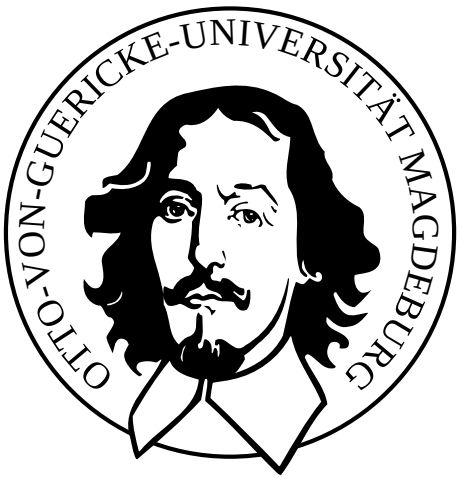
Aspektorientierte Realisierung eines generischen Systemmonitors

– Diplomarbeit –

Daniel Mahrenholz



„Otto-von-Guericke“ Universität Magdeburg
Fakultät für Informatik
Institut für Verteilte Systeme

Aufgabenstellung: Prof. Dr. Wolfgang Schröder-Preikschat
Betreuung: Dipl.-Inform. Olaf Spinczyk

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Betriebssysteme stellen eine Schlüsseltechnologie in der Informationstechnik dar. Dies gilt besonders für den Einsatz im Bereich der eingebetteten Systeme, da hier sehr spezielle und zum Teil sehr strenge Forderungen an das Betriebssystem gestellt werden. Gründe hierfür ergeben sich direkt aus dem Anwendungsumfeld. So werden heute in vielen Bereichen Mikrorechner als Ersatz für traditionelle (elektrische oder elektronische) Regel- und Steuerschaltungen verwendet. Man profitiert dabei davon, dass ein einmal entwickelter Mikrorechner durch Software (wozu auch das Betriebssystem zählt) an das Einsatzfeld angepasst wird.

Bei der Entwicklung von Betriebssystemen für den eingebetteten Bereich sind noch weitere Gesichtspunkte von entscheidender Bedeutung. So resultiert die enorme Verbreitung von eingebetteten Systemen zum einen in einer breiten Vielfalt von Hardware-Komponenten und zum anderen in vielfältigen Anforderungen seitens der Anwendungen. Somit sind für einen erfolgreichen Einsatz eines Betriebssystems eine möglichst große Hardware-Unabhängigkeit und Flexibilität von Bedeutung.[1]

Ein Beispiel ist das in der Arbeitsgruppe entwickelte PURE-Betriebssystem, das durch konsequente Nutzung der Methoden des familienbasierten Entwurfs und der objektorientierten Realisierung ein Höchstmaß an Flexibilität bei gleichzeitig sehr geringem Ressourcenverbrauch bietet [SSPSS98].

Für den praktischen Einsatz eines Betriebssystems im eingebetteten Bereich steht für den Anwender nicht unbedingt das Design im Vordergrund. Primär besteht er auf der Einhaltung der Vorgaben des Einsatzumfeldes. Die zum Teil sehr knappen Ressourcen sind zwar eine Herausforderung, schwerwiegender sind jedoch die Anforderungen an das Laufzeitverhalten. Bei vielen Regel- und Steuerungsprozessen kommt es auf die exakte Einhaltung verschiedener Zeitvorgaben an. Das Schlagwort der *Echtzeitfähigkeit* ist in diesem Zusammenhang in aller Munde.

Das Design eines Systems bestimmt sehr stark dessen Eigenschaften, zum Nachweis müssen diese jedoch messtechnisch erfaßt und ausgewertet werden. Dieser Vorgang wird

[1] Dies sollte eigentlich auch für die sogenannten *General purpose* Betriebssystem gelten.

als *Monitoring* bezeichnet. Ziel des Monitorings[2] eines Systems ist die Kontrolle und Überwachung des inneren Zustandes des Systems sowie die Protokollierung wesentlicher Zustandsänderungen im System. Die typische Vorgehensweise hierbei besteht darin, Zustände und Zustandsänderungen zu protokollieren und später auszuwerten. Dazu werden die interessanten Programmstellen (Funktionsaufrufe, Wertzuweisungen, Ein- und Austrittspunkte von Programmblöcken) um Programm-Code zur Protokollierung erweitert. Zur Messung des Zeitverbrauchs einer Funktion kann dann die Zeitdifferenz zwischen den passenden Ein- und Austrittspunkten bestimmt werden. Ähnliches passiert bei der Fehlersuche, bei der z.B. ungültige Zustände (Werte) von Variablen gesucht werden.

Die Verfahren zum Einbringen des zusätzlichen Programm-Codes sind recht unterschiedlich. Sie reichen von reiner „Handarbeit“ bis zu Werkzeugen, die an vorgegebenen Stellen Erweiterungen einfügen können. Diese Vorgehensweisen führen zu einer Reihe von Problemen beim Einfügen, Entfernen oder Warten des Monitor-Codes, wenn der Monitor-Code nur zeitweilig benötigt wird oder an eine geänderte Art der Protokollierung angepasst werden muss. Schwerwiegendere Probleme ergeben sich, wenn sehr viele verschiedene Monitoring-Funktionen zu verschiedenen Zeiten benötigt werden oder die Messpunkte für eine Aufgabe sehr stark von der Systemkonfiguration abhängen.

Nützlich wäre daher ein *generischer Systemmonitor*, der es erlaubt, alle interessanten Zustandsänderungen mit beliebigen Aktionen (z.B. zur Protokollierung) zu verbinden. Sowohl die Beschreibung der gesuchten Zustandsänderungen als auch die damit zu verbindenen Aktionen müssen vom eigentlichen System und speziell seiner Codebasis getrennt erfolgen, um diese nicht zu beeinträchtigen. Eine Möglichkeit dies zu realisieren ist die Nutzung des Konzepts der *Aspektorientierten Programmierung*, das im Abschnitt 2.2 näher erläutert wird.

1.2. Ziele der Arbeit

Eines der grundlegenden Ziele dieser Arbeit ist die Verallgemeinerung der zum Monitoring von Systemen eingesetzten Verfahren. Dies bedeutet insbesondere, dass die eingeführten Methoden sowohl von einem potentiellen Einsatzfeld als auch von einem zu untersuchenden System unabhängig sein sollen. Für die Praxis heißt dies, dass ein beliebiges Softwaresystem[3] nach allen Zustandsänderungen durchsucht wird und an jede einzelne eine beliebige Aktion gebunden werden kann. Dies muss nicht zwangsläufig bedeuten, dass diese Aktionen auch mit dem Einfügen von Programm-Code verbunden ist, es können z.B. auch nur die verschiedenen Aufrufe einer Funktion im Programm gezählt

[2] Es wird explizit davon ausgegangen, dass das Monitoring rein softwaretechnisch erfolgt. Die Möglichkeit, das Signalspiel der Hardware zu protokollieren wird nicht betrachtet.

[3] eingeschränkt auf die Programmiersprache C/C++

werden. Somit besteht auch keine Einschränkung auf die Verwendung zum Ausmessen des Systems, obwohl dies als Anwendungsbeispiel im Rahmen dieser Arbeit vorgesehen ist.

Zur Verallgemeinerung der Bindung von beliebigen Aktionen an verschiedene Zustandsänderungen ist eine ebenso allgemeine Beschreibungsform dieser Bindung notwendig. Die Frage nach dem, was getan werden soll, lässt sich im Allgemeinen recht einfach beantworten, da für eine Aufgabe (z.B. die Messung eines Zeitverbrauchs) an allen interessanten Punkten nahezu die gleiche Aktion erfolgt (z.B. Einfügen von Programmcode zur Speicherung eines Zeitstempels). Dagegen ist die Frage danach, wo die interessanten Punkte liegen, deutliche schwieriger. Für ein kleines System mit recht starren Strukturen können z.B. alle interessanten Funktionen einzeln aufgeführt werden. Dagegen ist dies für ein sehr umfangreiches System mit variablen Strukturen, wie dies bei PURE der Fall ist, nahezu unmöglich. Aus diesem Grund soll das System als eine Menge von *Komponenten* beschrieben werden, mit deren Hilfe wiederum die Frage nach dem Wo beantwortet werden kann. Hierzu mehr im Kapitel 3.

Als ein weiteres Ziel der Arbeit steht die Entwicklung einer problemadäquaten Benutzerschnittstelle, die es einem Nutzer gestattet mit einem Minimum an Einarbeitungszeit seine Monitoring-Aufgaben zu realisieren. Da sich derartige Aufgaben optimal mit dem Konzept der *aspektorientierten Programmierung* (s. Abschnitt 2.2) verwirklichen lassen, wird dieser Ansatz verwendet. Dazu soll aufbauend auf dem PUMA-Basissystem, das das Parsen von C/C++-Programmen sowie eine Reihe von Operationen auf den erzeugten Syntaxbäumen ermöglicht, eine Bibliothek von Funktionen zur Beschreibung von Komponenten, zur Analyse von Zustandsänderungen sowie zur Beschreibung und Durchführungen von Transformationen entwickelt werden.

Als letzte Zielstellung dieser Arbeit steht der Nachweis der praktischen Einsatzfähigkeit im Rahmen einer Anwendungsfallstudie zur Realisierung und Durchführung verschiedener Messungen am Beispiel des PURE-Systems. Dabei sollen auch verschiedene Formen der Beschreibung von Systemkomponenten demonstriert werden.

1.3. Gliederung

Die weitere Arbeit gliedert sich wie folgt:

Kapitel 2

gibt eine Einführung in das Problemgebiet und stellt die grundlegenden Konzepte vor, die im Rahmen dieser Arbeit von Bedeutung sind.

Kapitel 3

befasst sich mit Komponenten als Möglichkeit zur Gliederung und Beschreibung von komplexen Systemen sowie Verfahren zur automatischen Vervollständigung derartiger Beschreibungen.

Kapitel 4

erläutert die einzelnen Phasen der Realisierung eines Systemmonitors und die dabei beteiligten Systemkomponenten sowie die Erweiterungen zur praktischen Durchführung von Messprojekten. Daneben werden die Hilfsmittel zur Auswertung einer Messung vorgestellt.

Kapitel 5

demonstriert die Realisierung konkreter Monitoring-Aufgaben am Beispiel des PURE-Systems. Das Hauptaugenmerk wird dabei auf die messtechnische Bestimmung verschiedener Systemeigenschaften gelegt.

Kapitel 6

gibt einen Ausblick auf die Möglichkeiten der zukünftigen Verwendung und Weiterentwicklung des entwickelten Systems.

2. Grundlagen

2.1. Programmfamilien

Die im Rahmen dieser Arbeit entwickelten Methoden und Werkzeuge werden am PURE-System demonstriert. Ebenso werden alle Messungen an diesem System vorgenommen. Zum Verständnis der Besonderheiten des PURE-Systems und der damit verbundenen Schwierigkeiten eines klassischen Monitoransatzes (s. Abschnitt 1.1) ist die folgende Erläuterung des Konzepts der Programmfamilien bzw. des familienbasierten Entwurfes hilfreich.

Das Konzept der Programmfamilien wurde von Parnas vorgeschlagen.

> "We consider a set of programs to be a program family if they have so much in common that it pays to study their common aspects before looking at the aspects that differentiate them." [Par79]

An erster Stelle steht somit das Finden von Gemeinsamkeiten, erst danach werden die Unterschiede betrachtet. Für den Betriebssystementwurf bedeutet dies insbesondere, dass der Entwurf von einer *minimalen Basis* von Funktionalitäten ausgeht. Diese minimale Basis wird dann schrittweise um *minimale Erweiterungen* angereichert. Dieser Prozess setzt sich über mehrere Schritte kontinuierlich fort. Dabei ist zu beachten, dass man versucht, bindende Entscheidungen wie die Festlegung einer Strategie so weit wie möglich nach hinten zu verschieben, d.h. so spät wie möglich zu treffen, denn wichtiger sind die Mechanismen und Strukturen, die zur Realisierung verschiedener Strategien zur Lösung einer Aufgabe notwendig sind. Bestehen auf einer Ebene mehrere konkurrierende Möglichkeiten z.B. für die Realisierung einer Strategie, so resultiert dies in parallelen Erweiterungen einer Ebene („Geschwistern"). Diese Vorgehensweise ist in Abbildung 2.1 skizziert. Bei der Bestimmung der minimalen Erweiterungen ist darauf zu achten, dass diese zusammen mit der Basis aus Sicht der nächsten Schicht eine minimale Basis darstellen sollte.

Durch die konsequente Anwendung dieser Grundregeln entsteht nach und nach eine vielschichtige Hierarchie von Bausteinen unterschiedlichster, aufeinander aufbauender

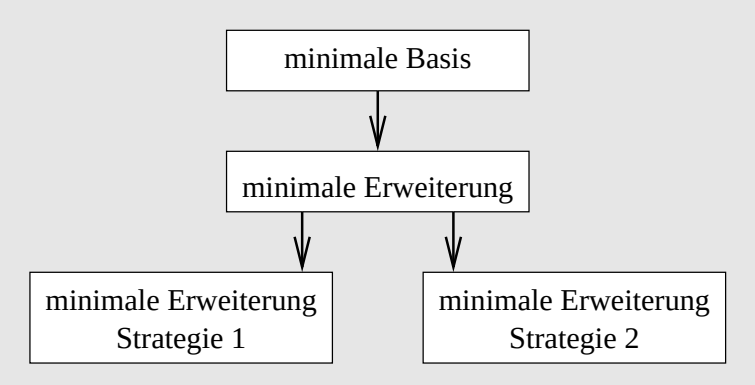


Abbildung 2.1.: Inkrementeller Systementwurf

Funktionalitäten. Dem Anwender steht es jetzt frei, die für seine Anwendung optimalen Familienmitglieder auszuwählen und um die Funktionalitäten seiner Anwendung zu erweitern.

Es ist dabei zu beachten, dass dieser Entwurfsansatz noch keine Vorschriften für die Implementierung beinhaltet. Als sehr vorteilhaft bei der Implementierung einer Programmfamilie haben sich die Methoden der objektorientierten Programmierung erwiesen. Grund hierfür ist die Möglichkeit, die Entwurfsprinzipien der Programmfamilien direkt auf ein Klassenmodell zu übertragen (s. Abbildung 2.2).

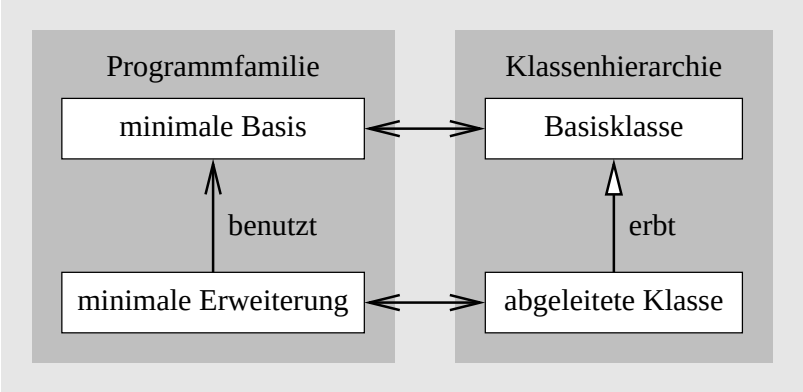


Abbildung 2.2.: Objektorientierte Implementierung von Programmfamilien

Wie in Abbildung 2.2 zu sehen, wird die minimale Basis in einer Klasse gekapselt. Die Erweiterung erfolgt dann im nächsten Schritt durch eine Vererbungsbeziehung. In die-

sem Zusammenhang ist zu erwähnen, dass es bei der praktischen Realisierung von Vorteil ist, eine Programmiersprache wie C++ zu verwenden, die die Möglichkeit der Mehrfachvererbung anbietet.

Für die praktische Anwendung bei der Betriebssystementwicklung ergeben sich aus der Nutzung des familienbasierten Entwurfs und der objektorientierten Implementierung eine Reihe von Vorteilen. So ist es möglich, das Betriebssystem in seiner Funktionalität sehr feingranular und somit hochmodular zu entwerfen. Dies führt bei einer sauberen, objektorientierten Umsetzung zu einem sehr schlanken System, das nur die wirklich benötigten Funktionalitäten enthalten muss – angesichts der Ressourcenknappheit im eingebetteten Bereich eine Voraussetzung für den erfolgreichen Einsatz.

2.2. Aspektorientierte Programmierung

In den letzten Jahren haben sich die objektorientierte Programmierung und die Methoden zum Entwurf von objektorientierten Softwaresystemen sehr weit entwickelt und sind zu einem Standard moderner Softwareentwicklung geworden. Auch sind moderne Compiler in der Lage, aus objektorientierten Programmen effizienten Programmcode zu erzeugen. Dies beseitigt eines der Hauptargumente gegen den Einsatz objektorientierter Progammierung in Bereichen, die auf sowohl speicherplatz- als auch laufzeiteffizienten Programmcode angewiesen sind, wie z.B. die Betriebssystementwicklung. In den in der Praxis eingesetzten Betriebssystemen findet die Objektorientierung zwar noch kaum Anwendung, entsprechende Entwicklungen sind aber im Gange. Ein Beispiel ist PURE, das zeigt, dass Objektorientierung und geringer Ressourcenverbrauch kein Widerspruch sein muss.

Trotz all dieser Fortschritte hat sich auch gezeigt, dass die objektorientierte Programmierung nicht das Allheilmittel der Softwareentwicklung darstellt. Ein Grund hierfür ist der Ansatz dieser Entwicklungsmethode.

Bei der objektorientierten Softwareentwicklung werden Systeme als eine Menge von Objekten modelliert, die jedes für sich Daten und die zu deren Manipulation notwendigen Funktionen kapseln. Diese Vorgehensweise ist meist sehr intuitiv – sicher mit ein Hauptgrund für die Durchsetzung der objektorientierten Programmierung. Betrachtet man nur den reinen Entwurf, so treten auch noch keine folgenreichen Probleme auf, da bisher nur rein fachliche Gesichtspunkte der Anwendung modelliert wurden.

Bei der Implementierung stößt man dagegen schnell auf eine Reihe nicht funktionaler, mehr technischer Anforderungen, auf die Rücksicht genommen werden muss. Einfache Beispiele sind Maßnahmen zur Fehlerbehandlung, der Debugging-Unterstützung und nicht zuletzt Mittel zum Messen oder Überwachen von Laufzeiteigenschaften. Darüber hinaus existieren eine Reihe von komplizierteren Systemeigenschaften, die nicht direkt

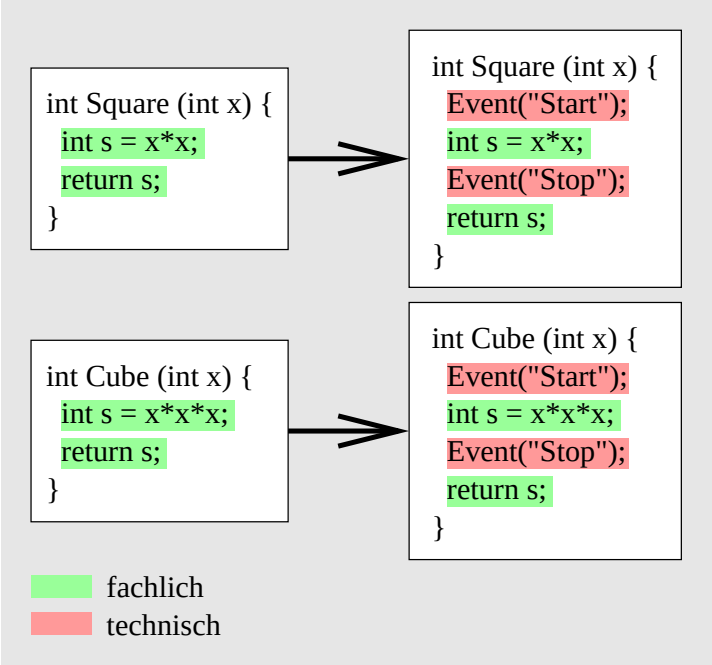


Abbildung 2.3.: Fachlicher und technischer Programmcode

zum funktionalen Teil des Systems gehören. Beispiele hierfür sind die Implementierung von Objektpersistenz, die Kommunikation zwischen entfernten Objekten, d.h. Objekten in getrennten Adressräumen oder auf getrennten Rechnern, und die Synchronisation paralleler Aktivitäten. Die Realisierungen dieser technischen Systemeigenschaften lassen sich selten vollständig in Klassen kapseln und hängen zuzätzlich noch stark vom konkreten Einsatzumfeld ab. Dies führt dazu, dass bei der Realisierung in nicht unerheblichem Maße in den rein fachbezogenen Implementierungsteil eingegriffen werden muss.[1]

Abbildung 2.3 zeigt ein sehr einfaches Beispiel für die Problematik. Im Beispiel soll der Zeitverbrauch zweier Berechnungen bestimmt werden. Dazu werden die Zeitpunkte vor und nach den Berechnungen als Ereignisse protokolliert. Dabei sind der jeweils fachliche und technische Programmcode unterschiedlich gekennzeichnet.[2] Es ist zu sehen, dass der technische Teil in beiden Fällen gleich ist. Es bleibt nur noch die Frage, wie und in welcher Form die Ereignisse selber erzeugt und gespeichert werden sollen – dies hat aber nichts mit dem eigentlichen Problem der Berechnung zu tun. Wir haben also zwei grundlegend verschiedene Teile des Programms zu modellieren, die sich im Programm-

[1] Dieser Vorgang wird auch als *code tangling* bezeichnet.
[2] Der fachliche Teil ist der, der das eigentliche Problem löst, d.h. die Berechnung ausführt. Der technische bzw. nichtfachliche Teil dient anderen Aufgaben wie dem Messen.

code zwangsläufig durchmischen. Diese Überlagerung zweier getrennt zu modellierenden Systemeigenschaften wird auch als *cross cutting* bezeichnet.

Nach [K+97] lassen sich die eigentlichen Programmkomponenten und Aspekte wie folgt definieren:

Komponenten sind die Teile des Systems, die sich sauber in Form verallgemeinerter Prozeduren (Objekte, Methoden, Prozeduren, APIs) kapseln lassen. Komponenten ergeben sich regelmäßig in Folge der funktionalen Dekomposition des Systems.

Aspekte sind die Teile des Systems, die sich nicht derart kapseln lassen.

Danach kann der oben als technisch bezeichnete Programmcode in Form eines Aspekts modelliert und nachträglich in das Programm eingebracht werden (s. Abbildung 2.4).

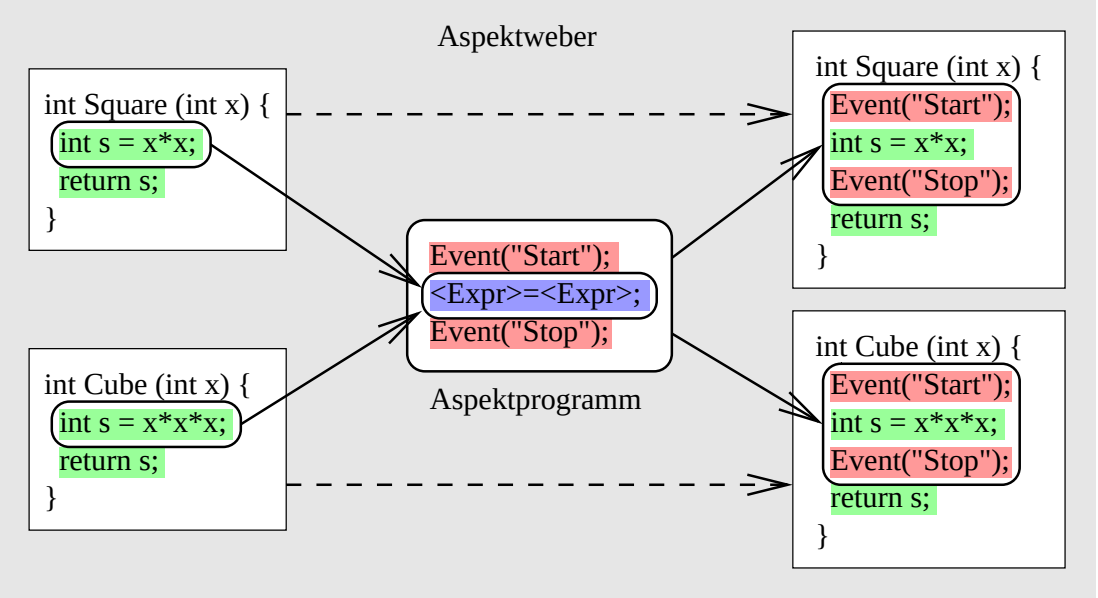


Abbildung 2.4.: Einweben von Aspektprogrammen

Fasst man das bisher dargestellte zusammen, so sind bei der aspektorientierten Programmierung (AOP) drei Bausteine zu beachten. Zum ersten ist da der Programmcode, der die rein fachlichen Aufgaben erfüllt. Daneben existieren die Beschreibungen, wie der Programmcode an verschiedene technische Anforderungen angepasst werden soll – die *Aspektprogramme*. Wie diese Programme aussehen, ist nicht festgelegt, sie stellen vielmehr eine Verfahrensanleitung für den dritten Baustein, den *Aspektweber* (oder einfach

nur kurz *Weber* genannt), bereit. Dieser Weber vollführt die eigentlichen Änderungen am ursprünglichen Programmcode – deshalb bezeichnet man den Vorgang auch als das *Weben* (s. Abbildung 2.4).

Das oberste Ziel der AOP ist somit die klare und saubere Trennung von herkömmlichen Programmkomponenten und Aspekten. Dies bedeutet insbesondere, dass beides getrennt voneinander modelliert und realisiert werden kann und Mechanismen bereitgestellt werden, um daraus das fertige System zu erzeugen.

Ein weiteres Beispiel, in dem das objektorientierte Modell bei der Implementierung unliebsame Grenzen offenbart, ist der Bereich kleverer Optimierungen.[3] Zur Verdeutlichung dieses Sachverhalts stelle man sich einen Baukasten modularer Funktionsbausteine vor. Die Bausteine dieses Baukastens werden entsprechend der gewünschten Systemfunktionen zu einem System zusammengesetzt. Diese Vorgehensweise hat den Vorteil, dass sie sehr flexibel ist und die einzelnen Bausteine aufgrund ihrer Einfachheit potentiell sehr gut wiederzuverwenden sind – ein Grundbestreben der objektorientierten Programmierung allgemein. Bei dieser Vorgehensweise wird man aber auch schnell auf einen entscheidenden Nachteil stoßen, speziell wenn die Funktionsbausteine nacheinander auf den selben, komplexen Datenstrukturen operieren. Hier ergeben sich meist aus der Betrachtung der konkreten Zusammensetzung der Bausteine Optimierungsmöglichkeiten, die sich quer durch die einzelnen Bausteine ziehen. Moderne Compiler können hier zum Teil wahre Wunder vollbringen, aber die Verfahren an sich können sie nicht optimieren, da ihnen das grundlegende Verständnis der Vorgänge fremd ist. Hier wird der Programmierer selber gefordert, wenn es auf ein effizientes Programmverhalten ankommt. Dabei wird aber meist die Kapselung durch die Objekte zu Gunsten der Optimierung durchbrochen, was eine Wiederverwendung fast unmöglich macht – eine scheinbar ausweglose Situation.

In einigen Fällen kann auch hier die aspektorientierte Programmiererung helfen. Die Forschungen auf diesem Gebiet stehen noch ziemlich am Anfang und Erfahrungen existieren nur sehr begrenzt, so dass das Einsatzpotential noch gar nicht richtig abgeschätzt werden kann. Eines kann aber bereits jetzt schon festgestellt werden, gelingt es, die Optimierungen, die man von Hand vornehmen würde, in irgendeiner Form formal zu beschreiben und dann automatisch durch ein Programm durchzuführen, so würde man dadurch die Möglichkeit erhalten, jede beliebige Bausteinkombination zu optimieren ohne die Wiederverwendbarkeit eunzuschränken.

[3] Ein Beispiel hierzu ist in [K+97] gegeben.

2.3. Instrumentierung

Da das Messen ein grundlegender Bestandteil der Entwicklung von Software ist, besteht auch schon länger der Wunsch, ein universelles Messwerkzeug zur Verfügung zu haben. Dabei lag ein Hauptteil der Konzentration bei der Erfüllung dieses Wunsches auf den Verfahren, mit denen die für die Messung notwendigen Coderweiterungen in das System eingebracht werden können (auch als *Instrumentierung* bezeichnet). Weit weniger Wert wurde dabei auf eine universelle Form zur Beschreibung einer Messung gelegt. Es existieren eine Reihe von Systemen zur Instrumentierung von Programmsystemen (siehe unten), aber keines davon bietet eine den angebotenen Instrumentierungsmöglichkeiten adequate Beschreibungsform, d.h. es werden zwar Möglichkeiten zur Veränderung eines beliebigen Systems aber keine passende, allgemeingültige Beschreibungsform angeboten. Die aspektorientierte Programmierung schafft hierbei in vielen Fällen Abhilfe. Sie bietet die Möglichkeit, die für eine Messung notwendige Instrumentierung allgemeingültig zu beschreiben, und hilft somit, das ganze Potential der bisherigen Ansätze voll auszuschöpfen.

Zur Verdeutlichung der Möglichkeiten bisheriger Systeme zur Instrumentierung werden im Folgenden die verschiedenen Verfahren zur Instrumentierung und ausgewählte Systeme vorgestellt.

Zur Instrumentierung eines bestehenden Systems sind zwei grundsätzliche Methoden vorstellbar. Die erste ist für die Fälle geeignet, in denen z.B. die Aufrufe von Teilen einer Programmbibliothek protokolliert werden sollen. Dabei wird die Instrumentierung unter der Schnittstelle der Bibliothek verborgen. Hierfür sind zwei Varianten möglich. Zum ersten können alle potenziellen Mess- bzw. Protokollpunkte explizit im Programm vorhanden sein und in Abhängigkeit von weiteren System- oder Konfigurationsparametern die Aufzeichnung von Ereignissen auslösen. Realisiert werden kann dies je nach Programmiersprache über die Abfrage eines gesetzten Parameters, durch geeignetes Setzen eines Funktionszeigers bzw. durch die Verwendung von virtuellen Methoden in objektorientierten Programmen. Von dieser Möglichkeit macht z.B. das *m4cp*-Tool [Ger94] Gebrauch. Vorteilhaft ist diese Methode aber nur, wenn von den potenziellen Messpunkten auch der Großteil aktiv ist oder sich die Messpunkte zur Laufzeit verschieben, da sonst sehr viel Zeit unnütz verbraucht wird, um festzustellen, dass nichts zu tun ist. Zum zweiten besteht die Möglichkeit, verschiedene Varianten der verwendeten Bibliothek bereitzustellen. Anwendung findet dies z.B. bei den Standard-C/C++-Bibliotheken, die parallel auch in einer zum Debugging und Profiling tauglichen Variante vorliegen. Aber auch diese Vorgehensweise ist nur sehr begrenzt tauglich, wenn es darum geht, das Gesamtsystem zu untersuchen. Auch ist sie sehr unflexibel.

Die zweite grundsätzliche Methode besteht darin, den Programmcode direkt zu verändern. Hierbei erzielt man je nach Vorgehensweise ein Höchstmaß an Flexibilität. Auch in diesem Fall sind mehrere Varianten zu betrachten. Zum ersten kann das Programm in

seiner Quelltextform und zum zweiten in seiner ausführbaren Form[4] manipuliert werden.

Die erste Variante – die Manipulation des Programmquelltextes – kann auf verschiedene Arten realisiert sein. Zum einen kann die Aufgabe durch einen typischen Präprozessor, der eine Menge von vorgesehenen Messpunkten passend ersetzt oder entfernt, erfüllt werden. Zum anderen aber auch durch einen Compiler, der das Programm in der Quellsprache analysiert, den Systaxbaum entsprechend manipuliert und das Resultat in der Zielsprache, die mit der Quellsprache identisch sein kann, speichert. Die Verwendung eines Präprozessors schränkt die Flexibilität sehr stark ein, da auch hier alle potenziellen Messpunkte bereits vorher im Quelltext vorgesehen sein müssen. Ein Compiler ermöglicht die detailierte Kontrolle des Manipulationsprozesses – aus diesem Grund wird dieser Ansatz auch im Rahmen dieser Arbeit verwendet. Auch im *AspectJ* [Xer99] wird dieser Ansatz verwendet. Hier bildet eine erweitere Java-Version die Quellsprache, die Zielsprache ist das Standard-Java.

Die zweite Variante ist die Manipulation des ausführbaren Programmcodes. Hierbei werden entweder Teile des Programms durch neue Teile oder durch Umleitungen zu Erweiterungen ersetzt. Diese Vorgehensweise findet seit vielen Jahren in Debuggern Anwendung, da viele Prozessoren keine Hardwareunterstützung für das Setzen von Breakpoints bieten. *KernInst* [TM98] zeigt die Anwendung dieses Verfahrens zur Instrumentierung des Solaris-Kernels. Die Vorteile dieses Verfahrens sind die Unabhängigkeit vom Quellcode des Programms (z.B. bei Nichtverfügbarkeit) und die Möglichkeit einer sehr feingranularen Beeinflussung. Dies führt auch gleich zu den entscheidenden Nachteilen. Neben den Risiken, die eine derartige Manipulation für das instrumentierte System birgt, besteht eine wesentliche Einschränkung gegenüber der Manipulation des Quelltextes darin, dass im bereits übersetzten Programmcode die meisten Informationen zur Analyse des Daten- oder Kontrollflusses fehlen. Dies schränkt auch die Möglichkeiten der Beschreibung von Messpunktpositionen drastisch ein.

[4] Unabhängig davon, ob es sich hierbei um den Maschinen-Code einer realen Maschine oder den Bytecode einer virtuellen Maschine (z.B. Java, Perl) handelt.

3. Komponenten und Grenzen

In der Einleitung wurde schon mehrfach der Begriff der *Komponente* verwendet. Speziell im Bereich der Softwareentwicklung werden diesem Begriff eine Reihe von Bedeutungen zugeordnet [TM97]. In dieser Arbeit wird er aber in seiner ursprünglichen Bedeutung als *Bestandteil eines Ganzen* verwendet. Dabei hängt es stark vom Standpunkt des Betrachters ab, unter welchen Gesichtspunkten er das Ganze in verschiedene Bestandteile aufteilt. Ein beliebiger Betrachter teilt ein System dadurch in Komponenten auf, dass er eine oder mehrere Eigenschaften auswählt, an denen sich verschiedene Komponenten unterscheiden lassen. So teilt sich das Gesamtsystem durch die Festlegung von Vergleichskriterien für die kleinsten Elemente des Systems automatisch in verschiedene Komponenten auf, zwischen denen zwangsläufig *Grenzen* existieren.

Da der Komponentenbegriff in so vielen Bedeutungen verwendet wird, ist es in der Praxis und speziell für die programmtechnische Verarbeitung wichtig, festzulegen was die kleinsten Elemente des Systems sind, um damit unter Angabe eines Kriteriums Komponenten zu bilden. Im betrachteten Anwendungsgebiet der objektorientierten Programmierung werden Klassen und einzelne Objekt als die kleinsten Elemente des Systems angesehen.[1]

Im Rahmen dieser Arbeit werden Komponenten dazu verwendet, zu beschreiben, an welchen Stellen des Systems Manipulationen vorgenommen werden sollen. Dies kann zum einen direkt erfolgen, d.h. es werden eine Reihe von Manipulationen nur an den Elementen einer Komponente vorgenommen, oder zum zweiten indirekt. Im zweiten Fall dienen Komponenten dazu, den Beziehungen zwischen Elementen dieser Komponenten eine erweiterte semantische Bedeutung zuzuordnen, d.h. die Art der Manipulation, die an einer bestimmten Stelle vorgenommen wird, ist abhängig von den Komponenten, denen die beteiligten Elemente angehören.

[1] Die Festlegung auf Objekte und Klassen als kleinste Elemente soll hier und im weiteren der Vereinfachung der verwendeten Begriffe dienen, da sie ein gebräuchliches Mittel zur Strukturierung von Programmsystemen geworden sind. Auch die Aufteilung von Programmcode z.B. in reinem C auf mehrere Übersetzungseinheiten wird in diesem Zusammenhang mit der Strukturierung durch Klassen gleichgesetzt (s. Abschnitt 3.2.1).

3.1. Einteilung in Komponenten

Auf der Basis von Klassen und Objekten als den kleinsten Einheiten des Systems lassen sich Programmsysteme unter verschiedenen Gesichtspunkten in Komponenten einteilen. Unabhängig von den Methoden, mit denen das System eingeteilt wird, und den Absichten, die mit der Einteilung verfolgt werden, ergeben sich drei grundlegende, orthogonale Kriterien, nach denen sich das System aufgliedern lässt. Diese sind:

1. *Funktionale Gruppierung*: eine funktions- bzw. aufgabenorientierte Zuordnung von Klassen zu Komponenten
2. *Laufzeitgruppierungen*: Gruppierung von Objekten zur Systemlaufzeit aufgrund von Gemeinsamkeiten
3. *Varianten*: Zuordnung von Objekten gleicher Implementierungsvarianten

Diese drei Kriterien werden im Folgenden näher untersucht.

3.1.1. Funktionale Gruppierung

Die funktionale Gruppierung von Systemteilen ist ein rein statischer Vorgang, der auf der Ebene der Klassenmodellierung stattfindet. Unabhängig davon, nach welcher Methodik man ein System entwirft oder ob es im Laufe der Zeit anwächst, lassen sich immer verschiedene Teile entsprechend ihrer Funktionen einordnen.

Eine Komponente wird dabei durch eine oder mehrere Klassen gebildet, die über eine Anzahl Gemeinsamkeiten verfügen. Derartige Gemeinsamkeiten können das geteilte Wissen über Datenstrukturen, Algorithmen oder spezielle Systemeigenschaften sein.

3.1.2. Laufzeitgruppierungen

Die Gruppierung zur Laufzeit des Systems erfolgt auf der Ebene von Objekten, d.h. von individuellen Instanzen von Klassen. Eine Gruppierung erfolgt hier an verschiedenen Stellen. So werden Objekte nahezu ständig in diversen Formen von Containern (Listen, Bäumen, usw.) gespeichert und verfügen somit über Gemeinsamkeiten. Eine Komponente wird aber erst dadurch gebildet, dass einer Menge von Objekten eine Bedeutung oder Aufgabe zugeordnet werden kann.

So existieren in Betriebssystemen häufig mehrere, parallele Prozesse, die jeweils durch einen Prozesskontrollblock beschrieben werden. Diese Prozesse können jetzt nach ihrem Zustand z.B. in einer Liste gruppiert werden. Die Liste der lauffähigen Prozesse ist ein typisches Beispiel – alle Prozesse dieser Liste verfügen über eine Gemeinsamkeit, den

Warte-Status. Sie können zusammen eine Komponente im Sinne der Laufzeitgruppierung bilden.

Ein weiteres Beispiel aus dem PURE-System sind die Fadenbündel. Hier können mehrere Programmfäden (leichtgewichtige Prozesse) zu einen Bündel gehören und somit auch als Komponente angesehen werden, da sie sich deutlich von Programmfäden anderer Bündel unterscheiden.

Dieser kleine Vorgriff auf die späteren Kapitel soll im Folgenden verallgemeinert werden. Aus modellierungstechnischer Sicht kommunizieren Objekte über Nachrichten miteinander – in diesem Aspekt stimmen alle Objekte einer Klasse überein. Es stellt sich jetzt aber die Frage, worin sich diese Objekte voneinander unterscheiden. Dabei geht es weniger um die Feststellung einer Identität als vielmehr um die Frage, ob es einen Unterschied macht, zwischen welchen Objekten die Kommunikation erfolgt. Mit anderen Worten geht es darum, dass über den reinen Vorgang der Nachrichtenübergabe an andere Objekte hinaus, unter Beachtung der Position der beteiligten Objekte im System bzw. der Zugehörigkeit zu einer Laufzeitgruppierung, den Kommunikationsbeziehungen verschiedene semantische Bedeutungen zugeordnet werden können.

Wieder zurück zum Anwendungsfall macht es semantisch bei der Kommunikation zweier Programmfäden einen Unterschied, ob zwei Fäden des selben oder verschiedener Bündel Nachrichten austauschen. In Abbildung 3.1 sind drei mögliche Kommunikationsbeziehungen zwischen je zwei Fäden dargestellt. Im Fall 1 liegen beide Fäden im selben Bündel bzw. Adressraum, hier erfolgt ein direkter Aufruf. In den Fällen 2 und 3 sind die Fäden durch Adressraum- bzw. Rechnergrenzen getrennt. Hier ist ein Fernaufruf notwendig.

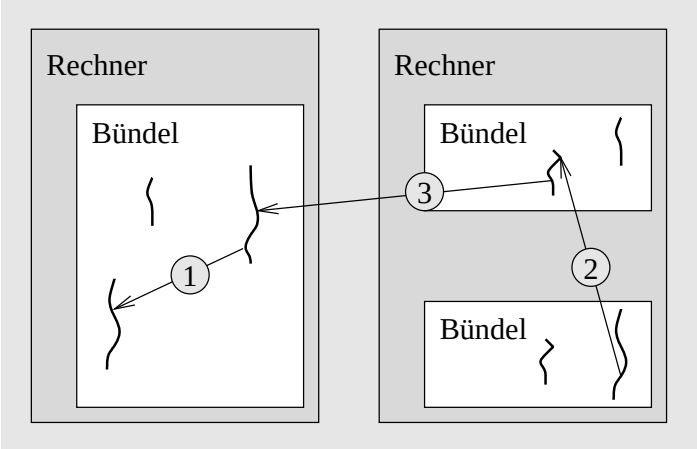


Abbildung 3.1.: Nah- und Fernaufrufe

In vielen Betriebssystemen trifft Gleiches auf die Unterscheidung von leichtgewichtigen Prozessen (*Threads*) in verschiedenen (schwergewichtigen) Prozessen mit getrennten Adressräumen zu. Hier können ebenfalls die Threads in einem Prozess direkt kommunizieren. Für den Nachrichtenaustausch mit Threads in anderen Prozessen muss dagegen auf die Mittel der Interprozess-Kommunikation zurückgegriffen werden.

3.1.3. Varianten

Der Begriff der *Varianten* beschäftigt sich mit der möglichen Vielgestaltigkeit[2] von Klassen im System. Diese Vielgestaltigkeit tritt in verschiedenen Ausprägungen auf.

Auf der Quelltextebene kann eine Klasse in verschiedenen Implementierungsvarianten vorliegen, von denen während der Übersetzung eine ausgewählt wird. Die Auswahl erfolgt dabei häufig unter Zuhilfenahme eines Präprozessors. Eine mögliche Anwendung ist die Anpassung der Implementierung an eine gewünschte Systemkonfiguration oder spezielle Hardware-Bedingungen.

Auf der Ausführungsebene können Klassen ebenfalls in verschiedenen Ausprägungen vorhanden sein. Gründe hierfür können unterschiedliche Übersetzer, unterschiedliche Optimierungen der Übersetzer oder verschiedene Zielplattformen sein, die alle in unterschiedlichen Codevarianten resultieren.

3.1.4. Bewertung der Kriterien

Nach der einzelnen Erläuterung der drei Kriterien ist jetzt noch zu betrachten, welche Zusammenhänge zwischen ihnen bestehen und welche Relevanz sie überhaupt für den praktischen Einsatz haben.

Funktionale Gruppierung und Varianten sind beide auf der Quelltextebene anzutreffen. Dabei spielt die funktionale Gruppierung die mit Abstand bedeutenste Rolle, da sie auch Bestandteil des Systementwurfs ist. Aus diesem Grund ist sie auch eine der intuitivsten Möglichkeiten, Komponenten zu beschreiben. Dies gilt speziell für die Anwendung beim Monitoring von Systemeigenschaften, da oftmals interessiert, welchen Anteil die verschiedenen Teile des Systems am gesamten Ressourcenverbrauch haben. Die Varianten eines Quelltextes spielen für das Monitoring eine geringere Rolle, da sich die Funktion der betroffenen Klasse im System nicht ändert. Lediglich die in der Klasse gekapselten Algorithmen verändern sich und müssen bei der Instrumentierung dieses Teils berücksichtigt werden. Für das Messen ist es zusätzlich von Bedeutung, die verschiedenen Varianten bei der Auswertung unterscheiden zu können.

Auf der Ausführungsebene sind die Laufzeitgruppierungen und verschiedene Varianten von Klassen anzutreffen. Dieser Punkt ist in Hinblick auf die unter 3.1.3 aufgeführten

[2] nicht zu verwechseln mit der Polymorphie in der objektorientierten Programmierung

Möglichkeiten näher zu betrachten. Eine angeführte Möglichkeit waren Klassen, deren Programm-Code gleichzeitig für mehrere Plattformen vorliegt. Derartiges ist in verteilten System zu finden, die für heterogene Umgebungen vorgesehen sind. Da aber jede dieser Varianten immer nur auf einer Plattformen lauffähig ist, überschneidet sich dieses Kriterium mit der Laufzeitgruppierung, wenn man Objekte nach den Adressräumen, in denen sie liegen, gruppiert. Für die Kommunikation zwischen zwei Objekten und somit der Auswahl des dafür notwendigen Verfahrens (Direktaufruf, Fernaufruf) ist es somit unerheblich, ob verschiedene Plattformvarianten oder die Adressräume unterschieden werden müssen. Es reicht, alle Objekte des Systems eindeutig unterschieden zu können, um das passende Kommunikationsverfahren auswählen zu können.

Die zweite aufgeführte Möglichkeit für Varianten sind verschiedene Implementierungsvarianten einer Klasse. Von dieser Möglichkeit kann z.B. zum Zwecke der Laufzeitkonfigurierung Gebrauch gemacht werden. Eine derartige Anwendung ist für das Monitoring nicht von Interesse, da sich die Objekte äußerlich nicht ändern. Davon bleibt aber das Monitoring des „Innenlebens“ ungerührt, denn dies ist ein Problem auf der Quelltextebene.

3.2. Beschreibung von Komponenten

Wie bereits in der Einleitung zu diesem Kapitel erwähnt, ergeben sich bei der Einteilung des Systems in Komponenten zwangsläufig Grenzen zwischen diesen Komponenten. Daraus folgt aber im Gegenzug auch, dass sich die Komponenten auch durch die Festlegung der Grenzen definieren lassen. Dieser Sachverhalt wird in Abbildung 3.2 vereinfacht durch eine Punktmenge in der Ebene dargestellt.

Zu sehen ist zum einen die Beschreibung der Komponenten durch die Angabe aller Elemente und zum zweiten durch die Angabe von Grenzen. Im Resultat unterscheiden sich beide Varianten nicht – beide ordnen die selben Elemente den jeweiligen Komponenten zu. Zu untersuchen wäre nur der Aufwand der Beschreibung in beiden Varianten.

Stellt man sich diese vereinfachte Darstellung körperlich vor, so bilden die Grenzen zwischen den Komponenten auch deren „Oberflächen“ – in der objektorientierten Terminologie wäre dies die Summe der öffentlichen Schnittstellen aller Klassen und Objekte einer Komponente. Aus diesem Grund soll vor der Betrachtung der Möglichkeiten zur Beschreibung von Komponenten ein Augenmerk auf die Beziehungen zwischen Klassen und Objekten eines objektorientierten Programmsystems gerichtet werden.

Ziel der Betrachtungen ist die Suche nach Systemeigenschaften, die durch ein Software-Werkzeug zur Unterstützung des Beschreibungsprozesses genutzt werden können. Motivierend hierfür sind zwei Tatsachen. Zum ersten bedeutet der Einsatz von Software-Werkzeugen eine deutliche Verringerung des Beschreibungsaufwands bei den zum Teil

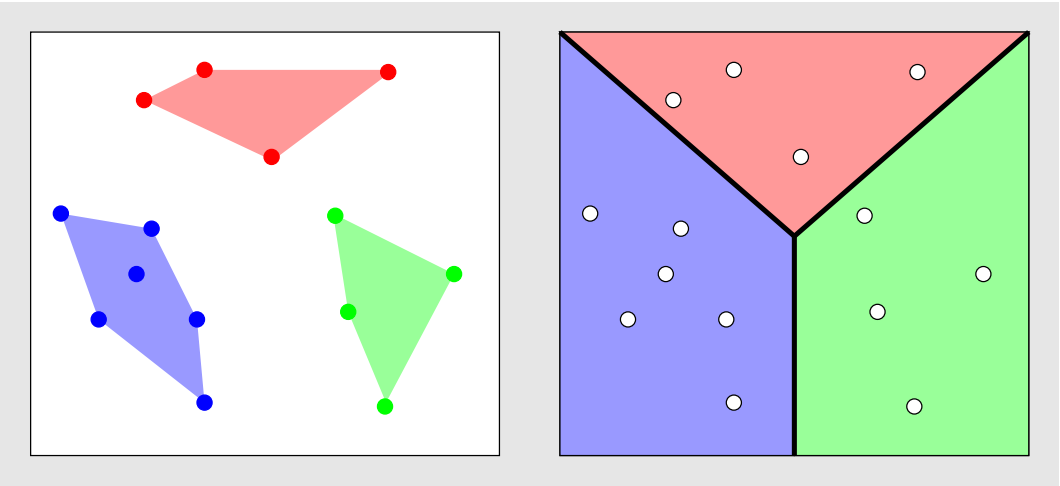


Abbildung 3.2.: Komponenten und Grenzen

recht umfangreichen Klassenbibliotheken. Zum zweiten ist die Klassenstruktur eines Betriebssystems wie in der Fallstudie, das hochmodular und für den Einsatzfall maßschneiderbar sein soll, in einigen Teilen sehr variabel. Hier bietet die Werkzeugunterstützung die Möglichkeit, die Abhängigkeit von der Konfiguration zu vermindern wenn nicht gar zu beseitigen.

3.2.1. Klassen und Module

In dieser Arbeit ist an vielen Stellen von Klassen und Objekten die Rede. An diesen Stellen drängt sich dann schnell die Frage auf, ob der in dieser Arbeit verfolgte Ansatz nur für objektorientierte Sprachen und hier speziell C++ zu gebrauchen ist. Die Beantwortung dieser Frage umfasst mehrere Aspekte, auf die im Folgenden eingegangen wird.

Zum ersten dienen Klassen der Strukturierung des Systems, sie definieren eine Schnittstelle zu den verschiedenen Funktionalitäten und legen Datenstrukturen fest. Die selben Möglichkeiten bieten auch rein prozedurale Sprachen, die durch einen Compiler in Objektmodule übersetzt werden. Auch bei der Softwareentwicklung mit einer prozeduralen Sprache wird das System strukturiert, da es sonst bei großen Projekten sehr unüberschaubar wird. Das Mittel hierzu sind einzelne Dateien. Sie bilden das Analogon zu Klassen, da sie eine Menge von Variablen und Funktionen in einem funktionalen Block kapseln. Zur Verbindung mit anderen Modulen müssen die im erzeugten Objektmodul vorhandenen Symbole der Variablen und Funktionen exportiert und den anderen

Modulen bekanntgegeben werden. Dies ist mit der Funktion einer Klassendefinition gleichzusetzen.

Daneben bestehen auch ein paar Nachteile. So definieren Klassen einen eigenen Namensraum. Dies ist bei den prozeduralen Modulen nicht gegeben. Auch fehlen den prozeduralen Sprachen die Möglichkeiten der objektorientierten Sprachen wie Vererbung und Überladung. Eine objektorientierte Programmierung und „virtuelle“ Methoden sind dagegen durch die Nutzung von verschiedenen Datenstrukturen und Zeigern auf Funktionen möglich. Aber diese Vorteile einer objektorientierten über eine prozedurale Sprache haben nur für den Systementwickler eine Bedeutung – für die Beschreibung der Systemstruktur als Vorbereitung der Instrumentierung haben sie keine Bedeutung. Aus diesem Grund werden Klassen als Beschreibungsmittel für Komponenten verwendet. Der fehlende Namensraum prozeduraler Sprachen wird dadurch kompensiert, dass einfach der Dateiname des Programmmoduls verwendet wird, denn dieser wurde vom Entwickler sicher mit Bedacht gewählt.

Zum zweiten sind die möglichen Programmiersprachen zu betrachten, für die dieser Ansatz verwendet werden kann. Vom Prinzip her ist der Ansatz sprachneutral und somit potenziell für beliebige Sprachen anwendbar. Die eigentliche Einschränkung ergibt sich durch die Voraussetzungen, die für den praktischen Einsatz geschaffen werden müssen. Grundvoraussetzung ist ein Parser für die zu instrumentierende Programmiersprache sowie Funktionen zur Manipulation des erzeugten abstrakten Syntaxbaums. Dazu kommen für die Messung die Software-Messinstrumente, die in den Quelltext eingebracht werden. Sie stellen aber im Gegensatz zum Parser und den Manipulationsfunktionen aufgrund ihrer geringen Funktionalität und Größe kein Hindernis dar.

Zusammenfassend ist somit festzustellen, dass der in dieser Arbeit entwickelte Ansatz praktisch nur für C und C++ unter Ausnutzung der hybriden Sprachstruktur zur Verfügung steht. Dies stellt jedoch im Hinblick auf die enorme Verbreitung dieser Sprache keine schwerwiegende Einschränkung dar.

3.2.2. Vorgehensweisen

Der eigentliche Beschreibungsprozess kann auf drei Arten erfolgen. Die erste ist die explizite Zuordnung von Mengen der kleinsten Elemente (Methoden, Attribute) zu Komponenten. Alternativ kann als zweite Variante eine Beschreibungsform verwendet werden, die gleichzeitig ganze Gruppen kleinster Elemente einer Komponente zuordnet. Als drittes besteht die Möglichkeit, nur die Grenzen einer Komponente anzugeben und die jeweils umschlossenen Mengen automatisch suchen zu lassen.

Die Beschreibung von Komponenten auf Basis der kleinsten Elemente des Systems stellt die flexibelste und gleichzeitig aufwendigste Variante dar, da die Menge dieser kleinsten Elemente recht umfangreich sein kann. Sie hat aber dennoch ihre Daseinsberechtigung,

wenn es darum geht, eine Systemkomponente zu beschreiben, die nur sehr wenige Elemente enthält. Eine Anwendung wird beim Monitoring der Prozessumschaltung im PURE-System (s. Abschnitt 5.3) gezeigt. Gleichzeitig ist sie die Basis für die höheren Beschreibungsformen.

Die Angabe von Gruppen von kleinsten Elementen kann auf verschiedene Arten erfolgen. Die gebräuchlichste wird sicher die Angabe von ganzen Klassen sein, wenn alle in ihr enthaltenen Elemente einer Komponente zugeordnet werden sollen. Noch allgemeiner ist die Verwendung von Mustern in Form regulärer Ausdrücke, die mit der Kombination aus Klassenname und Elementname verglichen werden. Es lassen sich sicher noch mehr Möglichkeiten finden – dies jedoch ist Aufgabe des Modellentwicklers, der erst am konkreten Beispiel die geeignetste Möglichkeit finden kann.

Im Abschnitt 3.2 wurde bereits der Zusammenhang von Komponenten und Grenzen erläutert. Nutzt man diesen Zusammenhang aus, so kann eine Komponente auch allein durch Angabe ihrer Grenzen definiert werden. Es besteht dann allerdings noch die Aufgabe, die von der angegebenen Grenze definierte Menge der kleinsten Elemente zu bestimmen. Dieser Vorgang ist nicht trivial, bildet aber die Grundlage für die Entwicklung eines Werkzeuges zur Unterstützung des Beschreibungsprozesses und wird deshalb unter 3.3 näher erläutert.

3.3. Automatische Erkennung von Komponenten

Die Erstellung einer vollständigen Komponentenbeschreibung ist ein recht aufwendiger Prozess. Um ihn in der Praxis effizienter zu gestalten, ist eine Kombination der in Abschnitt 3.2.2 vorgestellen Vorgehensweisen oftmals sehr vorteilhaft. Das Ziel ist es, unter Ausnutzung der Erfahrungen über übliche Programmierweisen und die aus dem System bestimmten Beziehungen zwischen den Klassen so viel Arbeit wie möglich bei der Definition von Komponenten automatisch durch ein Software-Werkzeug erledigen zu lassen. Die Grundlagen und Strategien dieser Vorgehensweise sind Bestandteil der folgenden Abschnitte.

3.3.1. Grundlagen

Die Grundlage der automatischen Erkennung von Komponenten bzw. der Bestimmung der von einer festgelegten Grenze umschlossenen Teile, ist die Analyse der elementaren Beziehungen, die zwischen Klassen und Objekten bestehen können. Klassen und Objekte können auf verschiedene Arten miteinander verbunden sein. Diese Beziehungen können ihrer Stärke nach sortiert werden:

Vererbung

Die Vererbung stellt einen der grundlegenden Mechanismen der objektorientierten Programmierung dar. Sie erlaubt es, ein System im *Bottom-Up*-Verfahren Schritt für Schritt aus kleinen funktionalen Einheiten aufzubauen sowie allgemeingültige Schnittstellen zu definieren, die danach spezialisiert werden können. Sie stellt die stärkste Form der Verbindung zweier Klassen dar, da die Basisklasse vollständig in der abgeleiteten Klasse eingebettet ist und die abgeleitete Klasse zudem meist einen umfassenderen Zugriff auf die Interna der Basisklasse erhält als andere Klassen.

Komposition

Bei der Komposition wird in einer Klasse ein Attribut vom Typ einer anderen Klasse verwendet. Dies hat zur Folge, dass die Daten der benutzten Klasse in die Klasse selber integriert werden und somit das so instantiierte Objekt von der Existenz der umgebenden Klasse abhängig ist. Der Zugriff auf die benutzte Klasse ist allerdings nur über deren öffentliche Schnittstelle möglich. Aus diesem Grund ist diese Beziehung auch schwächer als die Vererbung.

Aggregation

Die Aggregation ist mit der Komposition eng verwandt. Sie unterscheidet sich jedoch dadurch, dass nur eine Referenz bzw. ein Zeiger auf das verwendete Objekt als Attribut verwendet wird. Die Existenz des verwendeten Objekts ist somit nicht zwangsläufig an das Objekt selber gebunden. Der Zugriff auf das verwendete Objekt ist ebenfalls nur über die öffentliche Schnittstelle der entsprechenden Klasse möglich.

Methoden-Argumente

Die schwächste Form der Verwendung von Objekten anderer Klassen ist die Verwendung als Argument oder Resultat einer Methode. Auch hier kann noch einmal danach unterschieden werden, ob das Argument ein Objekt oder nur ein Zeiger darauf ist.

Betrachtet man nicht nur die Definitionen der Klassenschnittstellen, sondern auch die Implementierung der Methoden, so wird die Beziehung zu einer anderen Klassen über ein Argument dadurch aufgewertet, dass auf das übergebene Objekt z.B. durch einen Methodenaufruf zugegriffen wird. Diese Beziehung wird hierdurch mit der Aggregation gleichwertig. Hiervon wird aber standardmäßig kein Gebrauch gemacht, da hierzu die Implementierungen aller Methoden parsiert und analysiert werden müssten. Für

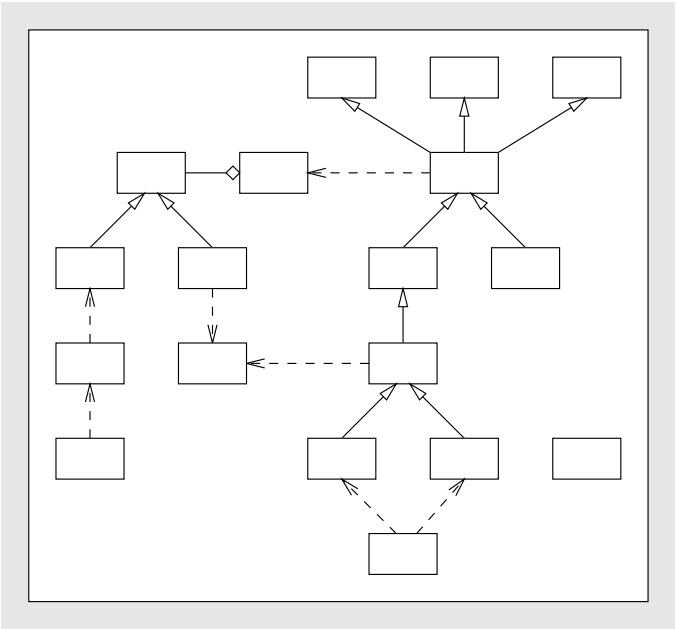


Abbildung 3.3.: Beschreibung von Komponenten (Klassenstruktur)

die meisten Anwendungen reicht das Parsen der Klassendefinitionen[3] zur Analyse der Klassenbeziehungen aus. Bei Bedarf können mit den vorhandenen Mitteln auch die Implementierungen analysiert werden, dies bedeutet jedoch einen deutlich erhöhten Ressourcenverbrauch.

3.3.2. Strategien

Ausgehend von den im Abschnitt 3.3.1 dargestellten möglichen Beziehungen zwischen Klassen ergeben sich drei Strategien für die automatische Zuordnung von Klassen zu Komponenten. Die Rangfolge der Strategien in der Praxis orientiert sich an der Stärke der ihnen zugrundliegenden Beziehung. Prinzipiell lässt sich jede der drei Strategien für jedes der drei Kriterien (s. Abschnitt 3.1) anwenden. Dies ist aber nicht unbedingt sinnvoll, wie in den folgenden Abschnitten erläutert wird.

Zur Erläuterung der Auswirkungen der verschiedenen Strategien sei die in Abbildung 3.3 dargestellte Klassenstruktur gegeben.

Die stärkste Beziehung zwischen zwei Klassen ist die Vererbung. So wird mit der ersten Strategie versucht, die Komponentenzugehörigkeit mitzuvererben, d.h. die Komponen-

[3] d.h. der entsprechenden Header-Dateien

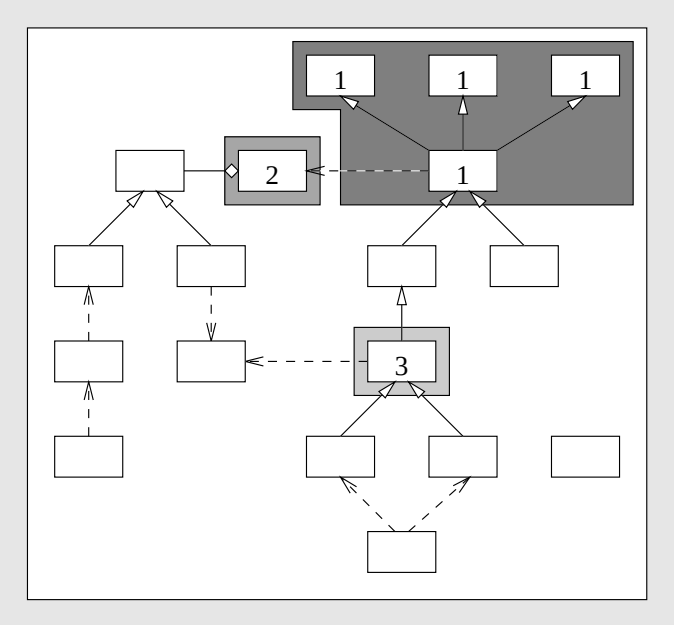


Abbildung 3.4.: Beschreibung von Komponenten (individuelle Zuordnungen)

tenzugehörigkeit wird vererbt, wenn die abgeleiteten Klassen noch keiner Komponente zugeordnet wurden.

Diese Strategie zeigt in der Praxis immer dann Wirkung, wenn sich in der Klassenhierarchie verschiedene funktionale Schichten finden lassen, was in den meisten umfangreichen Systemen der Fall sein dürfte. Zudem ist es oftmals so, dass es mehrere Stufen der Vererbung in einer Schicht gibt. Um keine Mehrdeutigkeiten bei der automatischen Zuordnung zu erzeugen, wird die Komponentenzugehörigkeit parallel zur Vererbung der Klasseneigeschaften festgelegt, d.h. von der Basisklasse zur abgeleiteten Klasse. So ist es für die Zuordnung aller Klassen einer Schicht zu einer Komponente nur noch notwendig, die jeweils ersten Klassen der Schicht manuell zuzuordnen.

Ist eine Klassenhierarchie sehr stark untergliedert und wird häufig von der Möglichkeit der Mehrfachvererbung Gebrauch gemacht, so ist es sinnvoll, die Komponentenzugehörigkeit auch rückwärts zu vererben, d.h. die Zugehörigkeit wird für eine Klasse festgelegt und auf die Basisklassen übertragen. Da zur Vermeidung von Mehrdeutigkeiten bereits festgelegt wurde, dass die Komponentenzugehörigkeit automatisch nur in Richtung der Vererbungsbeziehung weitergegeben wird, kann die Rückwärtsvererbung nur im Zuge der Zuweisung für eine festgelegte Anzahl von Stufen erfolgen. Abbildung 3.4 zeigt den Zustand nach der individuelle Zuordnung von Klassen zu Komponenten. Bei der Zuweisung der Komponente 1 wurde dabei von der Möglichkeit der gleichzeitigen Zuweisung

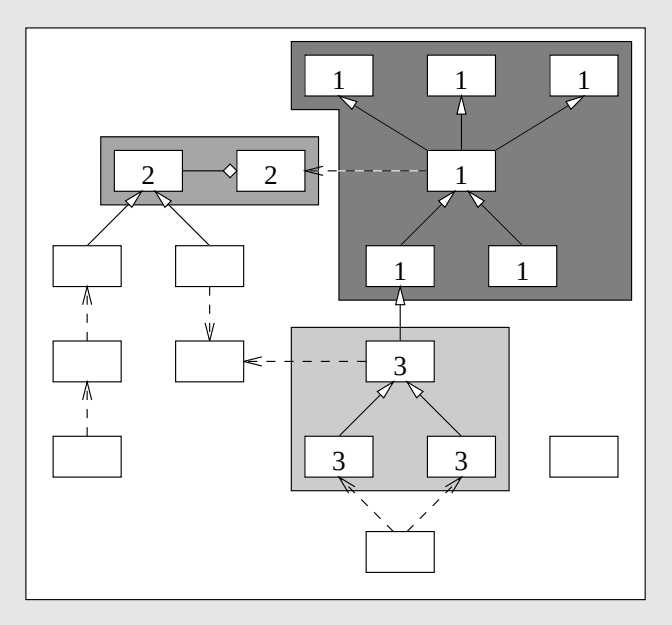


Abbildung 3.5.: Beschreibung von Komponenten (1. automatische Zuordnungen)

an die Basisklassen Gebrauch gemacht.

Die nächstschwächeren Klassenbeziehungen sind die Aggregation und die Komposition. Da sie sehr eng verwandt sind, werden sie in der zweiten Strategie zusammengelegt. Nach dieser zweiten Strategie wird versucht, allen Klassen, die von einer bereits zugeordneten im Sinne der Aggregation oder Komposition verwendet werden, die Komponentenzugehörigkeit zu übertragen. Diese Strategie zeigt sich dann als erfolgreich, wenn die benutzten Klassen Hilfsklassen sind. Ein anderes Beispiel für einen erfolgreichen Einsatz dieser Strategie sind die diversen Container-Klassen, die verschiedenste, zum Teil polymorphe Datenobjekte speichern. Hier reicht die Zuordnung des Containers zu einer Komponente. Alle darin speicherbaren Klassen werden durch die zweite Strategie, bei polymorphen Klassen auch in Kombination mit der ersten Strategie gefunden.

Beide bisher vorgestellten Strategien arbeiten rekursiv, d.h. die Komponentenzugehörigkeit wird schrittweise weitergegeben. Wendet man beide Strategien jeweils einmal, so erhält man das in Abbildung 3.5 dargestellte Resultat. Sofern nicht anders angegeben, entspricht die Reihenfolge der Anwendung der Strategien der der Darstellung im Text.

Das Verfahren zur automatischen Zuordnung wird mit den ersten beiden Strategien ein zweites Mal durchlaufen. Danach ergibt sich das in Abbildung 3.6 dargestellte Resultat, das nach den bisherigen Möglichkeiten den Endzustand der automatischen Erkennung kennzeichnet, d.h. mit den bisherigen Strategien ist keine weitere Zuordnung mehr

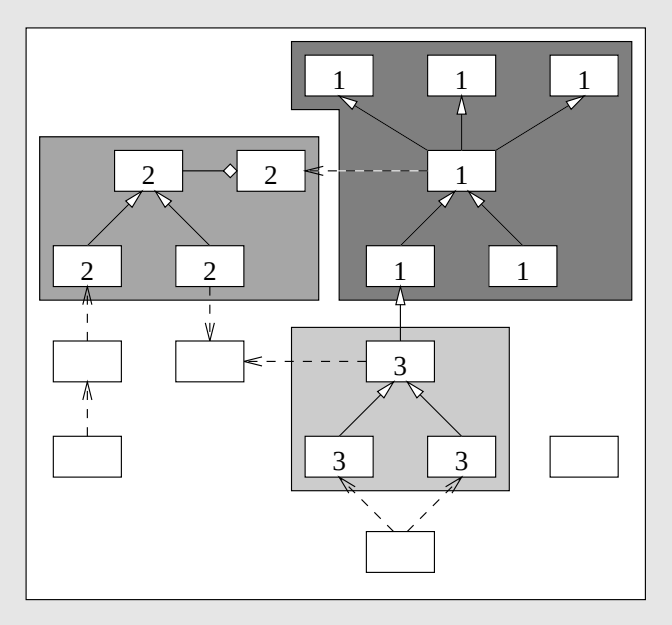


Abbildung 3.6.: Beschreibung von Komponenten (2. automatische Zuordnungen)

möglich.

Der in der Abbildung 3.6 dargestellte Endzustand kann noch nicht vollständig zufriedenstellen, da ein menschlicher Betrachter sicher noch ein paar Zuordnungen finden wird, die automatisch erfolgen könnten.

Aus diesem Grund existiert eine dritte Strategie, die das Problem auf eine andere Weise angeht. Abbildung 3.7 zeigt die vier wichtigen Phasen dieser Strategie. In einem ersten Schritt wird die dargestellte Klassenhierarchie (**A**) in einen Graphen (**B**) umgewandelt, in dem die Klassen die Knoten und die unterschiedlichen Beziehungen die Kanten bilden. Alle bisherigen Komponentenzuordnungen bleiben erhalten. Dabei spielt es keine Rolle, um welche Beziehung es sich handelt. Ziel der Untersuchung des Graphen ist das Auffinden von Teilgraphen, die eine eindeutige Zuordnung erlauben. Dazu werden nach der Erstellung des Graphen in einem weiteren Schritt alle Kanten entfernt, die eine Grenze zwischen zwei Komponenten durchkreuzen. Derartige Kanten sind dadurch gekenzeichnet, dass die Knoten, die durch sie verbunden werden, zu verschiedenen Komponenten gehören. Als Ergebnis (**C**) dieses Schrittes sollte sich der Graph in mehrere unabhängige Teilgraphen aufspalten.

Um die Suche nach isolierten Teilgraphen weiter zu vereinfachen, werden des Weiteren alle Kanten zwischen Knoten, die zur selben Komponente gehören, entfernt. Durch das Entfernen der unnützen Kanten haben sich zwölf unabhängige Teilgraphen gebildet (**D**).

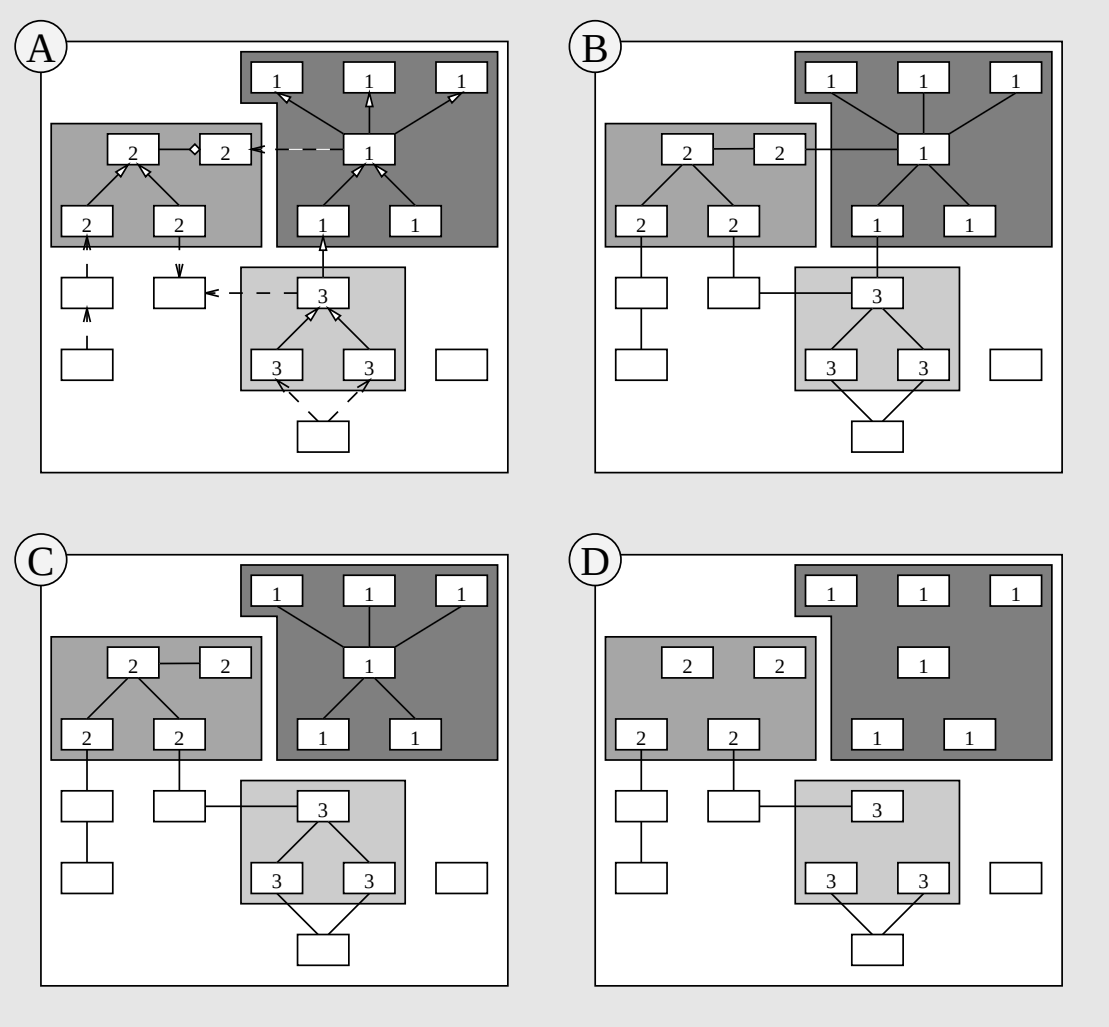


Abbildung 3.7.: Beschreibung von Komponenten (Teilgraphen)

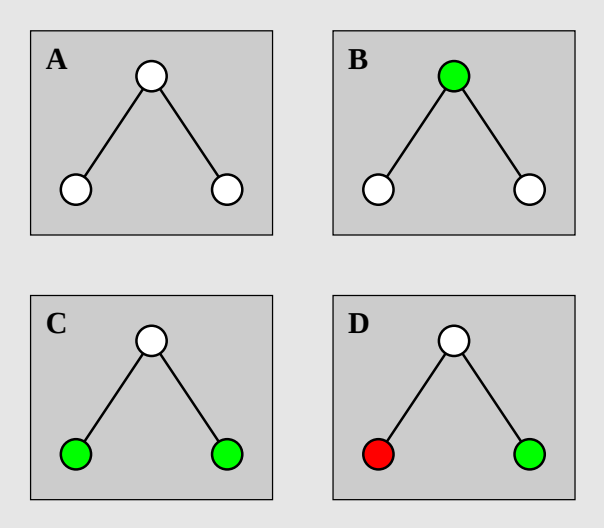


Abbildung 3.8.: Konstellationen der Teilgraphen

Betrachtet man sich diese Teilgraphen genauer, so fällt auf, dass es unter ihnen genau vier mögliche Konstellationen gibt. Diese vier Konstellationen sind noch einmal getrennt in Abbildung 3.8 zu sehen. Diese vier Konstellationen sind durch folgende Eigenschaften gekennzeichnet:

A: kein Knoten wurde einer Komponente zugeordnet

B: genau ein Knoten wurde einer Komponente zugeordnet

C: mehrere Knoten gehören zur selben Komponente

D: mindestens zwei Knoten gehören nicht zur selben Komponente

Bevor der Algorithmus zur Erkennung der einzelnen Konstellationen erläutert wird, ist noch zu betrachten, welche Informationen uns diese Konstellationen liefern.

A: die Knoten des Graphen sind vollständig isoliert, sie können nicht automatisch zugeordnet werden

B,C: alle nicht zugewiesenen Knoten stehen direkt oder indirekt nur mit Knoten einer Komponente in Verbindung, sie können nur zu dieser Komponente gehören

D: die Zuordnung zu einer Komponente kann nicht eindeutig und somit automatisch erfolgen

Bei der Analyse der Teilgraphen stellt sich als erstes die Frage danach, wo man mit der Suche beginnen soll. Betrachtet man allerdings die Umstände, unter denen die Teilgraphen entstanden sind, so ist klar, dass es reicht, bei den bereits zugewiesenen Knoten zu beginnen, da in jedem Teilgraphen mindestens ein zugewiesener Knoten am Rand existiert. Ausgenommen hiervon sind die Teilgraphen, in denen überhaupt keine Knoten zugeordnet sind – da sie aber keine neuen Erkenntnisse bringen, spielt es keine Rolle, dass sie nicht betrachtet werden. Im Gegenteil beschleunigt diese Auslassung sogar den Gesamtvorgang. Ausgehend von den zugewiesenen Knoten wird mit einer Tiefensuche der Teilgraph nach allen anderen zugewiesenen Knoten durchsucht. Jeder betrachtete Knoten wird dabei markiert, um Zyklen zu vermeiden. Da nur die Konstellationen B und C eine weitere Zuordnung ermöglichen, kann die Suche abgebrochen werden, sobald der erste zugewiesenen Knoten gefunden wird, der nicht zur selben Komponente wie der Startknoten gehört. Ist dies nicht der Fall, so wird der Teilgraph erneut traversiert. Diesmal werden alle noch nicht zugewiesenen Knoten der Komponente des Startknotens zugeordnet. Das Ergebnis der Anwendung der dritten Strategie lässt sich in Abbildung 3.9 erkennen.

In der Praxis können die Strategien auf zwei Arten angewendet werden. Zum ersten vollautomatisch und zum zweiten auch manuell. Bei der vollautomatisch Anwendung werden die drei Strategien nacheinander durchlaufen. Dabei wird immer mit der ersten begonnen und diese solange angewendet, bis keine Änderungen mehr erfolgen. Danach wird zur nächsthöheren übergegangen. Abgebrochen wird der Vorgang, wenn selbst die dritte Strategie keine Zuweisungen mehr vornehmen kann. Für das in diesem Abschnitt gegebene Beispiel nimmt der vollautomatische Zuordnungsprozess genau die gezeigten Zuweisungen vor. Auch für viele reale Klassenstrukturen führt die vollautomatische Zuordnung zum erwarteten Resultat. Der Standardalgorithmus kann aber auch versagen. In diesem Fall können die drei Strategien auch einzeln angewendet werden. Für spezielle Anwendungen besteht zudem die Möglichkeit eine einzelne Komponenten zu bevorzugen, d.h. dass die Weitergabe einer Komponentenzuordnung auf eine einzelne beschränkt werden kann. Auch werden bei dieser Möglichkeit Mehrdeutigkeiten bei der Verwendung einer Klasse ignoriert, an denen die bevorzugte Komponente beteiligt ist (s. Abbildung 3.8, Teil **D**). In einem solchen Fall erfolgt dann eine Zuordnung zur bevorzugten Komponente.

3.3.2.1. Funktionale Gruppierung

Bei der Festlegung funktionaler Gruppierungen können alle drei Strategien mit Erfolg angewendet werden – sie sind auch speziell für diesen Anwendungszwecke entworfen wurden. Grund hierfür ist die Tatsache, dass beim Messen als eine der Anwendungen des Monitorings oftmals das Leistungsverhalten einzelner Systembausteine interessiert.

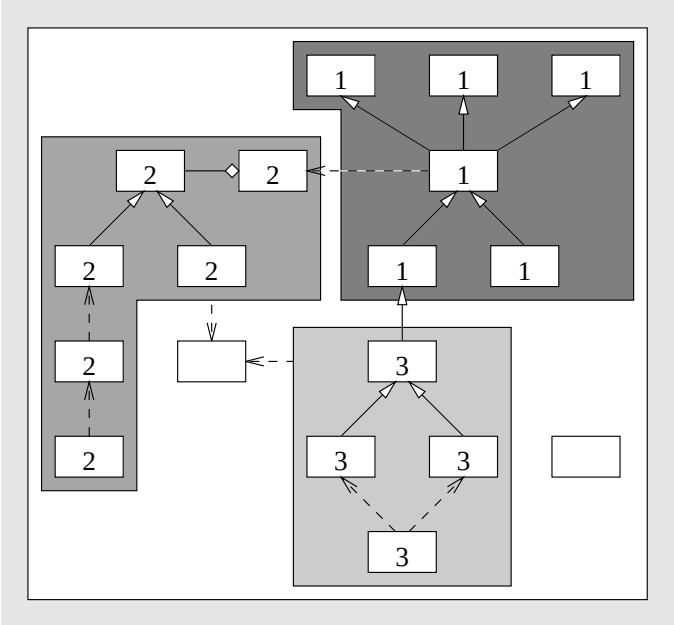


Abbildung 3.9.: Ergebnis der automatischen Zuordnung

3.3.2.2. Laufzeitgruppierungen

Für die Festlegung von Komponenten auf der Basis von Laufzeitgruppierungen stehen prinzipiell alle drei Strategien zur Verfügung, jedoch macht in den meisten Fällen nur die Vererbung einen Sinn. Grund hierfür ist die Tatsache, dass sich das Monitoring von Laufzeitgruppierungen in zwei Teile auftrennt. Zum ersten muss festgelegt werden, welcher Klassen Objekte unterscheidbar sein sollen und zum zweiten, an welche Stellen sie unterschieden werden sollen.

Die Fähigkeit zur Unterscheidung zweier Objekte einer Klasse muss nicht zwangsläufig in der Klasse integriert sein (s. Abschnitt 4.3.4). Da aber die Objekte abgeleiteter Klassen auf die Basisobjekte zurückgeführt[4] werden können, ist es notwendig, diese Eigenschaft zu vererben.

Die eigentliche Unterscheidung und die damit verbundenen Aktionen sind ein Aspekt des Programms und müsses somit in einigen oder meist in allen Bereichen des Systems erfolgen, in denen Objekte der unterscheidbaren Klassen verwendet werden. Somit fällt die Beschreibung dieses Teils des Monitorings unter die Festlegung funktionaler Gruppierungen.

[4] herunter ge-cast-et

3.3.2.3. Varianten

Die Beschreibung von Varianten gestaltet sich etwas komplizierter, da hier die Unterscheidung bzw. Auswahl verschiedener Varianten durch die verwendeten Präprozessoren und Übersetzer gesteuert wird. Dies bedeutet, dass eine Vorabbeschreibung nur in Verbindung mit der Ansteuerung der Übersetzungswerkzeuge erfolgen kann, d.h. die Beschreibung der ersten beiden Kriterien wird durch die Übersetzungsparameter beeinflusst. Dies bedeutet auch, dass keine der drei Strategien direkt verwendet wird, sondern eine höhere Beschreibungsform zur Definition variabler Beschreibungen für die Festlegung von funktionalen und Laufzeitgruppierungen Anwendung findet.

3.3.3. Grenzen des Verfahrens

Man darf nicht vergessen, dass der ganze automatische Zuordnungsprozess genau genommen nichts weiter ist als zielstrebiges Raten. Aus diesem Grund sind Fehler vorprogrammiert und die Verfahren sollten nur als Hilfsmittel betrachtet werden, d.h. die gelieferten Resultate sollten immer kritisch beurteilt werden. An einigen Stellen ist der Anwender gefordert, um hier Unstimmigkeiten aufzulösen.

Der automatische Zuordnungsprozess kann zur Folge haben, dass Klassen fälschlich oder nicht zugewiesen werden. Die falsche Zuweisung einer Klasse zu einer Komponente erfolgt immer dann, wenn die Klasse nur mit Klassen einer Komponente in Beziehung steht, da in diesem Fall davon ausgegangen wird, dass sie nur zu dieser Komponente gehören kann. Ein Gegenbeispiel ist eine Hilfsklasse zur Kapselung einer Datenstruktur, die aber nur an einer Stelle des Systems verwendet wird, da die anderen Teile, die auch diese Klasse verwenden würden, noch nicht vorhanden sind. An dieser Stelle kann der Modellentwickler aber leicht Abhilfe schaffen, indem er eine eigene Komponente für derartige Hilfsklassen definiert und somit auch allen zukünftigen Problemen aus dem Weg geht, die durch mögliche Mehrdeutigkeiten in der Beschreibung entstehen könnten.

Eine andere Fehlerquelle sind Programmteile, die den Daten- oder Kontrollfluss verschleiern und somit eine Weitergabe von Zugehörigkeitseigenschaften verhindert. Beispiele hierfür sind unsaubere Typumwandlungen von Zeigern oder externe Quellen, die den Kontrollfluss ändern. Bei derartigen Erscheinungen ist der Entwickler jedch auf sich gestellt.

Es zeigt sich auch, dass es nicht immer möglich ist, alle Klassen automatisch den verschiedenen Komponenten zuzuordnen. Die Ursache hierfür kann zum ersten in der Tatsache liegen, dass eine Klasse isoliert ist und zum zweiten darin, dass sie von mehreren Klassen unterschiedlicher Komponenten gleichzeitig verwendet wird. Dies muss aber nicht immer ein unvollständiges Ergebnis bedeuten. Kapseln die jetzt noch freien Klassen einfach nur Datentypen oder werden sie immer nur lokal zu den jeweiligen Komponenten benutzt

und nicht an andere Komponenten weitergegeben, besteht auch keine Notwendigkeit, sie einer eigenständigen Komponente zuzuordnen.

Zusammenfassend sieht der Prozess der Systembeschreibung wie folgt aus:

1. individuelle Zuordnung von Klassen zu Komponenten
2. automatische Vervollständigung der Komponentenbeschreibung
3. Kontrolle des Ergebnisses
4. manuelle Korrektur von Fehlern oder Unvollständigkeiten

4. Realisierung des generischen Systemmonitors

Abbildung 4.1 zeigt in Form eines Aktivitätsdiagramms die zur Durchführung eines System-Monitorings notwendigen Schritte. Ausgangspunkt ist die Quelltextform des zu instrumentierenden Systems. Dieses wird parsiert und auf seine strukturellen Elemente (Klassen, Funktionen, Variablen, ...) hin analysiert. Die Informationen der Strukturdatenbank verwendet der Aspektweber zusammen mit der Komponentenbeschreibung, die nach Bedarf durch die Komponentenanalyse vervollständigt wird, und dem Aspektprogramm zur Transformation des ursprünglichen Syntaxbaums. Ergebnis dieses Prozesses ist ein neuer Syntaxbaum, aus dem die instrumentierte Quelltextform des System synthetisiert werden kann. Diese wird anschließend in die ausführbare Form übersetzt. Wurde zum Zwecke einer Messung instrumentiert, so sammeln die eingebrachten Sensoren zur Laufzeit die gewünschten Informationen zusammen, die parallel oder im Anschluß ausgewertet werden können.

Die Konzepte und Details dieses Prozesses sowie die Umsetzung im Rahmen dieser Arbeit werden in den folgenden Abschnitten näher erläutert.

4.1. Analyse des Systems

Der erste Schritt bei der Instrumentierung eines bestehenden Systems ist die Analyse des Systems. Dabei wird es auf seine innere Struktur hin untersucht, die später für die Beschreibung von Komponenten sowie das Einweben der Aspekte benötigt wird. An dieser Analyse sind der Parser und die Strukturdatenbank, die alle ermittelten Informationen aufnimmt, beteiligt.

4.1.1. Der Parser

Aufgabe des Parsers ist die syntaktische und semantische Analyse des Quelltextes des zu instrumentierenden Systems. Er wandelt die flache Form des Quelltextes in eine strukturierte Form, den Syntaxbaum um. Für diese Aufgabe wird das PUMA-System

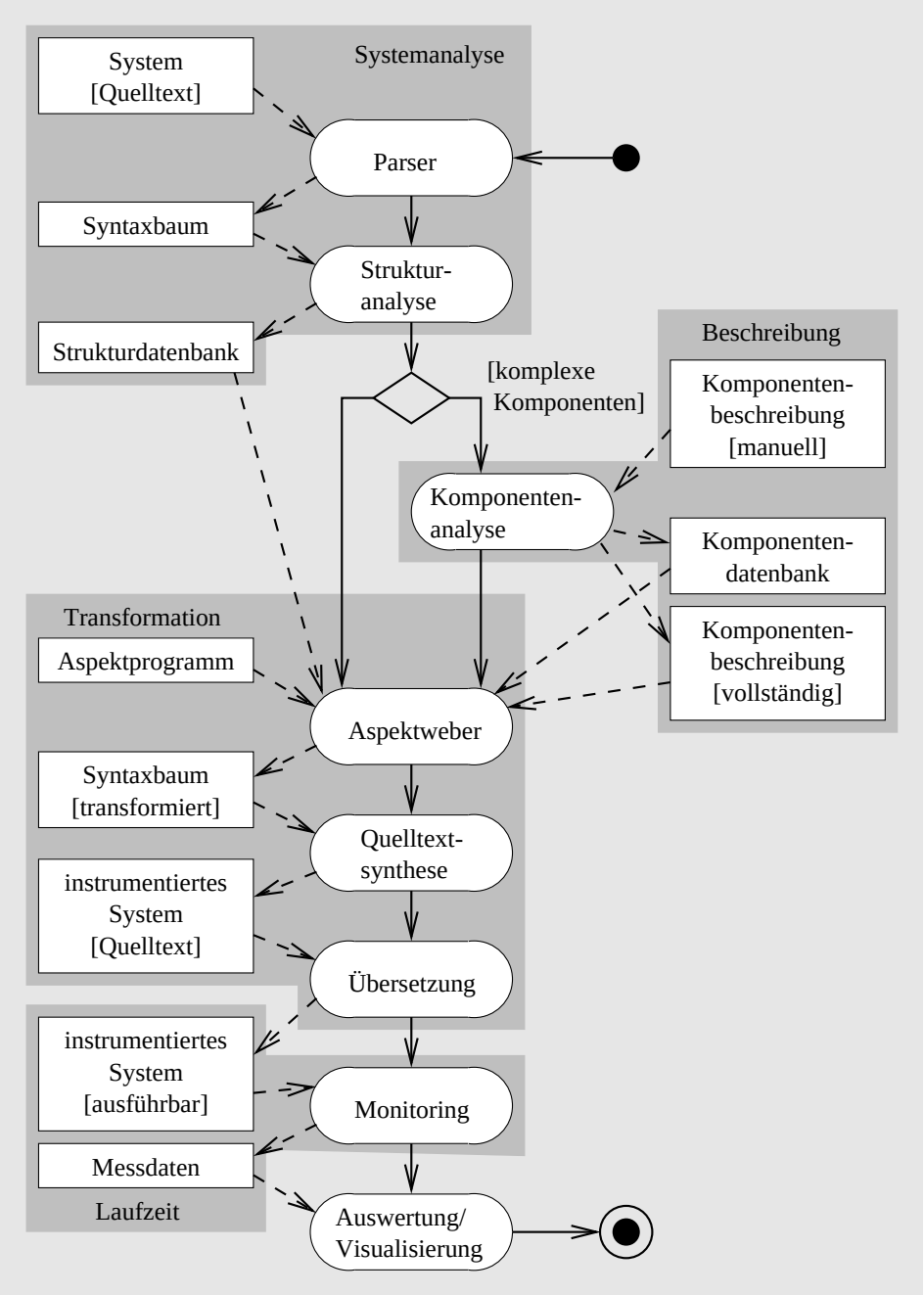


Abbildung 4.1.: Realisierung eines Monitoring-Projektes

verwendet, das neben dem Parsen von C- und C++-Quelltexten auch noch eine Reihe von grundlegenden Funktionen zur Manipulation des erzeugten Syntaxbaumes sowie zur Verwaltung von Systemen, die aus einer Reihe von Quelltextdateien bestehen, anbietet.

4.1.2. Die Strukturdatenbank

Die zweite Stufe der Analyse des Systems ermittelt die Struktur des Systems aus der Sicht des Programmierers, d.h. aus den sehr detaillierten Informationen, die der Syntaxbaum bietet, werden die für die spätere Arbeit wichtigen herausgesucht und so eine Art Suchindex über den Syntaxbaum erstellt. Da ein komplexes System in der Regel auf mehrere Quelltextdateien verteilt wird, müssen diese auch in verschiedenen Durchgängen parsiert werden und es entstehen auf diesem Wege auch mehrere Syntaxbäume. Es kommt dabei häufig vor, dass sich Teile der Syntaxbäume im Hinblick auf die repräsentierten Informationen überschneiden. In anderen Teilen dagegen ergänzen sie sich gegenseitig (s. Abbildung 4.2). Da aber die Gesamtstruktur des Systems interessiert, ist es notwendig die Informationen in einem einzigen Baum zu konzentrieren. Hierzu könnten die Syntaxbäume zusammengefügt werden. Da diese aber zum ersten in den Zuständigkeitsbereich der PUMA-Bibliothek fallen und zum zweiten alle Adressangaben jeweils innerhalb der einzelnen Bäume lokal sind, werden an dieser Stelle keine Änderungen vorgenommen. Statt dessen wird eine zweite Baumstruktur überlagert, die zwei Aufgaben erfüllt. Zum ersten ermöglicht sie es, Teile des Syntaxbaumes über die Klassen-, Attribut- und Methodennamen zu adressieren, da diese als die kleinsten strukturellen Elemente des Systems angesehen werden. Zum zweiten sorgt sie dafür, dass alle Syntaxbäume in einen einzigen, virtuellen Syntaxbaum integriert werden, wobei bei Doppelungen in den einzelnen Syntaxbäumen immer die Variante mit den meisten Informationen verwendet wird, so dass alle wichtigen Informationen über das System in diesem Baum enthalten sind. Die Informationen, die der virtuelle Syntaxbaum bereitstellt sowie die Verbindungen zu den realen Syntaxbäumen (gestrichelte Linien) stellt Abbildung 4.3 dar.

Zur effizienten Speicherung des virtuellen Syntaxbaumes und zur Verwaltung der über das Gesamtsystem gesammelten semantischen Informationen werden diese in der Strukturdatenbank abgelegt. Die Strukturdatenbank sorgt auch für eine ständige Aktualisierung des Suchindexes, über den an vielen Stellen im System die Elemente des analysierten Programmsystems sowie die mit ihnen verbundenen Teile des Syntaxbaumes addressiert werden können, um z.B. Klassenbeziehungen zu analysieren und Transformationen des Sytaxbaumes vornehmen zu können. Daneben ermöglicht die Datenbank auch die Verwendung von Alias-Namen für gespeicherte Klassen. Diese Fähigkeit kommt z.B. bei der Beschreibung von Komponenten eines Systems, das wie das PURE-System nach dem Konzept der Programmfamilien entworfen wurde zum Einsatz, um verschieden benannte Klassen des System, von denen immer nur eine gleichzeitig vorhanden ist,

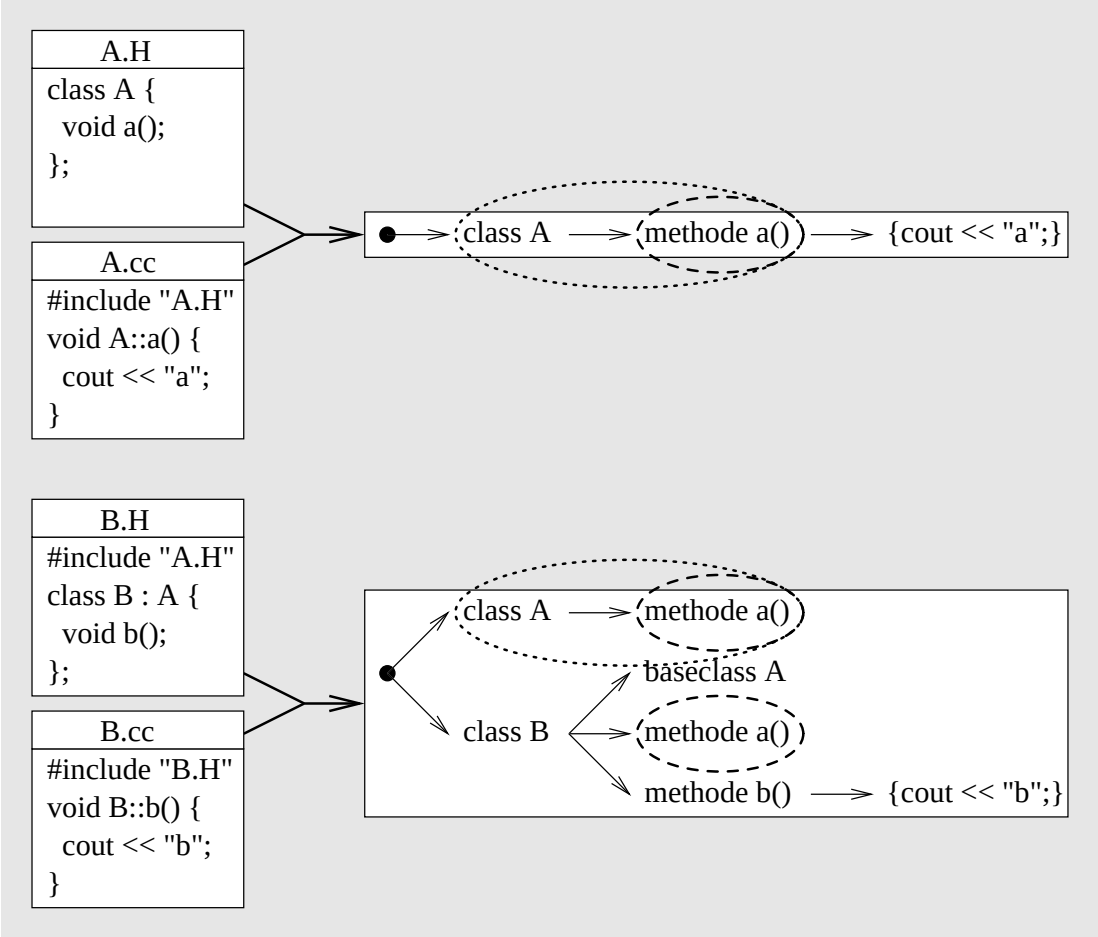


Abbildung 4.2.: Überlagerung von Syntaxbäumen

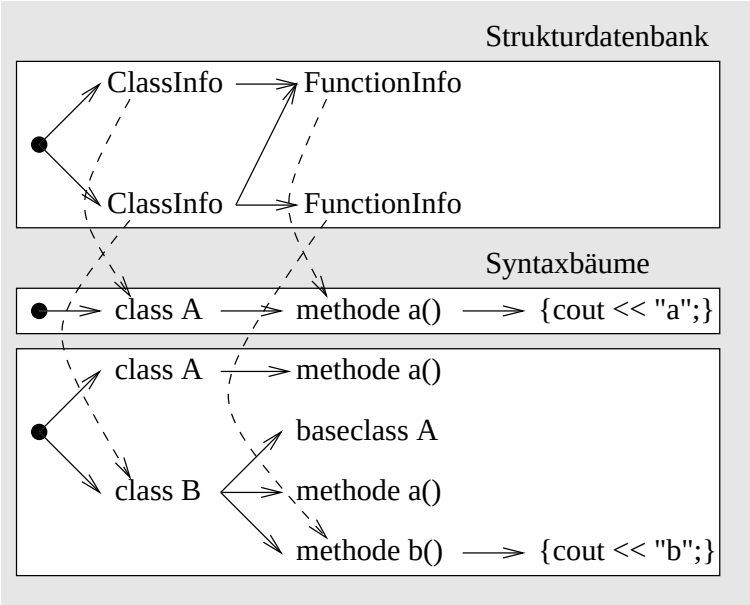


Abbildung 4.3.: Der virtuelle Syntaxbaum

über einen einzelnen Namen ansprechen zu können. Die praktische Bedeutung wird im Anwendungsbeispiel im Abschnitt 5.5 erläutert.

Im einzelnen gliedert sich der Analyseprozess in drei Phasen. In der ersten Phase, die nach jedem Durchlauf des Parsers erfolgt, wird der jeweilige Syntaxbaum traversiert und alle enthaltenen Klassen, globalen Variablen und Funktionen sowie die Basisklassen der definierten Klassen bestimmt. Als Ergebnis ergibt sich eine Liste von `ClassInfo`-Objekten, die jeweils eine Liste von Namen der Basisklassen enthalten. Eine Ausnahmebehandlung erfahren die globalen Variablen und Funktionen. Sie werden einer Pseudo-Klasse zugeordnet. Das entsprechende `ClassInfo`-Objekt enthält zusätzlich je eine Liste von `FunctionInfo`- und `AttributeInfo`-Objekten, die vorerst nur die Verweise auf die Positionen der entsprechenden Funktionen und Attribute im Syntaxbaum speichern.

Die zweite Phase folgt unmittelbar nach der ersten, d.h. auch nach jedem Durchlauf des Parsers. In der zweiten Phase wird versucht, die Liste der erkannten Klassen in die Strukturdatenbank einzutragen. Dabei wird vorausgesetzt, dass Klassennamen im gesamten System eindeutig sind. Ist eine Klasse noch nicht in der Datenbank enthalten, was über einen Namensvergleich mit den gespeicherten Klassen festgestellt werden kann, so wird sie eingefügt und es wird mit der Analyse der Klassenelemente wie Attribute und Methoden fortgefahren. Ist die Klasse schon in der Datenbank enthalten, so kann auf die Analyse der Elemente weitestgehend verzichtet werden, da die Deklaration der

Klasse in jedem Fall gleich sein muss. Der einzige Unterschied kann darin bestehen, dass eine Methode nicht nur deklariert sondern auch definiert wird. Aus diesem Grund wird die Analyse der Klassenelemente auf die Suche nach Methodendefinitionen (Implementationen) beschränkt und alle gefundenen Definitionen in das bereits bestehende `ClassInfo`-Objekt eingetragen.

Die Analyse der Klassenelemente durchsucht den Syntaxbaum nach allen Deklarationen von Attributen und Methoden. Wurde eine Attributdeklaration gefunden, so wird ein `AttributeInfo`-Objekt erzeugt und zur Attributliste der aktuellen Klasse hinzugefügt. Unterschieden wird dabei nach neu deklarierten und ererbten Attributen. Zusätzlich wird der Name der Klasse gespeichert, in dem das Attribut erstmalig deklariert wurde. Für jede Methodendeklaration wird ein neues `FunctionInfo`-Objekt in die Liste der Methoden der aktuellen Klassen eingefügt. Auch hier wird nach einer neuen oder ererbten Deklaration unterschieden. Da C++-Methoden gleichen Namens aber unterschiedlicher Signatur, d.h. verschiedener Argument- und Rückgabetypen erlaubt sind, ist es notwendig, diese unterscheiden zu können. Da die Liste der Argumente und der Rückgabewert später nicht weiter benötigt werden, wird mit einer speziell angepassten Hash-Funktion ein Hash-Wert der Signatur bestimmt und gespeichert. Dies ermöglicht später einen schnellen Vergleich von zwei Methoden. Die Argument- und Rückgabetypen werden zwar nicht im einzelnen gespeichert, sie werden aber der Reihe nach daraufhin untersucht, ob sie eine andere Klasse benutzen. Diese Information ist für die spätere Analyse der Kommunikationsbeziehungen zwischen den Klassen wichtig und so werden die Namen der benutzten Klassen gespeichert. Zusätzlich zu den bisherigen Informationen wird noch der Name der Klasse, in der die Methode erstmalig deklariert wurde, ein Verweis auf den Fundort im Syntaxbaum und der Name der Quelltextdatei (insofern bekannt) gespeichert. Der Verweis auf den Syntaxbaum stellt sicher, dass möglicherweise notwendige weitere Untersuchungen des Syntaxbaumes später effizient möglich sind. Die Speicherung des Namens der Quelltextdatei ermöglicht es, den gespeicherten Syntaxbaum zu löschen und bei Bedarf zu rekonstruieren. Dies setzt natürlich voraus, dass sich der Quelltext in der Zwischenzeit nicht verändert. Nach der Rekonstruktion des Syntaxbaumes müssen lediglich die Verweise auf die verschiedenen Knoten des Baumes angepasst werden. Aus Sicht eines Betrachters oberhalb der Datenbank und des Strukturbaums ist der Prozess des Entfernes und Rekonstruierens eines Syntaxbaumes jedoch vollkommen gekapselt.

Von der Möglichkeit des Entfernens und der Rekonstruktion von Syntaxbäumen wird bislang kein Gebrauch gemacht. Es ermöglicht aber die Erweiterung des Systems um eine problemadäquate Speicherverwaltung, ohne die die Analyse von sehr großen Projekten kaum möglich ist, da die Syntaxbäume ein Vielfaches an Speicherplatz gegenüber den Quelltextdateien benötigen.

Die dritte Phase ist getrennt von den beiden ersten und erfolgt erst nachdem alle Teile des Systems die ersten beiden Phasen durchlaufen haben und somit alle Informationen über

das System verfügbar sind. Ziel der dritten Analysephase ist es, die Informationen über die Struktur des Systems für die späteren Auswertungs- und Transformationsschritte so aufzubereiten, dass jederzeit alle Infomationen über die verschiedenen Beziehungen (s. Abschnitt 3.3.1) zu anderen Klassen verfügbar und gültig sind, d.h. dass auf sie unmittelbar und ohne weitere Prüfungen sicher zugegriffen werden kann.

Die Abschnitte der dritten Phase im einzelnen:

4.1.2.1. Prüfung der Sinnhaftigkeit

Ein Ziel der dritten Phase war es, den Zugriff auf die einzelnen Informationen ohne weitere Prüfungen zu ermöglichen. Dies bedeutet insbesondere, dass alle gespeicherten Zeiger auf Knoten des Syntaxbaumes gültig sein müssen. Um dies sicherzustellen, werden alle Einträge der Strukturdatenbank auf die Gültigkeit und Vollständigkeit der von ihnen gespeicherten und referenzierten Informationen untersucht.

Die Erkennung z.B. von Klassendefinitionen erfolgt beim Traversieren des Syntaxbaumes anhand spezieller Syntaxknoten. Ausgehend von diesen erfolgt danach die Analyse der Klasseneigenschaften. Dabei wird von einer bekannten Struktur des Syntaxbaumes ausgegangen. Entspricht diese Struktur nicht den Erwartungen, so kann es passieren, dass einige Informationen nicht ermittelt werden können. Diese Situation kann z.B. eintreten, wenn die Eingabe des Parsers syntaktische Fehler aufweist, die in einem unvollständigen Syntaxbaum resultieren oder Sprachkonstrukte verwendet werden, die bislang nicht berücksichtigt wurden. Fehlen nach der Analyse wesentliche Informationen, so werden die Einträge gelöscht. Fehlen dagegen unkritische Informationen, so werden sie durch Standardwerte ergänzt.

4.1.2.2. Namensauflösung

Bisher wurden alle Verweise auf andere Klassen in Attributen und Methoden nur in Form von Namen gespeichert. Erst nach dem Abschluss der ersten beiden Phasen werden sie durch richtige Verweise auf die entsprechenden `ClassInfo`-Objekte ersetzt. Diese Maßnahme scheint auf den ersten Blick überflüssig zu sein, da der Parser seinerseits in der semantischen Analyse eine Namensauflösung der im Syntaxbaum auftretenden Bezeichner durchführt. Bereits unter 4.1.2 wurde erläutert, dass ein System häufig auf mehrere Dateien und somit Syntaxbäume verteilt sein kann und sich dadurch Doppelungen in Teilen der Syntaxbäume ergeben. Dies tritt besonders häufig bei Deklarationen auf. Dies hat jedoch zur Folge, dass die Verwendung eines Bezeichners von der semantischen Analyse des Parsers immer nur relativ zu dieser Deklaration aufgelöst wird, da für ihn nur dieser Baum existiert. Da aber in der Strukturdatenbank und somit dem Baum des Gesamtsystems immer nur die erste Kopie einer Deklaration vermerkt ist, müssen

auch alle Namensverweise auf diese zeigen. Diese Namensauflösung findet aber nur im virtuellen Syntaxbaum statt und verändert die realen Syntaxbäume nicht.

Betroffen von der Namensauflösung sind die Klassenangaben von Basisklassen, der ursprünglichen Deklaration von Methoden und Attributen sowie von Typen von Attributen und Parametern bzw. Rückgabewerten von Methoden.

4.1.2.3. Auflösung von Nutzungsbeziehungen

Unter den Nutzungsbeziehungen werden alle Beziehungen zwischen zwei Klassen verstanden, bei denen der Nutzer über die Schnittstelle der zweiten Klasse auf deren Elemente zugreift. So wird eine Klasse durch eine andere benutzt, wenn diese direkt auf die Attribute zugreifen kann, wie dies oft bei abgeleiteten Klassen (auch über mehrere Stufen) der Fall ist. Eine Nutzungsbeziehungen liegt ebenfalls vor, wenn eine Klasse eine andere als Typ für ein Attribut oder die Argumente bzw. den Rückgabewert einer Methode verwendet.

Da für verschiedene Aufgaben wie z.B. die Komponentenanalyse Angaben darüber benötigt werden, mit welchen anderen Klassen eine untersuchte Klasse in verschiedenen Beziehungen steht und diese Informationen nicht immer direkt verfügbar sind, weil sie sich unter anderem erst aus der Analyse der Parameter der Methoden ergeben, werden sie in diesem Schritt gesammelt und gesondert gespeichert.

4.2. Beschreibung des Systems

Nach der Analyse des Ausgangssystems folgt mit der Beschreibung des Systems der zweite Schritt auf dem Wege zu einem Systemmonitor. Diese Beschreibung bildet die Voraussetzung dafür, in den weiteren Schritten Aspektprogramme zu entwickeln, die möglichst unabhängig vom zu transformierenden System sind.

Wie bereits in Abschnitt 3.2.2 erläutert, existieren zwei mögliche Ebenen der Beschreibung von Systemkomponenten. Die Basisschicht arbeitete auf den kleinsten Elementen des System und ermöglicht eine höhere Schicht, die mit Gruppen dieser kleinsten Elemente operiert. So umfasst auch das im Rahmen dieser Arbeit realisierte System Möglichkeiten zur Arbeit auf beiden Beschreibungsstufen. Diese Möglichkeiten werden in den folgenden Abschnitten dargestellt. Die praktische Anwendung wird dann im Kapitel 5 an einer Anwendungsfallstudie näher erläutert.

4.2.1. Zuweisung von Komponenten

Dieser Abschnitt beschäftig sich mit der Beschreibung von Komponenten auf Basis von Gruppen kleinster Elemente. Diese Gruppen können künstlich definiert werden oder man greift auf bereits bestehende Gruppierungen wie Klassen und Objekte zurück.

Für die meisten Anwendungen ist die Definition von Komponenten auf der Basis von Klassen hinreichend. Für einige Spezialanwendungen (s. Abschnitt 5.3) ist es jedoch notwendig, Mengen der kleinsten Elemente anzugeben. Hierbei müssen alle Elemente einer Komponente einzeln in eine Tabelle eingetragen werden, die dann die Komponente beschreibt.

Wie bereits in Abschnitt 3.1 ausgeführt, können Komponenten auf der Klassenebene anhand von drei Kriterien unterschieden werden, die zusammen in einem einzelnen `ComponentInfo`-Objekt gespeichert und der jeweiligen Klasse zugeordnet werden. Es wurde aber auch gezeigt, dass diese Kriterien in ihrer Bedeutung unterschiedlich stark zu wichten sind. Da aber nur die Praxis, d.h. das vorliegende System und die zu realisierende Monitoring-Aufgabe, über die wahre Gewichtung entscheiden können, kann bei der Zuweisung von Komponenten nach den drei Kriterien nach zwei Strategien verfahren werden.

Als erstes besteht die Möglichkeit, `ComponentInfo`-Objekte mit allen benötigten Kombinationen von Werten für die einzelnen Kriterien zu erzeugen und nach den drei beschriebenen Zuordnungsstrategien zuzuweisen. Diese Möglichkeit ist für die Praxis meist viel zu aufwendig, da das Kriterium der funktionalen Gruppierung häufig dominiert und nicht alle Zuordnungsstrategien für jedes Kriterium gleichermaßen benötigt werden.

Dies führt zur zweiten Möglichkeit, die für die häufigsten Anwendungsfälle optimiert ist. Dabei werden ebenfalls eine Reihe von `ComponentInfo`-Objekten erzeugt, diesmal allerdings nur je eins für die verschiedenen funktionalen Gruppierungen. Diese werden dann anschließend mit den drei Strategien den Klassen des Systems zugewiesen. In einem zweiten Schritt werden die Werte der anderen Kriterien manuell und optional durch die Vererbungsstrategie unterstützt zugewiesen.

Um die Verwendung von Komponenten für den Anwender intuitiver zu gestalten, können die einzelnen Komponenten auch benannt werden. Dazu wird die Abbildung der Namen in die interne Darstellung in einer Komponentendatenbank gespeichert.

Der gesamte Vorgang der Beschreibung von Komponenten sowie die Verwendung von Komponentennamen wird durch die Perl-Nutzerschnittstelle (s. Abschnitt 4.3.1.3) wesentlich vereinfacht. Auch werden in dieser Schnittstelle die technischen Details zur automatischen Vervollständigung einer Komponentenbeschreibung vollständig gekapselt.

4.2.2. Beschreibung von Systemgrenzen

Unter dem Begriff *Systemgrenzen* werden in diesem Zusammenhang alle Übergänge zwischen zwei Komponenten des Systems verstanden. Da Komponenten und Grenzen unmittelbar miteinander verbunden sind und es in der Praxis wesentlich einfacher ist, die Komponenten bzw. die Menge der kleinsten Elemente einer Komponente zu beschreiben, werden Grenzen nicht gesondert beschrieben. Es wird somit bei jedem Übergang zwischen zwei kleinsten Elementen geprüft, ob sie zu verschiedenen Komponenten gehören. Welche Übergänge dabei existieren und wie sie zu erkennen sind wird in Abschnitt 4.3.3 näher erläutert.

4.3. Transformation des Systems und das Aspektweben

Nachdem die ersten beiden Vorbereitungsschritte absolviert wurden kann nun mit der „eigentlichen“ Arbeit begonnen werden. Der interessanteste Teil eines Systems zur aspektorientierten Programmierung ist zweifelsohne der Aspektweber und die durch ihn verwendeten Aspektprogramme. Grund hierfür ist die Tatsache, dass es noch sehr wenige Erfahrungen im Umgang mit der aspektorientierten Programmierung im Allgemeinen und den Entwurfsprinzipien einer Aspektsprache gibt. Um die Idee des in dieser Arbeit verfolgten Ansatzes besser einordnen zu können, gibt der folgende Abschnitt einen kurzen Überblick über andere Ansätze und Systeme.

4.3.1. Aspektprogrammiersprachen

Bisherige Ansätze in der aspektorientierten Programmierung gehen von einer klaren Trennung von Aspektprogramm und Aspektweber aus, d.h. der Aspektweber ist ein eigenständiges Programm, das ein Programm in einer beliebigen Aspektsprache sowie das zu transformierende Programm in einer Quellsprache als Eingabe nimmt und daraus ein einziges Programm in einer Zielsprache erzeugt.

In [K+97] werden zwei Beispiele für die Gestalt von Aspektprogrammen gegeben, die jeweils eine dem Problem angepasste Aspektsprache und einen eigenen Aspektweber verwenden. Es ist klar zu sehen, wie sich auf verschiedene Arten Probleme effizient lösen lassen. Der entscheidende Nachteil dieser Ansätze ist die starke Orientierung an den Problemen, die mit dem Programm in der Quellsprache gelöst werden, was eine universellere Verwendung einschränkt.

Einen grundsätzlich anderen Ansatz verfolgt das *AspektJ*-Projekt [Xer99]. Hier wird das Standard-Java um zusätzliche Sprachkonstrukte erweitert, die es erlauben, Apekte in Java auszudrücken. Der verwendete Aspektweber versteht diese Java-Erweiterung und verwebt Quell- und Aspektprogramm zu einem einzigen Java-Programm. Dieser

Ansatz ist recht universell und erleichtert es dem Entwickler durch die Erweiterung des Java-Sprachstandards sehr, Aspekte zu formulieren, da die Umgewöhnung auf eine andere Sprache entfällt. Im Gegenzug werden durch die Erweiterung des Sprachstandards die Möglichkeiten auch ziemlich eingeschränkt, da die Sprache der Aspektprogramme kaum Freiraum für den Ausdruck von Systemeigenschaften lässt, die sich nicht direkt im eigentlichen Java-Programm verwirklichen lassen. So sind die beschriebenen Aspekte auch sehr stark an das zu transformierende Programm gebunden. Auch ist zu bemerken, dass sich durch die AspektJ-Erweiterungen sehr vielfältige Aspekte modellieren lassen, dieser Prozess aber wenig intuitiv ist, so dass es einem durchschnittlichen Anwender sicher schwer fällt, das Potential dieses Ansatzes auszunutzen.

Abschließend ist mit MDL [H+97] ein dritter Ansatz zu erwähnen. Ziel des Projektes ist zwar die dynamische Instrumentierung von Programmen, die verwendeten Methoden lassen sich mit der aspektorientieren Programmierung vergleichen, wenn man von der starren Bindung an das Einsatzfeld absieht, das die Instrumentierung von Programmen zur Ausmessung von Systemeingenschaften vorsieht. Ausgangspunkt dieses Ansatzes ist die Beschreibung einer Messung durch Definition einer anzuwendenen Metrik. Aus der Analyse des zu instrumentierenden Systems und der Metrikdefinition ermittelt das System zum einen was im System zu instrumentieren ist und zum anderen auch wo dies erfolgen soll. Danach wird das Ausgangssystem auf der Maschinencodeebene um die Instrumentierung erweitert. Das Besondere an diesem Ansatz ist die Orientierung an einer ganzen Klasse von zu lösenden Problemen, d.h. dieser Ansatz ermöglicht es, eine Vielzahl von Messaufgaben an einem System zu realisieren, bei dessen Entwicklung derartige Erweiterungen nicht vorgesehen wurden.

4.3.1.1. Ansatz dieser Arbeit

Der in dieser Arbeit verwendete Ansatz verwendet keine so klare Trennung von Aspektweber und Aspektprogramm, vielmehr besteht ein fließender Übergang zwischen beiden Teilen. Grund hierfür ist die Tatsache, dass der Aspektweber nicht wie in den bisherigen Ansätzen dargestellt ein Aspektprogramm als Eingabe verarbeitet, sondern durch das Aspektprogramm direkt angesteuert wird, d.h. der Aspektweber bzw. der Teil des Systems, der die grundlegenden Transformationen des Syntaxbaumes vornimmt, liegt als eine Bibliothek von Funktionen vor, die nach dem Baukastenprinzip zu immer komplexeren Transformationen kombiniert werden können, die dann von weiteren Programmteilen zur Beschreibung von Systemtransformationen verwendet werden können. Deshalb ist es auch rein formal möglich, zwischen je zwei Schichten (s. Abbildung 4.4) eine Grenze zu ziehen und den unteren Teil als den Aspektweber und den oberen Teil als das Aspektprogramm anzusehen.

Ziel dieses Ansatzes ist es, die Möglichkeiten für die Beschreibung von Aspekten nicht schon frühzeitig durch die Festlegung einer für den Anwender verbindlichen Aspektspra-

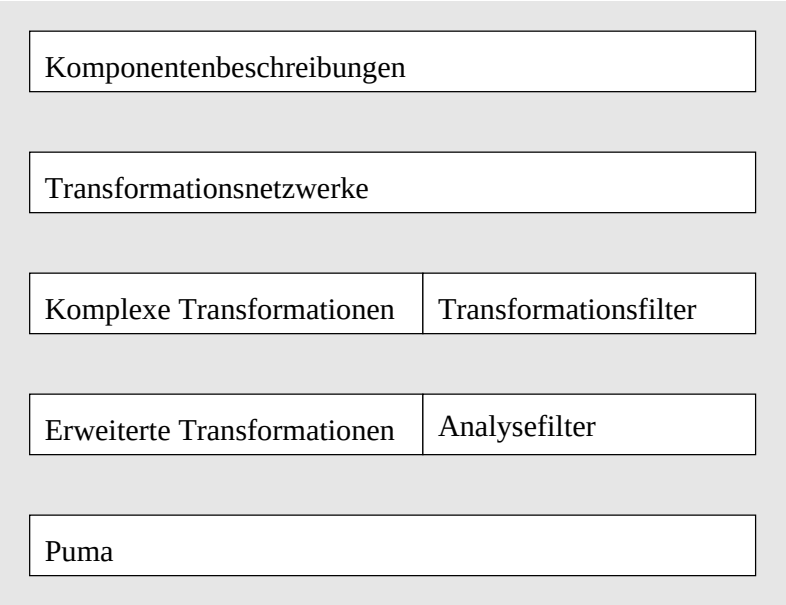


Abbildung 4.4.: Schichtenmodell des AOP-Ansatzes

che zu begrenzen. Die Beweggründe hierfür sind leicht einsichtig, wenn man bedenkt, wie wenig praktische Erfahrungen in der aspektorientieren Programmierung bisher gesammelt wurden, was auch bedeutet, dass es noch keine allgemeingültige Vorstellung davon gibt, wie man Aspekte beschreiben sollte. Somit können aufbauend auf den verschiedenen Schichten möglicher Transformationen verschiedenste problemadäquate Aspektweber-Aspektsprachen-Kombinationen entwickelt werden.

4.3.1.2. Schichtenmodell

Abbildung 4.4 gibt einen Überblick über die bisher im System vorzufindenen Schichten der AOP-Komponente. Aus Sicht der in Kapitel 5 aufgeführten Anwendungsbeispiele könnte man die unteren drei Schichten als den Aspektweber und die oberen beiden als das Aspektprogramm bezeichnen. Die Funktionen der verschiedenen Blöcke im einzelnen:

Puma bietet die elementaren Transformationsoperationen auf dem Syntaxbaum

Erweiterte Transformationen abstrahieren die elementaren Transformationen und kapseln viele der zu deren Anwendung notwendigen technischen Details

Analysefilter werten die syntaktischen und semantischen Informationen des Syntaxbaumes aus und ermöglichen so die bedingte Ausführung von Transformationen

Komplexe Transformationen setzen auf den erweiterten auf und führen ganze Blöcke von Transformationen durch, die meist durch weitere Informationen parametriert werden

Transformationsfilter ermöglichen die bedingte Ausführung komplexer Transformationen in Abhängigkeit von bestimmten Eigenschaften des Syntaxbaumes (meist in Verbindung mit den Informationen, die die komplexen Transformationen parametrieren)

Transformationsnetzwerke sind beliebige Verknüpfungen von erweiterten und komplexen Transformationen sowie verschiedensten Filtern (s. Abschnitt 4.3.2)

Komponentenbeschreibungen parametrieren die Transformationsnetzwerke und steuern so die Transformation eines gegebenen Systems

Im Kapitel 5 werden die Möglichkeiten und der praktische Einsatz verschiedener Transformationen und Komponentenbeschreibungen zur Realisierung unterschiedlicher Monitoring-Aufgaben ausführlich behandelt.

4.3.1.3. Aspektprogrammierung mit Skriptsprachen

Bevor im folgenden Abschnitt das Konzept der Realisierung von Aspektprogrammen näher erläutert wird, noch ein paar Worte zur verwendeten Programmiersprache. Wie bereits ausgeführt, baut der in dieser Arbeit verfolgte Ansatz auf einer schichtartig strukturierten Bibliothek von Transformationsfunktionen auf. All diese Funktionen sind in C++ implementiert, um die Leistungsfähigkeit einer nativen Programmiersprache auszunutzen. Der bisher fehlende Punkt ist die Klärung des Zugriffs auf diese Funktionen sowie die Beschreibung dieses Zugriffes – das Aspektprogramm. In den bisherigen Systemen tritt an dieser Stelle das als Aspektweber bezeichnete Programm in Aktion, das nach den Angaben des gegebenen Aspektprogramms die verschiedenen Transformationsfunktionen aufrufen würde. Wie bereits dargestellt, sollen bei diesem Ansatz die Transformationsfunktionen durch das Aspektprogramm direkt aktiviert werden. Somit muss das Aspektprogramm auch direkt auf die Transformationsfunktionen zugreifen können.

Für den Zugriff auf die Bibliotheksfunktionen (s. Abbildung 4.5) kann ebenfalls C++ verwendet werden. C++ hat jedoch für den praktischen Einsatz eine Reihe von Nachteilen, wie strenge Typisierung und ein recht zeitaufwendiges Übersetzen und Linken. Abhilfe schaffen hier Skriptsprachen, die es ermöglichen, dynamisch Programmteile aus Bibliotheken nachzuladen. So wird im Rahmen dieser Arbeit *Perl*[1] als die eigentliche

[1] Liebhaber anderer Programmiersprachen werden durch diese Wahl nicht benachteiligt, da für die Erzeugung der Perl-C++-Schnittstelle *SWIG* (URL:http://www.swig.org) verwendet wird, das gleichartige Schnittstellen für die meisten Skriptsprachen erzeugen kann.

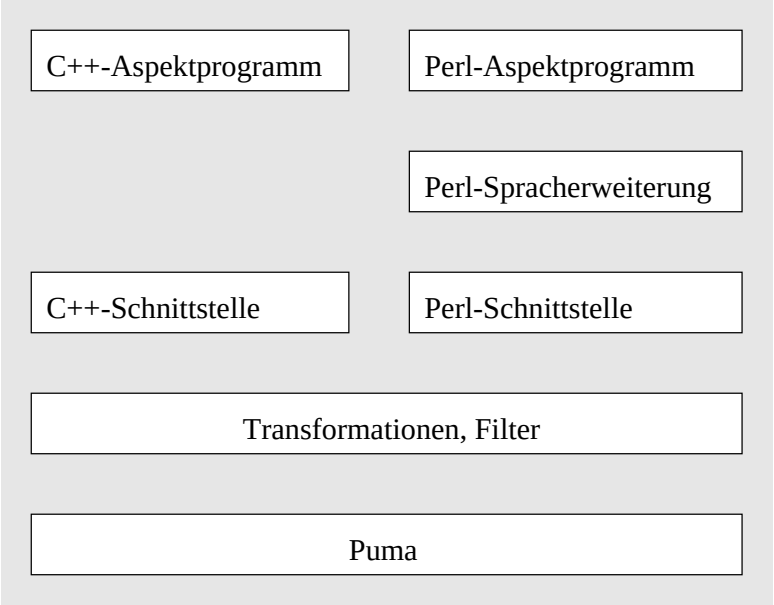


Abbildung 4.5.: Schnittstelle zu den Transformationsfunktionen

Programmiersprache für Aspektprogramme verwendet. Die Gründe für diese Wahl sind vielfältig. So ist Perl eine der am häufigsten eingesetzten Skriptsprachen. Dazu kommt, dass Perl von der Syntax her in vielen Punkten mit C und C++ übereinstimmt, der Programmierer muss sich somit kaum umgewöhnen. Der entscheidende Grund jedoch ist das dynamische Typsystem von Perl (und anderen Skriptsprachen), das es erlaubt, nicht einfach nur eine simple Schnittstelle zur Bibliothek der Transformationen zu definieren, sondern diese Schnittstelle mit vielen Zusätzen zu versehen, die den Umfang eines Aspektprogramms in Perl im Vergleich zu C++ deutlich reduziert. Nebenbei verkürzt sich die Entwicklungszeit von Aspektprogrammen durch die Verwendung der Skriptsprache deutlich, da der zeitaufwendige Teil des Übersetzungsprozesses nur einmalig für die Schnittstelle von Perl zu C++ erfolgen muss und nicht nach jeder Änderung des Aspektprogrammes.

Wie bereits unter 4.3.1.1 erklärt, ist die Idee hinter dem hier entwickelten Ansatz die Möglichkeit zur Implementierung verschiedenster Aspektprogramm-Aspektweber-Kombinationen. Für viele Anwendungen läuft dies auf eine beliebig geartete Beschreibung von Systemtransformationen hinaus. In den meisten Fällen sind diese Beschreibungen (wie z.B. die Komponentenbeschreibungen) aber nicht so kompliziert, dass sich die Entwicklung eines Parsers zur Erkennung der entwickelten Sprache lohnen würde. In diesem Fall ist der Einsatz einer Skriptsprache wie Perl, die sehr gut mit textuel-

len Eingabedateien umgehen kann, zur Analyse der Beschreibung lohnenswert. Somit liegt es auch aus diesem Grund nahe, eine Perl-Schnittstelle zur verwendeten Schicht der Transformationsbibliothek zu schaffen. Durch die speziellen Eigenschaften von Perl lässt sich die Perl-Schnittstelle sogar so gestalten, dass sie wie eine Spracherweiterung von Perl wirkt. Dies bedeutet im Resultat, dass man eine beliebig erweiterbare spezifische Aspektprogrammiersprache erhält, die „nebenbei“ die Vorzüge der Vielzwecksprache Perl bietet.

Noch ein paar Worte zu den Konsequenzen des Einsatzes einer Skriptsprache auf den Ressourcenverbrauch. An dieser Stelle steht die Ausführungsgeschwindigkeit an erster Stelle. Hier ergibt sich keine nennenswerte Einbuße, da die zeitaufwendigen Funktionen in C++ implementiert wurden. Da die Skriptsprache nur als Schnittstelle fungiert, um die C++-Funktionen indirekt aufzurufen, ist der hierdurch verursachte Mehraufwand vernachlässigbar. Der zweite Punkt betrifft den Speicherplatzverbrauch des Systems. Der Umfang der C++-Perl-Schnittstelle beträgt ca. 25% der Größe der Transformationsbibliothek und wächst bei Erweiterungen im gleichen Maßstab mit. Dies mag auf den ersten Blick sehr viel wirken, vergleicht man es hingegen mit dem Ressourcenverbrauch zur Transformation z.B. des PURE-Systems, so fällt der Mehraufwand deutlich unter 1% und ist somit ebenso zu vernachlässigen.

4.3.2. Filter, Transformationen und Zustandsspeicher

Basis für den in dieser Arbeit entwickelten Ansatz zur aspektorientieren Programmierung sind die Fragen danach, an welchen Stellen des Programms welche Transformationen vorgenommen werden sollen. Daneben orientiert sich der Ansatz streng am Prinzip eines Baukastens, der es erlaubt, komplexe Transformationen aus einer Menge einfacherer Transformationen zu konstruieren. Aus diesen Vorgaben ergibt sich die grundlegende Struktur eines Aspektprogramms wie in Abbildung 4.6 dargestellt. Gleichzeitig ist dort die im Folgenden verwendete Notation abgebildet.

4.3.2.1. Mustersuche

Die Mustersuche übernimmt die Aufgabe potenzielle Transformationsstellen zu finden. Dafür wird für den Mustervergleich ein Syntaxbaum gegeben, der spezielle Platzhalter-Knoten enthalten kann, die beim Vergleich mit einer Menge von Knotentypen statt nur mit einem Knotentyp übereinstimmen. Die Mustersuche erledigt dabei das PUMA-System, die Adressierung der interessanten Stellen im Syntaxbaum erfolgt meist über die Informationen der Strukturdatenbank (s. Abschnitt 4.1.2). Ergebnis der Suche ist eine Liste alle Suchtreffer, die nachfolgend gefiltert und transformiert werden können.

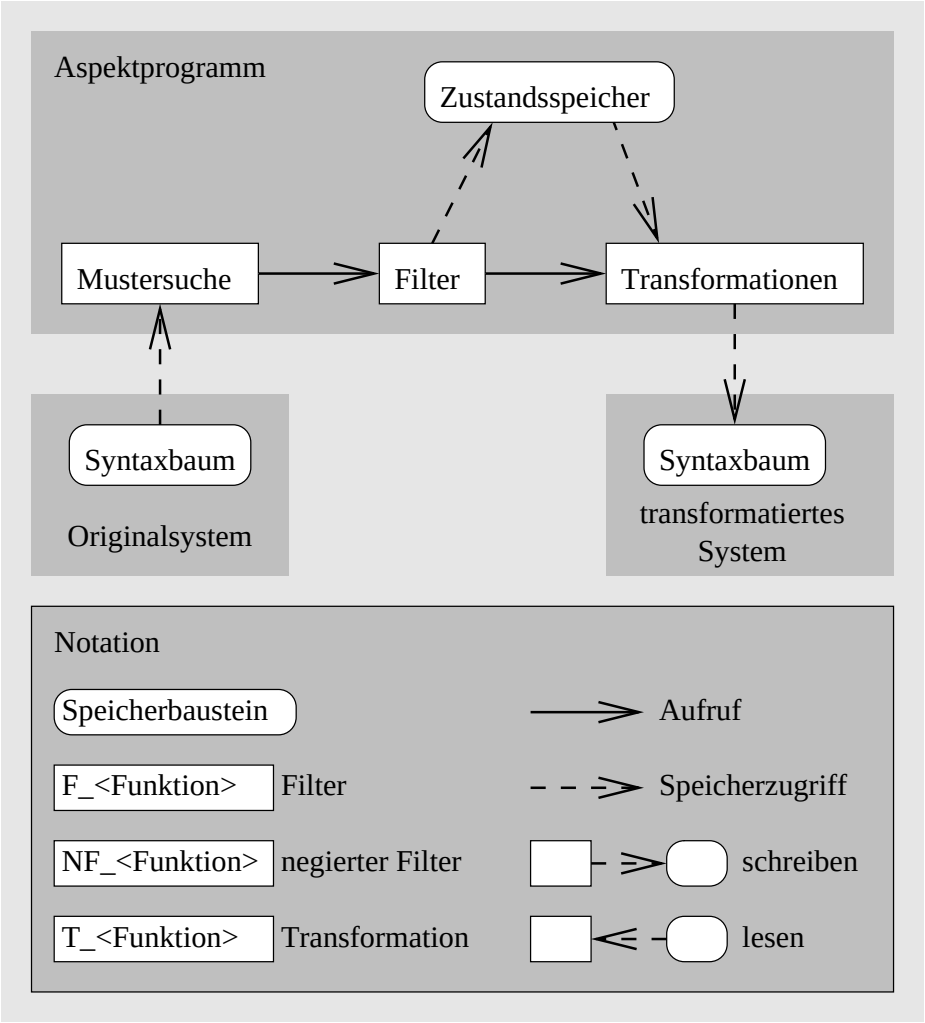


Abbildung 4.6.: Grundstruktur eines Aspektprogramms (Transformationsnetzwerk)

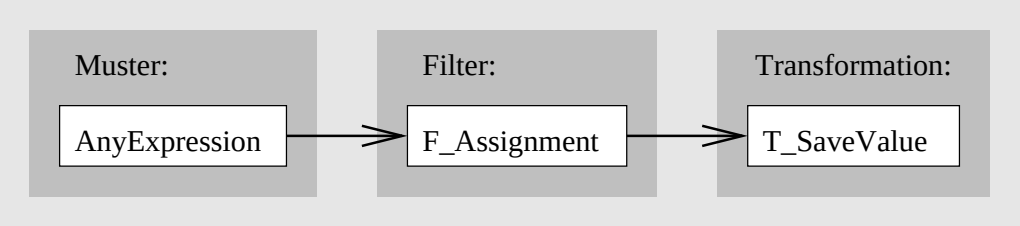


Abbildung 4.7.: Vereinfachtes Beispiel eines Aspektprogramms (Netzwerk)

4.3.2.2. Filter

Filter ermitteln aus der Menge der potentiellen Transformationsstellen diejenigen, die die für die Transformation notwendigen Voraussetzungen erfüllen und somit transformiert werden soll. Filter rufen die mit ihnen verbundenen Bausteine nur auf, wenn der aktuelle Suchtreffer die Bedingung des Filters erfüllt.

Ein vereinfachtes Beispiel für die Struktur eines Aspektprogramms zeigt Abbildung 4.7. Das Beispiel stellt die prinzipielle Vorgehensweise dar, wenn man bei allen Zuweisungen an eine Variable die Wertveränderungen aufzeichnen will. Dazu werden zuerst alle Ausdrücke eines Programms gesucht. Daraus werden alle gewünschten Zuweisungen herausgefiltert und um die Protokollierung erweitert.

4.3.2.3. Transformationen

Transformationen sind allgemein betrachtet beliebige Reaktionen auf das Finden von Mustern im Syntaxbaum mit bestimmten Eigenschaften. Meist wird eine Transformation im eigentlichen Sinne des Begriffes auch den Syntaxbaum transformieren, dies ist jedoch nicht festgeschrieben. Eine Transformation kann danach z.B. auch nur Informationen über den Suchtreffer speichern.

4.3.2.4. Zustandsspeicher

Bislang können Filter und Transformationen nur rein sequentiell verknüpft werden. Da das Aspektprogramm jedoch primär durch das Transformationsnetzwerk bestimmt wird, müssen noch ein paar weitere Konstrukte ermöglicht werden, um es praktisch nutzen zu können. Dazu zählen Scheifen und komplexe Bedingungen, die aus mehreren Teilbedingungen zusammengesetzt sind, sowie Variablen. Um diese Konstrukte zu unterstützen, existieren Zustandsspeicher als verallgemeinerte Variante einer Variablen. Eine wichtige Auswirkung haben Zustandsspeicher auch auf die Komplexität der sonstigen Bausteine

eines Transformationsnetzwerks. Durch die Möglichkeit, bereits gewonnene Informationen über die Eigenschaften des analysierten Syntaxbaumes speichern zu können, werden Redundanzen in der Funktionalität der Filter- und Transformationsbausteine vermieden, da oftmals die gleiche Information an verschiedenen Stellen im Netzwerk benötigt wird. Auch wird so eine mehrfache Analyse der gleichen Eigenschaft vermieden, was einen nicht unerheblichen Einfluss auf die Arbeitsgeschwindigkeit hat.

4.3.2.5. Transformationsnetzwerke

Die Struktur eines Aspektprogramms wird selten so einfach ausfallen, wie in Abbildung 4.7 dargestellt. Oftmals werden komplexe Bedingungen an die Eigenschaften des zu transformierenden Teils des Syntaxbaumes gestellt oder es sind mehrere Transformationen an verschiedenen Stellen des Syntaxbaumes notwendig. Da es sehr unpraktisch ist, für jede Bedingung einen eigenen Filter zu programmieren, existieren zahlreiche einfache Filter, die kombiniert werden können.

Prinzipiell kann jeder Filter mit je einer Liste von weiteren Filtern und Transformationen verbunden werden. Trifft die Bedingung eines Filters auf einen Suchtreffer im Syntaxbaum zu, so werden zuerst die verbundenen Transformationen in der vorgegebenen Reihenfolge aufgerufen, danach wird der Suchtreffer an die verbundenen Filter weitergegeben. Die Funktion eines Filters kann auch negiert werden, d.h. er gibt den Suchtreffer nur weiter, wenn seine Bedingung nicht erfüllt wird.

Zur Realisierung beliebiger boolscher Ausdrücke sind zusätzlich zur Negation die ODER- und die UND-Verknüpfung notwendig. Eine UND-Verknüpfung lässt sich leicht durch das direkte Hintereinanderschalten zweier Filter realisieren. Für eine ODER-Verknüpfung werden dagegen zwei Filter parallel geschaltet. Dabei ist es durchaus möglich, dass beide Bedingungen gleichzeitig zutreffen. In diesem Fall würden aus einem Suchtreffer zwei, die an das nachfolgende Netzwerk weitergegeben werden. Da dieser Fall nicht nur bei einer ODER-Verknüpfung sondern bei allen Verknüpfungen, die nicht nur UND-Verknüpfungen durchführen auftritt, ist eine allgemeingültige Lösung notwendig. Sie besteht darin, am Anfang der zusammengestetzten Bedingung einmal den Suchtreffer zu protokollieren und am Ende, d.h dem Punkt der Zusammenführung aller Ausgänge der Teilbedingungen, diesen genau einmal weiterzugeben. Dazu wird ein Zustandsspeicher benutzt, der den Suchtreffer zwischenspeichert. Die so entstehende Konstruktion ist in Abbildung 4.8 zu sehen.

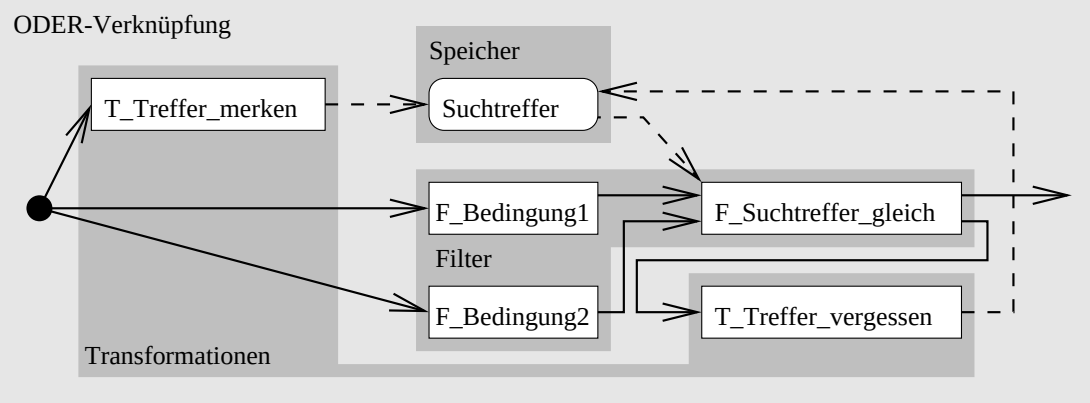


Abbildung 4.8.: Realisierung einer ODER-Verknüpfung

4.3.3. Erkennung von Systemgrenzen

Nachdem das allgemeine Funktionsprinzip der Aspektprogramme beschrieben wurde, soll im Folgenden der spezielle Aspekt des System-Monitorings betrachtet werden.

An erster Stelle steht nun wieder die Frage danach, wo eine Transformation erfolgen soll, d.h. wie man die passende Stelle im Syntaxbaum erkennt. Da prinzipell sehr viele Systemtransformationen denkbar sind, beschränken sich die Ausführungen auf die Erkennung der Punkte, die für das Monitoring von Interesse sind und auch im Sinne der Komponentenbeschreibung Grenzen darstellen könnten. Da die Syntaxanalyse ein Vorgang ist, der vor der Ausführung des Programms erfolgt, sind an dieser Stelle bei den Komponenten nur die funktionalen Gruppierungen von Bedeutung. Die Erkennung von Systemgrenzen zwischen Laufzeitgruppierungen ist Gegenstand von Abschnitt 4.3.4.

Komponentengrenzen können zwischen den kleinsten Elementen des Systems (Attributen und Methoden) bestehen. Somit kommen für eine Transformation alle Stellen in Frage, an denen ein Funktionsaufruf oder der Zugriff auf eine Speicherstelle erfolgt. Funktionsaufrufe treten in C++-Programmen an folgenden Stellen auf:

- direkter Methodenaufruf
- Verwendung überladener Operatoren
- Erzeugung, Zerstörung eines Objektes

Diese Aufrufstellen lassen sich relativ leicht erkennen. Beim Monitoring von Systemen, bei denen Unterbrechungen oder ähnliche Mechanismen wie z.B. Signale im untersuchten System selbst behandelt werden, wie dies bei Betriebssystemen eigentlich immer

der Fall ist, ist darauf zu achten, dass dieser Eintrittsweg in das System nicht erkannt werden kann. In Abhängigkeit von der Realisierung der Unterbrechungsbehandlung, des Übergangs von der Maschinen- zur C-/C++-Ebene und der zu erfüllenden Aufgabe ist diese Besonderheit vom Anwender zu behandeln. Besteht z.B. die Aufgabe, die tatsächlich verwendeten Systemteile zu bestimmen, so muss die Verwendung des jeweils ersten Eintrittspunktes auf der C-/C++-Ebene von Hand vorgegeben werden. Analog verhält es sich mit allen Stellen, an denen Aufrufe von der Maschinenebene auf die C/C++-Ebene erfolgen, da diese nicht mit den zur Verfügung stehenden Werkzeugen bestimmt werden können. Bei Systemen, die auf der Gastebene eines Betriebssystems laufen, sind entsprechende Übergänge von den Betriebssystemschnittstellen ebenfalls vorzugeben.

Nicht eindeutig aufzulösen sind Aufrufe zu virtuellen Methoden, wenn die verschiedenen möglichen Implementierungen in Klassen erfolgen, die verschiedenen Komponenten zugeordnet wurden. In diesem Fall bleibt nur die Erkennung zur Laufzeit (s. Abschnitt 4.3.4).

Etwas komplizierter gestaltet sich die Erkennung von Zugriffen auf Speicherstellen. Hierbei ist zuerst zwischen einem lesenden und schreibenden Zugriff zu unterscheiden. Ein lesender Zugriff erfolgt in folgenden Fällen:

- jedes Vorkommen auf der rechten Seite von Ausdrücken
- Verwendung als Argument eines Methodenaufrufs
- effektive linke Seite einer Zuweisung bei Verwendung der Operatoren `+=`, `-=`, ...
- Verwendung dereferenzierter Zeiger
- mehrfaches Dereferenzieren von Zeigern
- Verwendung der Operatoren `++` und `--`

Schreibzugriffe auf Speicherstellen erfolgen in folgenden Fällen:

- effektive linke Seite einer Zuweisung
- Zuweisung über Zeiger oder Referenzen
- Initialisierungsteil von Konstruktoren

Bei der Analyse von Zugriffen auf Speicherstellen sind noch ein paar Punkte genauer zu betrachten. Beim Dereferenzieren von Zeigern kommt es bei der statischen Analyse nicht darauf an, welchen Wert die Variable beinhaltet, sondern nur von welchem Typ die

Speicherstelle ist, auf die der Zeiger verweist. Beim mehrfachen Dereferenzieren kommt noch hinzu, dass die Betrachtung nach jedem Dereferenzierungsschritt wiederholt werden muss, da nach jedem Schritt ein neuer Lesezugriff auf die Speicherstelle, auf die zuvor verwiesen wurde, erfolgt.

Schwierigkeiten bereiten Referenzparameter, da bei einem Methodenaufruf von außen nicht erkannt werden kann, ob die übergebene Referenz tatsächlich verwendet wird. Um dies zweifelsfrei feststellen zu können, ist es notwendig, die aufgerufene Methode selber daraufhin zu untersuchen, ob die übergebene Referenz an einer erreichbaren Stelle verwendet wird. Da diese Analyse aber nur bei nicht-virtuellen Methoden möglich ist und einen nicht unwesentlichen Mehraufwand bedeutet, wird vollständig darauf verzichtet und statt dessen angenommen, dass die Angabe eines Referenzparameters auch zu einer Verwendung führt, solange die Referenz nicht als konstant deklariert wurde.

4.3.4. Erkennung von Systemgrenzen zur Laufzeit

Die folgenden Ausführungen zur Erkennung von Systemgrenzen zur Laufzeit beschränken sich wieder auf die für das allgemeine Monitoring bedeutsamen Konstellationen. Im Gegensatz zur Analyse des Quelltextes lassen sich zur Erkennung von Laufzeitgrenzen verschiedene Lösungen realiseren, von denen jeweils ein Ansatz für die verschiedenen Konstellationen vorgeschlagen wird.

Zuerst sollen wieder Grenzüberschreitungen beim Aufruf von Methoden untersucht werden. Dabei ist jeweils grundsätzlich zu unterscheiden, ob die aufgerufene Methode virtuell ist und ob verschiedene Objekte einer Klasse unterschieden werden sollen.

Im einfachsten Fall werden nur Klassen unterschieden und es gibt keine virtuellen Methoden. Hier können die Klassen zur einfachen Unterscheidbarkeit um ein Attribut, das die Komponentenzugehörigkeit speichert, erweitert werden. Werden virtuelle Methoden verwendet und soll bei einem Aufruf erkannt werden, in welcher Klasse die aufgerufene Implementierung erfolgte, so kann zum ersten die Klasse für jede virtuelle Methode um eine weitere Methode ergänzt werden, die die passende, statisch gespeicherte Komponentennummer liefert. Diese ebenfalls virtuelle Methode zur Komponentenabfrage wird dann parallel zur eigentlichen Methode redefiniert, d.h. zu jeder Variante einer virtuellen Methode existiert dann auch eine passende virtuelle Methode zur Abfrage der Komponentenzugehörigkeit.

Die zweite Möglichkeit besteht darin, eine systemweite Datenbank zu führen, die die Abbildung eines Namenspaares, bestehend aus dem aufgerufenen Methodenname und der Zielklasse, auf eine Komponente erledigt. Die zweite Variante hat den entscheidenden Vorteil, dass die Abbildung bei der Quelltextanalyse vorberechnet werden kann. Zudem verbraucht sie nur eine konstante Menge Speicherplatz, da bei der Erweiterung um

virtuelle Methoden die Methodentabellen der einzelnen Objekte anwachsen, womit der Speicherplatzbedarf linear mit der Objektanzahl wächst.

Als zweites ist der Fall zu betrachten, wenn verschiedene Objekte einer Klasse unterscheidbar sein sollen. Hier liegt die Speicherung der aktuellen (Laufzeit-)Komponentenzuordnung in einem weiteren Attribut der Objekte nahe. Bedenkt man jedoch, dass eine Unterscheidung von Objekten unter anderem dazu genutzt werden kann, festzustellen, in welchem Adressraum es sich aktuell befindet, um danach das Aufrufverfahren (direkt, Fernaufruf) zu bestimmen, so ist klar, dass die Unterscheidbarkeit ohne einen Zugriff auf das Objekt möglich sein muss. Dies lässt wieder nur die Option einer systemweiten Datenbank – einer Art Namensdienst – aller Objekte und ihrer momentanen Zuordnungen zu. Um dies zu ermöglichen, muss jede Klasse, deren Objekte unterscheidbar sein sollen, um eine Basisklasse erweitert werden, die bei der Erzeugung des Objektes dieses bei der Datenbank anmeldet und bei der Zerstörung auch wieder austrägt. Dabei ist darauf zu achten, dass diese neue Basisklasse vor allen anderen ererbt wird, um eine frühestmögliche Anmeldung und eine spätestmögliche Abmeldung zu gewährleisten.

Beim Zugriff auf Attribute von Objekten erfolgt die Erkennung von Laufzeitkomponenten analog zu den Methoden. Wesentlich schwieriger ist die Erkennung von Zugriffen auf beliebige Speicherstellen über Zeiger. Hier hilft nur ein exaktes Protokoll aller Speicherbelegungen, um jederzeit feststellen zu können, wem das Stück Speicher, auf das der Zeiger verweist, gehört.

4.3.5. Syntax-Transformationen

Das im Rahmen dieser Arbeit entwickelte System bietet zwei Klassen von Transformationen an. Die erste arbeitet auf der Ebene der Token des Quelltextes, die zweite transformiert die Struktur des Syntaxbaumes. In beiden Fällen wird dabei auf die Funktionalitäten der PUMA-Bibliothek zurückgegriffen, die durch neue Adressierungsmöglichkeiten unter Zuhilfenahme der Strukturdatenbank und die Zusammenfassung von Sequenzen von Basistransformationen erweitert werden.

Zur ersten Klasse zählen das Einfügen beliebiger Quelltextabschnitte am Beginn oder Ende von Methoden sowie das Einfügen zusätzlicher Basisklassen am Beginn oder Ende der Basisklassenliste einer Klassendeklaration. Von diesen Möglichkeiten wird hauptsächlich zum Einfügen von Sensoren Gebrauch gemacht. Dabei wird ein Sensor am Anfang einer Funktion eingefügt, um den Ein- und Austrittszeitpunkt zu protokollieren. Soll dagegen die Lebensdauer eines Objektes bestimmt werden, so wird die Klasse zusätzlich vom Sensor abgeleitet, wodurch er während der Erzeugung und Zerstörung des Objektes aktiviert werden kann.

Wie schon mehrfach angesprochen, soll der hier vorgestellte Ansatz nicht auf das Ausmessen von Systemeigenschaften beschränkt sein. So kann er auch zur Unterstützung der

Fehlersuche in einem Programm verwendet werden. Dazu sind aber oftmals Änderungen am System notwendig, die sich nicht durch das einfache Einfügen von Quelltextfragmenten am Anfang oder Ende von Funktionen realisieren lassen. Ein Beispiel ist im Abschnitt 5.6 gegeben. Dort wird gezeigt, wie ein Programm so transformiert werden kann, dass jeder Zeiger vor einer Dereferenzierung überprüft werden kann. Derartige Veränderungen setzen eine Transformation des Syntaxbaumes voraus, bei der Teile des Syntaxbaumes durch neue ersetzt werden, die ihrerseits Teile des ursprünglichen enthalten. Eine weitere Anwendung findet die allgemeine Syntaxbaumersetzung beim Einfügen von Sensoren zur Protokollierung beliebiger Ausdrücke. So wird es z.B. möglich, einen speziellen Funktionsaufruf vom Aufrufer aus zu vermessen. Ebenso ist es auch möglich, den Zeitverbrauch einer einfachen Rechenoperation zu bestimmen.

4.4. Laufzeitsystem

Als letzter, wesentlicher Block bei der Realisierung eines Systemmonitors ist das Laufzeitsystem zu betrachten. Hierbei beziehen sich die Ausführungen auf die für das Messen von Systemeigenschaften wichtigen Erweiterungen.

Das Laufzeitsystem gliedert sich in vier Teile, die in den folgenden Abschnitten näher beschrieben werden.

4.4.1. Datenquellen

Unter dem Begriff *Datenquellen* verbergen sich die Funktionen, die von den Sensoren benutzt werden, um verschiedene Systemeigenschaften zu ermitteln. Momentan sind diese die aktuelle Systemzeit und die Identifikation des aktuell laufenden Prozesses bzw. Fadens. Die Datenquellen kapseln somit die systemabhängigen Teile der Sensoren, die dadurch unabhängig vom untersuchten System werden.

4.4.2. Ereignisspeicher

Der Ereignisspeicher dient der Aufnahme und Speicherung aller zur Laufzeit durch die Sensoren erzeugten Ereignisse. Zur Gewährleistung einer minimalen Zeitverzögerung werden alle Ereignisse in einem linearen Puffer im Speicher abgelegt. Zur Unterstützung möglichst genauer Messungen kann bei der statischen Konfiguration des Ereignisspeichers deshalb detailliert festgelegt werden, welche Angaben für ein Ereignis von Interesse sind. Dabei ist zu beachten, dass in jedem System immer nur genau ein Ereignisspeicher vorhanden ist, der „hart verdrahtet“ wird, um dem Compiler bestmögliche Bedingungen bei der Codeoptimierung zu bieten. Diese Maßname ist notwendig, wenn man bedenkt, dass es bei vielen Messungen speziell an Betriebssystemen um Zeitdifferenzen im Mikro-

und Nanosekundenbereich geht. Ist der Ereignisspeicher voll, so versucht er sich je nach Konfiguration automatisch über eine Datensenke zu speichern.

4.4.3. Datensenken

Datensenken stellen die Verbindung zur Außenwelt her und ermöglichen es, Messwerte aus dem System hinaus zu transferieren. Eine Datensenke kann die Daten z.B. in eine Datei speichern oder über eine serielle Verbindung senden. Wie beim Ereignisspeicher kann es in einem System auch nur genau eine Datensenke geben.

4.4.4. Sensoren

Sensoren sind die Messinstrumente beim Monitoring eines Systems. Sie können an nahezu jeder Stelle in das System eingebracht werden und erzeugen je nach Art verschiedene Ereignisse, die dann im Ereignisspeicher abgelegt werden.

Es existieren zwei grundlegende Sorten von Sensoren. Die erste Sorte sind kleine Objekte, die über eine Variablendeklaration in einen Programmblock eingefügt werden. Sie dienen der Protokollierung des Aufenthalts in einem Programmblock bzw. des Betretens und Verlassens dieses Blockes. So werden sie hauptsächlich zur Protokollierung von Ein- und Austrittspunkten von Methoden verwendet. Ihre Funktion basiert darauf, dass C++ garantiert, dass der Konstruktor eines Objektes am Beginn seiner Gültigkeit, d.h. beim Eintritt in den Block, in dem es deklariert wird, aufgerufen wird. Analog ist garantiert, dass der Destruktor beim Verlassen des Gültigkeitsbereiches aufgerufen wird. Diese Eigenschaft ist speziell bei der Protokollierung von Methoden von Bedeutung, da diese an beliebigen Stellen verlassen werden können. Da nicht für alle Anwendungen der Ein- und der Austrittspunkt eines Blockes von Interesse ist, treten diese Sensoren in drei Varianten auf, die entweder nur jeweils ein Ereignis oder beide zusammen aufzeichnen.

Die zweite Sorte Sensoren stellen Makros dar, durch die beliebige Ausdrücke im Programm erweitert werden können. Sie rufen die zur Speicherung eines Ereignisses notwendigen Funktionen direkt auf. Diese Sensoren werden überall dort eingesetzt, wo die Ausführung beliebiger Ausdrücke protokolliert werden soll. Ein solcher Ausdruck kann z.B. eine Zuweisung, eine Rechenoperation oder ein Funktionsaufruf sein. Somit kann ein Funktionsaufruf an der Aufrufstelle protokolliert werden. Im Gegensatz zur ersten Sensorensorte ist es so möglich, die Ausführung einer Funktion nur dann aufzuzeichenen, wenn sie von einer bestimmten Stelle aus aufgerufen wird.

4.5. Erzeugen des Systems

Nach der Instrumentierung eines Systems sind nur noch ein paar Schritte notwendig, um eine praktische Messung durchführen zu können. Als erstes muss die Laufzeiterweiterung konfiguriert werden, um sie einerseits an die Laufzeitumgebung und andererseits an die Erfordernisse der Messung anzupassen. Soll eine hochpräzise Messung durchgeführt werden, so sind hierbei die Methode der Zeitermittlung und die zu speichernden Ereignisdaten besonders interessant.

Hochpräzise Messungen setzen eine hochauflösende Zeitquelle voraus. Dafür wird z.B. auf Systemen mit Pentium-Prozessoren der interne Taktzähler des Prozessors verwendet. Da nicht alle Systeme über eine derartige Zeitquelle verfügen, muss die jeweils beste Lösung vorab definiert und konfiguriert werden.

Ein weiterer Punkt, der die Präzision einer Messung beeinflusst, ist der Umfang der bei der Erzeugung eines Ereignisses zu speichernden Daten. Bei einem System wie PURE, das zur Zeit keine schwergewichtigen Prozesse unterstützt, ist es auch unnötig, eine Prozess-Identifikation zu speichern. Ebenso können in Abhängigkeit von der konkreten Messung andere Datenfelder (s. Anhang A.1) überflüssig sein. Aus diesem Grund kann die Verwendung jedes einzelnen Datenfeldes statisch konfiguriert werden. Das Weglassen einzelner Datenfelder hat zur Folge, dass die Sensoren kleiner vom Code-Umfang und schneller in der Ausführung werden, wodurch das untersuchte System weniger beeinflusst wird.

Nach der Konfiguration der Laufzeiterweiterung können diese und das instrumentierte System übersetzt und ausführbar gemacht werden.

4.6. Auswertung

Bei jeder Messung erzeugen die in das System eingebrachten Sensoren eine Reihe von Ereignissen, durch die der Systemzustand und Zustandsänderungen protokolliert werden. Diese Ereignisse liegen zunächst in einem linearen Stück Speicher vor und werden nach Abschluss der Messung durch die Datensenken in eine Datei gespeichert oder über eine serielle Schnittstelle an einen Empfänger übermittelt. Das Format des Ereignisprotokolls und der Ereignisse selber ist festgeschrieben (s. Anhang A.1). Es können jedoch eine ganze Reihe weiterer Informationen in den Datenfeldern der Ereignisse kodiert sein. In vielen Fällen sind dabei die Identifikationen und der Ereignistyp am interessantesten. Diese Angaben werden bei der Instrumentierung kodiert und in eine gesonderte Datei abgespeichert. Sie können bei der Auswertung wieder dekodiert und in eine lesbare Form gebracht werden. Da diese Aufgabe durch nahezu alle Auswertungsprogramme erledigt werden muss, wurde eine Perl-Bibliothek (`GMTracer`) entwickelt, die es ermöglicht, die Ereignisdaten aus den verschiedenen Quellen (Datei, serielle Schnittstelle) zu beziehen

und mit Hilfe der Informationen der Instrumentierung zu dekodieren. Zusätzlich werden eine Reihe von Funktionen zur Konvertierung der in den Ereignissen vermerkten Zeitstempel angeboten. Zu diesen Konvertierungen gehört z.B. die Umrechnung der Laufzeit von Prozessortakten in Millisekunden. Eine weitere wichtige Aufgabe dieser Bibliothek ist die Einrechnung des Zeitverbrauchs der Sensoren zur Korrektur der Messzeiten (s. Abschnitt 5.1.2.2). Genutzt wird diese Bibliothek durch das Programm `atrace.pl`, dass eine Kommandozeilenschnittstelle zur Ermittlung der Ereignisdaten und deren Dekodierung anbietet.

Auf Basis dieser Bibliothek wurde eine zweite Perl-Bibliothek (`GMGraph`) entwickelt, die es ermöglicht, aus dem Ereignisprotokoll verschiedene Diagramme zur Abbildung des Laufzeitverhaltens zu erstellen. Zur Nutzung dieser Bibliothek steht das Programm `tracegraph.pl` zur Verfügung, mit dessen Hilfe die meisten Diagramme im Kapitel 5 erstellt wurden.

Durch die Kompatibilität des Formats des Ereignisprotokolls zum *Jewel*-Tool der Arbeitsgruppe „Echtzeitsysteme und Kommunikation“ und dessen grundlegenden Ereignistypen ist eine Visualisierung der Messergebnisse auch mit diesem Programm möglich. Alle zusätzlich kodierten Informationen sind dort nicht zugänglich.

5. Monitoring des Pure-Systems

5.1. Vorbetrachtungen

In der Anwendungsfallstudie dieser Arbeit werden einige Messungen am bereits mehrfach erwähnten PURE-System durchgeführt. Um die Messergebnisse besser einordnen bzw. bewerten zu können sind ein paar Vorbetrachtungen über das Messumfeld bzw. die Randbedingungen der Messung notwendig.

5.1.1. Messanordnung

Die verschiedenen Mess- und Monitoringaufgaben werden zum einen auf einem nativen System und zum anderen auf der Linux-Gastebene durchgeführt. Für die Messungen, speziell für sehr präzise, kommt nur das native System zum Einsatz, da hier PURE die Hardware vollständig kontrolliert und somit die Störfaktoren minimiert werden können. Für viele andere Messungen und die sonstigen Monitoring-Aufgaben findet die Linux-Gastebene Verwendung.

Als natives System wird, soweit nicht anders gesagt, ein Pentium mit 75 MHz Takfrequenz verwendet, dessen EDO-Ram eine Zugriffszeit von 60 ns aufweist, was ohne Betrachtung von Cache-Effekten 4,5 Takten entspricht. Dies hat direkten Einfluss auf die Geschwindigkeit der Sensoren, die bei höheren Taktraten (bezogen auf die Speichergeschwindigkeit) mehr Zeit benötigen (s. 5.1.2.2). Der Transfer der Messwerte erfolgt über die serielle Schnittstelle nach dem Abschluss der Messung oder bei Erreichen der maximalen Kapazität des Ereignisspeichers. Im Standardfall wurde PURE im Develop-Modus mit Optimierungen übersetzt.

Die Messungen auf der Linux-Gastebene erfolgten auf einem Pentium II 400 MHz. Zur Minimierung von Störfaktoren wurde PURE dabei mit dem im Standard-Kernel integrierten Realtime-Scheduling ausgeführt, wodurch es nicht durch andere Prozesse verdrängt werden konnte. Die Ausgabe der Messwerte erfolgte ebenfalls nach der Messung direkt in eine Datei, wodurch für diese Messung lediglich ein Rechner benötigt wurde.

5.1.2. Mehraufwand der Instrumentierung

Durch die Instrumentierung vergrößert sich der Programmcode des Systems. Zudem verbrauchen die eingebrachten Sensoren eine bestimmte Menge Zeit.

5.1.2.1. Zusätzlicher Programmcode

In der Tabelle 5.1 ist die Größe der Laufzeiterweiterung für die beiden häufigsten Konfigurationen angegeben. Zu beachten ist, dass in beiden Fällen noch eine nicht unerhebliche Menge an Speicherplatzbedarf für den Ereignisspeicher hinzukommt. Dieser benötigt $(20 + 24 * Ereignisse)$ Byte Speicherplatz. Die Standardkonfiguration sieht 10000 Einträge vor, was einen zusätzlichen Speicherplatzbedarf von 240020 Byte bedeutet. Für die Sensoren werden im Schnitt pro erzeugtem Ereignis 100 Byte Programmcode hinzugefügt, d.h. Sensoren, die mehrere Ereignisse erzeugen, sind entsprechend größer. Die jeweils hinzugefügte Codegröße ist vom Einfügeort und stark von der Optimierung des Compilers abhängig.

Konfiguration	Code (Byte)	Data (Byte)	BSS (Byte)
native, seriell, statische Speicher	675	0	8
Unix, Datei, dynamischer Speicher	632	0	8

Tabelle 5.1.: Größe der Laufzeiterweiterung

5.1.2.2. Laufzeitverhalten der Sensoren

Für die konkrete Messung ist das Laufzeitverhalten eines Sensors wesentlich bedeutsamer als seine Größe, da der durch ihn verursachte Mehraufwand an Zeit wesentlichen Einfluss auf die Präzision einer Messung hat. Neben dem reinen Zeitverbrauch ist auch interessant, wie hoch die Varianz des Zeitverbrauches ist, da sie bestimmt, wie gut der verursachte Zeitverbrauch in der Auswertung kompensiert werden kann.

Zur Bestimmung des Laufzeitverhaltens der Sensoren wurden je ein Sensorobjekt mit einem und zwei Ereignissen sowie ein Sensoren-Makro jeweils zehnmal hintereinander ausgeführt. Zusätzlich wurde die Funktion zur Zeitermittlung ebenfalls zehnmal aufgerufen, um ihren Zeitbedarf zu bestimmen. Abbildung 5.1 zeigt den Zeitverbrauch der drei gemessenen Sensoren. Es ist gut zu sehen, welchen Einfluss der Cache dabei hat. Bei der ersten Verwendung eines Sensors dauert die Erzeugung eines Ereignisses ca. 125 Prozessortakte (inklusive Zeitbestimmung). Ab der zweiten Verwendung verkürzt sich die Zeit auf die Hälfte und bleibt von da an fast konstant.

Diese Werte können in der Auswertung hervorragend zur Fehlerkorrektur verwendet werden, da jeder Sensor später eindeutig identifiziert werden kann und so auch nachträglich

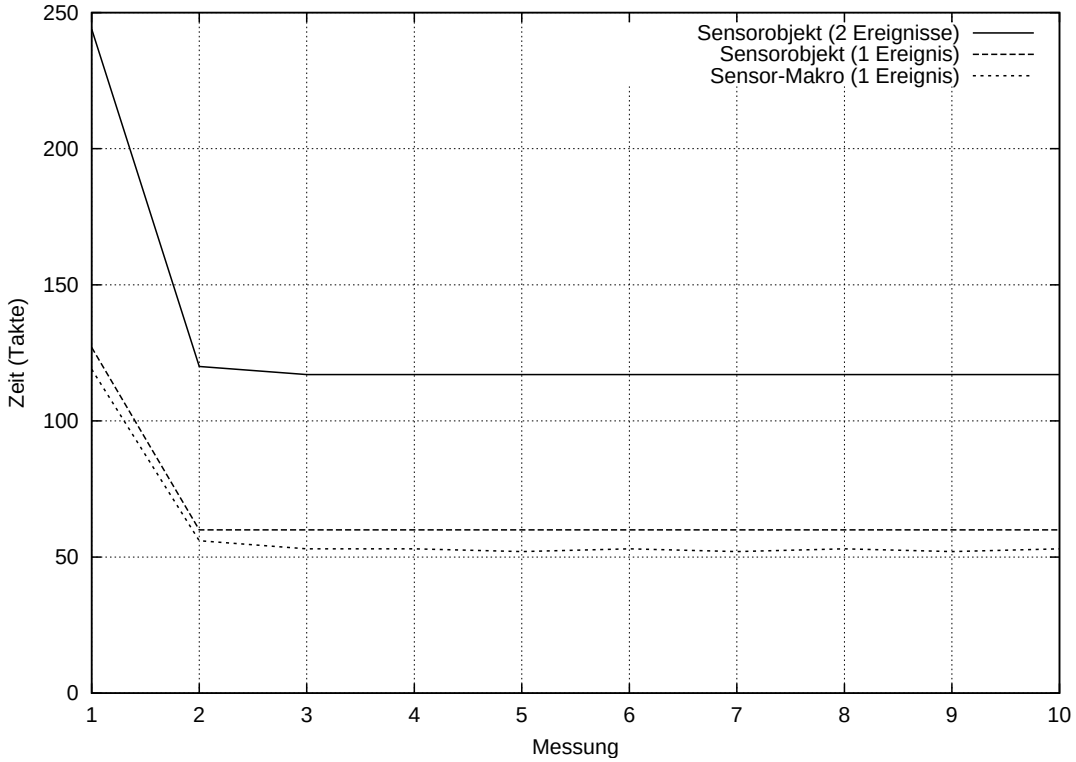


Abbildung 5.1.: Laufzeitverhalten von Sensoren

bestimmt werden kann, wie oft er bisher aufgerufen wurde. Danach kann der jeweilige Zeitverbrauch sehr gut herausgerechnet werden. Wird ein Sensor häufiger verwendet, dann reicht es meist schon, bei jeder Verwendung nur den minimalen Zeitverbrauch anzunehmen. Das dies auch für Sensoren gilt, die über das ganze System verstreut sind, zeigen die Messungen in diesem Kapitel. Grund hierfür ist die Tatsache, dass die PURE-Applikationen so klein sind, dass sie vollständig in den Cache passen.

Wie bereits erwähnt, hat der Umfang der für ein Ereignis zu speichernden Informationen direkten Einfluss auf den Zeitverbrauch des Sensors. Im oben angeführten Beispiel speichern die Sensoren nur die für eine Messung am PURE-System notwendigen Informationen (Zeit, Ereignistyp, Thread-ID). Die Felder für die Prozess-ID und den optionalen Parameter werden ignoriert. Die Aufschlüsselung des Zeitverbrauchs der einzelnen Sensoren und ihrer Teile zeigt Tabelle 5.2. Alle Werte geben den durchschnittlichen Zeitverbrauch in Taktzyklen ab der zweiten Verwendung an.

Sensor	Zeit-nahme	Ereignis-verwaltung	Ereignisspeicherung					$\sum$
			Zeit	Typ	T.-ID	P.-ID	Param.	
Objekt (2 Ereig.)	50	34	11	11	11	0	0	117
Objekt (1 Ereig.)	25	17	6	6	6	0	0	60
Makro (1 Ereig.)	25	10	6	6	6	0	0	53

Tabelle 5.2.: Zeitverbrauch der Sensorteile in Taktzyklen

Unter 5.1.1 wurde angedeutet, dass der Zeitbedarf eines Sensors auch vom Verhältnis von Prozessor- zu Speichergeschwindigkeit abhängt. Ohne dies weiter auszuführen, zeigt ein kleines Beispiel, dass dieser Zusammenhang nicht ganz zu vernachlässigen ist. So benötigt ein Sensorobjekt mit einem Ereignis bei gleichem Speicher auf einem P75 60 Takte und auf einem P133 bereits 68 Takte. Daraus folgt, dass es für eine hochpräzise Messung wichtig ist, die vorangegangenen Tests auf jedem Messsystem erneut durchzuführen.

5.2. Aufgabenstellungen

Die folgenden Aufgabenstellungen geben einen Überblick über die bisherigen Möglichkeiten und sollen helfen, dass Potential abschätzen zu können. Bei der Auswahl der Beispiele wurde deshalb mehr Wert darauf gelegt, die verschiedenen Techniken des entwickelten Ansatzes zu präsentieren, als eine umfassende Messung des PURE-Systems durchzuführen. Die Messaufgaben im Einzelnen:

Monitoring von Kontextwechseln

Im ersten Anwendungsbeispiel besteht die Aufgabe, den Kontextwechsel einer nativen PURE-Anwendung sowie die Aufrufwege dorthin durch eine hochpräzise Messung zu untersuchen. Die Systembeschreibung erfolgt dynamisch auf Basis von Funktionen, d.h. es werden keine komplexen Komponenten verwendet und die Beschreibung wird durch die Transformation selber beeinflusst.

Verweilzeit im Nukleus

Im zweiten Anwendungsfall wird der Aufenthalt im Kern (Nukleus) des Systems durch die Protokollierung der Änderungen einer Zustandsvariable untersucht. Für die Systembeschreibung wird eine einfache Komponente benutzt. Die Messung erfolgt auf der Linux-Gastebene.

Scheduling-Aufwand

Der dritte Anwendungsfall zeigt das Monitoring des Scheduling-Verhaltens bei der Verwendung verschiedener Scheduler. Zur Beschreibung des Systems wird eine einfache, konfigurationsunabhängige Komponente verwendet.

Überprüfung von Zeigerwerten

Im vierten Anwendungsfall wird die Verwendung des Monitorings zur Unterstützung einer Fehlersuche in einem bestimmten Systemteil demonstriert. Der ausgesuchte Systemteil wird dabei durch eine komplexe Komponente beschrieben, die größtenteils automatisch bestimmt wird.

5.3. Monitoring von Kontextwechseln

Als erste Anwendung soll das Verhalten bei Kontextwechseln ausgemessen werden. Für die Messung wird folgendes Programm verwendet:

Beispielprogramm Kontextwechsel

```
1  #include "thread/Genius.h"
2  #include "data/cout.h"
3  #include "thread/fame/fameScheme.h"

4  class Partner : public Bundle {
5    void action () {for (;;) {pause();}}
6  public:
7    Partner (unsigned int size) : Bundle(size) {alive();}
8  };

9  void Genius::action () {
10   Partner* son = new Partner(4096);
11   if (son) {
12     for (int loop = 0; loop < 10; loop++) {pause();}
13     delete son;
14   }
15 }
```

Für das Scheduling wird die nicht-preemtive FCFS-Strategie verwendet, d.h. als Schemer wird die Klasse `Contestant` eingesetzt. Die einzige Aktion der beiden Programmfäden besteht darin, sofort wieder die Kontrolle abzugeben. Dieses Wechselspiel wird zehnmal durchlaufen und dabei protokolliert.

5.3.1. Vorgehensweise

Zum Monitoring des Kontextwechsels ist nicht nur von Interesse, wie lange der eigentliche Wechsel dauert, vielmehr soll auch aufgezeichnet werden, wodurch der Wechsel ursprünglich ausgelöst wurde. Hierbei stellt sich die Frage, wer alles einen Kontextwechsel auslöst. Diese Frage lässt sich am besten rekursiv beantworten, denn jede Funktion, die eine andere aufruft, die ihrerseits einen Wechsel auslöst, tut dies auch selber. Zusammen mit der Feststellung, dass der grundlegende Kontextwechsel in `Coroutine::resume` erfolgt, können somit rekursiv alle zu protokollierenden Funktionen gefunden werden.

5.3.2. Transformationsprogramm

Das Transformationsprogramm gliedert sich grob in drei Abschnitte, die Projektdefinition, das Aspektprogramm und den Aspektweber. Die Übergänge zwischen den drei Teilen sind fließend - sie können sich auch durchmischen.

monitorcontext.pl (1/3: Projektdefinition)

```
1 Project({source => "pure/usr/src/sys",
2          include => "pure/usr/include",
3          application => "pure/usr/tst/approved/bundle2.cc",
4          protect => [".*GM.*", ".*.h\$","^/usr/.*"],
5          definitions => [qw(Coroutine Activity Contestant Patient Customer
6                             Jitterbug Counter Beamer Genius Contest Semaphore
7                             Timeslice pure)]
8         });
```

In der Projektdefinition werden die Pfade zu den Header- und Quelldateien sowie die zu untersuchende Applikation festgelegt. Die Angabe einer Applikation ist optional. Mit der Option `protect` werden die Dateien festgelegt, die nicht modifiziert werden dürfen. Die Option `definitions` legt fest, welche weiteren Quelltexte ebenfalls parsiert werden sollen. Als Wert kann entweder ein Feld von Namen oder `"all"` angegeben werden. Diese Option ist für den Fall konzipiert, dass alle Klassen in je einer Datei mit gleichem Namen (zuzüglich Dateiendung) gespeichert sind. Wird als Wert `"all"` angegeben, so wird versucht, alle bisher in der Datenbank nur deklarierten Klassen zu laden. Existiert eine derartige Zuordnung von Klassen zu Dateien nicht, oder sollen andere Dateien geladen werden, so ist dies über den Parameter `files` möglich. Die Angabe eines Feldes von Namen ist aber nur dann sinnvoll, wenn bereits bekannt ist, welche Klassen überhaupt für eine Transformation in Frage kommen. Diese Information kann z.B. aus einem früheren Lauf stammen und beschleunigt den Transformationsprozess deutlich.

Der zweite Teil, das Aspektprogramm, definiert das Transformationsnetzwerk, das der Aspektweber zur Transformation des Systems verwendet. Zum besseren Verständnis des im Abschnitt 2 des Quelltextes enthaltenen Aspektprogramms ist das Transformationsnetzwerk noch einmal in Diagrammform in Abbildung 5.2 dargestellt, an dem sich der Transformationsprozess leichter erläutern lässt.

Zuerst durchsucht der `Manipulator` alle Funktionen nach Funktionsaufrufen. Sobald ein solcher Aufruf gefunden wurde, prüft der Filter `F_FunctionTableLock` anhand einer gegebenen Tabelle, ob die untersuchte Funktion bereits transformiert wurde, damit dies nicht mehrfach erfolgt. Wurde sie noch nicht transformiert, so analysiert der Filter `F_FuntionTableCall` Aufrufquelle und -ziel und prüft, ob beide in den jeweils optional angebbaren Tabellen enthalten sind. Ist keine Tabelle gegeben, wird jede Quelle bzw. jedes Ziel akzeptiert. Damit wird geprüft, ob die aktuell untersuchte Funktion eine Funk-

monitorcontext.pl (2/3: Aspektprogramm)

```
10 $task = Manipulator
11   ({patternname => "AnyCall",
12     expression => "\any-call-expression("Call")"},
13    F_FunctionTableLock
14    ({table => $locktable = FunctionTable()},
15     $callfilter = F_FunctionTableCall
16     ({node => "Call"},
17      $callnote = T_NoteFunction(),
18      T_NoteFunction({table => $locktable}),
19      T_TraceFunction()
20     )
21    )
22   );
```

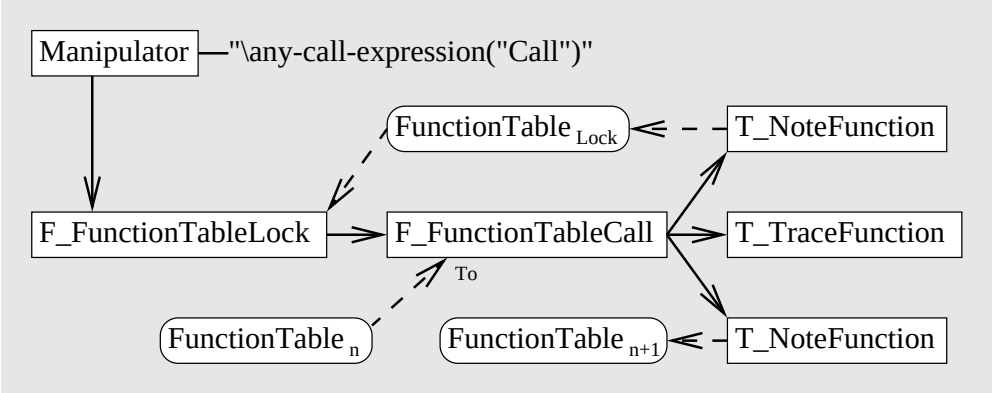


Abbildung 5.2.: Transformationsnetzwerk: Kontextwechsel

tion aufruft, die einen Kontextwechsel auslöst. Ist dies der Fall, so werden der Reihe nach drei Transformationen aktiviert (von oben nach unten). Die erste (`T_NoteFuntion`) vermerkt die Funktion als transformiert, um sie in Zukunft zu schützen. Die zweite (`T_TraceFunction`) fügt ein Sensorobjekt ein, um das Betreten und Verlassen der Funktion zu protokollieren. Die dritte Transformation speichert die Funktion in einer zweiten Tabelle, die dann beim nächsten Durchlauf der Transformationen durch den Filter `F_FunctionTableCall` verwendet wird. Durch diese Konstruktion können in aufeinanderfolgenden Transformationsläufen, bei denen jeweils alle Funktionen untersucht werden, rekursiv alle Aufrufwege zu `Coroutine::resume` gefunden werden.

Der dritte Teil des Transformationsprogramms, der als der Aspektweber bezeichnet werden kann, sorgt mit den Aufrufen von `SaveClassDB` und `CreateDBIndex` dafür, dass zum ersten die Strukturdatenbank gespeichert wird und zum zweiten, dass die Datenbank in-

monitorcontext.pl (3/3: Aspektweber)

```
24 SaveClassDB();
25 CreateDBIndex();

26 $calltable = FunctionTable("Coroutine::resume");
27 @func = Functions(".*::.*");
28 LOOP: {
29   $callfilter->To($calltable);
30   $runtable = FunctionTable();
31   $callnote->Table($runtable);
32   Run($task, @func);
33   last LOOP if $runtable->Empty();
34   $calltable = $runtable;
35   redo LOOP;
36 }

37 InsertRTSupport({function => "Global_pure::pure",
38                  size => 10000,
39                  target => "Native"});
40 InsertEvent({base => "context"}, "Coroutine::resume");
41 Commit();
42 Save();
43 SaveEvents("Events");
```

diziert wird, damit die eingebrachten Sensoren bei der Auswertung eindeutig identifiziert werden können. Als weiterer Schritt wird die initiale Tabelle der Aufrufziele (s.o.) belegt. Die Angabe von `Coroutine::resume` sorgt dafür, dass die Tabelle mit allen Funktionen dieses Namens, von denen es zwei gibt, gefüllt wird. Danach wird über einen Aufruf von `Functions()` ein Feld aller bekannten Funktionen erzeugt. Das übergebene Muster ist ein regulärer Ausdruck. Im Beispiel liefert der Aufruf alle bekannten Funktionen.

In den Zeilen 28 bis 36 ist die Schleife definiert, die für die rekursive Verfolgung der Aufrufe zu `Coroutine::resume` sorgt. Bei jedem Durchlauf wird dabei zuerst die Tabelle der Aufrufziele neu gesetzt, danach wird eine neue Tabelle erzeugt, in der die im aktuellen Durchlauf transformierten Funktionen für den nächsten Durchlauf gespeichert werden können. Durch den Aufruf von `Run` in Zeile 32 wird das Transformationsnetzwerk für jede im übergebenen Feld enthaltene Funktion durchlaufen. Danach wird überprüft, ob die Tabelle der in diesem Durchlauf transformierten Funktionen leer ist. Ist dies der Fall, so wird die Schleife beendet. Anderenfalls werden die Tabellen neu zugewiesen und der nächste Durchlauf erfolgt nach gleichem Muster.

In den Zeilen 37 bis 40 werden noch ein paar separate Transformationen vorgenommen. Zuerst wird durch den Aufruf von `InsertRTSupport` der Programmcode zur Initialisierung der Laufzeit-Messumgebung in die Funktion `pure` eingefügt, die sich in der Datei

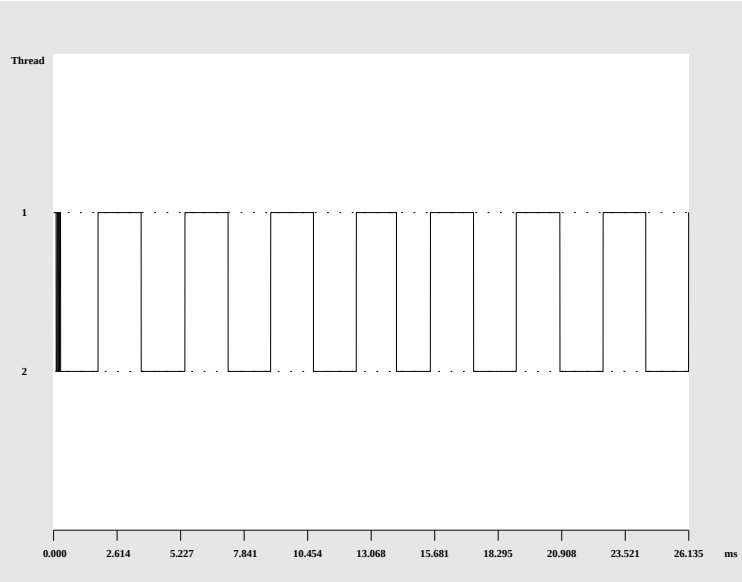


Abbildung 5.3.: Kontextwechsel (Übersicht)

`pure.cc` befindet. An dieser Stelle kann man sehen, wie C-Funktionen in verschiedenen Dateien mit C++-Klassen gleichgesetzt werden. Danach erfolgt in Zeile 39 das Einfügen eines speziellen Sensors in den zwei Varianten von `Coroutine::resume`. Der Sensortyp ist der gleiche wie bei allen vorherigen Transformationen, er wird aber mit einer speziellen Ereignisklasse (s. Anhang A.1) versehen, um ihn in der Auswertung erkennen zu können, ohne Informationen über das untersuchte System zu haben.

Damit sind alle Transformationen abgeschlossen. Durch den Aufruf von `Commit` werden sie tatsächlich durchgeführt, d.h. erst an dieser Stelle erfolgt die eigentliche Transformation der Syntaxbäume. Danach werden alle geänderten Dateien durch den Aufruf von `Save` gespeichert. Den Abschluss bildet der Aufruf von `SaveEvents`, der eine Liste der verwendeten Ereignisklassen und der Identifikationen der Elemente des Systems (Attribute, Methoden) enthält. Bei der Auswertung wird diese Datei benötigt, um die in den Ereigniscodes der Sensoren kodierten Identifikationen wieder durch Namen ersetzen zu können.

Nach der Abarbeitung dieses Programms ist das System geeignet instrumentiert. Es muss danach übersetzt werden und kann dann ausgeführt werden, um die Messung durchzuführen.

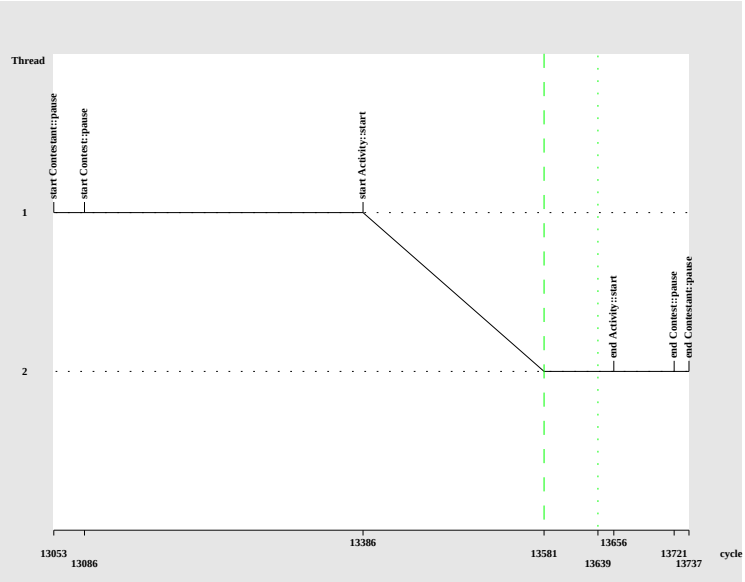


Abbildung 5.4.: Kontextwechsel (Ausschnitt)

5.3.3. Ergebnisse

Abbildung 5.3 zeigt das Diagramm des Kontrollflusses im Überblick über alle Ereignisse des Messlaufes. In dieser Darstellung ist eingetragen, welcher Thread gerade die Kontrolle über den Prozessor hat (durchgezogene Linie) sowie die seit dem ersten Ereignis verstrichene Zeit. An dieser Darstellung ist nur zu sehen, dass der Kontextwechsel nach der Initialisierung sehr regelmäßig erfolgt.

Die interessanten Details der Messung offenbaren sich bei einer Vergrößerung eines Ausschnitts wie in Abbildung 5.4 dargestellt. Im Ausschnitt ist ein Abschnitt zu sehen, in dem durch den Aufruf von `pause()` im Beispielprogramm ein Kontextwechsel ausgelöst wurde. Ebenfalls eingetragen sind die Zeiten der Ein- und Austrittspunkte der Methoden, die auf dem Weg zum Kontextwechsel durchlaufen wurden. Die Zeiten am unteren Rand geben die jeweilige Zeit in Taktzyklen an. Die angegebenen Werte wurden schon um den Zeitverbrauch der Sensoren bereinigt. Der eigentliche Kontextwechsel erfolgt zwischen den senkrechten Linien. Die gestrichelte kennzeichnet den Beginn und die gepunktete das Ende des Wechsels. Es ist zu sehen, dass der eigentliche Kontextwechsel gerade einmal 58 Taktzyklen dauert.

Das Ergebnis von 58 Takzyklen für einen Kontextwechsel gilt für einen Zeitpunkt, an dem bereits Cache-Effekte wirken. In Abbildung 5.5 ist die Entwicklung der Messzeiten in den zehn Messdurchgängen abgebildet.

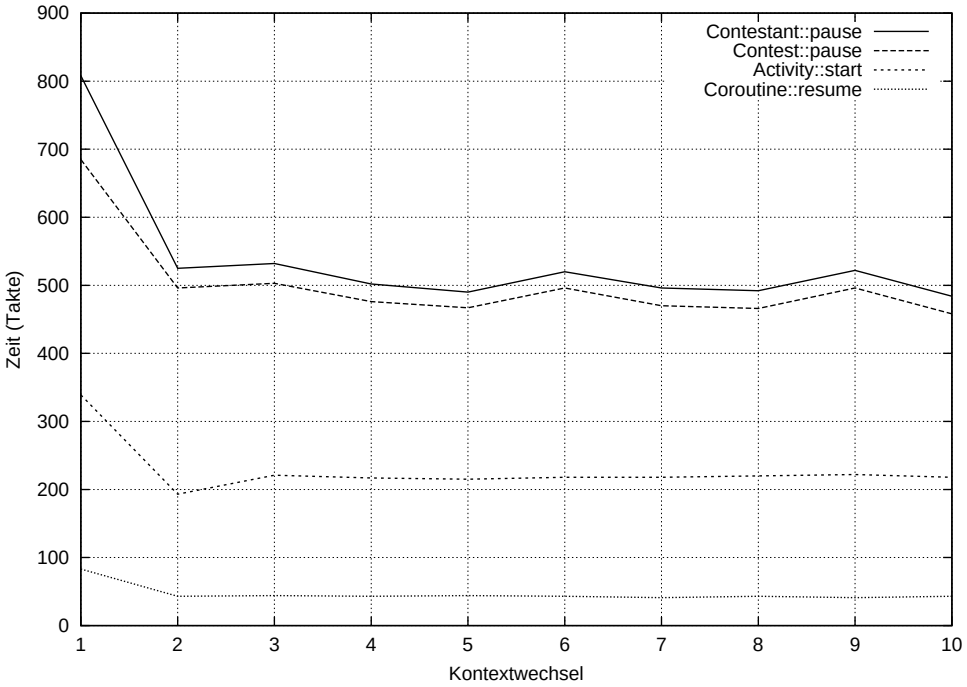


Abbildung 5.5.: Cache-Effekte beim Kontextwechsel

5.4. Verweilzeit im Nukleus

Bei der zweiten Anwendung des Monitorings geht es um die Protokollierung eines wesentlichen Systemzustandes, nämlich des Zustandes der globalen Lock-Variable, die den Nukleus schützt. Betritt ein Faden den Nukleus, so wird diese Variable gesetzt, wodurch er gesperrt wird. Durch diese Maßnahme wird der Zugriff auf den Nukleus synchronisiert. Für die Messung wird das PURE- Beispielprogramm der fünf speisenden Philosophen (`philo.cc`) verwendet. An dieser Stelle soll auch einmal die Verwendung einer sehr einfachen Komponentenbeschreibung demonstriert werden. Die Messung erfolgt auf der Linux-Gastebene.

5.4.1. Vorgehensweise

Zur Überwachung des Status des Nukleus-Locks ist es notwendig, jede Änderung seines Zustandes zu protokollieren. Der Zutritt zum Nukleus ist nur über eine Schleuse (`Sluice`) möglich, die auch die eigentliche Lock-Variable kapselt. Diese Kapselung erleichtet die Aufgabe der Zustandsüberwachung sehr, da der Zustand nur über die Methoden `enter` und `leave` der Schleuse geändert wird. Folglich müssen auch nur diese

monitornukleus.pl (1/4: Projektdefinition)

```
1 Project({source => "pure/usr/src/sys",
2          include => "pure/usr/include",
3          application => "pure/usr/tst/approved/philo.cc",
4          protect => [".*GM.*", ".*.h\$","^/usr/.*"],
5          definitions => "all",
6          files => ["pure/usr/src/sys/pure.cc"],
7         });
```

beiden Methoden überwacht werden. Daneben interessiert auch, wodurch der Zugang zum Nukleus erfolgt. Deshalb werden auch alle Methoden, die die Schleuse betreten aufgezeichnet.

Dazu werden als erstes die zwei Komponenten `World` und `Sluice` definiert. Danach werden alle Aufrufe gesucht, die von der ersten in die zweite Komponente führen.

5.4.2. Transformationsprogramm

In diesem Anwendungsfall gliedert sich das Transformationsprogramm in vier Teile, die Projektdefinition, die Komponentendefinition, das Aspektprogramm und den Aspektweber. Alle unterstützenden Programmteile werden in den Quelltextausschnitten nicht berücksichtigt, da sie bereits im vorigen Beispiel erläutert wurden.

Die Projektdefinition legt, wie im vorigen Anwendungsfall ausführlich beschrieben, die Rahmenparameter des bearbeiteten Projekts fest. In diesem Fall wird die Variante gezeigt, bei der man nicht genau weiss, welche Klassen transformiert werden. Deshalb wurde über den Parameter `definitions` mit dem Wert `"all"` angegeben, dass alle nach dem Parsen der Applikation definierten Klassen geladen werden sollen. Zusätzlich wird die Datei `pure.cc` geladen, in die später die Laufzeitunterstützung eingefügt wird.

Als zweiter, neuer Teil folgt die Definition der benötigten Komponenten. Da die Schleuse in den beiden Varianten `Guard` und `VoidGuard` auftreten kann, von denen immer nur eine verwendet wird, wird zuerst ein Alias definiert. Danach wird über den Alias die verwendete Klasse der Komponente `Sluice` zugeordnet. Der Parameter `base` gibt an, dass die Definition auch über eine Ebene auf die Basisklassen übertragen werden soll, die die Lock-Variable kapseln. Zuletzt werden über einen Aufruf von `AutoAssign` alle bisher nicht zugewiesenen Klassen der Komponente `World` zugeordnet. `AutoAssign` ist die allgemeine Schnittstelle zu den unter 3.3.2 besprochenen Strategienen zur automatischen Vervollständigung von Komponentenbeschreibungen, die einzeln oder zusammen verwendet werden können.

Den dritten Teil bildet das Aspektprogramm, das das Transformationsnetzwerk definiert. Auch hier lässt sich die Funktion besser an der Darstellung des Transformationsnetz-

monitornukleus.pl (2/4: Komponentendefinition)

```
9  ClassAlias({alias => "Sluice"},"Guard","VoidGuard");
10 AssignComponent({component => "Sluice", base => 1},"Sluice");
11 AutoAssign({default => "World"});
```

monitornukleus.pl (3/4: Aspektprogramm)

```
13 $lock = Manipulator
14   ({patternname => "AnyCall",
15     expression => "\any-call-expression("Call")"},
16    F_FunctionTableLock
17    ({table => $locktable = FunctionTable()},
18     F_ComponentTableCall
19     ({node => "Call",
20       from => ComponentTable("World"),
21       to => ComponentTable("Sluice")
22      },
23      T_NoteFunction({table => $locktable}),
24      T_TraceFunction()
25     )
26    )
27   );
```

werks erklären, das in Abbildung 5.6 zu sehen ist. Die Funktion entspricht annähernd der des Netzwerkes aus dem vorangehenden Anwendungsfall, weshalb an dieser Stelle nur das neue Element, der Filter `F_ComponentTableCall`, erklärt wird. Dieser Filter analysiert einen Funktionsaufruf und prüft danach, ob die Klassen, in denen Aufrufquelle und -ziel liegen, jeweils zu einer Menge von Komponenten gehören. Hierdurch werden alle Aufrufe der Komponente `World` zur Komponente `Sluice` ermittelt. Ist dies der Fall, so werden die verbundenen Transformationen aktiviert, die dafür sorgen, dass jede ermittelte Funktion mit genau einem Sensor versehen wird.

Der Aspektweber arbeitet ähnlich dem des vorigen Beispiels mit den Ausnahmen, dass diesmal ein System für eine UNIX-Gastebene vorbereitet wird und das Transformationsnetzwerk nur einmal durchlaufen wird. Als zusätzliche Ereignisse werden neben der Protokollierung der Timer-Interrupts die Methoden `enter` und `leave` der möglichen Schleusenvarianten mit Sensoren für spezielle Ereignisklassen bestückt, um in der Auswertung den Systemzustand gesondert anzeigen zu können, ohne dabei Wissen über die Details der Instrumentierung zu benötigen.

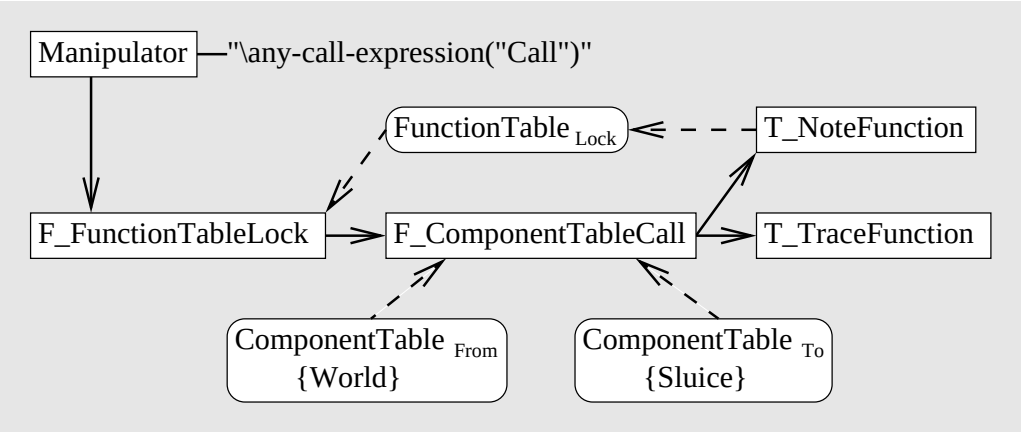


Abbildung 5.6.: Transformationsnetzwerk: Nukleusüberwachung

monitornukleus.pl (4/4: Aspektweber)

```
29 Run($lock, Functions(".*::.*"));
30 InsertRTSupport({function => "Global_pure::pure",
31                  size => 10000,
32                  target => "UNIX",
33                  file => "philo.trc"});
34 InsertTimerEvent("Timeslice::serve");
35 InsertEvent({user => "enter", out => 1}, "Sluice::enter");
36 InsertEvent({user => "leave", in => 1}, "Sluice::leave");
```

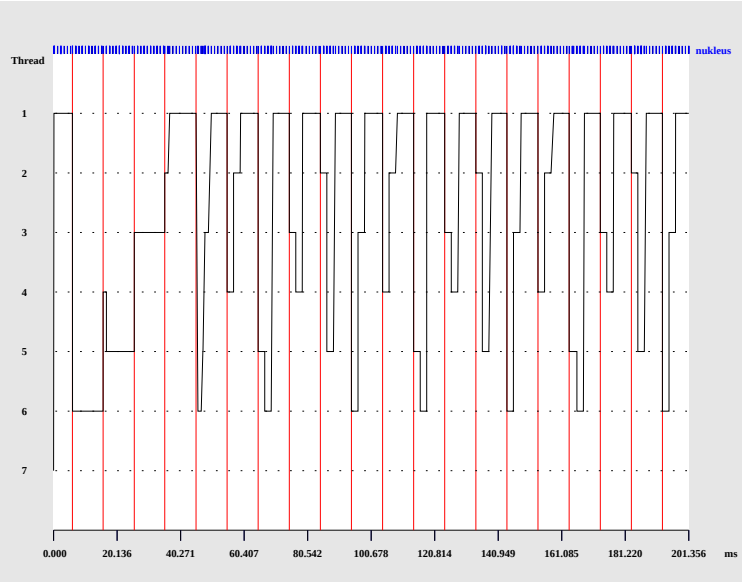


Abbildung 5.7.: Verweilzeit im Nukleus (Übersicht)

5.4.3. Ergebnisse

Abbildung 5.7 stellt das Diagramm der Messung im Überblick dar. Am oberen Rand ist die Verweilzeit im Nukleus verzeichnet, d.h. immer dann, wenn der Nukleus-Lock gesetzt ist, wird der Zeitbereich durch einen Balken gekennzeichnet. Die durchgezogenen, senkrechten Linien kennzeichnen die Timer-Ereignisse, die einen Kontextwechsel auslösen können. Die Vergrößerung eines kleinen Ausschnittes ist in Abbildung 5.8 zu sehen. Dort sind auch die Details der Methodenaufrufe angetragen. Der Ausschnitt zeigt einen Kontextwechel, der durch den Zeitgeber ausgelöst wurde und die nachfolgende Verwendung einer Semaphore.

Aus der Analyse des Ereignisprotokolls lassen sich noch weitere Informationen gewinnen. Da alle Sensoren aus den Ereignissen eindeutig identifiziert werden können, können so auch die paarweise sammengehörenden sowie die Zeitdifferenz zwischen ihnen bestimmt werden. So lässt sich auch der längste und kürzeste Aufenthalt im Nukleus ermitteln. Im Beispiel liegt die kürzeste Verweilzeit unterhalb der Messgenauigkeit von 1 μs. Die längste Verweilzeit betrug 13 μs bei einem zeitgesteuerten Kontextwechsel.

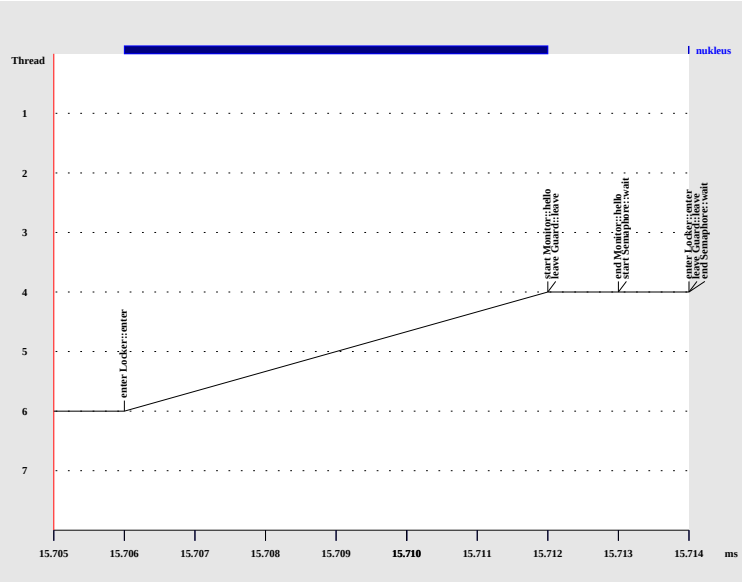


Abbildung 5.8.: Verweilzeit im Nukleus (Ausschnitt)

5.5. Scheduling-Aufwand

Der dritte Anwendungsfall demonstriert die Transformation des PURE-Systems zum Monitoring des Scheduling-Verhaltens. Dabei wird eine einfache Komponentenbeschreibung verwendet, die unabhängig von der Wahl des Schedulers ist, der in vielen verschiedenen Varianten auftreten kann.

5.5.1. Vorgehensweise

Zum Monitoring des Scheduling-Verhaltens werden alle Aufrufe von Funktionen des Schedulers protokolliert. Im Gegensatz zu den vorangegangenen Beispielen wird hier die „Außenseite“ eines Funktionsaufrufes transformiert, um zu sehen, von wo der Scheduler aufgerufen wird, die aufgerufenen Funktionen sind in diesem Zusammenhang uninteressant. Zusätzlich werden alle Aufrufe des Schedulers selbst, die aus ihm hinaus führen aufgezeichnet. Als Testprogramm wir das Testprogramm aus Abschnitt 5.3 verwendet. Als kleine Abwandlung werden nicht nur ein sondern zehn neue Programmfäden erzeugt, die jeweils zehnmal die Kontrolle freiwillig abgeben und sich danach selbst zerstören.

monitorscheduler.pl (1/4: Projektdefinition)

```
1  Project({source => "pure/usr/src/sys",
2           include => "pure/usr/include",
3           application => "pure/usr/tst/approved/bundle4.cc",
4           protect => [".*GM.*", ".*.h\$","^/usr/.*"],
5           definitions => "all",
6           files => ["pure/usr/src/sys/*.cc"]
7          });
```

monitorscheduler.pl (2/4: Komponentendefinition)

```
9  ClassAlias({alias => "Schemer"}, qw(Confidant Contestant Rival Competitor
10                                     Wizard Passenger Process));
11 ClassAlias({alias => "Scheme"}, qw(Confidant Contest Rivalry Competition
12                                    Passage));
13 AssignComponent({component => "Scheduler"}, "Schemer", "Scheme");
14 AutoAssign({default => "World"});
```

5.5.2. Transformationsprogramm

Auch in diesem Fall gliedert sich das Transformationsprogramm in vier Abschnitte, die Projektdefinition, die Komponentendefinition, das Aspektprogramm und den Aspektweber.

Die Projektdefinition entspricht bis auf die gewählte Applikation denen der vorangegangenen Beispiele.

Die Komponentenbeschreibung definiert die Komponenten `Scheduler` und `World`. Die Komponente `Scheduler` umfasst genau zwei Klassen, die deshalb auch explizit angegeben werden können. Das besondere in diesem Fall ist die Tatsache, dass es viele verschiedene Scheduler gibt, die alle die selbe Schnittstelle anbieten und von denen immer nur einer gleichzeitig aktiv ist. Welcher dies ist, wird in der PURE-Konfiguration festgelegt. Um die Komponentenbeschreibung von der Systemkonfiguration unabhängig zu machen, werden alle Varianten der Modellklassen `Schemer` und `Scheme` in den Zeilen 9 bis 12 über eine Alias-Definition den Modellklassen zugeordnet. Die eigentliche Beschreibung der Komponente erfolgt dann in Zeile 13 anhand der Aliasnamen. Alle anderen Klassen werden in Zeile 14 der Komponente `World` zugeordnet.

Wie gewohnt, lässt sich das Aspektprogramm wieder besser am erzeugten Transformationsnetzwerk erläutern, das in Abbildung 5.9 zu sehen ist. In diesem Fall werden zuerst wieder alle Funktionsaufrufe aus allen bekannten Funktionen herausgesucht. Diese werden dann nacheinander durch die beiden `F_ComponentTableCall`-Filter auf ihre Aufrufziele hin untersucht. Zusätzlich werden Aufrufquell und -ziel nach den Angaben der beiden Tabellen gefiltert. Dadurch werden alle Aufrufe zwischen den beiden Komponen-

monitorscheduler.pl (3/4: Aspektprogramm)

```
16 $sched = Manipulator
17   ({patternname => "AnyCall",
18     expression => "\any-call-expression("Call")"},
19    F_ComponentTableCall
20    ({node => "Call",
21      from => $world = ComponentTable("World"),
22      to => $scheduler = ComponentTable("Scheduler")},
23     T_TraceExpression({node => "Call", in => "start", out => "end"})
24    ),
25    F_ComponentTableCall
26    ({node => "Call",
27      to => $world = ComponentTable("World"),
28      from => $scheduler = ComponentTable("Scheduler")},
29     T_TraceExpression({node => "Call", in => "escape", out => "return"})
30    )
31   );
```

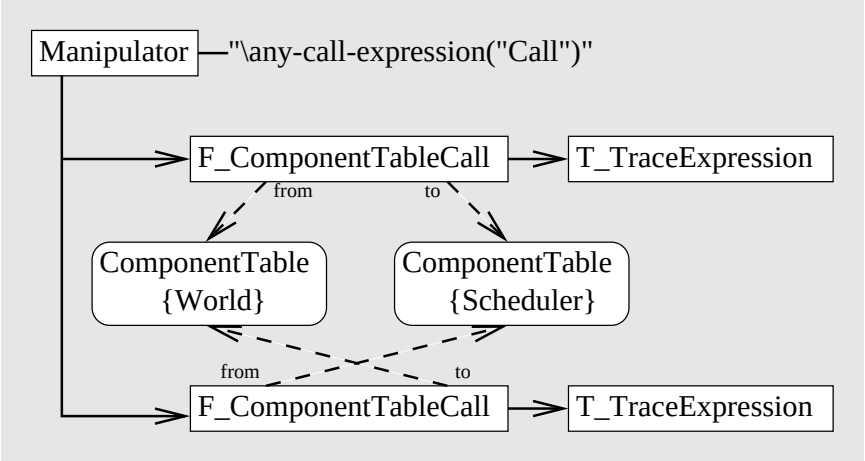


Abbildung 5.9.: Transformationsnetzwerk: Scheduler-Monitoring

monitorscheduler.pl (4/4: Aspektweber)

```
33 Run($sched, Functions(".*::.*"));
34 InsertRTSupport({function => "Global_pure::pure",
35                  size => 10000,
36                  target => "UNIX",
37                  file => "sched.trc"
38                 });
```

ten herausgesucht und durch die Transformation `T_TraceExpression` um die Sensoren zur Protokollierung der Aufrufe erweitert, d.h. Aufrufe innerhalb der Komponenten bleiben unverändert.

Der Aspektweber ist in diesem Beispiel sehr einfach. Er ruft das Transformationsnetzwerk einmal für alle Funktionen auf und fügt danach den Programmcode zur Initialisierung der Laufzeiterweiterung in die Funktion `pure` der Datei `pure.cc` ein.

5.5.3. Ergebnisse

Die Abbildungen 5.10, 5.11 und 5.12 zeigen die grafische Darstellung der Ereignisprotokolle im Überblick. Da es in diesem Anwendungsfall nicht um die Messung des detailierten Zeitverbrauchs des Schedulers und der verwendeten Algorithmen sondern um die Darstellung des Scheduling-Verhaltens, d.h. der Art wie die einzelnen Programmfäden im Verlaufe des Programms abgearbeitet werden, geht, wird auf Ausschnittsvergrößerungen verzichtet.

Aber auch aus den Übersichten sind einige Erkenntnisse zu gewinnen. So ist zu sehen, dass bei beiden FCFS[1]-Schedulern die Abfolge der Fäden gleich ist. Im Gegensatz dazu werden beim LCFS[2]-Scheduling alle Fäden der Reihe nach abgearbeitet. Das die Darstellungen der Flanken der Fadenwechsel so unterschiedlich aussehen liegt daran, dass beim FCFS-Thread-Scheduler die protokollierten Aufrufe jeweils näher am eigentlichen Kontextwechsel liegen, wodurch die Flanken wesentlich steiler erscheinen.

Bei den Zeiten ist zu erkennen, dass beide Bundle-Scheduler ca. den gleichen Laufzeitbedarf sowohl für die Erzeugung der Kind-Fäden (im vorderen Bereich) als auch für das Scheduling insgesamt haben. Der Thread-Scheduler ist in beiden Teilen etwas schneller.

[1] First Come First Serve
[2] Last Come First Serve

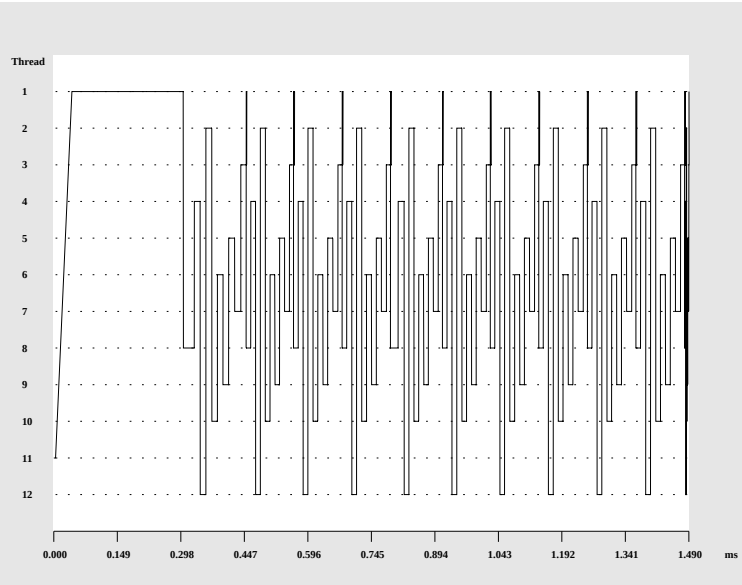


Abbildung 5.10.: Scheduler-Monitoring: FCFS-Thread-Scheduling

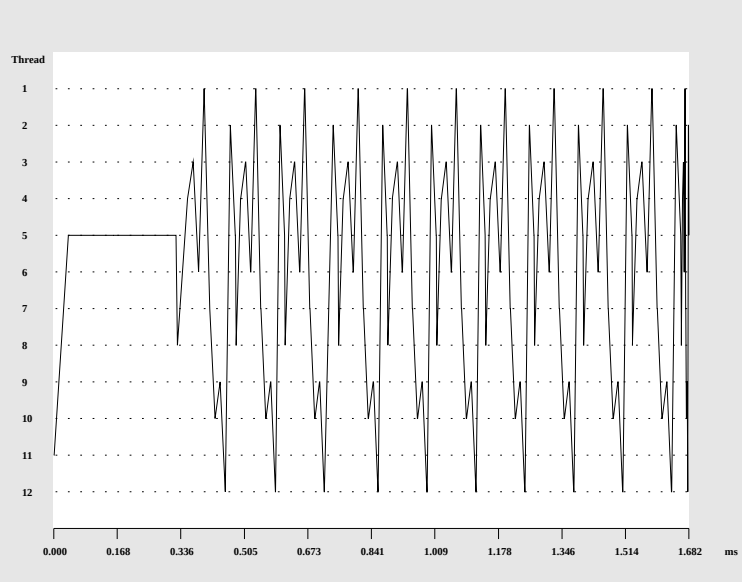


Abbildung 5.11.: Scheduler-Monitoring: FCFS-Bundle-Scheduling

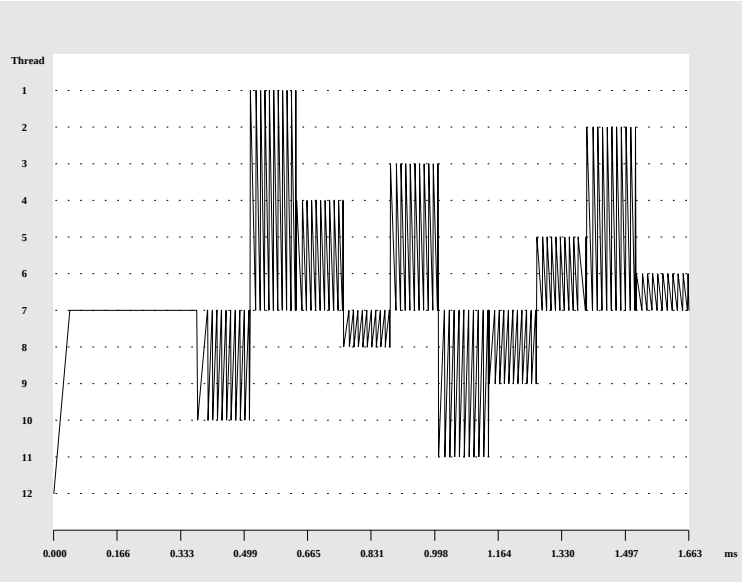


Abbildung 5.12.: Scheduler-Monitoring: LCFS-Bundle-Scheduling

5.6. Überprüfung von Zeigerwerten

Das vierte Anwendungsbeispiel soll demonstrieren, dass der hier dargestellt Ansatz zum Monitoring von Systemen nicht nur zum Messen von Systemeigenschaften verwendet werden kann, sondern auch andere praktische Anwendungen z.B. zur Unterstützung der Fehlersuche hat. Dazu soll im folgenden Beispiel gezeigt werden, mit welchen Mitteln ein System so transformiert werden kann, dass jeder Zeiger vor einer Dereferenzierung geprüft wird. Dabei soll der Einfachheit halber nur die Verschiedenheit von null getestet werden. In einer praktischen Anwendung kann natürlich auch eine beliebige andere Testbedingung eingefügt werden. Zusätzlich soll angenommen werden, dass ein Fehler in der Implementierung der Speicherverwaltung vermutet wird und deshalb nur dort das System transformiert werden soll. In diesem Zusammenhang soll auch die Verwendung des universellen Bausteins zur Transformation von Syntaxbäumen und der Einsatz dynamischer Muster demonstriert werden.

5.6.1. Vorgehensweise

Zur Überprüfung aller Dereferenzierungen eines Zeigers werden alle Stellen in der Komponente `Memory` gesucht, an denen ein Zeiger dereferenziert werden soll. Diese Ausdrücke werden durch einen zusammengesetzten Ausdruck ersetzt, der zuerst den Wert

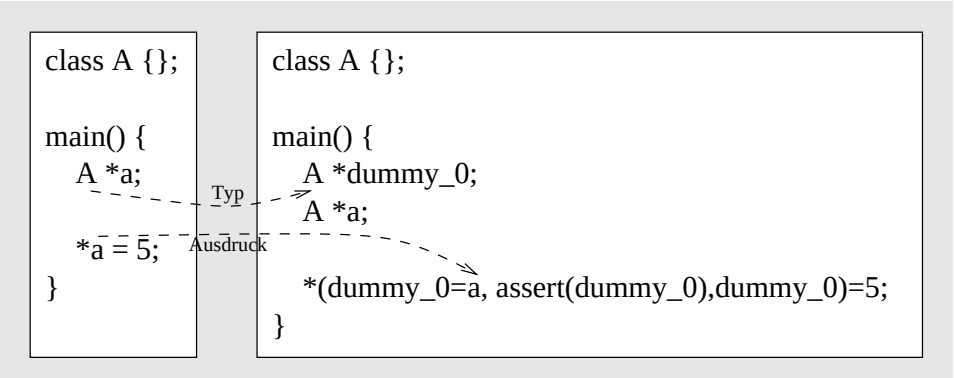


Abbildung 5.13.: Assert-Ersetzung: Beispiel

assertptr.pl (1/4: Projektdefinition)

```
1 Project({source => "pure/usr/src/sys",
2          include => "pure/usr/include",
3          application => "pure/usr/tst/approved/philo.cc",
4          protect => [".*GM.*", ".*.h\$","^/usr/.*"],
5          definitions => "all",
6          files => ["pure/usr/src/sys/device/memory/*.cc"]
7         });
```

des Basisausdrucks in einer temporären Variable speichert, danach den Wert dieser Variablen prüft und ihn zuletzt zurückliefert. Daneben wird die untersuchte Funktion um die Deklaration einer temporären Variable erweitert, deren Typ dem des Basisausdrucks entspricht. Dadurch werden die Seiteneffekte der Ermittlung und Dereferenzierung nicht verändert, da der Basisausdruck nur einmal ermittelt wird und zur Dereferenzierung auch der richtige, möglicherweise überladene, Stern-Operator verwendet wird. Die geplante Ersetzung ist in Abbildung 5.13 an einem einfachen Beispiel dargestellt.

5.6.2. Transformationsprogramm

Das Transformationsprogramm gliedert sich wieder in vier Abschnitte, die Projektdefinition, die Komponentendefinition, das Aspektprogramm und den Aspektweber.

Abbildung 5.14 stellt den Ausschnitt aus der PURE-Klassenhierarchie dar, der die Komponente `Memory` darstellt. Obwohl diese Komponente deutlich umfangreicher als die bisherigen ist, bleibt der Aufwand zu ihrer Beschreibung gleich. Als erstes wird wieder ein Alias für die Varianten von `Allocation` definiert. Dies ist hier zwar nicht zwingend notwendig, da nur eine Variante existiert. Das Modell lässt aber vermuten, dass an

assertptr.pl (2/4: Komponentenbeschreibung)

```
9  ClassAlias({alias => Allocation}, "NextFit");
10 AssignComponent({component => "Memory", base => 2}, "HeapManager");
11 AutoAssign({inheritance => 1, usage => 1, default => "World"});
```

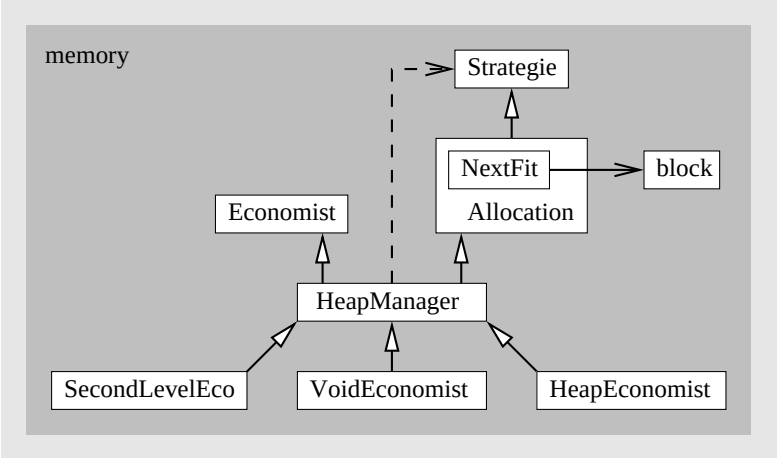


Abbildung 5.14.: Pure-Klassen: Speicherverwaltung

dieser Stelle Erweiterungen geplant sind, somit ist es auch nicht verkehrt. Danach wird die Klasse `HeapManager` zusammen mit ihren Basisklassen (über zwei Stufen) der Komponente `Memory` zugeordnet, da sie nach dem Klassendiagramm sehr zentral liegt. Der Rest wird durch die automatische Vervollständigung erledigt. Dazu werden die Komponentenzuordnung über Vererbungs- (`inheritance`) und Nutzungsbeziehungen (`usage`) weitergegeben. Alle anderen Klassen werden der Komponente `World` zugeordnet.

Auch in diesem Fall lässt sich die Arbeit des Aspektprogramms am besten am Bild des Transformationsnetzwerkes (s. Abbildung 5.15) erläutern. Zuerst wird diesmal nach Ausdrücken mit dem Muster `*<Ausdruck>` gesucht. Ein so gefundener Ausdruck wird als erstes durch den Filter `F_ComponentMember` daraufhin untersucht, ob er in einer Funktion der Komponente `Memory` liegt. Danach wird der Ausdruck analysiert und der Type des inneren Ausdrucks als String in einem Namensspeicher (`S_Name`) abgelegt. Angeschlossen sind dann drei Transformationen. Die erste baut nach einem vorgegebenen Muster einen Namen für eine temporäre Variable zusammen, der dann in einem zweiten Namensspeicher abgelegt wird. Dabei wird auf einen speziellen Nummernspeicher (`S_DummyNum`) zurückgegriffen, der eine einmalige Nummer liefert, um sicherzustellen, dass nicht zwei temporäre Variable mit dem selben Namen definiert werden. Die zweite Transformation (`T_ReplaceTree`) ersetzt Teile des Syntaxbaumes und greift dabei auf die temporäre

assertptr.pl (3/4: Aspektprogramm)

```
13 $assert = Manipulator
14   ({patternname => "Deref",
15     expression => "*\any-expression("Expr")"},
16    F_GetTypeName
17    ({store => $type = S_Name(), node => "Expr"},
18     T_Storepattern({pattern => ["dummy_", S_DummyNum()],
19                     store => $name = S_Name()}),
20     T_ReplaceTree
21     ({expression => ["*(",$name,"=\any-expression("Expr"),",
22                      "assert(",$name,"),",$name,")"]}),
23     T_InsertDummyVar({pattern => [$type, " *", $name, ";\n"]})
24    )
25   );
```

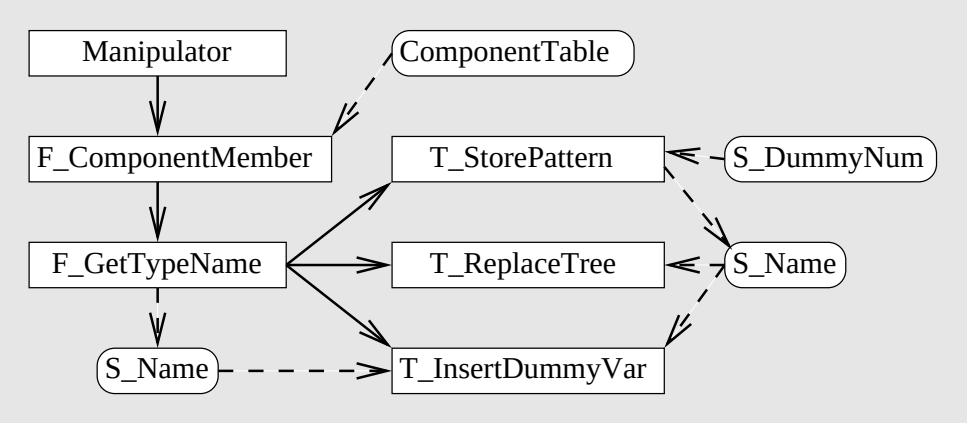


Abbildung 5.15.: Transformationsnetzwerk: Assert-Ersetzung

assertptr.pl (4/4: Aspektweber)

```
27 Run($assert, Functions(".*::.*"));
```

Variable zurück, deren Namen zuvor erzeugt wurde. Durch die letzte Transformation wird dann die temporäre Variable selbst am Beginn der Funktion eingefügt.

Der Aspektweber ist in diesem Fall sehr einfach, er durchläuft einfach das Transformationsnetzwerk für jede Funktion und speichert danach die geänderten Dateien.

Zwei Punkte sind an dieser Stelle noch genauer zu betrachten, zum ersten die dynamischen Muster und zum zweiten die Ersetzung von Teilen des Sytaxbaumes.

Dynamische Muster werden speziell bei der Ersetzung von Syntaxbäumen benötigt, da hier oftmals variable Namen oder Typen aus dem zu ersetzenden übernommen werden müssen. Sie bestehen dabei aus einem Feld von Strings und Referenzen auf einfache Speicherbausteine. Ein dynamisches Muster wird dann vor jeder Verwendung neu ausgewerte und das Ergebnis z.B. als Eingabe für den Parser benutzt.

Da die Syntaxbäume selbst für das einfache Beispiel aus Abbildung 5.13 schon sehr komplex sind, sollen die Vorgänge bei der Ersetzung von Teilen der Syntaxbäume an einem Modell verdeutlicht werden. Am Transformationsprozess sind insgesamt fünf Bäume beteiligt, die in Abbildung 5.16 dargestellt sind. Ausgangspunkt ist der Syntaxbaum in dem ein bestimmtes Baummuster gesucht wird. Dieser Suchbaum ist ein normaler Syntaxbaum, der jedoch über spezielle Syntaxknoten verfügen kann, die angeben, das an ihrer Stelle beliebige Knoten bzw. Unterbäume im Quellbaum stehen können. Wird bei der Suche ein passender Ausschnitt im Quellbaum gefunden, so wird der Ausschnitt durch einen neuen Teilbaum ersetzt. Dieser entsteht aus einem Zielbaum, der ebenfalls über variable Knoten verfügen kann dadurch, dass alle variablen Knoten durch die variablen Knoten des Quellbaums ersetzt werden. Die Zuordnung wird dabei über Namen realisiert, die den variablen Knoten zugeordnet werden. Nach der Vervollständigung dieses Zielbaumes wird der gesamte Teilbaum durch ihn ersetzt.

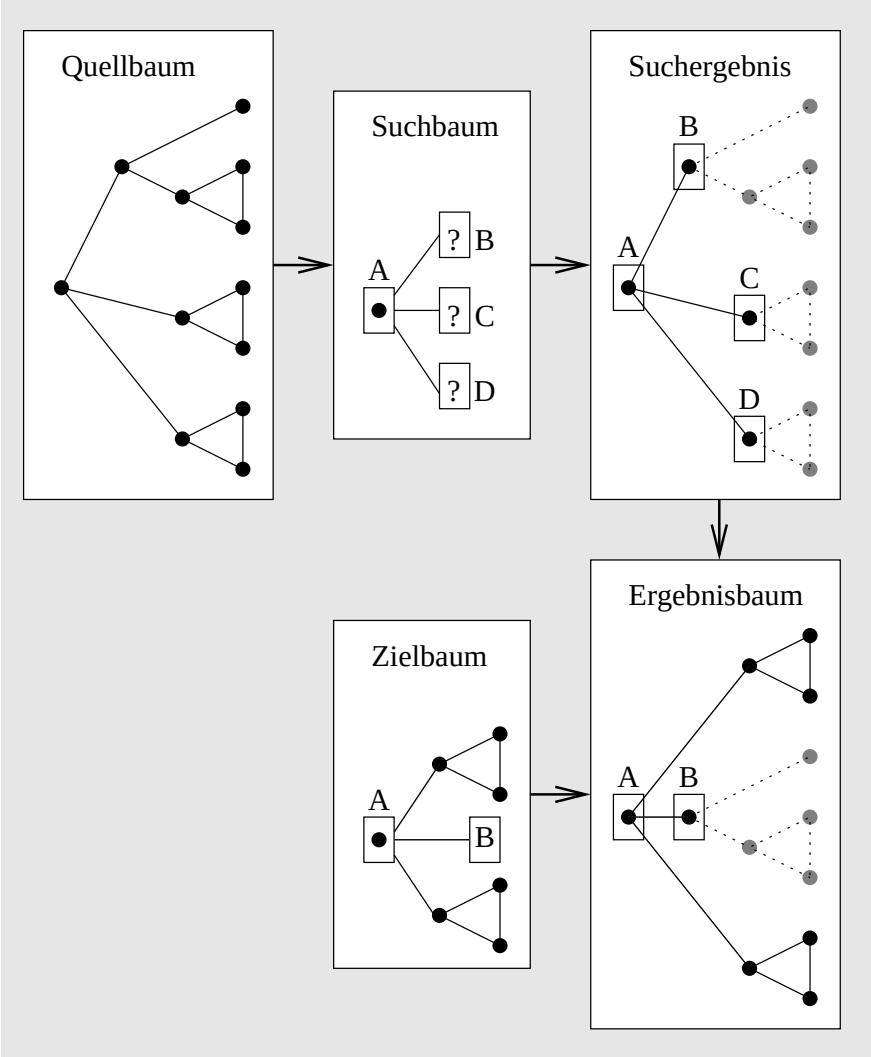


Abbildung 5.16.: Ersetzungen in Syntaxbäumen

6. Ausblick

Im Rahmen dieser Arbeit wurde gezeigt, dass das Monitoring eines Systems drei Bereiche umfasst. Zum ersten basiert das Monitoring darauf, alle Zustandsänderungen und den Kontrollfluss eines Programms analysieren zu können. Zum zweiten musst der Anwender die Möglichkeit bekommen auszudrücken, welche Zustandsänderungen aus seiner Sicht interessant sind. Und zum dritten muss der Anwender auf interessante Zustandsänderungen reagieren können. Dabei hat sich gezeigt, dass das Messen von Systemeigenschaften nur eine mögliche Reaktion auf interessante Zustandsänderungen ist.

Durch die Anwendung der Techniken der aspektorientierten Programmierung wurde es möglich, Systemeigenschaften wie das Monitoring klar vom untersuchten Sytem getrennt zu modellieren. Diese saubere Trennung bildet auch die Grundlage für eine Anwendung zur Untersuchung beliebiger Softwaresysteme durch ein einzelnes Monitoring-System. Ein weiterer wichtiger Baustein zur Verwirklichung der Unabhängigkeit vom untersuchten System und seinen möglichen Konfigurationen ist die Beschreibung eines Systems als eine Menge von konfigurationsunabhängigen Komponenten und deren Nutzung zur Festlegung und Steuerungen der Transformationen am System.

Ebenso wurde in der Anwendungsfallstudie nachgewiesen, dass Transformationsnetzwerke ein effektives Mittel zur Modellierung und Implementierung von Programmaspekten darstellen. Durch die klare Trennung der Bausteine eines Transformationsnetzwerkes in funktionale Gruppen zur Mustersuche in Syntaxbäumen, zur Analyse von Systemeigenschaften und zur Transformation von Syntaxbäumen werden alle Aufgabenbereiche des Monitorings durch jeweils eine Gruppe ausgefüllt. In den Anwendungsbeispielen wurde gezeigt, dass es mit der realisierten Lösung problemlos möglich ist, verschiedenste Messaufgaben umzusetzen. Daneben wurde auch gezeigt, dass das bisher realisierte System auch in ganz anderen Einsatzfeldern z.B. zur Unterstützung der Fehlersuche Anwendung finden kann.

Auf der Basis des bisherigen Ansatzes, der davon ausgeht, Muster in Syntaxbäumen zu finden, diese auf ihrer Eigenschaften hin zu untersuchen, um sie bei Bedarf transformieren zu können, lassen sich noch sehr vielfältige andere Anwendungen entwickeln, da sich jeder Teil eines Systems in bestimmten Mustern des Syntaxbaumes wiederspiegelt. So ist es möglich, aus einer Anwendung, die für den Einsatz auf einem Rechner gedacht ist, eine

verteilte Anwendung dadurch zu machen, dass man die Kommunikation zwischen Objekten als einen technischen Aspekt ansieht, der auch durch Fernaufrufe realisiert werden kann, und das System entsprechend transformiert. Des weiteren ermöglichen die umfangreichen Analysefunktionen auch die Ermittlung der in einem Programm überhaupt benutzbaren Teile. So könnte eine umfangreiche Programmbibliothek durch die Vorgabe einer konkreten Anwendung auf das notwendige Minimum reduziert werden, was speziell in einem Bereich akuter Ressourcenknappheit wie den eingebetteten Systemen, von außerordentlichem Vorteil wäre.

Das bisherige System konzentriert sich auf die Analyse und Transformation von Zustandsänderungen und des Kontrollflusses eines Programmes. Da der Ansatz aber nicht auf diese Anwendung beschränkt ist, kann das System leicht um weitere Analyse- und Transformationsbausteine erweitert werden, mit denen z.B. Datenstrukturen oder sogar das Design eines Systems im Hinblick auf eine konkrete Anwendung optimiert werden können. Der Großteil der bisher realisierten Bausteine wurde auch in den Anwendungsbeispielen verwendet, es bleibt somit die Aufgabe, den Satz der verfügbaren Bausteine zu vervollständigen, um damit die häufigsten Anwendungsfelder abdecken zu können.

In der Anwendungsfallstudie hat sich auch gezeigt, dass die Aspektprogramme zur Definition von Transformationsnetzwerken eine effektive Nutzerschnittstelle darstellen aber für das ungeübte Auge auch wenig verständlich sind, wogegen ihre Funktion an der bildlichen Darstellung der Transformationsnetzwerke leicht ersichtlich ist. Da zwischen beiden Ausdrucksformen eine 1:1 Abbildung besteht, liegt es nahe, ein grafisches Modellierungswerkzeug zur Definition von Transformationsnetzwerken zu entwickeln. Ebenso wäre ein grafisches Werkzeug zur Definition von Komponenten auf Basis der funktionalen Gruppierungen für die meisten Anwendungsfälle sehr hilfreich. Die hierfür notwendigen Analysemechanismen für Klassenbeziehungen wurden im Rahmen dieser Arbeit bereits realisiert. Für die Entwicklung dieser visuellen Werkzeuge bleiben zwei offene Frage. Werden die Aspektprogramme bzw. der Aspektweber grafisch modelliert, so sollte dies auch für den Aspektweber möglich sein, der wie in den Anwendungsfällen gezeigt, unterschiedlich komplexe Funktionen erfüllt. Hierfür ist somit eine geeignete Notation zu finden. Zum zweiten ist bei der grafischen Komponentendefinition zu klären, wie die Informationen der Strukturdatenbank visualisiert werden können und wie die variablen Teile einer Programmfamilie in einer solchen Darstellung berücksichtigt werden können.

A. Ereignisprotokoll

A.1. Datenformat

Das Ereignisprotokoll besteht aus einem Datensatz mit Verwaltungsinformationen und einem linearen Feld von Ereignissen. Die detailierte Struktur ist im Folgenden zu sehen:

Verwaltungsinformationen

Datenfeld	Format	Größe	Bedeutung
magic	unsigned long	4 Byte	Kennung zur Identifikation des Datenformats
frequency	double	8 Byte	Impulsfrequenz des verwendeten Zeitgebers
cpus	unsigned long	4 Byte	Anzahl der Prozessoren im System
records	unsigned long	4 Byte	Anzahl der gespeicherten Ereignisse
Summe: 20 Byte			

Ereignis

Datenfeld	Format	Größe	Bedeutung
type	unsigned long	4 Byte	Ereignistyp
time	unsigned long long	8 Byte	Messzeitpunkt (Anzahl vergangener Zeitimpulse)
proc_id	unsigned long	4 Byte	ID des laufenden Prozesses
thread_id	unsigned long	4 Byte	ID des aktiven Threads
param	unsigned long	4 Byte	freier Parameter
Summe: 24 Byte			

A.2. Basisereignistypen

Das Ereignisprotokoll stellt eine abwärtskompatible Erweiterung des Jewel-Formats dar. Die wichtigen Basisereignisse sind:

Timer Ereignisse eines Zeitgebers

Context Switch Kontextwechsel

IRQ allgemeine asynchrone Unterbrechungen

User nutzerdefinierte Ereignisse

A.3. Ereignisuntertypen

Das Jewel-Protokoll sieht für den Ereignistyp ein Datenfeld von vier Byte vor, von denen für die Basistypen nur eins verwendet wird. Deshalb werden die restlichen drei Byte zur Kodierung weiterer Informationen verwendet. Ein Byte wird dabei für einen Ereignisuntertyp verwendet. Die letzten zwei Byte tragen die Identifikation des erzeugenden Sensors. Die bisher verwendeten Untertypen sind:

Start Betreten eines Programmblocks

End Verlassen eines Programmblocks

Enter Eintritt in eine Komponente

Leave Verlassen einer Komponente und Rückkehr zum Aufrufer

Escape Verlassen einer Komponente durch Aufruf einer anderen

Return Rückkehr in eine Komponente

B. Transformationsbausteine

Zum Bau von Transformationsnetzwerken stehen folgende Bausteine zur Verfügung:

Filter

Name	Funktion
F_MatchPattern	vergleicht den untersuchten Funktionsnamen mit einer Liste regulärer Ausdrücke
F_MatchTypes	vergleicht den Typ eines Ausdrucks mit einer Liste von Typnamen
F_MatchOperator	vergleicht den Operator eines binären Ausdrucks mit einer Liste von Operatoren
F_AssignOperator	prüft, ob ein binärer Ausdruck eine Zuweisung ist
F_LockClass	blockiert die mehrfache Manipulation einer Klasse
F_DefinedFunction	prüft, ob die untersuchte Funktion eine Definition (Funktionskörper) besitzt
F_FunctionTableCall	analysiert einen Aufruf und filtert ihn nach Quelle und Ziel, die als Tabellen gegeben sind
F_FunctionTableLock	blockiert eine Liste (Tabelle) von Funktionen
F_ComponentMember	prüft, ob die untersuchte Funktion zu einer Menge von Komponenten gehört
F_ComponentTableCall	analysiert einen Aufruf und filtert ihn nach Quell- und Zielkomponenten, die als Tabelle gegeben sind
F_Void	hat keine Filterwirkung (Verbindungsknoten)
F_GetTypeName	analysiert und speichert den Typ eines untersuchten Ausdrucks

Bis auf die letzten beiden Filter liegen alle Filter auch in einer negierten Form vor (Vorsatz `NF_`).

Zustandsspeicher

Name	Funktion
FunctionTable	Tabelle von `FunctionInfo`-Objekten
ComponentTable	Tabelle von `ComponentInfo`-Objekten
S_Name	Speicher für einen String (Kopie)
S_Unsigned	Speicher für einen vorzeichenlosen Integer-Wert
S_AstInfo	Speicher für einen Knoten des Syntaxbaums
S_Match	Speicher für einen Suchtreffer der Mustersuche
S_DummyNum	spezieller Nummerngenerator für temporäre Variablen

Nur Zustandsspeicher mit dem Vorsatz `S_` können innerhalb dynamischer Muster verwendet werden.

Transformationen

Name	Funktion
T_ConstTextAtBegin	statischen Text am Funktionsanfang einfügen
T_ConstTextAtEnd	statischen Text am Funktionsende einfügen
T_BaseClassAtBegin	neue Basisklasse am Beginn der Vererbungsliste einfügen
T_BaseClassAtEnd	neue Basisklasse am Ende der Vererbungsliste einfügen
T_NoteFunction	untersuchte Funktion in einer Tabelle speichern
T_TraceFunction	Sensor für Ein- und Austritt in Funktion einfügen
T_TraceFunctionIn	Sensor für Eintritt in Funktion einfügen
T_TraceFunctionOut	Sensor für Austritt in Funktion einfügen
T_ReplaceTree	universelle Ersetzung von Syntaxbaumteilen
T_StorePattern	dynamisches Muster auswerten und in einem Namensspeicher ablegen
T_InsertDummyVar	temporäre Variable in Funktion einfügen
T_StoreAstInfo	speichern eines Syntaxknotens eines Suchtreffers in einen Speicher
T_StoreMatch	speichern eines Suchtreffers in einen Speicher
T_AddMatch	hinzufügen eines gespeicherten Suchtreffers zum aktuellen
T_TraceExpression	Sensor vor und nach einem Ausdruck einfügen
T_TraceExpressionIn	Sensor vor einem Ausdruck einfügen
T_TraceExpressionOut	Sensor nach einem Ausdruck einfügen
T_PrintMatch	Suchtreffer detailiert ausgeben

Abbildungsverzeichnis

Literaturverzeichnis

[BH98] Bernd Bauer and Riitta Höllerer. *Übersetzung objektorientierter Programmiersprachen*. Springer-Verlag, 1998.

[Böl99] Kai Böllert. *Die Weber*. *iX*, 5:74–78, 1999.

[Cza98] K. Czarnecki. *Generative Programming: Principles and Techniques of Software Engeneering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, Technische Universität Illmenau, Germany, 1998.

[ES92] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1992.

[Ger94] Martin Gergeleit. *Automatic Instrumentation of Object-Oriented Programs*. Technical report, German National Research Center for Information Technology, 1994.

[H+97] Jeffrey K. Hollingsworth et al. *MDL: A Language and Compiler for Dynamic Program Instrumentation*. Technical report, Computer Sciences Department, University of Maryland; Computer Sciences Department, University of Wisconsin, May 1997.

[HMC94] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille, editors. *Dynamic Program Instrumentation for Scalable Performance Tools*. Computer Sciences Department, University of Wisconsin-Madison, 1994.

[K+97] Gregor Kiczales et al. *Aspect-Oriented Programming*. Technical report, Xerox Palo Alto Research Center, 1997.

[Par79] D. L. Parnas. Designing software for ease of extension and contraction. In *Transaction on Software Engineering*. SE-5(2), 1979.

[SSPSS98] F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. *Design Rationale of the* PURE *Object-Oriented Embedded Operating System*. In

Proceeding of the International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems (Dipes '98), volume 5/6, Paderborn, Germany, October 1998.

[Str97] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3. edition, 1997.

[TM97] Craig Thompson and Frank Manola. *Component Software Clossary.* Technical report, Object Services and Consulting Inc., 1997.
URL: http://www.objs.com/survey/ComponentWareGlossary.htm.

[TM98] Ariel Tamches and Barton P. Miller. *Fine-Grained Dynamic Instrumentation of Commodity Operation System Kernels.* Technical report, Computer Sciences Department, University of Wisconsin, 1998.

[X+99] Zhichen Xu et al. *Dynamic Instrumentation of Threaded Applications.* Technical report, Computer Sciences Department, University of Wisconsin; Oracle Corporation, 1999.

[Xer99] Xerox Corporation. *AspectL Language Specification*, August 1999.