteme.* Skript zur Lehrveranstaltung, TU Chemnitz. 2004. – URL `http://rtg.informatik.tu-chemnitz.de/obj.php/ebs-2up.pdf?id=189&mime=application/pdf`

[Beilner und Stewing 1987] Beilner, H. ; Stewing, F.: Concepts and techniques of the performance modelling tool HIT. In: *Proceedings of the European Simulation Multiconference.* Wien : Beilner, März 1987, S. 84–89

[Black 2000] Black, Liz: *Software Performance Engineering.* 2000

[Brown 1995] Brown, Kurt: *Goal-Oriented Memory Allocation in Database Management Systems.* Dissertation, University of Wisconsin Madison. 1995. – URL `http://citeseer.ist.psu.edu/brown95goaloriented.html`

[Büttner u. a. 1999] Büttner, M. ; Fricke, B. ; Klaassen, O. ; Nolte, S. ; Szczittnick, M. ; Stahl, H. ; Weissenberg, N.: *HI-SLANG Reference Manual for the Hierarchical Evaluation Tool HIT.* 3.6.000. Universität Dortmund, Informatik IV: , 1999

[Cameron und Sun 2003] Cameron, Kirk W. ; Sun, Xian-He: Quantifying Locality Effect in Data Access Delay: Memory logP. In: *17th International Parallel and Distributed Processing Symposium (IPDPS '03)* (2003)

[Carbone und Santucci 2002] Carbone, Massimo ; Santucci, Giuseppe: *Fast&&Serious: a UML based metric for effort estimation.* 2002. – URL `http://alarcos.inf-cr.uclm.es/qaoose2002/docs/QAOOSE-Car-San.pdf`

[Chung u. a. 1994] Chung, Jen-Yao ; Ferguson, Donald ; Wang, George ; Nikolaou, Christos ; Teng, Jim: *Goal oriented dynamic buffer pool management for data base systems.* 1994. – submitted 1995 to Workshop on Quality of Service in Open Distributed Processing

[Clement und Quinn 1993] Clement, Mark J. ; Quinn, Michael J.: Analytical Performance Prediction on Multicomputers. In: *Proceedings Supercomputing '93* (1993). – URL `http://citeseer.ist.psu.edu/clement94analytical.html`

[Emam u. a. 1999] Emam, Khaled E. ; Benlarbi, Saida ; Goel, Nishith: The Confounding Effect of of Class Size on the Validity of Object-oriented Metrics / National Research Council of Canada. 1999. – Forschungsbericht

[Espinosa u. a. 1998] Espinosa, Antonio ; Margalef, Tomàs ; Luque, Emilio: Automatic Performance Evaluation of Parallel Programs. (1998). – URL `http://www.cs.uiuc.edu/homes/snir/PPP/aop/automatic.pdf`

[Fenton und Neil September/October 1999] Fenton, Norman ; Neil, Martin: *A Critique of Software Defect Prediction Models.* September/October 1999. – URL `http://citeseer.ist.psu.edu/fenton99critique.html`. – IEEE Transactions on Software Engineering, Vol. 25, No. 5

[Goseva-Popstojanova und Trivedi 2000] Goseva-Popstojanova, Katerina ; Trivedi, Kishor S.: Failure Correlation in Software Reliability Models. In: *IEEE Transactions on Reliability* 49 (2000), Nr. 1, S. 37–48. – URL `http://www.csee.wvu.edu/~katerina/Papers/TR-2000.pdf`

[Grove 2003] Grove, Duncan A.: *Performance Modelling of Message-Passing Parallel Programs*, University of Adelaide, Australia, Dissertation, 2003. – URL `http://www.dhpc.adelaide.edu.au/reports/138/dhpc-138.pdf`

[Henkel u. a. 1993] Henkel, Jörg ; Benner, Thomas ; Ernst, Rolf: Hardware generation and partitioning effects in the COSYMA system. In: *2nd IEEE/ACM International Workshop on Hardware-Software Codesign* (1993). – URL `http://citeseer.ist.psu.edu/henkel93hardware.html`

[Hoeben 2000] Hoeben, Fried: *Using UML Models for Performance Calculation.* 2000. – URL `http://doi.acm.org/10.1145/350391.350410`

[Horgan u. a. 1995] Horgan, J. R. ; Mathur, Aditya P. ; Pasquini, Alberto ; Rego, Vernon J.: Perils of Software Reliability Modeling. (1995). – URL `http://citeseer.ist.psu.edu/horgan95software.html`

[Hunter und McLaughlin 2004] Hunter, Jason ; McLaughlin, Brett: *JDOM.* 2004. – URL `http://www.jdom.org/`

[IBM 2001] IBM: *Simplify XML programming with JDOM.* 2001. – URL `http://www-128.ibm.com/developerworks/java/library/j-jdom/`

[Jain 1991] Jain, Raj: *The Art of Computer Systems Performance Analysis, Techniques for Experimental Design, Measurement, Simulation, and Modeling.* John Wiley & Sons, Inc., 1991

[Jeckle 2004] Jeckle, Mario: *UML*. 2004. – URL http://www.jeckle.de/

[Joines u. a. 2002] Joines, Stacy ; Willenborg, Ruth ; Hygh, Ken: *Performance Analysis for Java WebSites*. 2002

[Jonkers 1995] Jonkers, Hendrik: *Performance Analysis of Parallel Systems: A Hybrid Approach*, Technische Universiteit Delft, Dissertation, 1995

[Kinnersley 2005] Kinnersley, Jonathan: *Jigloo SWT/Swing GUI Builder for Eclipse and WebSphere*. 2005. – URL http://cloudgarden.com/jigloo/

[Leung 1988] Leung, Clement H.: *Quantitative Analysis of Computer Systems*. John Wiley & Sons, Inc., 1988

[Martinka 1995] Martinka, Joseph J.: Functional Requirements for Client/Server Performance Modeling: An Implementation using Discrete Event Simulation. (1995). – URL http://www.hpl.hp.com/techreports/95/HPL-95-92.html

[Marwedel 2003a] Marwedel, Peter: *Embedded System Design*. Kluwer Academic Publishers, 2003

[Marwedel 2003b] Marwedel, Peter: *Rechnerarchitektur/Rechensysteme*. Skript zur Lehrveranstaltung, Universität Dortmund. 2003. – URL http://ls12-www.cs.uni-dortmund.de/edu/scripts-de.html

[Mason 2002] Mason, David: *Probabilistic Program Analysis for Software Component Reliability*, University of Waterloo, Ontario, Canada, Dissertation, 2002. – URL http://etd.uwaterloo.ca/etd/dmason2002.pdf

[MathWorks 2005] MathWorks: *SimuLink*. 2005. – URL http://www.mathworks.com/products/simulink/?BB=1

[MetaMill Software 2004] MetaMill Software: *MetaMill*. 2004. – URL http://www.metamill.com

[OMG 2005] OMG: *Unified Modeling Language*. 2005. – URL http://www.uml.org/

[OMONDO 2002] OMONDO: *EclipseUML*. 2002. – URL http://www.omondo.de/

[Oxenrider 2005] Oxenrider, Keith: *Performance Programming*. 2005. – URL http://sol-biotech.com/code/PerformanceProgramming.html

[Petriu 2001] Petriu, Dorina C.: *Layered Software Performance Constructed from Use Case Map Specifications*, Carleton University Ottawa, Canada, Diplomarbeit, 2001

[Petriu und Shen 2002] Petriu, Dorina C. ; Shen, H.: Applying the UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Specifications. In: *12th International Conference on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation, TOOLS 2002* (2002)

[Pezzè u. a. 1995] Pezzè, Mauro ; Taylor, Richard N. ; Young, Michal: Graph Models for Reachability Analysis of Concurrent Programs. In: *ACM Trans. on Software Engineering and Methodology (TOSEM)* 4 (1995), Nr. 2. – URL http://dx.doi.org/10.1145/210134.210180

[PG438 2004] PG438: *Entwicklung eines Performance-Moduls zur leistungsorientierten Software Entwicklung.* Projektgruppenbericht. 2004. – Universität Dortmund

[PG459 2005] PG459: *Software Performance Engineering - Entwicklung eines Modellkonverters zur quantitativen Bewertung von Software(modellen).* Zwischenbericht. 2005. – HIPE

[Piastowski 2003] Piastowski, Martin: HFC-Netze: Verkehrs- und Performanceanalyse / Institut für Nachrichtentechnik, TU Braunschweig. 2003. – Forschungsbericht

[Sahner und Trivedi 1992] Sahner, Robert A. ; Trivedi, Kishor S.: *SHARPE: Symbolic Hierarchical Automated Reliability and Performance Evaluator - Introduction and Guide for Users.* 1992. – URL `http://www.cs.uidaho.edu/~krings/CS449/sharpe_userguide.pdf`

[Siebert 2000] Siebert, Fridtjof: *Guaranteeing Non-Disruptiveness and Real-Time Deadlines in an Incremental Garbage Collector (corrected version).* 2000. – URL `http://www.fridi.de/rts/papers/ismm98_corrected.pdf`

[Smith 1990] Smith, Connie U.: *Performance Engineering of Software Systems.* Addison Wesley, 1990

[Smith und Llado 2004] Smith, Connie U. ; Llado, Catalina M.: *Performance Model Interchange Format (PMIF 2.0): XML Definition and Implementation Technical Report.* 2004. – URL `http://www.perfeng.com/papers/pmif2techrpt.pdf`

[Smith und Williams 2000] Smith, Connie U. ; Williams, Lloyd G.: Software Performance AntiPatterns. In: *Proc. Workshop on Software and Performance (WOSP)* (2000). – URL `http://www.perfeng.com/papers/antipat.pdf`

[Smith und Williams 2002] Smith, Connie U. ; Williams, Lloyd G.: New Software Performance Antipatterns. In: *Proc. Workshop on Software and Performance (WOSP)* (2002). – URL `http://www.perfeng.com/papers/moreanti.pdf`

[Smith und Williams 2003] Smith, Connie U. ; Williams, Lloyd G.: More New Software Performance Antipatterns. In: *Proc. Computer Measurement Group (CMG)* (2003). – URL `http://www.perfeng.com/papers/evenmore.pdf`

[Spaniol und Güneş 2001] Spaniol, Otto ; Güneş, Mesut: *Modellierung und Bewertung von Kommunikationssystemen.* Vorlesungsskript. 2001. – URL `http://stinfwww.informatik.uni-leipzig.de/~mai99haa/Downloads/Skripte/Alt/Kommunikationssysteme/Kommsys-Skript-mobko.pdf`

[Tanenbaum und Woodhull 1997] Tanenbaum, Andrew ; Woodhull, Albert: *Operating Systems - Design and Implementation.* Addison Wesley, 1997

[Tregunno 2003] Tregunno, Peter: *Practical Analysis of Software Bottlenecks*, Carleton University Ottawa, Canada, Diplomarbeit, 2003

[Vastola 1996] Vastola, Kenneth S.: Performance Modeling and Analysis of Computer Communication Networks. (1996). – URL `http://poisson.ecse.rpi.edu/ vastola/pslinks/perf/perf.html`

[Vrsalovic u. a. 1988] Vrsalovic, Dalibor F. ; Siewiorek, Daniel P. ; Segall, Zary Z. ; Gehringer, Edward F.: Performance Prediction and Calibration for a Class of Multiprocessors. In: *IEEE Trans. on Computers* 37 (1988), Nr. 11, S. 1353–1365. – URL http://doi.ieeecomputersociety.org/10.1109/12.8701

[W3.org 2004] W3.org: *XML*. 2004. – URL http://www.w3.org/TR/2004/REC-xml-20040204/

[Watzlawick 1983] Watzlawick, Paul: *Anleitung zum Unglücklichsein*. R. Piper GmbH & Co KG, 1983

[Weißenberg u. a. 1999] Weissenberg, N. ; Wilde, A. ; Müller-Clostermann, B. ; Shaban, S. ; Dittrich, W.: *HIT and HI-SLANG An Introduction*. Universität Dortmund, Informatik IV: , 1999

[Willig 1999] Willig, Andreas: A Short Introduction to Queueing Theory. (1999). – URL http://www.tkn.tu-berlin.de/curricula/ws0203/ue-kn/qt.pdf

[Xu u. a. 1996] Xu, Zhichen ; Zhang, Xiaodong ; Sun, Ling: Semi-empirical Multiprocessor Performance Predictions. In: *Journal of parallel and distributed computing* 39 (1996), S. 14–28. – URL http://citeseer.ist.psu.edu/95872.html

[Xylomenos 1999] Xylomenos, George: *Multi Service Link Layers: An Approach to Enhancing Internet Performance over Wireless Links*. San Diego, University of California, Dissertation, 1999

Abbildungsverzeichnis

Tabellenverzeichnis

8 Anhang

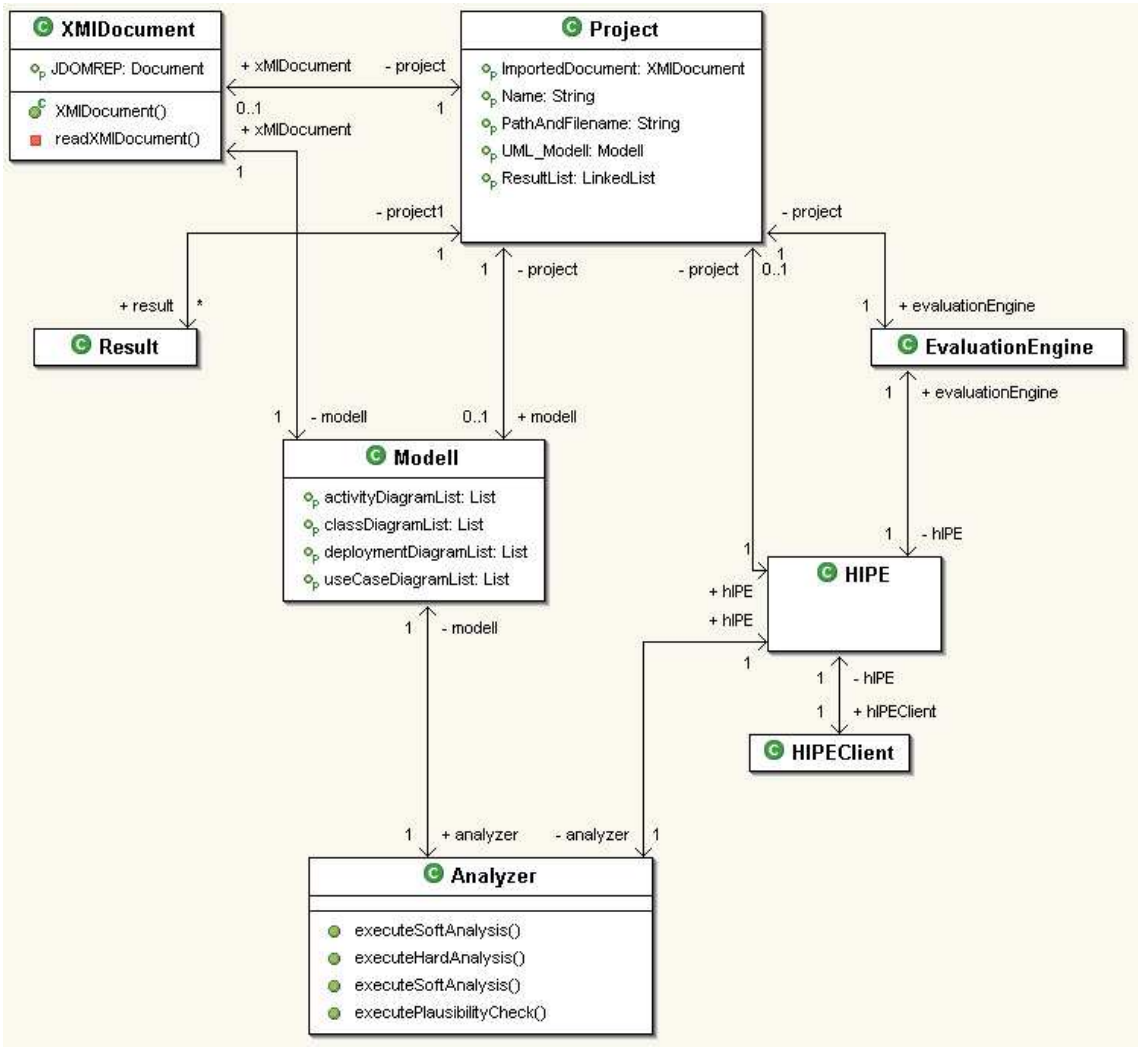

Abbildung 8.1: Das Haupt-Klassendiagramm für das Projekt HIPE

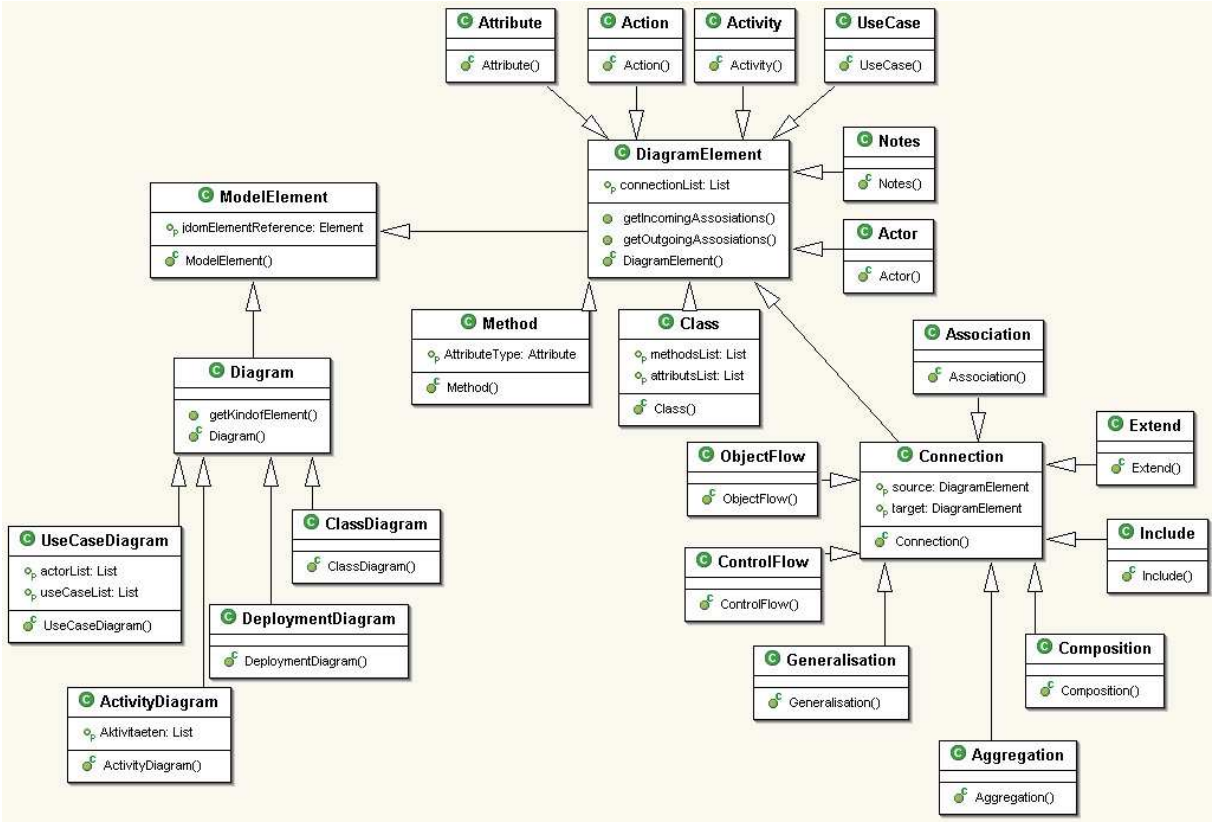

Abbildung 8.2: Das UML-Struktur-Diagramm um die Klasse ModelElement im Projekt HIPE

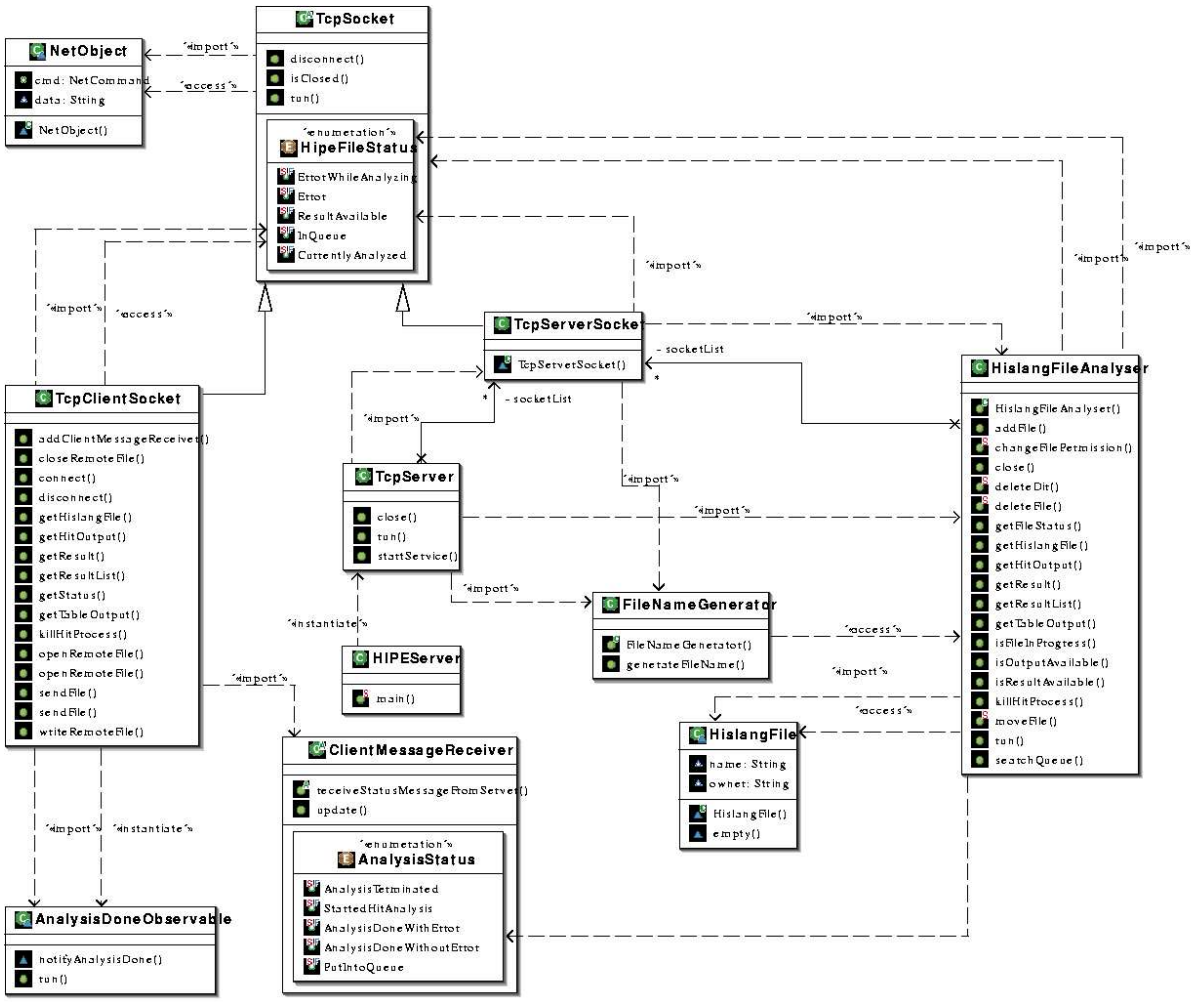

Abbildung 8.3: Das Klassendiagramm für den HIPE-Client/Server