Interoperable Laufzeit-Werkzeuge für parallele und verteilte Systeme

Habilitationsschrift

vorgelegt von

Roland Wismüller

Lehrstuhl für Rechnertechnik und Rechnerorganisation
(LRR-TUM)
Institut für Informatik
Technische Universität München
D-80290 München, Germany
tel.: +49-89-289-28243
fax: +49-89-289-28232
email: wismuell@in.tum.de

März 2001

Hiermit versichere ich, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Hilfsmittel verwendet habe.

München, 9. März 2001

Zusammenfassung

In der vorliegenden Arbeit wird eine Methodik und eine Infrastruktur zur Realisierung interoperabler Laufzeit-Werkzeuge entwickelt. Bislang führte der Versuch, den Ablauf eines Programms gleichzeitig durch mehrere Werkzeuge zu überwachen und insbesondere auch zu steuern, fast immer zu einem massiven Fehlverhalten der Werkzeuge oder des untersuchten Programms, was eine Zusammenarbeit der Werkzeuge unmöglich machte. Anhand einer Fallstudie wird daher zunächst die Frage untersucht, welche Probleme dafür verantwortlich sind: Zum einen sind dies strukturelle und logische Konflikte, die im Zuge der Datenintegration zu lösen sind, zum anderen die Realisierung von Interaktionen zwischen Werkzeugen, d.h. die Verhaltensintegration.

Strukturelle Konflikte werden durch gleichzeitige Zugriffswünsche auf exklusive Betriebsmittel verursacht. Sie lassen sich meist nur durch ein gemeinsames Monitorsystem vermeiden, das die Schnittstelle zwischen Werkzeugen und ihrem Zielsystem darstellt. Logische Konflikte entstehen, wenn ein Werkzeug eine Modifikation am Zielsystem vornimmt, die die Funktion anderer Werkzeuge beeinflußt. Die systematische Analyse dieser Konflikte wird durch ein Modell des Zielsystems und der Werkzeug-Zugriffe ermöglicht. Erkannte Konflikte können dann durch geeignete Dienste des Monitorsystems behandelt oder von vorneherein vermieden werden.

Eine Verhaltensintegration, die nicht zu einer wechselseitigen Abhängigkeit der Werkzeuge führt, wird durch Mediatoren erreicht, die Ereignisse in einem Werkzeug überwachen und daraufhin Aktionen in anderen Werkzeugen auslösen. In der Regel sind in den Werkzeugen jedoch noch keine Schnittstellen für diese Ereignisse und Aktionen vorhanden. Die Schnittstellen werden daher erst zur Laufzeit mit Hilfe des Monitorsystems in die Werkzeuge eingebunden. Die Aufgaben des Mediators, nämlich die Ereignis/Aktions-Verknüpfung sowie die Nachrichtenübermittlung, werden ebenfalls durch das Monitorsystem unterstützt, das diese Funktionen ohnehin bereitstellen muß. Es wird dadurch faktisch zu einer ereignisorientierten Middleware.

Die beschriebenen Techniken unterstützen Interoperabilität sowohl bei Neuentwicklungen als auch bei *Legacy*-Systemen: Neu entwickelte Werkzeuge können (und sollten) bereits von sich aus Vorkehrungen zur Lösung von Konflikten und zur Interaktion mit anderen Werkzeugen treffen. Andererseits können diese Vorkehrungen weitestgehend auch noch nachträglich in ein Werkzeug eingebaut werden, ohne dessen Quellcode zu modifizieren. Dies erlaubt zum einen die Wiederverwendung existierender Werkzeuge, zum anderen entfällt beim Entwurf neuer Werkzeuge die Notwendigkeit, alle für eine Verhaltensintegration möglicherweise benötigten Ereignisse und Aktionen vorherzusehen.

Die entwickelten Konzepte wurden in das Monitorsystem OCM integriert, auf dessen Basis die in der Fallstudie untersuchte Werkzeugumgebung realisiert wurde. Dabei zeigte sich, daß trotz einiger Schwierigkeiten bei der Implementierung eine Infrastruktur erstellt werden kann, die die Interoperabilität von Werkzeugen so weit unterstützt, daß ihre einfache Kombination zu leistungsfähigen Umgebungen möglich ist. In der Praxis ermöglicht die vorliegende Arbeit damit erstmals die Erstellung von Werkzeugumgebungen auf Basis bereits existierender Werkzeuge, die nicht speziell für den Einsatz in diesen Umgebungen entwickelt wurden.

Danksagung

Der erfolgreiche Abschluß einer Forschungsarbeit, die sich über mehrere Jahre erstreckt, wäre ohne eine tatkräftige Unterstützung durch Freunde und Kollegen niemals möglich. Ich möchte mich an dieser Stelle bei all jenen Freunden und Kollegen ganz herzlich bedanken.

Zunächst gilt mein Dank Herrn Prof. Dr. Arndt Bode, an dessen Lehrstuhl ich seit über zehn Jahren in einem äußerst angenehmen Klima arbeiten durfte. Durch die hervorragenden Arbeitsbedingungen und die Freiheit, die Sie, Herr Bode, für jeden einzelnen am Lehrstuhl schaffen, haben Sie es mir und allen Kollegen ermöglicht, unserer Forschungsarbeit optimal nachzugehen. Besonders beeindruckt hat mich, daß Sie trotz aller organisatorischer Aufgaben, die Sie innerhalb und außerhalb der TU München übernommen haben, die inhaltliche und fachliche Diskussion am Lehrstuhl, aber auch die menschliche Seite niemals zu kurz kommen ließen.

Speziellen Dank schulde ich auch Herrn Prof. Bernd Brügge, Ph.D. und Herrn Prof. Dr. Helmar Burkhart, die sich trotz ihrer angespannten zeitlichen Situation bereit erklärt haben, die Zweitgutachten für meine Arbeit zu erstellen.

Bei meinem Kollegen Herrn Prof. Dr. Thomas Ludwig, mit dem ich in den letzten Jahren sehr eng und freundschaftlich zusammengearbeitet habe, möchte ich mich besonders herzlich bedanken. Wir haben in dieser Zeit gemeinsam nicht nur die Arbeitsgruppe „Werkzeuge für Parallelrechner“ geleitet, sondern auch die Projekte OMIS, OCM und THE TOOL-SET definiert und durchgeführt, ohne die meine Habilitationsarbeit in dieser Form nie entstanden wäre.

Neben den Ideen von Thomas Ludwig haben auch die zweier amerikanischer Forscher meine Arbeit maßgeblich beeinflußt. Herr Prof. Al Geist brachte mich auf den Gedanken, ein „symmetrisches“ System zu realisieren, bei dem die Unterscheidung zwischen Anwendung und Werkzeugen praktisch aufgehoben ist. Die Idee der dynamischen Anpassung der Werkzeuge geht auf Herrn Prof. Barton P. Miller zurück, an dessen Lehrstuhl ich einige Monate verbringen durfte. Vielen Dank für diese Anregungen!

Ein herzliches Dankeschön möchte ich meinen ehemaligen Kollegen Dr. Peter Braun, Dr. Michael Oberhuber, Dr. Stefan Petri, Dr. Georg Stellner und Dr. Jörg Trinitis aussprechen, auf deren solider Vorarbeit ich bei der Durchführung meines Forschungsprojekts aufbauen konnte. Aber auch mit meinen anderen Kollegen aus der Werkzeugentwickler-Gruppe, von denen viele den Lehrstuhl ebenfalls bereits verlassen haben, hatte ich viele anregende fachliche und auch nicht-fachliche Diskussionen, die mir nicht zuletzt dabei geholfen haben, meine Forschungsziele und -probleme unter neuen Blickwinkeln zu betrachten. Mein Dank geht an Dr. Rolf Borgeest, Bernward Dimke, Philipp Drum, Dr. Olav Hansen, Dr. Johann Krammer, Dr. Stefan Lamberts, Markus Lindermeier, Günther Rackl, Dr. Christian Röder, Dr. Matthias Schumann, Dr. Thomas Treml, Josef Weidendorfer und Dr. Ivan Zoraja.

Ebenso dankbar bin ich den vielen Studenten, die mich durch eine praktische Semesterarbeit, durch eine Diplomarbeit oder als Hilfskraft unterstützt haben. Ohne Ihre Hilfe wäre die Realisierung der Werkzeuge und des OCM sicherlich nicht möglich gewesen.

Viele Anregungen, insbesondere in Bezug auf die Nutzung und Erweiterung des OCM erhielt ich aus mehreren Kooperationen mit auswärtigen Universitäten, die das Monitorsystem und auch einige der Werkzeuge für ihre Forschungsarbeit verwenden. Meinen Kooperationspartnern Prof. Dr. Peter Kacsuk, Gabor Dózsa, Daniel Drótos, Dr. Marian Bubak, Dr. Wlodimierz Funika, Bartosz Balis, Prof. Dr. Peter Brezany und Krzysztof Sowa möchte ich hiermit für die äußerst angenehme und produktive Zusammenarbeit danken.

Nicht vergessen will ich diejenigen meiner Kolleginnen und Kollegen am Lehrstuhl, die seinen reibungslosen Betrieb ermöglichen und aufrechterhalten und dafür meinen größten Dank verdienen. Ich bedanke mich recht herzlich bei Esther Achammer, Renate Brunnhuber, Heike Eberhardt, Gabriele Wöllgens und Helga Springer, die mir in unserem Sekretariat stets bei allen organisatorischen und administrativen Problemen freundlich und kompetent weitergeholfen haben. Ebenso danke ich Herrn Klaus Tilk für die Einrichtung und Wartung unserer Rechner-Infrastruktur und seine Hilfsbereitschaft bei allen technischen Problemen.

Herr Dr. habil. Peter Luksch und Herr Prof. Dr. Thomas Ludwig haben die Aufgabe übernommen, meine Arbeit sehr gewissenhaft und kritisch durchzusehen. Dafür und für die zahlreichen Anregungen und Verbesserungsvorschläge möchte ich Euch recht herzlich danken.

Ganz spezielle Dankbarkeit schulde ich schließlich Frau Barbara Greese für ihre Hilfe und ihren Beistand in einer für mich sehr schwierigen, persönlichen Situation.

Roland Wismüller

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Nicht nur in der Computerindustrie sondern auch in anderen Industriebereichen wie Maschinenbau oder Kommunikationstechnik bildet die Erstellung von Software einen wesentlichen Teil des Entwicklungs- und Herstellungsaufwands eines Produkts. Dabei steigt die Komplexität der Software und damit auch die ihrer Entwicklung stetig an. Es sind nicht mehr nur einfache, sequentielle Abläufe zu programmieren, sondern nebenläufige Prozesse, die mit anderen Prozessen kooperieren müssen. Oft entstehen weit verteilte Systeme mit über Rechnergrenzen hinweg kommunizierenden Programmpaketen. Ein Trend geht derzeit zu mobilen Systemen, die sich durch dynamisch wechselnde Umgebungen und Kooperationen auszeichnen. All diese Entwicklungen führen dazu, daß es nahezu unmöglich geworden ist, das Verhalten eines Gesamtsystems allein aus dem statischen Programmcode seiner einzelnen Komponenten heraus zu verstehen. Daher ist eine *Beobachtung des dynamischen Verhaltens* derartiger Softwaresysteme in ihrer aktuellen Umgebung – gegebenenfalls ergänzt durch steuernde Eingriffe – unverzichtbar geworden. Zur Unterstützung dieses Prozesses dienen Laufzeit-Werkzeuge für parallele und verteilte Systeme.

1.1.1 Laufzeit-Werkzeuge

Laufzeit-Werkzeuge sind als Software-Werkzeuge definiert, deren Zielobjekte *aktive* Programmausführungen sind. Ihre allgemeine Aufgabe ist es, Information über das Ablaufverhalten eines Programms zu ermitteln und darzustellen sowie eine Modifikation dieses Verhaltens zu ermöglichen. Anwendung finden derartige Werkzeuge dabei naturgemäß in den späten Phasen des Programmentwicklungszyklus [32], insbesondere beim Test, bei der Fehlersuche und bei der Optimierung von Software. Ihr Einsatz ist jedoch nicht mit der Auslieferung eines Software-Produkts beendet. Vielmehr sind Laufzeit-Werkzeuge auch bei der Software-Wartung unverzichtbar, etwa bei erst im Betrieb auftretenden Fehlern. Daneben gibt es auch Werkzeuge, die speziell den Betrieb von Software unterstützen, z.B. Lastausgleichs- und Sicherungspunktsysteme.

Trotz ihrer offenkundigen Wichtigkeit hinkt die Entwicklung von Laufzeit-Werkzeugen der allgemeinen Entwicklung der Informatik hinterher. Speziell gilt dies für Online-Werkzeuge, die die erfaßten Daten sofort auswerten und daher auch gezielt in Ablauf eines Systems eingreifen können. Warum ist dies so? Ein entscheidender Grund ist die starke Abhängigkeit ihrer Realisierung von nahezu allen Komponenten der Ausführungsumgebung ihrer Zielprogramme. Dazu zählen insbesondere:

1. Die Hardware des Zielsystems.
   Da das Zielprogramm im allgemeinen als Maschinencode ausgeführt wird, ist dessen Überwachung inhärent von der Befehlssatzarchitektur des Zielsystems abhängig. Einige Überwachungsaufgaben sind zudem nur mit Hilfe spezieller Hardware-Unterstützung akzeptabel realisierbar. Dazu zählt die Überwachung von Speicher- und Cache-Zugriffen, für die Prozessoren verschiedenste Arten von Debug- und Zählerregistern bereitstellen [105, 278].

2. Das Betriebssystem.
   Zur Steuerung und Überwachung von Prozessen müssen Dienste des Betriebssystems eingesetzt werden (z.B. `ptrace` und `/proc` unter UNIX [205]), deren Syntax und Semantik nicht standardisiert ist.

3. Das Laufzeitsystem.
   Da eine wesentliche Aufgabe von Laufzeit-Werkzeugen für parallele und verteilte Systeme die Darstellung von Interaktionen zwischen Prozessen ist, die über ein Laufzeitsystem erfolgt, müssen die Werkzeuge notwendigerweise auch dieses überwachen. Bis auf seltene Ausnahmen (z.B. `MPI` [78, 48]) bieten Laufzeitsysteme jedoch keine definierten Schnittstellen für diese Überwachung an. Die Werkzeuge sind daher regelmäßig nicht nur von der Art des Laufzeitsystems abhängig, sondern sogar von dessen konkreter Implementierung.

4. Der Übersetzer.
   Von Laufzeit-Werkzeugen wird erwartet, daß sie die in der Regel auf Maschinenebene erfaßte Information dem Benutzer auf der Ebene der Quellsprache präsentieren. Dazu ist eine detaillierte Kenntnis über die Abbildung des Quellcodes in Maschinencode erforderlich, die vom Übersetzer in Form verschiedenster Symboltabellen (z.B. COFF [7], `.stabs` [156], DWARF2 [243], STI [205]) beschrieben wird, die nur bedingt standardisiert sind.

5. Die Programmiersprache.
   Einige Werkzeuge, wie z.B. Debugger müssen Anfragen in der Syntax der Programmiersprache entgegennehmen und auch Ergebnisse, etwa die Inhalte von Datenstrukturen, in der entsprechenden Form ausgeben. Andere Werkzeuge realisieren die Datenerfassung durch eine automatische Modifikation des Quellcodes und werden dadurch ebenfalls sprachabhängig.

Aufgrund dieser vielfältigen Abhängigkeiten ergaben sich bislang für die Werkzeugentwicklung nur drei Alternativen:

1. Es wird ein extremer Entwicklungs- und Wartungsaufwand betrieben, um ein Werkzeug für unterschiedlichste Zielsysteme zur Verfügung stellen zu können. Um den Aufwand in Grenzen zu halten, werden dabei meist werkzeuginterne, plattformunabhängige Schnittstellen definiert. Ein Beispiel für diese Vorgehensweise ist das Werkzeug `totalview` [69], das den einzigen parallelen Debugger darstellt, der derzeit für eine große Zahl von Zielsystemen verfügbar ist.

2. In kleineren Forschungs- und Entwicklungsprojekten wird zur Vermeidung dieses Aufwands nur eine sehr eingeschränkte Zahl von Zielplattformen unterstützt, oft sogar nur eine einzige. Angesichts der schnell fluktuierenden Systemlandschaft sind diese Zielplattformen bei Projektabschluß vielfach bereits veraltet, die Werkzeuge damit in der Praxis nicht mehr anwendbar. In dem ESPRIT-Projekt HAMLET [106] entstand beispielsweise eine Werkzeugumgebung für spezielle Transputer- und PowerPC-basierte Zielsysteme, die angesichts der hohen Portierungskosten nach Projektabschluß nicht vermarktet werden konnte.

3. Eine andere Möglichkeit zur Reduktion des Entwicklungs- und Wartungsaufwands ist die Einschränkung eines Werkzeugs auf spezielle Aspekte der Programmausführung. Dadurch können Abhängigkeiten zu einigen der angesprochenen Komponenten vermieden werden. So erlauben übliche „parallele" Debugger zwar die Untersuchung mehrerer Prozesse, nicht aber deren Interaktion mit Hilfe des Laufzeitsystems. Eine der wenigen Ausnahmen stellt der Debugger `totalview` dar, der über eine Schnittstelle zur Kommunikationsbibliothek MPI verfügt. Eine umgekehrte Situation ist bei Werkzeugen zur Leistungsanalyse paralleler und verteilter Programme zu beobachten. Sie liefern in der Regel nur Daten über die Kommunikation zwischen Prozessen. Praktisch kein Werkzeug bietet den zum Auffinden eines Leistungsengpasses eigentlich unverzichtbaren exakten Rückbezug zum Quellcode, um die dadurch entstehende Compilerabhängigkeiten zu vermeiden. Lediglich die neueste Version von `Vampir` stellt zumindest die statische Quellcodeposition von Kommunikationsaufrufen dar. Die dynamische Prozeduraufrufsequenz ist jedoch nicht verfügbar, so daß der Ort eines Engpasses immer noch im Dunkeln bleibt, wenn das untersuchte Programm z.B. Kommunikationsaufrufe in eigenen Funktionen kapselt.

Während die erste dieser Alternativen für die Werkzeugentwickler problematisch ist, resultieren die beiden anderen in Problemen für die Anwender der Werkzeuge. Wenn Werkzeuge nur jeweils für ein spezifisches Zielsystem verfügbar sind, führt dies bei einem Wechsel der Ausführungsplattform zu einem hohen Lernaufwand für den Anwender, der nun andere Werkzeuge benutzen muß. Wenn die Funktionalität der Werkzeuge eingeschränkt wird, sind im allgemeinen nicht mehr alle relevanten Aspekte eines Programmablaufs überwachbar, so daß die Werkzeuge für den Anwender nur noch bedingt brauchbar sind. Beides führt letztlich dazu, daß Laufzeit-Werkzeuge nur relativ spärlich eingesetzt werden [216].

Zur Lösung dieser Probleme bieten sich zwei einander ergänzende Ansätze an: Das Problem der geringen Portierbarkeit von Werkzeugen läßt sich durch die Einführung von Standard-Schnittstellen entschärfen, die Zielsystem-Abhängigkeiten kapseln. Ein Beispiel ist die schon erwähnte Schnittstelle zwischen `totalview` und MPI, die die Abhängigkeit des Debuggers von der jeweiligen MPI-Implementierung vermeidet [48]. Ein weitergehender Ansatz, der die Abhängigkeit von Hardware, Betriebssystem und Laufzeitsystem kapselt, ist die Definition universeller Schnittstellen zu Monitorsystemen, d.h. Software-Komponenten, die Basisfunktionen zur Überwachung und Steuerung von Programmabläufen auf niederer Abstraktionsebene zur Verfügung stellen. Eine derartige Schnittstelle wurde vom Autor zusammen mit Thomas Ludwig bereits in den Projekten OMIS und OCM spezifiziert und implementiert (siehe z.B. [270] und § 3.2.1).

Der eingeschränkte Funktionsumfang der einzelnen Werkzeuge wäre bei genauerer Betrachtung eigentlich kein Problem, wenn die Werkzeuge – was derzeit leider nicht der Fall ist – gleichzeitig in koordinierter Weise zur Überwachung desselben Programmablaufs verwendet werden könnten. Diese Eigenschaft von Werkzeugen wird mit dem Begriff *Interoperabilität* bezeichnet. Die Untersuchung, welche Probleme sich dabei ergeben, und wie diese gelöst werden können, stellt das Thema dieser Arbeit dar.

1.1.2 Interoperabilität von Laufzeit-Werkzeugen

Die Motivation für eine gleichzeitige Verwendung mehrerer Werkzeuge, sowie deren inhärente Schwierigkeit, läßt sich gut an Analogien außerhalb der Informatik demonstrieren. So stellt beispielsweise in der Medizin die Positronen-Emissions-Tomographie (PET) ein hervorragendes Mittel zur Beobachtung von Stoffwechselvorgängen im menschlichen Körper dar. Sie liefert äußerst kontrastreiche Bilder, die z.B. die sichere Erkennung von Krebs-Metastasen erlauben. Leider sind jedoch auf den Bildern die gewohnten Gewebestrukturen meist nicht erkennbar, was eine genaue Lokalisierung der Metastasen erschwert. Auf der anderen Seite liefert die Magnetresonanz-Tomographie (MRT) deutliche Bilder genau dieser Strukturen. Kombinierte PET/MRT-Aufnahmen erweisen sich daher als extrem wertvoll. Da die Aufnahmegeräte jedoch die Form von Röhren bzw. dicken Ringen besitzen, durch die der Patient hindurchgeschoben wird, ist eine gleichzeitige Aufnahme derzeit technisch unmöglich. Die Bilder müssen daher mühsam manuell überlagert werden, wobei eine leicht unterschiedliche Orientierung bzw. eine Bewegung des Patienten berücksichtigt werden muß [164].

Ein weiteres, alltägliches Beispiel ist das Kontern zweier Muttern. Offensichtlich ist dies unter Verwendung nur eines Schraubenschlüssels absolut unmöglich. Das Beispiel macht auch deutlich, daß die Fähigkeit zur Interoperabilität von den konkreten Realisierungen der Werkzeuge abhängt. Man denke an einen Mechaniker, dem nur zwei Steckschlüssel zur Verfügung stehen.

Doch zurück zur Informatik. Auch hier wären häufig einander ergänzende Informationen verschiedener Werkzeuge hilfreich. So zeigt ein Visualisierer den zeitlichen Verlauf der Kommunikation einer parallelen Anwendung, jedoch ohne genauen Bezug zu deren Quellcode. Auf der anderen Seite erlaubt ein Debugger, den Ablauf des Quellcodes im Detail zu betrachten, liefert jedoch keine Daten über die Kommunikation. Wie bei den PET/MRT-Bildern geht bei einer Erfassung diese Informationen in zwei Durchgängen die genaue Korrelation verloren.

Die gleichzeitige Verwendung zweier Werkzeuge zur Manipulation eines Programmablaufs kann wie in dem Kontermutter-Beispiel auch Funktionen ermöglichen, die mit einem Werkzeug alleine nicht realisierbar sind. Beispielsweise ermöglicht ein Werkzeug zur deterministischen Ablaufsteuerung zusätzlich zu einem Sicherungspunktgenerator das Rücksetzen eines Programms nicht nur auf gesicherte, sondern auf beliebige vergangene Zustände. Wir werden darauf in § 3 noch näher eingehen.

In den bisher genannten Fällen ist es der explizite Wunsch eines Anwenders, mehrere Werkzeuge simultan zu verwenden. Es gibt andere Situationen, in denen dies nicht so ist, trotzdem aber mehrere Werkzeuge gleichzeitig zum Einsatz kommen müssen:

- Anwendung eines Werkzeugs auf einem System, das standardmäßig von einem automatischen Laufzeit-Werkzeug kontrolliert wird. Beispielsweise kann ein Programmierer einen Debugger oder einen Leistungsanalysator auf einem Rechensystem einsetzen wollen, das über einen integrierten dynamischen Lastausgleich durch Prozeßverschiebung verfügt.

- Unabhängige Überwachung eines Systems durch den Programmierer und den Systemadministrator. Beide inspizieren dabei eventuell gemeinsame Aspekte des Ablaufs, jedoch aus unterschiedlichen Gründen. Ein Beispiel ist das transparente Monitoring paralleler Anwendungen in Rechenzentren, um Anwender auf mögliche Leistungsprobleme hinweisen zu können [194].

- Kooperative Untersuchung derselben Anwendung durch ein verteiltes Team. In diesem Fall überwachen in der Regel mehrere identische Werkzeuge dasselbe Zielsystem.

In der Praxis sind solche Szenarien allerdings meist noch Utopie. Oft scheitert eine simultane Anwendung zweier Werkzeuge auf dasselbe Zielprogramm bereits bei deren Start. Viele Werkzeuge benötigen eine spezielle Vorbereitung des zu untersuchenden Programmcodes bzw. die Verwendung spezieller, instrumentierter Programmierbibliotheken. Beispielsweise verwenden sowohl die Monitorumgebung TAU [157] als auch das Leistungsanalysewerkzeug `Vampir` [34] die *Profiling*-Schnittstelle von MPI [78], um Aufrufe dieser Bibliothek zu überwachen. Die Verwendung der Schnittstelle bedingt, daß das Zielprogramm mit einer zum Werkzeug gehörigen Bibliothek gebunden werden muß, die instrumentierte MPI-Funktionen unter deren Originalnamen bereitstellt. Naturgemäß kann ein Programm aber nur mit jeweils *einer* derartigen Bibliothek gebunden werden. Aus einem analogen Grund scheitert auf älteren UNIX-Systemen auch die zeitgleiche Verwendung des Werkzeugs `strace` zur Protokollierung von Systemaufrufen [4] mit dem Debugger `gdb` [224], der diese Aufrufe nicht in einfacher Weise überwachen kann. Die Ursache ist hier die Verwendung der Schnittstelle `ptrace` zur Prozeßkontrolle [205], deren mehrfache Nutzung für denselben Zielprozeß das Betriebssystem verbietet.

Selbst wenn der Start der Werkzeuge nicht an derartigen exklusiven Betriebsmitteln scheitert, kann ihre gegenseitige Beeinflussung die ordnungsgemäße Funktion beeinträchtigen. Ein Beispiel liefert die genannte Kombination von `gdb` und `strace` auf neueren UNIX-Systemen, die zur Prozeßkontrolle das Dateisystem `/proc` anbieten. Als Dateischnittstelle ist `/proc` grundsätzlich von mehreren Prozessen gleichzeitig nutzbar. Werden aber über diese Schnittstelle Modifikationen des Zielprogramms veranlaßt, muß das Werkzeug die Rückmeldung aus einer Datei lesen, die einen Kommunikationskanal realisiert. Wartet in diesem Moment zufällig auch ein zweites Werkzeug an dem Kanal, ist unbestimmt, welches der beiden diese Rückmeldung erhält. Ist `strace` bereits aktiv, hängt `gdb` in der Regel unmittelbar nach dem Start, da der Debugger vergeblich auf die Rückmeldung wartet, wenn er den Zielprozeß beim Ankoppeln anhält.

Eine gemeinsame Nutzung von Werkzeugen ohne negative Beeinflussungen ist zur Zeit nur möglich, wenn diese völlig disjunkte Aspekte eines Ablaufs überwachen. Beispielsweise könnte `strace` zusammen mit `Vampir` verwendet werden. Dies gilt jedoch auch hier nur angenähert, da `strace` das durch `Vampir` beobachtete Zeitverhalten stark verändert. Zudem ist keine zeitliche Korrelation der jeweils erfaßten Daten möglich.

Dazu wäre eine gezielte Zusammenarbeit der Werkzeuge notwendig, etwa der Austausch von (logischen) Zeitstempeln zur zeitlichen Synchronisation der Information. Eine derartige Zusammenarbeit ist derzeit jedoch nur in fest integrierten Werkzeugumgebungen realisiert, deren Werkzeuge gemeinsam und speziell für diese Zusammenarbeit entwickelt wurden.

1.2 Wissenschaftliche Fragestellung

Aus der obigen Diskussion geht eine grundlegende Fragestellung hervor, die mit dem Schlagwort „*interoperable Laufzeit-Werkzeuge*“ charakterisiert werden kann. Was bedeutet das im Detail? Zunächst nichts anderes, als daß mehrere Laufzeit-Werkzeuge dasselbe Zielsystem bzw. identische Teile davon in korrekter Weise überwachen und steuern. Uns interessiert hier jedoch speziell der Fall, in dem diese Werkzeuge weitgehend unabhängig voneinander sind, also gerade keine speziell entwickelte, fest integrierte Umgebung bilden. Die Motivation dafür kommt sowohl von Seiten des Werkzeugbenutzers als auch von Seiten des Werkzeugentwicklers.

Wie bereits angesprochen, weisen Werkzeuge bzw. Umgebungen, die eine umfassende Überwachung eines Programmablaufs erlauben, eine starke Abhängigkeit von der Zielplattform auf und sind daher aus Aufwandsgründen nur für spezifische Systeme verfügbar. Für den Anwender ergibt sich daher bei Systemwechsel ein hoher Umlernaufwand. Auf der anderen Seite hat jeder Programmierer gewisse, meist relativ einfache Lieblingswerkzeuge, mit denen er vertraut ist. Bei einem Problem, das unterschiedliche Sichten auf einen Programmablauf erfordert, entsteht daher intuitiv der Wunsch, diese Werkzeuge auch gleichzeitig einsetzen zu können. Sie wurden aber in der Regel von verschiedenen Herstellern unabhängig voneinander entwickelt.

Aus Sicht eines Werkzeugentwicklers ist die relativ starke Koppelung der Komponenten einer integrierten Umgebung nachteilig, da sie die Entwicklung, Wartung und Portierung dieser Komponenten erschwert. Eine unabhängige Entwicklung einzelner, kleiner Werkzeuge wäre erheblich vorteilhafter, wenn sichergestellt werden kann, daß diese später trotzdem bei Bedarf zusammenarbeiten können. Insbesondere erlaubt ein solches Vorgehen das „Ausfaktorisieren“ von Zielplattform-Abhängigkeiten und damit eine bessere Portabilität. Beispielsweise kann ein Debugger vorteilhaft in unabhängige Werkzeuge zur Prozeßkontrolle, Datenvisualisierung, Ablaufspur-Generierung etc. zerlegt werden. Schließlich böte die genannte Vorgehensweise auch kleinen und mittleren Software-Unternehmen die Chance, Umgebungen, die Synergieeffekte zwischen Werkzeugen nutzen, unter Wiederverwendung existierender Werkzeuge zusammenzustellen.

Das Ziel der Arbeit ist also nicht die Integration im Sinne einer Vereinigung verschiedener Werkzeuge, derart daß diese nicht mehr als einzelne Entitäten in Erscheinung treten. Vielmehr sollen die Werkzeuge unabhängig bleiben und nur bei Bedarf *auch* zusammen verwendbar sein. Die Relevanz dieses Ziels unterstreicht ein Zitat aus der Zusammenfassung eines von der NASA organisierten Workshops mit dem Ziel der Erhöhung der Effektivität und Benutzbarkeit von Hochleistungsrechnern [216]:

> *A high priority should be the establishment and support of a software tool development infrastructure in the context of which, individual developers can produce pieces of capability and functionality with the assurance that these can interoperate with other elements of the overall HPC system developed by others.*

Vor diesem Hintergrund will die vorliegende Arbeit folgende Fragen beantworten:

1. Aus welchen Gründen scheitert derzeit die Interoperation von Laufzeit-Werkzeugen? Welche negativen Beeinflussungen der Werkzeuge sind dafür verantwortlich?

2. Wie lassen sich diese Beeinflussungen so klassifizieren und modellieren, daß sie in systematischer Weise für gegebene Werkzeuge ermittelt und behandelt werden können?

3. Welche Anforderungen müssen von einer Software-Infrastruktur erfüllt werden, die es erlaubt, unabhängige Werkzeuge interoperabel zu machen und bei Bedarf zu funktionierenden Werkzeugumgebungen zusammenzustellen?

4. Wie ist bei diesem Prozeß des Zusammenstellens vorzugehen?

5. Ist die notwendige Infrastruktur praktikabel und mit adäquatem Aufwand realisierbar?

1.3 Aufbau der Arbeit

Die Arbeit ist derart aufgebaut, daß die genannten Fragen in der angegebenen Reihenfolge beantwortet werden. In § 2 erfolgt zunächst eine detailliertere Einführung in die betroffenen Themengebiete „Laufzeit-Werkzeuge“ und „Interoperabilität“, kombiniert mit einer Darstellung des Stands der Technik. Insbesondere werden existierende Ansätze für interoperable Werkzeugumgebungen betrachtet. Vor dem Hintergrund dieser Ansätze und der gegebenen Möglichkeiten von verteilten Monitorsystemen wird am Ende dieses Kapitels eine erste Idee entwickelt, wie Interoperabilität mit Hilfe derartiger Monitorsysteme unterstützt werden kann. In § 3 wird dann ein konkretes Anwendungsszenario für interoperable Werkzeuge vorgestellt. Anhand dieses Fallbeispiels werden die gegenseitigen Beeinflussungen der Werkzeuge analysiert. Eine Klassifikation und eine Modellierung dieser Beeinflussungen wird in § 4 vorgenommen. Dort wird gleichzeitig eine Vorgehensweise definiert, wie Werkzeuge zu interoperablen Umgebungen kombiniert werden können. Das folgende Kapitel 5 beschäftigt sich mit den Software-Infrastrukturen, die ein derartiges Vorgehen unterstützen. Zunächst werden die aus dem Fallbeispiel abgeleiteten Anforderungen an eine derartige Infrastruktur definiert. Ein Rückblick auf die in § 2 vorgestellten Ansätze zeigt, daß diese die gestellten Anforderungen nur teilweise erfüllen. Im weiteren Verlauf des Kapitels wird daher ein Konzept für eine Infrastruktur auf der Basis des Monitorsystems OCM entwickelt, das alle Anforderungen umsetzt. Die Implementierung der so entstehenden Monitoring-Middleware ist in § 6, die darauf aufbauende Realisierung des Fallbeispiels in § 7 beschrieben. Auf Basis der während dieser Realisierung gewonnenen Erfahrungen wird in § 8 eine erste Bewertung der entwickelten Konzepte und deren Umsetzung vorgenommen. Kapitel 9 faßt schließlich die wichtigsten Ergebnisse und Erkenntnisse der Arbeit zusammen und skizziert mögliche Erweiterungen und zukünftige Forschungsfelder.

Die Arbeit verfolgt dabei zwei sich ergänzende Ziele. Dies ist zum einen die Untersuchung und Realisierung einer konkreten Umgebung interoperabler Werkzeuge. Sie bildet den eher praktischen orientierten Teil der Arbeit. Der methodische Teil setzt sich die Definition einer allgemeinen Vorgehensweise für die Realisierung derartiger Umgebungen sowie die Konzeption einer unterstützenden Infrastruktur zum Ziel. Methodik und Infrastruktur sind dabei nicht auf die im praktischen Teil untersuchte Umgebung eingeschränkt, sondern für beliebige andere Szenarien nutzbar.

1.4 Umfeld der Arbeit

Die vorliegende Arbeit entstand im Umfeld langjähriger Forschungstätigkeiten auf dem Gebiet der parallelen Laufzeit-Werkzeuge, die am Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM) der Technischen Universität München im Rahmen des Sonderforschungsbereichs 342 und anderer Förderprojekte durchgeführt wurden und werden. Sie verknüpft viele Einzelresultate dieser Forschung zu einem konsistenten Gesamtergebnis.

Die Entwicklung begann dabei 1989 mit dem Projekt TOPSYS [17], in dem verschiedene Online-Werkzeuge zur Untersuchung paralleler Programme auf nachrichtengekoppelten Multiprozessorsystemen realisiert wurden. Dazu zählten zunächst ein Debugger, ein Visualisierer und ein Leistungsanalysesystem. Sie nutzten bereits ein gemeinsames Monitorsystem, dessen jeweilige Teilfunktionalitäten jedoch speziell auf diese Werkzeuge zugeschnitten waren. Später kamen unter anderem ein dynamisches Lastverwaltungssystem [141], ein Werkzeug zur deterministischen Ablaufsteuerung [168] und ein System zur Generierung konsistenter Sicherungpunkte [225] hinzu. Bereits damals gab es die Idee, einige der Werkzeuge zu einer Entwicklungsumgebung zusammenzufassen, insbesondere Debugger und Visualisierer, die sich in ihrer Funktion sehr gut ergänzten. Da jedoch weder Konzepte für ihre integrierte Benutzung noch für die sinnvolle Realisierung einer Umgebung interoperierender Werkzeuge existierten, konnte die Idee zu dieser Zeit noch nicht umgesetzt werden.

Basierend auf den mit TOPSYS gesammelten Erfahrungen wurde dann 1995 das Projekt OMIS gestartet, das Thema der Habilitationsarbeit von T. Ludwig ist [142]. Ziel des Projekts war eine erweiterbare, universelle Monitorschnittstelle für parallele Systeme, deren Dienste von mehreren, eigenständigen Werkzeugen simultan genutzt werden können. Gleichzeitig wurde das Projekt THE TOOL-SET definiert, das die Zusammenstellung der am LRR-TUM entwickelten Werkzeuge zu einer lose gekoppelten Umgebung auf Basis von OMIS anstrebte.

Nach der Fertigstellung des OMIS-konformen Monitorsystems OCM und der Anpassung erster Werkzeuge wurde daher das Thema „interoperable Werkzeuge“ im Rahmen der vorliegenden Arbeit genauer untersucht. Einen ersten Schritt stellte die Verbindung des OCM mit dem Sicherungspunktgenerator `CoCheck` dar, um die bis dahin lediglich spezifizierten Dienste zur Sicherungspunkterstellung zu realisieren. Technisch wurde dabei zwar noch eine feste Integration durch Kapselung vorgenommen, es wurden aber bereits erste Konzepte für die systematische Analyse und Behandlung von entstehenden Konflikten erstellt [235]. Diese Untersuchungen wurden in der vorliegenden Habilitationsschrift konsequent weitergeführt, so daß es nun auf Basis des OCM möglich ist, existierende Werkzeuge ohne bzw. mit nur geringfügigen Modifikationen interoperabel zu machen. Der OCM erfüllt dabei neben der Aufgabe des Monitoring auch die einer Middleware-Umgebung.

Zeitgleich wurden die Aktivitäten am LRR-TUM auf dem Gebiet „Monitoring“ in Richtung auf middleware-basierte Anwendungen ausgedehnt. In zwei Dissertationen wurden dabei Monitoring-Konzepte für derartige Anwendungen sowie darauf aufbauende Lastausgleichsmechanismen entwickelt [196, 132]. Auch bei diesen Arbeiten zeigte sich eine konzeptuelle Ähnlichkeit von Monitor- und Middleware-Systemen.

Kapitel 2

Laufzeit-Werkzeuge und Interoperabilität

Laufzeit-Werkzeuge und *Interoperabilität* sind die beiden zentralen Begriffe im Titel der vorliegenden Arbeit.

In den nachfolgenden Abschnitten werden diese Begriffe zunächst unabhängig voneinander mit Leben gefüllt und ihre Bedeutung im Kontext der Arbeit geklärt. Insbesondere wird der Begriff der Interoperabilität auf die hier interessierende Situation unabhängig entwickelter, kooperierender Software-Komponenten eingeschränkt werden. Eine Diskussion existierender Ansätze zur Unterstützung dieser Art von Interoperabilität zeigt den allgemeinen Stand der Technik in diesem Bereich auf.

Nach dieser getrennten Einführung in die beiden relevanten Gebiete wird eine Idealvorstellung interoperabler Laufzeit-Werkzeuge beschrieben, um in unmittelbarem Anschluß daran festzustellen, wie weit die Realität derzeit noch von dieser Vorstellung entfernt ist. Eine Analyse der Gründe für diese Diskrepanz zwischen Wunsch und Wirklichkeit führt schließlich zu einer Lösungsidee und zur Definition der exakten Zielsetzung der Forschungsarbeit.

2.1 Werkzeuge zur Laufzeituntersuchung paralleler und verteilter Programme

Das IEEE Glossar zur Software-Engineering-Terminologie [102] definiert ein ***Werkzeug*** als ein Computerprogramm, das die Entwicklung, den Test, die Analyse oder die Wartung eines anderen Computerprogramms oder dessen Dokumentation unterstützt. Unter einem ***Werkzeug zur Laufzeituntersuchung*** (kurz ***Laufzeit-Werkzeug***) wollen wir dementsprechend ein Programm verstehen, das zur Untersuchung und ggf. Modifikation des dynamischen Ablaufverhaltens eines anderen auf einem Computersystem ablaufenden Programms dient.

In der vorliegenden Arbeit betrachten wir ohne weitere explizite Erwähnung sowohl parallele als auch verteilte Computersysteme. Der Einfachheit halber werden wir im folgenden – sofern der Zusammenhang eindeutig ist – ein Laufzeit-Werkzeug für ein derartiges System oft schlicht als ***Werkzeug*** bezeichnen.

Das durch ein Werkzeug überwachte Programm wird üblicherweise als ***Zielprogramm*** bezeichnet. Dabei ist stets das ablaufende Programm, nicht dessen statischer Programmtext gemeint. Das Zielprogramm zusammen mit dem Computersystem, auf dem es ausgeführt wird, wird ***Zielsystem*** genannt.

2.1.1 Einsatz und Funktion

Laufzeit-Werkzeuge können in zwei grundlegend verschiedenen Anwendungsbereichen eingesetzt werden. Zum einen existieren Werkzeuge, die die effiziente *Ausführung* von Anwendungs-Software unterstützen. Zum anderen – und dies ist der weitaus häufigere Fall – kommen sie bei der *Software-Entwicklung* zum Einsatz. Im Gegensatz zu den hier ebenfalls verwendeten CASE-Werkzeugen, die zur Bearbeitung von Spezifikationen und statischen Programmrepräsentationen dienen, betrachten Laufzeit-Werkzeuge die Ausführung bereits ablauffähigen Programmcodes. Sie sind daher im Software-Entwicklungszyklus erst in den späteren Phasen von Nutzen. In dem in Abb. 2.1 dargestellten, weitverbreiteten Wasserfall-Modell sind dies die Phasen der Implementierung, der Integration sowie der Wartung. Nach Schach [209] verursachen diese Phasen mit einem mittleren Anteil von 12%, 6%, bzw. 67% den weitaus größten Anteil der Herstellungskosten von Software. Dem Einsatz von Laufzeit-Werkzeugen mit dem Ziel der Produktivitätssteigerung kommt daher trotz der zum Teil etwas stiefmütterlichen Behandlung im Rahmen der Informatik-Forschung und -Lehre eine erhebliche wirtschaftliche Bedeutung zu.

Betrachtet man die interne Funktionsweise von Laufzeit-Werkzeugen genauer, so kann man vier grundlegende Funktionsgruppen unterscheiden:

1. **Gewinnung von Information über einen Ablauf des Zielprogramms.**
   Wir können hier unterscheiden zwischen Information über den Programmzustand an einem bestimmten Punkt der Programmausführung und Information über Ereignisse während der Ausführung.

   Der ***Zustand*** des Zielprogramms umfaßt z.B. die ausführenden Rechnerknoten, die beteiligten Prozesse, sowie deren Prozeßzustand, der Befehlszähler, Speicher, Register und Betriebssystemzustand beinhaltet. Programmierbibliotheken, die im Zielprogramm benutzt werden, können ebenfalls (evtl. abstraktere) Zustandskomponenten definieren, z.B. Kommunikatoren in MPI [63, 78]. Nicht zuletzt aus diesem Grunde ist eine umfassende Definition des Programmzustands im Rahmen dieser Arbeit weder möglich noch sinnvoll. Wir werden dennoch in § 5.3.3.1 noch einmal auf dieses Thema zurückkommen müssen.

   Ein ***Ereignis*** kann man formal als einen Zustandsübergang definieren [272, 247]. Einem Ereignis kann in der Regel ein Zeitpunkt zugeordnet werden, jedoch keine Zeitdauer; Ereignisse sind in diesem Sinne atomar. Typische Ereignisse sind das Erreichen einer bestimmten Stelle im Programmcode, der Aufruf von bzw. die Rückkehr aus Funktionen und Systemaufrufen und das Versenden von Nachrichten (genauer: der Abschluß des Sendens). Auch hier können und wollen wir keine umfassende Definition aller denkbaren Ereignisse angeben.

2. **Aufbereitung und Analyse der Daten.**
   Die Art der Datenaufbereitung bzw. -analyse hängt naturgemäß stark von der Zielsetzung des jeweiligen Werkzeugs ab. Im allgemeinen fällt sie jedoch in eine oder mehrere der folgenden Klassen:

   (a) **Umsetzung auf ein höheres Abstraktionsniveau.**
   Information über den Ablauf des Zielprogramms wird oft auf dem Niveau des ablaufenden Maschinencodes erfaßt. Zur weiteren Verwendung wird diese Information auf ein höheres Abstraktionsniveau, z.B. das der Quellsprache oder des verwendeten parallelen Programmiermodells, transformiert. So könnte das Ereignis „*der Pro-*

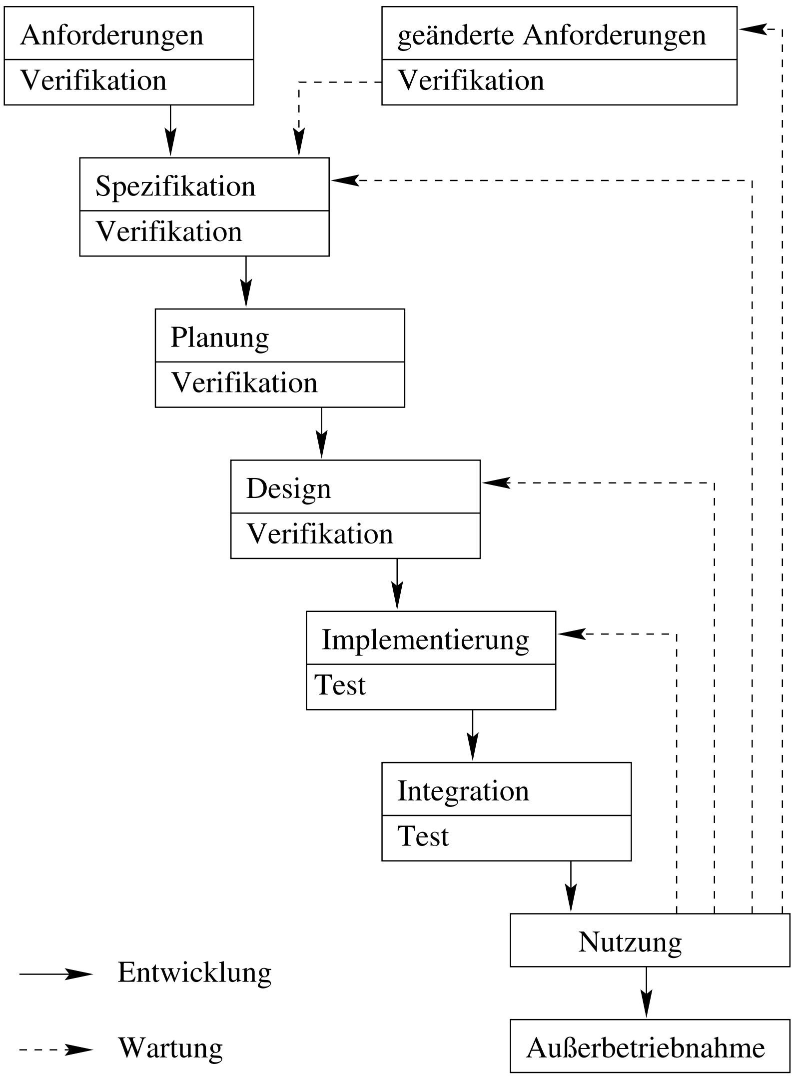


Abbildung 2.1: Wasserfallmodell des Software-Lebenszyklus (nach [209])

grammzähler erreicht den Wert 123456_{16}" umgesetzt werden in „*der Programmablauf erreicht Zeile 17 in Modul* `main.C`" oder in „*das Senden einer Nachricht ist abgeschlossen*".

(b) **Bestimmung „verborgener" Zustandskomponenten.**
Oft ist es für ein Werkzeug unmöglich bzw. unpraktikabel, bestimmte Komponenten des Zielprogramm-Zustands direkt zu erfassen. Als Beispiel sei die Empfangs-Nachrichtenwarteschlange eines Prozesses genannt, die in einer nicht offengelegten Art und Weise in einem Dienstprozeß des Laufzeitsystems gespeichert ist. Solche Zustandskomponenten können im Werkzeug oft aus zugehörigen Ereignissen berechnet werden, im Beispiel etwa aus Sende- und Empfangsereignissen.

(c) **Zusammenfassung von Information.**
Besteht die Aufgabe eines Werkzeugs darin, dem Programmierer einen Überblick über einen Programmablauf zu verschaffen, ist die geeignete Zusammenfassung der erfaßten Information die wesentliche Funktion der Datenanalyse. Im einfachsten

Fall kann dies das Zählen von Ereignissen oder die Bestimmung der Zeitdauer zwischen Paaren von Ereignissen sein, etwa zum Zweck der Leistungsanalyse.

Durch die Korrelation und Synthese verschiedener Daten kann darüber hinaus oft auch neue, für den Benutzer aus den Rohdaten nicht unmittelbar ersichtliche Information über einen Programmablauf gewonnen werden.

(d) **Bewertung der Information.**
„Intelligente" Werkzeuge, die den Benutzer selbständig auf Probleme im untersuchten Zielprogramm hinweisen (z.B. automatische Leistungsanalysewerkzeuge [152, 76]) oder den Ablauf des Zielprogramms selbsttätig modifizieren (z.B. Werkzeuge zum dynamischen Lastausgleich [141]) müssen schließlich in der Lage sein, die gewonnene Information in bezug auf vorgegebene Kriterien maschinell zu bewerten.

Jede dieser Funktionen benötigt zusätzliches Wissen, das über die erfaßte, reine Laufzeitinformation hinausgeht. Dieses Wissen kann entweder explizit, z.B. in Form einer Symboltabelle [205], oder implizit im Werkzeugcode gespeichert sein. Im Interesse der Portierbarkeit und Wartbarkeit des Werkzeugs sollte die explizite Darstellung vorgezogen werden. Das Leistungsanalyse-Werkzeug `Paradyn` [152] verwendet beispielsweise eine explizite Beschreibung des Zusammenhangs zwischen Leistungsmetriken und den zugrundeliegenden Ereignissen [98].

3. **Darstellung der aufbereiteten Information.**
Die Ergebnisse der Datenaufbereitung und -analyse können vom Werkzeug z.B. textuell, graphisch in Form von Kurven und Diagrammen, oder als Quellcode-Annotationen an den Benutzer weitergegeben werden.

4. **Manipulation des Programmablaufs.**
Als Ergebnis einer Auswertung der Information – entweder durch das Werkzeug selbst oder durch den Benutzer – kann ein Werkzeug schließlich auch den Ablauf des Zielprogramms modifizieren. Die Modifikation des Ablaufs erfolgt dabei indirekt durch eine entsprechende Änderung des Programmzustands. Die Bandbreite reicht dabei vom einfachen Anhalten und Weiterführen eines Prozesses bis hin zur Verlagerung eines Prozesses auf einen anderen Rechnerknoten.

Je nach Werkzeugtyp kann der Umfang der genannten Funktionsgruppen sehr unterschiedlich sein; einzelne der Gruppen können auch gänzlich fehlen. Die Funktionen, die unmittelbar mit dem Programmablauf interagieren, also die Gruppen 1 und 4, werden in dieser Arbeit unter dem Begriff ***Monitoring*** zusammengefaßt. Da Werkzeuge, die Programmabläufe manipulieren können, in der Literatur eher selten betrachtet werden, wird mit dem Begriff Monitoring sonst oft nur die reine Beobachtungstätigkeit verbunden.

2.1.2 Beispiele

Um unsere Diskussion nicht zu abstrakt werden zu lassen, wollen wir im folgenden einige Beispiele für konkrete Ausprägungen von Laufzeit-Werkzeugen vorstellen. Die Auswahl erhebt keinen Anspruch auf Vollständigkeit sondern soll in erster Linie die Bandbreite und die verschiedenen Konzepte existierender Werkzeuge verdeutlichen. Für eine vollständigere Diskussion verschiedener Laufzeit-Werkzeuge sei auf die einschlägige Literatur in Form von Büchern und Konferenzbänden, z.B. [101, 183] sowie auf verschiedene im WWW verfügbaren Listen und Übersichten [77, 161, 70, 182, 16] verwiesen.

Die nachfolgende Diskussion ist entsprechend der vorwiegenden Einsatzgebiete der Werkzeuge gegliedert:

- Software-Test
- Software-Analyse und Fehlersuche
- Software-Optimierung
- Software-Nutzung

Eine detailliertere Beschreibung von vier Werkzeugen, die das im Rahmen dieser Arbeit untersuchte Beispielszenario interoperabler Werkzeuge definieren, findet sich in § 3.2.

2.1.2.1 Software-Test

Da das Testen von Software eine weitgehend mechanische Tätigkeit ist, läßt es sich wirkungsvoll durch verschiedenste Werkzeuge unterstützen.

So können etwa Regressions- bzw. Protokolltests automatisch durchgeführt werden. Ein Werkzeug überwacht dazu vorgegebene Ereignisse im Zielprogramm und vergleicht diese bezüglich ihrer Art und Reihenfolge mit einer gegebenen Spezifikation des Sollverhaltens. Eine Abweichung führt zu einer Fehlermeldung bzw. zum Anhalten des Programms. Beispiele für solche Techniken aus dem Bereich des parallelen Programmierens sind *Event Based Behavioral Abstraction* [10], Cerberus [47], sowie der auf Temporallogik basierende Ansatz von Frey und Oberhuber [75]. Ein weiteres Beispiel ist das Werkzeug `purify`, das die korrekte Verwendung von Speicherverwaltungsfunktionen und Zeigern in C/C++ Programmen überprüft. Dazu wird vor dem Start des Zielprogramms der Objektcode so modifiziert, daß bei jedem Speicherzugriff über Zeiger sowie beim Aufruf relevanter Bibliotheksfunktionen entsprechende Prüfroutinen von `purify` aufgerufen werden [198].

Häufig werden während der Testphase auch Werkzeuge zur Ermittlung der Testüberdeckung eingesetzt. Diese überwachen die Ausführung von Basisblöcken des Quellcodes, wobei für jeden Block ein Ausführungszähler aktualisiert wird. Die Ausgabe des Werkzeugs kann in einer Liste nicht vom Test abgedeckter Code-Bereiche, einem mit der Ausführungshäufigkeit annotierten Quellcode sowie verschiedenen Statistiken bestehen. Ein kommerzielles Beispiel eines solchen Werkzeugs ist `PureCoverage` [197].

Für das Testen und auch die Fehlersuche in paralleler und verteilter Software sind Werkzeuge zur deterministischen Ausführung von zentraler Bedeutung. Nebenläufige Programme weisen einen inhärenten Nichtdeterminismus auf, der durch (absichtlich oder unabsichtlich) unsynchronisierte Zugriffe auf gemeinsame Objekte entsteht. Diese Situation wird als *Race* bezeichnet [166]. Bei nachrichtengekoppelten Programmen können beispielsweise zwei Prozesse unabhängig voneinander an einen dritten senden, d.h. Zugriffe auf dessen Nachrichtenwarteschlange ausführen. Die Reihenfolge, in der dieser die Nachrichten empfängt, kann dann von zufälligen Laufzeitschwankungen der Sender bzw. Schwankungen der Übertragungszeit abhängig sein. Reproduzierbares Testen ist somit nicht mehr möglich.

Werkzeuge zur deterministischen Ausführung überwachen die relevanten Objektzugriffe zur Laufzeit. Bei jedem versuchten Zugriff entscheidet das Werkzeug, ob dieser im Moment stattfinden darf oder bis zu einem späteren Zeitpunkt verzögert werden muß, was in der Regel durch Blockieren des zugreifenden Prozesses geschieht. Existierende Werkzeuge unterscheiden sich im wesentlichen in der Methode, wie diese Entscheidung gesteuert wird. `Mdb` [61] überläßt die Entscheidung völlig dem Benutzer und stellt als Hilfsmittel lediglich die Liste aller zugriffswilligen Prozesse zur Verfügung. *Replay*-Werkzeuge [134, 62] verwenden die in einem früheren Programmlauf aufgezeichnete Reihenfolgeinformation, können also lediglich einen vorherigen Ablauf reproduzieren. Die Werkzeuge `ATEMPT` [117] und `NOPE` [127] erlauben darüber hinaus die interaktive bzw. automatische Modifikation einzelner Zugriffsreihenfolgen und bietet damit eine bessere Testunterstützung. `Codex` [169] verzichtet auf aufgezeichnete Reihenfolgeinformation und basiert die Entscheidung über die Durchführung eines Zugriffs ausschließlich auf ein vom Programmierer spezifiziertes Zugriffsmuster (siehe § 3.2.5).

2.1.2.2 Software-Analyse und Fehlersuche

Ein häufiges und wichtiges Einsatzgebiet für Laufzeit-Werkzeuge ist die Analyse des logischen Ablaufverhaltens von Software. Diese kann sowohl der Fehlersuche (Debugging) als auch einfach dem Verständnis dienen, z.B. zum Zweck der Einarbeitung in einen fremden Code.

Eine weit verbreitete – da einfach zu realisierende – Klasse von derartigen Werkzeugen sind solche zur Ereignisspur-Aufzeichnung. Sie überwachen zur Laufzeit eine Klasse von Ereignissen, die in der Regel fest ist, aber vom Benutzer beim Programmstart geeignet eingeschränkt werden kann. Die registrierten Ereignisse werden dann entweder sofort dargestellt oder zur späteren Darstellung in einer Spurdatei gespeichert. Als Darstellungsformen finden textuelle Protokolle oder Graphiken, z.B. Gantt-Diagramme oder Prozeß-Zeit-Diagramme Verwendung. Die Datenverarbeitung innerhalb des Werkzeugs beschränkt sich dabei im wesentlichen auf das Ausfiltern nicht interessierender Ereignisse. Die Werkzeuge automatisieren die Erstellung jener Art von Ablaufprotokollen, die oft manuell mit Hilfe bedingter Ausgabeanweisungen in den Programmcode integriert wird. Beispiele solcher Werkzeuge sind `strace` [4], das ein textuelles Protokoll aller Systemaufrufe eines Prozesses erstellt, `xpvm` [88, 116, 123] zur graphischen und textuellen Ausgabe von Kommunikationsereignissen in PVM-Programmen zur Laufzeit, sowie `picl/paragraph` [267, 268, 92, 93], das derartige Ereignisse zunächst in einer Spurdatei speichert und nach Beendigung des Zielprogramms in vielfältiger Weise graphisch darstellt.

Eine Weiterentwicklung einfacher Ereignisspur-Werkzeuge stellen Visualisierer dar. Sie dienen der graphischen Veranschaulichung bestimmter Aspekte eines Programmablaufs, z.B. des Kommunikations- und Synchronisationsverhaltens oder der schrittweisen Berechnung einer Ergebnis-Datenstruktur. Für einen iterativen Gleichungslöser kann ein Visualisierer etwa die zeitliche Entwicklung der berechneten Näherungslösung darstellen. Auch Visualisierer überwachen eine Klasse vordefinierter Ereignisse im Zielprogramm, können aber beim Auftreten eines Ereignisses zusätzlich auch größere Teile des Zielprogramm-Zustands erfassen, z.B. den Inhalt von Datenstrukturen. Die Abgrenzung zu Ereignisspur-Werkzeugen ist nicht scharf; Visualisierer zeichnen sich jedoch durch eine komplexere Auswertung und/oder graphische Darstellung aus. Als Beispiele für Visualisierungs-Werkzeuge seien `MQM` [181] zur Untersuchung von Nachrichtenwarteschlangen, `VISTOP` zur Darstellung globaler Kommunikations- und Synchronisationszustände, `GDDT` [118] zur Visualisierung verteilter Felder sowie die integrierte Umgebung `Trapper` [220] genannt.

Debugger sind Werkzeuge, die der detaillierten Untersuchung eines Programmablaufs auf der Ebene des Quellprogramms oder evtl. auch des Maschinencodes dienen. Sie erlauben das Anhalten des Zielprogramms beim Auftreten von Ereignissen, die vom Benutzer zur Laufzeit definiert werden können (Haltepunkte). Mögliche Ereignisse sind das Erreichen einer bestimmten Zeile oder Prozedur im Quellprogramm, die Ausführung der nächsten Quellcode-Zeile (Einzelschrittabarbeitung) oder die Rückkehr der gerade aktiven Prozedur. Einige Debugger erlauben auch die Überwachung von Speicherzugriffen. Durch das Anhalten des Programms bzw. einzelner Prozesse wird deren Zustand sozusagen eingefroren und kann anschließend mit Hilfe des Debuggers im Detail untersucht und falls erforderlich auch modifiziert werden. Debugger erlauben dabei den Zugriff auf Register, Keller und Speicher der untersuchten Prozesse. Die Hauptaufgabe der Datenaufbereitung innerhalb eines Debuggers besteht in der Transformation der auf Maschinenebene gewonnenen Zustandsinformation auf die Ebene des Quellcodes, bzw. umgekehrt im Falle von Modifikationen. Um diese Aufgabe zu erleichtern, setzen Debugger eine sehr einfache, geradlinige Übersetzung des Quellcodes in ein Maschinenprogramm voraus, so daß sich die ergebende Abbildung in einfach strukturierten Symboltabellen darstellen läßt, die vom Compiler an den Debugger weitergereicht werden.

Obwohl gute Debugger auf der Werkzeug-Wunschliste der Programmierer an oberster Stelle stehen [179], sind sie zugleich die komplexesten und am schwierigsten zu realisierenden Laufzeit-Werkzeuge. Technische Probleme sind u.a. Skalierbarkeit im Hinblick auf die Anzahl der Prozesse im Zielprogramm [207], Unterstützung paralleler Programmiermodelle [48] sowie komplexe, evtl. mehrstufige Übersetzung des Zielprogramms mit dem Zweck der Optimierung oder Parallelisierung [254]. Die Implementierung eines Debuggers ist zudem von fast allen Komponenten des Zielsystems abhängig, wie z.B. der Hardware, dem Betriebssystem, der Programmiersprache, dem Übersetzer, der Laufzeitbibliothek sowie deren Implementierung, so daß der wirtschaftliche und ingenieurtechnische Aufwand zur Realisierung eines für verschiedene Zielsysteme verfügbaren Debuggers enorm ist. Aus diesem Grund existieren derzeit im wesentlichen nur zwei derartige Werkzeuge: der unter der GNU Lizenz der Free Software Foundation verfügbare Debugger `gdb` [224] für sequentielle Programme sowie das kommerzielle Produkt `totalview` [69] für parallele und verteilte Software. Forschungsprojekte auf dem Gebiet des parallelen und verteilten Debuggings, wie z.B. `p2d2` [49], stützen sich daher meist auf `gdb` ab. Der in § 3.2.2 beschriebene `DETOP` stellt eine der wenigen Ausnahmen dar.

2.1.2.3 Software-Optimierung

Der Prozeß der Software-Optimierung kann durch Werkzeuge unterstützt werden, die dem Programmierer helfen, Leistungsengpässe in seinem Programm aufzuspüren. Dies sind Programmabschnitte, die einen hohen Beitrag zur Gesamtlaufzeit des Programms leisten, sei es durch tatsächliche Berechnung oder durch Wartezeiten aufgrund von Kommunikation und Synchronisation. Fahringer et al. stellen in [73] einen ersten Ansatz zur Formalisierung solcher Leistungsengpässe dar.

Die Datenerfassung in einem Leistungsanalyse-Werkzeug kann prinzipiell auf zwei verschiedene Arten erfolgen: Beim *Sampling* wird der Ablauf des Zielprogramms in regelmäßigen Abständen (typischerweise 10 ms) unterbrochen und der aktuelle Befehlszählerstand bestimmt. Der Zuwachs der gemessenen Metrik seit der letzten Unterbrechung wird dann der Quellcodezeile oder -funktion zugeschlagen, in der sich der Befehlszähler befindet. Es entsteht so ein Profil dieser Metrik über Codezeilen bzw. -funktionen, das freilich aufgrund der Meßmethode nur statistischen Charakter hat. Die gemessene Metrik ist dabei in der Regel die reale Zeit oder CPU-Zeit, einige Werkzeuge können aber auch die Werte anderer, in der CPU integrierter Zähler erfassen, z.B. für die Anzahl von *Cache Misses* [278]. Beispiele für solche *Profiling*-Werkzeuge sind `prof/gprof` [81] und SGI `Speedshop` [212, 278].

Beim ereignisbasierten Ansatz überwacht das Werkzeug Beginn und Ende relevanter Aktio-

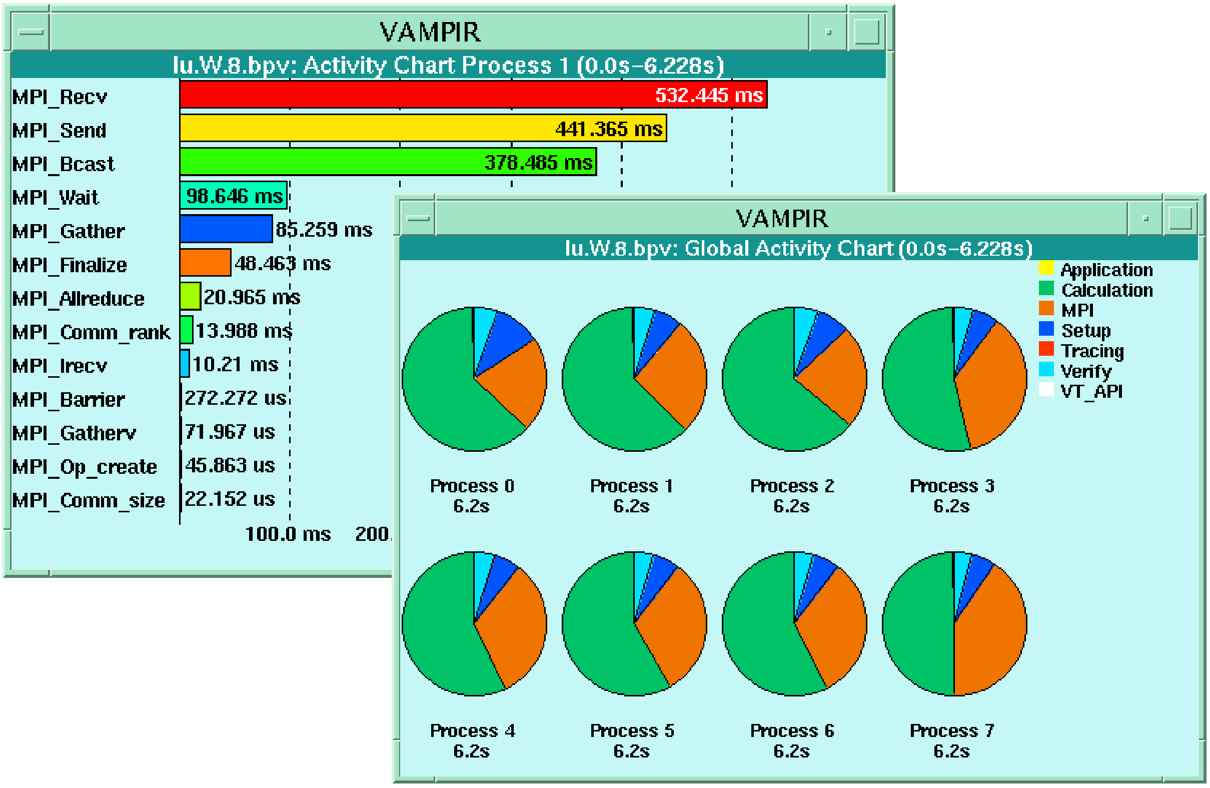


Abbildung 2.2: Verschiedene Darstellungen von Leistungsmeßgrößen im Werkzeug `Vampir`

---

nen im Zielprogramm, wie der Ausführung einer Funktion, Schleife, Kommunikationsoperation etc. Diese Ereignisse können entweder wie bei Ereignisspur-Werkzeugen aufgezeichnet und offline, d.h. nach der Beendigung des Zielprogramms ausgewertet oder online, also sofort beim Auftreten bearbeitet werden. Das Ergebnis der Auswertung kann im einfachsten Fall die Anzahl der aufgetretenen Ereignisse einer bestimmten Klasse sein oder der integrierte Wert einer Metrik (z.B. CPU-Zeit) für eine durch Anfangs- und Endereignisse begrenzte Aktion. Die so ermittelten Meßwerte werden i.a. vom Werkzeug in regelmäßigen Abständen ausgelesen und graphisch dargestellt. Die Meßwerte können dabei sowohl als Kurven über der Zeit als auch über längere Zeiträume gemittelt als Einzelwerte oder im Vergleich mit anderen Meßwerten angezeigt werden. Eine weit verbreitete Darstellungsform ist die Verteilung der Rechen- oder Kommunikationszeit auf die einzelnen Knoten, Prozesse oder Funktionen eines Programms (siehe Abb. 2.2). Beispiele für spurbasierte Leistungsanalyse-Werkzeuge sind `Vampir` [163, 177] und `TATOO` [15]. `PATOP` [266] und `Paradyn` [152] sind Vertreter der Klasse der Online-Werkzeuge, die die Auswertung der Ereignisse bereits zur Laufzeit vornehmen.

Eine einfache Leistungsanalyse kann meist bereits mit Ereignisspur-Werkzeugen durchgeführt werden. Komfortablere Werkzeuge wie `Vampir` bieten eine weit bessere Skalierbarkeit, indem sie ein Vorgehen unterstützen, das von einem groben Überblick über das Gesamtprogramm bis hin zur detaillierten Analyse mit Quellcodebezug reicht. `Paradyn` geht sogar einen Schritt weiter und mechanisiert dieses Vorgehen, indem Meßergebnisse zur Laufzeit bewertet und aufgrund der Bewertung automatisch neue, detailliertere Messungen gestartet werden, um einen vermuteten Leistungsengpaß zu verifizieren. Das automatische Aufspüren von Leistungsengpässen ist auch Gegenstand der ESPRIT Arbeitsgruppe APART [76].

2.1.2.4 Software-Nutzung

Wie in § 2.1.1 angesprochen, finden Laufzeit-Werkzeuge nicht nur bei der Programmentwicklung sondern auch beim Produktionseinsatz von Software Anwendung. Alltägliche Beispiele sind einfache UNIX-Kommandos zur Überwachung und Manipulation von Prozessen, wie `ps` und `kill`. Das andere Ende des Spektrums solcher Werkzeuge wird von *Steering*-Systemen gebildet. Prinzipiell kann man ein solches System als einen Visualisierer sehen, der zusätzlich die Möglichkeit bietet, in den Ablauf des untersuchten Programms einzugreifen, in der Regel durch Änderung von Parametern des implementierten Algorithmus. Ein häufiges Einsatzgebiet sind iterative Löser im Bereich numerischer Simulationen, die z.B. von den Systemen `Viper` [130, 202] und `Falcon` [86] adressiert werden. Diese Werkzeuge stellen das bisher berechnete Zwischenergebnis sowie andere Programmgrößen graphisch dar, die z.B. eine Bewertung des Konvergenzverhaltens erlauben. Durch Ändern bestimmter Parameter des Lösers (z.B. der Länge der Zeitschritte) kann durch einen erfahrenen Benutzer die Simulation gesteuert werden, etwa um eine schnellere Konvergenz zu erreichen oder numerische Instabilitäten zu vermeiden.

Eine andere Zielsetzung verfolgen Werkzeuge zur Sicherungspunkterstellung. Sie ermöglichen eine persistente Sicherung des gesamten momentanen Zustands eines Anwendungsprogramms derart, daß nach einem Programmabbruch – z.B. durch einen Hardware- oder Betriebssystem-Ausfall – die Programmausführung vom gesicherten Zustand an wieder aufgenommen werden kann. Die bis zu diesem Zeitpunkt durchgeführten Berechnungen müssen somit nicht wiederholt werden. Sicherungspunkte werden auch in Ressourcenverwaltungs-Systemen eingesetzt, die die Zuteilung verfügbarer Rechenressourcen an Rechenaufträge (*jobs*) regeln. Sie erlauben es diesen Systemen, laufende Aufträge nach dem Schreiben eines Sicherungspunkts temporär abzubrechen, um die Ressourcen höherpriorisierten Aufträgen zur Verfügung stellen zu können. Ein Beispiel eines solchen Systems ist `Condor` [139], dessen Sicherungspunktmechanismus allerdings für kommunizierende Prozesse und damit insbesondere für parallele Programme nicht geeignet ist (vgl. [244], Kap. 1.5). Eine Erweiterung dieses Mechanismus für parallele Programme ist mit `CoCheck` realisiert worden [218, 225]. Da ein Teil der zu sichernden Zustandsinformation im Betriebssystem (oder verwandten Software-Komponenten) gespeichert und somit für den Sicherungspunkt-Generator nicht direkt zugreifbar ist, berechnen die Werkzeuge oft, wie in § 2.1.1 besprochen, einen Teil des Zustands aus überwachten Ereignissen. `Condor` überwacht z.B. die Systemaufrufe `open`, `read`, `write`, etc. um die benötigte Information über offene Dateien zu gewinnen.

Eine Art automatischer *Steering*-Werkzeuge stellen Lastausgleichssysteme dar. Sie realisieren einen Regelkreis, dessen Ziel die möglichst gleichmäßige Verteilung der Last einer Anwendung auf die zur Verfügung stehenden Ressourcen ist. Die Last kann z.B. die Anzahl rechenbereiter Prozesse pro Knoten einer parallelen Maschine sein, wobei die Knoten die betrachteten Ressourcen darstellen. Eine ausführliche Diskussion des Lastbegriffs sowie verschiedener Szenarien und Implementierungsstrategien für Lastausgleichssysteme findet sich in [204]. Ein Lastausgleichssystem besteht aus vier Komponenten, die den genannten Regelkreis realisieren [141]. Die Lasterfassungskomponente mißt in unserem Beispiel die Auslastung der einzelnen Knoten. Die lokal gemessenen Lastwerte werden von einer Datensammlungskomponente zusammengefaßt. Eine Bewertungskomponente muß anschließend entscheiden, ob ein nennenswertes Ungleichgewicht vorliegt und ob dieses durch Verschieben eines oder mehrerer Prozesse auf einen anderen Knoten verringert werden kann. Ist dies der Fall, erfolgt eine Prozeßverschiebung durch die Verschiebekomponente durch Erstellen eines Sicherungspunkts, Übertragen desselben auf einen anderen Knoten, sowie Wiederanlauf des gesicherten Prozesses auf dem neuen

Knoten. Der Sicherungspunktmechanismus muß dabei sicherstellen, daß die Verschiebung für die Anwendung transparent bleibt.

2.1.3 Aufbau und Implementierung von Werkzeugen

Für unsere nachfolgenden Diskussionen ist es notwendig, eine Vorstellung von Aufbau und Funktionsweise von Laufzeit-Werkzeugen zu besitzen. Wir werden an dieser Stelle deshalb ein Modell für die Struktur solcher Werkzeuge beschreiben, das nahezu alle denkbaren Werkzeug-Realisierungen abdeckt. Im Anschluß daran folgt eine Übersicht über die Techniken, mit deren Hilfe Laufzeit-Werkzeuge Daten gewinnen bzw. Programmabläufe manipulieren.

2.1.3.1 Struktureller Aufbau von Laufzeit-Werkzeugen

In Abb. 2.3 ist die typische abstrakte Struktur eines Laufzeit-Werkzeugs dargestellt, wie sie in der Literatur häufig anzutreffen ist [142, 49, 152, 54]. Die stark umrandeten bzw. schattierten Komponenten gehören dabei zum Werkzeug, die restlichen stellen das Zielsystem dar, das aus der verteilten Hardware, dem Betriebssystem, dem Laufzeitsystem, einer Anzahl von Anwendungsprozessen[1] sowie eventuell weiteren, passiven Objekten besteht, z.B. Dateien oder Nachrichtenpuffer. Der Übersichtlichkeit halber sind die Kommunikationsbeziehungen im Zielsystem in der Abbildung nicht dargestellt.

Da wir parallele bzw. verteilte Zielsysteme betrachten, besteht ein Werkzeug im Regelfall aus einer lokalen Komponente pro Rechnerknoten[2] sowie einer zentralen Komponente und einer graphischen Benutzerschnittstelle, die meist auf einem Entwicklungsrechner ausgeführt werden. Je nach Plattform und Werkzeug-Typ können einzelne dieser Komponenten fehlen. Die Komponenten werden in der Regel durch je einen Prozeß realisiert, möglich ist jedoch auch ihre Realisierung durch mehrere Prozesse oder die Zusammenfassung mehrerer Komponenten in einem einzigen Prozeß.

Im allgemeinen Fall kommunizieren die lokalen Werkzeug-Komponenten sowohl untereinander als auch mit der zentralen Komponente, die wiederum in Kommunikationsbeziehung mit der Benutzerschnittstelle steht. Zur Vereinfachung der Implementierung wird jedoch – mit Ausnahme von Werkzeugen, die keine zentrale Komponente besitzen – meist auf eine direkte Kommunikation der lokalen Komponenten verzichtet, so daß eine baumartige Kommunikationsstruktur entsteht. Die Art und Weise der Kommunikation ist werkzeugspezifisch und kann über Kommunikationsbibliotheken wie MPI [78], PVM [83] und CORBA [211, 171], *Sockets* [226], UNIX-*Pipes* [227] oder auch Dateien erfolgen. Im letzteren Fall arbeiten die kommunizierenden Komponenten zeitlich versetzt im off-line Betrieb, so daß z.B. eine Darstellung der Daten durch die graphische Benutzerschnittstelle erst nach Beendigung der anderen Werkzeug-Komponenten erfolgt.

Betrachten wir die Aufteilung der in § 2.1.1 vorgestellten Funktionsgruppen auf die Werkzeug-Komponenten, wird deutlich, daß die Aufgaben der Informationsgewinnung sowie der Modifikation des untersuchten Programmablaufs von den lokalen Komponenten erfüllt werden müssen. Sie werden deshalb – zusammen mit dem zu ihrer Koordination notwendigen Teil der

[1] Die Tatsache, daß ein Prozeß mehrere *Threads* beinhalten kann, ist in unserem Zusammenhang nicht relevant und wird daher nicht weiter betrachtet.

[2] Ein Rechnerknoten kann mehrere Prozessoren beinhalten, sofern er eine einheitliche Systemsicht (*single system image*) bietet.

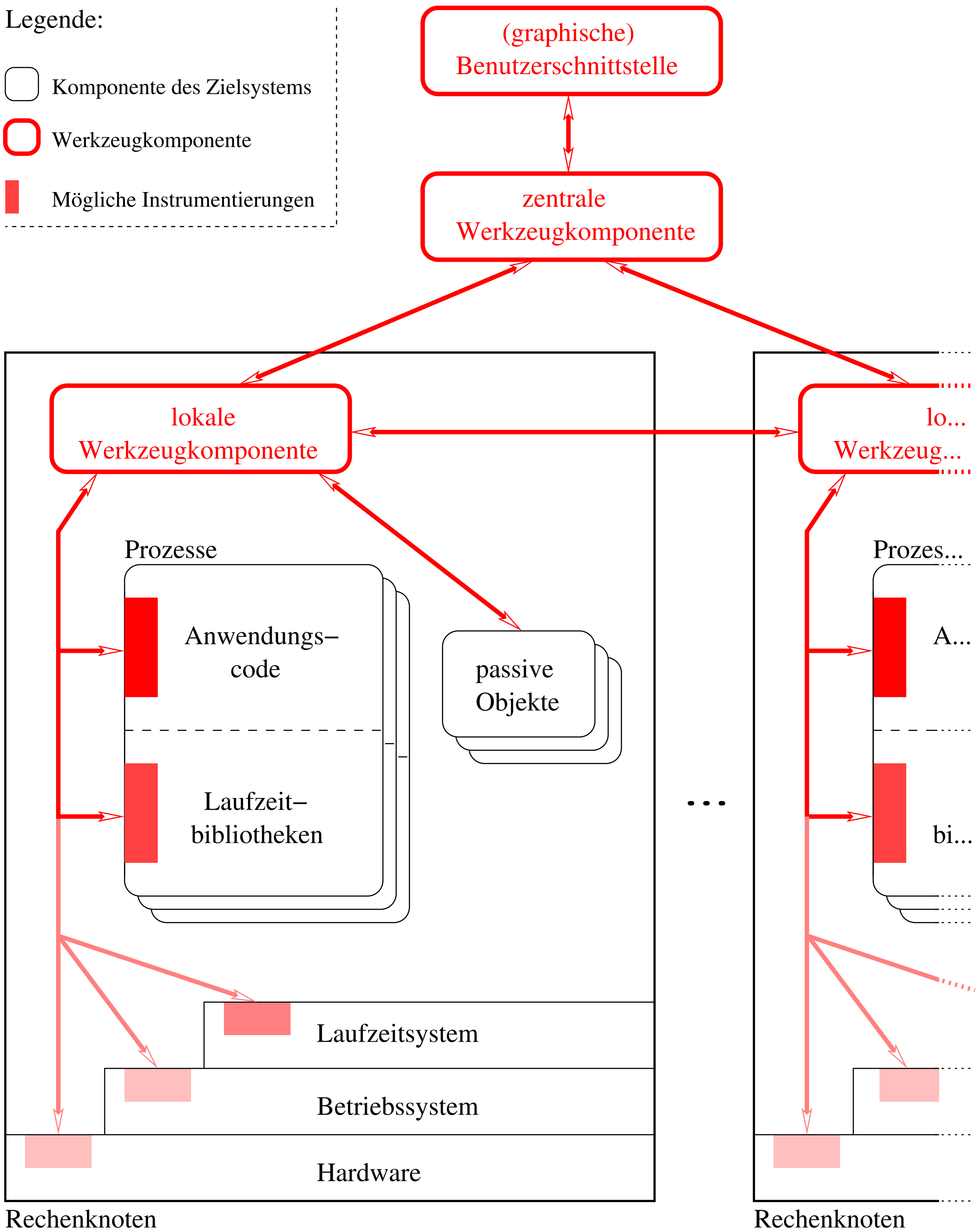


Abbildung 2.3: Struktur eines Laufzeit-Werkzeugs für parallele und verteilte Programme

zentralen Komponente – oft als ***Monitorsystem*** bezeichnet. Eine lokale Komponente auf jedem Knoten ist erforderlich, da die verwendeten Datenerfassungs- und Manipulationstechniken (siehe § 2.1.3.2) nicht über Knotengrenzen hinweg arbeiten. Eine Aufbereitung und Analyse der Daten kann in allen Komponenten vorgenommen werden, wobei in der zentralen Komponente die Zusammenfassung von Information verschiedener Knoten einen Hauptaspekt darstellt. Die Darstellung der Information erfolgt schließlich über die graphische Benutzerschnittstelle.

Das Monitorsystem arbeitet dabei häufig nach dem ***Ereignis/Aktions-Konzept***. Seine Hauptaufgabe besteht also darin, Ereignisse im Zielsystem zu überwachen und bei deren Auftreten vordefinierte Aktionen auszuführen. Die Verknüpfung von Ereignissen und Aktionen kann dabei – oft allerdings nur in begrenztem Umfang – durch das eigentliche Werkzeug festgelegt werden. Durch dieses Konzept kann das Monitorsystem Ereignisse, die sehr häufig auftreten können, weitgehend autonom verarbeiten, was die Rückwirkung auf das untersuchte Programm reduziert.

Wie in Abb. 2.3 angedeutet ist, steht das Monitorsystem mit Komponenten in Verbindung, die in den Anwendungscode, die Laufzeit-Bibliotheken, das Laufzeit- und Betriebssystem sowie gelegentlich sogar in die Hardware eingebunden sind. Sie dienen in erster Linie der Erkennung von Ereignissen im Zielsystem, können jedoch bei Bedarf auch Manipulationen am System vornehmen. Das Einbringen von Teilen des Monitorsystems in das Zielsystem wird als ***Instrumentierung*** bezeichnet und stellt eine grundlegende Realisierungstechnik aller Laufzeit-Werkzeuge dar. Wir werden die verschiedenen Techniken zur Instrumentierung im nachfolgenden Abschnitt erläutern.

2.1.3.2 Informationsgewinnung und Manipulation

Da die Art und Weise, wie Werkzeuge Informationen gewinnen bzw. den Ablauf des Zielprogramms manipulieren, einen zentralen Aspekt darstellt, wenn wir über die Zusammenarbeit mehrerer Werkzeuge nachdenken, wollen wir diese im folgenden etwas näher betrachten. Wir werden zunächst untersuchen, welche Schnittstellen für Laufzeit-Werkzeuge in den hier betrachteten Zielsystemen vorhanden sind, wobei wir in der in Abb. 2.3 dargestellten Schichtung von unten nach oben vorgehen wollen.

Hardware-Unterstützung für Werkzeuge findet sich in neueren Mikroprozessoren in Form spezieller Haltepunktregister und Zählerregister. Haltepunktregister erlauben die Erzeugung einer synchronen Unterbrechung bei Erreichen eines vorprogrammierten Befehlszählerstands bzw. in einigen Prozessoren, etwa dem Intel Pentium, auch bei Schreib- bzw. Lesezugriffen auf vorgegebene Speicheradressen [105]. Zählerregister zählen verschiedene leistungsrelevante Ereignisse innerhalb des Prozessors, z.B. die Anzahl der ausgeführten Gleitpunktoperationen oder die Anzahl von Cache-Fehlzugriffen [278]. Wie alle Hardware-Ressourcen können diese Zähler allerdings nur über das Betriebssystem angesprochen werden, wobei sich wegen der Diversität verschiedener Hardware jedoch noch keine Standards ausgebildet haben.

UNIX-ähnliche Betriebssysteme bieten Laufzeit-Werkzeugen zwei Mechanismen zur Informationsgewinnung und Manipulation. Zum einen kann ein Werkzeug – entsprechende Zugriffsrechte vorausgesetzt – über das Pseudo-Gerät `/dev/kmem` auf interne Datenstrukturen des Betriebssystemkerns, etwa die Prozeßtabelle, zugreifen. Ein einfaches Beispiel für ein derartiges Werkzeug ist das bekannte UNIX-Kommando `ps`. Zum anderen können Werkzeuge Mechanismen zur Interprozeßkommunikation nutzen. Neben den Standardmechanismen, wie z.B. dem Senden eines Signals an einen Zielprozeß, kommen vor allem spezielle Dienste zur Prozeßüberwachung und -steuerung zum Einsatz. Ein Quasi-Standard ist der `ptrace`-Systemaufruf [205], der das Anhalten und Weiterführen eines überwachten Prozesses, das Abfangen von Signalen an und Systemaufrufen durch diesen Prozeß, sowie den Lese- und Schreibzugriff auf dessen Adreßraum ermöglicht. Eine weiterentwickelte, jedoch zwischen verschiedenen Betriebssystemen noch sehr uneinheitliche Schnittstelle stellt das `/proc`-Dateisystem dar [205]. Es bietet neben der Funktionalität von `ptrace` weitere Prozeßinformationen über z.B. verbrauchte CPU-Zeit, offene Dateien, etc. und unterstützt auf einigen Plattformen auch leichtgewichtige

Prozesse. Im Gegensatz zur `ptrace`-Schnittstelle, die die Überwachung eines Zielprozesses durch nur jeweils einen einzigen anderen Prozeß erlaubt, gestattet das `/proc`-Dateisystem prinzipiell mehreren Prozessen gleichzeitig den Zugriff. Ereignisse wie z.B. Signale an den Zielprozeß werden jedoch bei mehreren wartenden Prozessen an nur einen, „zufällig“ ausgewählten Prozeß gemeldet.

Laufzeitsysteme und -bibliotheken, wie die parallelen Programmierbibliotheken PVM und MPI oder die Kommunikationsmiddleware CORBA, bieten in der Regel leider wenig direkte Unterstützung für Werkzeuge an. Diese wäre in erster Linie erforderlich, um Konfigurationsinformation und andere Zustandsinformation zu gewinnen [256]. Bei einer parallelen Programmierbibliothek wären dies etwa die beteiligten Rechnerknoten und Prozesse des Zielprogramms bzw. der Inhalt von Nachrichtenwarteschlangen. Die erstgenannte Information kann in PVM durch die Bibliotheksaufrufe `pvm_config` und `pvm_tasks` gewonnen werden. Zudem bietet PVM eine Schnittstelle zur Überwachung der Prozeßerzeugung an.

Übersetzer stellen Information für Laufzeit-Werkzeuge durch Symboltabellen bereit, die die Abbildung des Quellcodes in Maschinencode beschreiben [205, 137]. Sie werden von Werkzeugen benötigt, um Information zwischen diesen verschiedenen Abstraktionsniveaus transformieren zu können. Wie bereits in § 2.1.2.2 erwähnt wurde, ist die von den Übersetzern in der Symboltabelle bereitgestellte Information für diesen Zweck jedoch nur dann geeignet, wenn keine Optimierung bzw. automatische Parallelisierung des Quellcodes vorgenommen wurde, die zu einer wesentlich komplexeren Abbildung führt. Andere, vom Übersetzer aus dem Quellcode extrahierte Information, die für Laufzeit-Werkzeuge relevant wäre, wird nur von wenigen Übersetzern nach außen zur Verfügung gestellt. Beispiele sind die statische Programmstruktur, einschließlich der Schleifenstruktur, statische Prozeduraufrufbäume, Datenflußinformation etc. Ein Anregung, diese Information in Form einer „Super-Symboltabelle“ bereitzustellen, wurde 1995 von Bruce Irvin und Barton P. Miller unternommen [103], führte aber nicht zum Erfolg.

Die bisher beschriebenen, im Zielsystem standardmäßig vorhandenen Schnittstellen können für die meisten Werkzeuge nicht alle benötigten Informationen liefern. Für eine detaillierte Beobachtung bzw. Kontrolle des Zielsystems muß das Werkzeug daher das beobachtete System um Codeteile (bei Software-Komponenten) bzw. um zusätzliche Hardware (bei Hardware-Komponenten) erweitern. Dieser Vorgang wird als Instrumentierung bezeichnet. Die auf diese Weise in das Zielsystem integrierten ***Monitorkomponenten*** können neben der reinen Datenerfassung und Manipulation auch der Daten(vor)verarbeitung dienen. Sie sind in Abb. 2.3 durch die grau schattierten Kästchen dargestellt, wobei der unterschiedliche Schwärzungsgrad andeutet, wie häufig die jeweilige Instrumentierung in der Praxis vorkommt. Eine Instrumentierung der Zielhardware ist wegen des damit verbundenen manuellen Eingriffs sowie aus Kostengründen nur für spezielle Entwicklungssysteme praktikabel. Von einer Instrumentierung des Betriebssystems und des Laufzeitsystems wird meist ebenfalls Abstand genommen. Gründe dafür sind u.a., daß der notwendige Quellcode bzw. eine genügend detaillierte Dokumentation nicht verfügbar ist, für die Instrumentierung eine Privilegierung erforderlich wird und Sicherheitsbedenken bestehen, und nicht zuletzt die resultierende, starke Systemabhängigkeit des Werkzeugs, da z.B. bei einem Versionswechsel des Betriebs- bzw. Laufzeitsystems eine neue Instrumentierung erarbeitet werden muß.

Entsprechend der Methode, wie Monitorkomponenten in das Zielsystem eingebracht werden, kann man fünf Instrumentierungstechniken unterscheiden:

1. **Quellcode-Instrumentierung.**
   Hier wird zusätzlicher Code in den Quellcode des zu untersuchenden Programms ein-

gefügt, was manuell durch den Programmierer, automatisch mit Hilfe spezieller Präprozessoren oder bei Sprachen mit Makrounterstützung, wie z.B. C und C++ auch durch geeignet definierte Makros geschehen kann. Da diese Instrumentierungstechnik völlig plattformunabhängig ist und zudem einen sehr einfachen Quellcodebezug erlaubt, wird sie oft bei portablen Ereignisspur-Werkzeugen eingesetzt. Als Beispiele seien `Tape/PVM` [151] und `AIMS` [277] genannt, deren Präprozessoren u.a. alle Aufrufe von PVM-Funktionen so abändern, daß vor bzw. nach dem Aufruf ein Eintrag an die Ereignisspur angefügt wird.

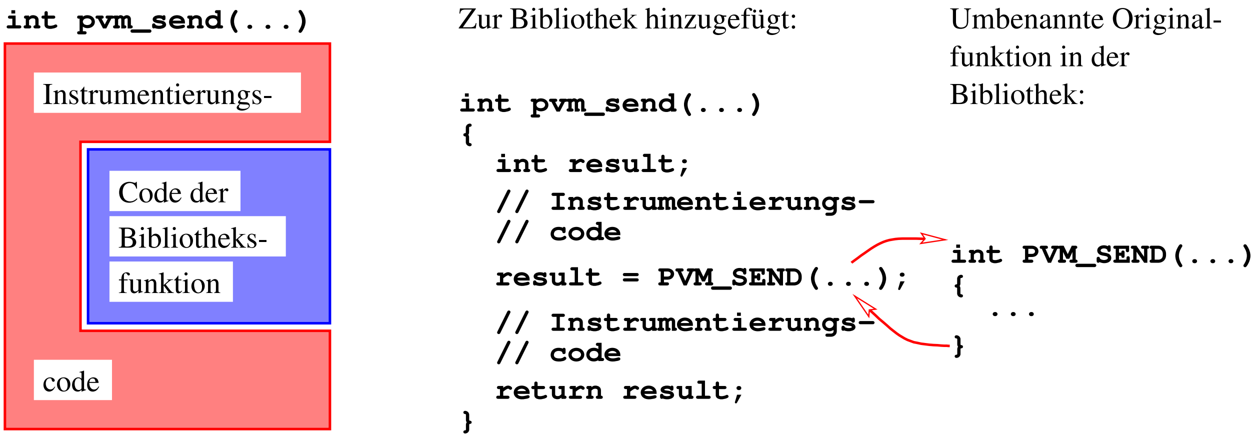
2. **Instrumentierung durch den Übersetzer.**
Die für ein Werkzeug benötigte Instrumentierung kann auch durch den Übersetzer in ein Programm eingefügt werden. Wegen der notwendigen Kooperation zwischen Werkzeug- und Übersetzerentwickler ist diese Instrumentierungstechnik eher selten, bietet aber den Vorteil, daß bei der Erzeugung des Instrumentierungscodes Wissen des Übersetzers über das zu untersuchende Programm ausgenutzt und somit indirekt dem Werkzeug zur Verfügung gestellt werden kann. Das bekannteste Beispiel für diese Instrumentierungstechnik ist die Option '`-pg`' von UNIX C-Compilern für das Leistungsanalyse-Werkzeug `gprof`. Die Option veranlaßt den Compiler u.a., in jede Funktion Code einzufügen, der die Anzahl der Aufrufe dieser Funktion mißt.

3. **Assemblercode-Instrumentierung.**
Prinzipiell kann die Instrumentierung auch in dem vom Übersetzer erzeugten Assemblercode vorgenommen werden, jedoch hat diese Technik bei den hier betrachteten Zielsystemen keine praktische Bedeutung. `SALTO` ist eine Bibliothek zur Unterstützung der notwendigen Assemblercode-Transformationen [13, 104].

4. **Objektcode- und Binärcode-Instrumentierung.**
Diese mit Abstand am häufigsten verwendete Instrumentierungstechnik läßt sich in drei Unterklassen gliedern, deren charakteristisches Unterscheidungsmerkmal der Zeitpunkt der Instrumentierung ist.

   Oft stellen Werkzeuge vorinstrumentierte Laufzeitbibliotheken zur Verfügung, die um zusätzliche Funktionen für das Werkzeug angereichert sind. Sie können prinzipiell durch die vorstehend beschriebenen drei Instrumentierungstechniken erzeugt werden. Da jedoch meist der Quellcode der Bibliotheken nicht zur Verfügung steht, wird im allgemeinen die Methode des *Binary Wrapping* [55] benutzt, die in Abb. 2.4 veranschaulicht ist. Sie erlaubt es – bildlich gesprochen – Funktionen der Bibliothek in Instrumentierungscode einzuwickeln, indem zunächst in der binären Version der Bibliothek der Funktionsname geändert und anschließend eine neue Funktion unter dem alten Namen in die Bibliothek eingefügt wird, die neben dem Aufruf der Originalfunktion zusätzliche Monitoring-Aktivitäten durchführt. Auf Ausführungsplattformen, die dynamisches Binden [131] unterstützen, kann eine analoge Technik auch gänzlich ohne Modifikation der eigentlichen Bibliothek vorgenommen werden.

   Während diese Technik nur ein Einfügen von Code vor und nach dem Aufruf von Bibliotheksfunktionen erlaubt, kann durch Umschreiben des Bibliothekscodes (*code rewriting*) während des Bindens oder Ladens des zu untersuchenden Programms eine beliebige Instrumentierung vorgenommen werden. Diese Instrumentierungstechnik wird beispielsweise von dem in § 2.1.2.1 erwähnten Werkzeug `purify` verwendet, das alle Schreib- und Lesezugriffe auf den Speicher um entsprechenden Prüfcode erweitert.



a) logische Sicht   b) technische Realisierung

Abbildung 2.4: Bibliotheksinstrumentierung durch *Binary Wrapping*

Spezielle *Code-Rewriting*-Bibliotheken, wie z.B. `EEL` [138, 242] realisieren plattformunabhängige Schnittstellen für diese inhärent maschinenabhängige Instrumentierungstechnik.

Noch einen Schritt weiter geht die Technik der dynamischen Instrumentierung [97, 98]. Die Instrumentierungsbibliothek `dyninst` [99, 241] bietet eine portable Schnittstelle zur Binärcode-Instrumentierung bereits laufender Prozesse, d.h. der Prozeßcode wird während der Laufzeit abgeändert. Ein Werkzeug kann dazu Code in Form eines abstrakten Syntaxbaums spezifizieren, der von der Bibliothek in Maschinencode übersetzt und mit Hilfe der Betriebssystem-Schnittstellen `ptrace` oder `/proc` an einer vom Werkzeug spezifizierten Stelle in den Prozeßcode eingefügt wird. Mögliche Instrumentierungspunkte, d.h. Einfügestellen, sind derzeit Beginn und Ende von Funktionen sowie Funktionsaufrufstellen. An einer Erweiterung für beliebige Instrumentierungspunkte wird noch gearbeitet. Neben dieser Art der Codemodifikation erlaubt `dyninst` auch den Aufruf von Funktionen im Kontext des überwachten Prozesses durch das Werkzeug (***Inferior RPC***), indem über `ptrace` bzw. `/proc` ein Kellerbereich für die Funktion abgelegt und der Befehlszähler des Prozesses entsprechend manipuliert wird. Auf Plattformen mit dynamischem Binden erlaubt dies das Einbinden auch größerer Codeteile durch einen vom Werkzeug erzwungenen Aufruf des dynamischen Binders (genauer: der Funktion `dlopen`), die dann durch den eingefügten Instrumentierungscode aufgerufen werden können. Die Technik der dynamischen Instrumentierung kann entsprechend angepaßt selbst zur Instrumentierung von laufenden Betriebssystem-Kernen verwendet werden [234].

5. **Hardware-Instrumentierung.**
Im Gegensatz zu den obigen Techniken, die ein Einfügen von Monitorcode in Software-Komponenten bewirken, wird in diesem Fall die Hardware des Zielsystems um Monitorkomponenten erweitert, die in Hardware realisiert sind. Wegen des technischen und finanziellen Aufwands wird Hardware-Instrumentierung nur in Ausnahmefällen eingesetzt. Als relevantes Beispiel aus dem Bereich paralleler und verteilter Systeme sei das Hardware-Monitorsystem des SMiLE-Projekts [96] genannt, das Zugriffe auf nicht-lokale Speicherbereiche überwacht. Eine Software-Instrumentierung kommt hier wegen des un-

akzeptablen Laufzeitaufwands nicht in Frage. Aus demselben Grund werden Hardwaremonitore oft auch in Umgebungen mit harten Echtzeitanforderungen eingesetzt.

2.1.4 Charakterisierung von Werkzeugen

Im folgenden wollen wir einige Begriffe zur Charakterisierung von Werkzeugen definieren, die wir für unsere nachfolgenden Diskussionen benötigen.

Wir wollen zunächst unterscheiden, ob ein Werkzeug das Zielprogramm lediglich beobachtet, ohne dessen Ablauf zu beeinflussen, oder ob es eine gezielte Modifikation des Ablaufs vornimmt. Wir sprechen demzufolge von ***beobachtenden*** bzw. ***modifizierenden Werkzeugen***. So naheliegend diese Unterscheidung sein mag, ist sie doch im Detail relativ problematisch. Wie wir im vorigen Abschnitt gesehen haben, erfolgt die Informationsgewinnung in jedem komplexeren Werkzeug durch Instrumentierung des Zielprogramms, die selbstverständlich eine Modifikation darstellt. In der Regel wollen wir diese Art der Modifikation jedoch ignorieren, da sie den Ablauf des Programms nicht beeinflußt, sofern wir ihn auf genügend abstrakter Ebene betrachten. Die geringe zeitliche Verzögerung des Programmablaufs durch die Instrumentierung kann jedoch durch den in Abschnitt § 2.1.2.1, Seite 13 angesprochenen Nichtdeterminismus paralleler Programme im ungünstigsten Fall zu erheblichen Veränderungen führen. Dies wird als *Probe Effect* bezeichnet [82]. Je nach betrachteter Situation werden wir deshalb manchmal ein an sich nur beobachtendes Werkzeug als modifizierend einstufen müssen.

Eine weitere Charakterisierung betrifft die Benutzerinteraktion. Wir unterscheiden zwischen ***interaktiven*** und ***automatischen*** Werkzeugen, je nachdem, ob der Benutzer während der Anwendung des Werkzeugs Entscheidungen über die zu erfassenden Daten bzw. durchzuführenden Manipulationen treffen muß oder ob diese vom Werkzeug selbst getroffen werden. Von den in § 2.1.2 erwähnten Werkzeugen zählen `purify`, `PureCoverage` und `Codex` zu den automatischen, ebenso wie Lastausgleichssysteme. `Paradyn` ist interaktiv, besitzt jedoch auch einen Modus zur automatischen Leistungsanalyse.

Betrachten wir den in Abb. 2.3 dargestellten verteilten Aufbau eines Werkzeugs, können wir weiterhin unterscheiden nach der Verteiltheit der Steuerung des Werkzeugs. Prinzipiell kann diese vollständig ***verteilt*** oder ***zentralisiert*** sein. Abgesehen von Werkzeugen zur kooperativen Untersuchung von Programmen durch verteilte Programmiererteams müssen interaktive Werkzeuge zwangsläufig über eine zentrale Benutzerschnittstelle verfügen und sind daher zentral gesteuert. Beispiele für vollständig verteilte Werkzeuge sind einige Lastausgleichssysteme sowie (abgesehen von der Initialisierung) das Werkzeug `Codex`.

Nach der zeitlichen Verschränkung der Ausführung der in § 2.1.1 beschriebenen Funktionsgruppen bzw. gleichbedeutend der in Abb. 2.3 dargestellten Werkzeugkomponenten können wir zwischen ***Online-*** und ***Offline-Werkzeugen*** unterscheiden. Bei Online-Werkzeugen werden alle diese Funktionen bzw. Komponenten zeitgleich mit dem zu beobachteten Programm ausgeführt. Bei Offline-Werkzeugen erfolgt die Darstellung der Information sowie der Großteil der Datenaufbereitung und -analyse erst nach Beendigung des Zielprogramms. Da eine Manipulation des Programmablaufs eine vorherige Analyse voraussetzt, ist diese bei Offline-Werkzeugen offenbar nicht möglich. Es sei darauf hingewiesen, daß jedes Offline-Werkzeug zwangsläufig einen Online-Teil besitzt, der für die Datenerfassung und eventuell für die Datenvorverarbeitung zuständig ist und für sich genommen als beobachtendes, automatisches, meist verteiltes Online-Werkzeug betrachtet werden kann.

2.2 Interoperabilität und Integration

2.2.1 Begriffsdefinition

2.2.1.1 Interoperabilität

Interoperabilität—(ist ein in neuerer Zeit gerne verwendeter, sehr schillernder Begriff, der eine wie auch immer geartete Zusammenarbeit zwischen Software-Komponenten ausdrücken soll. Die Ziele, die mit einer solchen Zusammenarbeit verbunden werden, sind dabei klar und unstrittig:

- Mehrwert für den Benutzer durch den Einsatz mehrerer kooperierender, auch verteilter Software-Komponenten.
- Vereinfachtes Software-Engineering durch Wiederverwendung von Komponenten sowie bessere Modularisierungsmöglichkeiten.

Die Ansichten darüber, was Interoperabilität jedoch konkret technisch bedeutet, gehen in der Literatur weit auseinander. Dementsprechend existieren viele verschiedene Definitionen des Begriffs, von denen wir einige exemplarisch zitieren wollen. Die ANSI/IEEE Definition aus [237] lautet:

> *Interoperability: [...] a quality metric to measure [...] the ability of two or more systems to exchange information and to mutually use the information that has been exchanged.*

Hervorgehoben ist hier also der Austausch und die Benutzung von Information, wobei letzteres bedeutet, daß die Komponenten mit den ausgetauschten Daten eine gemeinsame Semantik verbinden müssen. Fisher [74] stellt diesen Aspekt etwas klarer heraus:

> *Interoperability provides the capability for applications to communicate with each other reliably and to exchange information both syntactically and semantically.*

Wileden und Kaplan [261] betonen das Problem unterschiedlicher Implementierungssprachen:

> *Software interoperability is defined as the ability for multiple software components written in different programming languages (e.g, C++, Java, etc.) to communicate and interact with each other.*

Die Heterogenität bezüglich Sprachen und Ausführungsplattformen steht auch bei Howie, Kunz und Law im Vordergrund [95]:

> *Interoperability is usually defined as "the capability with which two or more programs can share and process information irrespective of their implementation language and platform."*

Das amerikanische *National Institute of Standards and Technology* betont dagegen Verteiltheit und Kooperation [11]:

> *Interoperability refers to the capability for applications running on different computers to exchange information and operate cooperatively using this information.*

Wegner [251] stellt schließlich noch einen neuen Aspekt heraus, nämlich daß die interagierenden Komponenten unabhängig voneinander entwickelt wurden und autonom agieren können:

> *Interaction among systems of heterogeneous independently developed systems of autonomous components is referred to as interoperation.*

Ebenso vielschichtig wie der Begriff selbst sind auch die mit ihm verbundenen Forschungsaspekte. Sie reichen von Infrastrukturen für eine sprach- und plattformunabhängige Kommunikation, z.B. CORBA [211, 171], über Koordinationsmethoden und -sprachen, z.B. Mediatoren [229], bis hin zu anwendungsorientierten Untersuchungen, z.B. Multi-Datenbanksysteme [120] und zur Definition von Standard-Schnittstellen für gemeinsame Daten, Ressourcen und Dienste, z.B. STEP für CAD-Daten [162, 80].

Interoperabilität an sich ist als Forschungsgegenstand dagegen sehr selten. In [95] werden vier Ebenen der Interoperabilität definiert:

1. Die physikalische Ebene bezieht sich auf die Medien, mit deren Hilfe Information ausgetauscht wird.

2. Die Datentyp-Ebene adressiert das Problem der Sprach- und Maschinenunabhängigkeit.

3. Auf der Spezifikations-Ebene müssen Unterschiede in der Repräsentation abstrakter Datentypen behandelt werden.

4. Die semantische Ebene muß schließlich sicherstellen, daß die beteiligten Komponenten dasselbe Verständnis der ausgetauschten Daten besitzen.

Unberücksichtigt bleibt in diesem Modell allerdings der Aspekt der Koordination, auf den wir in § 2.2.3 noch näher eingehen. Ähnliche Schichtenmodelle wurden auch in [31] und [112] aufgestellt.

2.2.1.2 Integration

Während Interoperabilität eine Eigenschaft von Software-Modulen ausdrückt, beschreibt der Begriff Integration zunächst einen technischen Vorgang. Bei Thyler [237] wird ***Integration*** definiert als „der Akt des Verbindens eines Software-Elements (oder mehrerer Elemente) mit einem anderen". Dabei wird implizit angenommen, daß diese Elemente miteinander interoperieren sollen – anderenfalls wäre die Integration sinnlos.

Für Ben-Shaul et al. [36] stellt die Integration die niedrigste Ebene der Programmierung von Interoperabilität dar, bei der existierende Komponenten in einen gemeinsamen Rahmen eingebaut werden, der gegenseitige Zugriffe der Komponenten ermöglicht. Die beiden nächsten Ebenen sind dann die Konfiguration der Komponentenbenutzung und der Kommunikation sowie die Koordination des Kontroll- und Datenflusses.

Häufig taucht in der Literatur auch der Begriff der ***integrierten (Entwurfs-)Umgebung*** auf. Er bezeichnet eine Menge von Werkzeugen, die einen (Entwurfs-)Prozeß durchgängig unterstützen. Die Bedeutung des Themas Werkzeugintegration im Bereich des computerunterstützten Entwurfs wird durch eine Anzahl von Buchveröffentlichungen, z.B. [232, 12], unterstrichen, bei denen jedoch die konkreten Realisierungen im Vordergrund stehen.

Die konzeptuellen Überlegungen gehen dabei meist auf Wassermann [248] zurück. Wassermann beschreibt Integration als eine dreidimensionale Menge von Attributen, die in Abb. 2.5

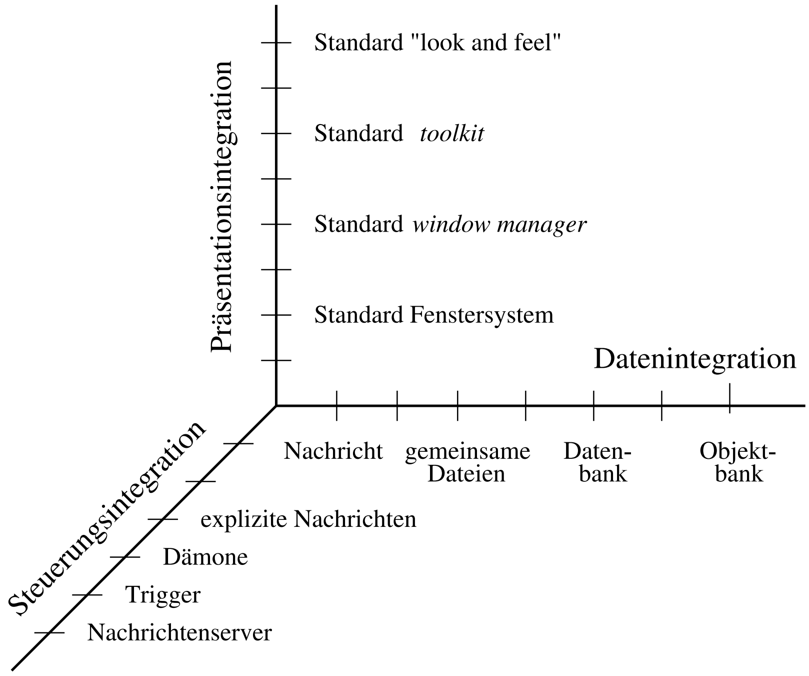


Abbildung 2.5: Drei Dimensionen der Werkzeugintegration (nach [248])

veranschaulicht ist. Die Präsentationsdimension beschreibt, wie einheitlich die verschiedenen Werkzeuge mit dem Benutzer interagieren, die Datendimension, wie der Zugriff auf gemeinsame Daten geregelt wird. Die Steuerungsdimension zeigt an, mit welchen Mechanismen sich die Werkzeuge untereinander koordinieren. Die letzten beiden Dimensionen beschreiben damit, wie die Interoperabilität der Werkzeuge realisiert wird.

Neben dieser implementierungsorientierten Sicht kann man integrierte Werkzeugumgebungen auch aus einer benutzerorientierten Perspektive betrachten. Brown et al. stellen sogar fest, „daß es wenig Zweck hat, die Integration von Werkzeug A mit Werkzeug B zu beschreiben, es sei denn, man erwartet von diesen zwei Werkzeugen, daß sie interagieren, um einen bestimmten Prozeß zu unterstützen“ [12]. Das dort entwickelte Integrationsmodell unterscheidet dementsprechend die Schichten „Mechanismen“, „Dienste“ und „Prozesse“. Brown et al. stellen auch heraus, daß zwischen „gut integriert“ und „leicht integrierbar“ unterschieden werden muß, da die Tatsache, daß eine Werkzeugumgebung gut integriert ist nicht zwangsläufig bedeutet, daß weitere Teile leicht integriert werden können.

2.2.2 Abgrenzung

Unsere Sichtweise des Begriffs Interoperabilität lehnt sich eng an die von Wegner [251] an. Wir betrachten eine Situation, in der wir unterschiedliche, unabhängig entwickelte Software-Module, die bereits fertig vorliegen, in möglichst einfacher Weise zu einem größeren Ganzen zusammenfügen wollen. Die Module weisen ein aktives, eigenständiges Verhalten auf, in dem Sinne, daß sie von sich aus mit ihrer Umgebung interagieren, also nicht lediglich reaktive Server darstellen, die ausschließlich auf externe Anfragen reagieren. Jedes Modul stellt ein vollständiges Computerprogramm dar, das in Form eines oder mehrerer Prozesse ausgeführt wird. Diese Sichtweise entspricht auch derjenigen aus der Literatur über *Megaprogramming* [271] und Koordinationssprachen und -modelle [1, 229, 191].

Die folgende Definition soll diese Sicht kurz zusammenfassen:

> ***Interoperabilität*** ist die Fähigkeit von unabhängig entwickelten, autonom agierenden Software-Komponenten, auf syntaktischer und semantischer Ebene zu kooperieren.

Die vorliegende Arbeit wird sich mit Techniken beschäftigen, die Interoperabilität in diesem Sinne ermöglichen und unterstützen. Im Sinne der vorher genannten Unterscheidung von Brown konzentrieren wir uns also auf die leichte Integrierbarkeit von Software-Komponenten. Ob diese aus Benutzersicht dann letztendlich gut integriert erscheinen, hängt vom Programmierer ab, der die eigentliche Integration durchführt und ist nicht primärer Gegenstand dieser Arbeit.

Da wir uns auf die technische Realisierung konzentrieren, ist die oben erwähnte Strukturierung von Wasserman in Daten-, Steuerungs- und Präsentationsintegration hilfreich. Wie Brown et al. feststellen, sind diese Aspekte jedoch nicht völlig orthogonal. So ist für den Zugriff auf gemeinsame Daten eine Steuerung in Form von Synchronisation erforderlich, während Nachrichten zur Steuerung natürlich auch Daten darstellen [12]. Der Aspekt der Präsentationsintegration beinhaltet sowohl die (graphische) Darstellung als auch die Benutzerführung. Erstere wird durch Standard-Fenstersysteme wie Motif [94] und Microsoft Windows unterstützt und ist nicht Gegenstand dieser Arbeit, zumal die Art der graphischen Darstellung einer bereits existierenden Software-Komponente nur mit sehr hohem Aufwand zu ändern ist. Die Koordination der Benutzerführung wird, wie Bergstra und Klint feststellen, bereits durch Mechanismen zur Steuerungsintegration abgedeckt [29]. Wir werden im folgenden daher nur noch zwei Aspekte der Integration unterscheiden: Die ***Datenintegration*** betrachtet den koordinierten Zugriff auf gemeinsame Daten sowie die damit implizit verbundene Synchronisation. Die Aufgaben der Steuerung der zu integrierenden Komponenten sowie der koordinierten Benutzerführung bedeuten abstrakt das Zusammenführen des individuellen Komponentenverhaltens zu einem konsistenten Gesamtverhalten. Sullivan bezeichnet dies in [229] mit dem Begriff ***Verhaltensintegration***, den wir im folgenden verwenden werden.

Unsere Sichtweise von Interoperabilität läßt sich auch charakterisieren, indem wir den Kopplungsgrad der einzelnen Software-Module betrachten. Qualitativ läßt sich unterscheiden zwischen lose und fest gekoppelten Modulen; einige Autoren, z.B. Trinitis [235] verwenden gleichbedeutend die Begriffe interoperable bzw. integrierte Komponenten. Wir werden uns im folgenden mit unterstützenden Infrastrukturen für lose gekoppelte Module befassen, d.h. wir betrachten unabhängig entwickelte Software-Komponenten, die interaktiv sind und eine eigenständige Ablaufsteuerung aufweisen. Sie sind zueinander nebenläufig als Prozesse realisiert, die sich über Nachrichtenaustausch koordinieren. Die Komponenten sind jedoch noch soweit voneinander unabhängig, daß sie einzeln ausführbar bleiben.

Im Gegensatz dazu zeichnen sich eng gekoppelte Umgebungen durch eine gemeinsame Entwicklung der Komponenten aus. Diese werden in der Regel von einer übergeordneten zentralen Komponente gesteuert und teilen sich wesentliche Datenstrukturen. Die Komponenten werden daher oft zu einem einzigen Prozeß zusammengefaßt und sind nicht einzeln ablauffähig.

Zur sprachlichen Vereinfachung werden wir im folgenden oft von unabhängigen Komponenten sprechen. Der Begriff ***Unabhängigkeit*** soll dabei die folgenden Eigenschaften ausdrücken:

1. Die Komponenten wurden *unabhängig entwickelt*, besitzen also kein Wissen übereinander.

2. Die Komponenten sind auch einzeln einsetzbar, also *nicht von anderen Komponenten abhängig*.

Der Begriff soll jedoch keinesfalls implizieren, daß die Komponenten bei ihrer Ausführung keine unerwünschten oder problematischen Wechselwirkungen zeigen. Die Analyse solcher Wechselwirkungen bildet im Gegenteil einen Schwerpunkt dieser Arbeit.

2.2.3 Ansätze zur Unterstützung von Interoperabilität

Wir wollen nun allgemeine Ansätze vorstellen, die in den letzten Jahren entwickelt wurden, um die Koppelung unabhängiger Software-Module zu unterstützen. Wegen der Vielzahl der unterschiedlichen Ansätze aus den verschiedenen Bereichen der Informatik wollen wir nur repräsentativ ausgewählte Beispiele betrachten. Eine etwas ausführlichere Darstellung findet sich in [208] und [12].

2.2.3.1 Datenintegration

Wie im letzten Abschnitt ausgeführt, subsumieren wir unter diesem Begriff nur den koordinierten Zugriff auf gemeinsame Daten. Prinzipiell unterstützt damit jedes Standard-Datenformat und jede Schnittstelle zum Zugriff auf Daten, z.B. Datenbanken, die Datenintegration. Für Laufzeit-Werkzeuge sind in diesem Zusammenhang vor allem Standard-Monitorschnittstellen und -systeme von Bedeutung, auf die wir in § 2.3.4.1 noch gesondert eingehen werden.

Als wichtigsten Vertreter speziell für die Werkzeugintegration entwickelter Schnittstellen werden wir an dieser Stelle PCTE (*Portable Common Tool Environment*) betrachten.

PCTE. PCTE ist eine Integrations-Infrastruktur für verteilte Werkzeugumgebungen im Bereich des Software-Engineering, deren Entwicklung 1983 mit einem gleichnamigen ESPRIT Projekt begann und bis zur Standardisierung durch die ECMA [64, 65] und ISO/IEC [107, 108] weitergeführt wurde. Die genaueren Hintergründe und Entwicklungswege des Projekts sind in [233] und [133] nachzulesen.

Das Ziel von PCTE war, alle in Abschnitt 2.2.1.2 genannten Integrationsaspekte abzudecken; der Schwerpunkt liegt aber deutlich auf der Datenintegration, sowie in geringerem Ausmaß auf der Verhaltensintegration. Datenintegration wird in PCTE durch ein verteiltes Objektverwaltungssystem unterstützt, das prinzipiell ein einfaches *Entity-Relationship*-Datenbanksystem darstellt.

Die Daten werden in Form von Objekten gespeichert, die neben dem eigentlichen Inhalt Attribute sowie Verbindungen (Relationen) zu anderen Objekten besitzen. Die Navigation durch die Daten erfolgt durch Verfolgen dieser Verbindungen. Die Objekte ersetzen dabei die in herkömmlichen Software-Engineering-Umgebungen verwendeten Dateien, die Spezifikationen, Quellcode etc. enthalten. Die Verwendung feinergranularer Objekte ist aus Gründen des Laufzeit-Overheads von PCTE fraglich [133, 12]. Objekte, Attribute und Relationen sind typisiert, wobei die Typinformation in Schemadefinitionen enthalten ist, die ihrerseits wieder als Objekte gespeichert sind.

Neben der eigentlichen Datenspeicherung beinhaltet PCTE Mechanismen für die Versionskontrolle, für die Datensicherheit und für die Kontrolle nebenläufiger Zugriffe. Eine einfache Versionskontrolle wird durch die Verwaltung von Vorgängerbäumen erreicht. Eine neue Version eines Objekts ist eine vollständige Kopie, die einen Verweis auf die ursprüngliche Version enthält. Das Anlegen und die Verwaltung dieser Versionen liegt in der Verantwortlichkeit der mit PCTE integrierten Werkzeuge.

Datensicherheit umfaßt zwei Aspekte: Sicherheit gegen den Ausfall von Rechnern oder Platten kann durch Replikation von Objekten erreicht werden. Sicherheit vor unbefugtem Zugriff wird durch Zugriffskontroll-Listen erreicht, die für jedes Objekt die Zugriffsrechte von einzelnen Benutzern, Benutzergruppen und Programmgruppen festlegen. Zusätzlich existiert ein zweiter Mechanismus, der über Sicherheitsstufen von Benutzern, Programmen und Objekten gesteuert wird.

Zur Koordination nebenläufiger Objektzugriffe stellt PCTE ein komplexes Sperrprotokoll mit insgesamt 11 verschiedenen Lese-, Schreib- und Löschsperren bereit. Die Möglichkeiten des Protokolls reichen von völlig ungeschützten Zugriffen über *Multiple-Reader-Locks* bis hin zu Transaktionen. Mechanismen zur Vermeidung oder Behandlung von Deadlocks sind in PCTE allerdings nicht definiert.

Trotz der sehr mächtigen Konzepte und der Standardisierung hat sich PCTE bisher nicht durchsetzen können, da die Umstellung existierender Werkzeuge auf PCTE sehr aufwendig ist (vgl. § 2.3.2.5). Eine allgemeine Infrastruktur für die Datenintegration ist ohnehin nicht realisierbar, da unterschiedliche Anwendungsgebiete auch unterschiedliche Datenmodelle erfordern. Für den Bereich der Laufzeit-Werkzeuge sind daher spezielle Ansätze notwendig, die in § 2.3.4.1 diskutiert werden.

2.2.3.2 Verhaltensintegration

Der Vorgang der Verhaltensintegration kann nach Sullivan [229] wie folgt beschrieben werden: In der Ausgangssituation liegen zwei (oder mehrere) unabhängig voneinander funktionierende Software-Komponenten A und B vor. Jede dieser Komponenten besitzt ein eigenes, charakteristisches Verhalten, z.B. die Reaktion auf äußere Eingaben durch den Benutzer. Das integrierte Gesamtsystem aus A und B soll nun nicht nur das vereinigte Verhalten von A und B aufweisen, sondern zusätzliche Relationen erfüllen, die das individuelle Verhalten dieser Komponenten miteinander koppeln. Diese Relationen lassen sich auf die allgemeine Form „falls A die Aktion x durchführt, soll B mit der Aktion y darauf reagieren“ zurückführen.

Als anschauliches Beispiel beschreibt Sullivan die Integration eines Editors mit einem Compiler: Wenn durch den Editor ein Programm-Quelltext nach einer Änderung in einer Datei abgespeichert wird, soll der Compiler diese Datei neu übersetzen.

Prinzipielle Entwurfsmethoden. Um diese Verhaltensintegration zu erreichen, sind nach Sullivan prinzipiell fünf verschiedene Entwurfsmethoden möglich.

1. ***Kapselung***
   Hierbei werden die zu integrierenden Module in einen gemeinsamen Code (*wrapper*) eingebettet, der die Schnittstellen dieser Module nach außen spiegelt, dabei aber auch die zusätzlichen Verhaltensrelationen realisiert. In dem Beispiel würde der *Wrapper*-Code eine Methode zum Speichern einer editierten Datei zur Verfügung stellen, die zunächst die Speicher-Methode des eingekapselten Editors aufruft und anschließend die Übersetzungs-Methode des Compilers (siehe Abb. 2.6).

   Der entscheidende Nachteil dieser Methode ist die entstehende monolithische Systemstruktur, die eine einfache Wartbarkeit und Erweiterbarkeit verhindert. Die nachfolgen-

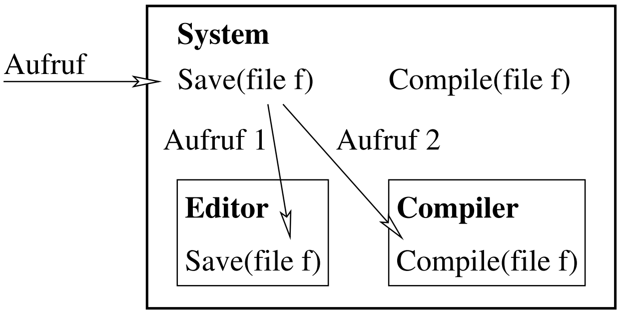


Abbildung 2.6: Integration durch Kapselung

den Methoden vermeiden dies, indem sie das Gesamtsystem als Netz kommunizierender Objekte strukturieren.

2. ***Feste Verdrahtung***
   Bei dieser Methode werden erforderliche Aktionen in anderen Objekten an den notwendigen Stellen explizit aufgerufen. Im Beispiel bedeutet dies, daß der Editor so modifiziert wird, daß er nach dem Speichern einer Datei explizit, z.B. über einen entfernten Prozeduraufruf (RPC, *Remote Procedure Call*) die Übersetzungs-Methode des Compilers aktiviert, wie in Abb. 2.7 dargestellt ist.

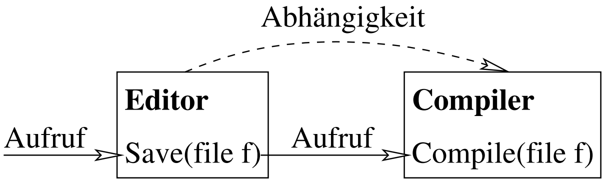


Abbildung 2.7: Integration durch feste Verdrahtung

Die offensichtlichen Probleme dieses Vorgehens sind zum einen, daß der Editor modifiziert werden muß, zum anderen, daß dieser nun eine Abhängigkeit vom Compiler aufweist und damit außerhalb der integrierten Umgebung nicht mehr benutzbar ist.

3. ***Impliziter Aufruf***
   Dieser Ansatz ist eine spiegelbildliche Variante des vorhergehenden. Der Übersetzer registriert hier beim Editor sein Interesse am Auftreten des Ereignisses „Datei geschrieben". Wenn dieses Ereignis gemeldet wird, startet der Compiler den Übersetzungsvorgang (siehe Abb. 2.8).

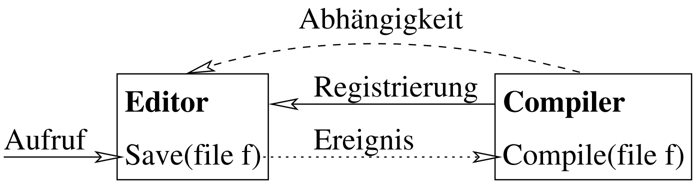


Abbildung 2.8: Integration durch impliziten Aufruf

Anstelle des Editors muß nun der Compiler angepaßt werden, der dadurch vom Editor abhängig wird und nicht mehr ohne diesen lauffähig ist.

4. ***Broadcast-Nachrichtenserver***
   Eine Standardtechnik der Informatik zum Auflösen von Abhängigkeiten ist die Indirektion. Sie ist die Basis dieser Methode: Die zu integrierenden Komponenten kommunizieren nicht direkt miteinander, sondern ausschließlich über einen Broadcast-Nachrichtenserver (BNS). In unserem Beispiel würde der Editor beim BNS einen Nachrichtentyp t registrieren, der vereinbarungsgemäß für das Ereignis „Datei gespeichert" steht. Der Compiler registriert seinerseits beim BNS sein Interesse an Nachrichten des Typs t. Nach dem Sichern einer Datei sendet der Editor durch einen expliziten Methodenaufruf eine Nachricht dieses Typs an den BNS, der diese an alle Interessenten weiterleitet. Empfängt der Compiler eine Nachricht vom Typ t, übersetzt er die in dieser Nachricht genannte Datei neu. Dies ist in Abb. 2.9 veranschaulicht.

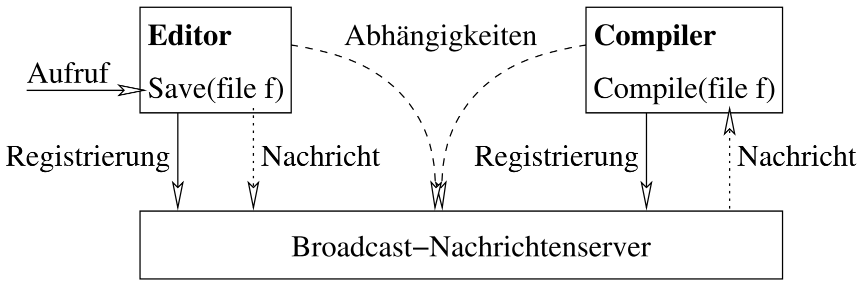


Abbildung 2.9: Integration durch Broadcast-Nachrichtenserver

   Der Vorteil, daß keinerlei Abhängigkeit zwischen den Komponenten mehr besteht, wird durch den Nachteil erkauft, daß nun beide Komponenten verändert werden müssen und vom BNS abhängen, ohne diesen also nicht mehr funktionieren. Als weitere Nachteile stellt Sullivan in [229] heraus, daß die Verhaltensrelationen, die das integrierte Verhalten bestimmen, nicht explizit repräsentiert sind und daß der BNS eine monolithische Komponente bildet, die die Vereinigung der Schnittstellen aller zu integrierenden Objekte darstellt.

5. ***Mediatoren***
   Diese Nachteile werden durch die Methode der Mediatoren vermieden. Ein ***Mediator*** ist ein explizites Objekt, das die Verhaltensbeziehung zweier (oder auch mehrerer) anderer Objekte repräsentiert und steuert. Ein Mediator realisiert Verhaltensregeln in Form von Ereignis/Aktions-Relationen sowie, falls erforderlich, die Übergabe und Anpassung von Parametern des Ereignisses und die auszuführenden Aktionen. Im Beispiel überwacht das Mediatorobjekt den Editor, wie in Abb. 2.10 dargestellt ist. Wird festgestellt, daß dieser eine Datei f sichert, ruft der Mediator die Übersetzungsmethode im Compiler auf, wobei f als Parameter übergeben wird.

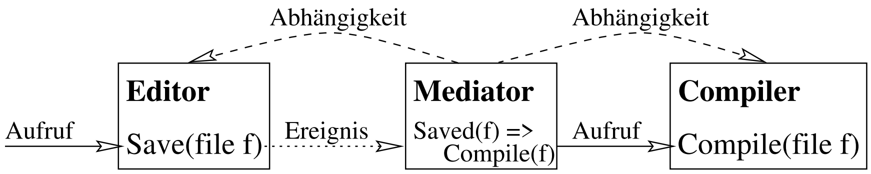


Abbildung 2.10: Integration durch einen Mediator

Voraussetzung für das Funktionieren dieser Methode ist, daß die zu integrierenden Objekte in ihren Schnittstellen neben Methoden (Aktionen) auch Ereignisse, wie z.B. das Sichern einer Datei im Editor, nach außen hin definieren. Sullivan spricht dabei in Anlehnung an abstrakte Datentypen von abstrakten Verhaltenstypen.

Prinzipiell können damit die integrierten Komponenten weiterhin unabhängig voneinander und auch unabhängig vom Mediator verwendet werden. Lediglich der Mediator ist von den kontrollierten Objekten abhängig, dies ist jedoch natürlich und unproblematisch. Abstrakte Verhaltenstypen stellen jedoch zunächst nur eine Designmethode dar. Konkrete Implementierungen erfordern bisher, daß die Ereignisse durch die Objekte explizit über einen Methodenaufruf gemeldet werden [229, 213, 191]. Dies erfordert wieder eine Modifikation des Objekts, die dieses von der verwendeten Kommunikationsinfrastruktur abhängig macht. Wir werden in § 2.3.4.2 sehen, daß Techniken des Online-Monitoring eine Chance bieten, auch dieses Problem noch zu lösen.

Nach dieser Erläuterung der prinzipiellen Methoden zur Verhaltensintegration wollen wir einige existierende Integrations-Infrastrukturen vorstellen.

CORBA. Die *Common Object Request Borker Architecture* CORBA ist eine von der *Object Management Group* (OMG) [170] standardisierte Middleware, die ein verteiltes, objektorientiertes Komponentenmodell realisiert [211, 171, 203]. Die Objekte stellen dabei Dienste in Form von Methoden nach außen hin bereit und können andererseits über Methodenaufrufe Dienste anderer Objekte verwenden. Im ersten Fall spricht man von einem Server-Objekt, im zweiten Fall von einem Klienten-Objekt, wobei diese Rollen dienstspezifisch sind, ein Objekt also auch gleichzeitig beide Rollen innehaben kann. Methodenaufrufe eines Klienten-Objekts werden immer zunächst an den ORB (*Object Request Broker*) geleitet, der sie an das zuständige Server-Objekt weiterleitet. Dadurch werden vor dem Klienten die Verteiltheit des Systems, der Ort des Server-Objekts, seine Implementierung sowie sein Ausführungszustand verborgen und damit Zugriffs-, Orts- und Implementierungstransparenz erreicht.

Neben dem ORB als zentralem Bestandteil gehören zu einer CORBA-Umgebung immer auch zwei Datenverzeichnisse: das *Interface Repository* sowie das *Implementation Repository*. Ersteres enthält Informationen über alle im System bekannten Objektklassen, insbesondere die Signaturen der zur Verfügung gestellten Methoden. Dies erlaubt Objekten, zur Laufzeit Information über andere Objekte einzuholen. Das *Implementation Repository* enthält Information über die tatsächliche Implementierung der Objektklassen und wird vom ORB zur Aktivierung neuer Objekte sowie zur Lokalisierung existierender Objekte verwendet.

CORBA erlaubt eine sprach- und plattformunabhängige Kommunikation zwischen Objekten durch Methodenaufrufe. Dies wird durch eine Schnittstellenbeschreibungssprache (IDL, *Interface Description Language*) sowie einen entsprechenden IDL-Übersetzer erreicht. Bei der Einbindung von neuen Objektklassen in eine CORBA-Umgebung muß deren Schnittstelle mit Hilfe der IDL spezifiziert werden. Der IDL-Übersetzer erzeugt aus dieser Beschreibung die entsprechenden Einträge im *Interface Repository* sowie zwei Code-Rümpfe für Klienten und Server (*Client Stub* bzw. *Server Skeleton*) in der jeweils verwendeten Programmiersprache (siehe Abb. 2.11). Über den *Client Stub* können Klienten Dienste eines entfernten Objekts mit den üblichen Sprachmitteln durch lokale Methodenaufrufe anfordern; er stellt somit ein lokales Stellvertreterobjekt dar. Der *Client Stub* stellt transparent über den ORB die Verbindung mit dem *Server Skeleton* her und übermittelt diesem die Argumente des Methodenaufrufs. Das *Server Skeleton* bildet das Gegenstück auf Serverseite und ruft dort die entsprechende Methode

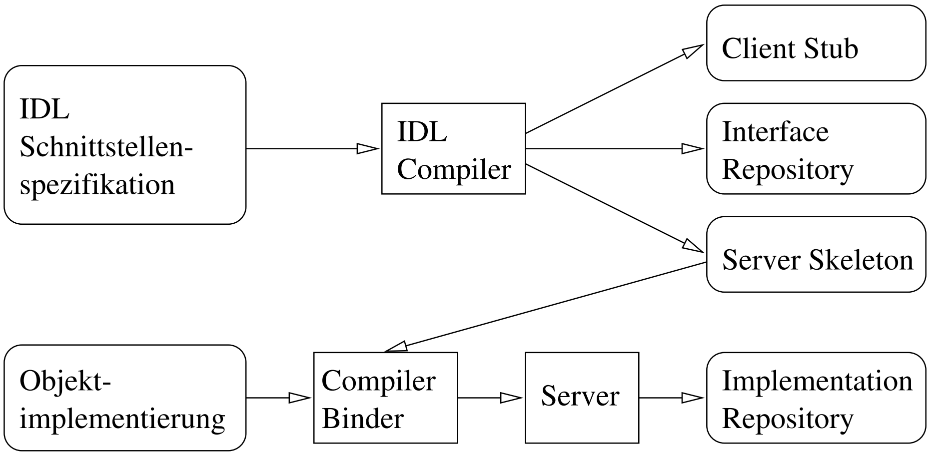


Abbildung 2.11: Erzeugung von Objektimplementierungen und Code-Rümpfen in CORBA

der Objektimplementierung auf. Die beiden Code-Rümpfe übernehmen somit alle notwendigen Sprach- und Datentypanpassungen und kapseln die erforderliche Interprozeßkommunikation.

Falls der Name bzw. die Signatur einer aufzurufenden Methode zur Übersetzungszeit des Klienten noch nicht bekannt sind, was z.B. bei Interpretern regelmäßig der Fall ist, kann der Klient anstelle des *Stubs* die dynamische Aufrufschnittstelle von CORBA verwenden. Die benötigte Schnittstelleninformation wird in diesem Fall zur Laufzeit dem *Interface Repository* entnommen, was jedoch einen zusätzlichen Laufzeitaufwand bedingt. Ein analoges Vorgehen ist auch auf Serverseite möglich.

Neben dieser Kommunikationsinfrastruktur definiert CORBA auch zahlreiche eigene Dienste, von denen an dieser Stelle vor allem der Ereignisdienst relevant ist. Er erlaubt eine anonyme, asynchrone Benachrichtigung anderer Objekte über das Auftreten von Ereignissen. Jedes Ereignis muß dabei vom auslösenden Objekt über den Aufruf einer Methode eines Ereigniskanals bekanntgemacht werden. Ein Ereigniskanal ist ein CORBA-Objekt, das die registrierten Empfänger verwaltet und die Ereignisnachricht durch den Aufruf einer speziellen, von allen Empfängern zu implementierenden Methode an diese weiterleitet. Von der Funktion her entspricht der Ereignisdienst also den zu Beginn des Abschnitts eingeführten Broadcast-Nachrichtenservern. Eine Implementierung des Mediator-Modells als CORBA-Dienst wurde von Paul und Sattler [191] vorgenommen.

DCOM. DCOM (*Distributed Component Object Model*) ist ein von Microsoft definiertes Komponentenmodell, das auf dem ursprünglich nur für MS-Windows entwickelten COM beruht, inzwischen aber auch für andere Betriebssysteme verfügbar ist [200, 155, 203]. Von der grundsätzlichen Funktion her ist DCOM mit CORBA vergleichbar, allerdings ist DCOM eine Spezifikation auf Binärcode-Basis und bietet nicht wie CORBA eine Anzahl zusätzlicher Unterstützungsdienste.

Eine DCOM-Komponente ist ein gekapseltes Objekt, das – im Gegensatz zu einem CORBA-Objekt – mehrere Schnittstellen besitzen kann. Jedes Objekt muß dabei eine Schnittstelle mit dem Namen `IUnknown` anbieten, deren Methoden es erlauben, Information über alle weiteren Schnittstellen des Objekts abzufragen. Es existiert also kein zentrales *Interface Repository*, das alle verfügbaren Schnittstellen dokumentiert. Durch das Zulassen mehrerer Schnittstellen pro Objekt können in einfacher Weise Objekte durch Kapselung zu größeren Objekten zusammen-

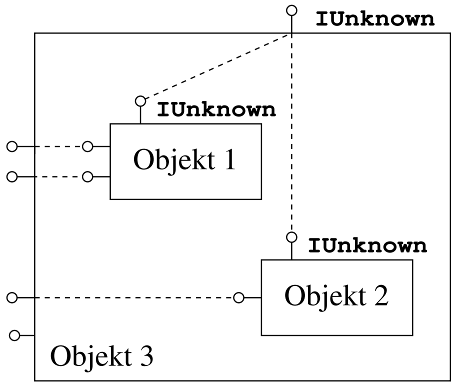


Abbildung 2.12: Kapselung von Objekten in DCOM (nach [208])

gefaßt werden, wie dies in Abb. 2.12 veranschaulicht ist.

Ebenso wie CORBA verwendet auch DCOM eine Schnittstellenbeschreibungssprache um Sprach- und Plattformunabhängigkeit zu erreichen. Der zugehörige Übersetzer erzeugt wieder *Stubs* und *Skeletons* für Klienten und Server.

DCOM dient als Grundlage weiterer Komponentenarchitekturen für spezielle Anwendungsbereiche, etwa ActiveX für Web-Anwendungen oder OLE für Verbunddokumente.

ToolTalk. ToolTalk ist ein von Sun Microsystems entwickeltes System, das die netzwerkweite Kommunikation zwischen unabhängigen Anwendungsprozessen ermöglicht [109, 231]. Es unterstützt asynchronen, anonymen Nachrichtenaustausch, d.h. die Kommunikationspartner müssen einander nicht explizit bekannt sein. Neben Prozessen werden als Kommunikationspartner auch Objekte unterstützt, wobei vom Programmierer explizit definiert werden muß, durch welchen Prozeß ein Objekt implementiert wird. In der nachfolgenden Beschreibung beschränken wir uns auf Prozesse, sie gilt aber analog auch für objektbasierte Kommunikation.

Nach der zu Beginn des Abschnitts beschriebenen Klassifikation ist ToolTalk den Broadcast-Nachrichtenservern zuzuordnen. Ein Prozeß versendet eine Nachricht durch den expliziten, nichtblockierenden Aufruf einer ToolTalk-Funktion. Die Nachricht besteht dabei aus einer Menge von Attributen. Neben dem eigentlichen Inhalt, der analog einem Funktions- bzw. Methodenaufruf in einen Operationsnamen und die zu übergebenden, typisierten Argumente strukturiert ist, sind die zwei wichtigsten Attribute die Nachrichtenklasse und die Adresse. Die Nachrichtenklasse definiert, ob die Nachricht eine Anfrage darstellt, auf die eine Antwort erwartet wird, oder lediglich eine Notiz, d.h. eine Ereignismeldung ist. Die Adresse legt fest, ob die Nachricht an einen bestimmten Prozeß zugestellt werden soll oder an beliebige Prozesse, die ein Interesse an Nachrichten dieses Typs registriert haben.

ToolTalk besitzt eine Datenbasis mit Mustern über diesen Attributen, die definiert, welche Prozesse an welchen Nachrichten interessiert sind. In der Datenbasis ist auch spezifiziert, ob der Prozeß die Nachricht lediglich beobachten will oder ob er sie bearbeitet, also gegebenenfalls eine Antwort zurücksendet. Die Einträge in der Datenbasis können statisch aus einer Beschreibung des Prozeßtyps generiert werden, vergleichbar der IDL-Beschreibung von CORBA-Objekten. Zusätzlich können sie auch vom jeweiligen Prozeß dynamisch zur Laufzeit definiert werden.

Erhält ToolTalk eine Nachricht, die nicht bereits an einen festen Prozeß adressiert ist, wird ein Vergleich zwischen den Nachrichtenattributen und den in der Datenbasis gespeicherten Mustern durchgeführt. Eine Notiz wird dann an alle Beobachter sowie maximal einen Bearbeiter gesandt, für die der Mustervergleich positiv ausfiel. Anfragen werden ebenso behandelt, jedoch sendet in diesem Fall der Bearbeiter der Nachricht eine Antwort zurück. Diese wird von ToolTalk an den ursprünglichen Sender sowie alle Beobachter mit passenden Mustern weitergeleitet.

Technisch erfolgt die Benachrichtigung der Empfänger über einen UNIX-Dateideskriptor. Der Empfänger muß selbst dafür sorgen, daß beim Eintreffen von Daten die ToolTalk-Empfangsfunktion aufgerufen wird, die entweder die Nachricht als Funktionsergebnis zurückliefert oder sie gegebenenfalls an vordefinierte Rückruffunktionen verteilt.

Wird für eine Anfrage in der Datenbasis kein passender, laufender Prozeß zur Bearbeitung gefunden, sucht ToolTalk nach der Definition eines Prozeßtyps, der ein auf die Nachricht passendes Muster aufweist. In dieser Definition ist dann das weitere Vorgehen festgelegt: Die Nachricht wird entweder in einer Warteschlange gespeichert, oder ein neuer Prozeß dieses Typs wird gestartet, dem die Nachricht zugestellt wird. Findet sich kein passender Prozeßtyp, wird ein Fehler an den Sender gemeldet.

PCTE. Die bereits in § 2.2.3.1 vorgestellte Umgebung PCTE beinhaltet auch einige Mechanismen zur Verhaltensintegration. Zur Kommunikation zwischen Werkzeugen sind Nachrichtenwarteschlangen vorgesehen, die Objekte im Sinne von PCTE darstellen. Für diese sind Sende- und Empfangsmethoden definiert, die eine bytestrom-orientierte Kommunikation auf relativ niedrigem Abstraktionsniveau erlauben. Sender und Empfänger müssen sich dabei selbst um das Ein- und Auspacken der Nachricht sowie eventuell notwendige Datentypkonvertierungen kümmern. Es werden blockierendes und nicht-blockierendes Senden und Empfangen sowie asynchroner Empfang über Nachrichtenbehandlungsfunktionen zur Verfügung gestellt.

Eine bemerkenswerte Eigenschaft von PCTE ist die Möglichkeit einer automatischen Benachrichtigung bei Objektzugriffen. Ein Prozeß kann veranlassen, daß bei Änderung, Umbenennung oder Löschung eines spezifizierten Objekts durch das System eine Nachricht an eine seiner Warteschlangen gesandt wird. Der Nachrichtentyp codiert dabei die Art des Zugriffs. Weitere Information über den Zugriff, wie z.B. der zugreifende Prozeß oder die veränderten Objektkomponenten sind allerdings nicht standardisiert verfügbar, da PCTE den genauen Inhalt der Nachricht offen läßt. Immerhin kann zumindest das betroffene Objekt indirekt durch Einrichten von objektspezifischen Nachrichtenwarteschlangen ermittelt werden.

Schließlich bietet PCTE auch Dienste zur Prozeßkontrolle auf Betriebssystemebene an. Diese vereinheitlichen die verschiedenen Betriebssystemschnittstellen, wie `ptrace` oder `/proc` und stellen sie netzwerktransparent zur Verfügung. Wie die genannten Betriebssystemdienste erlaubt PCTE u.a. das Erzeugen, Anhalten und Weiterführen von Prozessen, das Ändern von Prozeßattributen, das Setzen von Haltepunkten im Maschinencode sowie den Zugriff auf den Adreßraum anderer Prozesse.

Koordinationssprachen. Im weitesten Sinne läßt sich auch die Erstellung eines parallelen Programms aus sequentiellen Einzelbausteinen als ein Akt der Verhaltensintegration beschreiben. Bereits Anfang der 90er Jahre wurde vorgeschlagen, parallele Programmiermodelle in ein Berechnungsmodell und ein Koordinationsmodell zu partitionieren, wobei ersteres ausschließlich der Erstellung der sequentiellen Komponenten, letzteres ausschließlich deren Koordination dient. Gelernter und Carriero diskutieren in [84] die Vor- und Nachteile dieses Ansatzes. Seitdem ist eine Vielzahl von Koordinationssprachen entstanden, von denen einige, nämlich Linda, Gamma und Manifold in [1] beschrieben werden. Weitere Ansätze sind in den Tagungsbänden der Konferenz *Coordination Languages and Models* zu finden [50, 89, 58].

Als ein Beispiel einer Koordinationssprache sei hier kurz Manifold [5, 3] vorgestellt, das bereits viele Eigenschaften aufweist, die für die im nachfolgenden Unterabschnitt beschriebenen Konfigurationssprachen typisch sind. Manifold unterstützt neben der parallelen Programmierung auch die Koordination unabhängiger Prozesse in verteilten Systemen [3]. Es basiert auf einem idealisierten Arbeiter/Manager-Modell, dessen wesentliche Konzepte die Trennung von Berechnung (Arbeiter) und Koordination (Manager) sowie die anonyme Kommunikation sind. Die Kommunikation zwischen Prozessen erfolgt über unidirektionale *Ports*. Die Verbindungen zwischen diesen *Ports* werden ausschließlich von Koordinationsprozessen hergestellt, so daß die Kommunikationspartner sich nicht explizit zu kennen brauchen. Die Koordinationsprozesse werden in der Sprache Manifold programmiert und realisieren ereignisgesteuerte Zustandsübergangssysteme. Bei Eintreffen eines Ereignisses führt der Prozeß einen Block von Aktionen aus und geht in einen neuen Zustand über. Eine Aktion kann der Aufruf einer in Manifold oder C geschriebenen Funktion sein, das Erzeugen eines neuen Prozesses, das Schalten bzw. Auftrennen einer Verbindung zwischen *Ports* sowie das Melden eines neuen Ereignisses. Manifold erinnert damit an das Mediator-Modell.

Als Beispiel eines in Manifold beschriebenen Koordinationsprozesses ist in Abb. 2.13 die Definition eines dinierenden Philosophen dargestellt. Im Zustand `begin` denkt der Philosoph zunächst, holt sich dann eine Essensmarke, signalisiert die beiden Ereignisse `request` und `ready` und wartet dann auf eintreffende Ereignisse. Trifft sowohl von der linken als auch von der rechten Gabel das Ereignis `ready` ein, geht der Prozeß in den nächsten Zustand über, und der Philosoph beginnt zu essen. Nach dem Essen, im Zustand `end` signalisiert der Prozeß das Ereignis `done`, gibt die Essensmarke zurück und beginnt von neuem.

```
export Philosopher()
{
  event ready.

  begin: while true do {
           begin: Think(self);
                  getTicket();
                  (raise(request, ready), WAIT).

           ready.*lfork & ready.*rfork: Eat(self, lfork, rfork).

           end: raise(done);
                returnTicket().
         }.
}
```

Abbildung 2.13: Beispiel eines Manifold-Codes (aus [1])

Konfigurationssprachen. Koordinationssprachen, wie sie soeben vorgestellt wurden, sind für die parallele Programmierung vorgesehen. Ihr Ziel ist die bessere Wiederverwendbarkeit der sequentiellen Berechnungen und des Koordinationsteils. Von den Ausdrucksmöglichkeiten her sind Konfigurationssprachen sehr ähnlich, sie verfolgen jedoch andere Zielsetzungen.

Sie unterstützen in erster Linie die verteilte Programmierung, bei der existierende Software-Komponenten gekoppelt werden müssen, die nicht von vorneherein für eine solche Zusammenarbeit entworfen wurden. Wir wollen im folgenden drei Beispiele für derartige Sprachen vorstellen. Weitere Vertreter werden in [208] diskutiert.

a) ToolBus. ToolBus [29] weist konzeptuell eine starke Ähnlichkeit mit Manifold auf, verfolgt jedoch wie erwähnt andere Ziele und ist daher auch gänzlich anders implementiert. Die hinter ToolBus stehende Metapher ist die eines Software-Busses, an den analog zu einem Hardware-Bus verschiedene Komponenten – hier Software-Werkzeuge – über eine definierte Schnittstelle angeschlossen werden können.

Diese Schnittstelle regelt die folgenden Interaktionen zwischen den Werkzeugen und dem Bus: Den Empfang von Nachrichten (mit und ohne Antwortwunsch) vom Bus durch ein Werkzeug, das Senden von Antworten durch ein Werkzeug an den Bus, sowie die Meldung von Ereignissen. Wie auch bei Manifold ist also keine direkte Kommunikation zwischen Werkzeugen möglich, vielmehr werden Nachrichten durch den ToolBus an die Empfänger weitergereicht. Der ToolBus ist dabei durch ein Netz von (Koordinations-)Prozessen realisiert, deren Verhalten jeweils durch ein Skript gesteuert wird.

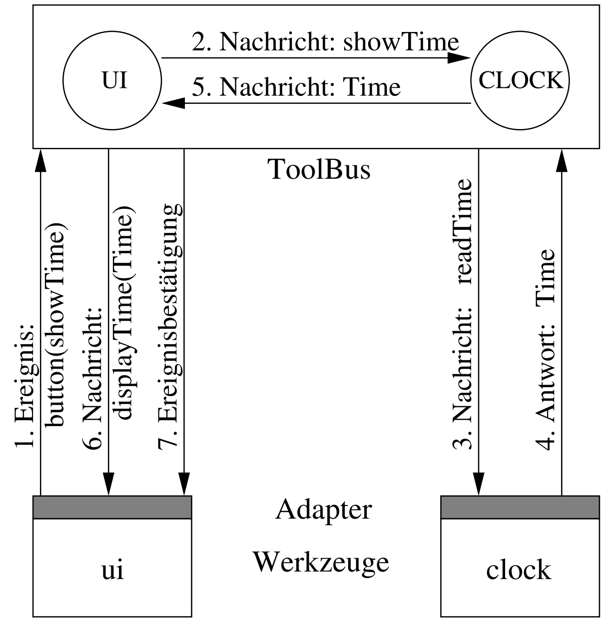
Die Skriptsprache basiert auf einer Term-Algebra, die unter anderem die folgenden Primitive unterstützt:

1. Blockierender Empfang einer Nachricht von einem Werkzeug,
2. blockierender Empfang einer Nachricht von einem anderen ToolBus-Prozeß,
3. blockierendes Warten auf ein Ereignis,
4. Erzeugen eines neuen ToolBus-Prozesses,
5. Senden einer Nachricht an ein Werkzeug,
6. Senden einer Nachricht an einen anderen ToolBus-Prozeß.

Neben der sequentiellen Komposition von Anweisungen werden auch Iteration und Alternative unterstützt. Letzteres erlaubt z.B. das gleichzeitige Warten auf zwei verschiedene Ereignisse. Die Skriptsprache ermöglicht damit das beliebige Zusammenschalten der Schnittstellen der an den ToolBus angeschlossenen Werkzeuge. Mangels Operationen auf Daten kann mit ihrer Hilfe jedoch keine Anpassung der Daten an den Schnittstellen vorgenommen werden.

Abb. 2.14 zeigt ein Beispiel für die Anwendung von ToolBus, bei dem ein Werkzeug (ui), das eine Benutzerschnittstelle realisiert, mit einer Uhrenkomponente (clock) verschaltet ist. Die Nummern in der Graphik und den Skripten zeigen den Ablauf der Kommunikation an: Wird an der Schnittstelle der Knopf `showTime` gedrückt, so soll von der Uhrenkomponente die aktuelle Zeit erfragt und an die Benutzerschnittstelle ein Kommando zur Anzeige dieser Zeit geschickt werden. In dem Beispiel ist jeweils ein ToolBus-Prozeß für ein Werkzeug zuständig, was von der ToolBus-Architektur aber nicht vorausgesetzt wird.

b) CHAIMS. CHAIMS ist ein in Entwicklung befindliches System der Universität Stanford zur „Megaprogrammierung" [193, 37, 222]. Darunter wird die Integration großer, verteilter Software-Komponenten („Megamodule") verstanden. Diese stellen unabhängig verwaltete Berechnungsserver dar, auf deren Methoden bereits über verteilte Protokolle wir CORBA, DCOM oder RMI zugegriffen werden kann, wobei für die verschiedenen Megamodule jedoch kein einheitliches Protokoll vorausgesetzt wird.



ToolBus Skript

```
   define UI =
1.  ( rec-event(ui, button(showTime))
2.     . snd-msg(showTime)
5.     . recv-msg(showTime, Time)
6.     . snd-do(ui, displayTime(Time))
7.     . snd-ack-event(ui)
    ) * delta


   define CLOCK =
2.  ( rec-msg(showTime)
3.     . snd-eval(clock, readTime)
4.     . rec-value(clock, Time)
5.     . snd-msg(showTime, Time)
    ) * delta
```

Abbildung 2.14: Beispiel einer ToolBus Anwendung

CHAIMS definiert eine Konfigurationssprache mit deren Hilfe solche Module zu einem größeren Programm verbunden werden können, indem der notwendige Datenaustausch sowie einfache Kontrollstrukturen programmiert werden. Das Konfigurationsprogramm spielt dabei die Rolle eines Klienten, das alle betroffenen Megamodule als Server anspricht. Bei der Übersetzung des Konfigurationsprogramms erzeugt der CHAIMS-Compiler automatisch Code zur Kommunikation mit Hilfe desjenigen Protokolls, das der jeweils angesprochene Server voraussetzt. Derzeit werden CORBA/C++ und RMI/Java unterstützt.

Als Kommunikationsmethode verwendet CHAIMS asynchrone RPCs, die mit Hilfe dreier Primitive programmiert werden: `invoke` ruft die spezifizierte Methode mit den übergebenen Parametern im Server auf und kehrt sofort zurück. Mittels `examine` kann festgestellt werden, wann der Aufruf beendet ist. Das Ergebnis wird dann mit dem Sprachkonstrukt `extract` ermittelt. Darüber hinaus stellt CHAIMS Sprachmittel zur Ablaufkontrolle in Form von Schleifen und bedingten Anweisungen bereit. Weitere Primitive erlauben das Einrichten einer Verbindung zu einem Megamodul, das Lesen und Schreiben von Modul-Parametern sowie die Abschätzung der Laufzeit eines Methodenaufrufs. Letzteres dient der Programmoptimierung. Geplant sind eine Optimierung der Ausführungsreihenfolge von RPC-Primitiven sowie eine Optimierung des Datenflusses durch direkte Übertragung vom Erzeuger zum Verbraucher ohne Umweg über das Konfigurationsprogramm [37].

CHAIMS wurde ursprünglich als reine Konfigurationssprache entworfen, die die Ein-/Ausgabedaten von Modulen nicht interpretieren kann. Dies machte jedoch eine Ablaufsteuerung in Abhängigkeit des Ergebnisses von Berechnungen unmöglich, weshalb vor kurzem die Sprache so erweitert wurde, daß zumindest Boolesche Ergebnisse ausgewertet werden können [223].

c) TIL. In einer an der Universität Magdeburg durchgeführten Dissertation definiert Sattler ein formales Komponentenmodell, das die Werkzeug-Komposition in integrierten Entwurfswerkzeugen für Ingenieuranwendungen unterstützt [208, 192]. Zusätzlich zu der nach außen hin von einer Komponente zur Verfügung gestellten Schnittstelle führt Sattler auch Schnittstellen für diejenigen Methoden ein, die die Komponente von ihrer Umgebung benötigt. Ereignismeldungen durch eine Komponente werden als Spezialfall dieser Schnittstelle modelliert. Eine Komponente kann damit wie in Abb. 2.15 veranschaulicht werden.

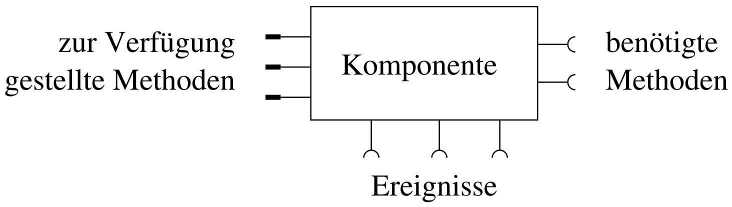


Abbildung 2.15: Eine einfache Komponente

Zur Komposition solcher Komponenten wird eine Erweiterung der Mediator-Idee verwendet. Während ein Mediator nur die Ereignisse einer Komponente mit den von anderen zur Verfügung gestellten Methoden verbindet, erfüllt ein ***Interaktor*** diese Aufgabe allgemein für alle von der Komponente benötigten Dienste. Dies ist in Abb. 2.16 dargestellt. Ein Interaktor legt somit fest, was beim Aufruf einer externen Methode[3] passieren soll. Der Interaktor kann den Aufruf z.B. direkt weiterleiten, ihn an mehrere Komponenten verteilen oder ihn durch eine Sequenz von Methodenaufrufen realisieren. Interaktoren können interne Zustände besitzen, was u.a. die Synchronisation mehrerer angeschlossener Komponenten ermöglicht. Ebenso kann ein Interaktor Änderungen an den übergebenen Aufrufparametern vornehmen und so auch an sich inkompatible Schnittstellen einander anpassen. Eine formale Definition von Interaktoren findet sich in [208].

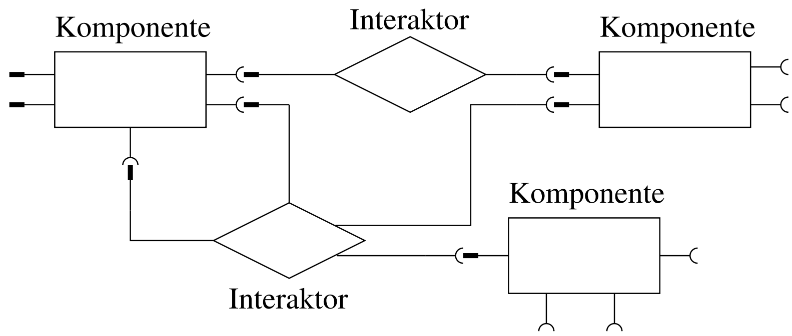


Abbildung 2.16: Komposition von Komponenten mit Hilfe von Interaktoren

Auf der Basis dieses Modells wurde eine Integrationsinfrastruktur sowie die Konfigurationssprache TIL (*Tool Integration Language*) entwickelt. Die vollständige Beschreibung einer Werkzeugumgebung besteht dabei aus drei Teilen:

1. **Schnittstellen**
   Die Beschreibung der einzelnen Schnittstellen erfolgt mittels einer erweiterten CORBA-IDL. Im Gegensatz zu CORBA werden neben den exportierten auch die importierten Schnittstellen sowie die Ereignisschnittstellen definiert.

2. **Komponenten und Interaktoren**
   Eine Komponente wird durch die von ihr zur Verfügung gestellten und benötigten Schnitt-

[3] Dies schließt Ereignismeldungen mit ein.

stellen sowie durch die Angabe ihrer Implementierung beschrieben. Im Sinne der objektorientierten Programmierung definiert diese Beschreibung also eine Klasse.

Die Definition eines Interaktors ähnelt einer Prozedurdefinition. Die formalen Parameter des Interaktors sind die zu verbindenden Komponenten. Der Rumpf der Definition kann einem Methodenaufruf einer Komponenten direkt eine von der Signatur her passende Methode einer anderen Komponente zuordnen. Alternativ kann für Methodenaufrufe oder Ereignismeldungen auch eine operationale Beschreibung der auszuführenden Aktionen angegeben werden. Da TIL im Gegensatz zu CHAIMS die Datentypen der Schnittstellen kennt, können Aufrufparameter sowie Ergebnisse von Methoden problemlos weiterverarbeitet und z.B. auch für die Ablaufsteuerung verwendet werden.

3. **Konfiguration**
   Hier werden schließlich die benötigten Komponenten und Interaktoren instantiiert, wobei den Interaktoren die Instanzen der zu verbindenden Komponenten als Aktualparameter übergeben werden.

2.2.3.3 Zusammenfassung

Wir wollen zum Abschluß kurz die relevanten Eigenschaften der besprochenen Ansätze zur Daten- und Verhaltensintegration zusammenfassen. Wir werden an dieser Stelle noch keine Bewertung vornehmen, sondern diese auf das Ende des nächsten Abschnitts verschieben, in dem wir interoperable Laufzeit-Werkzeuge genauer betrachten. Dies wird die generellen Anforderungen besser verdeutlichen, die wir an eine unterstützende Infrastruktur stellen müssen.

Die existierenden Methoden zur Datenintegration basieren auf der Idee eines gemeinsamen Datenbestandes oder einer Datenbank, auf die alle Komponenten über eine gemeinsame Schnittstelle gleichzeitig zugreifen können. Da alle Zugriffe über diese Schnittstelle erfolgen, kann sie Mechanismen zur Konsistenzsicherung realisieren, etwa in Form von Sperrprotokollen oder Benachrichtigungen bei Datenmodifikationen. Die Schnittstelle regelt jedoch nur syntaktische Aspekte der Datenstrukturen und der Zugriffe; Wissen über die eigentliche Semantik der Daten enthält sie nicht, um eine universelle Verwendbarkeit zu ermöglichen.

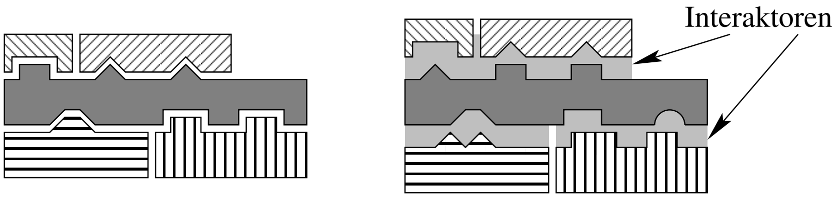
Einige wichtige Eigenschaften der zur Verhaltensintegration verwendbaren, diskutierten Infrastrukturen sind in Tabelle 2.1 zusammengefaßt. Als allgemeinster Ansatz stellt sich dabei der von TIL gewählte heraus, wobei viele Aspekte auch den anderen Infrastrukturen gemeinsam sind. Eine Software-Komponente wird dabei als Baustein mit drei Arten von Koppelpunkten aufgefaßt, die bereitgestellte Dienste, benötigte Dienste und erzeugte Ereignisse repräsentieren. Die Verbindung zwischen den Bausteinen erfolgt indirekt über Interaktoren. Dadurch wird es möglich, daß die Komponenten selbst nichts voneinander wissen müssen. Zudem wird eine Anpassung von nicht unmittelbar kompatiblen Schnittstellen ermöglicht. Um eine Analogie zu gebrauchen, werden die Bausteine also nicht wie Lego-Steine zusammengesteckt, was nur bei exakt zusammenpassenden Schnittstellen möglich ist, sondern vielmehr mit Hilfe von Klebstoff verbunden (siehe Abb. 2.17).

Die Schnittstellen der Komponenten werden in Form einer IDL-Datei beschrieben, aus der ein spezieller Compiler *Stubs* und *Skeletons* erzeugt. Sie passen die programmiersprachlichen Schnittstellen der Komponente an die in der Implementierung der Infrastruktur verwendeten Kommunikationsschnittstellen an. Dieser Anpassungscode muß jeweils zu jeder Komponente

	CORBA	DCOM	Tool-Talk	PCTE	Manifold	Toolbus	CHAIMS	TIL
Ansatz[1]	V/BNS	V	BNS	V	M	M	M	M
Unterstützung für Ereignisse	+	−	+	∘	+	+	−	+
IDL + Compiler f. *Stubs* und *Skeletons*	+	+	−	−	−[2]	−[3]	+	+
Programmierbare Komp.verknüpfung	−	−	−	−	+	+	+	+
– Schnittstellen-anpassung mögl.	n.a.	n.a.	n.a.	n.a.	∘[4]	∘[4]	∘[4]	+
– Synchronisation möglich	n.a.	n.a.	n.a.	n.a.	+	∘[5]	+	+

[1] V = feste Verdrahtung, BNS = Broadcast-Nachrichtenserver, M = Mediator/Interaktor.
[2] Schnittstellen (*Ports*) müssen Manifold- bzw. Betriebssystemkonventionen genügen [5].
[3] Notwendige Adapter müssen handcodiert werden.
[4] Konfigurationssprache kann keine Operationen auf Daten ausführen.
[5] Konfigurationssprache erlaubt keine explizite Darstellung von Zuständen.

Tabelle 2.1: Vergleich von Mechanismen zur Verhaltensintegration



a) direkte Komposition b) Komposition über Interaktoren

Abbildung 2.17: Direkte und indirekte Komposition von Komponenten

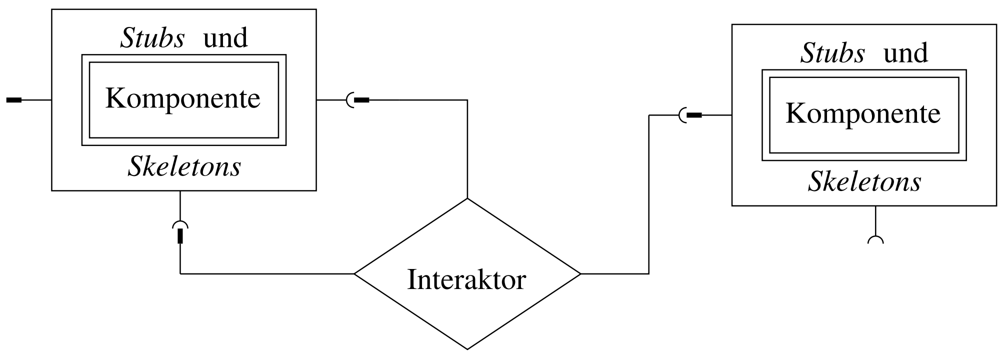


Abbildung 2.18: Ein vollständiges Komponentensystem

gebunden werden. So entsteht eine speziell für die jeweilige Infrastruktur gekapselte Komponente[4]. Die Struktur des so entstehenden Gesamtsystems ist in Abb. 2.18 veranschaulicht.

Viele der vorgestellten Infrastrukturen unterstützen Ereignisse, auf die Komponenten optional reagieren können. Die Behandlung von Ereignissen erfolgt dabei stets asynchron, d.h. die auslösende Komponente setzt die Ausführung unmittelbar nach der Ereignismeldung fort, unabhängig von der jeweiligen Behandlung.

2.3 Interoperable Werkzeuge und integrierte Werkzeugumgebungen

In den vorangegangenen Abschnitten wurden die Funktionen und der Aufbau von Einzelwerkzeugen zur Laufzeituntersuchung paralleler Programme sowie verschiedene Methoden zur Unterstützung von Interoperabilität, also der Zusammenarbeit verschiedener Software-Komponenten, untersucht. Im folgenden wollen wir diese beiden bisher getrennt behandelten Gebiete zusammenführen. Wir werden zunächst die grundlegenden Ziele und Anforderungen interoperabler Laufzeit-Werkzeuge definieren, dann einen Blick auf existierende Umgebungen werden und schließlich die derzeitigen Probleme und deren Ursachen analysieren. Dabei zeigt sich, daß nicht nur Werkzeuge von den in § 2.2.3 diskutierten Methoden der Interoperabilität profitieren, sondern auch umgekehrt Techniken der Werkzeuge, insbesondere dynamische Instrumentierung, zur Unterstützung von Interoperabilität eingesetzt werden können.

2.3.1 Ziele und Anforderungen

Wie wir in § 2.1.2 bereits gesehen haben, existiert eine Vielzahl von Laufzeit-Werkzeugen, die einzelne Aspekte der späten Phasen des Software-Entwurfs, d.h. der Implementierung, der Integration, der Nutzung und der Wartung, unterstützen. Aus der Sicht eines Benutzers dieser Werkzeuge ergibt sich der Wunsch einer Werkzeugumgebung, die alle genannten Aspekte *durchgängig* unterstützt. Für die frühen Entwurfsphasen ist die Notwendigkeit solcher durchgängigen Umgebungen bereits allgemein anerkannt [2, 12] und führte z.B. zu dem auf europäischer Ebene geförderten Projekt PCTE [233]. Für die genannten späten Phasen bedeutet dies vor allem, daß eine solche Umgebung die – möglichst auch gleichzeitige – Überwachung und Steuerung *aller* Aspekte eines Programmablaufs in kohärenter Weise erlauben muß. Dadurch wird es dem Programmierer möglich, einfacher an bessere Laufzeitinformation zu gelangen. Beispielsweise kann das globale Kommunikationsverhalten einer Anwendung, das üblicherweise von einem Visualisierer dargestellt wird, gleichzeitig mit dem lokalen Verhalten einzelner Prozesse untersucht werden, über das ein Debugger Auskunft gibt. Zudem bewirkt die simultane Überwachung und Steuerung verschiedener Aspekte des Laufzeitverhaltens oft einen darüber hinaus gehenden Zusatznutzen. Einige Beispiele hierfür sind:

- Die Nutzung von Sicherungspunkten während des Debuggings kann zu deutlich kürzeren Debugging-Zyklen führen [235].
- Werkzeuge zur deterministischen Ausführung erlauben reproduzierbare Debugging-Zyklen.

[4]Eine Ausnahme bildet hier CHAIMS, das voraussetzt, daß die Komponenten bereits eine Komponentenschnittstelle für CORBA, DCOM oder RMI besitzt und lediglich für die Interaktoren Code zur Anpassung an das jeweilige Protokoll erzeugt.

- Funktionen zum Anhalten und Fortführen von Prozessen, wie sie ein Debugger anbietet, können genutzt werden, um Leistungsanalyse auf bestimmte Phasen des Programmablaufs einzuschränken.

Eine aus Benutzersicht wichtige Anforderung an eine solche Umgebung ist Modularität in dem Sinne, daß die Umgebung vom Benutzer aufgrund seiner jeweiligen, momentanen Anforderungen nach dem Baukastenprinzip zusammengestellt werden kann. Da dann nur die jeweils aktuell benötigten Werkzeuge in der Umgebung vorhanden sind, wird eine unnötige Komplexität der Benutzungsoberfläche sowie eine unnötig hohe Beeinflussung des überwachten Programms vermieden. Dieses Zusammenstellen sollte möglichst dynamisch erfolgen können. Ideal wäre es, für jeden interessierenden Aspekt des Laufzeitverhaltens einfach ein Werkzeug starten zu können, wobei die Werkzeuge dann automatisch zusammenarbeiten.

Die Forderung nach Modularität beinhaltet auch die Austauschbarkeit von Komponenten einer Werkzeugumgebung. Da die Einarbeitungszeit für viele Laufzeit-Werkzeuge verhältnismäßig hoch ist, sind Benutzer nur zögerlich bereit, ihnen unbekannte Werkzeuge zu verwenden. Die Möglichkeit, ein Werkzeug durch ein äquivalentes, dem Anwender bereits bekanntes auszutauschen sollte daher von der Umgebung unterstützt werden. In der Regel wird dies dazu führen, daß Werkzeuge verschiedener Hersteller, die unabhängig voneinander entwickelt wurden, in der Umgebung zusammenarbeiten müssen. Insbesondere folgt daraus, daß die Entwickler der Werkzeugumgebung nicht mit denen der Einzelwerkzeuge identisch sein müssen.

Aus der Sicht eines Werkzeugentwicklers stehen deutlich verschiedene Anforderungen im Vordergrund. Wie in § 2.1.3 gezeigt wurde, sind Laufzeit-Werkzeuge komplexe Softwaresysteme, die insbesondere sehr viele Abhängigkeiten zu den jeweils anvisierten Zielplattformen aufweisen. Im Bereich der parallelen Systeme ändern sich die Zielplattformen derzeit immer noch sehr schnell, da sich für Hardware und Programmiermodelle noch keine allgemein akzeptierten Standards herausgebildet haben. Da der Markt für Werkzeuge zur parallelen Programmierung ohnehin relativ klein ist, können Anbieter solcher Werkzeuge nur überleben, wenn sie sich auf ein Werkzeug fokussieren und dieses so portabel wie irgend möglich gestalten. Als Beispiele seien hier `totalview` von Etnus [68] und `Vampir` von Pallas [178] genannt. Die generelle Komplexität der Werkzeuge, verbunden mit der Notwendigkeit der Portabilität führt dazu, daß sie oft nur einen eingeschränkten Funktionsumfang anbieten können. So zeigte z.B. `totalview` bis zur Einführung einer portablen Schnittstelle zu MPI [48] keinerlei Information über den Nachrichtenaustausch zwischen Prozessen an, da dies zu einer zusätzlichen Abhängigkeit von der jeweiligen MPI-Implementierung geführt hätte. `Vampir` erlaubt nur eine Offline-Leistungsanalyse der MPI-Kommunikation. Eine detailliertere Analyse des Quellcodes ist nur durch manuelle Quellcode-Instrumentierung möglich, da eine automatische Objektcode-Instrumentierung oder gar eine dynamische Instrumentierung erheblich mehr Entwicklungsaufwand erfordert und weniger portabel ist. Neben der reinen Portabilität sind auch ein modularer Entwurf und die Wiederverwendbarkeit von Komponenten entscheidend, um die Entwicklungskosten der Werkzeuge zu senken.

Sowohl die Entwickler- als auch die Benutzeranforderungen können durch zwei ineinandergreifende Maßnahmen umgesetzt werden:

- Die Möglichkeit einer Interoperabilität zwischen Laufzeit-Werkzeugen erlaubt die einfache Realisierung von Werkzeugumgebungen auf der Basis vorhandener Werkzeuge. Dadurch wird es möglich, die bisherigen, relativ komplexen und monolithischen Werkzeuge in kleine, modulare Werkzeuge aufzuspalten, die einfacher zu programmieren und auch einfacher zu bedienen sind. Ebenso können auf diese Weise Zielsystemabhängigkeiten ausfaktorisiert werden. Ein paralleler Debugger wie `totalview` kann beispielsweise in einen reinen Mehrprozeß-Debugger und ein Werkzeug zur Untersuchung der Kommunikation, in der Art von `MQM` [181] aufgespalten werden.

- Die Benutzung einheitlicher Monitorsysteme und -schnittstellen gestattet eine Wiederverwendung von Instrumentierungstechniken. So könnte z.B. `Vampir` eine derartige Monitorschnittstelle für die Objektcode-Instrumentierung verwenden, ohne dadurch an Portabilität zu verlieren. Für einen Anwender ergibt sich durch einheitliche Monitorsysteme der Vorteil, daß nicht länger eine unterschiedliche Vorbehandlung der zu untersuchenden Programme erforderlich wird.

2.3.2 Stand der Technik

Wir wollen zunächst die Situation bei Einzelwerkzeugen untersuchen (siehe auch [140, 142, 265]). Praktisch alle der in § 2.1.2 angesprochenen Werkzeuge verwenden eigene, proprietäre Monitorsysteme. Diese verwenden meist eine statische Instrumentierung, die während der Übersetzung oder des Bindens in das Zielprogramm eingebracht wird (siehe § 2.1.3.2). Für die Untersuchung durch ein anderes Werkzeug muß das Zielprogramm daher neu übersetzt bzw. neu gebunden werden. Dadurch können unterschiedliche Werkzeuge nicht einmal nacheinander auf dieselbe Version des ausführbaren Codes angewandt werden. Ein weiterer Grund, der eine gleichzeitige Verwendung mehrere Laufzeit-Werkzeuge oft verhindert, sind Betriebssystemschnittstellen zur Prozeßüberwachung, z.B. `ptrace` (siehe § 2.1.3.2), die jeweils nur exklusiv von einem Prozeß genutzt werden können.

Selbst wenn zwei Werkzeuge gleichzeitig auf dasselbe Zielprogramm gestartet werden können, entstehen durch die durchgeführten Instrumentierungen i.a. Konflikte, die ein korrektes Funktionieren der Werkzeuge verhindern. Trinitis [235] unterscheidet zwischen strukturellen Konflikten und logischen Konflikten. ***Strukturelle Konflikte*** werden durch inkompatible Monitorsysteme oder die Verwendung exklusiver Betriebssystemschnittstellen verursacht. Diese Art von Konflikten wird also bereits durch die Monitorinfrastrukturen alleine bewirkt, unabhängig von den eigentlichen Aktionen der Werkzeuge. ***Logische Konflikte*** entstehen dagegen durch Aktionen der Werkzeuge, die den Zustand des Zielsystems in einer Weise ändern, die ein korrektes Weiterfunktionieren der anderen Werkzeuge verhindert. Hält beispielsweise ein Debugger einen Zielprozeß an, so sind die Zeitmessungen eines gleichzeitig verwendeten Leistungsanalysewerkzeugs nicht mehr korrekt, da dieses die Möglichkeit einer derartigen Aktion nicht berücksichtigt.

Solange Werkzeuge nicht konfliktfrei simultan benutzbar sind, ist an Synergieeffekte durch eine Zusammenarbeit dieser Werkzeugen nicht einmal zu denken. Existierende Einzelwerkzeuge bieten zudem auch keine externen Schnittstellen für eine mögliche Verhaltensintegration an.

Um die derzeitige Situation im Bereich integrierter Werkzeugumgebungen zu verdeutlichen, werden in den nächsten Unterabschnitten einige Beispiele für derartige Umgebungen betrachtet[5]. Wir werden dabei feststellen, daß in keiner dieser Umgebungen die Laufzeit-Werkzeuge

[5] Da in den Veröffentlichungen über Werkzeugumgebungen Integrations- bzw. Interoperabilitätsaspekte nicht oder bestenfalls nur recht oberflächlich beschrieben werden, basieren die Darstellungen zum Teil auf persönlichen Aussagen der Entwickler:

- Brian Wylie, Univ. of Wisconsin, Madison, USA
- Gabor Dozsa und Norbert Podhorszki, MTA SZTAKI, Budapest, Ungarn

die in § 2.2.2 aufgestellte Definition der Interoperabilität erfüllen – es gibt derzeit noch keine in diesem Sinne interoperablen Laufzeit-Werkzeuge. Dabei sind die vorgestellten Umgebungen selbst noch Forschungsprototypen, abgesehen von `Annai`, das von NEC zur Kommerzialisierung übernommen wurde.

2.3.2.1 Annai

`Annai` [43, 42, 44, 45] ist eine am *Centro Svizzero di Calcolo Scientifico* CSCS [46] in Manno (Schweiz) in Kooperation mit der Firma NEC entstandene Werkzeugumgebung für die parallele Programmierung, die sowohl HPF [122, 201] als auch MPI [63] unterstützt. Die Umgebung besteht aus fünf Komponenten:

1. **PST** (*Parallelization Support Tool*) ist ein Parallelisierungswerkzeug, das grob als erweiterter HPF-Compiler beschrieben werden kann.

2. **PDT** (*Parallel Debugging Tool*) ist ein quellsprachorientierter Debugger. Er unterstützt HPF durch globale Haltepunkte und die Visualisierung verteilter Arrays. Für Programme, die auf explizitem Nachrichtenaustausch über MPI basieren, ist eine Erkennung von Nachrichten-*Races* sowie eine deterministische Wiederausführung durch *Replay* [165] vorgesehen.

3. **PMA** (*Performance Monitoring and Analysis*) unterstützt die Leistungsanalyse durch die Erfassung und Darstellung von *Profiling*-Daten sowie ereignisspur-basierter Daten. Die durchgeführten Instrumentierungen des Zielprogramms und damit die gemessenen Metriken können vom Benutzer dynamisch zur Laufzeit geändert werden.

4. **UI** (*User Interface*) ist die gemeinsame Benutzungsoberfläche aller Werkzeuge. Sie besteht aus einem Quellcode- und Programmstruktur-Browser und kann sowohl von PDT als auch von PMA gesteuert werden, um z.B. Quellcodeannotationen vorzunehmen.

5. **TSA** (*Tool Services Agent*) stellt nach der in § 2.1.3.1 gegebenen Definition einen Teil des Monitorsystems der Umgebung dar. Es basiert auf dem GNU Debugger `gdb` [224] und erlaubt die Kontrolle der Zielprozesse sowie das An- und Ausschalten von Instrumentierungen. Daneben existiert eine in [42] als Monitor bezeichnete Komponente, die über spezielle, zum Zielprogramm gebundene Laufzeitbibliotheken [44] größere Datenmengen wie Ereignisspuren und Arrayinhalte aus den überwachten Prozessen ausliest und den Werkzeugen zur Verfügung stellt. Diese beiden Module realisieren somit die Datenintegration in `Annai`.

Alle genannten Komponenten sind als getrennte UNIX Prozesse realisiert. Die Kommunikationsverbindungen zwischen diesen Prozessen sind in Abb. 2.19 dargestellt, wobei die dickeren Pfeile die größeren Datenflüsse, die dünneren die zur Steuerung, d.h. Verhaltensintegration dienende Kommunikation markieren. Man erkennt, daß PST eine unabhängige Komponente darstellt, deren Verbindung zum Rest der Umgebung lediglich über die erzeugten ausführbaren Binärdateien mit der darin enthaltenen Zusatzinformation über Symbole und Programmstruktur

---

- Dieter Kranzlmüller, GUP Linz, Österreich
- Barbara Chapman, Univ. of Southampton, UK

Ihnen sei an dieser Stelle herzlich für ihre Hilfe gedankt!

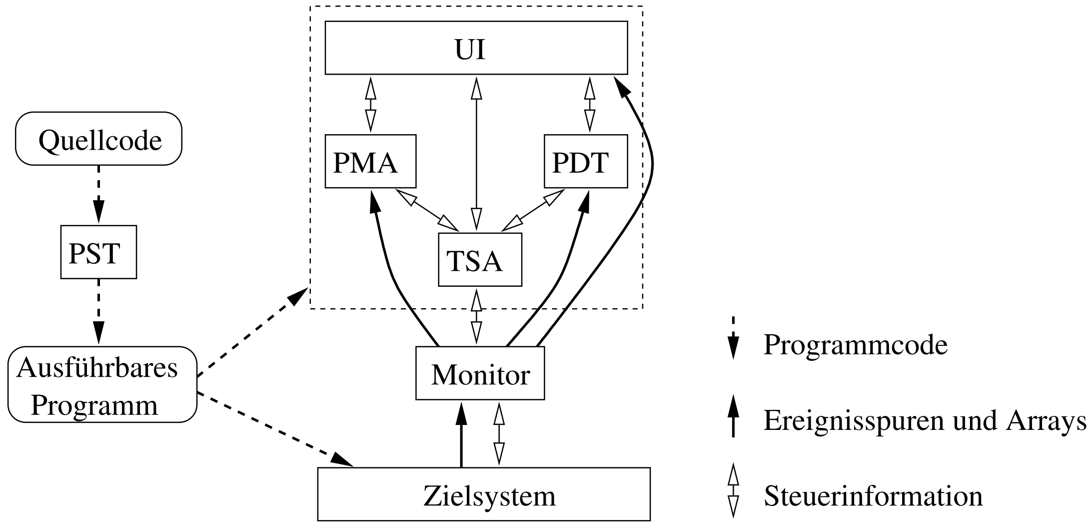


Abbildung 2.19: Komponentenstruktur von `Annai`

besteht. Die anderen Komponenten kommunizierten ursprünglich über UNIX-*Pipes* und einfache, proprietäre ASCII-Protokolle [42]. Inzwischen werden jedoch RPCs zur Kommunikation verwendet. In jedem Fall wird die Verhaltensintegration durch feste Verdrahtung der Komponenten erreicht. Aus Abb. 2.19 ist ersichtlich, daß die beiden Laufzeit-Werkzeuge PDT und PMA ausschließlich mit der Oberfläche UI und dem Monitorsystem TSA interagieren, jedoch nicht untereinander. Daraus läßt sich schließen, daß die beiden Werkzeuge nicht interoperieren. Von einem `Annai`-Entwickler wurde bestätigt, daß sie tatsächlich nicht gleichzeitig anwendbar sind.

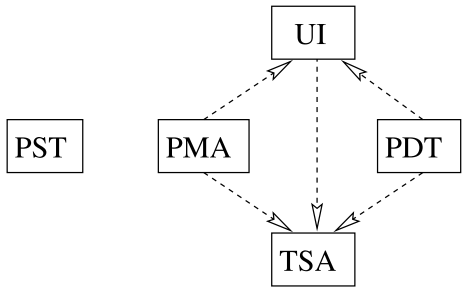


Abbildung 2.20: Abhängigkeit der Komponenten in `Annai`

Der Abhängigkeitsgraph in Abb. 2.20 zeigt, daß zudem nur wenige Komponenten der Umgebung unabhängig voneinander arbeiten können. Im Sinne der im vorigen Abschnitt genannten Anforderungen wäre auch eine stärkere Modularisierung wünschenswert, um etwa *Race*-Erkennung und deterministische Wiederausführung ohne Debugger nutzen zu können, da diese Funktionalitäten unabhängig und orthogonal zueinander sind.

2.3.2.2 GRADE

Am MTA SZTAKI Forschungsinstitut [159] in Budapest (Ungarn) wird die Programmierumgebung `GRADE` zur graphischen Programmierung paralleler Systeme entwickelt [114, 115, 158]. In `GRADE` werden die Kommunikation zwischen Prozessen sowie die Kontrollstrukturen, die diese Kommunikation steuern, graphisch spezifiziert. Das System erzeugt aus dieser Spezifikation automatisch C-Code, der sich wahlweise auf die Kommunikationsbibliotheken PVM oder MPI abstützt.

Die Programmierumgebung besteht aus vier für den Anwender sichtbaren Komponenten:

1. Dem graphischen Editor GRED, mit dessen Hilfe die Prozeßstruktur sowie die Ablaufstruktur innerhalb von Prozessen definiert werden kann. Sequentielle Berechnungen werden dabei als Blöcke mit textuellem C-Code eingebunden.

2. Dem Compiliersystem, das die graphische Spezifikation in C-Code umsetzt und mit Hilfe eines Standard-C-Compilers ein ausführbares Programm erzeugt.

3. Dem Debugger DIWIDE, der den Ablauf des erzeugten Programms auf der Ebene der graphischen Darstellung und auch der sequentiellen Codeblöcke überwachen und steuern kann.

4. Dem Visualisierer PROVE, der ein Prozeß-Zeit-Diagramm aller Kommunikationsereignisse darstellt. Eine Verbindung zwischen dieser Darstellung und der graphischen Spezifikation wird erreicht, indem bei Selektion eines Ereignisses in PROVE die dazugehörigen Kommunikationsaufrufe von GRED hervorgehoben werden (*click back*), bzw. auch umgekehrt.

Wie bei Annai sind auch hier die Komponenten durch eigenständige UNIX-Prozesse realisiert, die über *Sockets* und *Pipes* unter Verwendung eines eigenen Protokolls kommunizieren. Die Struktur der Umgebung ist in Abb. 2.21 dargestellt.

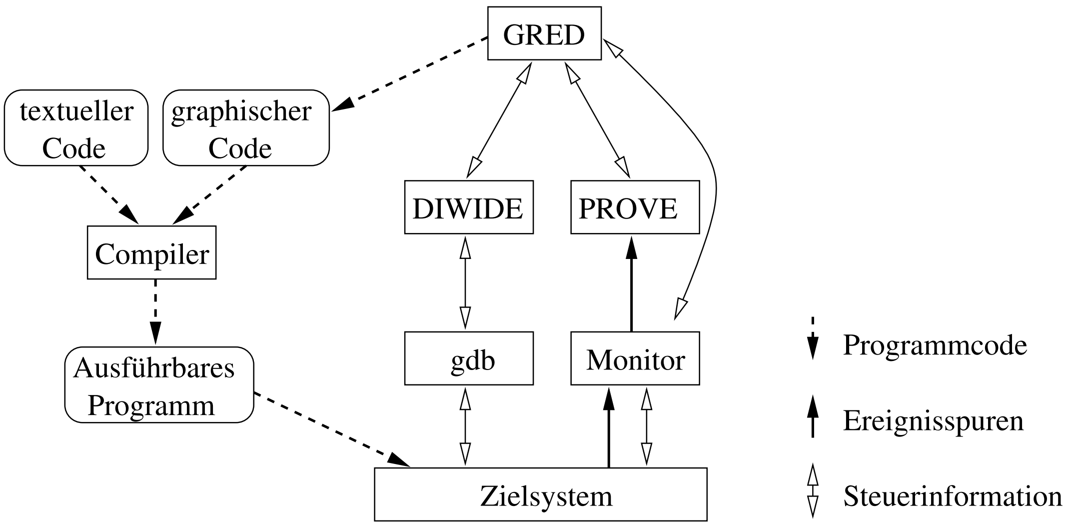


Abbildung 2.21: Komponentenstruktur von GRADE

DIWIDE besitzt keine eigene Benutzungsoberfläche, ist also völlig von GRED abhängig und nicht außerhalb der Umgebung nutzbar. Allerdings besitzt der Debugger eine definierte, wenn auch derzeit nicht dokumentierte Schnittstelle, die alle Funktionen in Form einer C-Bibliothek bereitstellt. Als Monitorsystem zur Kontrolle der Zielprozesse benutzt auch DIWIDE den GNU Debugger gdb.

Der Visualisierer ist loser an die Umgebung gekoppelt und kann als Offline-Werkzeug auch einzeln verwendet werden. In diesem Fall stellt PROVE fest, daß der normalerweise von GRED erzeugte *Socket* nicht existiert und versucht keine Kommunikation. Für den Online-Betrieb benutzt PROVE ein eigenes verteiltes Monitorsystem, das eine feste Klasse von Ereignissen überwacht. Da dieses Monitorsystem von GRED gestartet wird, der auch den für die Kommunikation zu benutzenden *Socket* an PROVE weitergibt, kann der Visualisierer als Online-Werkzeug nicht eigenständig genutzt werden.

Abb. 2.21 zeigt, daß zwischen den beiden Laufzeit-Werkzeugen von GRADE wegen der getrennten Monitorsysteme keine Datenintegration gegeben ist, ebenso wurde keine Verhaltensintegration durchgeführt. Beides wird von den Entwicklern jedoch geplant (siehe [121, 113]). Im Rahmen des Förderprojekts GRADE-OMIS wurde angestrebt, OMIS (siehe § 3.2.1) als gemeinsame Monitorschnittstelle für beide Werkzeuge zu verwenden. Innerhalb des vorgegebenen Zeitrahmens konnte jedoch nur PROVE entsprechend angepaßt werden [249, 250].

2.3.2.3 MAD

MAD ist eine Werkzeugumgebung, die an der Abteilung für graphische und parallele Datenverarbeitung der Johannes Kepler Universität Linz, Österreich [110] entwickelt wird [119, 111]. Sie unterstützt das Debugging und die Visualisierung der Kommunikation in parallelen Programmen. Insbesondere können Nachrichten-*Races* erkannt und deren Resultate manipuliert werden. MAD beinhaltet derzeit acht Werkzeuge [111]:

- Das Monitorsystem EMU zeichnet Kommunikationsereignisse auf und stellt sie in Form von Spurdateien zur Verfügung.

- ATEMPT visualisiert diese Spurdateien in Form eines Prozeß-Zeit-Diagramms [117]. Dabei werden Nachrichten-*Races* farblich hervorgehoben. Der Benutzer kann nun interaktiv die Empfangsreihenfolge der an diesem *Race* beteiligten Nachrichten verändern.

- Die derart manipulierte Spurdatei dient als Eingabe für das *Replay*-Werkzeug PARASIT, das nachfolgende Programmabläufe so steuert, daß die in der Spurdatei angegebene Reihenfolge der Ereignisse eingehalten wird. Somit kann der Einfluß von *Races* auf das Programmverhalten untersucht werden.

- Die zwei Werkzeuge FB und CDFA dienen der Darstellung des Quellcodes, zum einen in textueller Form, zum anderen als Struktogramm, das den Kontrollfluß visualisiert. CDFA kann zusätzlich auch noch eine Datenflußanalyse vornehmen, also für eine Variable die möglichen Stellen der letzten Wertzuweisung ermitteln.

- MUCH visualisiert Hardware-Aspekte des Zielsystems, insbesondere die Netzwerktopologie.

- Schließlich kann der Konvertierer TCON die von EMU erzeugten Ablaufspuren in andere Formate wandeln, so daß diese auch mit dem Werkzeug paragraph [92, 93] untersucht werden können.

Im Gegensatz zu den bisher vorgestellten Werkzeugumgebungen ist MAD von Anfang an so konzipiert worden, daß alle Werkzeuge auch einzeln, unabhängig von den anderen verwendet werden können. Als Umgebung werden sie von einer zentralen Benutzungsoberfläche gesteuert, die mit Hilfe des objektorientierten Anwendungs-Frameworks ET++ [253, 252] realisiert wurde.

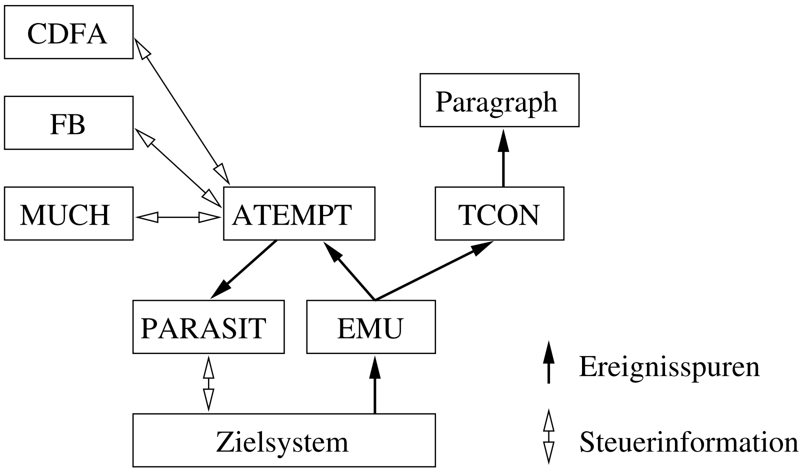


Abbildung 2.22: Komponentenstruktur von MAD

Das sogenannte *Interapplication Framework* in ET++ erlaubt die Verhaltensintegration durch direkte Kommunikation zwischen den Werkzeugen. Jeder Prozeß muß sich dazu mit Namen beim ET++-Laufzeitsystem anmelden. Nachrichten, bestehend aus einem Kommando mit Parametern in Form eines ASCII-Strings, können dann entweder an bestimmte Prozesse oder an alle verschickt werden. Im ersten Fall sind sowohl synchrones Senden mit Warten auf ein Resultat als auch asynchrones Senden möglich. Beim Eintreffen einer Nachricht wird vom ET++-Laufzeitsystem eine vom Programmierer zur Verfügung zu stellende *Dispatch*-Methode aufgerufen, die das in der Nachricht spezifizierte Kommando dekodiert und entweder eine entsprechende Methode zur Bearbeitung aufruft oder die Nachricht ignoriert.

In MAD wird diese Kommunikation eingesetzt, um die Anzeigen der vier Werkzeuge ATEMPT, FB, CDFA und MUCH konsistent zu halten. Bei der Auswahl eines Kommunikationsereignisses in ATEMPT wird von FB der entsprechende Quellcode angezeigt. CDFA zeigt für beide beteiligten Prozesse die jeweiligen Blöcke des Struktogramms, in denen die Kommunikation stattfindet. MUCH visualisiert schließlich das Kommunikationsereignis aus Hardware-Sicht.

Die Datenintegration in bezug auf Laufzeitdaten erfolgt durch das Monitorsystem EMU, das Ereignisspuren für alle Werkzeuge erzeugt. Diese werden entweder in Dateien geschrieben oder über UNIX-*Sockets* direkt an die Werkzeuge weitergereicht. Bemerkenswert ist, daß EMU und PARASIT nicht gleichzeitig verwendet werden können, eine Beobachtung des von PARASIT kontrollierten Programmablaufs mit EMU also nicht möglich ist. In letzter Zeit wurden daher diese beiden Komponenten zu einem universelleren Monitorsystem NOPE zusammengefaßt [127, 126].

2.3.2.4 PREPARE

Im Rahmen des ESPRIT Projekts PREPARE wurde eine Umgebung zur interaktiven Analyse und Übersetzung von HPF-Programmen entwickelt [245]. Obwohl sie nur ein einziges Laufzeit-Werkzeug beinhaltet, soll sie kurz vorgestellt werden, da sie einen Prototyp einer Werkzeugumgebung mit unabhängig voneinander entwickelten Komponenten darstellt und als einziges einen der in § 2.2.3 beschriebenen Integrationsmechanismen, nämlich ToolTalk verwendet.

Die eigentliche Übersetzungsumgebung besteht aus etwa einem Dutzend Module, die über die Verwendung einer gemeinsamen Zwischendarstellung des zu übersetzenden Programms integriert sind. An diese ist relativ lose ein Werkzeug zur Online-Leistungsanalyse angeschlossen [125]. Aus Benutzersicht besteht die Umgebung aus fünf interaktiven Komponenten. Das *Welcome Tool* stellt den Einstiegspunkt dar. Es startet und beendet über ToolTalk-Nachrichten den *Package Manager*, der die eigentliche Benutzungsoberfläche des Compilers ist und ggf. wieder über ToolTalk einen Quellcode-Browser, einen Aufrufgraph-Visualisierer und das Leistungsanalysewerkzeug `EM` startet. Diese drei Komponenten sind nur sehr lose über ToolTalk gekoppelt. Im wesentlichen ist eine *Click-Back*-Funktion realisiert, die z.B. bei Auswahl einer Prozedur im Aufrufgraph diese auch im Quellcode-Browser und in der Auswahlliste des Messungsdefinitionsfensters von `EM` selektiert, bzw. umgekehrt.

2.3.2.5 PPPE

Im Rahmen des ESPRIT Projekts PPPE (*Portable Parallel Programming Environment*) wurde ein Satz von Werkzeugen zur parallelen Programmierung entwickelt [239]. Dazu gehören die Programmierbibliothek PARMACS [176], verschiedene Compiler und *Source-To-Source*-Übersetzer (HPF Mapper der Firma NA Software [160], ADAPTOR [85] und der Vienna-Fortran-Übersetzer [238], sowie ein Konvertierer zwischen PARMACS und MPI), ein Leistungsvorhersagewerkzeug sowie vier Laufzeit-Werkzeuge:

- Der Debugger `totalview` [69],
- der HPF Debugger von NA Software [160],
- TraceGen [176] und `paragraph+`, eine Weiterentwicklung von `paragraph` [87],
- das *Profiling*-Werkzeug Weight Finder [71].

Die Umgebung ist aber lediglich in dem Sinne integriert, als Fortran als gemeinsame Quellsprache unterstützt und PARMACS als gemeinsame Kommunikationsbibliothek verwendet wird [87]. Eine weitergehende Integration der statischen Daten, wie Quellcodes, Profile und Ereignisspuren mit Hilfe von PCTE war zwar ursprünglich geplant, wurde aber im Verlauf des Projekts verworfen [51]. Die Gründe dafür waren zum einen, daß die Verwendung von PCTE bei den Benutzern den Import aller existierenden Daten in das PCTE-System erforderlich gemacht hätte, was einen nicht unerheblichen Aufwand verursacht. Zum anderen wurde von den Herstellern befürchtet, daß durch die angestrebte enge Integration Abhängigkeiten zu den Werkzeugen anderer Hersteller entstehen. Eine Interoperabilität der Laufzeit-Werkzeuge im Sinne von § 2.2.2 wurde in PPPE nicht vorgesehen.

2.3.2.6 FIRST

Mitte 1997 startete in der Schweiz das Projekt FIRST [187, 124, 188] als Kooperation der Universität Basel und dem *Centro Svizzero di Calcolo Scientifico* CSCS. Ziel dieses noch laufenden Projekts ist die Bereitstellung eines Rahmenwerks für interoperable Laufzeit-Werkzeuge. Es stützt sich dabei auf CORBA (siehe § 2.2.3.2, Seite 33) als Kommunikations-Middleware und `dyninst` (siehe § 2.1.3.2, Seite 23) als Basis für die Programm-Instrumentierung.

FIRST definiert ein Architekturmodell für Werkzeugumgebungen, das aus drei Schichten besteht und in Abb. 2.23 skizziert ist [124]. Die Monitor-Schicht beinhaltet einen Prozeßmanager auf jedem Knoten des Zielsystems, der die folgenden Dienste über eine CORBA-Schnittstelle zur Verfügung stellt:

- Ankoppeln an existierende Prozesse und Erzeugung neuer Prozesse,

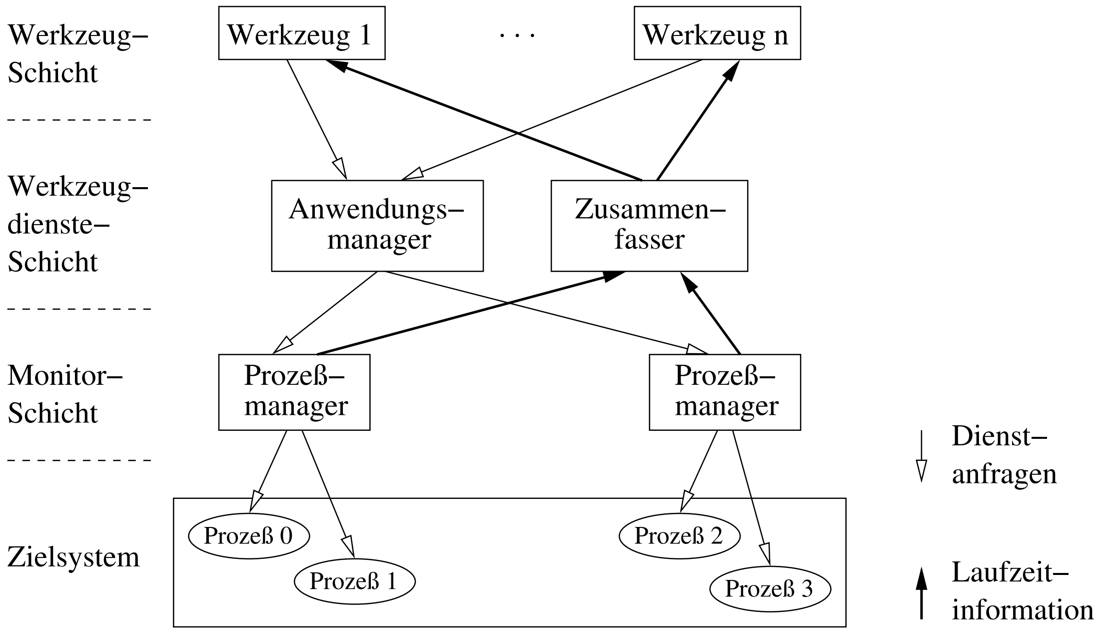


Abbildung 2.23: Architekturmodell von `FIRST`

- Anhalten und Weiterführen von Prozessen,
- Bereitstellung von Information über Codestruktur von Prozessen sowie über mögliche Instrumentierungspunkte,
- Instrumentierung mit Hilfe von `dyninst`,
- Aufsammeln der über die Instrumentierung gewonnenen Laufzeitdaten.

Die Werkzeugdienste-Schicht bietet in gewisser Anlehnung an die OMIS-Schnittstelle (siehe § 3.2.1) höhere Dienste, die von den Werkzeugen genutzt werden. Dabei wird zwischen Informations-, Manipulations- und Benachrichtigungsdiensten unterschieden. *Informationsdienste* geben Auskunft über statische und dynamische Größen des untersuchten Programms, z.B. die Codestruktur oder die Inhalte von Programmvariablen. *Manipulationsdienste* fügen vordefinierte Instrumentierungen in Zielprozesse ein. Sie können Zähler und Stoppuhren realisieren, Ereignisspuren aufzeichnen und Benachrichtigungen auslösen. Ein steuernder Eingriff in Programmabläufe, der über das temporäre Anhalten von Prozessen hinausgeht, wird durch diese Dienste derzeit aber offenbar nicht ermöglicht. *Benachrichtigungsdienste* leiten schließlich Ereignisse durch CORBA-Methodenaufrufe an Werkzeuge weiter.

Die Werkzeug-Schicht beinhaltet schließlich eine Reihe relativ einfacher Werkzeuge, die auf den genannten Diensten aufbauen:

- `OCB` ist ein Objektcode-Browser, der die Codestruktur eines Prozesses darstellt, insbesondere die im Code vorhandenen und instrumentierbaren Funktionen,
- `1st_top` listet die Funktionen eines Zielprozesses mit der höchsten Rechenzeit auf,
- `1st_trace` druckt eine Ereignisspur aller Funktionsaufrufe aus,
- `1st_prof` erstellt einen Prozeduraufrufgraph, der mit Ausführungszeiten und Aufrufhäufigkeiten annotiert ist,
- `1st_cov` ist ein auf Funktionsebene arbeitendes Testüberdeckungswerkzeug,
- `1st_debug` realisiert schließlich einen einfachen Debugger.

FIRST sieht vor, daß Werkzeuge direkt miteinander interagieren können, um eine Verhaltensintegration zu realisieren. Diese Interaktionen müssen jedoch außerhalb des Rahmenwerks, z.B. durch direkte Verwendung von CORBA erfolgen. Ob derartige Interaktionen zwischen den genannten Werkzeugen realisiert wurden, ist unklar. In [188] werden zwar Beispiele für die notwendigen Interaktionen zwischen einem Debugger und einem Visualisierer genannt, jedoch keine Aussagen zu einer möglichen Implementierung gemacht. Durch das Rahmenwerk unterstützt wird jedoch die Koordination von Anfragen. Werkzeuge können gemeinsame Prozesse überwachen, wobei bereits vorhandene Instrumentierungen gegebenenfalls durch die Werkzeugdienste-Schicht automatisch wiederverwendet werden. Ebenso ist das Sperren von Ressourcen für einzelne Werkzeuge möglich. Genauere Details über die Realisierung von FIRST wurden bisher noch nicht veröffentlicht, so daß nicht nachvollziehbar ist, ob neben strukturellen auch logische Konflikte durch das Rahmenwerk vermieden bzw. behandelt werden können.

Das Projekt FIRST stellt derzeit die der vorliegenden Arbeit am nächsten kommende Entwicklung dar, verfolgt jedoch eine etwas andere Zielsetzung. Es will einen Baukasten für modulare Werkzeugumgebungen zur Verfügung stellen, wobei implizit vorausgesetzt wird, daß die Einzelwerkzeuge *gemeinsam* entwickelt werden. Notwendige Interaktionen zwischen den Werkzeugen müssen explizit in diese einprogrammiert werden. Damit wird der in § 2.2.2 hervorgehobene Aspekt der Unabhängigkeit in FIRST – jedenfalls zur Zeit – nicht berücksichtigt.

2.3.2.7 Zusammenfassung

Die beschriebenen Beispiele zeigen, daß existierende Werkzeugumgebungen meist nur relativ wenige Laufzeit-Werkzeuge umfassen. In einigen Fällen ist zwar eine Datenintegration dieser Werkzeuge in dem Sinne erfolgt, daß ein gemeinsames Monitorsystem zur Datenerfassung verwendet wird, sie ist jedoch mit Ausnahmen von FIRST nicht so weit fortgeschritten, daß eine gleichzeitige Anwendung verschiedener Werkzeuge möglich wäre. Wie im Beispiel von `Annai` deutlich wird, führt dies dazu, daß gleichzeitig benötigte Funktionalitäten in großen, monolithischen Komponenten realisiert werden müssen, was Wartbarkeit und auch Portierbarkeit der Umgebung erschwert.

Eine Verhaltensintegration wird meist über feste Verdrahtung und ad-hoc Protokolle realisiert. Dies hat zur Konsequenz, daß die Werkzeuge durch die gegenseitigen Abhängigkeiten nicht mehr unabhängig voneinander verwendet werden können und auch eine Erweiterung erschwert wird. Ausnahmen stellen hier die PREPARE-Umgebung dar, die den Broadcast-Nachrichtenserver ToolTalk verwendet, sowie `MAD`, das auf `ET++` aufsetzt, welches eine ähnliche, aber eingeschränktere Funktionalität zur Verfügung stellt. Zwischen den Laufzeit-Werkzeugen ist bisher in keiner der Umgebungen eine Verhaltensintegration durchgeführt worden. Das FIRST-Projekt ist noch nicht weit genug fortgeschritten, so daß die Untersuchung von Interoperabilitäts-Szenarien lediglich als künftige Arbeit geplant ist [188]. In den anderen Umgebungen sind die Werkzeuge ohnehin nicht gleichzeitig anwendbar.

Tabelle 2.2 stellt einige relevante Eigenschaften der betrachteten Werkzeugumgebungen zusammen. Die Anforderungen, die zu Beginn dieses Abschnitts herausgearbeitet wurden, werden von den bisherigen Werkzeugumgebungen damit nur sehr unzureichend erfüllt.

2.3.3 Probleme

Die vorangegangene Diskussion von Einzelwerkzeugen und Werkzeugumgebungen hat gezeigt, daß eine Interoperabilität zwischen Laufzeit-Werkzeugen bisher meist schon an inkompati-

	Anzahl Laufzeit-Werkzeuge	gemeinsames Monitorsystem	gleichzeitige Nutzung	unabhängige Nutzung	Verhaltensintegration zwischen Laufzeit-Werkzeugen	mit den anderen Werkzeugen[3]
		der Laufzeit-Werkzeuge				
`Annai`	2	+	–	–	–	V
`GRADE`	2	–	–	∘[2]	–	V
`MAD`	2	–/+[1]	–/+[1]	+	–	V/BNS
PREPARE	1	n.a.	n.a.	n.a.	–	BNS
PPPE	4	–	–	+	–	n.a.
FIRST	5	+	+	+	?[4]	?[4]

[1] + gilt für die auf `NOPE` basierende Version.
[2] Nur `PROVE` ist offline unabhängig verwendbar.
[3] V = feste Verdrahtung, BNS = Broadcast-Nachrichtenserver.
[4] Aus der Literatur geht nicht hervor, ob eine Verhaltensintegration der Werkzeuge durchgeführt wurde.

Tabelle 2.2: Eigenschaften der diskutierten Werkzeugumgebungen

blen Monitorsystemen scheitert. Im Falle von `Annai` ist zwar ein gemeinsames Monitorsystem vorhanden, dieses kann jedoch nur exklusiv von jeweils einem Werkzeug genutzt werden. Eine Grundvoraussetzung für interoperable Werkzeuge ist daher ein gemeinsames „mehrbenutzerfähiges“ Monitorsystem. Im Sinne von § 2.2.2 stellt dieses die Datenintegration sicher. Wir werden diesen Aspekt in § 2.3.4.1 weiter diskutieren.

Zur Verhaltensintegration werden in integrierten Umgebungen meist ad-hoc Protokolle eingesetzt, was in diesen Fällen möglich ist, da alle Komponenten innerhalb eines einzigen Teams entwickelt werden. Die Möglichkeit einer Verhaltensintegration mit anderen, externen Werkzeugen ist weder in Werkzeugumgebungen noch bei Einzelwerkzeugen vorgesehen. Neben der generellen Schwierigkeit der gemeinsamen Benutzung durch inkompatible Monitorsysteme hat dies auch noch andere Gründe:

- Ein einzelnes Werkzeug kann prinzipiell nicht wissen, mit welchen anderen Werkzeugen es ein Benutzer später gemeinsam anwenden will. Daher ist nur äußerst schwer vorhersehbar, welche Funktionen und Ereignisse innerhalb des Werkzeugs für andere interessant sein könnten und somit exportiert werden sollten (vgl. [229], Kap. 8).
- Trotzdem muß eine Interoperabilität mit anderen Werkzeugen ohne Änderung des Quelltextes möglich sein. Eine solche Änderung müßte vom Hersteller durchgeführt werden, was bestenfalls ein langwieriger Prozeß ist. Wahrscheinlicher sind die Entwickler nicht bereit, derartige Änderungen durchzuführen, da sie vermeiden wollen, eine Vielzahl unterschiedlicher Programmversionen weiterzugeben und jede Änderung auch die Gefahr von Programmierfehlern in sich birgt.
- Zudem darf durch das Vorsehen einer möglichen Zusammenarbeit mit anderen Werkzeugen keine Abhängigkeit zu diesen entstehen, da der Werkzeughersteller nicht auf andere Firmen angewiesen sein will. Dies verdeutlicht das Beispiel PPPE in § 2.3.2.5. In geringerem Maße gilt dies auch für Abhängigkeiten zu Integrationsinfrastrukturen, insbesondere, wenn nur eine einzige Implementierung vorliegt.

Ein möglicher Ansatz zur Lösung dieser Probleme wird in § 2.3.4.2 vorgestellt.

2.3.4 Lösungsansätze

2.3.4.1 Datenintegration von Laufzeit-Werkzeugen

Die in § 2.2.3.1 diskutierten, allgemeinen Ansätze zur Datenintegration basieren auf der Idee einer gemeinsamen Datenablage, z.B. einer Datenbank, für alle Werkzeuge. Eine solche Strategie ist für passive Daten geeignet, die von den Werkzeugen abgespeichert und später wieder ausgelesen werden. Im Kontext von Laufzeit-Werkzeugen tritt diese Situation nur bei Offline-Werkzeugen auf, wo entsprechende Techniken auch zum Teil eingesetzt werden, z.B. bei der Ablage von Ereignisspuren in standardisierten Spurdateiformaten wie SDDF [8, 240].

Bei Online-Werkzeugen, die in dieser Arbeit vornehmlich betrachtet werden, stehen dagegen aktive Daten, d.h. Daten über ein in Ausführung befindliches System im Vordergrund, so daß die existierenden datenbankbasierten Integrationsmethoden ungeeignet sind. Es werden in diesem Fall also spezielle Ansätze benötigt. Aus der Diskussion in den vorhergehenden Abschnitten ist klar, daß ein solcher Ansatz in der Verwendung gemeinsamer, universeller Monitorschnittstellen bestehen muß.

Die Schaffung einer solchen Schnittstelle mit dem Ziel der einfachen und effizienten Entwicklung paralleler Laufzeit-Werkzeuge war Gegenstand eines gemeinsamen Projekts zwischen dem Autor und T. Ludwig [142], das zur Spezifikation der Monitorschnittstelle OMIS führte und eine wichtige Grundlage der vorliegenden Arbeit bildet. OMIS, sowie einige andere Schnittstellen werden im Hinblick auf ihre Eignung für Umgebungen interoperabler Werkzeuge im folgenden kurz beschrieben. Eine detailliertere Diskussion von OMIS und deren Implementierung OCM erfolgt in § 3.2.1, § 6.1 und § 6.2.

p2d2. Ein erster Schritt in Richtung definierter, universeller Monitorschnittstellen wurde 1993 mit der Entwicklung des verteilten Debuggers `p2d2` an der *Numerical Aerospace Simulation Facility* NAS des *NASA Ames Research Center* unternommen [100, 91, 49]. Das Entwicklungsziel war ein Debugger, der einheitlich auf verschiedenen Zielplattformen zur Verfügung steht. Um dieses Ziel zu erreichen, wurde eine Schnittstelle zwischen dem architekturunabhängigen Teil des Debuggers und den zielsystemabhängigen Teilen definiert, die im wesentlichen das Monitorsystem darstellen.

Das gesamte System ist in einer Client-Server-Architektur realisiert, bei der der Klient den plattformunabhängigen Teil darstellt. Pro Knoten des verteilten Systems ist ein Server vorgesehen, der im Idealfall vom Rechnerhersteller realisiert werden sollte. Die Schnittstelle zwischen dem Klienten und den Servern ist objektorientiert realisiert, wobei dem Klienten für die unterschiedlichen Objekte des verteilten Zielsystems – z.B. Programmcode, Prozesse und Keller – lokale Stellvertreterobjekte mit fest definierten Methoden zur Verfügung gestellt werden. Ein Zugriff auf diese Stellvertreterobjekte führt zu einer transparenten Kommunikation mit dem jeweils zuständigen Server, der die gewünschten Daten des realen Objekts liefert bzw. die angeforderte Manipulation an ihm vornimmt.

Die `p2d2`-Schnittstelle bewirkt eine klare Trennung zwischen Werkzeug und Monitorsystem, kann aber Interoperabilität nicht unterstützen, da sie zum einen nur Funktionen für das Debugging enthält, zum anderen auch nur für den Anschluß jeweils eines Klienten konzipiert wurde.

UMA. Das *Measurement Layer Interface* MLI [276] der *Universal Measurement Architecture* UMA ist dagegen von Anfang an auf die gleichzeitige Bedienung mehrerer Werkzeuge ausgelegt. UMA ist eine Architektur, die die Leistungsmessung in offenen, verteilten Systemen unterstützt. Sie wurde 1994 von X/Open, einem Zusammenschluß nahezu aller wichtigen Hard- und Softwarehersteller vorgestellt [275, 276, 274, 273].

Die MLI-Schnittstelle erlaubt einen netzwerktransparenten, einheitlichen Zugriff auf Leistungsdaten, z.B. CPU und Speicherauslastung verschiedener Rechnerplattformen, die von diesen i.a. in sehr heterogener Weise bereitgestellt werden. Darüber hinaus koordiniert sie die Datenerfassungswünsche der verschiedenen angeschlossen Werkzeuge, so daß z.B. eine Mehrfachmessung identischer Daten vermieden wird. Da diese Schnittstelle nur beobachtende Werkzeuge unterstützt, ist die Koordination vergleichsweise einfach.

Wegen der Einschränkung auf rein beobachtende, systemorientierte Leistungsmessung ist UMA als allgemeine Grundlage interoperabler Laufzeit-Werkzeuge ebenfalls nicht geeignet.

TAU. TAU ist eine an der Universität von Oregon in Eugene, USA entwickelte Umgebung für die Leistungsanalyse paralleler Programme [215, 157, 173]. Das Monitorsystem unterstützt dabei unterschiedlichste Instrumentierungstechniken und erlaubt somit eine flexible Datenerfassung. Die Daten wurden ursprünglich erst am Ende eines Programmlaufs in Form von Profilen und Ereignisspuren für die Werkzeuge zugänglich gemacht. Eine erweiterte Client/Server-basierte Monitorschnittstelle erlaubt nun jedoch auch den Zugriff auf die erfaßte Information zur Laufzeit, wobei mehrere Werkzeuge die Schnittstelle gleichzeitig nutzen können [217]. Wie auch UMA unterstützt die Schnittstelle nur die reine Beobachtung von Programmabläufen, wobei sie jedoch lediglich Dienste für das Lesen der Daten, nicht aber zur Instrumentierung des Zielprogramms bereitstellt. Die notwendige Koordination bei Nutzung durch mehrere Werkzeuge beschränkt sich daher auf das Sperren des Servers während des Lesens von Daten [172].

Eine Realisierung allgemeiner Laufzeit-Werkzeuge auf alleiniger Basis dieser Schnittstelle ist wegen der genannten Einschränkungen nicht möglich.

BEE++. An der Carnegie Mellon Universität in Pittsburgh, USA wurde das objektorientierte Rahmenwerk BEE++ entwickelt, das ein ereignisbasiertes Monitoring verteilter Programme unterstützt [27, 35]. Die Objektorientierung von BEE++ erlaubt eine leichte Anpaßbarkeit der überwachten Ereignisklassen sowie der Ereignisvisualisierer, d.h. der auf BEE++ aufsetzenden Werkzeuge.

Das BEE++ zugrundeliegende Modell der Ereignis-Verarbeitung unterscheidet zwischen Erzeugung, Interpretation und Übertragung von Ereignissen. Während die letztere Aufgabe durch das BEE++-System erfüllt wird, können Ereigniserzeugung und -interpretation beliebig auf Zielprozesse und Werkzeuge verteilt werden. Insbesondere erlaubt BEE++ auch, daß Werkzeuge Ereignisse an einen Zielprozeß senden. Solche Ereignisse können diesen z.B. anhalten oder weiterführen sowie Sensoren, d.h. Instrumentierungen, für die Ereigniserzeugung im Zielprozeß aktivieren. Spezielle Sensoren bzw. die für deren Einbau in das Zielprogramm benötigten Instrumentierungstechniken sind allerdings nicht Bestandteil von BEE++. BEE++ ist somit selbst noch kein Monitorsystem im Sinne der Definition aus § 2.1.3.1, sondern eben ein Rahmenwerk zur einfachen Erstellung solcher Systeme.

BEE++ unterstützt die Weiterleitung von Ereignissen an mehrere Interpreter und somit mehrere beobachtende Werkzeuge. Mechanismen zur Koordination der Werkzeuge sind jedoch nicht explizit vorgesehen. Da das Ereignismodell von BEE++ es aber prinzipiell zuläßt, daß Werkzeuge auch Ereignisse in anderen Werkzeugen überwachen, wäre eine Verhaltensintegration durch impliziten Aufruf leicht realisierbar.

DPCL. Die *Dynamic Probe Class Library* DPCL, die derzeit bei IBM realisiert wird, stellt eine Infrastruktur zur Verfügung, die beliebige Laufzeit-Werkzeuge unterstützt [184, 185, 180]. DPCL basiert auf der Idee der dynamischen Instrumentierung (siehe § 2.1.3.2) und stützt sich in der ersten Implementierung auch auf die `dyninst`-Bibliothek ab. Durch eine Client-Server-Architektur erlaubt DPCL im Gegensatz zu `dyninst` jedoch auch die Instrumentierung und Überwachung nichtlokaler Prozesse.

Analog zu `p2d2` ist auch die DPCL-Schnittstelle objektorientiert und stellt den Klienten für die Objekte des verteilten Systems lokale Stellvertreterobjekte zur Verfügung. Beim Aufruf einer Methode wird transparent eine Kommunikation mit einem DPCL-Dämon durchgeführt, der als Server agiert und die geforderte Instrumentierung vornimmt bzw. Daten über das reale Objekt zurückliefert. Im Gegensatz zu `p2d2` können diese Dämone gleichzeitig von mehreren Klienten benutzt werden.

Allerdings bietet DPCL keine Dienste an, um die Klienten einander gewahr zu machen, was insbesondere bedeutet, daß eine Koordination der Klienten bei konfligierenden Instrumentierungsanforderungen nicht möglich ist. Dies ist jedoch zumindest als Option für zukünftige Erweiterungen vorgesehen [184].

DAMS. DAMS ist ein an der *Universidade Nova di Lisboa* in Portugal entwickeltes verteiltes Monitorsystem, das die simultane Benutzung durch mehrere Werkzeuge unterstützt [41]. Um eine universelle Nutzbarkeit für unterschiedliche Werkzeuge zu gewährleisten, definiert DAMS zunächst lediglich eine verteilte Infrastruktur, die aus einem zentralen Service-Manager sowie je einem lokalen Manager pro Rechnerknoten besteht und damit der in Abb. 2.3 gezeigten Struktur entspricht.

Eine Bibliothek, die zu den Werkzeugen gebunden wird, kapselt die notwendige Kommunikation zwischen dem Werkzeugprozeß und dem Service-Manager. Konkrete Monitordienste werden durch Einbinden von Erweiterungen realisiert. Dazu wird an den Service-Manager ein Dienstmodul angebunden, sowie an jeden lokalen Manager ein Diensttreiber. Monitoranfragen werden zunächst an das zentrale Dienstmodul geleitet, das die Anfragen analysiert und gegebenenfalls über die lokalen Manager an einen oder mehrere Diensttreiber zur Ausführung weiterreicht.

Eine Koordination von Anfragen verschiedener Werkzeuge ist in DAMS möglich, muß allerdings durch die jeweiligen Dienstmodule realisiert werden. Die erste derartige Erweiterung für DAMS ist PDBG zur Unterstützung des verteilten Debuggings [129, 52]. PDBG ist dabei eine Nachfolgeentwicklung des Debuggers DDBG, der zunächst in die Programmierumgebung `GRADE` integriert war [53]. Zur Unterstützung von Werkzeuginteroperabilität stellt PDBG einen Benachrichtigungsdienst zur Verfügung. Ein Werkzeug kann sich dabei für eine Menge vorgegebener Zielprozesse über zwei Klassen von Ereignissen informieren lassen:

1. Änderungen des Ausführungszustands eines Prozesses, z.B. Anhalten oder Terminierung, unabhängig davon, ob diese vom Prozeß selbst oder von einem anderen Werkzeug verursacht wurden.

2. Abschluß einer Monitoraktion auf einem Prozeß, die von einem beliebigen Werkzeug angefordert wurde. Auf diese Weise werden Werkzeuge über die Ergebnisse von Anfragen anderer informiert.

Die angeschlossenen Werkzeuge können so eine konsistente Sicht des Zielsystems aufrechterhalten.

FIRST. In Abschnitt § 2.3.2.6 wurde bereits beschrieben, daß auch die Umgebung FIRST auf eine gemeinsamen Monitor- bzw. Werkzeugdienste-Schicht basiert. Interoperabilität wird von diesen Schichten durch die Koordination von Instrumentierungswünschen verschiedener Werkzeuge unterstützt. Darüber hinaus erlauben die Ereignisdienste das Anhalten und Weiterführen von Prozessen durch andere Werkzeuge zu überwachen. Weitergehende Konzepte sind in der Literatur [187, 124, 188] nicht dokumentiert.

OMIS/OCM. Das unter anderem auch im Rahmen der vorliegenden Arbeit am Lehrstuhl für Rechnertechnik und Rechnerorganisation der Technischen Universität München durchgeführte Projekt OMIS (*On-line Monitoring Interface Specification*) hat das Ziel, eine Standard-Schnittstelle für paralleles und verteiltes Monitoring zu schaffen, die unterschiedliche Werkzeuge auch bei gleichzeitiger Benutzung unterstützt [150]. OMIS ist zunächst nur die Definition der Schnittstelle, begleitend dazu wird in dem Projekt OCM eine Implementierung eines OMIS-konformen Monitorsystems für Netze von Arbeitsplatzrechnern durchgeführt [270].

Wie bei DAMS ist die Implementierung aufgeteilt in eine Infrastrukturkomponente und Erweiterungsmodule, die die eigentlichen Monitordienste realisieren. Die Basisinfrastruktur ist allerdings deutlich mächtiger als bei DAMS und beinhaltet z.B. bereits die Verteilung von Anfragen sowie gewisse Koordinationsdienste, so daß die Erweiterungsmodule im wesentlichen nur noch die eigentlichen lokalen Monitorfunktionen realisieren müssen. Auf diese Weise können für Werkzeuge mit neuen Anforderungen leicht passende Erweiterungen des Monitorsystems vorgenommen werden. Neben den Basisdiensten für Prozeßkontrolle und Debugging enthält der OCM inzwischen Module für Ereignisspuraufzeichnung [249, 250], Leistungsanalyse [24, 25] und Sicherungspunkterstellung [235].

OMIS beinhaltet wie DAMS Ereignisdienste, die Änderungen des Prozeßzustandes der überwachten Zielprozesse an die Werkzeuge melden. Darüber hinaus definiert OMIS Mechanismen, die eine konsistente Abarbeitung von gleichzeitigen, verteilten Anfragen mehrerer Werkzeuge unterstützen. Im Vergleich zu DAMS fehlt bei OMIS dagegen ein Mechanismus, der Werkzeugen bei Bedarf die Ergebnisse von Anfragen anderer Werkzeuge zur Verfügung stellt.

Zusammenfassung. Wie in § 2.2.2 erläutert wurde, muß ein Mechanismus zur Datenintegration zwangsläufig auch eine Synchronisation des Datenzugriffs unterstützen. Bei Monitorsystemen heißt dies vor allem, daß Konflikte, die durch unterschiedliche Instrumentierungswünsche der Werkzeuge und vor allem auch durch Modifikationen des Zielsystemzustands entstehen, in geeigneter Weise behandelt und aufgelöst werden müssen. Von den vier zuletzt genannten, universellen Monitorsystemen bieten DAMS, FIRST und OMIS einzelne Dienste an, die zur Behandlung solcher Konflikte in Frage kämen [236]. In erster Linie sind dies Benachrichtigungsdienste, die Werkzeuge über Zustandsänderungen im Zielsystem unterrichten. Die Lösung der durch eine Änderung eventuell entstehenden Konflikte bleibt den Werkzeugen überlassen. Ob und inwieweit diese Dienste ausreichend sind, um interoperable Laufzeit-Werkzeuge zu ermöglichen, werden wir erst in § 3.5 feststellen können, wenn die zu erfüllenden Anforderungen anhand eines Beispielszenarios konkretisiert wurden.

2.3.4.2 Verhaltensintegration von Laufzeit-Werkzeugen

Die Diskussion in § 2.3.1, § 2.3.2 und § 2.3.3 haben verdeutlicht, daß interoperable Werkzeuge Methoden zur Verhaltensintegration benötigen, die ihnen eine Zusammenarbeit ermöglichen, ohne daß sie dadurch in irgendeiner Weise voneinander abhängig werden. Von den in § 2.2.3.2

beschriebenen prinzipiellen Entwurfsmethoden scheiden daher Kapselung, feste Verdrahtung und impliziter Aufruf von vorneherein aus. Um die in § 2.3.1 angesprochene Modularität zu erreichen, d.h. vorhandene Werkzeuge dynamisch in relativ einfacher Weise zu Werkzeugumgebungen zusammenstellen zu können, muß die Verknüpfung zwischen den Werkzeugen programmierbar gehalten sein. Dies bedeutet, daß nicht der (feste) Programmcode eines Werkzeugs dessen Interaktionen mit den anderen Werkzeugen bestimmt, vielmehr werden diese Interaktionen durch ein separates Programm oder Skript gesteuert, das dadurch die Funktion der Werkzeugumgebung definiert. Wie Tabelle 2.1 zeigt, ist eine solche programmierbare Komponentenverknüpfung nur bei Mechanismen realisiert, die auf dem Mediator- bzw. Interaktor-Ansatz beruhen.

Während die Meldung und Behandlung von Ereignissen innerhalb eines Werkzeugs zur Integration unabdingbar ist, ist eine Unterstützung für den direkten Aufruf externer Funktionen, wie sie bei Interaktoren berücksichtigt ist, bei unserer Vorstellung einer Werkzeugumgebung nicht erforderlich. Ein solcher Aufruf würde ja eine explizite Abhängigkeit von dem Werkzeug erzeugen, das die Funktion zur Verfügung stellt. Dagegen wäre es durchaus wünschenswert, in einem Werkzeug Stellen definieren zu können, an denen gegebenenfalls Funktionen anderer Werkzeuge eingeklinkt werden können. Ein derartiges Konzept realisieren z.B. der erweiterbare Editor `emacs` [79] in Form von *Hooks*, bzw. Internet-Browser und die parallele Programmierumgebung Harness [60, 14] in Form von *Plug-ins*. In der Mediator-Methode kann das Konzept durch eine Erweiterung der Ereignisbehandlung um einen synchronen Modus realisiert werden, bei dem die Komponente, die das Ereignis auslöst, erst dann weiterarbeiten kann, wenn der Mediator alle für dieses Ereignis vorgesehenen Aktionen abgearbeitet hat. Eine solche Erweiterung ist bisher bei keiner der Realisierungen verwirklicht.

Da hier davon ausgegangen wird, daß die Werkzeuge, die in einer Umgebung zusammenarbeiten sollen, unabhängig voneinander entwickelt wurden, kann nicht vorausgesetzt werden, daß die von diesen definierten Schnittstellen in irgendeiner Weise kompatibel sind. Die Möglichkeit einer Schnittstellen-Anpassung durch den verwendeten Mechanismus zur Verhaltensintegration ist daher essentiell.

Betrachtet man nochmals Tab. 2.1, wird deutlich, daß von den mediatorbasierten Realisierungen TIL die genannten Anforderungen noch am ehesten erfüllt. Als Problem bleibt jedoch bestehen, daß der Mediator-Ansatz erfordert, daß die Werkzeuge zur Übersetzungszeit ihre Aufruf- und Ereignisschnittstelle in Form eines abstrakten Verhaltenstyps nach außen zur Verfügung stellen. Wie bereits angesprochen, ist aber im allgemeinen bei der Programmierung eines Werkzeugs noch nicht bekannt, welche Ereignisse und Funktionen möglicherweise von anderen Werkzeugen benötigt werden. Wollte man alle denkbaren Fälle abdecken, entstünde eine riesige Schnittstelle, die zudem zu einem sehr aufgeblähten Werkzeug führen würde, da für jedes Element der Schnittstelle ein Stub bzw. ein Skelett erzeugt wird.

In dieser Arbeit wird daher eine gänzlich andere Vorgehensweise vorgeschlagen. Sie baut auf einem Grundgedanken der dynamischen Instrumentierung auf, nämlich ausführbare Programme nicht als unveränderlich und endgültig zu betrachten, sondern als Objekte, die dynamisch angepaßt und erweitert werden können. So werden beim *Process Hijacking* Prozesse nachträglich migrierbar gemacht [279], `kerninst` erlaubt sogar die Optimierung und Spezialisierung von Betriebssystemen zur Laufzeit [234]. Wir werden in diesem Sinne daher untersuchen, ob durch Verwendung dynamischer Monitoringtechniken ein Einbau der für die Verhaltensintegration notwendigen Schnittstellen sowie der zugehörigen Stubs und Skelette in die Werkzeuge zur Laufzeit möglich ist. Das Monitorsystem überwacht und steuert dabei nicht mehr nur das Zielsystem, sondern auch die Werkzeuge, so daß es aus Sicht des Monitorsystems praktisch keinen Unterschied zwischen einem Werkzeugprozeß und einem Anwendungsprozeß gibt[6] [264].

So merkwürdig dieser Ansatz, in dem das Monitorsystem auch die Verhaltensintegration realisiert, zunächst erscheinen mag, hat er doch eine Reihe wichtiger Vorteile:

1. Werkzeuge müssen die Schnittstelle zu anderen Werkzeugen nicht a-priori mit Hilfe vorgegebener Mechanismen definieren.

   Zunächst wird dadurch die Abhängigkeit des Werkzeugs von der konkreten Infrastruktur zur Verhaltensintegration vermieden. Beispielsweise kann `EM` zwar ohne den Rest der `PREPARE`-Umgebung benutzt werden, benötigt aber trotzdem ein installiertes und laufendes `ToolTalk`, obwohl es in diesem Fall überflüssig ist.

   Entscheidender ist jedoch, daß die Schnittstelle nun jederzeit, selbst zur Laufzeit erweitert werden kann, wenn Bedarf dazu besteht. Konkret kann ein geeignetes Monitorsystem den Aufruf und die Rückkehr beliebiger Funktionen als Ereignisse überwachen, sowie beliebige Funktionen innerhalb des Werkzeugprozesses aufrufen. Zusätzlich kann das Monitorsystem auch neuen Code zum Werkzeug hinzubinden (siehe die Diskussion von `dyninst` in § 2.1.3.2). Die Erweiterung der Schnittstelle ist also zur Laufzeit ohne Änderung des Werkzeug-Quelltextes möglich, wenn die relevanten Ereignisse mit Aufruf bzw. Rückkehr von Funktionen korrelieren und Schnittstellen für Aktionen durch reines Hinzufügen von Code geschaffen werden können. Erwartungsgemäß sollte diese Bedingung bei gut strukturierten Werkzeugen erfüllt sein. Praktische Erfahrungen hierzu werden in § 7 beschrieben.

   Notwendig bleibt auch bei dieser Vorgehensweise eine Dokumentation der für die Schnittstelle benötigten Funktionen, insbesondere deren Namen, Parameter und Bedeutung. Aber auch hier muß die Information nicht für alle möglicherweise interessierenden Funktionen auf einmal vorliegen, sondern kann sozusagen dynamisch auf Anfrage vom Hersteller bereitgestellt werden. Bei entsprechenden Kontakten zu den Werkzeug-Entwicklern sind solche Informationen im allgemeinen einfacher zu erhalten als Änderungen im Quelltext der Werkzeuge. Gegebenenfalls läßt sich die Information meist sogar aus den vom Compiler erzeugten Symboltabellen des Werkzeugs – eventuell mit Hilfe eines Debuggers – ermitteln[7]. Selbst bei *Open Source*-Werkzeugen, bei denen der Quelltext für eventuelle Änderungen zur Verfügung steht, hat der dynamische Einbau der benötigten Schnittstellen den nicht zu unterschätzenden Vorteil, daß die Anwender weiterhin die normale, bei ihnen installierte Version des Werkzeugs verwenden können. Akzeptanzprobleme durch die Notwendigkeit einer erneuten Installation, durch verzögerte Bereitstellung neuer Versionen sowie durch mangelndes Vertrauen in modifizierte Programme können so vermieden werden.

   Insgesamt bietet der Ansatz also auch Drittentwicklern eine Chance, bereits vorhandene Werkzeuge interoperabel zu machen. Eine unvermeidbare Voraussetzung dazu ist allerdings, daß diese bereits ein entsprechend mächtiges, gemeinsames Monitorsystem verwenden.

[6] Die Idee zur Schaffung dieser Symmetrie geht auf eine persönliche Diskussion mit Al Geist, Oak Ridge National Lab. zurück, dem an dieser Stelle herzlich gedankt sei.

[7] Natürlich stellt sich hier, wie auch bereits bei der dynamischen Veränderung von Programmcode zur Laufzeit, die Frage nach lizenzrechtlichen Konsequenzen. Wir werden diese Frage im Rahmen dieser technisch orientierten Arbeit jedoch nicht aufgreifen.

2. Vergleicht man die abstrakte Funktion eines Mediators mit der eines Monitorsystems, stellt man fest, daß in beiden Fällen Ereignisse in externen Prozessen überwacht und aufgrund vorgegebener Regeln beim Eintreten eines Ereignisses passende Aktionen ausgelöst werden, die andere Prozesse, eventuell auf anderen Rechnerknoten betreffen. Da bei einem universell verwendbaren Monitorsystem diese Regeln programmierbar sein müssen, kann die Funktion eines Mediators über die Ereignis/Aktions-Regeln des Monitorsystems beschrieben werden. Die Ereignis-/Aktions-Verwaltung sowie die Kommunikationsmechanismen des Monitorsystems können also wiederverwendet werden, um das Mediatorkonzept zu realisieren.

3. Benutzt ein Werkzeug zwei verschiedene Infrastrukturen für das Monitoring und die Verhaltensintegration, so müssen diese selbstverständlich gleichzeitig verwendbar, d.h. interoperabel sein.

   In der Praxis ergeben sich technische Schwierigkeiten dadurch, daß das Werkzeug mit beiden Infrastrukturen asynchron kommunizieren muß. Vom Monitorsystem werden auf diese Weise Ereignisse gemeldet, vom Mediator die Aufrufe von Aktionen angefordert. Ein alternatives Warten auf Nachrichten verschiedener Quellen ist jedoch abhängig von den jeweiligen Kommunikationsmechanismen nicht immer möglich. Die Einführung von Threads oder unterbrechungsgesteuerter Kommunikation löst zwar dieses Problem, schafft aber gleichzeitig eine ganze Anzahl neuer, z.B. durch nicht-wiedereintrittsfähige Bibliotheksfunktionen oder die Notwendigkeit der Synchronisation bei Zugriff auf Daten.

   Die Verwendung nur eines Mechanismus vermeidet Interoperabilitätsprobleme auf dieser Ebene vollständig.

2.4 Zusammenfassung

In diesem Kapitel wurden die Grundlagen beschrieben, die für das in den folgenden Kapiteln entwickelte Konzept zur technischen Realisierung interoperabler Laufzeit-Werkzeuge relevant sind. Die Diskussion der Funktion und des Aufbaus von Werkzeugen hat gezeigt, daß sich die verschiedenen Werkzeuge in ihrer Struktur sehr ähnlich sind und zumindest konzeptuell in ein Monitorsystem zur Datenerfassung und Manipulation des Zielsystems sowie das eigentliche Werkzeug zur Datenanalyse und -darstellung zergliedern lassen. Es hat sich auch gezeigt, daß Werkzeuge nur jeweils einen relativ spezifischen Aspekt des Laufzeitverhaltens eines Anwendungsprogramms untersuchen bzw. manipulieren können, so daß der Wunsch entsteht, auch mehrere Werkzeuge gleichzeitig verwenden zu können.

Dies führt zum Begriff der Interoperabilität, der wegen seiner Vielschichtigkeit zunächst genau abgegrenzt wurde: Wir betrachten unabhängig entwickelte und verwendbare Software-Komponenten, die bei Bedarf gleichzeitig anwendbar sind und dann auch zusammenarbeiten können. Bei der Integration, d.h. der Herstellung der Interoperabilität, sind prinzipiell zwei verschiedene Aspekte zu berücksichtigen: die Datenintegration, die den Zugriff auf gemeinsam genutzte Daten regelt, sowie die Verhaltensintegration, die die Zusammenarbeit der beteiligten Komponenten bestimmt.

Nach dieser zunächst unabhängigen Diskussion von Laufzeit-Werkzeugen und Methoden zur Unterstützung von Interoperabilität wurde die Idealvorstellung einer Umgebung interoperabler Werkzeuge skizziert und der derzeitigen Realität gegenübergestellt. Für die beobachtete

Diskrepanz wurden mehrere technische Gründe identifiziert. Zum einen erfordert die Datenintegration von Laufzeit-Werkzeugen ein universell verwendbares Monitorsystem, das alle Werkzeuge gleichzeitig bedient. Derartige Monitorsysteme sind aber noch kaum verfügbar. Von den existierenden Systemen ist auch noch nicht bekannt, ob die dort realisierten Mechanismen in der Praxis tatsächlich ausreichend sind. Zum anderen bereitet auch die Verhaltensintegration Schwierigkeiten. Bei der Entwicklung einer Werkzeugs ist noch nicht bekannt, mit welchen anderen Werkzeugen ein Anwender dieses später zusammen verwenden will. Die Definition einer geeigneten externen Schnittstelle ist daher zum Entwicklungszeitpunkt praktisch unmöglich. Zudem muß vermieden werden, daß durch die Art der Integration Abhängigkeiten zu anderen Werkzeugen entstehen.

Um die zuletzt genannten Probleme zu lösen, wurde die Idee entwickelt, auch die Verhaltensintegration mit Hilfe des ohnehin benötigten Monitorsystems zu realisieren. Durch Anwendung dynamischer Instrumentierungstechniken können die erforderlichen Schnittstellen zur Laufzeit in die Werkzeuge eingefügt werden. Zudem kann die Ereignis/Aktions-Verwaltung des Monitorsystems zur Realisierung des Mediator-Konzepts wiederverwendet werden.

2.5 Zielsetzung der Arbeit

Die vorliegende Arbeit verfolgt zwei hauptsächliche Zielsetzungen. Ein Ziel ist zunächst, detailliert die technischen Anforderungen zu ermitteln, die interoperable Laufzeit-Werkzeuge an Infrastrukturen zur Daten- und Verhaltensintegration stellen. Diese wurden bisher nur äußerst pauschal und abstrakt beschrieben, wirken sich aber im Detail wesentlich auf die Realisierung derartiger Infrastrukturen aus. Zur Definition dieser Anforderungen wird zunächst in einer Fallstudie ein konkretes Szenario aus vier Werkzeugen betrachtet, die zum Zweck der effizienten Fehlersuche gleichzeitig angewendet werden sollen. Die dabei gewonnenen Ergebnisse werden dann verallgemeinert, so daß zum einen die gesuchten Anforderungen resultieren, zum anderen eine allgemeine Vorgehensweise zur Integration von Laufzeit-Werkzeugen definiert werden kann. Anhand des entstehenden Katalogs von Anforderungen kann dann noch einmal im Detail geprüft werden, inwieweit diese von den vorhandenen Infrastrukturen bereits erfüllt werden.

Die zweite Zielsetzung ist die Beantwortung der Frage, ob die Anforderungen interoperabler Werkzeuge auf Basis der oben skizzierten Idee auch alleinig in einem Monitorsystem umgesetzt werden können, und falls ja, wie. Die Untersuchung dieser Frage wird anhand der Monitor-Schnittstelle OMIS und ihrer Implementierung durch das Monitorsystem OCM durchgeführt. Schließlich wird die prinzipielle Tragfähigkeit der Idee auch durch eine prototypische Implementierung des Beispielszenarios aus der Fallstudie überprüft. Dabei kann zumindest qualitativ evaluiert werden, wie gut die angestrebten Ziele erreicht werden können.

Kapitel 3

Ein Szenario für interoperable Werkzeuge

In diesem Kapitel werden die Anforderungen und Voraussetzungen genauer definiert, die erfüllt sein müssen, um Werkzeugumgebungen der Art realisieren zu können, wie sie in § 2.3.1 skizziert wurden. Wir werden dazu ein Szenario beschreiben, bei dem vier unabhängig voneinander entwickelte Werkzeuge zusammen verwendet werden, um den Fehlersuchzyklus für parallele Programme mit langen Ausführungszeiten zu verkürzen [263, 258, 264]. Eine detaillierte Untersuchung der möglichen Realisierung und ihrer Probleme wird die gesuchten Anforderungen offenlegen.

3.1 Eine Werkzeugumgebung zur effizienten Fehlersuche

Die Suche nach Software-Fehlern ist typischerweise ein zyklischer Prozeß, bei dem aus der nach außen hin sichtbaren Wirkung eines Fehlers, z.B. einem Programmabbruch durch einen unzulässigen Speicherzugriff, auf dessen Ursache, z.B. ein falsch deklariertes Feld, zurückgeschlossen werden muß [246]. Ein übliches Vorgehen unter Verwendung von haltepunktbasierten Online-Debuggern besteht darin, den Programmablauf an einigen Stellen zu unterbrechen und den *internen* Programmzustand auf mögliche Auswirkungen des Fehlers, d.h. auf Abweichungen vom erwarteten Zustand hin zu untersuchen. Geeignete Stellen für diese Unterbrechungen ergeben sich in der Regel aus einer Anfangshypothese über die Fehlerursache. Falls keine initiale Hypothese vorliegt, kann z.B auch die Technik der binären Suche angewandt werden [228].

Aus der Untersuchung des Programmzustands an einem Haltepunkt ergeben sich neue Hypothesen über die Fehlerursache und über den Zeitpunkt des Fehlers im Programmablauf. Wird der Zustand als noch fehlerfrei eingestuft, wird davon ausgegangen, daß sich die Ursache des Fehlers in der Zukunft befindet und das Programm bis zum nächsten Haltepunkt weitergeführt. Ist der Zustand fehlerhaft, liegt die Fehlerursache in der Vergangenheit. Um sie genauer einzugrenzen muß der Programmlauf daher wiederholt und entsprechend früher angehalten werden. Diese Vorgehensweise, die als *Cyclic Debugging* [154] bezeichnet wird, ist in Abb. 3.1 veranschaulicht.

In der Praxis können bei komplexen Fehlern durchaus 5-10 Durchläufe durch diesen Zyklus notwendig werden, um deren Ursache genau zu ermitteln. Während dies bei den meisten Programmen unerheblich ist, kann eine derartige Fehlersuche äußerst langwierig werden, falls ein Programm eine sehr lange Laufzeit besitzt und erst gegen Ende ein Fehler auftritt. Anwendungen aus dem Bereich des parallelen Hochleistungsrechnens sind gute Kandidaten für diese

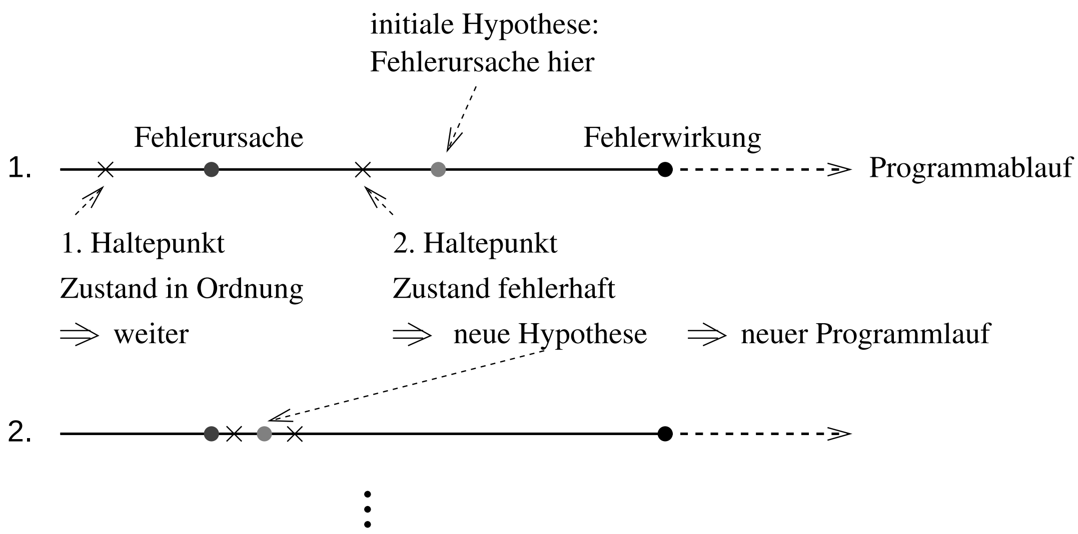


Abbildung 3.1: Vorgehensweise beim zyklischen, haltepunktbasierten Debugging

Kategorie, da ihre Rechenzeiten meist Stunden oder gar Tage betragen. Zwar können solche Anwendungen an kleinen Beispielen getestet werden, es sind jedoch nicht alle Fehler mit solchen Testeingaben zu provozieren bzw. zu reproduzieren. Oft treten Fehler erst im realen Betrieb mit großen Eingabedatensätzen auf. Als ein durch die Medien populär gewordenes Beispiel sei die 1995 aufgetretene Panne bei der Installation des Computersystems zur Steuerung des Stellwerks Hamburg-Altona genannt [153], die durch einen Stack-Überlauf verursacht wurde, der erst im realen Betrieb auftrat. Ähnliche Situationen können bei parallelen Programmen auftreten, etwa wenn Nachrichtenpuffer der Programmierbibliothek zu klein werden. Bei MPI können in einer derartigen Situation z.B. Deadlocks entstehen, die bei kleinen Testbeispielen nicht auftreten.

Der beschriebene zyklische Fehlersuchprozeß kann beschleunigt werden, wenn es ermöglicht wird, schnell zu früheren Programmzuständen zurückzukehren, ohne dazu das Programm vollständig neu ausführen zu müssen. Eine naheliegenden Idee zur Erreichung dieses Ziels ist die Kombination des Debuggers mit einem System zur Sicherungspunkterstellung. Während der Ausführung des Programms können dann periodisch Sicherungspunkte erzeugt werden. Wird während der Fehlersuche festgestellt, daß der Programmzustand fehlerhaft ist, kann relativ schnell zu einem gesicherten, früheren Zustand zurückgekehrt werden. Im Idealfall sollte eine solche Rückkehr nicht nur zu Sicherungspunkten, sondern zu beliebigen vom Benutzer spezifizierten Zuständen möglich sein. Dann wäre ein einfaches, schrittweises Rückwärtsgehen zur Prüfung von Fehlerhypothesen durchführbar, von der Fehlerwirkung ausgehend hin zur Fehlerursache. Eine solche Funktion kann im Prinzip relativ einfach realisiert werden, indem das Programm auf den letzten vor dem Zielzustand erzeugten Sicherungspunkt zurückgesetzt und anschließend automatisch bis zum Erreichen des Zielzustandes weiter ausgeführt wird.

Die beschriebene Idee ist in Abb. 3.2 veranschaulicht. Zu ihrer praktischen Umsetzung sind jedoch noch zwei Probleme zu lösen:

1. Wie kann der Benutzer in einfacher Weise den gewünschten Zielzustand beim Rücksetzen spezifizierten? Da wir parallele Programme betrachten, handelt es sich dabei um einen globalen Zustand, der alle Prozesse und auch den Kommunikationsstatus mit einschließt. Eine einfache und komfortable Möglichkeit besteht in der Nutzung eines Programmablauf-Visualisierers, der globale Zustände graphisch darstellt. Ein Beispiel eines solchen Vi-

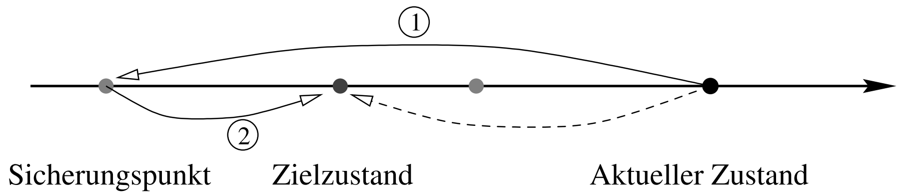


Abbildung 3.2: Rücksetzen eines Programmablaufs auf einen vorgegebenen Zustand

sualisierers stellt `VISTOP` dar [33]. `VISTOP` zeigt jeweils einen globalen Zustand als „Schnappschuß" des Programmablaufs an und erlaubt durch Steuertasten ähnlich einem Videorecorder im Standbildmodus das Vor- und Zurückfahren innerhalb der aufgezeichneten Zustandsfolge. Das Werkzeug kann daher in idealer Weise zur Spezifikation des Zielzustands verwendet werden. Der Benutzer muß den gewünschten Zustand lediglich mit Hilfe der Steuertasten anfahren.

2. Wie in § 2.1.2.1 erläutert wurde, können parallele Programme aufgrund von *Races* nichtreproduzierbares Verhalten zeigen. Bei der Wiederausführung des Programms zwischen Sicherungspunkt und ausgewähltem Zielzustand ist daher nicht sichergestellt, daß sich das Programm genauso verhält wie beim vorherigen Ablauf. Wie Abb. 3.3 veranschaulicht, ist es daher möglich, daß es den Zielzustand verfehlt und einem ganz anderen möglichen Ablauf folgt. Um dieses unerwünschte Verhalten auszuschließen, kann ein Werkzeug zur deterministischen Ablaufsteuerung eingesetzt werden.

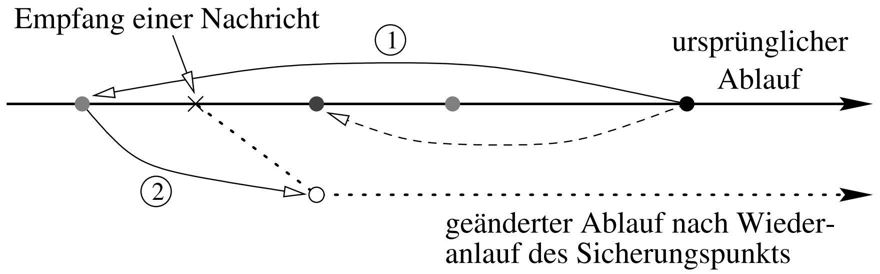


Abbildung 3.3: Fehlerhaftes Rücksetzen aufgrund von Nichtdeterminismus

Insgesamt besteht das Szenario somit aus vier recht unterschiedlichen Werkzeugen: Dem Debugger, einem Sicherungspunktgenerator, einem Ablauf-Visualisierer und einer deterministischen Ablaufsteuerung. Die konkret verwendeten Werkzeuge werden im nächsten Abschnitt näher untersucht.

Die Idee der Kombination von Debugging und Sicherungspunkten ist freilich nicht neu. Beispiele derartiger Systeme sind IGOR [72], Bugnet [260], Recap [189] und der kommerzielle Debugger CDBX von Cray [56]. Auch die Dissertation von J. Trinitis [235], die eine Grundlage der vorliegenden Arbeit darstellt, behandelt dieses Thema. Das skizzierte Szenario unterscheidet sich von diesen Arbeiten vor allem in zwei Aspekten:

1. Es ermöglicht ein Rücksetzen paralleler Programme auf beliebige frühere Zustände, nicht nur auf Sicherungspunkte. IGOR bietet eine ähnliche Funktion, jedoch nur für sequentielle Programme.

2. Die Realisierung erfolgt durch Kombination unabhängiger Werkzeuge, die kein Wissen über das Vorhandensein oder gar die Funktion der anderen Werkzeuge besitzen. Trinitis geht einen ersten Schritt in diese Richtung, indem Debugger und Sicherungspunktgenerator nicht wie bei früheren Systemen fest gekoppelt oder monolithisch realisiert sind, sondern lose gekoppelte, interoperable Software-Komponenten darstellen. Obwohl die Komponenten unabhängig voneinander verwendbar sind, bleibt aber eine gewisse Abhängigkeit bestehen, da der Debugger die Funktionsweise des Sicherungspunktsystems explizit kennen und bereits entsprechenden Code zu dessen Ansteuerung beinhalten muß.

Schließlich sollte erwähnt werden, daß diese Fallstudie auch ein anderes Ziel verfolgt als die genannten früheren Arbeiten. Für sie gilt das Sprichwort „Der Weg ist das Ziel". Neben dem eigentlichen Resultat in Form einer nutzbaren Werkzeugumgebung stehen also vor allem die Techniken im Vordergrund, die notwendig sind, um dieses Resultat erreichen zu können.

3.2 Die Einzelwerkzeuge des Beispielszenarios

Bevor wir die Anforderungen näher untersuchen können, die das skizzierte Szenario an diese Techniken stellt, erfolgt in den nachfolgenden Abschnitten eine kurze Beschreibung der vier verwendeten Einzelwerkzeuge. Die Diskussion wird sich dabei auf die relevanten Funktionen der Werkzeuge aus Benutzersicht sowie eine abstrakte Beschreibung der Teile der Implementierung beschränken, die für die anschließende Analyse von Bedeutung sind. Für eine detailliertere Beschreibung sei auf die jeweils angegebene Literatur verwiesen.

Konkret soll das Szenario durch die folgenden Werkzeuge realisiert werden, die alle in § 2.1.4 definierten Klassen von Online-Werkzeugen repräsentieren:

1. Dem parallelen Debugger `DETOP`, einem zentralisierten, interaktiven und modifizierenden Werkzeug,
2. dem Sicherungspunktgenerator `CoCheck`, der als zentralisiert, automatisch und modifizierend klassifiziert werden kann,
3. dem Visualisierer `VISTOP`, der wie der Debugger zentralisiert und interaktiv ist, jedoch ausschließlich beobachtend,
4. der deterministischen Ablaufsteuerung `Codex`, die abgesehen von der Initialisierungsphase ein verteiltes, automatisches und modifizierendes Werkzeug darstellt.

Da drei dieser Werkzeuge verändernd in den Ablauf des Zielprogramms eingreifen können, ist von vorneherein abzusehen, daß ihre gleichzeitige Verwendung nicht ohne weiteres möglich sein wird. Bei rein beobachtenden Werkzeugen ergäbe sich eine wesentlich einfachere Situation. Zudem sollen die Werkzeuge nicht nur gleichzeitig angewandt werden, sondern sich auch gegenseitig unterstützen. Technisch bedeutet dies, daß neben einer Datenintegration auch eine Verhaltensintegration durchzuführen ist. Dies stellt sozusagen die höchste Stufe der Interoperabilität dar [265], die bisher zwischen Laufzeit-Werkzeugen noch nicht erreicht wurde (vgl. Tab. 2.2).

In § 2.5 wurde als Zielsetzung dieser Arbeit die Unterstützung der Integration von unabhängig entwickelten Werkzeugen genannt, die sogar von verschiedenen Herstellern stammen können. Es stellt sich daher die Frage, inwieweit die ausgewählten Werkzeuge repräsentativ für diese Situation sind. Zwar wurden alle genannten Werkzeuge im Rahmen des Sonderforschungsbereichs 342[1] am LRR-TUM[2] entwickelt, sie entstammen jedoch unterschiedlichen Einzelprojekten, die in jeweils anderem Kontext von verschiedenen Entwicklerteams durchgeführt wurden, wobei eine mögliche Zusammenarbeit der Werkzeuge zunächst kein Ziel war:

- `DETOP` wurde ursprünglich für eine Multithread-Umgebung auf Intel iPSC Parallelrechnern entwickelt [38, 39], später in Kooperation mit der Fa. Parsytec auf deren Rechner und das Programmiermodell PARIX übertragen [174, 175, 266].
- `CoCheck` entstand als Ergebnis der Dissertation von G. Stellner in Form einer Zusatzbibliothek für PVM und MPI [225, 218, 221].
- `VISTOP` wurde im Rahmen der Doktorarbeit von P. Braun zunächst ebenfalls für iPSC Rechner entwickelt [33], später entstand eine ereignisspur-basierte Offline-Version für das Programmiermodell PVM [40, 255].
- `Codex` schließlich wurde als unabhängiges Werkzeug für PVM-Programme in der Dissertation von M. Oberhuber realisiert [169]. Im Gegensatz zu den anderen Werkzeugen ist `Codex` zum Teil in Java implementiert.

Erst im Rahmen des Projekts THE TOOL-SET [145, 262, 265] und nach Fertigstellung des Monitorsystems OCM (siehe § 3.2.1) wurde `DETOP` an PVM angepaßt, `CoCheck` als Erweiterungsmodul in den OCM integriert [235] und `VISTOP` wieder in einer Online-Version realisiert. In diesem Projekt wurde erstmals auch eine Zusammenarbeit der Werkzeuge als Ziel definiert, das mit der vorliegenden Arbeit erreicht wird.

In der Gesamtbetrachtung ergibt sich für den Zweck der Fallstudie dadurch eine nahezu ideale Situation: Zum einen sind die Werkzeuge heterogen genug, um repräsentativ zu sein für den Fall, in dem beliebige unabhängig entwickelte Werkzeuge miteinander kombiniert werden. Zum anderen ist das Implementierungswissen über diese Werkzeuge noch genügend konzentriert, um die von uns angestrebte Analyse in einfacher Weise durchführen zu können: Wir werden jeweils aus Benutzer-, algorithmischer und Implementierungssicht die gewünschten Zusammenarbeit der Werkzeuge, aber auch ihre unerwünschten Wechselwirkungen untersuchen. Daraus ergibt sich die notwendige Interaktion und Koordination zwischen den Werkzeugen. Aus diesen können schließlich Anforderungen an eine notwendige unterstützende Infrastruktur sowie die Werkzeuge selbst abgeleitet werden.

3.2.1 OMIS und OCM: Verteiltes Monitoring

Da die Monitorschnittstelle OMIS und deren Implementierung OCM die Realisierungsbasis der Werkzeuge `DETOP`, `CoCheck` und `VISTOP` darstellen, ist an dieser Stelle eine kurze Beschreibung erforderlich. Weitere Details, insbesondere über die im Rahmen dieser Arbeit durchgeführten Erweiterungen zur Unterstützung der Werkzeuginteroperabilität werden in Kapitel 6 diskutiert.

3.2.1.1 Die Schnittstellendefinition OMIS

Ziel des Projekts OMIS (*On-line Monitoring Interface Specification*) war es, eine standardisierte Schnittstelle zwischen Laufzeit-Werkzeugen und den zugrundeliegenden Monitorsystemen zu

[1] „Werkzeuge und Methoden zur Nutzung paralleler Rechnerarchitekturen“, gefördert von der Deutschen Forschungsgemeinschaft.

[2] Lehrstuhl für Rechnertechnik und Rechnerorganisation an der Fakultät für Informatik der Technischen Universität München.

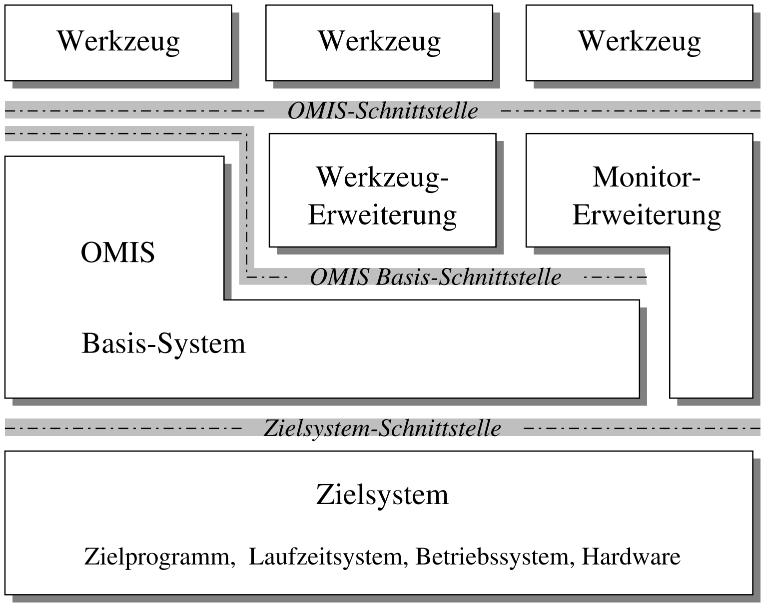


Abbildung 3.4: Das Schichtenmodell von OMIS

schaffen. Dadurch wird ein systematischer, effizienter Werkzeugentwurf ermöglicht und die Portabilität der Werkzeuge verbessert [142]. Ein Fernziel war die Unterstützung interoperabler Werkzeuge.

Um sowohl eine hohe Flexibilität als auch eine effiziente Implementierbarkeit zu gewährleisten, definiert OMIS eine programmierbare Schnittstelle [150]. Sie basiert auf dem Ereignis/Aktions-Paradigma. Werkzeuge können dabei Relationen zwischen Ereignissen im Zielsystem und einer Menge auszuführender Aktionen definieren. Das Monitorsystem überwacht dann das Eintreten dieser Ereignisse. Jedesmal wenn ein Ereignis erkannt wird, führt das Monitorsystem autonom, d.h. ohne Notwendigkeit für eine weitere Interaktion mit dem Werkzeug, die definierten Aktionen aus. Wird eine Ereignis/Aktions-Relation mit einer leeren Ereignisdefinition spezifiziert, so werden die Aktionen sofort und genau einmal ausgeführt. Die Motivation für dieses Paradigma bildet die Beobachtung, daß unterschiedliche Werkzeuge oft dieselben Ereignisse beobachten, diese jedoch verschieden bearbeiten, d.h. ihnen jeweils andere Aktionen zuordnen. Durch die Programmierbarkeit ist eine gute Wiederverwendung bereits vordefinierter Ereignisse und Aktionen gewährleistet.

Trotzdem läßt sich nicht vermeiden, daß manche Werkzeuge spezielle Dienste benötigen. Die OMIS-Schnittstelle ist daher so ausgelegt, daß sie in definierter Weise um Ereignisse und Aktionen erweitert werden kann. Dieses Erweiterungskonzept erlaubt es auch, plattformspezifische Dienste des Monitorsystems sauber von den universellen Diensten zu trennen. OMIS unterscheidet Werkzeug-Erweiterungen, die sich ausschließlich auf die Dienste der OMIS-Schnittstelle abstützen und damit unabhängig von der konkreten Implementierung des Monitorsystems sind, und Monitor-Erweiterungen, die auch direkt auf das Zielsystem zugreifen (siehe Abb. 3.4). Die Dienste solcher Erweiterungen werden mit Hilfe einer Schnittstellenbeschreibungssprache definiert und als Bibliotheken zum Monitorsystem gebunden.

OMIS definiert eine objektbasierte Schnittstelle. Ihre Grundlage sind also abstrakte Objekte, auf die nur mit Hilfe definierter Methoden (Dienste) zugegriffen werden kann. Es gibt jedoch keine Vererbung. Die Objekte der Schnittstelle sind unterteilt in Systemobjekte, die das Zielsy-

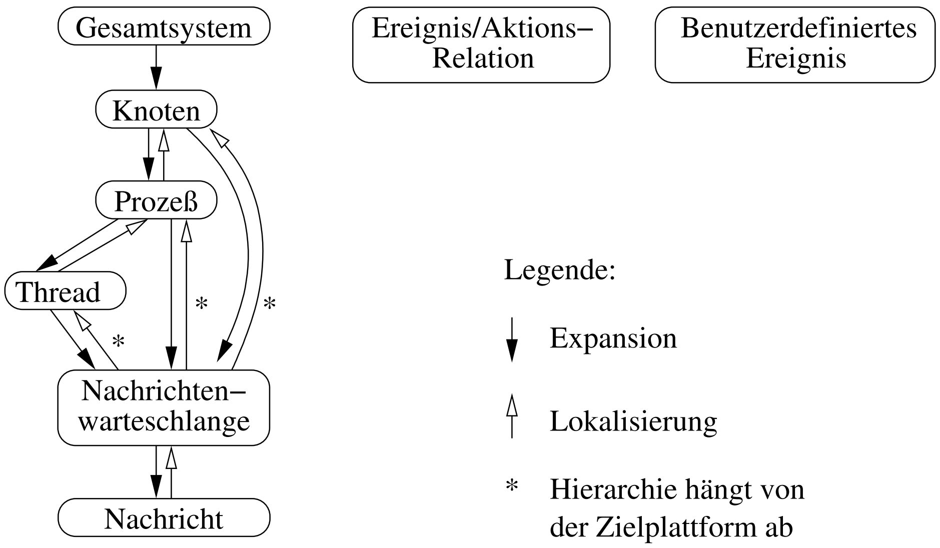


Abbildung 3.5: Objekthierarchie in OMIS

stem beschreiben, und Monitorobjekte, die vom Monitorsystem selbst erzeugt werden. Die von der Basisschnittstelle definierten Systemobjekte sind das Zielsystem als ganzes, seine Rechnerknoten, Prozesse, Threads, Nachrichtenwarteschlangen und Nachrichten. Das Zielsystem muß dabei von den Werkzeugen selbst definiert werden, indem sie sich über entsprechende Dienste explizit an Knoten und Prozesse ankoppeln. Dadurch können sich einzelne Werkzeuge auch unterschiedliche Sichten des zu überwachenden Systems aufbauen. Monitorobjekte der Basisschnittstelle sind die definierten Ereignis/Aktions-Relationen sowie benutzerdefinierte Ereignisse. Erweiterungen können jeweils neue Objekttypen hinzufügen.

Die Objekte werden durch opake Bezeichner, sogenannte ***Token***, repräsentiert. Als Zugriffsmethoden stehen drei verschiedene Arten von Diensten zur Verfügung:

1. Informationsdienste liefern Daten über das Objekt bzw. dessen Zustand zurück,
2. Manipulationsdienste erlauben eine Veränderung des Objektzustands,
3. Ereignisdienste erkennen Zustandsübergänge in Objekten.

Die Aktionen einer Ereignis/Aktions-Relation bestehen aus Diensten der ersten beiden Klassen. Die Überwachung des Ereignisses wird durch einen Ereignisdienst realisiert. Ein Parameterübergabemechanismus (Ereigniskontextparameter) erlaubt die Übertragung von Information über das Ereignis an die Aktionen.

Im Gegensatz zu üblichen objektbasierten Programmiermodellen erlaubt OMIS auch die gleichzeitige Anwendung von Diensten auf Mengen von Objekten, um die Skalierbarkeit der Schnittstelle auch für große Zielsysteme sicherzustellen. Unterstützt wird dies weiterhin durch das Konzept einer automatischen Konvertierung von Objekttoken. Sie basiert auf der Beobachtung, daß die Objekte des Zielsystems in einer natürlichen „ist enthalten in“-Beziehung stehen: Das System enthält Rechnerknoten, jeder Knoten „enthält“ Prozesse, die wiederum Threads beinhalten, usw. Auf Basis dieser Beziehung sind zwei komplementäre Konvertierungen von Token definiert: Expansion und Lokalisierung (siehe Abb. 3.5). Bei der Expansion wird ein Objekt (genauer: dessen Token) in die Menge (der Token) aller in ihm enthaltenen Unterobjekte umgewandelt. Die Lokalisierung liefert umgekehrt das „Behälterobjekt“ eines Objekts.

Das nachfolgende Beispiel einer Ereignis/Aktions-Relation verdeutlicht die genannten Konzepte:

```
thread_reached_addr([p_12,p_28,p_352], 0x1234) :
    node_get_info([$thread],257)
    thread_get_backtrace([$thread],0)
    thread_stop([a_])
```

Jedesmal, wenn ein beliebiger Thread innerhalb eines der Prozesse `p_12`, `p_28` oder `p_352` die Adresse `0x1234` erreicht, wird Information über den Knoten dieses Threads sowie der Prozeduraufrufkeller des Threads an das Werkzeug übermittelt. Der Ereigniskontextparameter `$thread` beinhaltet dabei das Token des Threads, der das Ereignis auslöste. Gleichzeitig werden alle Threads des Zielsystems – spezifiziert durch das Token `a_` für das Gesamtsystem – angehalten. Der Ausdruck „alle Threads" bezieht sich dabei auf die momentane Sicht desjenigen Werkzeugs, das die Ereignis/Aktions-Relation definierte.

Das Beispiel zeigt auch eine weitere wichtige Eigenschaft von OMIS auf: die Ortstransparenz. Sie bedeutet letztendlich, daß die Werkzeuge bei der Formulierung von Anfragen die Verteilung der überwachten Objekte auf die Rechnerknoten in keiner Weise berücksichtigen müssen. Im Beispiel spielt es also keine Rolle, ob die Prozesse auf demselben Knoten liegen oder beliebig im System verteilt sind. Auch die Expansion von Token, wie sie z.B. bei dem Dienst `thread_stop([a_])` ausgenutzt wird, setzt diese Ortstransparenz voraus.

Ereignis/Aktions-Relationen werden von Werkzeugen durch den Aufruf der Bibliotheksfunktion `omis_request` in Form einer Zeichenkette als Anfragen an das Monitorsystem gesendet. Die Funktion kann dabei entweder blockierend auf eine Antwort warten oder Antworten asynchron durch Aufruf einer Rückruffunktion zurückmelden. Ersteres ist nur bei unbedingten Anfragen sinnvoll, die eine leere Ereignisdefinition beinhalten, da bedingte Anfragen in einer unbestimmten Anzahl von Antworten resultieren.

3.2.1.2 Das verteilte Monitorsystem OCM

Das Monitorsystem OCM (*OMIS Compliant Monitoring system*) stellt eine Implementierung der OMIS-Schnittstelle für PVM-Programme auf Netzen von Arbeitsplatzrechnern bereit [270, 269, 147, 146]. Eine Portierung auf das Programmiermodell MPI ist in einer ersten Version ebenfalls verfügbar [21, 23, 22], Anpassungen an Metacomputing-Umgebungen sind geplant [26].

Der OCM ist gemäß des in § 2.1.3.1 definierten Architekturmodells für Laufzeit-Werkzeuge strukturiert. Er besitzt also pro Knoten des überwachten Systems eine lokale Monitorkomponente, sowie eine übergeordnete zentrale Komponente, die mit dem eigentlichen Werkzeug kommuniziert. Die lokalen Komponenten sind als jeweils ein Monitorprozeß realisiert, der als unabhängiger Server fungiert, also eingehende Aufträge ohne Wissen über das Gesamtsystem oder andere lokale Monitorprozesse abarbeiten kann. Dadurch wurde die Entwicklungskomplexität des Systems erheblich reduziert.

Ein zentraler Prozeß, die *Node Distribution Unit* NDU, ist für sämtliche globalen Aspekte des Monitorings sowie die Koordination der lokalen Monitorprozesse verantwortlich. Die NDU analysiert die Ereignis/Aktions-Relationen in den eingehenden Anfragen der Werkzeuge und bestimmt, auf welchen Rechnerknoten die in der Anfrage spezifizierten Objekte lokal überwacht werden können. Die Anfrage wird dann entsprechend dieser Information in Unteranfragen aufgeteilt, die an die lokalen Monitorprozesse versandt werden. Gegebenenfalls

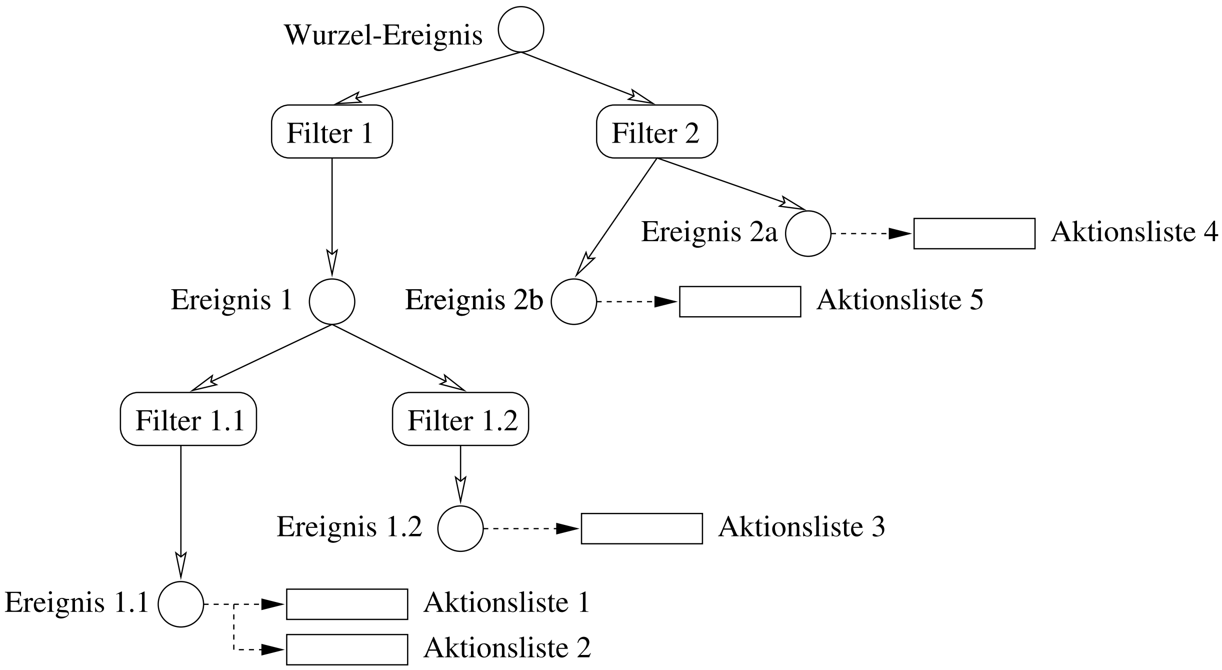


Abbildung 3.6: Ereignis- und Filterhierarchie im OCM

werden diese Unteranfragen um spezielle Aktionen und Ereignisdefinitionen erweitert, die die Koordination der lokalen Prozesse steuern (siehe § 6.2.1). Die NDU hat auch die Aufgabe, die zurückgelieferten lokalen Ergebnisse zu einer Gesamtantwort zu kombinieren und diese an das Werkzeug zurückzusenden, sobald sie vollständig ist. Die notwendige Kommunikation zwischen den Werkzeugen und der NDU wird für die Werkzeuge transparent durch eine Bibliothek durchgeführt, die zu diesen gebunden wird.

Der lokale Monitorprozeß nutzt die Betriebssystemschnittstellen `ptrace` und `/proc` zur Überwachung und Manipulation der auf seinem Rechnerknoten befindlichen Zielprozesse. Seine Arbeitsweise ist dabei ereignisorientiert:

- Wird eine unbedingte Anfrage ohne Ereignisdefinition empfangen, so wird die Aktionsliste der Anfrage zur späteren Bearbeitung an eine Warteschlange angefügt.

- Beim Empfang einer bedingten Anfrage wird zunächst eine zur Ereignisdefinition gehörige Funktion ausgeführt, die die notwendige Instrumentierung der zu überwachenden Objekte vornimmt bzw. eine schon vorhandene Instrumentierung aktiviert. Diese Instrumentierung meldet das Eintreten des speziellen Ereignisses – z.B. das Erreichen der Adresse 1234 in einem Prozeß – durch das Auslösen eines anderen, allgemeineren Ereignisses, z.B. eines Signals. Bei dessen Erkennung muß dann also mit Hilfe eines Filters überprüft werden, ob das spezielle Ereignis vorliegt. Da das Schema rekursiv angewandt wird, – z.B. könnte das Ereignis „Beginn einer Kommunikation" wiederum auf das Ereignis „Erreichen der Adresse 1234" zurückgeführt werden – entsteht ein Baum von Ereignissen und Filtern, an dessen Wurzel das vollkommen unspezifische Ereignis „etwas ist passiert" steht. An jeden Ereignis-Knoten innerhalb dieses Baumes werden die jeweils auszuführenden Aktionslisten angehängt (siehe Abb. 3.6).

- Beim Auftreten eines Ereignisses wird der Filterbaum gemäß dem Algorithmus in Abb. 3.7

```
process_event(Event_node node)
{
    Für alle Filter, die zum Ereignisknoten node gehören:
        Werte Filter aus
        Für jeden vom Filter zurückgelieferten Ereignisknoten n:
            process_event(n)
    Für alle Aktionslisten, die zum Ereignisknoten node gehören:
        Ersetze die Ereigniskontextparameter der Aktionsliste durch die
            aktuellen Parameterwerte
        Füge die Aktionsliste an die Aktionswarteschlange an
}
```

Abbildung 3.7: Algorithmus zur Ereignisabarbeitung im OCM

durchlaufen. Um der Ereigniserkennung und -bearbeitung Priorität vor der Aktionsausführung einzuräumen, werden die Aktionslisten beim Baumdurchlauf nicht sofort ausgeführt, sondern zunächst in der Warteschlange gespeichert.

- Ist die Aktions-Warteschlange nicht leer, wird die erste Aktionsliste aus ihr entnommen und ausgeführt.

Aus Effizienzgründen ist dieses relativ einfache Schema der Ereigniserkennung und Aktionsausführung durch den lokalen Monitorprozeß jedoch noch unzureichend. Betrachten wir als Beispiel die folgende Anfrage zum Zweck der Leistungsmessung:

```
thread_has_started_lib_call([p_1],"pvm_recv") :
    pa_integrator_local_increment(pa_il_1, 1, $time);
thread_has_ended_lib_call([p_1],"pvm_recv") :
    pa_integrator_local_increment(pa_il_1, -1, $time);
```

Der Dienst `pa_integrator_local_increment` ist Bestandteil einer OMIS-Erweiterung zur Leistungsanalyse [24]. Im Beispiel aktualisiert er den integrierenden Zähler `pa_il_1` so, daß die Gesamtverweilzeit von Prozeß `p_1` in der Funktion `pvm_recv` berechnet wird. Falls Aktionen nur vom lokalen Monitorprozeß ausgeführt werden könnten, müßte bei jedem Aufruf von `pvm_recv` durch Prozeß `p_1` zweimal ein Prozeßwechsel zum Monitorprozeß und wieder zurück erfolgen, nur um eine Zählervariable zu aktualisieren. Dies hätte eine nicht vertretbare Rückwirkung auf das Zielprogramm zur Folge, die das Meßergebnis unbrauchbar machen würde.

Der OCM kann daher die Ausführung von Aktionslisten bedingter Anfragen auch vollständig im Kontext des Zielprozesses durchführen, in dem das auslösende Ereignis stattfindet. Jeder Dienst besitzt dazu ein Attribut, das bei dessen Registrierung vom Programmierer definiert werden muß und angibt, ob der Dienst für diesen Ausführungsmodus geeignet ist. Bei Aktionen kann dieses Attribut sinnvoll nur dann gesetzt werden, wenn die Aktion kein Ergebnis an das Werkzeug zurückgibt, da das Senden des Ergebnisses wieder Prozeßwechsel verursachen würde. In Frage kommen daher in erster Linie Manipulationsdienste an Monitorobjekten, wie dem integrierenden Zähler des Beispiels.

Ereignisse, die im Kontext eines Zielprozesses erkannt werden können – dies sind in erster Linie Beginn und Ende von Bibliotheksaufrufen – werden im OCM mittels *Binary Wrapping*

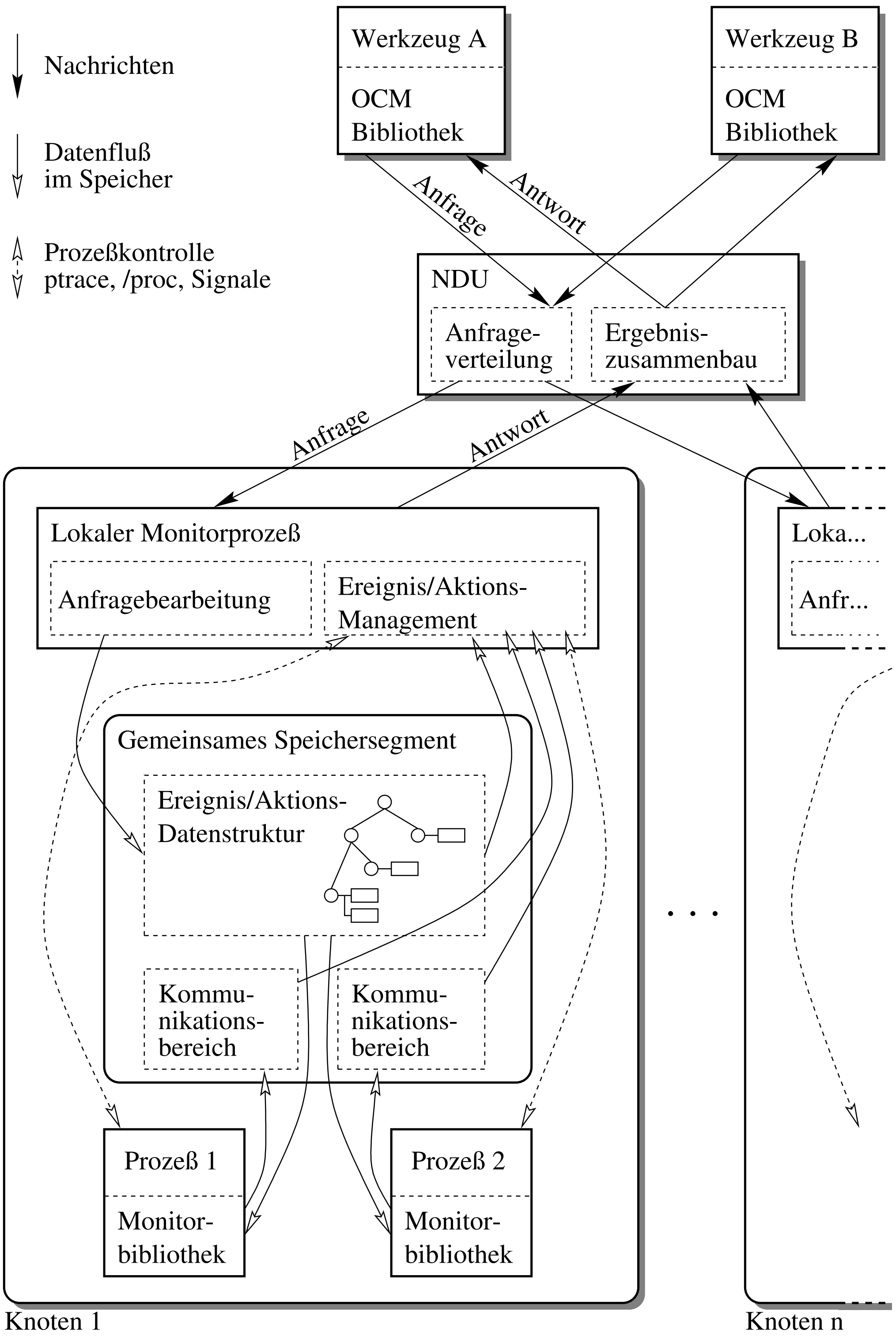


Abbildung 3.8: Detailstruktur des verteilten Monitorsystems OCM

(siehe § 2.1.3.2) überwacht. Jede einzelne Instrumentierung kann durch einen Eintrag in das entsprechende Element eines im Zielprozeß gespeicherten Zeigerfeldes (*Hook Array*) aktiviert werden. Diese Aktivierung erfolgt bei der Definition einer bedingten Anfrage durch den Monitorprozeß. Jeder aktivierten Instrumentierung wird dabei ein eigener Filterbaum zugeordnet, wie er in Abb. 3.6 dargestellt ist. Dieser wird vom Monitorprozeß aufgebaut und ist in einem gemeinsam genutzten Speichersegment abgelegt. Der Zeiger auf den Baum wird dann im *Hook Array* abgelegt. Der Monitorprozeß nutzt dazu Betriebssystemdienste zum Schreiben auf den Adreßraum des Zielprozesses.

Wird später der Instrumentierungscode vom Zielprozeß ausgeführt, prüft er beim Durchlauf durch den Baum zunächst, ob alle auszuführenden Aktionen tatsächlich im Zielprozeß abgearbeitet werden können. Falls ja, werden diese direkt ausgeführt. Anderenfalls, erfolgt ein expliziter Kontextwechsel zum Monitorprozeß. Die notwendige Information über die zu bearbeitenden Aktionslisten werden in einem gesonderten Kommunikationsbereich des gemeinsamen Speichersegments übergeben. Der von Monitor- und Zielprozessen gemeinsam genutzte Speicher dient auch zur Ablage von Monitorobjekten, z.B. Zählern, Integrierern oder Ereignispuffern. Diese können daher durch Aktionen im Zielprozeß manipuliert und später durch andere Aktionen vom Monitorprozeß ausgelesen werden, der die Informationen über die NDU an die Werkzeuge weitergibt.

Abb. 3.8 zeigt die beschriebene Struktur des OCM im Detail.

3.2.2 `DETOP`: Quellsprach-Debugging

Der Debugger `DETOP` stellt dem Benutzer die üblichen Funktionen eines sequentiellen Debuggers zur Verfügung, jedoch erweitert auf die Überwachung und Steuerung mehrerer Zielprozesse. Insbesondere erlaubt `DETOP` das unmittelbare Anhalten und Weiterführen von Prozessen, das Abfangen von Signalen und Ausnahmen, das Setzen von Haltepunkten auf Quellprogrammzeilen und Prozeduren, die Einzelschritt-Abarbeitung des Quellcodes, die Anzeige des momentanen Prozeduraufrufkellers, sowie die Ausgabe und Modifikation der Inhalte von Quellcodevariablen eines überwachten Prozesses. Diese Funktionen können jeweils auf eine beliebige Menge von Prozessen gleichzeitig angewandt werden.

Als zusätzliche Erweiterung für parallele Programme erlaubt `DETOP` beim Erreichen eines Haltepunktes das globale, asynchrone Anhalten des gesamten Zielprogramms, d.h. aller seiner Prozesse, wodurch dessen Globalzustand sozusagen eingefroren wird. Darüber hinaus unterstützt `DETOP` die dynamische Prozeßerzeugung durch PVM. PVM-Programme starten üblicherweise zunächst mit einem einzigen Prozeß, der dann alle anderen Prozesse der Anwendung zur Laufzeit erzeugt. Dementsprechend erzeugt und startet `DETOP` beim Aufruf zunächst ebenfalls nur diesen initialen Prozeß, überwacht jedoch die dynamische Erzeugung neuer Prozesse. Paßt ein derartiger Prozeß auf ein vom Benutzer vorgegebenes Muster, koppelt sich `DETOP` selbständig an diesen an und stoppt ihn vor der Ausführung seiner ersten Anweisung. Der neue Prozeß kann somit ebenfalls von Anfang an in seinem Ablauf überwacht werden.

Wie die meisten parallelen Debugger – mit Ausnahme der neuesten Version von `totalview` [48] – besitzt `DETOP` keine weitergehenden speziellen Funktionen um die Interaktionen zwischen Prozessen bzw. den Zustand des Kommunikationssystems überwachen zu können, da hierdurch eine Abhängigkeit von der Implementierung dieses Kommunikationssystems entstünde. Die einzige PVM-spezifische Information, die von `DETOP` verwendet wird, sind die PVM-Task-Identifikatoren der überwachten Prozesse. Diese werden als Zusatzinformation an der Benutzerschnittstelle verwendet, um die Identifikation von Prozessen zu erleichtern.

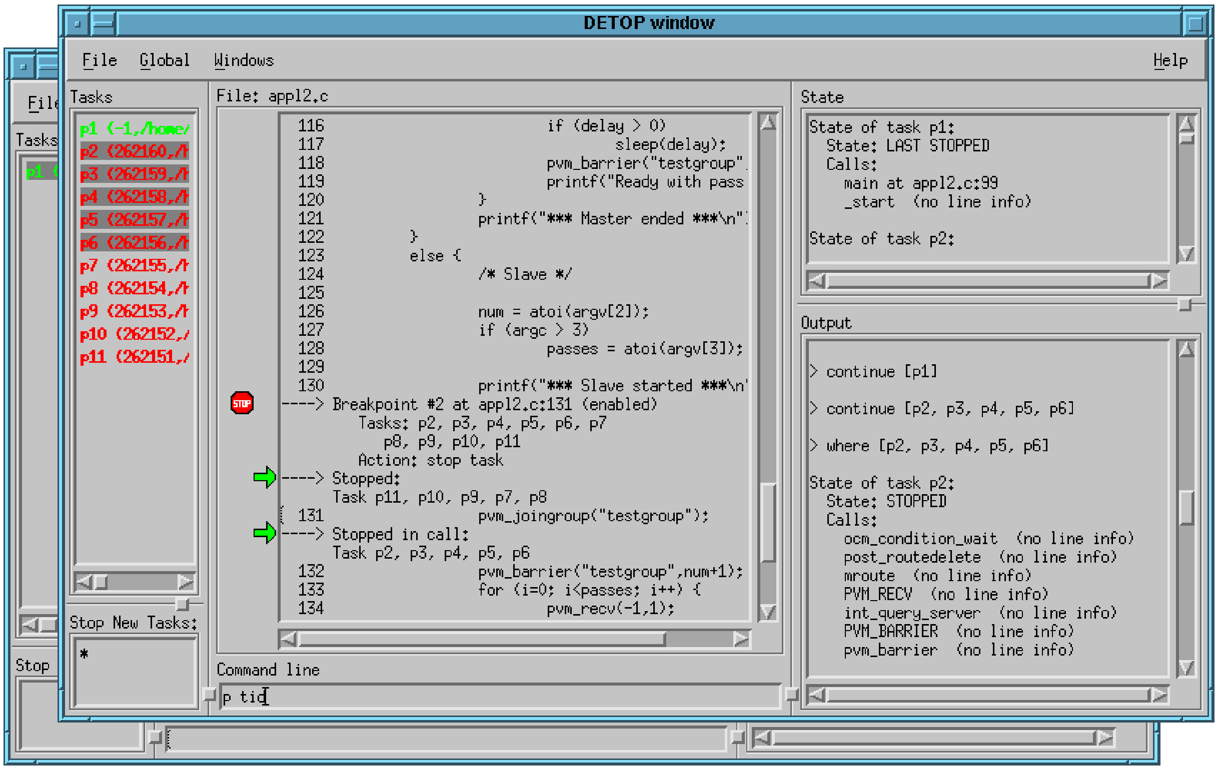


Abbildung 3.9: Bedienoberfläche des Debuggers DETOP

DETOP besitzt eine graphische Bedienoberfläche, die aus beliebig vielen Debuggerfenstern bestehen kann (siehe Abb. 3.9). Jedem Fenster kann eine beliebige Prozeßmenge zugeordnet werden, die in der Prozeßliste im linken Teil des Fensters angezeigt wird. Diese Liste wird unter anderem dazu verwendet, die Prozesse auszuwählen, auf die ein gegebenes Kommando wirken soll. Die Kommandoeingabe ist dabei über kontextsensitive *Pop-up*-Menüs oder auch textuell über eine Kommandozeile möglich. Ausgaben erfolgen in den beiden rechten Teilen des Debuggerfensters. Der untere Teil stellt ein einfaches Ausgabeprotokoll dar, bei dem jedes neue Ergebnis des Debuggers am Ende angefügt wird. Im Gegensatz dazu zeigt der obere Teil immer Information über den gerade aktuellen Programmzustand an. Genauer stellt dieser Teil die Prozeduraufrufketten aller Prozesse sowie die Inhalte ausgewählter Variablen dar, wobei die Daten beim Anhalten des jeweiligen Prozesses am Platz aktualisiert werden. Zusätzlich zu diesen speziellen Ausgabefeldern werden im Programmlisting in der Mitte des Debuggerfensters Markierungen eingefügt, die die aktuelle Position der überwachten Prozesse sowie die aktuell spezifizierten Haltepunkte anzeigen.

Die aktuelle Version von DETOP ist auf Basis des OCM realisiert. DETOP benutzt dabei fast ausschließlich die Dienste der OMIS-Basisschnittstelle; lediglich für den Einzelschrittbetrieb auf Quellsprachebene wird aus Effizienzgründen ein spezieller Ereignisdienst verwendet, der von einer DETOP-spezifischen Werkzeug-Erweiterung zur Verfügung gestellt wird. Eine zentrale Transformationsschicht zwischen der Bedienoberfläche und der OMIS-Schnittstelle verwaltet die Symboltabellen aller Zielprozesse und setzt die Debuggerkommandos sowie deren Parameter von der Hochsprachebene auf Maschinensprachniveau um, z.B. Variablennamen in Adressen. Umgekehrt werden Resultate des OCM vor der Darstellung auf der Bedienoberfläche wieder soweit erforderlich vom Maschinensprach- auf das Hochsprachniveau transformiert.

3.2.3 `CoCheck`: Erstellung konsistenter Sicherungspunkte

3.2.3.1 Prinzipielle Funktionsweise

`CoCheck` ist ein Werkzeug zur Erstellung konsistenter Sicherungspunkte, das unter anderem auf PVM-Programme anwendbar ist [225, 139, 218]. Die Basis des Werkzeugs bildet die Sicherungspunktbibliothek `Condor`, die den Zustand einzelner, unabhängiger Prozesse sichern und wieder restaurieren kann. Dazu werden im wesentlichen das Datensegment des Prozesses sowie zusätzliche Information abgespeichert, die aus der Überwachung von Systemaufrufen gewonnen wird, z.B. Information über offene Dateien.

Für parallele Programme muß zusätzlich darauf geachtet werden, daß stets ein konsistenter Globalzustand gesichert wird. Dazu müssen zum einen die Zeitpunkte, zu denen die Sicherung der einzelnen Prozesse erfolgt, geeignet synchronisiert werden. Zum anderen muß im allgemeinen auch der Zustand des Kommunikationssystems, z.B. die noch in Übertragung befindlichen Nachrichten, mitgesichert werden. Eine genauere Anforderungsdiskussion für die konsistente Sicherungspunkterstellung findet sich in [235].

`CoCheck` realisiert eine koordinierte Sicherungspunkterstellung [235], bei der sich alle beteiligten Prozesse durch ein Konsistenzprotokoll auf einen Zeitpunkt für die lokale Sicherung verständigen. Dieses Protokoll wird durch einen zentralen Manager ausgelöst und überwacht. Es stellt gleichzeitig sicher, daß sich keine Nachrichten mehr in Übertragung befinden. Der Zustand des Kommunikationssystems muß daher bei `CoCheck` nicht explizit gesichert werden.

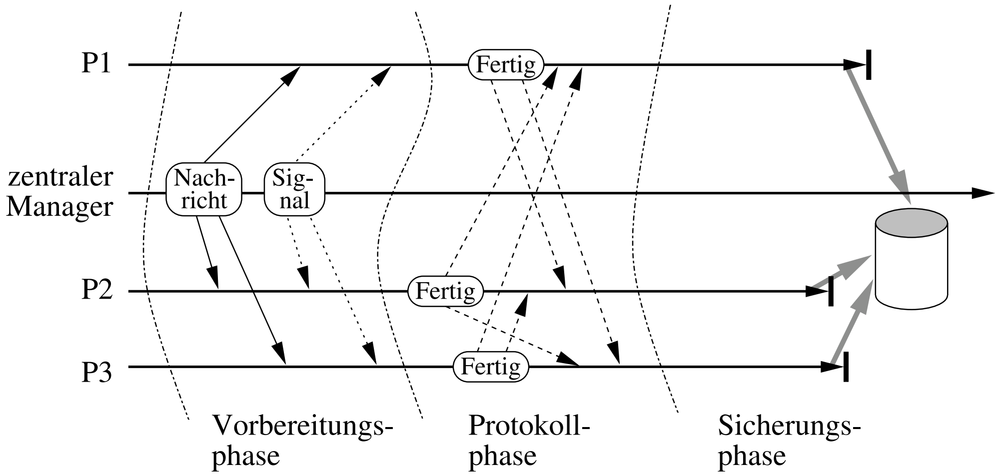


Abbildung 3.10: Erstellung eines globalen Sicherungspunkts mit `CoCheck`

Erstellung eines Sicherungspunkts. Der Ablauf der Sicherungspunkterstellung ist dann wie in Abb. 3.10 dargestellt: Der zentrale Manager benachrichtigt zunächst jeden beteiligten Prozeß, daß ein Sicherungspunkt erstellt werden soll. Aus Gründen, die in [235] genauer erläutert werden, erfolgt diese Benachrichtigung sowohl durch eine PVM-Nachricht als auch durch ein UNIX-Signal. Sobald ein Prozeß die Benachrichtigung erhält, tritt er in die Protokollphase ein. Dabei wird zunächst eine spezielle „Fertig"-Nachricht an alle anderen beteiligten Prozesse versandt. Anschließend werden solange Nachrichten vom Netzwerk gelesen und lokal im Adreßraum des Prozesses gespeichert, bis von jedem anderen Prozeß die „Fertig"-Nachricht empfangen wurde. Zu diesem Zeitpunkt ist sichergestellt, daß der Prozeß alle Nachrichten, die vor Eintritt in die Protokollphase an ihn abgeschickt wurden, in seinem Adreßraum gespeichert hat, und bis zum Abschluß der Sicherungspunkterstellung keine weiteren Nachrichten an ihn versendet werden. Nun meldet sich der Prozeß beim PVM-System ab und kann als jetzt vollständig isolierter Prozeß mittels `Condor` gesichert werden. Die Weiterführung eines Prozesses nach erfolgter Sicherung entspricht dann exakt der bei einem Wiederanlauf, wie sie im folgenden beschrieben wird.

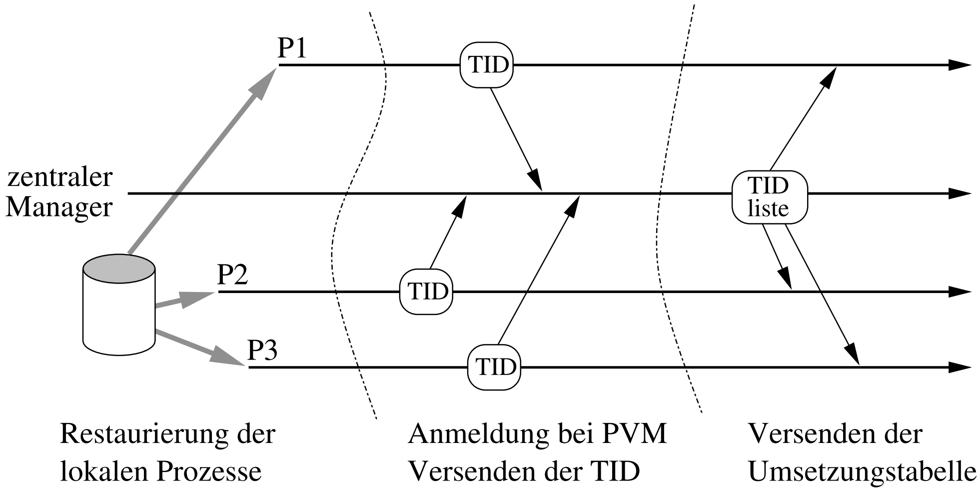


Abbildung 3.11: Wiederanlauf eines gesicherten, parallelen Programms

Wiederanlauf eines Programms. Der Wiederanlauf eines gesicherten, parallelen Programms ist in Abb. 3.11 dargestellt. Er beginnt mit der Restaurierung der einzelnen Prozesse mit Hilfe der `Condor`-Mechanismen. Jeder Prozeß meldet sich anschließend wieder bei der virtuellen Maschine von PVM an. Er erhält dadurch einen neuen Task-Identifikator (TID) vom PVM-System. Dieser wird zusammen mit dem ursprünglichen Identifikator, der noch im Adreßraum des Prozesses gespeichert ist, an den zentralen Manager gesandt, der eine Umsetzungstabelle zwischen alten und neuen TIDs erstellt und an alle Prozesse versendet. Bei der anschließenden Weiterführung der Prozesse sorgt eine durch `CoCheck` vorgenommene Instrumentierung der PVM-Aufrufe für die Erledigung der beiden folgenden Aufgaben:

1. Da der Prozeß intern noch seinen alten TID gespeichert hat, für PVM nun jedoch einen neuen TID besitzt, muß bei allen PVM-Aufrufen, die einen TID als Argument erfordern oder als Ergebnis zurückliefern, eine entsprechende Übersetzung mit Hilfe der Umsetzungstabelle durchgeführt werden.

2. Bei Empfangsaufrufen muß zunächst geprüft werden, ob die zu empfangene Nachricht nicht bereits während des Konsistenzprotokolls bei der Sicherungspunkterstellung empfangen und zwischengespeichert wurde. Falls dies der Fall ist, wird die gespeicherte Nachricht zurückgegeben, anderenfalls wird der Aufruf an PVM weitergereicht.

3.2.3.2 Implementierung

Der skizzierte Mechanismus ist in der Dissertation von J. Trinitis [235] als Monitor-Erweiterung in das Monitorsystem OCM integriert worden. Die Motivation für dieses Vorgehen war zum einen, die bereits in der Schnittstellenspezifikation [150] definierten Dienste zur Sicherungspunkterstellung für alle Werkzeuge zur Verfügung zu stellen, zum anderen aber auch die Interoperabilität mit `DETOP`. Durch die Integration wurden die strukturellen Konflikte zwischen `CoCheck` und OCM beseitigt; zudem können Ereignisse, die Beginn und Ende von Sicherungspunkterstellung und Wiederanlaufphase anzeigen, in einfacher Weise realisiert werden.

Nach unserem jetzigen Erkenntnisstand wäre für das letztere Ziel eine vollständige Integration von `CoCheck` in den OCM nicht notwendig gewesen. Wie das Beispiel `Codex` noch zeigen wird, können in geeigneten Fällen strukturelle Konflikte auch ohne ein gemeinsames Monitorsystem gelöst werden. Ebenso ist das Verfügbarmachen von Ereignissen in `CoCheck` für andere Werkzeuge, wie es für die Verhaltensintegration benötigt wird, auch durch die in § 2.3.4.2 skizzierte Technik zu erreichen. Bei genauerer Betrachtung liegt der Arbeit von Trinitis auch nicht exakt der in § 2.2.2 definierte Interoperabilitätsbegriff zugrunde, bei dem gleichberechtigte, im wesentlichen unabhängige Werkzeuge betrachtet werden. Vielmehr basiert bei Trinitis die Zusammenarbeit von `DETOP` und `CoCheck` auf einem *Master-Slave*-Verhältnis, bei dem `DETOP` die Funktionen und die Schnittstelle von `CoCheck` explizit kennt und nutzt. Da die Implementierung von `CoCheck` nun aber bereits so vorliegt, bleiben wir der Prämisse dieser Arbeit treu, möglichst wenig an vorhandenen Werkzeugen zu ändern, und übernehmen die existierende Realisierung.

Die `CoCheck`-Erweiterung des OCM bietet drei wesentliche Aktionen:

1. `proc_checkpoint` erstellt einen globalen Sicherungspunkt aller als Parameter angegebenen Prozesse.

2. `proc_restart` realisiert den Wiederanlauf eines Sicherungspunkts, wobei die Prozesse neu erzeugt werden.

3. `proc_reset` setzt im Prinzip die als Parameter übergebenen Prozesse auf den im spezifizierten Sicherungspunkt gespeicherten Zustand zurück. Da ein Zurücksetzen existierender Prozesse mit `Condor` nicht realisierbar ist, werden in Wirklichkeit die übergebenen Prozesse terminiert und bei der Restaurierung neue Prozesse erzeugt, die jedoch wieder die ursprünglichen Prozeßtoken erhalten.

Daneben sind drei Ereignisdienste realisiert: Die Dienste `proc_will_be_checkpointed` und `proc_has_been_checkpointed` zeigen den Beginn und die Beendigung der Sicherungspunkterstellung an, `proc_has_been_restarted` signalisiert das Ende der Restaurierung.

Die bei der Beschreibung der Funktionsweise von `CoCheck` genannten Aktionen, die am Beginn und Ende von PVM-Aufrufen durchgeführt werden müssen, werden in der vorliegenden Implementierung nicht mit Hilfe der Ereignis/Aktions-Verwaltung des OCM abgewickelt, sondern durch fest definierten Code, der mittels *Binary Wrapping* um diese Aufrufe gelegt wird (vgl. § 2.1.3.2). Für die nachfolgende Diskussion sind folgende Aktivitäten dieses Instrumentierungscodes relevant:

- Er modifiziert die Ein- und Ausgabeparameter der PVM-Aufrufe, insbesondere die TIDs.

- Er verwendet PVM-Aufrufe für interne Aufgaben, z.B. zum Versenden und Empfangen von Nachrichten bei der Abarbeitung des Konsistenzprotokolls.

- Er benutzt UNIX-Signale, z.B. zum Einleiten des Protokolls und der lokalen Sicherungspunkterstellung mittels `Condor`.

3.2.4 VISTOP: Zustandsorientierte Programmablauf-Visualisierung

Das Visualisierungswerkzeug VISTOP basiert seit der ersten Realisierung [33] auf der Grundidee, nicht einfach nur Ereignisspuren graphisch darzustellen, sondern konsistente, globale Zustände. Gegenüber anderen Visualisierern, wie z.B. xpvm [88, 116, 123], enthält VISTOP daher eine zusätzliche Komponente, die das parallele Laufzeitsystem modelliert, um dessen Zustand, etwa den Inhalt von Nachrichtenwarteschlangen, die Zusammensetzung von Prozeßgruppen oder den Status von Synchronisationsobjekten berechnen und darstellen zu können. Eine erste Version von VISTOP für PVM arbeitete zunächst offline auf der Basis von SDDF-Ereignisspuren [8, 240], wie sie von xpvm erzeugt werden [206]. Nach Verfügbarwerden des OCM wurde VISTOP um die Fähigkeit des Online-Betriebs erweitert. Abbildung 3.12 zeigt Struktur und Funktion dieses Werkzeugs.

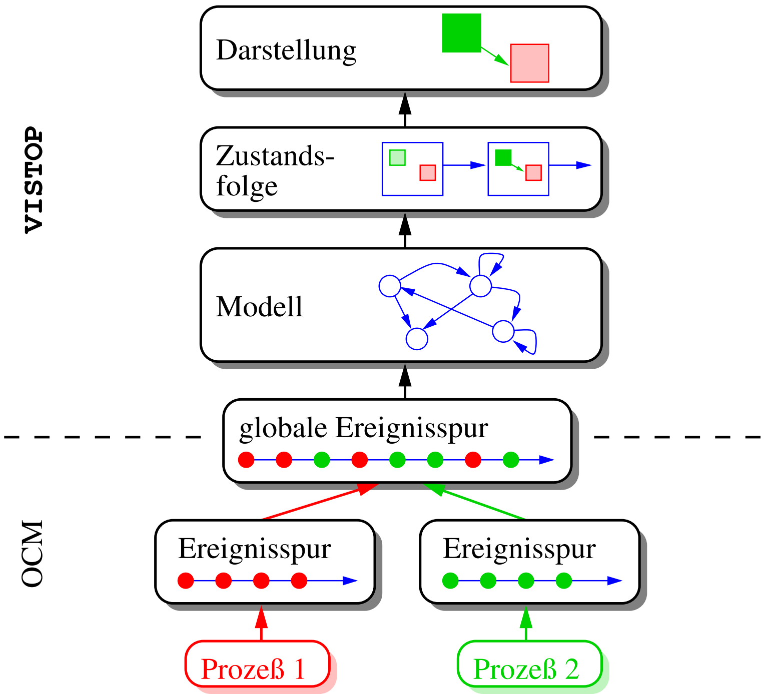


Abbildung 3.12: Struktur des Visualisierers VISTOP

Die Grundlage der Datenerfassung bildet eine Werkzeug-Erweiterung des OCM, die die Aufzeichnung beliebiger Ereignisdaten ermöglicht [249, 250]. Die Erweiterung stellt drei Aktionen zur Verfügung, mit deren Hilfe ein virtuell globaler Ereignispuffer erzeugt, Information an diesen Puffer angefügt, sowie der gesamte Puffer an das Werkzeug ausgegeben werden kann. Der Puffer ist dabei aus Effizienzgründen verteilt realisiert, wobei auf jedem Knoten der lokale Teil des Puffers im gemeinsamen Speichersegment des OCM angelegt wird. Das Eintragen von Information kann daher lokal durch Aktionen erfolgen, die im Kontext der Anwendungsprozesse ausgeführt werden können. Beim Auslesen werden die Ereignisinformationen der lokalen Puffer entsprechend ihrer Zeitstempel zu einer globalen Ereignisspur zusammengemischt. Eine (näherungsweise) Synchronisation der lokalen Uhren kann durch einen weiteren Dienst der OCM-Erweiterung regelmäßig angefordert werden. Sie ist zumindest auf Netzen von Arbeitsplatzrechnern genau genug, um beim Zusammenmischen eine Reihenfolge der Ereignisse zu erhalten, die mit ihrer kausalen Ordnung [214] verträglich ist [250].

VISTOP zeichnet mit Hilfe dieses Mechanismus und der beiden OMIS Ereignisdienste

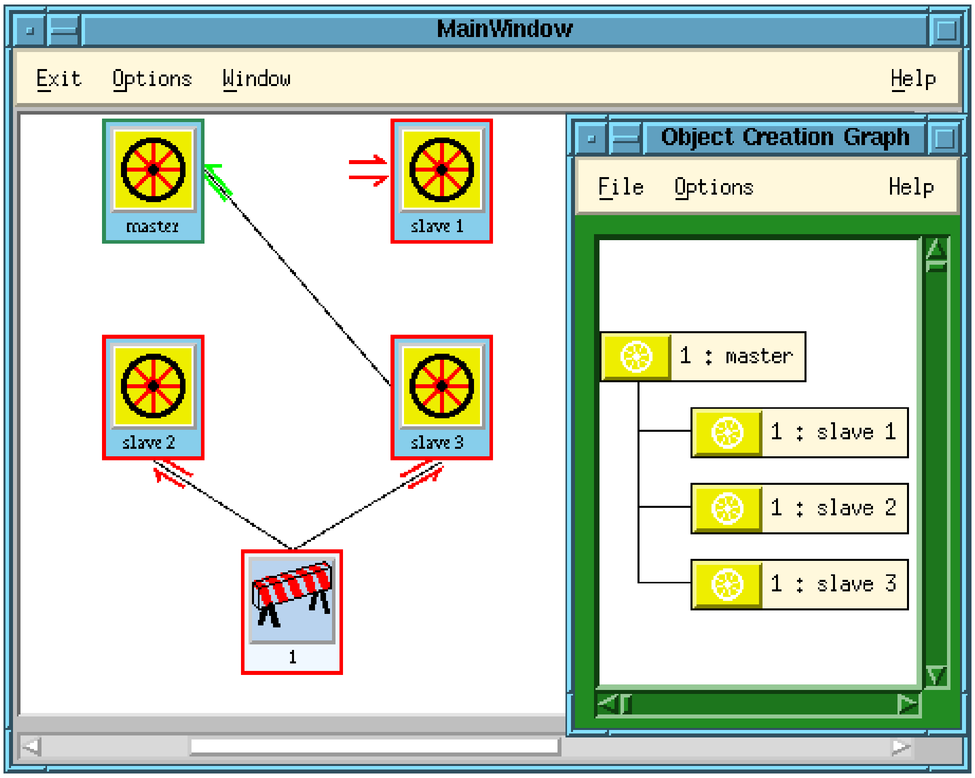


Abbildung 3.13: Visualisierung eines globalen Programmzustands durch `VISTOP`

`thread_has_started_libcall` und `thread_has_ended_libcall` die Parameter beim Aufruf bzw. die Ergebnisse bei der Rückkehr von PVM-Funktionen auf. Der so gewonnene Ereignisstrom wird periodisch ausgelesen und an die Modellierungskomponente weitergereicht, die mit Hilfe eines Modells der Funktionsweise von PVM [206] aus diesem Strom eine Folge globaler Zustände errechnet. Dieses Modell ist im wesentlichen ein endlicher Automat, der Ereignisse des Zielprogramms als Eingabezeichen akzeptiert. Die Bearbeitungsreihenfolge der Ereignisse muß dabei so gewählt werden, daß ein Ereignis e erst dann abgearbeitet wird, wenn alle kausal vor e liegenden Ereignisse behandelt wurden und e in dem aktuell modellierten Zustand auftreten kann [255].

Ein spezielles Problem der PVM-Version von `VISTOP` ist die Modellierung von Nachrichtenwarteschlangen und die Visualisierung von Empfangsoperationen. Letztere sollte so erfolgen, daß, sofern eine empfangbare Nachricht vorliegt, während des Empfangens durch einen Pfeil angedeutet wird, von welchem Prozeß diese gesendet wurde (siehe Abb. 3.13: Prozeß `master` empfängt hier von Prozeß `slave 3`). Anderenfalls wird eine Wartebedingung symbolisiert (siehe Prozeß `slave 1` in Abb. 3.13). Problematisch sind hier nun Empfangsoperationen mit Sender-*Wild-Card*, da hier der tatsächliche Sender der Nachricht erst aus dem Ereignis bei Beendigung der Operation hervorgeht. Bei der Offline-Version von `VISTOP` wurde in solchen Situationen eine Vorausschau im Ereignisstrom durchgeführt. Diese Lösung mußte für den Online-Betrieb verworfen werden. Zwar ist die Vorausschau nur dann nötig, wenn der Empfangsaufruf nicht blockiert, so daß die kurze Wartezeit bis zum Eintreffen des Endeereignisses für die ohnehin nur zeitverzögert arbeitende Visualisierung unerheblich wäre, beim gleichzeitigen Einsatz eines anderen Werkzeugs könnte jedoch ein Prozeß während der Empfangsoperation von diesem angehalten werden. Obwohl der Prozeß dann bereits die Empfangsoperation durchführt, würde dies noch nicht von VISTOP angezeigt, die Visualisierung wäre daher nicht mit dem tatsächlichen Zustand des Zielprogramms konsistent. Daher wurde in Form einer PVM-spezifischen Monitor-Erweiterung ein zusätzlicher Ereignisdienst in den OCM aufgenommen, der den Eintrag einer Nachricht in die Empfangs-Nachrichtenwarteschlange eines Prozesses anzeigt. Dieser Dienst wurde durch die Überwachung interner Funktionsaufrufe innerhalb der PVM-Bibliothek realisiert. Mit Hilfe der nun vorhandenen Ereignisse beim Eintrag und beim Entfernen von Nachrichten aus der Warteschlange, sowie deren bekannter FIFO-Semantik kann VISTOP nun exakt die Reihenfolge der gespeicherten Nachrichten modellieren. Somit kann bereits bei Beginn einer Empfangsoperation bestimmt werden, welche Nachricht von dieser empfangen werden wird. Wir werden später sehen, daß sich dadurch nicht unerhebliche Probleme bei der Zusammenarbeit mit Codex ergeben.

Aus der von der Modellierungsschicht erzeugten Folge globaler Zustände kann der Benutzer mit Hilfe von Steuertasten und eines Rollbalkens einen Zustand auswählen, der im Detail graphisch auf dem Bildschirm dargestellt wird. Wird dabei über den letzten gespeicherten Zustand hinausgegangen, fordert die Modellierungsschicht neue Ereignisse vom OCM an und erweitert die gespeicherte Zustandsfolge.

3.2.5 `Codex`: Gesteuerte deterministische Ausführung

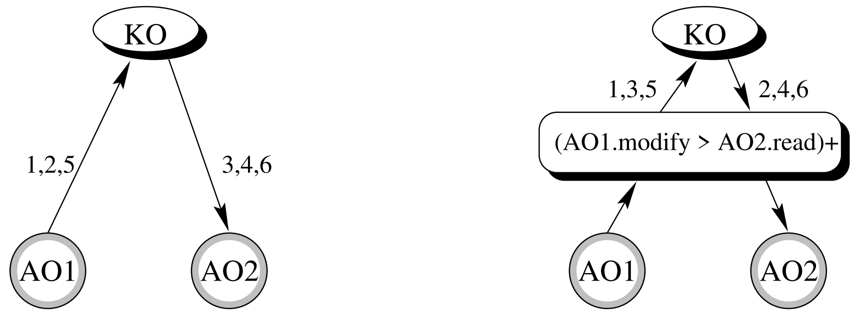
Das Werkzeug Codex unterstützt die deterministische Ablaufsteuerung paralleler Programme, wie sie in § 2.1.2.1 vorgestellt wurde. Es ist dabei nicht auf die deterministische Wiederholung eines vorher aufgezeichneten – aber aufgrund des Nichtdeterminismus natürlich zufälligen – Programmablaufs eingeschränkt, sondern bietet die Möglichkeit der vollständigen a-priori Steuerung. Dadurch können alle denkbaren Ablaufvarianten des Programms erzwungen werden, was unter anderem einen vollständigen, systematischen Test nichtdeterministischer paralleler Programme ermöglicht [75].

Die Codex zugrundeliegende Methodik basiert auf einem POEM genannten, abstrakten Modell paralleler Ausführungen. POEM ist objektbasiert und unterscheidet zwischen aktiven Ausführungsobjekten, z.B. Prozessen oder Threads, und passiven Kommunikationsobjekten, etwa Nachrichtenpuffer oder gemeinsame Variable. Ausführungsobjekte können auf Kommunikationsobjekte modifizierend und lesend zugreifen. Das Modell ist dabei abstrakt genug, um sowohl parallele Programme zu beschreiben, die auf Nachrichtenaustausch basieren, als auch solche, die gemeinsamen Speicher nutzen.

Um einen deterministischen Ablauf zu erreichen, wird für jedes Kommunikationsobjekt eines Programms die Sequenz der Zugriffe aus Sicht dieses Objekts betrachtet. Durch Kontrollmuster, die in ihrer Struktur regulären Ausdrücken ähneln, kann nun die Menge der möglichen Sequenzen eingeschränkt werden, so daß nur noch Abläufe zulässig sind, die eine auf das Kontrollmuster passende Zugriffssequenz ergeben. Als einfaches Beispiel sei die Situation in Abb. 3.14a gegeben, bei der zwei Ausführungsobjekte $AO1$ und $AO2$ auf das Kommunikationsobjekt KO zugreifen, wobei $AO1$ nur Schreib-, $AO2$ nur Lesezugriffe durchführt. Die Verzahnung dieser Zugriffe ist dabei nicht durch Synchronisation eingeschränkt, kann also beliebig und zufällig erfolgen. In der Abbildung ist eine mögliche Verzahnung von sechs Zugriffen durch die Zahlen angedeutet. Durch die Festlegung des Kontrollmusters

$$M(KO) : (AO1.modify \succ AO2.read)^+$$

wird erreicht, daß die Zugriffe der beiden Ausführungsobjekte in deterministischer Weise abwechselnd erfolgen (siehe Abb. 3.14b). Für eine eingehendere Beschreibung der Methodik wird



a) Ausgangssituation b) Steuerung durch Kontrollmuster

Abbildung 3.14: Steuerung der Zugriffsreihenfolge durch Kontrollmuster

auf die Dissertation von M. Oberhuber [169] verwiesen.

Die Implementierung des Werkzeugs für PVM nutzt zur Vereinfachung die speziellen Gegebenheiten bei PVM aus. Insbesondere sind die Kommunikationsobjekte in diesem Fall Nachrichtenwarteschlangen, die an den Empfängerprozeß gebunden sind. Die Semantik der Warteschlangen hat zur Folge, daß deren bisheriger Inhalt durch das Senden einer Nachricht nicht verlorengeht, sondern lediglich erweitert wird. Aus diesem Grund kann auf eine Überwachung und Steuerung der Sendeoperationen verzichtet werden; es genügt, die Empfangsaufrufe zu kontrollieren. Konkret bedeutet dies, daß auch bei einem durch `Codex` gesteuerten Ablauf Nachrichten in beliebiger Reihenfolge in eine Nachrichtenwarteschlange eingetragen werden können. Das Entfernen von Nachrichten beim Empfang erfolgt aber nicht mehr in der Reihenfolge des Eintrags, sondern ausschließlich aufgrund des spezifizierten Kontrollmusters.

Die notwendige Kontrolle wird durch eine Instrumentierung der PVM-Empfangsfunktionen mittels *Binary Wrapping* realisiert. `Codex` benutzt für diese Instrumentierung ein eigenes Hilfswerkzeug, basiert also im Gegensatz zu den anderen vorgestellten Werkzeugen nicht auf dem OCM. Der Instrumentierungscode prüft bei jedem Empfangsaufruf des Zielprogramms zunächst, ob dieser überhaupt Freiheitsgrade zuläßt, d.h. für Nachrichtenkennung oder Absender *Wild-Cards* übergeben wurden. Falls ja, wird bestimmt, welche Nachricht aufgrund des spezifizierten Kontrollmusters empfangen werden muß und diese Information dazu benutzt, die *Wild-Cards* durch konkrete Werte zu ersetzen. Der eigentliche Empfang der Nachricht erfolgt dann durch einen Aufruf der ursprünglichen PVM-Funktion mit den geänderten Parametern. Anschließend wird der Zustand des Kontrollmusters, das durch einen endlichen Automaten realisiert ist, entsprechend dem gerade durchgeführten Zugriff weitergeschaltet.

Der Instrumentierungscode kommuniziert zusätzlich intern mittels PVM sowohl mit dem entsprechenden Instrumentierungscode in anderen Zielprozessen, als auch mit der graphischen Bedienoberfläche, die in Java realisiert ist. Nach dem Start der PVM-Anwendung sendet diese alle vom Benutzer definierten Kontrollmuster an den Instrumentierungscode im Wurzelprozeß der Anwendung. Bei dynamischer Erzeugung neuer Prozesse werden die Kontrollmuster dann jeweils über eine analoge Nachricht an die Kindprozesse weitergegeben. Schließlich ist auch vor dem Versenden einer Nachricht durch die gesteuerte Anwendung die Übertragung einer `Codex`-internen Kontrollnachricht zwischen den kommunizierenden Prozessen erforderlich.

3.3 Verhalten der Werkzeugumgebung aus Sicht eines Anwenders

In den nachfolgenden drei Abschnitten werden wir auf drei verschiedenen Abstraktionsebenen das gewünschte Verhalten sowie die Realisierung der skizzierten Werkzeugumgebung untersuchen. In diesem Abschnitt wird zunächst die Ebene der Benutzerschnittstelle betrachtet, im nächsten die algorithmische Realisierung der benötigten Zusatzfunktionen. Daraus werden sich bereits erste, relativ allgemeine Anforderungen an eine Werkzeug-Middleware ergeben, also an eine Software-Schicht, die die Interaktion der Werkzeuge unterstützt. Der letzte Abschnitt, der die Realisierungsebene behandelt, wird schließlich in sehr konkreten Anforderungen resultieren, die sich aus der Forderung nach Interoperabilität ergeben. Dies ist insofern nicht verwunderlich, als Interoperabilität in erster Linie ein Problem der Implementierung, weniger der abstrakten Funktionalität von Werkzeugen ist. Der Schwerpunkt der Untersuchung wird auf allen Abstraktionsebenen auf den Beeinflussungen und Interaktionen der Werkzeuge liegen.

Aus Benutzersicht sollen zunächst die gewünschten Funktionen der Werkzeugumgebung definiert werden, ohne daß dabei bereits im Detail auf deren Realisierung Bezug genommen wird. Zusätzlich zu den schon vorhandenen Funktionen der Einzelwerkzeuge soll die Umgebung dem Benutzer lediglich eine neue Funktion „Rücksetzen“ anbieten, die das Zielprogramm auf den momentan von `VISTOP` dargestellten Zustand zurücksetzt. Die Entwurfsalternativen und -entscheidungen, die sich daraus und aus der vorgesehenen Nutzung der Werkzeuge ergeben, werden in den nachfolgenden Unterabschnitten diskutiert.

3.3.1 Start und Beendigung der Werkzeugumgebung

Zunächst stellt sich die Frage, wie die Umgebung vom Benutzer gestartet werden soll. Denkbar sind hier zwei Alternativen:

1. Die Werkzeuge können vom Anwender einzeln und erst bei Bedarf nacheinander gestartet werden.

2. Die Werkzeugumgebung wird immer als Ganzes gestartet.

Im vorliegenden Szenario ist ein unabhängiger Start der Werkzeuge nicht sinnvoll, da die Umgebung nur als Ganzes ihren Zweck erfüllen kann. Gründe dafür sind unter anderem:

- Das Rücksetzen ist nur auf einen Zustand möglich, der von `VISTOP` angezeigt wird. Der Visualisierer hat aber kein Wissen über Zustände, die vor seinem Start aufgetreten sind.

- Zudem werden für eine korrekte Modellierung des globalen Zustands die Ereignisspuren aller Zielprozesse vom Zeitpunkt ihrer Erzeugung an benötigt. Zwar könnte dies im Prinzip geändert werden, indem der Zustand des Modells beim Start des Visualisierers mit dem tatsächlichen globalen Zustand initialisiert wird, jedoch würde dafür interne Zustandsinformation von PVM benötigt, die nicht ohne weiteres zugänglich ist.

- Die Anwendung eines Werkzeugs zur deterministischen Ablaufsteuerung ist offensichtlich nur dann sinnvoll, wenn dieses den Programmablauf von Anfang an kontrolliert.

- Da zur Sicherungspunkterstellung Information über Betriebssystemaufrufe gesammelt werden muß, um betriebssystem-interne Komponenten des Prozeßzustands zu ermitteln, muß auch der Sicherungspunktgenerator vom Start des Zielprogramms an die Kontrolle über dieses ausüben können.

Aus diesen Gründen wurde bisher in den Werkzeugen `VISTOP`, `Codex` und `CoCheck` die Möglichkeit eines nachträglichen Ankoppelns an eine bereits laufende Anwendung auch gar nicht vorgesehen.

Damit steht fest, daß alle Werkzeuge der Umgebung vor bzw. beim Start des Zielprogramms gemeinsam und gleichzeitig gestartet werden müssen. Die Umgebung sollte daher mit einem einzigen Kommando aus dem Kommandointerpreter des Betriebssystems heraus aufgerufen werden können, das die einzelnen Werkzeuge startet und geeignet initialisiert. Dementsprechend sollten die Werkzeuge dann auch gemeinsam terminieren.

So bleibt nur noch die Frage zu klären, ob der Aufruf durch eine spezielle Option eines Werkzeugs, z.B. des Debuggers, oder durch ein separates Programm erfolgen soll, das im folgenden ***Rahmenprogramm*** oder ***Rahmenwerkzeug*** genannt wird. Eine der wesentlichen Prämissen dieser Arbeit ist, daß Werkzeuge interoperieren können sollen, ohne daß sie dazu Wissen übereinander benötigen. Diese Voraussetzung schließt das Starten der Gesamtumgebung durch eines der Werkzeuge aus. Der Start über ein eigenes Kommando führt auch zu einer klareren Benutzerschnittstelle, da ja nicht ein Werkzeug mit Zusatzfunktionen, sondern wirklich eine Umgebung mehrerer Werkzeuge aufgerufen wird.

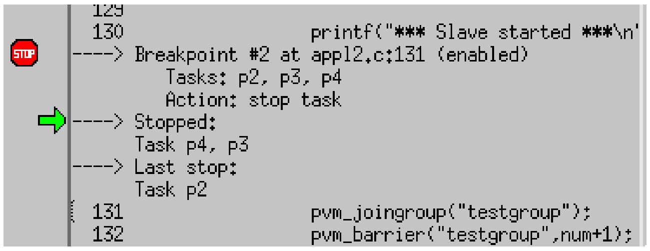
3.3.2 Anforderungen an die Konsistenz der dargestellten Information

Für eine Werkzeugumgebung, die aus mehreren Einzelwerkzeugen besteht, ist eine geradezu selbstverständliche Benutzeranforderung, daß die von diesen präsentierten Informationen nicht (und sei es auch nur scheinbar) widersprüchlich sein dürfen. In unserem Szenario gibt es zwei Werkzeuge, die Information über den Zustand des Zielprogramms anzeigen: den Debugger `DETOP` und den Visualisierer `VISTOP`. Es ist daher festzulegen, welche Anforderungen an die Konsistenz der von diesen beiden Werkzeugen dargestellten Zustandsinformation zu stellen sind.

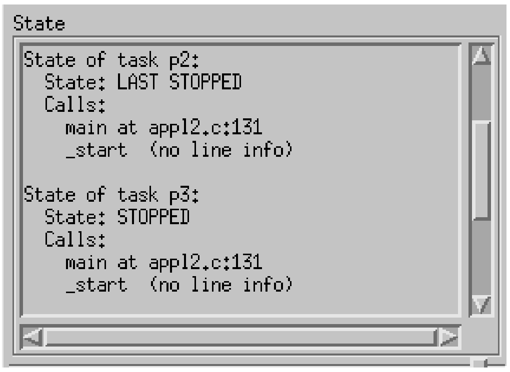
Die Situation ist zunächst die folgende: Abgesehen von dem Ausgabeprotokoll (in Abb 3.9 rechts unten) stellt `DETOP` immer Information über den aktuellen Zustand des Zielprogramms dar. Genau genommen gilt dies allerdings nur für gestoppte Prozesse. Für einen laufenden Prozeß zeigt `DETOP` weiterhin die Information an, die beim letzten Anhalten gültig war, versieht sie jedoch mit einem entsprechenden Hinweis, wie in Abb. 3.15 dargestellt ist. Im Gegensatz dazu besitzt `VISTOP` Information über den gesamten bisherigen Ablauf des Zielprogramms und kann insbesondere auch bereits vergangene Zustände darstellen. Aus Sicht eines unbedarften Benutzers wäre es jedoch reichlich verwirrend, wenn `VISTOP` einen vergangenen Zustand zeigte, während `DETOP` gleichzeitig den im obigen Sinne aktuellen Zustand darstellte. Dies gilt jedenfalls dann, wenn nicht klargestellt ist, daß die Anzeigen nicht konsistent sind, d.h. eine der beiden Darstellungen ungültig ist.

Eine mögliche Lösung dieses Konsistenzproblems läßt sich wie folgt erreichen:

- Während der Benutzer mit `DETOP` arbeitet, ist davon auszugehen, daß er den aktuellen Programmzustand untersuchen will. Daher sollte in diesem Fall auch `VISTOP` den aktuellen Zustand darstellen. Da vor der Untersuchung eines Prozesses mit einem Debugger



a) Markierung der aktuellen Position im Programmcode



b) Zustandsanzeige

Abbildung 3.15: Darstellung des letzten Zustands laufender Prozesse in DETOP

dieser Prozeß i.a. angehalten wird, ist es ausreichend, wenn der Visualisierer für alle gestoppten Prozesse den momentanen Zustand anzeigt. Dies bedeutet, daß VISTOP beim Anhalten eines Prozesses sein internes Zustandsmodell und die graphische Darstellung entsprechend aktualisieren muß.

- Kehrt der Benutzer mit Hilfe der Steuertasten von VISTOP zur Darstellung eines früheren Programmzustands zurück, so bildet nun dieser und nicht mehr der aktuelle Zustand den Fokus des Benutzerinteresses. Dementsprechend sollte auch DETOP im Idealfall die zu diesem Zustand passende Information anzeigen, d.h. diejenigen Daten, die dargestellt wurden, als der im VISTOP ausgewählte Zustand der aktuelle Programmzustand war. Diese Lösung ist allerdings relativ aufwendig, da sie das Sichern alter, von DETOP dargestellter Information erforderlich macht. Eine einfachere Lösung besteht darin, die betreffenden Anzeigen des Debuggers zumindest als ungültig zu markieren, etwa durch Schattieren der entsprechenden Ausgabebereiche der Debuggerfenster.

- Sobald der Benutzer wieder zu DETOP zurückkehrt und mit diesem weiterarbeitet, sollte DETOP zusammen mit VISTOP wieder den aktuellen Zustand darstellen.

Die skizzierte Prinziplösung wird im Detail noch etwas durch die Tatsache kompliziert, daß beide Werkzeuge nur bestimmte Teilaspekte des Zielprogrammzustands darstellen können, wobei DETOP allerdings deutlich mehr Details zeigen kann. Es stellt sich daher die Frage, was die Forderung „beide Werkzeuge zeigen denselben Zustand an“ bedeuten soll. Wir benötigen also eine exaktere Definition der Konsistenzbedingung. Diese ist am einfachsten zu ermitteln, wenn die Situation a-posteriori betrachtet wird, also nach vollendetem Ablauf des Zielprogramms. Der Ablauf kann dann durch ein Prozeß-Zeit-Diagramm dargestellt werden, bei dem man sich die Zeitpfeile als mikroskopische Folgen von Zuständen und Zustandsübergängen vorstellen kann. Einige dieser Zustandsübergänge sind dabei als Ereignisse durch die Werkzeuge beobachtbar.

Wird das Zielprogramm angehalten, sieht DETOP einen Globalzustand , der durch die Stellen des Anhaltens in jedem Prozeß gegeben ist und einen konsistenten Schnitt [214] definiert. Bezeichnen wir den Zustand, in dem Prozeß P_i angehalten wurde, mit $Z(P_i)$, so kann dieser Globalzustand durch das Tupel $Z = (\ Z(P_1), ..., Z(P_n)\ )$ definiert werden.

Da VISTOP nur eine Teilmenge der tatsächlichen Zustandsübergänge als Ereignisse beobachtet, entspricht ein von VISTOP dargestellter Globalzustand dagegen einem rechts offenen Intervall, das in jedem Prozeß jeweils von zwei für VISTOP beobachtbaren Ereignissen begrenzt wird. Diese Zustandsintervalle werden im folgenden zur besseren Unterscheidung als ***Modellzustände*** bezeichnet. Formal kann ein Modellzustand M wie folgt definiert werden:

$$M = [Z_1, Z_2[:= \{Z \mid \forall P_i : Z_1(P_i) \leq Z(P_i) < Z_2(P_i)\}$$

Die Ordnung auf lokalen Zuständen ist dabei eindeutig durch die Reihenfolge ihres Auftretens in dem jeweiligen Prozeß definiert.

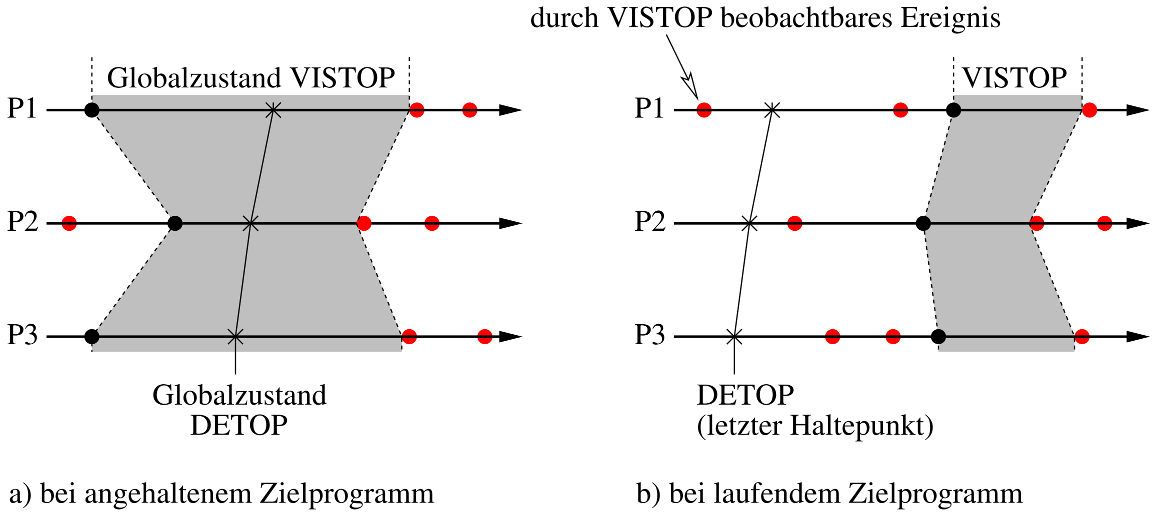


Abbildung 3.16: Zusammenhang der dargestellten Zustände in DETOP und VISTOP

Ist das Zielprogramm angehalten, so ist die zu erfüllende Konsistenzbedingung zwischen dem mit DETOP untersuchbaren Globalzustand Z und dem von VISTOP dargestellten Modellzustand M durch $Z \in M$ gegeben. Wie in Abb. 3.16a angedeutet ist, muß also der tatsächliche Globalzustand des Programms innerhalb des Intervalls liegen, das dem visualisierten Modellzustand entspricht. Für die Praxis heißt diese Bedingung, daß VISTOP beim Anhalten eines Prozesses alle Ereignisse abarbeiten muß, die der Prozeß bis dahin ausgelöst hat.

Für den Fall, daß sich ein Prozeß des Zielprogramms in Ausführung befindet, ergibt sich eine andere Situation: Wie erwähnt, zeigt DETOP immer noch die beim letzten Anhalten des Prozesses gültigen Daten an, versieht sie jedoch mit einer entsprechenden, erklärenden Markierung. VISTOP dagegen aktualisiert seine Darstellung von Zeit zu Zeit, diese folgt also zeitverzögert dem Programmablauf. Wie Abb. 3.16b verdeutlicht, sind die Anzeigen der Werkzeuge dann nicht mehr im eben definierten Sinne konsistent. Für eine solche Datenkonsistenz besteht insofern allerdings auch keine Notwendigkeit, als DETOP seine Darstellung sowieso bereits als veraltet markiert hat.

3.3.3 Verhalten beim Rücksetzen

Zu Beginn von § 3.3 wurde bereits informell festgelegt, daß die Werkzeugumgebung eine Funktion zum Rücksetzen des Zielprogramms auf den zu diesem Zeitpunkt von VISTOP dargestellten Modellzustand realisieren soll. Wir wollen diesen im folgenden mit dem Begriff ***Ziel-Modellzustand*** bezeichnen. Nach den obigen Ausführungen entspricht dieser Ziel-Modellzustand jedoch einem ganzen Intervall im Ablauf des parallelen Programms. Der eigentliche Zielzustand des Rücksetzens ist daher noch nicht eindeutig festgelegt. Es ergeben sich zwei Alternativen:

1. Es kann ein Rücksetzen auf einen beliebigen Zielzustand innerhalb des Intervalls erlaubt werden, das dem Ziel-Modellzustand entspricht.

2. Das Rücksetzen erfolgt immer auf den Beginn dieses Intervalls. Jeder Prozeß wird also auf den Zustand zurückgesetzt, den er unmittelbar nach Auslösen des Ereignisses besaß, das in `VISTOP` zum Übergang in den momentan visualisierten Modellzustand geführt hat.

Aus Benutzersicht ist die Realisierung der zweiten Alternative zu fordern, die den Zielzustand eindeutig definiert. Um eine Untersuchung des Programmablaufs unmittelbar nach dem Zielzustand zu ermöglichen, müssen zudem alle Prozesse nach dem Rücksetzen angehalten sein, so daß der Benutzer vor deren Weiterführung an geeigneten Stellen Haltepunkte setzen kann.

Neben der eben behandelten Frage, wie sich die Rücksetzfunktion auf das Zielprogramm auswirkt, ist auch zu klären, wie die beteiligten Werkzeuge reagieren müssen.

`DETOP`**:** Da die Zielprozesse nach dem Rücksetzen gestoppt sind und sich ihr Zustand verändert hat, müssen zunächst alle Zustandsanzeigen aktualisiert werden. Im Ausgabeprotokoll sollte eine eindeutige Markierung den Rücksetzzeitpunkt kenntlich machen. Ein Rücksetzen des Protokolls ist nicht angebracht, da hierdurch für den Benutzer wertvolle Information verlorengeht, um etwa Ausgaben während des erneuten Ablaufs mit den vorherigen vergleichen zu können.

Ob ein Rücksetzen der Haltepunktdefinitionen bzw. der Auswahl von zu beobachtenden Variablen vorteilhaft ist, hängt stark von der jeweiligen Situation ab. Es ist denkbar, daß die Definitionen, die seit dem Zeitpunkt durchgeführt wurden, auf den zurückgesetzt wird, überflüssig geworden sind, da sie der Überprüfung einer inzwischen widerlegten Hypothese dienten. Öfter wird jedoch der Fall eintreten, daß die Wiederholung der Programmausführung für die detailliertere Untersuchung einer Hypothese erforderlich ist, so daß die bisherigen Definitionen erhalten bleiben sollten. Da es zudem für den Benutzer einfacher ist, überflüssige Haltepunkte zu deaktivieren, als gelöschte neu zu setzen, entscheiden wir, Haltepunkte und Variablenbeobachtungen beim Rücksetzen nicht zu verändern.

`VISTOP`**:** Nach dem Rücksetzen entspricht der aktuelle Zustand des Zielprogramms dem dargestellten Modellzustand. Daher muß dieser im Visualisierer auch wieder zum letzten modellierten Zustand gemacht werden. Alle nachfolgenden – nun ja sozusagen in der Zukunft liegenden – Zustände müssen aus der Modellierungsschicht gelöscht werden.

`Codex`**:** Wie in § 3.2.5 erwähnt, realisiert `Codex` alle Kontrollmuster durch endliche Automaten, deren Zustand bei Kommunikationsoperationen aktualisiert wird. Beim Rücksetzen der Zielprozesse muß daher auch der Zustand dieser Automaten entsprechend zurückgesetzt werden, um ein korrektes Weiterfunktionieren des Werkzeugs zu gewährleisten.

`CoCheck`**:** Hier stellt sich die Frage der erforderlichen Reaktion auf das Rücksetzen nicht, da `CoCheck` dieses Rücksetzen selbst durchführt.

3.3.4 Weitergehende Integration

In den vorangegangenen Abschnitten wurde lediglich eine minimale Integration der Werkzeuge betrachtet. Selbstverständlich wäre auch eine wesentlich stärkere Integration denkbar und wünschenswert, wobei hier die Integration aus Benutzersicht, d.h. nach Brown et al. [12] auf der Prozeßebene, gemeint ist. So könnten die folgenden Funktionen realisiert werden, die nur eine kleine Auswahl der Möglichkeiten darstellen:

- Die Auswahl von Prozessen könnte nicht nur in der Prozeßliste von `DETOP`, sondern auch in der graphischen Anzeige von `VISTOP` erfolgen.
- `VISTOP` könnte eine Visualisierung der `Codex`-Kontrollmuster bzw. deren Wirkung auf den Programmablauf vornehmen.
- Möglich wäre auch eine graphische Manipulation der Kontrollmuster mit Hilfe von `VISTOP`, so daß z.B. nach einem Rücksetzen mit einem veränderten Kontrollmuster weitergearbeitet werden kann.
- Eine weitergehende Nutzung von Sicherungspunkten durch den Debugger könnte zur Realisierung neuer Debuggerfunktionen, z.B. einer Rückwärtsausführung des Programms [189] führen. Eine ausführliche Liste derartiger Funktionen findet sich in der Dissertation von J. Trinitis [235].
- Die gleichzeitige Verfügbarkeit von Ablaufspuren und Online-Debuggerfunktionen läßt schließlich auch eine erweiterte Debuggingmethodik für parallele Programme zu, etwa in Form von Makroschritten [113].

Wir werden derartige Erweiterungen in dieser Arbeit jedoch nicht betrachten, da es uns hier nicht primär um die entstehende Werkzeugumgebung geht, sondern um die Basismechanismen, die benötigt werden, um eine derartige Umgebung interoperabler Werkzeuge zu realisieren. Unter diesem Gesichtspunkt betrachtet, resultieren die erwähnten Erweiterungen nicht in zusätzlichen Anforderungen, da sie – neben eventuell notwendigen Anpassungen der Bedienoberfläche – ebenfalls nur Mechanismen zur Kommunikation zwischen Werkzeugen erfordern, wie sie z.B. auch zur Lösung der in § 3.3.2 beschriebenen Konsistenzproblematik notwendig sind.

3.3.5 Bedienoberfläche

Die Bedienoberfläche der Werkzeugumgebung bietet damit nur eine neue Funktion, nämlich das Rücksetzen des Zielprogramms auf einen früheren Globalzustand. Es ist also nur noch festzulegen, wie diese Funktion in die graphische Bedienoberfläche der Werkzeugumgebung eingebaut wird. Letztendlich ist zu entscheiden, in welchem Fenster der entsprechende Knopf oder Menüeintrag eingefügt werden soll. Zur Auswahl stehen dabei nur zwei sinnvolle Alternativen:

1. Die Funktion wird über ein neues, zentrales Fenster aufgerufen. Wie weiter oben festgestellt wurde, wird die Werkzeugumgebung durch ein eigenes Rahmenprogramm gestartet und soll entsprechend auch wieder gemeinsam beendet werden. Es ist also naheliegend, diesem Rahmenprogramm ein eigenes Fenster in der graphischen Bedienoberfläche zuzuordnen, das Bedienelemente für alle Funktionen enthält, die die Werkzeugumgebung als Ganzes betreffen. Im vorliegenden Fall wären dies zwei Knöpfe zum Rücksetzen des Zielprogramms und zum Beenden der Umgebung.

2. Die Rücksetz-Funktion wird über das `VISTOP`-Fenster aufgerufen. Diese Variante ist deshalb sinnvoll, da `VISTOP` den Parameter der Funktion definiert, nämlich den anzusteuernden Zielzustand.

Im folgenden wird die erste Variante realisiert. Es wird sich jedoch zeigen, daß auch die zweite Alternative mit den vorgeschlagenen Mechanismen realisierbar wäre.

3.3.6 Zusammenfassung

Nachfolgend werden die herausgearbeiteten Benutzeranforderungen und die getroffenen Entwurfsentscheidungen noch einmal kurz zusammenfaßt:

1. Alle Werkzeuge der Umgebung sowie sowie das zu untersuchende Zielprogramm werden gleichzeitig durch ein gemeinsames Rahmenprogramm gestartet.

2. Die graphische Bedienoberfläche der Werkzeugumgebung wird durch ein Fenster für globale Kommandos erweitert, die alle Werkzeuge betreffen: Rücksetzen des Zielprogramms und Beenden der Umgebung. Dieses Fenster wird vom Rahmenprogramm erzeugt und verwaltet.

3. Die Zustandsanzeigen von `DETOP` und `VISTOP` werden wie folgt konsistent gehalten:

   - Während der Benutzung von `DETOP` zeigt `VISTOP` den letzten gespeicherten Modellzustand. Der Zustand wird aktualisiert, wann immer ein Prozeß angehalten wird. Dies schließt insbesondere das Lesen der Ereignisspur vom Monitorsystem und deren Bearbeitung ein.

   - Wird `VISTOP` beauftragt, einen Zustand in der Vergangenheit zu visualisieren, werden die Zustandsanzeigen im `DETOP` schattiert, d.h. als ungültig markiert.

4. Das Rücksetzen des Zielprogramms erfolgt auf den frühest möglichen Zustand, der dem von `VISTOP` dargestellten Modellzustand entspricht. Die Zustandsfolge in der Modellierungsschicht von `VISTOP`, sowie die Kontrollmuster-Automaten im Instrumentierungscode von `Codex` müssen dabei ebenfalls zurückgesetzt werden. `DETOP` aktualisiert dagegen lediglich seine Zustandsanzeigen und fügt eine Markierung in das Ausgabeprotokoll ein.

Betrachtet man die Interaktionen, die zwischen zwei Werkzeugen bzw. zwischen einem Werkzeug und dem Monitorsystem notwendig werden, um die genannten Funktionen zu realisieren, stellt man fest, daß diese alle durch Ereignis/Aktions-Regeln beschrieben werden können. Ein Beispiel einer solchen Regel ist:

> Wenn `VISTOP` einen vergangenen Zustand visualisiert, soll `DETOP` seine Zustandsanzeigen schattieren.

Allgemeiner sind die Regeln von der Form:

> Wenn A eine Aktion x ausführt, soll B die Aktion y ausführen.

```
class DETOP {
operations:
    // deaktiviere (schattiere) bzw. aktiviere die Zustandsanzeigen
    void activate_state_display(boolean activate);
events:
    // Benutzer arbeitet mit DETOP (z.B.: DETOP erhält Eingabefokus
    // vom Fenstersystem)
    IsActive();
}

class VISTOP {
operations:
    // lies neue Ereignisse vom Monitorsystem und aktualisiere Modellzustandsfolge
    void read_events();
    // zeige letzten Modellzustand an
    void show_last_state();
events:
    // VISTOP zeigt vergangenen Zustand
    VisualizePastState();
    // VISTOP zeigt letzten Zustand
    VisualizeCurrentState();
}

Ereignis/Aktions-Regeln:
    thread_has_been_stopped([]) :
          VISTOP.read_events() VISTOP.show_last_state()
    VISTOP.VisualizeCurrentState() :
          DETOP.activate_state_display(true)
    VISTOP.VisualizePastState() :
          DETOP.activate_state_display(false)
    DETOP.IsActive() :
          VISTOP.show_last_state()
```

Abbildung 3.17: Abstrakte Verhaltenstypen und Verhaltensregeln zur Integration von DETOP und VISTOP auf Ebene der Benutzerschnittstelle

Genau diese Art von Regeln können durch das in § 2.2.3.2 vorgestellte Mediator-Konzept realisiert werden. Da sie nicht fest mit den einzelnen Werkzeugen verknüpft sind, sondern vielmehr von deren Zusammenstellung zu einer konkreten Werkzeugumgebung abhängen, dürfen sie nicht in die Werkzeuge einprogrammiert werden, sondern müssen durch das Rahmenprogramm festgelegt werden. Dieses übernimmt damit teilweise die Rolle des Mediators. Die abstrakten Verhaltenstypen für DETOP und VISTOP, sowie die Verhaltensregeln, die sich aus der vorangegangenen Diskussion ergeben, sind in Abb. 3.17 zusammengefaßt[3].

[3]Für die Verhaltensregeln verwenden wir hier und im folgenden eine Schreibweise, die sich an der Syntax von OMIS-Anfragen orientiert.

3.4 Algorithmische Definition der Zusatzfunktionen

Die Realisierungsidee der Rücksetzfunktion wurde bereits in § 3.1 skizziert: VISTOP verwaltet intern eine Folge von globalen Zuständen des Zielprogramms, die über ein Modell des Kommunikationssystems aus Ereignissen berechnet wurde. Der Benutzer kann einen beliebigen Modellzustand aus der Folge für die genauere Darstellung auswählen. Das Rücksetzen soll das Zielprogramm wieder in den dargestellten Zustand versetzen, bzw. genauer jeden Prozeß unmittelbar hinter das letzte von VISTOP überwachte Ereignis zurücksetzen, das vor dem visualisierten Modellzustand aufgetreten war. Dazu werden mittels CoCheck periodisch Sicherungspunkte erzeugt und gespeichert. Beim Rücksetzen muß nun zunächst ein geeigneter Sicherungspunkt gefunden und restauriert werden. Mit Hilfe von Codex und der in VISTOP gespeicherten Ereignisinformation muß anschließend der Zielzustand deterministisch „angefahren" werden.

3.4.1 Periodische Sicherungspunkterstellung

Zur periodischen Erstellung von Sicherungspunkten muß durch einen Zeitgeber regelmäßig ein Aufruf der CoCheck-Funktion `proc_checkpoint` erfolgen. Dadurch wird nicht nur der Zustand des Zielprogramms gesichert, sondern automatisch auch der Zustand der Kontrollmuster-Automaten von Codex (siehe § 3.5.2.2). Dies ist der Fall, weil die Automaten durch den von Codex in die Zielprozesse eingebauten Instrumentierungscode fortgeschaltet werden, sie selbst also auch im Adreßraum dieser Prozesse liegen müssen.

Um beim späteren Rücksetzen einen geeigneten Sicherungspunkt finden zu können, muß bei der Sicherung eine Abbildung zwischen dem gesicherten Programmzustand und dem entsprechenden VISTOP-Modellzustand erstellt werden. Mit Hilfe dieser Abbildung kann dann für jeden von VISTOP dargestellten Modellzustand der letzte vor diesem Zustand erstellte Sicherungspunkt ermittelt werden. Dazu werden zunächst eindeutige Bezeichner für die Modellzustände im VISTOP und die erstellten Sicherungspunkte benötigt. Da die Modellzustände eine total geordnete Folge bilden, können einfache Sequenznummern verwendet werden. Die Sicherungspunkte werden durch ihre eindeutigen Dateinamen identifiziert.

Zum Aufbau der Abbildung muß beim Erstellen eines Sicherungspunkts die Sequenznummer desjenigen VISTOP-Modellzustands ermittelt werden, der zum aktuellen Zustand des Zielprogramms gehört. Dazu müssen von VISTOP die eventuell noch im Monitorsystem gepufferten Ereignisspuren gelesen und bearbeitet werden. Die gesuchte Sequenznummer ist dann die des letzten berechneten Modellzustands. Um die Konsistenz der Abbildung sicherzustellen, muß allerdings gewährleistet werden, daß sich der Zustand des Zielprogramms zwischen der Erstellung des Sicherungspunkts und dem Auslesen der Ereignisspuren durch VISTOP nicht verändern kann. Dies kann am einfachsten dadurch erreicht werden, daß alle beteiligten Zielprozesse vor der Sicherung temporär angehalten werden.

Mit der in Abb. 3.23 spezifizierten Erweiterung des Verhaltenstyps von VISTOP um eine Funktion `get_last_pos` zur Bestimmung der Sequenznummer des letzten Modellzustands ergibt sich damit der in Abb. 3.18 dargestellte Algorithmus[4].

Nachfolgend wird für einen Modellzustand M dessen Sequenznummer mit $\mathrm{seq}(M)$ bezeichnet; für einen Sicherungspunkt S bezeichnet $\mathrm{seq}(S)$ die Sequenznummer des Modellzustands, der ihm zugeordnet ist. Abb. 3.19 veranschaulicht diese Zuordnung graphisch.

[4]Die Syntax orientiert sich an der Programmiersprache C. Zur Vereinfachung werden auch OMIS-Anfragen syntaktisch als normale Funktionsaufrufe dargestellt.

```
checkpoint() {
   ckpt_name = eindeutiger Dateiname
   thread_suspend([])
   proc_checkpoint([], ckpt_name)
   VISTOP.read_events()
   pos = VISTOP.get_last_pos()
   thread_resume([])
   speichere Abbildung zwischen pos und ckpt_name
}
```

Abbildung 3.18: Algorithmus für die periodische Sicherung

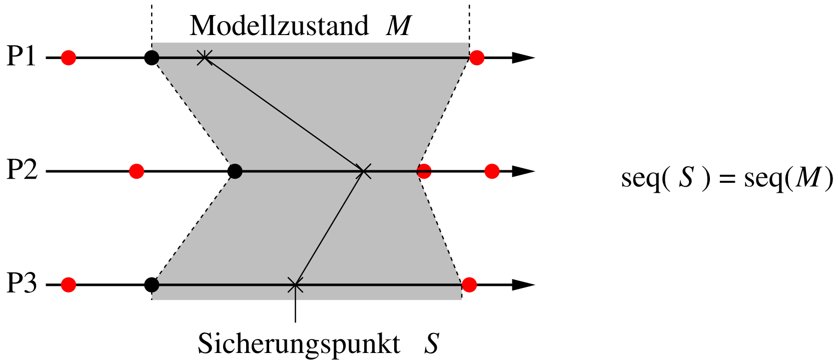


Abbildung 3.19: Abbildung zwischen Sicherungspunkten und Modellzuständen

3.4.2 Rücksetzen

Das Rücksetzen des Zielprogramms auf einen von VISTOP dargestellten Zustand umfaßt vier Schritte:

1. Ermittlung eines geeigneten Sicherungspunkts,
2. Restaurieren des Sicherungspunkts,
3. Deterministische Weiterführung des Zielprogramms,
4. Anhalten des Zielprogramms bei Erreichen des Zielzustands.

Diese Schritte werden in den nachfolgenden Unterabschnitten näher spezifiziert.

3.4.2.1 Ermittlung eines geeigneten Sicherungspunkts

Zunächst ist aus den periodisch erstellten Sicherungspunkten ein geeigneter Kandidat auszuwählen, der im nachfolgenden Schritt restauriert wird. Intuitiv ist dies der letzte Sicherungspunkt, der vor dem ausgewählten Zielzustand generiert wurde. Genauer wird der letzte Sicherungspunkt S gesucht, für den gilt: Ist Z_S der in S gespeicherte Globalzustand und Z_Z der Zielzustand, so gilt für alle Prozesse P_i: $Z_S(P_i) < Z_Z(P_i)$[5]. Stellt VISTOP aktuell einen Modellzustand $M = [Z_1, Z_2[$ dar, so ist der Zielzustand durch $Z_Z = Z_1$ festgelegt (vgl. Abb. 3.20).

[5]Die Gleichheit muß an dieser Stelle ausgeschlossen werden, da bei der Weiterführung von P_i das Erreichen von $Z_Z(P_i)$ nur durch das Auftreten eines Ereignisses erkannt werden kann (siehe Seite 94).

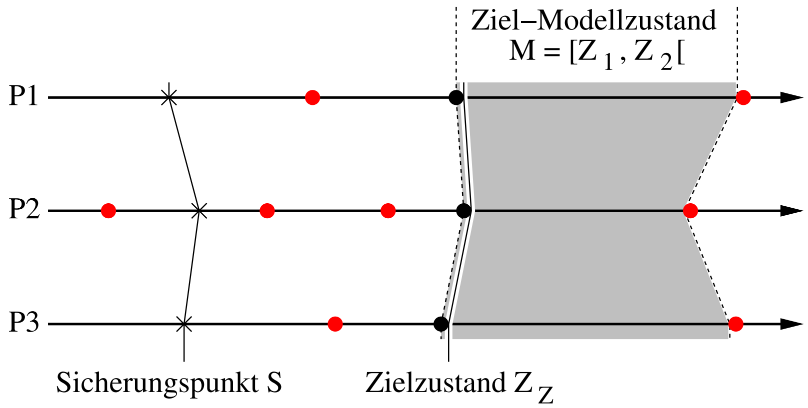


Abbildung 3.20: Definition des Zielzustands beim Rücksetzen

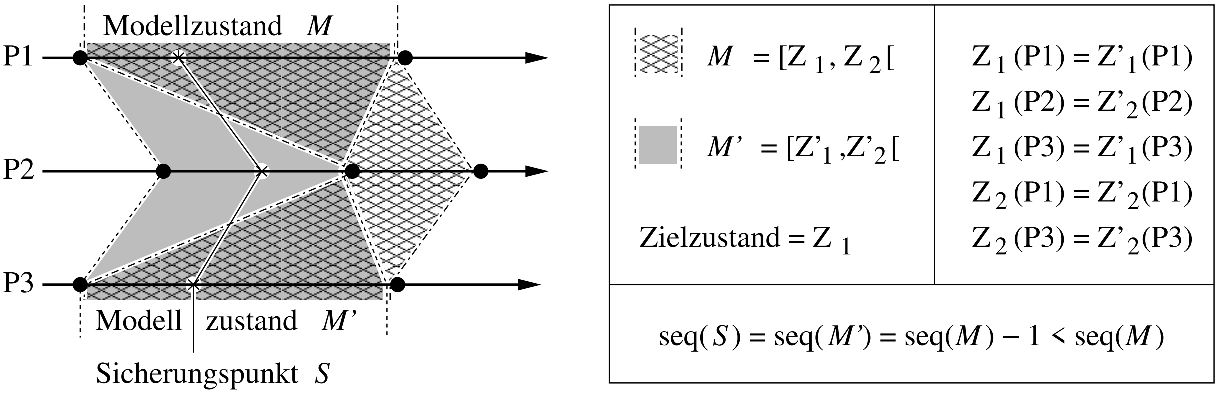


Abbildung 3.21: Zur Relation zwischen Sicherungspunkten und Modellzuständen

Da `VISTOP` für jedes bearbeitete Ereignis einen neuen Modellzustand generiert, unterscheiden sich zwei aufeinanderfolgende Modellzustände nur im lokalen Zustand eines einzigen Prozesses. Der gesuchte Sicherungspunkt S ist daher nicht notwendigerweise derjenige mit der größten Sequenznummer $\mathrm{seq}(S) < \mathrm{seq}(M)$. Dies ist anhand eines Beispiels in Abb. 3.21 verdeutlicht, bei dem M aus M' durch die Bearbeitung eines Ereignisses in Prozeß 2 hervorgeht. Es muß daher rückwärts der letzte Sicherungspunkt S gesucht werden, für den gilt:

1. $\mathrm{seq}(S) < \mathrm{seq}(M)$

2. In jedem Prozeß P_i, der im Zustand Z_Z existiert, gibt es mindestens ein von `VISTOP` beobachtetes Ereignis zwischen Z_S und Z_Z.

Problematisch kann dies in Situationen werden, in denen ein Prozeß lange Zeit kein von `VISTOP` beobachtetes Ereignis generiert, etwa bei einer *Master-Slave*-Anwendung, in der der *Master*-Prozeß nach der Initialisierung lediglich noch auf die Beendigung der anderen Prozesse wartet. In solchen Fällen müßte eventuell ein sehr weit zurückliegender Sicherungspunkt restauriert und damit ein Großteil des Programmablaufs wiederholt werden. Mit der in § 3.3.3 genannten weicheren Anforderung an den Zielzustand beim Rücksetzen könnte dagegen immer der letzte Sicherungspunkt S mit $\mathrm{seq}(S) \leq \mathrm{seq}(M)$ verwendet werden. Ein für die Praxis geeigneter Kompromiß wäre die Spezifikation einer Menge von Prozessen durch den Benutzer, für die ein „exaktes" Rücksetzen erfolgen soll. Zusätzlich könnte dies mit einer Warnung durch das Werkzeug kombiniert werden, falls ein sehr weit zurückliegender Sicherungspunkt ausgewählt werden muß. Für die hier untersuchte Fragestellung der Interoperabilität von Werkzeugen sind diese Erweiterungen jedoch ohne Belang, so daß im folgenden nur die eingangs skizzierte Lösung betrachtet wird.

3.4.2.2 Restaurieren des Sicherungspunkts

Sobald ein geeigneter Sicherungspunkt ausgewählt wurde, können die Prozesse des Zielprogramms durch eine entsprechende OMIS-Anfrage an die `CoCheck`-Erweiterung im OCM auf diesen Sicherungspunkt zurückgesetzt werden. Dadurch werden gleichzeitig die Kontrollmusterautomaten von `Codex` auf ihren Zustand bei Erstellung des Sicherungspunkts zurückgesetzt.

Da bei der anschließenden Weiterführung der Prozesse bis zum Erreichen des eigentlichen Zielzustands diese erneut von `VISTOP` beobachtbare Ereignisse produzieren, ist bei der Restaurierung des Sicherungspunkts S die Liste der Modellzustände im `VISTOP` nach dem Modellzustand mit der Sequenznummer $\mathrm{seq}(S)$ abzuschneiden. Ebenso sind alle eventuell noch nicht bearbeiteten Ereignisse zu löschen. Bei der Weiterführung der Prozesse baut `VISTOP` die Modellzustände bis zum Ziel-Modellzustand dann wieder auf. Alternativ wäre es möglich, die Überwachung von Ereignissen durch den Visualisierer während der Weiterführung bis zum Zielzustand abzuschalten und nur die Modellzustände nach dem Ziel-Modellzustand zu löschen.

Nach dem Rücksetzen der Prozesse auf den Sicherungspunkt müssen bis zum Erreichen des Zielzustands alle von `DETOP` definierten Haltepunkte temporär abgeschaltet werden, um ein unplanmäßiges, verfrühtes Anhalten der Prozesse zu verhindern.

3.4.2.3 Deterministische Weiterführung des Zielprogramms

Mit dem Rücksetzen des Zielprogramms auf den Sicherungspunkt werden gleichzeitig auch die Kontrollmusterautomaten von `Codex` in den alten Zustand gebracht. Bei der Wiederholung des Programmablaufs bis zum Zielzustand erzwingt `Codex` daher automatisch wieder exakt dasselbe Zugriffsmuster wie im ursprünglichen Ablauf.

Ein Problem kann an dieser Stelle allerdings auftreten, wenn das vom Benutzer spezifizierte Kontrollmuster nicht vollständig ist, d.h. den Ablauf nicht gänzlich determiniert. Um derartige Fälle zu berücksichtigen, könnte die von `VISTOP` aufgezeichnete Ereignisspur zur Erweiterung des Kontrollmusters verwendet werden, da sie alle notwendigen Informationen des ursprünglichen Ablaufs enthält. Die Möglichkeit einer dynamischen Erweiterung des Kontrollmusters, bei der der Zustand des ursprünglichen Musters erhalten bleiben muß, würde jedoch massive Änderungen in `Codex` erfordern, was nicht im Sinne der hier betrachteten Art von Interoperabilität ist. Die Werkzeugumgebung setzt daher voraus, daß vom Benutzer ein Kontrollmuster definiert wurde, das den Programmablauf vollständig deterministisch macht. Da für das Debugging paralleler Programme die Reproduzierbarkeit des Ablaufs aber ohnehin essentiell ist, stellt diese Voraussetzung in der Praxis keine zusätzliche Einschränkung dar.

3.4.2.4 Anhalten des Zielprogramms bei Erreichen des Zielzustands

Es bleibt nun noch zu klären, wie erkannt werden kann, daß das Zielprogramm den gewünschten Zielzustand erreicht hat. Bei gegebenem Ziel-Modellzustand $M = [Z_1, Z_2[$ ist dieser Zustand durch Z_1 gegeben. Jeder Prozeß P_i muß daher unmittelbar nach dem Auslösen des Ereignisses $e(P_i)$ angehalten werden, das ihn in den Zustand $Z_1(P_i)$ bringt. Unmittelbar nach dem Restaurieren des Sicherungspunkts S ist also für jeden Prozeß P_i über die OMIS-Schnittstelle eine Ereignis/Aktions-Relation zu spezifizieren, deren Ereignisdefinition das Ereignis $e(P_i)$ umfaßt. Die dazu notwendige Information kann aus der in `VISTOP` gespeicherten Ereignisspur entnommen werden, da das betreffende Ereignis notwendigerweise der Klasse der von `VISTOP` beobachteten Ereignisse angehört. Das gesuchte Ereignis $e(P_i)$ ist jedoch nicht notwendigerweise das erste auf diese Ereignisdefinition passende Ereignis, das in P_i bei der Weiterführung nach dem Sicherungspunkt auftritt. Daher muß aus der in `VISTOP` gespeicherten Ereignisspur für jeden Prozeß P_i ermittelt werden, wie oft ein die jeweilige Ereignisdefinition triggerndes Ereignis eintritt, bevor der Zustand $Z_1(P_i)$ erreicht wird. Dazu wird eine Funktion benötigt, die Information über alle von `VISTOP` aufgezeichneten Ereignisse zwischen den Modellzuständen mit den Sequenznummern $\mathrm{seq}(S)$ und $\mathrm{seq}(M)$ ausgibt.

Nach dem Start der Prozesse sind dann bei jedem Triggern einer Ereignisdefinition die folgenden Aktionen auszuführen:

1. Temporäres Suspendieren des auslösenden Prozesses,
2. Erhöhen und Prüfen eines Ereigniszählers. Falls die erforderliche Anzahl von Ereignissen noch nicht erreicht ist: Weiterführung des Prozesses,
3. Ist die erforderliche Anzahl in allen Prozessen erreicht: Anhalten des Zielprogramms.

Bei Erreichen des Zielzustands wird eine Markierung an das Ausgabeprotokoll des Debuggers angefügt, die auf das Rücksetzen des Zielprogramms hinweist. Das Anhalten des Programms führt zudem automatisch zur Aktualisierung der Zustandsanzeigen in `DETOP`. Über die in Abb. 3.17 definierten Regeln wird auch `VISTOP` zu einer Aktualisierung der Anzeige veranlaßt. Dabei wird durch die erneut generierten Ereignisse die hinter dem Sicherungspunkt abgeschnittene Folge von Modellzuständen neu aufgebaut und das letzte Element dieser Folge, also der Ziel-Modellzustand, visualisiert. Die Kontrollmusterautomaten von `Codex` besitzen an dieser Stelle ebenfalls wieder exakt denselben Zustand wie im ursprünglichen Ablauf, so daß alle in § 3.3.3 genannten Anforderungen erfüllt sind.

Der sich ergebende, gesamte Algorithmus für das Rücksetzen des Zielprogramms ist in Abb. 3.22 dargestellt. Er setzt das Vorhandensein der in Abb. 3.23 definierten Funktionen in `DETOP` bzw. `VISTOP` sowie eine Schnittstelle zu deren Aufruf voraus.

3.4.3 Zusammenfassung

Die Realisierung der Rücksetzfunktion kann durch die in Abb. 3.18 und 3.22 gezeigten Algorithmen erfolgen. Sie benötigen dabei zusätzliche Schnittstellen in `DETOP` und `VISTOP`, die in Abb. 3.23 definiert sind. Da die Algorithmen im wesentlichen der Koordination der einzelnen Werkzeuge in der Gesamtumgebung dienen, sind sie durch das Rahmenprogramm auszuführen. Dazu müssen folgende Voraussetzungen gegeben sein:

1. Das Rahmenprogramm muß Relationen zwischen Ereignissen in Werkzeugen und Aktionen in (möglicherweise) anderen Werkzeugen definieren können, um in seiner Funktion als Mediatorobjekt die Interaktionen zwischen den Werkzeugen zu realisieren. Diese werden nachfolgend als ***direkte Interaktionen*** bezeichnet.

```
reset() {
    cur_pos = VISTOP.get_pos()
    events = ()
    repeat {
        ckpt_pos = Postion des letzten Sicherungspunkts vor dem
                   dem Modellzustand cur_pos
        ckpt_name = Dateiname dieses Sicherungspunkts
        events += VISTOP.get_events(ckpt_pos, cur_pos)
        cur_pos = ckpt_pos
    } until (events enthält für jeden Prozeß mindestens ein Ereignis)
    DETOP.disable_bps()
    proc_restore(ckpt_name)
    VISTOP.clear_after_pos(ckpt_pos)
    for each process p {
        e[p] = letztes Ereignis von Prozeß p in events
        n[p] = Anzahl des Auftretens von e[p] in events
        Aktiviere Überwachung von Ereignis e[p] im Prozeß p
    }
    thread_continue([])
    np = Anzahl der Prozesse
    when (Ereignis e[p] tritt in Prozeß p auf) {
        n[p] = n[p] - 1
        if (n[p] == 0) {
            thread_stop([p])
            Deaktiviere Überwachung von Ereignis e[p] im Prozeß p
            np = np - 1
            if (np == 0) {
                DETOP.enable_bps()
                DETOP.append_output("—– restart —–")
                // Ende der Rücksetzphase
            }
        }
    }
}
```

Abbildung 3.22: Algorithmus für das Rücksetzen des Programmablaufs

2. Wie das Beispiel der Anzeigeaktualisierung in `VISTOP` beim Anhalten von Zielprozessen zeigt, müssen auch Relationen zwischen Ereignissen im Zielprogramm und Aktionen in Werkzeugen durch das Rahmenprogramm definiert werden können. Da die hier interessierenden Ereignisse in der Regel durch Aktionen anderer Werkzeuge verursacht werden, sprechen wir von ***indirekten Interaktionen*** zwischen den Werkzeugen.

3. Das Rahmenprogramm muß Methoden der von ihm kontrollierten Werkzeuge aufrufen können, beispielsweise um die Sequenznummer des aktuellen Modellzustands von `VISTOP` zu erfragen.

```
class DETOP {
operations:
    // füge text an das Ausgabeprotokoll an
    void append_output(string text);
    // deaktiviere alle Haltepunkte
    void disable_bps();
    // aktiviere die Haltepunkte wieder
    void enable_bps();
}

class VISTOP {
operations:
    // liefere die Sequenznummer des visualisierten Modellzustands
    integer get_pos();
    // liefere die Sequenznummer des letzten Modellzustands
    integer get_last_pos();
    // liefere die Liste aller Ereignisse zwischen den Modellzuständen mit den
    // Sequenznummern from und to. Genauer: Ist M' = [Z'_1, Z'_2[ der Modell-
    // zustand mit seq(M') = from und M = [Z_1, Z_2[ derjenige mit seq(M) = to,
    // so wird die Menge der Ereignisse { e | ∃ Prozeß P_i : e ist Übergang in einen
    // Zustand Y(P_i) mit Z'_2(P_i) ≤ Y(P_i) ≤ Z_1(P_i)} ausgegeben.
    events[] get_events(integer from, integer to);
    // lösche alle Modellzustände und Ereignisse nach der Sequenznummer pos
    void clear_after_pos(integer pos);
}
```

Abbildung 3.23: Erweiterung der abstrakten Verhaltenstypen von DETOP und VISTOP zur Realisierung der Rücksetzfunktion

4. Schließlich ist es erforderlich, daß das Rahmenprogramm auf das Monitorsystem zugreifen kann, um das Zielsystem auch direkt zu kontrollieren.

Betrachtet man diese Voraussetzungen abstrakter, wird deutlich, daß das Rahmenprogramm Ereignis/Aktions-Relationen – mit möglicherweise leerer Ereignisdefinition – spezifizieren können muß, wie sie von OMIS definiert werden. Ereignisse und Aktionen beziehen sich dabei allerdings nicht nur auf das Zielsystem, sondern auch auf die einzelnen Werkzeuge der Umgebung. Das Beispiel der indirekten Interaktionen zeigt, daß sogar „gemischte“ Relationen auftreten, die sowohl Zielsystem als auch Werkzeuge betreffen. Neben den eher implementierungstechnischen Aspekten aus § 2.3.4.2 ist dies ein weiteres Argument dafür, die Trennung von Zielsystem und Werkzeugen aufzugeben und die Relationen einheitlich mit Hilfe des Monitorsystems zu realisieren.

In Abb. 3.24 ist die entstehende Architektur der Werkzeugumgebung gezeigt, wobei das Monitorsystem hier nicht mehr als eigene Komponente dargestellt ist, sondern im Sinne einer Middleware nur indirekt als Vermittler der Interaktionen auftritt. In der Abbildung wird deutlich, daß direkte Interaktionen nur zwischen DETOP und VISTOP auftreten, da nur diese als interaktive Werkzeuge ein nicht vom Rahmenprogramm kontrolliertes Verhalten aufweisen, auf

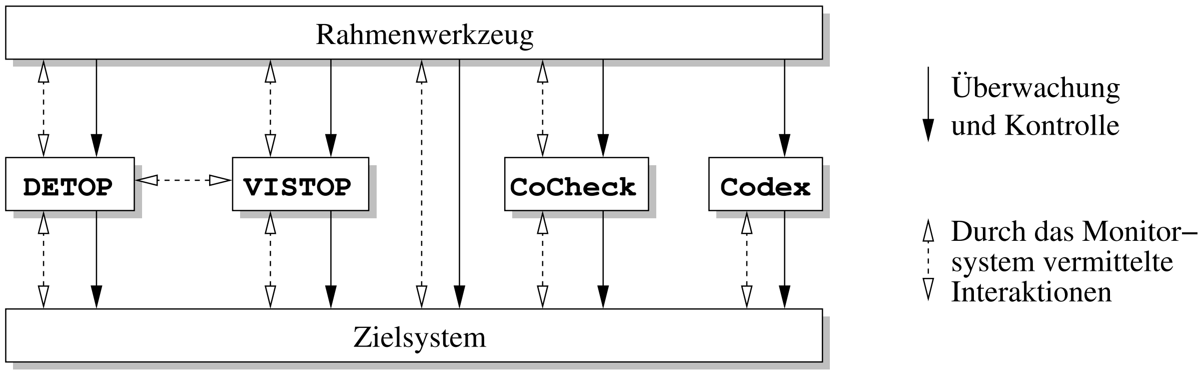


Abbildung 3.24: Architektur und Interaktionen der Werkzeugumgebung

das reagiert werden muß. Die automatischen Werkzeuge `CoCheck` und `Codex` werden dagegen vollständig vom Rahmenprogramm gesteuert[6].

3.5 Wechselseitige Beeinflussung der Werkzeuge

In diesem Abschnitt soll untersucht werden, wie sich die vier Werkzeuge des Beispielszenarios durch ihre gleichzeitige Anwendung auf dasselbe Zielsystem gegenseitig beeinflussen. Die Beeinflussungen entstehen dabei dadurch, daß die Werkzeuge das Zielsystem nicht nur passiv beobachten, sondern auch aktiv modifizieren. Abweichend von der sonst benutzten Klassifikation der Werkzeuge (siehe § 2.1.4) muß in diesem Zusammenhang bereits die Instrumentierung, also das Einbringen von Teilen des Monitorsystems in das Zielprogramm, als Modifikation betrachtet werden. Maßgebend für die Beeinflussung der Werkzeuge sind somit ausschließlich deren Monitorsysteme bzw. ihre jeweilige Nutzung. Die gegebene Situation soll daher noch einmal kurz verdeutlicht werden: `DETOP` und `VISTOP` nutzen OCM über die OMIS-Schnittstelle als gemeinsames Monitorsystem, `CoCheck` ist als Monitor-Erweiterung in den OCM integriert, `Codex` nutzt ein eigenes Monitorsystem, das im wesentlichen aus Instrumentierungscode besteht, der mittels *Binary Wrapping* um PVM-Bibliotheksfunktionen gelegt wird.

Die sich bei gleichzeitiger Nutzung einstellenden Wechselwirkungen sind in den nachfolgenden sechs Unterabschnitten zu funktional zusammengehörigen Klassen gruppiert.

3.5.1 Start des Zielprogramms

Der erste Konflikt zwischen den Werkzeugen entsteht bereits bei deren Start: In ihrer ursprünglichen Form erzeugen sowohl `DETOP` als auch `VISTOP` und `Codex` während der Initialisierung eine Instanz der zu untersuchenden parallelen Anwendung. Dies erfolgt bei `DETOP` und `VISTOP` über den entsprechenden OMIS-Dienst zur Prozeßerzeugung, bei `Codex` direkt durch den als Monitorsystem fungierenden Teil des Werkzeugs. Beim Start der Werkzeugumgebung würden daher ohne Gegenmaßnahmen drei unabhängige Instanzen des Zielprogramms erzeugt. Zur Lösung des Problems bieten sich zwei Alternativen an:

1. Eines der Werkzeuge startet das Zielprogramm wie bisher, die anderen koppeln sich während ihrer Initialisierung an das existierende Programm an.

[6]Eine Ausnahme bildet die Initialisierung von `Codex`, die in diesem Zusammenhang jedoch unwesentlich ist.

2. Das Zielprogramm wird vom Rahmenwerkzeug erzeugt, alle Werkzeuge müssen sich nachträglich mit diesem Zielprogramm verbinden.

Von allen Werkzeugen (evtl. abgesehen von einem) muß daher gefordert werden, daß sie sich nachträglich an ein bereits existierendes Zielprogramm ankoppeln können. Wie leicht einzusehen ist, ist dies eine Grundvoraussetzung für interoperable Online-Werkzeuge. In § 3.3.1 wurde allerdings bereits dargelegt, daß eine derartige Funktion für `VISTOP`, `Codex` und `CoCheck` nur unter großen Schwierigkeiten zu realisieren ist. Die Situation ist hier jedoch insofern einfacher, da die Werkzeuge alle gemeinsam mit dem Zielprogramm starten. Die Möglichkeit zum Ankoppeln eines Werkzeugs muß also nur für den Spezialfall gegeben sein, in dem die Zielprozesse zwar bereits erzeugt sind, die Ausführung jedoch noch nicht begonnen haben. Die in § 3.3.1 genannten Probleme treten in diesem Fall nicht auf.

Eine derartige, eingeschränkte Funktion ist für alle Werkzeuge einfach zu realisieren. Unter dieser Voraussetzung ist Alternative 2 wegen der einfacheren Koordination und der besseren Systematik vom Standpunkt des Software-Engineerings aus vorzuziehen. Das Rahmenprogramm kann dabei über eine OMIS-Anfrage die initialen Zielprozesse erzeugen und nach erfolgreicher Erzeugung die Werkzeuge mit entsprechenden Optionen unter Angabe geeigneter Prozeß-Identifikatoren aufrufen. Diese koppeln sich dann an die frisch erzeugten, aber noch angehaltenen Prozesse an.

3.5.2 Restaurierung des Zielprogrammzustands aus einem Sicherungspunkt

Beim Wiederanlauf des Zielprogramms ab einem Sicherungspunkt wird der gesamte Programmzustand modifiziert. Daher ergeben sich hierbei besonders viele Konflikte mit den anderen Werkzeugen, die durch geeignete Maßnahmen behandelt werden müssen.

3.5.2.1 Verlust dynamischer Instrumentierungen

Da sich das Code-Segment eines Prozesses bei dessen regulärer Ausführung nicht verändert und in Form der ausführbaren Programmdatei ohnehin schon dauerhaft gespeichert ist, verzichtet `CoCheck` zur Optimierung bei der Erzeugung von Sicherungspunkten auf das explizite Sichern dieses Segments [225]. Wie § 2.1.3.2 gezeigt hat, werden beim Einsatz von Online-Werkzeugen jedoch durchaus während der Laufzeit Änderungen im Code der Zielprozesse durchgeführt, um Instrumentierungscode einzubringen. Beispiele sind das Einfügen von *Trap*-Instruktionen zur Realisierung von Haltepunkten in Debuggern oder die dynamische Instrumentierungstechnik von `dyninst`. Derartige Instrumentierungen gehen bei der Restaurierung eines Prozesses verloren, da immer der Zustand des Code-Segments bei Programmstart wiederhergestellt wird. Statische Instrumentierungen, die z.B. durch *Binary Wrapping* in den Objektcode eingebracht wurden, bleiben dagegen erhalten.

Nach erfolgter Restaurierung eines Prozesses müssen daher sämtliche dynamischen Instrumentierungen neu eingefügt werden. Zu beachten ist dabei, daß gemäß der Diskussion in § 3.3.3 dabei nicht der Zustand zum Zeitpunkt der Sicherung, sondern derjenige unmittelbar vor der Restaurierung hergestellt werden muß. Kompliziert wird das Wiederherstellen der Instrumentierungen dadurch, daß weder die Werkzeuge noch die Ereignis/Aktions-Verwaltung des OCM Information darüber besitzen, mit welcher Instrumentierungstechnik ein Ereignisdienst realisiert ist. Zur Erzielung einer hohen Modularität und Erweiterbarkeit kann die Implementierung jedes Dienstes völlig unabhängig und eigenständig die Instrumentierungstechnik festlegen. Als Schnittstelle zur Ereignis/Aktions-Verwaltung dienen dabei zwei (möglicherweise leere) Methoden zum Einfügen und Entfernen der zur Laufzeit notwendigen Instrumentierung eines Prozesses (siehe § 6.2.2.2). Dadurch wird es jedoch möglich, über diese Methoden pauschal einfach für alle aktiven Ereignisdienste zunächst die Instrumentierungen zu entfernen und nach erfolgter Restaurierung des Prozesses wieder einzufügen. Um keine Ereignisse zu verlieren, muß dabei sichergestellt sein, daß währenddessen kein Benutzercode ausgeführt wird.

Zur Realisierung dieses Vorgehens muß `CoCheck` an zwei Stellen während des Rücksetzprotokolls eine Benachrichtigung des OCM durchführen, damit dieser die entsprechenden Aktionen ausführen kann:

1. Unmittelbar vor dem Restaurieren, wenn der alte Prozeß schon angehalten, aber noch nicht beendet wurde.

2. Unmittelbar nach erfolgtem Wiederanlauf des restaurierten Prozesses, bevor dieser beginnt, Benutzercode auszuführen. Gemäß der Semantik der `CoCheck`-Dienste bleibt ein Prozeß so lange in diesem Zustand angehalten, bis er explizit über eine OMIS-Anfrage wieder weitergeführt wird.

An der Stelle 1 müssen dann vom OCM alle Instrumentierungen entfernt und bei 2 wieder eingefügt werden. Da die Instrumentierungen beim Restaurieren verlorengehen, ist das explizite Entfernen zwar an sich unnötig, der Aufruf der entsprechenden Methoden muß jedoch erfolgen, um die internen Verwaltungsdatenstrukturen des OCM und der Ereignisdienste auf einen konsistenten Stand zu bringen.

3.5.2.2 Rücksetzen der Datenbereiche von Instrumentierungscode

Weitere Wechselwirkungen zwischen `CoCheck` und den anderen Werkzeugen entstehen dadurch, daß der für diese ins Zielprogramm eingefügte Instrumentierungscode im allgemeinen auch einen zugehörigen Datenbereich besitzt. Dessen Zustand wird aber von `CoCheck` gesichert und wieder restauriert. Im vorgestellten Szenario sind für jeden Zielprozeß davon folgende Daten betroffen:

1. Der Kontrollmusterautomat von `Codex`.

2. Das Zeigerfeld (*Hook Array*), das für jede im OCM durch *binary wrapping* instrumentierte Bibliotheksfunktion spezifiziert, ob der Instrumentierungscode zu Beginn bzw. am Ende der Funktion aufgeführt werden soll. Falls ja, zeigt der entsprechende Feldeintrag auf die Wurzel des abzuarbeitenden Filterbaums, ansonsten ist der Eintrag ein `NULL`-Zeiger (siehe § 3.2.1.2). Das Feld wird von der Ereignis/Aktions-Verwaltung des OCM bei der Definition bzw. Löschung von bedingten Monitoranfragen zur Laufzeit aktualisiert. Es dient der Geschwindigkeitsoptimierung, da eine Analyse eines Ereignisses nur dann notwendig wird, wenn tatsächlich Aktionen dafür im Filterbaum vorliegen.

Alle anderen Datenstrukturen des OCM, auf die vom Instrumentierungscode aus zugegriffen wird, liegen im gemeinsamen Speichersegment (vergleiche Abb. 3.8), dessen Inhalt von `CoCheck` vollständig ignoriert wird.

Das automatische Sichern und Restaurieren der Kontrollmusterautomaten durch `CoCheck` ist dabei vorteilhaft, da diese sowieso immer zusammen mit den Prozessen zurückgesetzt werden müssen (siehe § 3.3.3). Das Rücksetzen des Instrumentierungscodes von `Codex` erfordert jedoch noch eine genauere Betrachtung, da dieser über PVM mit der graphischen Bedienoberfläche von `Codex` kommuniziert. Graphische Oberflächen können von `CoCheck` prinzipiell nicht gesichert werden; wegen der Implementierung in Java kann die `Codex`-Bedienoberfläche noch nicht einmal zum Protokollumfang (siehe [225]) des Sicherungspunkts hinzugenommen werden. Etwaige, zum Zeitpunkt der Sicherung übertragene Nachrichten zwischen der Oberfläche und dem Instrumentierungscode könnten daher verlorengehen. Da eine derartige Kommunikation jedoch nur zur Initialisierung des Kontrollmusters beim Programmstart erfolgt, tritt diese Situation in der Praxis nicht auf. Sicherheitshalber meldet sich die Oberfläche nach der Initialisierung bei der virtuellen Maschine von PVM ab, bzw. könnte sich an diesem Punkt sogar beenden.

Die Restaurierung der *Hook Arrays* ist dagegen problematisch, da sie zu Inkonsistenzen zwischen den *Hook Arrays* und der aktuellen Information im gemeinsamen Speichersegment führen kann. Das *Hook Array* eines Prozesses muß also nach dem Rücksetzen explizit wieder so aufgebaut werden, daß es mit dieser Information konsistent ist. Das bereits im letzten Abschnitt als notwendig erkannte Entfernen und Wiedereinfügen aller dynamischen Instrumentierungen baut als Seiteneffekt automatisch auch die *Hook Arrays* wieder auf, so daß als zusätzliche Maßnahme lediglich ein vorheriges Löschen der Arrays erforderlich wird, um die restaurierte, nicht mehr aktuelle Information zu entfernen.

3.5.2.3 Erzeugung neuer Prozesse beim Wiederanlauf

Die Funktionsweise von `Condor`, das die Basis von `CoCheck` darstellt, bedingt, daß beim Rücksetzen eines Prozesses nicht der existierende Prozeß modifiziert wird, sondern ein neuer Prozeß mit dem entsprechenden Zustand erzeugt und der existierende gelöscht wird. Wie in § 3.2.3 erläutert wurde, melden sich diese Prozesse während des Wiederanlaufs neu beim PVM-Laufzeitsystem an, so daß sie auch neue Task-Identifikatoren (TIDs) erhalten. `CoCheck` stellt sicher, daß diese Änderungen für die Anwendung selbst transparent bleiben. Dies sollte daher auch für die angeschlossenen Werkzeuge gelten.

Die in der Arbeit von J. Trinitis [235] durchgeführte feste Integration von `CoCheck` in den OCM hat es ermöglicht, das Rücksetzen durch den Dienst `proc_reset` so zu realisieren, daß die Token der ursprünglichen Prozesse für die neu erzeugten wiederverwendet werden. In der jedem Token zugeordneten Prozeßdatenstruktur werden dabei lediglich die für die interne Funktion des OCM notwendigen Daten aktualisiert. Dadurch wurde zum einen erreicht, daß die Werkzeuge weiterhin die alten Token benutzen können, zum anderen, daß sie noch immer die ursprünglichen, in den Prozeßdatenstrukturen gespeicherten TIDs sehen.

Da wir in dieser Arbeit aber primär an lose gekoppelten Werkzeugen interessiert sind, soll im folgenden dargelegt werden, wie das beschriebene Verhalten auch ohne eine feste Integration zwischen `CoCheck` und OCM realisiert werden kann. Der OCM würde in diesem Fall den neuen Prozessen wie üblich neue Token zuweisen, in deren zugeordneten Prozeßdatenstrukturen auch die neuen TIDs gespeichert wären. Um diese Änderungen vor den anderen Werkzeugen zu verbergen, müßte `CoCheck` die relevanten Anfragen dieser Werkzeuge abfangen und entsprechend modifizieren, in ähnlicher Weise, wie dies auch mit den PVM-Bibliotheksaufrufen der Anwendung geschieht. Die durchzuführenden Modifikationen wären dabei:

1. Das Ersetzen von als Parameter übergebenen ehemaligen Token durch die entsprechenden aktualisierten Token der neuen Prozesse, bzw. umgekehrt, falls eine Anfrage Prozeßtoken als Ergebnis liefert.

2. Das Ersetzen der durch die Realisierung des Rücksetzens geänderten Information über einen Prozeß, z.B. dessen TID, durch die ursprüngliche Information.

Eine Frage, die dabei zu klären ist, betrifft die Ebene, auf der das Abfangen erfolgen sollte. Prinzipiell käme die Ebene der Monitorschnittstelle oder auch die tiefere Ebene der Zugriffe auf Prozeßdaten innerhalb des OCM in Betracht. Wir werden diesen Aspekt in § 5.3.3.1 genauer diskutieren.

3.5.3 Überwachung und Nutzung von Bibliotheksfunktionen

Die Beschreibung der Einzelwerkzeuge in § 3.2 hat verdeutlicht, daß sowohl `CoCheck` als auch `VISTOP` und `Codex` Aufrufe von Funktionen der PVM-Bibliothek durch das Zielprogramm überwachen müssen. Während `VISTOP` dabei die Parameter bzw. Ergebnisse der Aufrufe lediglich aufzeichnet, können diese von den beiden anderen Werkzeugen auch verändert werden. Abstrakt im Ereignis/Aktions-Modell betrachtet, bedeutet dies, daß Aktionen verschiedener Werkzeuge durch dasselbe Ereignis im Zielprogramm ausgelöst werden. Die Aktionen können dabei eventuell den Zustand des Zielprogramms verändern. Die Reihenfolge, in der die Aktionen eines Ereignisses ausgeführt werden, ist dabei durch deren Spezifikation selbst zunächst nicht festgelegt. Sie ist für die korrekte Funktion der Werkzeugumgebung jedoch essentiell, da sie z.B. bestimmt, ob ein Werkzeug – genauer: die auszuführende Aktion, die dieses Werkzeug definiert hat – einen PVM-Aufruf vor oder nach der Modifikation der Parameter durch die Aktionen eines anderen Werkzeugs zu sehen bekommt. Bei der Komposition der Werkzeugumgebung muß daher durch geeignete Maßnahmen eine Festlegung der für die korrekte Funktion jeweils benötigten Aktionsreihenfolge getroffen werden.

Bei den drei hier relevanten Werkzeugen werden Ereigniserkennung und Aktionsausführung durch *Binary Wrapping* (siehe Abb. 2.4) realisiert. Bei `CoCheck` und `Codex` sind dabei die auszuführenden Aktionen fest im Instrumentierungscode programmiert, während `VISTOP` seine Aktionen über die Ereignis/Aktions-Verwaltung des OCM definiert, die von dessen Standard-Instrumentierungscode aktiviert wird. Die einfachste Vorgehensweise zur gleichzeitigen Nutzung der drei Werkzeuge besteht nun darin, die gesamte PVM-Bibliothek dreimal einzuwickeln: jeweils in den Instrumentierungscode von `CoCheck`, von `Codex` und des OCM. Die erforderliche Reihenfolge ist dabei noch zu ermitteln, wobei die Instrumentierungstechnik bewirkt, daß zum einen die Aktionsreihenfolge für alle PVM-Funktionen identisch ist, zum anderen die Aktionen am Beginn einer PVM-Funktion in der umgekehrten Reihenfolge abgearbeitet werden wie am Ende einer Funktion (vgl. Abb. 3.25). Diese Realisierung bewirkt zudem einen weiteren Zusatzeffekt: Wie in § 3.2.3.2 und § 3.2.5 erwähnt, ruft der Instrumentierungscode von `CoCheck` und `Codex` PVM-Funktionen für interne Zwecke auf. Die Reihenfolge des Einwickelns legt daher auch fest, ob ein anderes Werkzeug diese internen Aufrufe sieht oder nicht. Dies ist in Abb. 3.25 veranschaulicht.

Unklar ist an dieser Stelle freilich noch, ob eine derartige, pauschale Festlegung der Aktionsreihenfolge ausreichend ist, ob also eine gemeinsame Instrumentierungsreihenfolge für die gesamte PVM-Bibliothek gefunden werden kann, die eine korrekte Funktion der Werkzeugumgebung sichert, oder ob für die möglichen Einzelereignisse, d.h. Beginn bzw. Ende einer jeden PVM-Funktion, unterschiedliche Reihenfolgen notwendig werden. Um diese Frage zu klären, werden nachfolgend alle potentiellen Konflikte untersucht.

Zur Vereinfachung verwenden wir die Schreibweise $A \prec B$ um auszudrücken, daß die PVM-Bibliothek zunächst in den Instrumentierungscode von Werkzeug A und erst danach in

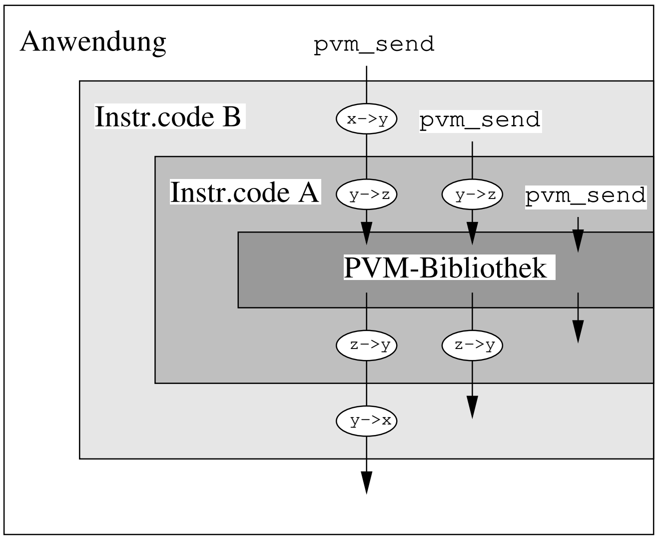


Abbildung 3.25: Zweifache Instrumentierung der PVM-Bibliothek durch *Binary Wrapping*

den von B eingewickelt wird, so daß das in Abb. 3.25 gezeigte Ergebnis entsteht. Dies bedeutet, daß ein Funktionsaufruf zuerst von Werkzeug B erkannt und behandelt wird und anschließend von A, wobei A den bereits durch B modifizierten Zustand sieht. Die Rückkehr aus einer PVM-Funktion wird zunächst von A erkannt und behandelt, dann von B. Schließlich sieht A die internen PVM-Aufrufe von B, umgekehrt gilt dies jedoch nicht.

Als rein beobachtendes Werkzeug ist `VISTOP` relativ einfach abzuhandeln. `VISTOP` visualisiert das Verhalten des beobachteten Anwendungsprogramms, muß also genau die Information erfassen, die auch die Anwendung selbst zu sehen bekommt. Für die Bibliotheksinstrumentierung bedeutet dies, daß der Instrumentierungscode von `VISTOP` ganz außen zu liegen kommen muß. `VISTOP` sieht also die Aufrufe einer PVM-Funktion als erstes, ihre Rückkehr dagegen als letztes Werkzeug. Gleichzeitig wird dadurch erreicht, daß `VISTOP` keine Kenntnis der internen PVM-Aufrufe im Instrumentierungscode anderer Werkzeuge erhält. Als Resultat halten wir `CoCheck` $\prec$ `VISTOP` und `Codex` $\prec$ `VISTOP` fest.

Komplizierter ist die Situation zwischen `CoCheck` und `Codex`, da beide Werkzeuge den Zustand des Zielprogramms, speziell die Parameter und Ergebnisse seiner PVM-Aufrufe, verändern und zudem PVM auch für interne Zwecke nutzen. Zunächst ist festzuhalten, daß `Codex` das Kommunikationsverhalten der Anwendung aufgrund eines Modells dieser Anwendung und darüber spezifizierte Zugriffsmuster steuert. Somit sollte `Codex` die interne Kommunikation von `CoCheck` nicht bemerken, da diese im Modell und den Mustern nicht berücksichtigt ist und dadurch zu Fehlentscheidungen des Werkzeugs führen kann. Zudem darf diese Kommunikation auch nicht durch `Codex` gesteuert werden, da sie nicht Teil der Anwendung ist. Dies kann erreicht werden, indem die Bibliotheksinstrumentierung zunächst für `CoCheck`, dann für `Codex` durchgeführt wird, also `CoCheck` $\prec$ `Codex` gesetzt wird.

In § 3.5.2.3 wurde ausgeführt, daß die Änderung der TIDs der Zielprozesse beim Rücksetzen für die Werkzeuge nicht sichtbar sein sollte. Dies gilt insbesondere auch für den Instrumentierungscode, der von `Codex` und `VISTOP` in die Zielprozesse eingebracht wird. Da dieser Code Information über TIDs nur durch die Parameter bzw. Ergebnisse der überwachten PVM-Funktionen erhält, muß er beim Aufruf einer solchen Funktion vor, bei deren Rückkehr dagegen nach der Umsetzung der TIDs durch den Instrumentierungscode von `CoCheck` ausgeführt werden. Es ergibt sich damit die Forderung `CoCheck` $\prec$ `VISTOP`, `CoCheck` $\prec$ `Codex`.

Neben dem Zustand der Anwendung wird bei der Erstellung eines Sicherungspunkts durch `CoCheck` auch der aktuelle Zustand von `Codex` mitgesichert (siehe § 3.4.1). Um die richtige Funktion des `CoCheck`-Protokolls zum Leeren der Kommunikationskanäle sicherzustellen, muß `CoCheck` daher auch die internen, vom `Codex`-Instrumentierungscode versandten Nachrichten sehen und berücksichtigen. Die Verwendung der von `CoCheck` modifizierten PVM-Aufrufe durch `Codex` ist auch erforderlich, um nach dem Rücksetzen den korrekten Empfang der zwischengespeicherten Nachrichten zu gewährleisten. Daher muß `CoCheck` $\prec$ `Codex` gelten.

Verallgemeinert kann man feststellen, daß `CoCheck` alle PVM-Aufrufe exakt so zu Gesicht bekommen muß wie das PVM-Laufzeitsystem. Dies bedeutet, daß die PVM-Bibliothek generell zunächst mit dem `CoCheck`-Code instrumentiert werden muß, also `CoCheck` $\prec$ `Codex`, `CoCheck` $\prec$ `VISTOP`.

Betrachtet man alle oben aufgestellten Relationen über die Instrumentierungsreihenfolge, so stellt man fest, daß diese alle mit der Reihenfolge `CoCheck` $\prec$ `Codex` $\prec$ `VISTOP` verträglich sind. Im vorliegenden Szenario kann also tatsächlich pauschal für alle Funktionen der PVM-Bibliothek eine einheitliche Instrumentierungsreihenfolge verwendet werden. Diese ist in Abb. 3.26 veranschaulicht. Ob diese Optimalsituation hier ein seltener Zufall ist oder eher den Regelfall darstellt, kann aufgrund nur eines Szenarios natürlich nicht empirisch entschieden werden. Da aber zumindest automatische, modifizierende Werkzeuge meist dazu dienen, vorhandene Programmierschnittstellen, z.B. PVM, in transparenter Weise zu erweitern, ergibt sich in diesen Fällen eine natürliche Schichtung der Schnittstellen, die sich in einer entsprechenden Instrumentierungsreihenfolge widerspiegelt. Insofern ist davon auszugehen, daß unser Szenario durchaus einen für die Praxis repräsentativen Fall darstellt. Zur Wahrung der Allgemeingültigkeit wird in § 5.3.3.2 trotzdem ein detaillierteres Modell entwickelt, das die Ausführungsreihenfolge von Aktionen auch für einzelne Ereignisse individuell zu bestimmen erlaubt.

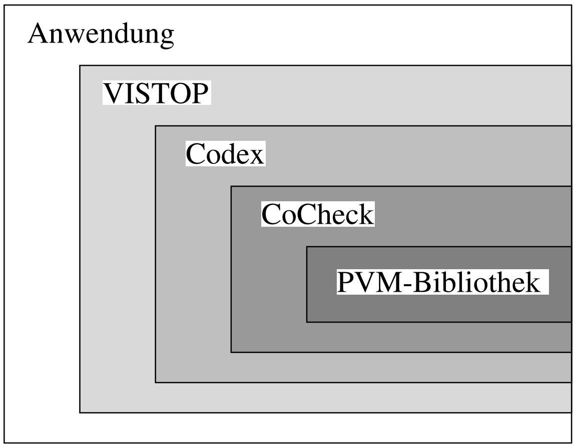


Abbildung 3.26: Instrumentierungsreihenfolge für die PVM-Bibliothek

Die Tatsache, daß alle Konflikte zwischen den Instrumentierungscodes der drei Werkzeuge `CoCheck`, `Codex` und `VISTOP` allein durch eine geeignete Instrumentierungsreihenfolge gelöst werden können, hat den positiven Nebeneffekt, daß ein gemeinsames Monitorsystem für `Codex` und die anderen Werkzeuge im betrachteten Szenario nicht erforderlich wird. Eine weitere notwendige Bedingung dafür ist, daß die verwendeten Instrumentierungstechniken miteinander verträglich sind, also keine strukturellen Konflikte (siehe § 4.3.1) entstehen. Diese

Bedingung ist hier erfüllt, da in allen Fällen *Binary Wrapping* verwendet wird.

3.5.4 Anhalten des Zielprogramms

In § 3.4.2.4 wurde beschrieben, daß beim Rücksetzen jeder Prozeß des Zielprogramms beim Erreichen des Ereignisses angehalten wird, das in `VISTOP` zum ausgewählten Ziel-Modellzustand führte. Dies wird durch Definition einer Ereignis/Aktions-Relation im OCM durch das Rahmenwerkzeug erreicht. Bis zum Anhalten des Programms zeichnet `VISTOP` dabei wie üblich Ereignisse auf, mit deren Hilfe die modellierte Zustandsfolge nach dem Abschneiden während des Wiederanlaufs (siehe § 3.4.2.2) erneut aufgebaut wird.

Damit `VISTOP` dabei wieder korrekt den Ziel-Modellzustand erreicht, muß sichergestellt werden, daß das Werkzeug auch das Ereignis noch sieht, das zum Anhalten eines Prozesses geführt hat. Die Aktion, die die erforderliche Ereignisaufzeichnung durchführt, wird im Kontext der Anwendungsprozesse ausgeführt und muß daher noch vor deren Anhalten abgearbeitet werden. Daraus ergibt sich scheinbar analog zu § 3.5.3 eine Anforderung an die Reihenfolge der Ausführung der vom Rahmenwerkzeug bzw. `VISTOP` definierten Aktionen.

Tatsächlich erfordert jedoch bereits die Semantik der OMIS-Schnittstelle, daß das Anhalten eines Anwendungsprozesses durch eine Aktion die sofortige Bearbeitung der weiteren Aktionen für dasselbe Ereignis nicht verhindert. Beim Eintreten eines Ereignisses ist gemäß dieser Semantik zunächst der auslösende Prozeß temporär zu suspendieren, anschließend werden alle dem Ereignis zugeordneten Aktionen abgearbeitet, zuletzt wird der Prozeß wieder weitergeführt, sofern er nicht zwischenzeitlich von einer Aktion dauerhaft gestoppt wurde. Dieses Schema stellt sicher, daß alle Werkzeuge beim Anhalten eines Prozesses konsistente Information erhalten, also insbesondere noch sofort das Ereignis sehen, das zum Anhalten geführt hat. Für Ereignisse, die im Kontext eines Monitorprozesses erkannt werden, kann es unmittelbar wie beschrieben realisiert werden. Bei der Ereigniserkennung im Kontext eines Anwendungsprozesses darf die genannte Suspendierung jedoch lediglich bewirken, daß der Prozeß keinen Anwendungscode mehr bearbeitet. Stattdessen wird Instrumentierungscode des OCM abgearbeitet, der die für das Ereignis definierten Aktionen ausführt. Es ist daher ein spezielles Synchronisationsprotokoll erforderlich, das sicherstellt, daß ein Anwendungsprozeß während der Ausführung dieses Codes vom Monitorprozeß nicht angehalten werden kann.

3.5.5 Weitere Modifikationen des Zielprogramm-Zustands

In § 3.2.3.1 wurde beschrieben, daß `CoCheck` eine Sicherungspunkterstellung unter anderem durch ein UNIX-Signal einleitet. Während der Erzeugung der lokalen Sicherungspunkte werden auch von `Condor` zwei weitere Signale an die betroffenen Zielprozesse gesendet. Das Auftreten dieser Signale ist für den normalen Anwendungscode nicht sichtbar, da der Instrumentierungscode von `CoCheck` eine geeignete Signalbehandlungsroutine definiert, während die von `Condor` verwendeten Signale von einem UNIX-Prozeß standardmäßig ignoriert werden. Debugging-Werkzeuge wie `DETOP` informieren den Benutzer üblicherweise aber über das Auftreten aller Signale im überwachten Programm. `DETOP` nutzt dazu den OMIS-Ereignisdienst `thread_received_signal`, verbunden mit einer Aktion, die `DETOP` bei deren Aufruf benachrichtigt.

Um zu verhindern, daß der Debugger und damit der Benutzer über die intern von `CoCheck` verwendeten Signale informiert wird, muß das entsprechende Ereignis bei der Bearbeitung im OCM an geeigneter Stelle abgefangen werden. Dies muß während des Durchlaufs durch den

Filterbaum erfolgen, der die Analyse von Ereignissen vornimmt (siehe § 3.2.1.2). Ein vorheriges Abfangen ist nicht möglich, da erst während des Durchlaufs erkannt wird, ob das aktuelle Ereignis ein Signal an einen der Zielprozesse war und, falls ja, welches. Vor dieser Analyse kann somit nicht entschieden werden, ob ein Abfangen erforderlich ist. Nach dem Durchlauf sind andererseits die zum Ereignis gehörigen und daher auszuführenden Aktionen bereits bestimmt. Eine Realisierungsmöglichkeit besteht in der Einführung spezieller Filter, die ihre Eingangsereignisse konsumieren, falls die Filterbedingung erfüllt ist. Dieses Konzept wird in § 5.3.3.2, Seite 150 näher ausgebaut.

Schließlich verursachen nicht nur die vom Instrumentierungscode der Werkzeuge durchgeführten Aktionen Änderungen im Zustand des Zielprogramms. Vielmehr ist bereits die bloße Anwesenheit dieses Codes eine derartige Änderung. Wird z.B. ein Zielprozeß zufällig im Instrumentierungscode von `Codex` angehalten, so zeigt `DETOP` eine Prozeduraufrufkette, die bei alleiniger Verwendung des Debuggers nicht auftreten kann. Ebenso kann der Instrumentierungscode z.B. beim Inspizieren des Speicherinhalts sichtbar werden.

Im Prinzip kann ein transparentes Verhalten hier auf dieselbe Weise erreicht werden, wie in dem in § 3.5.2.3 geschilderten Fall der durch das Rücksetzen veränderten TIDs: `Codex` könnte Anfragen nach dem Inhalt betroffener Speicherzellen abfangen und so manipulieren, daß sie die Werte der uninstrumentierten Programmversion zurückliefern. In der Praxis ist dies allerdings wegen des großen Umfangs der durch den Instrumentierungscode veränderten Speicherzellen zu aufwendig, so daß eine Transparenz noch nicht einmal für das Zielprogramm selbst hergestellt wird. Dieses Vorgehen ist in den vorliegenden Fällen insofern vertretbar, als die Instrumentierung nur Bibliotheksfunktionen verändert, über deren interne Implementierung die Anwendung bzw. der Programmierer keine Annahmen machen sollte. Somit können sich die durch die Instrumentierung bedingten Änderung dieser Funktionen nicht auf die Korrektheit der Anwendung auswirken. Bestimmte Fehler, wie z.B. Speicherzugriffe über nichtinitialisierte Zeiger, können jedoch aufgrund des veränderten Speicherlayouts durchaus unterschiedliche Auswirkungen zeigen, was bei allen existierenden Werkzeugumgebungen in Kauf genommen wird.

Etwas anders ist die Situation dagegen, falls der eigentliche Anwendungscode modifiziert wird, z.B. durch Einfügen von Code mittels `dyninst` [99, 241]. In diesem Fall sollten die Änderungen für Anwendung und Werkzeuge transparent sein, obwohl ihre Sichtbarkeit auch hier nur in seltenen Fällen negative Auswirkungen hat, da Anwendungen in der Regel nicht lesend auf ihren eigentlichen Programmcode zugreifen.

3.5.6 Implizite Annahmen über den Zielprogrammzustand

Die bisher beschrieben unerwünschten Wechselwirkungen lassen sich ohne wesentliche Änderungen am Quellcode der Werkzeuge beseitigen, falls diese Aufgabe durch eine adäquate Monitor- und Interoperabilitätsinfrastruktur unterstützt wird. In dieser Situation kann der in § 2.3.1 geäußerte Wunsch nach einer beliebigen, einfachen Kombination existierender Laufzeit-Werkzeuge erfüllt werden. In der Praxis, so auch in dem hier betrachteten Szenario, können sich jedoch auch Wechselwirkungen ergeben, die nicht allein mit Hilfe einer geeigneten Middleware beseitigt werden können, sondern nur durch maßgebliche Änderungen von Teilen der Werkzeugimplementierung.

Die beiden Probleme, die im folgenden diskutiert werden besitzen dabei eine gemeinsame Ursache: Die Realisierung eines Werkzeugs basiert auf impliziten Annahmen über das Zielsystem, die durch die Anwendung eines anderen Werkzeugs verletzt werden.

3.5.6.1 Erstellung von Sicherungspunkten

Die Sicherungspunkterstellung in `CoCheck` ist intern, d.h. sie erfolgt durch die Anwendungsprozesse selbst [235]. Der dazu notwendige Code wird durch die Instrumentierung der PVM-Bibliothek in die Anwendung eingeschleust. Er wird durch eine während der Initialisierung definierte Signalbehandlungsroutine aufgerufen und kann damit von außen über ein entsprechendes Signal aktiviert werden. Diese Vorgehensweise funktioniert problemlos, wenn `CoCheck` das einzige an die Anwendung angeschlossene Werkzeug ist. Sie geht aber implizit davon aus, daß die Prozesse – abgesehen von kurzen, zeitlich begrenzten Signalsperren – jederzeit Signale annehmen und behandeln können. Wird die Anwendung gleichzeitig durch einen Debugger gesteuert, können Prozesse jedoch von diesem für unbegrenzte Zeit angehalten werden. Eine Sicherungspunkterstellung ist in diesem Zustand dann nicht mehr möglich.

Eine Lösung dieses Problems wurde bereits bei der Integration von `CoCheck` in den OCM realisiert [235]. `CoCheck` muß dazu bei der Erstellung eines Sicherungspunkts den Zustand der Zielprozesse explizit abfragen. Falls ein Prozeß gestoppt ist, also nicht auf Signale reagieren kann, muß der Aufruf des notwendigen Codes in diesem Prozeß direkt durch einen *Inferior RPC* (siehe § 2.1.3.2, Seite 23) erfolgen. Dabei ist der Fall zu berücksichtigen, daß ein Prozeß innerhalb einer PVM-Funktion angehalten wurde. Er muß speziell behandelt werden, um einen unzulässigen Wiedereintritt in dieselbe PVM-Funktion zu vermeiden.

Die Berücksichtigung angehaltener Prozesse erforderte nicht unerhebliche Modifikationen der Implementierung von `CoCheck`, resultiert aber in einem nun uneingeschränkt verwendbaren Werkzeug.

3.5.6.2 Modellierung von Nachrichtenwarteschlangen

In § 3.2.4 wurde erläutert, daß `VISTOP` unter anderem den Inhalt der Nachrichtenwarteschlangen der überwachten PVM-Prozesse modelliert. Dieser wird aus Ereignissen berechnet, die beim Eintragen einer Nachricht in die Warteschlange bzw. beim Empfang und somit beim Entfernen einer Nachricht ausgelöst werden. Der Berechnung liegt die an sich berechtigte, aber implizite Annahme zugrunde, daß die Warteschlangen nach der FIFO-Strategie verwaltet werden.

Werkzeuge zur deterministischen Ablaufsteuerung, wie z.B. `Codex`, manipulieren jedoch die Empfangsreihenfolge von Nachrichten so, daß diese nicht mehr in derselben Ordnung empfangen werden, in der sie in die Warteschlange eingetragen wurden. Bei der gleichzeitigen Verwendung mit `Codex` modelliert `VISTOP` daher die Warteschlangen nicht mehr korrekt. Dies führt unter anderem dazu, daß bei der Visualisierung eines Nachrichtenempfangs ein falscher Senderprozeß angezeigt werden kann, da dieser aus der ersten laut modellierter Warteschlange empfangbaren Nachricht bestimmt wird. In der Praxis resultiert dies in der Folge in einer zwangsweisen Terminierung von `VISTOP`, da eine eincodierte Zusicherung verletzt wird.

Zur Lösung dieses Problems sind zwei Alternativen denkbar:

1. Eine Änderung der Visualisierungsmetaphern.

   Statt geordneter Warteschlangen könnte `VISTOP` lediglich ungeordnete Mengen empfangbarer Nachrichten darstellen. Die Visualisierung des Nachrichtenempfangs müßte dann so verändert werden, daß der Sender erst bei Abschluß der Operation angezeigt wird, da erst zu diesem Zeitpunkt aus dem aufgezeichneten Ereignis der tatsächliche Sender hervorgeht. Dadurch würde die Brauchbarkeit des Visualisierers jedoch deutlich reduziert, da z.B. bei einer blockierenden Empfangsoperation nicht mehr ersichtlich wäre, auf welchen Prozeß gewartet wird.

2. Eine modifizierte Erzeugung der Ereignisse, die den Eintrag einer Nachricht in eine Warteschlange anzeigen.

   Die Ereignisse müßten dabei von der instrumentierten PVM-Bibliothek so generiert werden, daß sie in derselben Reihenfolge auftreten, in der die Nachrichten später aus der Warteschlange entfernt und vom Zielprozeß empfangen werden. Die von `VISTOP` modellierte Warteschlange spiegelte damit die Sicht der Anwendungsprozesse korrekt wider.

Die zweite Variante kann realisiert werden, da `VISTOP` nur den logischen Ablauf des Zielprogramms visualisiert und damit nicht der genaue Zeitpunkt von Ereignissen relevant ist, sondern lediglich deren kausale Ordnung.

In der folgenden Diskussion wird das Ereignis, das die Ankunft einer Nachricht und damit deren Eintrag in eine Warteschlange anzeigt, mit a bezeichnet, das Ereignis, das den Beginn einer Empfangsoperation für diese Nachricht markiert, mit e. Beim Ereignis a ist dabei für `VISTOP` nur maßgebend, ob es vor oder nach e auftritt. Tritt a vor e auf, ist die Nachricht bereits vor dem Empfangswunsch eingetroffen und kann somit sofort empfangen werden. Wird umgekehrt zuerst e ausgelöst, muß die Empfangsoperation so lange blockieren, bis die Nachricht eintrifft, was durch a angezeigt wird. Beide Situationen werden von `VISTOP` entsprechend visualisiert. Im ersten Fall kann aber nun das Ereignis a ohne Einfluß auf die Funktion von `VISTOP` solange verzögert werden, bis feststeht, daß die betroffene Nachricht die gemäß des `Codex`-Kontrollmusters als nächstes zu empfangende ist. Letztere kann durch Aufruf der durch `Codex` instrumentierten Funktion `pvm_probe` bereits vor dem eigentlichen Empfang ermittelt werden. Das Ereignis wird dabei maximal bis unmittelbar vor e verzögert, da `Codex` spätestens beim Empfangswunsch über die zu empfangende Nachricht entscheiden muß. Im zweiten Fall ist diese Entscheidung bereits getroffen, so daß das Ereignis entsprechend erzeugt werden kann.

Die beschriebene Lösung kann zwar ohne Modifikation des eigentlichen Visualisierungswerkzeugs realisiert werden, erfordert aber eine deutliche Anpassung der von `VISTOP` benötigten Instrumentierung der PVM-Bibliothek.

3.5.7 Zusammenfassung

In den vorangegangenen Anschnitten wurden verschiedene Wechselwirkungen untersucht, die bei gleichzeitigem Einsatz der Werkzeuge `DETOP`, `CoCheck`, `Codex` und `VISTOP` auftreten. Sie werden weniger durch die grundlegende Funktion dieser Werkzeuge als vielmehr durch deren Implementierung bedingt. Die Diskussion hat gezeigt, daß zur Vermeidung der unerwünschten Wechselwirkungen die folgenden Anforderungen an die Werkzeuge sowie die zugrundeliegende Monitorinfrastruktur erfüllt sein müssen:

- Die Werkzeuge müssen dazu in der Lage sein, sich an bereits existierende Prozesse anzukoppeln. Dabei ist es ausreichend, wenn dies lediglich für noch nicht gestartete Prozesse möglich ist.

- Analog zu den in § 3.4.3 definierten direkten Werkzeuginteraktionen müssen Interaktionen zwischen Werkzeugen und dem Monitorsystem definiert werden können, so daß beispielsweise nach dem Wiederanlauf eines Prozesses durch `CoCheck` eine Aktion im OCM ausgeführt werden kann, die die Instrumentierungen wiederherstellt.

- Um Änderungen an Zustandskomponenten überwachter Objekte vor anderen Werkzeugen verbergen zu können, muß ein Werkzeug dazu in der Lage sein, Zugriffe auf diese Zustandskomponenten abfangen und gegebenenfalls manipulieren zu können.
- Analog müssen Werkzeuge befähigt werden, durch ihre Instrumentierung ausgelöste Ereignisse abzufangen, bevor andere Werkzeuge von diesen Kenntnis erlangen.
- Fundamental ist die Forderung nach einem Mechanismus, der es erlaubt, die Reihenfolge festzulegen, in der Aktionen verschiedener Werkzeuge für ein und dasselbe Ereignis ausgeführt werden. Dabei ist zu berücksichtigen, daß die Werkzeuge die Aktionen unabhängig voneinander definieren, also kein Wissen über die Aktionen anderer Werkzeuge besitzen.
- Schließlich verbleibt die Forderung an Werkzeugentwickler, implizite Annahmen über den Zustand des Zielsystems zu vermeiden. Idealerweise sollte die prinzipiell mögliche, gleichzeitige Anwendung anderer Werkzeuge bereits bei der Entwicklung berücksichtigt werden.

Kapitel 4

Eine Methodik zur Erstellung interoperabler Werkzeugumgebungen

Im letzten Abschnitt wurde als Fallstudie untersucht, wie basierend auf einem konkreten Satz von Werkzeugen eine interoperable Werkzeugumgebung realisiert werden kann. In diesem Kapitel werden die gewonnenen Erkenntnisse auf allgemeine Szenarien erweitert. In § 4.1 wird die in der Fallstudie zur Kombination der Werkzeuge angewandte *Top-Down*-Vorgehensweise in verallgemeinerter Form zusammengefaßt. Die Zusammenarbeit der Werkzeuge wird dabei zunächst aus Sicht eines Benutzers betrachtet (§ 4.2), anschließend werden die durch das gemeinsame Zielsystem bedingten Beeinflussungen der Werkzeuge analysiert (§ 4.3). Aufgrund der in der Fallstudie gesammelten Erfahrung werden am Ende des Abschnitts in § 4.4 Entwurfsrichtlinien für neu zu entwickelnde Werkzeuge angegeben, deren Einhaltung zu einer besseren Interoperabilität führt.

4.1 Vorgehensweise

Zur Erstellung einer Werkzeugumgebung auf der Basis vorhandener Werkzeuge wird eine Vorgehensweise in zwei Schritten vorgeschlagen:

1. Zunächst wird das gewünschte Verhalten der Werkzeugumgebung auf der Ebene der Benutzerschnittstelle definiert. Dabei stehen die Konsistenz der von den verschiedenen Werkzeugen dargestellten Information sowie die erwünschten Zusatzfunktionen im Vordergrund. Dieser Schritt führt zu abstrakten Verhaltenstypen und zu Verhaltensregeln für alle beteiligten Werkzeuge. Die Verhaltensintegration erfolgt dann auf Basis eines erweiterten Mediator-Konzepts, das ein dynamisches Einbinden der in den Werkzeugen benötigten Ereignis- und Aktions-Schnittstellen erlaubt. Die Rolle des Mediators übernimmt ein zentrales oder selbst wieder in Komponenten gegliedertes Rahmenwerkzeug, das auch die notwendigen Zusatzfunktionen der Werkzeugumgebung realisiert.

2. Anschließend sind die Implementierungen der Werkzeuge zu analysieren, um Konflikte durch den gemeinsamen Zugriff auf dasselbe Zielsystem zu ermitteln. Diese Konflikte werden dann mit Hilfe einer geeigneten Monitorinfrastruktur eliminiert. Soweit dies ohne konkrete Kenntnis der anderen Werkzeuge möglich ist, sollten die Einzelwerkzeuge bereits selbst die erforderlichen Maßnahmen zur Lösung möglicher Konflikte realisieren. Anderenfalls werden die notwendigen Maßnahmen zur Konfliktauflösung vom Rahmenwerkzeug veranlaßt.

Eine detailliertere Diskussion dieser Schritte folgt in den nächsten beiden Abschnitten.

4.2 Verhaltensdefinition

Sollen mehrere Werkzeuge zu einer Werkzeugumgebung kombiniert werden, dient dies letztendlich immer der besseren Unterstützung eines bestimmten Prozesses während der Software-Entwicklung, -Wartung oder -Nutzung. Demzufolge sollten bei der Realisierung einer solchen Kombination die Benutzeranforderungen im Vordergrund stehen.

Grundlegend ist zunächst die genaue Definition des Arbeitsprozesses, der durch die Werkzeugumgebung unterstützt werden soll, da dieser die erforderliche Zusammensetzung und das Gesamtverhalten der Umgebung bestimmt. In der in § 3 beschriebenen Fallstudie ist dieser Prozeß die zyklische Fehlersuche in Programmen mit hoher Ausführungszeit. Auf der Basis dieser Definition sind dann detailliertere Anforderungen festzulegen, die in die vier im folgenden diskutierten Kategorien unterteilt werden können. Die Reihenfolge der Aufzählung entspricht dabei dem Vorgehen beim Entwurf der Werkzeugumgebung.

4.2.1 Verwendungsmodus der Werkzeugumgebung

Hier ist zunächst festzulegen, ob die Werkzeuge der Umgebung zeitgleich gestartet werden müssen, oder ob diese vom Benutzer dynamisch bei Bedarf aktiviert und wieder beendet werden können. In beiden Fällen ist es zur besseren Benutzerführung sinnvoll, den Start der Einzelwerkzeuge und gegebenenfalls auch deren gemeinsame Beendigung durch ein übergeordnetes Rahmenwerkzeug mit eigener Bedienoberfläche zu steuern. Dies ist auch aus technischen Gründen notwendig, da eine Instanz existieren muß, die die Bestandteile der Umgebung kennt und koordiniert. Da wir erreichen wollen, daß die Einzelwerkzeuge kein Wissen über die anderen Werkzeuge oder über ihren eigenen Einsatz innerhalb einer Werkzeugumgebung besitzen müssen, kann diese Aufgabe nicht von einem dieser Werkzeuge übernommen werden.

4.2.2 Konsistenzkriterien

Bereits ohne Berücksichtigung eventueller Zusatzfunktionen basiert eine sinnvolle gleichzeitige Verwendung mehrerer Werkzeuge auf zwei notwendigen Bedingungen: Zum einen muß die Kombination technisch möglich sein. Dies soll an dieser Stelle zunächst vorausgesetzt werden. Implementierungstechnische Details werden erst im nächsten Schritt der Werkzeugintegration betrachtet (siehe § 4.3). Zum anderen dürfen die Werkzeuge unter keinen Umständen widersprüchliche Information über das Zielsystem darstellen. Um dies zu erreichen, müssen zunächst geeignete Konsistenzkriterien festgelegt werden. Dazu sind zwei Fragen zu beantworten:

1. Wann, d.h. unter welchen Voraussetzungen, zeigen die Einzelwerkzeuge Information über den aktuellen Zustand des Zielsystems an?

2. Wann gelten die von zwei verschiedenen Werkzeugen präsentierten Zustandsinformationen als identisch bzw. miteinander konsistent?

Die letzte Frage ist insofern nicht trivial, als Werkzeuge den Zustand des Zielsystems mit unterschiedlicher Granularität darstellen, d.h. verschiedene, nur zum Teil überlappende Teilaspekte dieses Zustands berücksichtigen.

Sind beide Fragen beantwortet, kann bestimmt werden, wie die Konsistenz der Anzeigen gesichert werden kann. Prinzipiell erfolgt eine Realisierung so, daß bei Auftreten eines Ereignisses, das zur Darstellung eines bestimmten Zustands in einem der Werkzeuge führt, eine Aktion in den anderen Werkzeugen ausgelöst wird, die deren Anzeigen konsistent aktualisiert oder, falls dies nicht möglich ist, als ungültig markiert.

4.2.3 Expliziter Informationsaustausch

Sobald die Werkzeuge in konsistenter Weise zusammen benutzbar sind, kann mit der Konzeption von zusätzlichen Funktionen begonnen werden, die über den kombinierten Funktionsumfang der einzelnen Werkzeuge hinausgehen. Ein erster Schritt hierzu ist die bessere Integration der Benutzungsoberfläche. So kann z.B. die Auswahl eines Objekts in Werkzeug A gleichzeitig auch in Werkzeug B dieses Objekt auswählen. Ebenso könnte ein Kommando zum Inspizieren des Objekts in A dazu führen, daß auch B Information über das Objekt ausgibt. Zur Realisierung derartiger Funktionen ist eine direkte Kommunikation zwischen den Werkzeugen erforderlich, wie sie in § 2.2.3.2 untersucht wurde. Auch hier müssen also beim Eintreten eines Ereignisses in einem Werkzeug Aktionen in anderen Werkzeugen ausgelöst werden.

4.2.4 Zusatzfunktionen

Der letzte Schritt der Integration besteht schließlich darin, höherwertige Funktionen auf der Basis der von den kombinierten Werkzeugen zur Verfügung gestellten Funktionalität zu realisieren und so deren Synergieeffekte auszunutzen. Im Sinne eines komponentenorientierten Software-Entwurfs sollten diese Funktionen durch eine von den Werkzeugen getrennte Komponente realisiert werden, die über eine geeignete Schnittstelle die Funktionen der Werkzeuge aufrufen kann. Diese kann im einfachsten Fall das Rahmenwerkzeug sein, bei komplexeren Funktionen ist wegen der besseren Wiederverwendbarkeit dagegen eine Realisierung als vollkommen eigenständige Komponente vorzuziehen. Die komponentenbasierte Implementierung sichert dabei die Unabhängigkeit der Werkzeuge, so daß diese auch nach wie vor außerhalb der Umgebung einsetzbar sind oder als Bestandteile anderer Werkzeugumgebungen fungieren können. Für eine andere Umgebung ist dabei lediglich ein neues Rahmenwerkzeug erforderlich.

4.2.5 Resultat der Verhaltensdefinition

Der zweite und dritte Schritt führt in natürlicher Weise zu Ereignis/Aktions-Regeln der Form „wenn in Werkzeug A x passiert, muß in Werkzeug B y ausgeführt werden“. Der vierte Schritt hat dagegen zur Voraussetzung, daß Aktionen innerhalb eines Werkzeugs von einer anderen Komponente, nämlich dem Rahmenwerkzeug, direkt aufgerufen werden können.

Insgesamt wird es also erforderlich, für jedes der beteiligten Werkzeuge externe Schnittstellen im Sinne der von Sullivan [229] definierten abstrakten Verhaltenstypen (siehe § 2.2.3.2, Seite 32) zu definieren, die mögliche Ereignisse innerhalb des Werkzeugs sowie die von ihm angebotenen Aktionen beschreiben. Wie prinzipiell alle Schnittstellen sollten diese so entworfen werden, daß sie minimal, aber trotzdem allgemein verwendbar sind. Bereits vorhandene Schnittstellen sollten soweit möglich wiederverwendet werden. Es ist an dieser Stelle wichtig, festzuhalten, daß es hier zunächst nur um die abstrakte Definition der Schnittstellen geht, deren technische Realisierung also noch vollkommen irrelevant ist.

Das Resultat des beschriebenen Prozesses, der i.w. nur die Benutzeranforderungen berücksichtigt, besteht damit aus vier Teilen:

1. Einem Rahmenprogramm, das Start und Koordination der Werkzeugumgebung realisiert,
2. abstrakten Verhaltenstypen aller beteiligter Werkzeuge, die die Schnittstellen relevanter Ereignisse und Aktionen innerhalb dieser Werkzeuge festlegen,
3. darauf aufbauenden Ereignis/Aktions-Regeln, die das Verhalten der Werkzeugumgebung beschreiben,
4. zusätzlichen Funktionen, die in einer eigenen Komponente, bzw. auch dem Rahmenprogramm realisiert sind und Aktionen der Werkzeuge benutzen.

Dies entspricht dem in § 2.2.3.2, Seite 32 beschriebenen Mediator-Ansatz. Die Erweiterung des Ansatzes in Form von Interaktoren ist hier nicht notwendig, da die Forderung nach einer auch individuellen Verwendbarkeit der Werkzeuge impliziert, daß diese keine Methoden von ihrer Umgebung (d.h. den anderen Werkzeugen) benötigen dürfen. Lediglich das Rahmenprogramm bzw. die Zusatzfunktionen aus Punkt 4 können externe Methoden aufrufen. Werden diese Zusatzfunktionen durch mehrere Komponenten realisiert, bietet sich für deren Kommunikation jedoch das Interaktor-Konzept an.

In bezug auf die Klassifikation aus § 2.2.2 bewirkt der in diesem Abschnitt beschriebene Prozeß in erster Linie die Verhaltensintegration der Werkzeuge, obwohl die Überlegungen zur Konsistenz der Darstellung auch Aspekte der Datenintegration berühren.

4.3 Ermittlung und Behandlung wechselseitiger Beeinflussungen

Ist die grundsätzliche Funktion der Werkzeugumgebung festgelegt, kann deren technische Realisierung betrachtet werden. Es sind nun also die konkreten Werkzeug-Implementierungen zu analysieren, insbesondere auch deren Monitorkomponenten, die verwendeten Instrumentierungstechniken sowie die von diesen durchgeführten Modifikationen des Zielsystemzustands. Dabei müssen strukturelle und logische Konflikte (siehe § 2.3.2) zunächst erkannt und anschließend entweder vermieden oder geeignet behandelt werden.

4.3.1 Strukturelle Konflikte

Nach Trinitis ([235], S. 45) werden strukturelle Konflikte bereits durch unterschiedliche Werkzeug-Infrastrukturen, d.h. Monitorsysteme, an sich verursacht, selbst wenn die Werkzeuge noch überhaupt keine Aktionen auf dem Zielsystem ausführen. Trinitis führt aus, daß diese Konflikte nur durch die Verwendung eines gemeinsamen Monitorsystems zufriedenstellend gelöst werden können. Bei existierenden Werkzeugen stellen sie daher das größte Hindernis für deren Interoperabilität dar, da die Werkzeuge zur Konfliktvermeidung an eine neue, gemeinsame Monitorschnittstelle angepaßt werden müssen, was i.a. einen erheblichen Aufwand bedeutet. Dies ist ein gewichtiges Argument für die Verwendung von „Standard"-Monitorschnittstellen wie z.B. OMIS.

Glücklicherweise ist diese Aussage jedoch bei genauerer Betrachtung in einigen Fällen etwas zu pessimistisch. Die nachfolgende Diskussion wird dies verdeutlichen.

4.3.1.1 Exklusive Betriebsmittel

Ein tatsächliches Problem stellen Betriebsmittel dar, die nur exklusiv von jeweils einem Monitorsystem genutzt werden können.

Viele Werkzeuge bzw. Monitorsystem erfordern das Binden des Zielprogramms mit vorinstrumentierten Versionen der relevanten Programmierbibliotheken. Diese stellen ein exklusives Betriebsmittel dar, da ein Programm nur jeweils mit einer derartigen Bibliothek gebunden werden kann. Eine einfache Abhilfe besteht hier darin, nicht vorgefertigte instrumentierte Bibliotheken zu verwenden, sondern das Werkzeug mit einem Hilfsprogramm auszuliefern, das die notwendige Instrumentierung z.B. mittels *Binary Wrapping* in eine vorhandene Bibliothek einfügt, die dann zum Zielprogramm gebunden wird. Die Bibliothek stellt dann kein exklusives Betriebsmittel mehr dar, da sie vor dem Binden nacheinander durch mehrere Werkzeuge instrumentiert werden kann. Natürlich kann dabei nicht ausgeschlossen werden, daß die eingefügten Instrumentierungen untereinander konfligieren. Diese logischen Konflikte können aber mit den später beschriebenen Techniken aufgelöst werden.

Ein weiteres exklusives Betriebsmittel vieler Monitorsysteme sind Betriebssystem-Schnittstellen zur Prozeßkontrolle. Diese können tatsächlich nur durch eine gemeinsame Verwaltungsschicht für mehrere Prozesse gleichzeitig nutzbar gemacht werden. Die Situation ist dabei vollkommen analog zur Virtualisierung der Betriebsmittel CPU und Hauptspeicher durch ein Betriebssystem. Daraus wird deutlich, daß eigentlich auch die Virtualisierung der Prozeßkontrolle Aufgabe des Betriebssystems wäre. Immerhin ist bei UNIX ein erfreulicher Trend von der vollkommen exklusiven Schnittstelle `ptrace` zu dem zumindest teilweise von mehreren Prozessen nutzbaren `/proc` Dateisystem bemerkbar. Da aber auch `/proc` noch nicht vollständig virtualisiert ist, können Werkzeuge, die diese Schnittstelle nutzen, derzeit nur dann simultan verwendet werden, wenn sie über ein gemeinsames Monitorsystem verfügen.

4.3.1.2 Andere Konflikte

Alle anderen strukturellen Konflikte zwischen Werkzeugen werden bei genauer Analyse durch logische Konflikte verursacht, die allerdings auf der Ebene des Monitorsystems zutage treten. Sie kommen dadurch zustande, daß die Monitorsysteme auf gemeinsame Objekte des Zielsystems zugreifen und deren Zustand in einer Weise verändern, die zu einem Fehlverhalten der anderen Monitorsysteme führt. Das bedeutet, daß Techniken zur Lösung logischer Konflikte zwischen Werkzeugen auch diese Art der strukturellen Konflikte beseitigen können, ohne daß notwendigerweise für alle Werkzeuge ein gemeinsames Monitorsystem erforderlich wird.

Falls die Monitore kritische Zustandsänderungen zur Laufzeit vornehmen, müßten dazu allerdings entweder explizit Schnittstellen für Ereignisse und Aktionen zur Koordination der Monitorsysteme vorgesehen werden, oder aber ein Meta-Monitorsystem, das derartige Schnittstellen zur Laufzeit einfügen kann. Ob der Aufwand dafür letztendlich geringer ist als für eine Anpassung der Werkzeuge an eine standardisierte Monitor-Schnittstelle ist jedoch fraglich.

Für den Fall, daß zur Laufzeit keine kritischen Zustandsänderungen erfolgen, insbesondere, falls ein Monitorsystem ausschließlich auf einer statischen Bibliotheksinstrumentierung beruht, können Konflikte jedoch oft in einfacher Weise durch die Wahl einer geeigneten Instrumentierungsreihenfolge vermieden werden. Dies hat das Beispiel `Codex` in der Fallstudie verdeutlicht (siehe § 3.5.3).

Wir werden daher nachfolgend die Erkennung, Vermeidung und Behandlung logischer Konflikte näher untersuchen, wobei die Ergebnisse sowohl auf Wechselwirkungen zwischen Werkzeugen als auch zwischen Monitorsystemen anwendbar sind. Die Lösung dieser Konflikte entspricht dabei im wesentlichen dem Vorgang der Datenintegration.

4.3.2 Logische Konflikte

Wie durch die Beispiele in § 3.5 verdeutlicht wurde, entstehen logische Konflikte dadurch, daß ein Werkzeug das Zielsystem in einer Weise beeinflußt, die die Funktion anderer Werkzeuge beeinträchtigt. Mögliche Konflikte zwischen gegebenen Werkzeugen lassen sich in systematischer Weise ermitteln, wenn eine geeignete Modellierung des Zielsystems sowie der Zugriffe durch die Werkzeuge vorgenommen wird.

4.3.2.1 Konfliktmodellierung nach Trinitis

Die im Umfeld der vorliegenden Arbeit entstandene Dissertation von Trinitis [235] definiert ein Modell des Zielsystems, das grobe Aussagen über logische Konflikte zwischen Werkzeugen zuläßt. Das Zielsystem wird dabei zunächst als Menge von ***Objekten*** strukturiert, wobei eine konkrete Definition offen gelassen wird und auch nicht erforderlich ist. Werkzeuge können auf Objekte mittels ***Lese- und Schreiboperationen*** zugreifen, die mit der üblichen Semantik definiert sind. Für die Analyse logischer Konflikte sind jedoch nicht nur die durchgeführten Operationen an sich relevant, vielmehr muß auch berücksichtigt werden, wie lange deren Auswirkungen auf das Zielsystem bzw. ein Werkzeug fortbestehen. Zu diesem Zweck werden Lese- und Schreibzugriffe definiert, denen wie folgt eine Zeitdauer zugeordnet ist:

- Ein ***Lesezugriff*** auf ein Objekt beginnt einer Leseoperation und endet, wenn das Objekt terminiert oder das Werkzeug das Resultat der Leseoperation nicht mehr benutzt.

- Ein ***Schreibzugriff*** beginnt mit einer Schreiboperation auf ein Objekts und endet, wenn das Objekt terminiert oder das Werkzeug den geschriebenen Zustand nicht mehr als gültig annimmt und der Objektzustand so wiederhergestellt wird, als ob die Schreiboperation nie stattgefunden hätte.

Ein Konflikt tritt nun dann auf, wenn ein Zugriff auf ein Objekt zeitlich mit einem Schreibzugriff auf daselbe Objekt überlappt. Trinitis unterscheidet zwischen zwei Arten von Konflikten:

- Führt ein Werkzeug B einen Schreibzugriff auf ein Objekt x durch, während ein beliebiger Zugriff eines anderen Werkzeugs A auf x noch aktiv ist, so ergibt sich ein ***Konsistenzproblem*** für A. Durch den Schreibzugriff wird die Annahme von A verletzt, daß x noch den von A gelesenen oder geschriebenen Wert besitzt.

- Führt ein Werkzeug B einen beliebigen Zugriff auf ein Objekt x durch, während ein Schreibzugriff eines anderen Werkzeugs A auf x noch aktiv ist, so ergibt sich ein ***Transparenzproblem*** für B. Das Werkzeug sieht die Änderung von A anstelle des Originalzustands.

Obwohl diese einfache Analysemethode bei der Integration von `CoCheck` und `DETOP` in [235] erfolgreich angewandt wurde, besitzt sie drei deutliche Unzulänglichkeiten:

1. Sie enthält implizite Annahmen darüber, ob eine durch ein Werkzeug vorgenommene Änderung des Zielsystemzustands durch andere Werkzeuge beobachtbar sein soll. Bei der

Definition von Konsistenzproblemen wird davon ausgegangen, daß die Antwort auf diese Frage „ja“ ist, bei der Definition von Transparenzproblemen jedoch von der gegenteiligen Antwort. Die Analyse ist daher erheblich zu konservativ und liefert in der Regel Konflikte, die in Wirklichkeit nicht vorhanden sind.

2. Die Analysemethode berücksichtigt nicht, daß Schreibzugriffe auf Zustandskomponenten in Folge zu Änderungen an weiteren Zustandskomponenten führen können. Falls die ursprüngliche Änderung vor den anderen Werkzeugen verborgen werden soll, muß dies aber auch für deren Folgen gelten.

3. Schließlich liefert die Methode keinerlei Hinweise darauf, wie mögliche Konflikte aufgelöst werden können.

Diese Defizite werden im nachfolgenden Unterabschnitt durch eine verfeinerte und erweiterte Modellierung beseitigt.

4.3.2.2 Erweiterte Konfliktmodellierung

Objekte und Zustandskomponenten. Um das Modell besser an die übliche Vorstellung von Objekten in der Informatik anzupassen, unterscheiden wir zunächst zwischen Objekten, die verschiedenen Klassen angehören können, z.B. Knoten oder Prozesse, ihrem Zustand sowie einzelnen ***Zustandskomponenten***. Zugriffe erfolgen dabei immer auf Zustandskomponenten.

Wie auch bei Trinitis beinhaltet das Modell keine konkrete Definition der Objektklassen, des Zustands von Objekten sowie dessen Aufteilung in Komponenten, da diese für den Prozeß der Konfliktermittlung ohne Belang ist. Die Modellierung des Zielsystems kann also mit beliebiger Granularität erfolgen, so daß eine Abwägung zwischen Modellierungsaufwand und Genauigkeit der Ergebnisse möglich wird. Eine grobgranulare Modellierung, bei der z.B. der Zustand von Prozessen nicht weiter untergliedert wird, führt dabei zu weniger exakten, jedoch stets konservativen Ergebnissen. Es werden also möglicherweise Konflikte erkannt, die bei einer feinergranularen Analyse nicht mehr auftreten. Andererseits kann eine derart grobe Modellierung ausreichen, um die Anwesenheit von Konflikten auszuschließen, z.B. wenn zwei Werkzeuge disjunkte Prozeßmengen modifizieren. Dadurch wird ein schrittweises Vorgehen möglich, bei dem zunächst mit einer grobgranularen Modellierung Konfliktkandidaten bestimmt werden, die anschließend mit einem verfeinerten Zustandsmodell im Detail untersucht werden. Diese Vorgehensweise wurde in § 3.5.2.2 bei der Analyse der durch `CoCheck` verursachten Konflikte angewandt, wo zunächst das gesamte Datensegment eines Prozesses als eine Zustandskomponente aufgefaßt und später in einzelne relevante Datenstrukturen verfeinert wurde.

Zur Vereinfachung der Analyse könnte gefordert werden, daß die Zustandskomponenten aller Objekte disjunkt sind, eine Schreiboperation auf eine Komponente eines Objekts also nicht gleichzeitig auch eine andere Komponente eines (möglicherweise anderen) Objekts modifiziert. In der nachfolgenden Diskussion wird jedoch deutlich werden, daß bei der Analyse in jedem Fall auch indirekte Auswirkungen von Schreiboperationen berücksichtigt werden müssen, so daß auf diese Forderung auch verzichtet werden kann.

Schnittstellen und Funktionen. Die Fallstudie in § 3 hat verdeutlicht, daß Werkzeuge neben dem Zustand des Zielprogramms vor allem auch Aufrufe von Bibliotheksfunktionen überwachen und gegebenenfalls modifizieren. Diese Tatsache muß in der Konfliktmodellierung berücksichtigt werden. Wir betrachten daher in dem Modell eine Hierarchie von Funktionen, über die das Zielprogramm letztendlich auf Zustandskomponenten zugreift. Dabei wird nicht vorausgesetzt, daß der Zugriff ausschließlich über diese Funktionen erfolgt, d.h. es werden zusätzlich auch direkte Lese- und Schreiboperationen auf die Zustandskomponenten berücksichtigt. Dies führt zu einer Modellierung des Zielsystem, wie sie beispielhaft in Abb. 4.1 dargestellt ist .

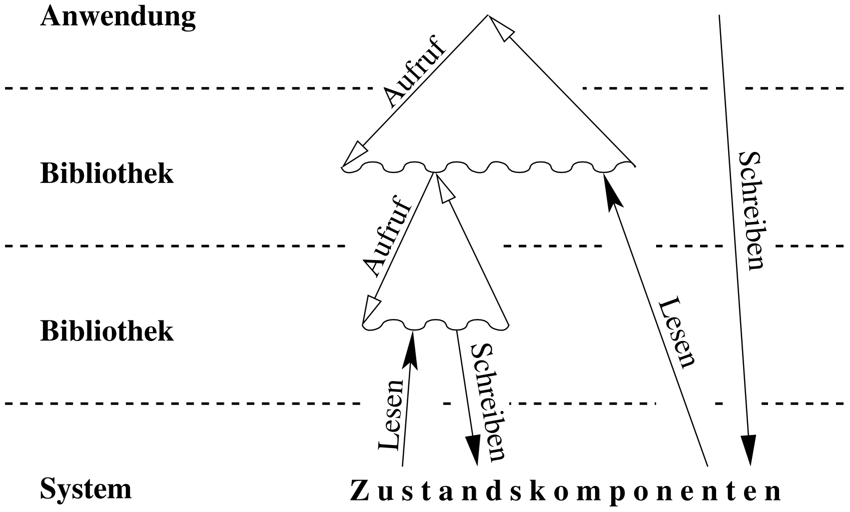


Abbildung 4.1: Modellierung des Zielsystems

Aktionen der Werkzeuge. Laufzeit-Werkzeuge beobachten und manipulieren dieses Zielsystem in der Regel auf der Ebene genau einer Schnittstelle. Im Beispielszenario aus § 3 ist dies die Schnittstelle zum PVM-System. Arbeitet ein Werkzeug auf mehreren Schnittstellen zugleich, kann es zum Zweck der Modellierung in mehrere logische Werkzeuge aufgeteilt werden. Unter der Voraussetzung, daß eine geeignete Infrastruktur zur Unterstützung der Interoperabilität vorhanden ist, ist meist auch eine reale Aufteilung möglich und wegen der besseren Modularisierung auch empfehlenswert.

Untersucht ein Werkzeug das Zielsystem auf Ebene einer Schnittstelle, unterscheidet es die Teilsysteme oberhalb und unterhalb dieser Schnittstelle. Das obere Teilsystem, dessen Aufrufe und Zugriffe überwacht werden, ist aus Sicht des Werkzeugs die Anwendungsebene, die anderen, nicht näher überwachten unteren Schichten die Systemebene. Da verschiedene Werkzeuge das Zielsystem aber auf jeweils unterschiedlichen Ebenen überwachen können, ist diese Aufteilung nicht notwendigerweise einheitlich. So könnte in dem Beispielszenario anstelle von `CoCheck` ein System zur Sicherungspunkterstellung auf der Ebene der Systemaufrufe, wie z.B. ‹PBEAM [186], eingesetzt werden. Für ‹PBEAM wäre dann die PVM-Bibliothek ein Teil der Anwendung, während sie z.B. für `Codex` einen Teil der Systemebene darstellt.

Das Modell erlaubt nun zunächst für jedes einzelne Werkzeug eine genauere Klassifizierung der von ihm durchgeführten Aktionen, auf deren Basis dann mögliche Konflikte erkannt werden können. Ein Werkzeug kann

1. Zustandskomponenten lesen.
2. Zustandskomponenten modifizieren. Dies kann so erfolgen, daß die Modifikation
   (a) für die Anwendung sichtbar ist,
   (b) für die Anwendung unsichtbar ist.

3. Funktionsaufrufe der beobachteten Schnittstelle überwachen und ihre Ein- bzw. Ausgabeparameter
   (a) lesen,
   (b) verändern. Dadurch kann ein Werkzeug insbesondere die Auswirkungen der Modifikation einer Zustandskomponente vor der Anwendung verbergen.

Bei einer Analyse möglicher Konflikte, die durch eine modifizierende Aktionen eines Werkzeugs verursacht werden, muß wie erwähnt berücksichtigt werden, ob die Auswirkungen dieser Modifikation für ein anderes Werkzeug sichtbar sein sollen oder nicht. Die konkrete Antwort auf diese Frage hängt jedoch nicht nur von der Art der Modifikation, sondern auch von *beiden* betrachteten Werkzeugen ab. Im Beispielszenario aus § 3 muß etwa die Änderung der Eingabeparameter von `pvm_recv`, die durch `Codex` vorgenommen wird, für `CoCheck` sichtbar sein, während sie für `VISTOP` transparent sein muß.

Hier stellt sich die Frage, ob es eine Charakterisierung der Aktionen gibt, die nur vom ausführenden Werkzeug abhängt, derart, daß aus den Attributen einer lesenden Aktion R und einer schreibenden Aktion W ableitbar ist, ob W für R sichtbar ist oder nicht. Diese Frage bestimmt zum einen entscheidend den Aufwand der Konfliktanalyse, da eine derartige Charakterisierung für jeden Einsatz eines Werkzeugs in einer Werkzeugumgebung wiederverwendet werden kann, während ansonsten jede Aktion immer wieder in bezug auf jeweils andere Werkzeuge zu untersuchen wäre. Sie ist auch essentiell, um auf Basis des Modells Methoden zur Konfliktvermeidung und -behandlung in einer Monitorinfrastruktur realisieren zu können. Eine der Grundvoraussetzungen dieser Arbeit ist ja, daß die in einer Umgebung zusammengefaßten Werkzeuge kein Wissen übereinander besitzen müssen. Ob eine durch ein Werkzeug A verursachte Modifikation für ein anderes Werkzeug B sichtbar sein soll, kann somit weder von A noch von B alleine, sondern nur – aufgrund geeigneter Attribute – von der Infrastruktur entschieden werden.

Eine Möglichkeit, dieses Ziel zu erreichen, besteht darin, auch lesende Aktionen dahingehend zu klassifizieren, auf welcher Ebene diese (bei möglicher Anwesenheit anderer, modifizierender Werkzeuge) Daten erfassen sollen. Das Lesen einer Zustandskomponente kann somit auf zwei Ebenen erfolgen:

1. Auf Anwendungsebene, d.h. das Werkzeug will denselben Wert sehen, den auch die Anwendung sieht.

2. Auf Systemebene, d.h. das Werkzeug will den „tatsächlichen“ Wert der Komponente lesen, der auf Systemebene sichtbar ist.

Diese beiden Werte können sich aufgrund von Modifikationen anderer Werkzeuge unterscheiden, die für die Anwendung transparent gemacht werden. Eine analoge Einteilung kann bei der Überwachung von Funktionen für Ein- und Ausgabeparameter vorgenommen werden.

Konflikte durch Zugriffe auf Zustandskomponenten. In dem Modell kann nun jedes Werkzeug vier Arten von Operationen auf Zustandskomponenten durchführen:

r_A: Eine Leseoperation, die den Wert einer Zustandskomponente ermittelt, wie ihn auch die untersuchte Anwendung sieht.

r_S: Eine Leseoperation, die den „tatsächlichen“ Wert einer Zustandskomponente ermittelt, der auf der Systemebene sichtbar ist.

w_A: Eine Schreiboperation, die für die Anwendung sichtbar ist.

w_S: Eine Schreiboperation, deren Auswirkungen (durch geeignete Maßnahmen des Werkzeugs) von der Anwendung nicht wahrgenommen werden.

Diese vier Klassen können dabei von dem Werkzeug, das die Operation ausführt, ohne Wissen über eventuelle andere Werkzeuge unterschieden werden.

Wie Trinitis definieren wir entsprechende Zugriffe, die im Gegensatz zu den Operationen eine Zeitdauer besitzen, wobei wir die Definitionen etwas präzisieren:

- Ein ***Lesezugriff*** R_A (R_S) auf eine Zustandskomponente beginnt mit einer Leseoperation r_A (r_S) und endet, wenn die Komponente nicht mehr existiert oder das Werkzeug nicht mehr annimmt, daß sie noch den gelesenen Wert besitzt.

- Ein ***Schreibzugriff*** W_A (W_S) auf eine Zustandskomponente beginnt mit einer Schreiboperation w_A (w_S) und endet, wenn die Komponente nicht mehr existiert oder sie einen neuen Wert annimmt, der nicht kausal von dem geschriebenen Wert abhängt. Die zweite Bedingung ist z.B. dann erfüllt, wenn der alte Wert der Komponente restauriert wird oder diese einen Wert erhält, den sie auch ohne die Schreiboperation bekommen hätte.

Wir nehmen zunächst an, daß alle beteiligten Werkzeuge das Zielprogramm auf derselben Abstraktionsebene überwachen. Diese Annahme ist in vielen praktischen Fällen erfüllt, da die meisten Laufzeit-Werkzeuge eine Anwendung auf der Ebene der Schnittstelle zu den Programmierbibliotheken untersuchen. Betrachtet man in diesem Fall zwei zeitlich überlappende Zugriffe, so ergeben sich deutlich weniger Konflikte als in dem einfacheren Modell von Trinitis, wie Tabelle 4.1 zeigt. Wir unterscheiden neben Konsistenz- und Transparenzproblemen explizit auch Schreib/Schreib-Konflikte, da diese in der Regel gesondert betrachtet werden müssen.

1. Zugriff \ 2. Zugriff	R_A	R_S	W_A	W_S
R_A	+	+	K	+
R_S	+	+	K	K
W_A	+	+	S	S
W_S	T	+	S	S

+ = kein Konflikt

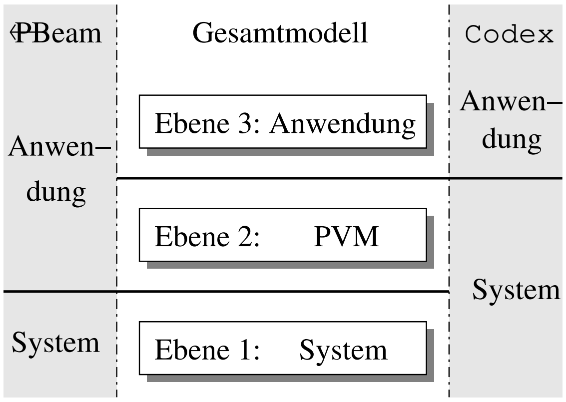
K = Konsistenzproblem

T = Transparenzproblem

S = Schreib/Schreib–Konflikt

Tabelle 4.1: Konflikte zwischen überlappenden Zugriffen auf Zustandskomponenten

Beobachten die Werkzeuge das Zielsystem auf unterschiedlichen Ebenen, ist eine genauere Differenzierung der Zugriffe vonnöten. Betrachtet man das oben genannte Beispiel von PBEAM und `Codex`, so ergibt sich eine Situation, wie in Abb. 4.2 dargestellt. Die Zugriffe finden also auf drei Schichten statt, was zu einer entsprechend erweiterten Konfliktmatrix führt. Allgemein kann das Modell auf derartige Situationen erweitert werden, indem für n Schichten n unterschiedliche Schreib- und Leseoperationen w_i bzw. r_i $(i = 1 \dots n)$, sowie entsprechende Zugriffe W_i und R_i betrachtet werden. r_i liefert dabei den Wert einer Zustandskomponente, wie ihn Schicht i sieht, w_i modifiziert die Komponente derart, daß die Änderung nur von den Schichten $i, i-1, \dots 1$ bemerkt wird. Die Konfliktmatrix ergibt sich dabei direkt aus mehrfacher Anwendung von Tabelle 4.1.



1. \ 2.	R_3	R_2	R_1	W_3	W_2	W_1
R_3	+	+	+	K	+	+
R_2	+	+	+	K	K	+
R_1	+	+	+	K	K	K
W_3	+	+	+	S	S	S
W_2	T	+	+	S	S	S
W_1	T	T	+	S	S	S

Abbildung 4.2: Überwachung eines System auf drei Ebenen und entstehende Konfliktmatrix

Eine Unschönheit bei dieser Vorgehensweise ist, daß kein für alle denkbaren Werkzeuge gültiges, universelles Schichtenmodell angegeben werden kann. Ein Modell wie in Abb. 4.2 ist daher erst bestimmbar, wenn alle Werkzeuge der Umgebung bekannt sind, da es Wissen über die Funktion und Implementierung dieser Werkzeuge beinhaltet. Ist dieses allerdings erst einmal erstellt, kann in praktisch allen Fällen die für die Werkzeuge bereits vorgenommene Klassifikation der Zugriffe wiederverwendet werden, indem die von einem Werkzeug definierte Schnittstelle zwischen Anwendung und System auf die entsprechende Schnittstelle des Gesamt-Schichtenmodells abgebildet wird. Zugriffe auf Anwendungs- bzw. Systemebene werden dann auf Zugriffe direkt ober- bzw. unterhalb dieser Schnittstelle abgebildet. Im Beispiel PBEAM und `Codex` ergibt sich dabei die Abbildung in Tab. 4.2.

PBEAM	`Codex`
$R_A \mapsto R_2$	$R_A \mapsto R_3$
$R_S \mapsto R_1$	$R_S \mapsto R_2$
$W_A \mapsto W_2$	$W_A \mapsto W_3$
$W_S \mapsto W_1$	$W_S \mapsto W_2$

Tabelle 4.2: Abbildung lokaler auf globale Zugriffsebenen für das Beispiel in Abb. 4.2

Diese Vorgehensweise impliziert, daß eine durch einen W_S-Zugriff von PBEAM vorgenommene Modifikation von einem R_S-Zugriff durch `Codex` nicht wahrgenommen werden soll. Dies ist insofern gerechtfertigt, als `Codex` auf einer höheren Abstraktionsebene angesiedelt ist als PBEAM. Offensichtlich muß diese Implikation jedoch nicht immer zwangsläufig richtig sein, d.h. das Modell ist nicht unbedingt in jeder Situation unmittelbar anwendbar. Ein universell anwendbares Modell ist jedoch auch nicht realisierbar, da beliebig komplexe (und unsystematische) Werkzeuge denkbar sind, die sich jedem Versuch einer einfachen, einheitlichen Beschreibung entziehen.

Konflikte durch Überwachung von Funktionsaufrufen. Logische Konflikte bei der Überwachung von Funktionsaufrufen können entstehen, wenn ein Werkzeug deren Ein- bzw. Ausgabeparameter modifiziert. Analog zur oben genannten Vorgehensweise bei Zustandskomponenten können auch hier unterschiedliche Lese- und Schreiboperationen und darauf aufbauende

Zugriffe definiert werden. Zu beachten ist in diesem Fall jedoch, daß die Ein- und Ausgabeparameter temporäre Objekte sind, die ausschließlich der Informationsübertragung zwischen Anwendungs- und Systemebene dienen. Eingabeparameter werden also von der Anwendung gesetzt und nur vom System gelesen, Ausgabeparameter entsprechend umgekehrt. Damit ist eine Unterscheidung zwischen w_A und w_S-Operationen unsinnig, da alle Schreiboperationen immer nur auf einer Ebene sichtbar sind. Bei Leseoperationen ist dagegen relevant, ob ein Werkzeug beobachten will, welche Werte die Anwendung an das System liefert bzw. von ihm zurückbekommt, oder ob beobachtet werden soll, welche Daten das System als Eingabe erhält bzw. als Ausgabe an die Anwendung weitergibt. Diese beiden Sichtweisen sind nicht identisch, wenn andere Werkzeuge die Parameter modifizieren können. Wir unterscheiden daher drei Operationen:

1. r_A sieht den Wert eines Parameters so wie die Anwendung, die die überwachte Funktion aufgerufen hat,

2. r_S liest den Wert eines Parameters, den die aufgerufene Funktion sieht,

3. w modifiziert den Wert eines Parameters.

Basierend auf diesen Operationen werden analog zu Seite 120 die Zugriffe R_A, R_S und W definiert. Da Ein- und Ausgabeparameter entgegengesetzte Datenflußrichtungen festlegen, entstehen zwei unterschiedliche Konfliktmatrizen, die in Tabelle. 4.3 dargestellt sind.

1. Zugriff \ 2. Zugriff	R_A	R_S	W
R_A	+	+	+
R_S	+	+	K
W	T	+	S

a) Aufruf: Eingabeparameter

1. Zugriff \ 2. Zugriff	R_A	R_S	W
R_A	+	+	K
R_S	+	+	+
W	+	T	S

b) Rückkehr: Ausgabeparameter

Tabelle 4.3: Konflikte zwischen überlappenden Zugriffen auf Funktionsparameter

Bemerkenswert ist, daß so definierte Konflikte nur zwischen Werkzeugen auftreten, die das Zielprogramm auf derselben Ebene betrachten, da sie nur dann auch dieselben Funktionen überwachen können. Bei einer Überwachung auf unterschiedlichen Ebenen ergibt sich die wechselseitige Sichtbarkeit von Modifikationen automatisch durch die Aufrufhierarchie der betroffenen Funktionen. Natürlich ist auch hier der Fall denkbar, daß sich durch die Veränderung der Eingabeparameter einer Funktion f_i in Schicht i durch ein Werkzeug A die Eingabeparameter einer Funktion f_j in Schicht $j < i$ so verändern, daß ein Werkzeug B, das f_j überwacht, nicht mehr korrekt funktioniert. Wie jedoch bereits erläutert wurde, ist ein alle theoretisch denkbaren Situationen erfassendes Modell nicht erstellbar. Entscheidend ist vielmehr, daß die in der Praxis auftretenden Fälle erfaßt werden, was beim vorliegenden Modell der Fall ist. So ist das genannte Beispiel insofern akademisch, als die Schicht $j + 1$ den Aufruf von f_j auch von sich aus mit den geänderten Parametern hätte durchführen können, worauf Werkzeug B eingestellt sein sollte.

Konfliktvermeidung. Wird durch die Konfliktmodellierung ein möglicher Konflikt erkannt, so muß dieser entweder vermieden oder behandelt werden. Eine Konfliktvermeidung ist bei Konsistenz- und Transparenzproblemen möglich, wenn die Reihenfolge der betroffenen Zugriffe gesteuert werden kann. Dies wird aus den Konfliktmatrizen ersichtlich, da eine Umkehr der Zugriffsreihenfolge einer Spiegelung an der Hauptdiagonalen entspricht. Beispielsweise ergibt bei einem Funktions-Eingabeparameter eine Überlappung W-R_A, d.h. ein R_A-Zugriff startet, während ein W-Zugriff noch aktiv ist, ein Transparenzproblem (siehe Tab. 4.3a), während eine Überlappung R_A-W ohne Probleme möglich ist.

Falls also a-priori bekannt ist, daß zur selben Zeit mehrere, konfligierende Zugriffe durchzuführen sind, kann deren Reihenfolge so festgelegt werden, daß Konsistenz- und Transparenzprobleme vermieden werden. Dies ist in der Regel nur bei Zugriffen auf Funktionsparameter der Fall, da nur hier der exakte Zeitpunkt der Zugriffe (Aufruf bzw. Rückkehr der Funktion) a-priori bekannt ist. Die in § 3.5.3 zunächst eher intuitiv ermittelte Reihenfolge der Bibliotheksinstrumentierung kann somit durch das Konfliktmodell auch formal nachvollzogen werden. Betrachten wir als Beispiel die weiter oben erwähnte Änderung der Eingabeparameter von `pvm_recv` durch `Codex`, die einen W-Zugriff darstellt. `CoCheck` benötigt Information auf Systemebene (R_S-Zugriff), `VISTOP` dagegen auf Anwendungsebene (R_A-Zugriff). Tabelle 4.3a zeigt, daß Konflikte durch die Reihenfolge R_A-W-R_S vermieden werden, im Einklang mit dem Ergebnis aus § 3.5.3.

Der Frage, inwieweit diese Reihenfolgesteuerung durch eine Monitorinfrastruktur automatisch vorgenommen werden kann, werden wir in § 5.3.3.2 nachgehen.

Konfliktbehandlung. Kann ein Konflikt nicht wie beschrieben vermieden werden, ist eine geeignete Konfliktbehandlung erforderlich. Ein *Konsistenzproblem* wird durch Zwischenspeicherung einer Zustandskomponente (bzw. eines Parameters) x innerhalb eines Werkzeugs A hervorgerufen. Das Werkzeug A besitzt also eine lokale Kopie von x, die ungültig wird, sobald x von einem anderen Werkzeug B verändert wird. Die Situation ist dabei analog zum Problem der Cache-Kohärenz in symmetrischen Multiprozessorsystemen [57]. Die Strategie zur Behandlung des Problems ist daher ebenfalls analog: A muß bei einer Veränderung von x eine Benachrichtigung erhalten. Diese löst entweder lediglich die Invalidierung der lokalen Kopie von x aus, so daß A den aktuellen Wert bei Bedarf wieder explizit lesen muß (Invalidierungsprotokoll bei Caches), oder sie enthält bereits den aktualisierten Wert (*Update*-Protokoll).

Überwacht das Werkzeug A Änderung von x bereits von sich aus, führt dies dazu, daß die Dauer seiner Lesezugriffe auf x Null wird, da sich A nicht darauf verläßt, daß x den gelesenen Wert für eine gewisse Zeit beibehält. Unter der Voraussetzung, daß das entsprechende Ereignis auch bei einer für Anwendung sichtbaren Modifikation von x durch ein anderes Werkzeug ausgelöst wird, treten in diesem Fall keine Transparenzprobleme auf.

Werkzeuge überwachen aus Effizienzgründen jedoch oft trotz der Zwischenspeicherung einer Zustandskomponente x deren Veränderung nicht, da das Werkzeug aufgrund impliziten Wissens über das Zielsystem sicher ist, daß x während des Lesezugriffs nicht *von der Anwendung* geändert werden kann. Wird x jedoch von einem anderen Werkzeug geändert, entsteht ein Konsistenzproblem. Um dieses zu behandeln, muß Werkzeug A (statisch oder dynamisch) so modifiziert werden, daß es bei einer Änderung von x die notwendigen Aktualisierungen vornimmt[1]. Die Realisierung unterstützender Mechanismen in einem Monitorsystem wird in § 5.3.3.1 und

[1]Da dadurch die Dauer der Lesezugriffe auf x zu Null gemacht wird, ist dies streng genommen eine Konfliktvermeidung.

§ 5.3.3.3 diskutiert.

Zum Teil verwenden Werkzeuge auch abgeschwächte Konsistenzmodelle, die eine Übereinstimmung der intern gespeicherten bzw. auf der Benutzungsoberfläche dargestellten Information mit dem aktuellen Zustand des Zielprogramms nur in bestimmten Situationen fordern. Diese werden im Modell durch die verkürzte Dauer der betroffenen Lesezugriffe berücksichtigt. Ein Beispiel ist `DETOP`, der konsistente Information nur für angehaltene Zielprozesse besitzt; die Lesezugriffe enden also jeweils mit der Weiterführung des Prozesses. Derartige Konsistenzmodelle können daher eine Benachrichtigung bei einer Zustandsänderung überflüssig machen, die durch ein anderes Werkzeug verursacht wurde.

Zur Behandlung von *Transparenzproblemen* muß erreicht werden, daß ein Werkzeug B, das eine Zustandskomponente x liest, die zuvor durch ein Werkzeug A vorgenommene Modifikation von x nicht wahrnimmt. Werkzeug A muß also genau wie für die untersuchte Anwendung auch für die anderen Werkzeuge eine angepaßte Sicht des Zielsystems aufbauen, in der die Änderung unsichtbar ist. Dazu müssen Lesezugriffe auf x abgefangen und deren Ergebnisse entsprechend der gewünschten Sicht manipuliert werden. Da Werkzeug B jedoch im allgemeinen mit Hilfe anderer Mechanismen auf x zugreift als das Zielprogramm, muß das Monitorsystem Dienste zum Abfangen derartiger Zugriffe bereitstellen. Auf die Realisierungsmöglichkeiten für solche Dienste wird in § 5.3.3.4 eingegangen.

Es soll nicht unerwähnt bleiben, daß diese Vorgehensweise ein Potential für neue Konflikte beinhaltet, falls zwei Werkzeuge die Zugriffe auf dieselbe Zustandskomponente x abfangen wollen. Dieser Fall kann allerdings nur dann eintreten, wenn vorher beide Werkzeuge x manipuliert haben. Der Konflikt beim Abfangen ist also eine direkte Folge eines bereits vorhandenen Schreib/Schreib-Konflikts. Falls dieser vermieden oder behandelt werden kann, ist dies in entsprechender Weise auch für den Abfang-Konflikt möglich.

Schreib/Schreib-Konflikte entstehen dadurch, daß ein Werkzeug B eine bereits durch ein anderes Werkzeug A veränderte Zustandskomponente überschreibt. Hier ist nur im konkreten Einzelfall entscheidbar, ob tatsächlich ein Konflikt vorliegt, d.h. ob durch den Schreibzugriff von B die korrekte Funktionsweise von A beeinträchtigt wird. Liegt ein echter Konflikt vor, bedeutet dies in der Regel, daß die betroffenen Werkzeuge nicht gemeinsam verwendbar sind. Eine allgemeine Methode zur Konfliktbehandlung existiert für diesen Fall nicht.

Eine spezielle Situation tritt jedoch bei der Überwachung von Funktionsaufrufen ein, da für deren Parameter eine Datenflußrichtung definiert ist. Werkzeuge nehmen daher Modifikationen der Parameter typischerweise so vor, daß zunächst die aktuellen Parameterinhalte gelesen und auf Basis dieser Inhalte die zu schreibenden Werte bestimmt werden. Schreib/Schreib-Konflikte lassen sich in solchen Fällen oft durch eine geeignete Zugriffsreihenfolge auflösen. Diese Reihenfolge ist dadurch gekennzeichnet, daß sie Konsistenz- und Transparenzprobleme vermeidet. Als Beispiel sie die Modifikation des Parameters `tid` der Funktion `pvm_recv` durch `Codex` und `CoCheck` genannt. Berücksichtigt man, daß `Codex` den Wert des Parameters so sehen muß, wie ihn die Anwendung übergibt (R_A-Zugriff), `CoCheck` jedoch so, wie ihn – ohne `CoCheck` – das PVM-System sieht (R_S-Zugriff), ergibt sich ein zeitlicher Ablauf der Zugriffe, wie er in Abb. 4.3 dargestellt ist. Ein Vergleich mit Tabelle 4.3a zeigt, daß bei dieser Reihenfolge keine Konsistenz- oder Transparenzprobleme auftreten[2].

Oft speichern Werkzeuge auch nach der Veränderung einer Zustandskomponente den geschriebenen Wert intern als aktuellen Wert zwischen. In dem entwickelten Modell wäre dies als

[2]Es ist sehr leicht zu sehen, daß diese Vorgehensweise allerdings nur bei genau zwei am Konflikt beteiligten Werkzeugen funktioniert.

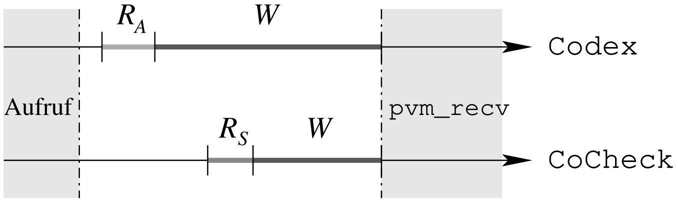


Abbildung 4.3: Lösung eines Schreib/Schreib-Konflikts zwischen `Codex` und `CoCheck`

eine Schreiboperation darzustellen, die unmittelbar von einer Leseoperation gefolgt wird. Wird die Komponente anschließend von einem anderen Werkzeug modifiziert, so kann unabhängig von einem eventuellen Schreib/Schreib-Konflikt ein Konsistenzproblem entstehen, das wie angesprochen behandelt werden kann.

Als Beispiel sei das Anhalten eines Prozesses genannt, das als Schreibzugriff auf eine Zustandskomponente `running` des Prozesses modelliert werden kann. Hält ein Werkzeug A einen Prozeß an, geht es üblicherweise davon aus, daß dieser gestoppt bleibt, bis es selbst ihn wieder fortsetzt. Wird das Fortsetzen jedoch von einem anderen Werkzeug B ausgelöst, so muß A in jedem Fall darüber benachrichtigt werden. Natürlich kann es vorkommen, daß eine korrekte Funktion von A nur dann gewährleistet ist, wenn der Prozeß gestoppt bleibt, bis er durch A selbst fortgesetzt wird. In diesem Fall läge ein nicht ohne weiteres auflösbarer Schreib/Schreib-Konflikt vor[3].

Indirekte Folgen von Schreibzugriffen. In den bisherigen Betrachtungen wurde außer Acht gelassen, daß Schreibzugriffe auf Zustandskomponenten in der Folge zu weiteren Änderungen des Zielsystemzustands bzw. Funktionsparametern führen können. Umgekehrt kann die Modifikation des Parameters einer überwachten Funktion in Folge eine Änderung des Zielsystemzustands bewirken. Diese indirekten Auswirkungen eines Schreibzugriffs können daher zu weiteren Konsistenz- oder Transparenzproblemen führen.

Ein Konsistenzproblem kann z.B. dann auftreten, wenn ein Werkzeug A den Wert der Zustandskomponente y lokal zwischengespeichert hat, ein Werkzeug B die Komponente x ändert, und dies indirekt zu einer Änderung von y führt. Ist in diesem Fall eine explizite Benachrichtigung von A zur Aktualisierung von y erforderlich? Die Antwort auf diese Frage ist davon abhängig, ob die Kausalkette von x nach y über die untersuchte Anwendung führt oder nicht. Im ersten Fall ist die Änderung von y direkt oder indirekt durch die Anwendung erfolgt. Kann y jedoch durch die Anwendung geändert werden, so muß A, um die Konsistenz seiner Darstellung zu sichern, ohnehin bereits die entsprechenden Zugriffe der Anwendung überwachen, oder aber A verwendet ein abgeschwächtes Konsistenzmodell, bei dem die Darstellung von y nur an bestimmten Stellen im Programmablauf mit dem aktuellen Wert übereinstimmt. In beiden Fällen sind keine zusätzlichen Maßnahmen zur Konsistenzerhaltung erforderlich. Führt die Kausalkette jedoch nicht über die Anwendung, so entsteht ein Konsistenzproblem, das entsprechend behandelt werden muß.

Ähnlich ist die Situation in bezug auf Transparenzprobleme. Es sei angenommen, daß ein

[3]Das Verhalten der beiden Werkzeuge könnte natürlich durch Verhaltensregeln so erweitert werden, daß während der Zeit, in der A den Prozeß als gestoppt voraussetzt, die Funktion in B deaktiviert wird, die dessen Weiterführung zur Folge hat. Allerdings könnte dadurch wiederum die Funktionsweise von B beeinträchtigt werden.

Werkzeug A die Zustandskomponente x ändert, dies indirekt zu einer Änderung von y führt, und anschließend Werkzeug B die Komponente y liest. Wenn die Kausalkette von x nach y über die Zielanwendung führt, folgt, daß diese die Änderung von x sieht. Daher müssen auch alle Werkzeuge diese Änderung beobachten können und somit auch deren Auswirkungen. Es ergibt sich also kein Transparenzproblem. Im anderen Fall muß gegebenenfalls auch die Folgeänderung an y vor B verborgen werden.

Die Sicherung der Konsistenz und Transparenz bei Anwesenheit derartiger indirekter Modifikationen des Zielsystemzustands ist nicht trivial, da die Auswirkungen von Schreibzugriffen der Werkzeuge genauestens analysiert werden müssen. Da nach der obigen Argumentation die relevanten, aus einem Schreibzugriff folgenden Änderungen aber nicht von der untersuchten Anwendung abhängig sein können, ist eine allgemeingültige, d.h. von der Anwendung unabhängige Auflösung entstehender Konsistenz- und Transparenzprobleme aber zumindest möglich.

Ereignisse. Das beschriebene Modell berücksichtigt die Überwachung von Ereignissen, d.h. Änderungen von Zustandskomponenten, die asynchron an ein Werkzeug gemeldet werden, nicht explizit. Logische Konflikte, die im Zusammenhang mit Ereignissen stehen, können mit den bisher eingeführten Konzepten untersucht werden, wenn die Ereignisüberwachung durch einen *Polling*-Vorgang modelliert wird. Während einer aktivierten Ereignisüberwachung führt das Werkzeug somit direkt aneinandergereihte Lesezugriffe aus, deren Dauer gegen Null geht. Die Zugriffe können dabei je nach Semantik des Werkzeugs vom Typ R_A oder R_S sein.

Bei dieser Modellierung können beim Monitoring von Ereignissen keine Konsistenzprobleme auftreten. Dies ist auch in der Realität so, vorausgesetzt, die Realisierung der Ereignisüberwachung erkennt immer auch Änderungen der betroffenen Zustandskomponente durch andere Werkzeuge. Führt ein Werkzeug einen W_S-Zugriff auf eine Zustandskomponente durch, die von einem anderen Werkzeug auf Anwendungsebene überwacht wird, ergeben sich jedoch Transparenzprobleme. Der in unserem Beispielszenario durch die Verwendung von Signalen verursachte Konflikt zwischen `CoCheck` und `DETOP` (siehe § 3.5.5) ergibt sich durch das Modell wie folgt: Die Überwachung durch `DETOP` entspricht einer Folge von R_A-Zugriffen auf die Menge der anhängigen Signale eines Prozesses, während das Senden des Signals durch `CoCheck` einem W_S-Zugriff auf ein Element dieser Menge entspricht. Der Zugriff ist wegen der von `CoCheck` eingesetzten Signalbehandlungsfunktion für die Anwendung unsichtbar. Tabelle 4.1 zeigt nun unmittelbar die Anwesenheit des Transparenzproblems an.

Eine Instrumentierung durch ein Werkzeug, die ein Ereignis auslöst, z.B. das Einfügen einer *Trap*-Instruktion in den Code eines Zielprozesses, entspricht einer für die Anwendung unsichtbaren Modifikation, also einem W_S-Zugriff. Da in diesem Fall auch das Ereignis selbst, z.B. das durch die *Trap*-Instruktion ausgelöste Signal, für die Anwendung unsichtbar sein muß, ist die entsprechende Zustandsänderung ebenfalls durch einen W_S-Zugriff zu modellieren. Dieser ist wie im vorherigen Abschnitt diskutiert eine indirekte Folge der ersten Zugriffs und bei der Konfliktanalyse entsprechend zu berücksichtigen.

Mehrebenen-Monitoring. Einige Monitorinfrastrukturen, wie z.B. MiMo [195, 196] unterstützen die Beobachtung von Programmen auf unterschiedlichen Abstraktionsebenen. Bei MiMo sind dies die Ebenen der Anwendung, der Schnittstellen, der verteilten Objekte, deren Implementierung, der Laufzeit-Umgebung und der Hardware. Der Unterschied zu der bereits behandelten Beobachtung auf mehreren Software-Ebenen besteht darin, daß nun jede Ebene ein eigenes Zustandsmodell des Systems, also Objekte und Zustandskomponenten, definiert, wobei das Zustandsmodell auf höheren Ebenen eine Abstraktion der Modelle tieferer Ebenen darstellt. Bei der Analyse von Konflikten ergibt sich daher die Schwierigkeit, daß der Zusammenhang zwischen diesen verschiedenen Modellen exakt bekannt sein muß. Leider wird dieser in der Regel von den Monitorsystemen nicht formal definiert. MiMo beinhaltet zumindest eine formale Definition der Zuordnung zusammengehöriger Objekte auf verschiedenen Abstraktionsebenen. Ist die Relation der Abstraktionsebenen zueinander jedoch bekannt, können die Auswirkungen einer Modifikation auf einer Ebene auf eine andere Ebene umgerechnet und damit mögliche Konflikte mit Zugriffen auf dieser Ebene ermittelt werden.

4.4 Richtlinien für den Werkzeugentwurf

Die in der Fallstudie aus § 3 gewonnenen Erfahrungen haben gezeigt, daß eine Interoperabilität von Werkzeugen erheblich einfacher erreicht werden kann, wenn bereits beim Entwurf der Werkzeuge gewisse Richtlinien beachtet werden.

So wirkt sich eine klar strukturierte Implementierung der Werkzeuge positiv aus, da in diesem Fall für alle wichtigen Werkzeugfunktionen bereits definierte Schnittstellen vorhanden sind. Mit der in § 2.3.4.2 skizzierten Technik können diese Funktionen dann auch durch externe Komponenten, etwa einen Mediator, aufgerufen werden. Zwar kann durch dieselbe Technik auch beliebiger Code zur Realisierung der nötigen Funktionen in das Werkzeug eingeschleust werden, die Nutzung vorhandener Schnittstellen ist jedoch deutlich einfacher und sicherer. Die Forderung nach dem Vorhandensein von Schnittstellen bedeutet dabei noch nicht, daß diese explizit vom Werkzeug exportiert werden müssen, etwa in Form einer `CORBA`-Schnittstelle. Dies wäre oft auch nicht sinnvoll, da die Schnittstelle aller potentiell interessanten Werkzeugfunktionen i.a. sehr groß ist und in einer konkreten Umgebung nur ein kleiner Bruchteil davon wirklich benötigt wird.

Beim Entwurf eines Werkzeugs sollte prinzipiell die gleichzeitige Anwendung anderer Werkzeuge berücksichtigt werden, soweit dies möglich ist, ohne diese im einzelnen zu kennen und ohne Abhängigkeiten zu erzeugen. Insbesondere Konsistenzprobleme können so von vorneherein vermieden werden. Die oft typische Annahme „*der Zustand des Zielsystems wird nur durch das Zielsystems selbst sowie das zu entwerfende Werkzeug geändert*" muß dazu aufgegeben werden. Ein Beispiel dafür ist die schon erwähnte Annahme, daß ein vom Werkzeug angehaltener Prozeß solange gestoppt bleibt, bis das Werkzeug selbst ihn wieder fortsetzt (da er ja nicht durch das Zielprogramm selbst fortgesetzt werden kann). Das Werkzeug sollte die vom Monitorsystem angebotenen Ereignisdienste nutzen, um die relevante Zustandsänderung zu überwachen, selbst wenn das Ereignis bei alleiniger Verwendung des Werkzeugs nie auftreten kann. Stellt das Monitorsystem Dienste zur Vermeidung oder Behandlung anderer Konflikte zur Verfügung, sollten diese ebenso genutzt werden. Zwar kann auch das Rahmenwerkzeug noch nachträglich die erforderlichen Dienste definieren und ggf. benötigten, zusätzlichen Code durch dynamische Instrumentierung in das Werkzeug einbauen, dies ist jedoch ein komplexer Prozeß, der zudem für alle Umgebungen repliziert werden müßte, an denen das betrachtete Werkzeug Teil hat. Im Idealfall sollte ein Werkzeug von sich aus alle Mechanismen zur Konfliktvermeidung und -behandlung nutzen, die unabhängig von den konkreten anderen Werkzeugen nutzbar sind. Das Rahmenwerkzeug realisiert dann nur noch die Teile, die Wissen über mehr als ein an der Umgebung beteiligtes Werkzeug benötigen.

Als weitere bedeutsame Entwurfsrichtlinie ist die Vermeidung impliziter Annahmen über das Zielsystem zu nennen. Zwei Beispiele für derartige Annahmen wurden bereits in § 3.5.6 diskutiert. Eine weitere, oft anzutreffende Annahme ist die, daß beim Ankoppeln des Werkzeugs an einen Zielprozeß alle Einstellungen der Prozeßkontroll-Schnittstelle des Betriebssystems noch den Standardwerten entsprechen, die beim Prozeßstart vom Betriebssystem gesetzt wurden [257]. Zusätzlich sollte ein defensiv programmiertes Werkzeug beim Beenden, soweit dies möglich und sinnvoll ist, alle vorgenommenen Änderungen am Zielsystem sowie an den Einstellungen der verwendeten Monitorschnittstellen rückgängig machen. Dies ermöglicht, daß Werkzeuge ohne weitere Koordinierungsmaßnahmen zumindest nacheinander auf dasselbe Zielsystem angewandt werden können [257].

Kapitel 5

Infrastrukturen für interoperable Online-Werkzeuge

Nachdem im letzten Abschnitt die prinzipielle Vorgehensweise bei der Kombination von Online-Werkzeugen dargelegt wurde, sollen in § 5.1 zunächst die abstrakten Anforderungen zusammengefaßt werden, die an eine Monitoring-Middleware gestellt werden müssen, um diese Vorgehensweise zu ermöglichen. In § 5.2 werden wir anschließend untersuchen, inwieweit existierende Infrastrukturen diese Anforderungen bereits erfüllen. Da sich herausstellen wird, daß keiner der bisherigen Ansätze ausreichend ist, werden schließlich in § 5.3 Konzepte für eine Monitoring-Middleware erarbeitet, die die einfache Realisierung von Umgebungen interoperabler Werkzeuge ermöglicht.

5.1 Anforderungen an eine Werkzeug-Middleware

Aus der Diskussion in § 3 und § 4 lassen sich vier grundlegende Anforderungen ableiten, die von einer Werkzeug-Middleware erfüllt werden müssen, um Umgebungen interoperabler Werkzeuge unterstützen zu können [263, 258, 264].

5.1.1 Universelle, gemeinsam nutzbare Monitor-Schnittstelle

Obwohl sich in § 4.3.1 gezeigt hat, daß in geeigneten Fällen eine gleichzeitige Nutzung von Werkzeugen auch ohne Verwendung eines einheitlichen Monitorsystems möglich ist, können im allgemeinen strukturelle Konflikte zwischen Werkzeugen doch nur durch eine gemeinsam genutzte Monitor-Schnittstelle beseitigt werden. Eine Middleware für interoperable Laufzeit-Werkzeuge muß also zunächst eine solche Schnittstelle bereitstellen. Sie muß dabei universell, d.h. für sehr unterschiedliche Werkzeuge geeignet sein. Die langjährige Erfahrung im Bereich des Monitorings paralleler Systeme im Sonderforschungsbereich 342 der TU München hat dabei gezeigt, daß derartige Schnittstellen auf dem Ereignis/Aktions-Konzept basieren sollten, um die erforderliche Flexibilität und Programmierbarkeit zu erreichen [30].

Zusätzlich zu der reinen Monitoring-Funktionalität sind zur Unterstützung interoperabler Werkzeuge auch Basismechanismen zur Koordination von Monitoranfragen erforderlich. Bisher wurde implizit davon ausgegangen, daß Anfragen vom Monitorsystem unteilbar abgearbeitet werden, d.h. die Ausführungen verschiedener Anfragen sich nicht zeitlich überlappen[1]. Da

[1] Dies ist nicht zu verwechseln mit der in § 4.3.2 betrachteten Überlappung von Schreib- und Lesezugriffen.

Monitorsysteme verteilte Anwendungen darstellen, wäre eine unteilbare Abarbeitung aller Anfragen jedoch sehr ineffizient. Zudem ist diese auch nicht immer erforderlich, z.B. wenn zwei Anfragen disjunkte Teile des Zielsystems betreffen. Die Serialisierbarkeit von Anfragen auf Einzelobjekten muß aber in jedem Fall gewährleistet bleiben. Zusätzlich sollten Sperrmechanismen vorgesehen werden, mit denen bei Bedarf eine Unteilbarkeit einzelner Anfragen erzielt werden kann, wobei Verklemmungsfreiheit sichergestellt sein muß.

Es ist praktisch undenkbar, daß eine Monitorschnittstelle alle Funktionen vorsieht, die jemals von Laufzeit-Werkzeugen benötigt werden. Eine fundamentale Anforderung an die Schnittstelle ist daher deren Erweiterbarkeit um neue Dienste. Der Erweiterbarkeit kommt speziell bei Schnittstellen von Monitorsystemen eine besondere Bedeutung zu, da diese sehr effizient realisiert werden müssen, um die Rückwirkung auf das untersuchte System zu minimieren. Oft wird es daher erforderlich, auch solche Funktionen in das Monitorsystem zu integrieren, die sich ausschließlich auf Dienste der vorhandenen Schnittstelle abstützen, weil dadurch z.T. ein erheblicher Kommunikationsaufwand eingespart werden kann.

5.1.2 Unterstützung direkter Interaktionen

Um die Verhaltensintegration der Werkzeuge einer Umgebung zu ermöglichen, muß eine Werkzeug-Middleware direkte Interaktionen zwischen diesen Werkzeugen ermöglichen. Zusätzlich muß ein Aufruf von Diensten der Werkzeuge durch koordinierende Komponenten der Werkzeugumgebung ermöglicht werden, um erweiterte Funktionen dieser Umgebung realisieren zu können. Dies kann durch eine eigene Middleware-Komponente, aber auch direkt durch das Monitorsystem erfolgen.

Die Interaktionen sollten dabei auf der Basis von Ereignissen und Aktionen erfolgen, deren Verbindung durch Ereignis/Aktions-Regeln definiert wird. Um die Werkzeuge ohne Änderung in verschiedenen Umgebungen einsetzen zu können, dürfen diese Regeln nicht in die betroffenen Werkzeuge eincodiert werden. Sie sind vielmehr durch Mediatoren zu realisieren. Mediatoren erlauben es zudem, möglicherweise inkompatible Schnittstellen von Ereignisse und Aktionen aneinander anzupassen.

5.1.3 Unterstützung indirekter Interaktionen

Indirekte Interaktionen zwischen Werkzeugen sind erforderlich, um die Konsistenz der Information sicherzustellen, die von verschiedenen Werkzeugen an den Benutzer weitergegeben wird. Die Middleware muß dazu insbesondere Dienste bereitstellen, mit deren Hilfe ein Werkzeug eine Benachrichtigung erhalten kann, wenn eine Zustandskomponente des Zielsystems durch ein anderes Werkzeug verändert wurde. Derartige Dienste werden auch benötigt, um logische Konflikte zu behandeln (siehe unten).

5.1.4 Mechanismen zur Behandlung logischer Konflikte

Eine Infrastruktur für interoperable Werkzeuge muß effiziente Mechanismen anbieten, mit deren Hilfe logische Konflikte zwischen Werkzeugen gelöst werden können. In § 4.3.2 wurden drei derartige Mechanismen als notwendig erkannt:

1. Ein Verfahren, um die Reihenfolge von Aktionen festzulegen, die unterschiedliche Werkzeuge für dasselbe Ereignis (Start oder Rückkehr einer überwachten Bibliotheksfunktion) definiert haben.

2. Die Möglichkeit, alle Modifikationen des Zielsystemzustands durch andere Werkzeuge als Ereignis beobachten zu können. Diese Anforderung ist identisch mit der oben genannten, verwandten Forderung nach Unterstützung indirekter Werkzeuginteraktionen.

3. Eine Schnittstelle, die es Werkzeugen ermöglicht, Zugriffe anderer Werkzeuge auf Zustandskomponenten des Zielsystems abzufangen und bei Bedarf zu modifizieren, um Transparenzprobleme aufzulösen.

5.1.5 Nebenbedingungen

Als Nebenbedingung muß sichergestellt sein, daß die genannten Mechanismen von Werkzeugen benutzbar sind, ohne daß sie dazu Wissen über die konkreten anderen Werkzeuge der Umgebung benötigen. Insbesondere müssen die Werkzeuge auch außerhalb der Umgebung lauffähig bleiben. Nicht zuletzt sollte der Einsatz eines Werkzeugs innerhalb einer Werkzeugumgebung mit möglichst wenigen – im Idealfall sogar gänzlich ohne – Quellcode-Modifikationen ermöglicht werden können. Nur so können in einfacher Weise Umgebungen aus existierenden Werkzeugen zusammengestellt werden.

Unvermeidbar benötigtes, globales Wissen über die Werkzeugumgebung sollte der Infrastruktur zur Laufzeit bekanntgegeben werden können. Sie kann von einem Rahmenwerkzeug definiert werden, das die Zusammensetzung der Umgebung kennt.

5.2 Umsetzung der Anforderungen in existierender Middleware

Nachdem nun die genauen Anforderungen interoperabler Werkzeuge ermittelt wurden, kann nun eine abschließende Bewertung der in § 2.2.3.2 und § 2.3.4.1 vorgestellten Infrastrukturen vorgenommen werden. Die Diskussion von OMIS/OCM bezieht sich dabei auf den Stand vor der Durchführung der vorliegenden Arbeit.

5.2.1 Universelle, gemeinsam nutzbare Monitor-Schnittstelle

Von den in § 2.3.4.1 beschriebenen Monitorsystemen besitzen DPCL, DAMS, FIRST und OCM eine universelle, von mehreren Werkzeugen nutzbare Schnittstelle. Alle vier Systeme basieren auf dem Ereignis/Aktions-Konzept, das jedoch sehr unterschiedlich umgesetzt ist. Der OCM erlaubt dabei als einziges die explizite Zuordnung vorhandener Aktionen zu Ereignissen. In DPCL und FIRST ist diese Zuordnung durch die Programmierung des Instrumentierungscodes sowie seine Einfügestelle im Code des Zielprogramms festgelegt. DAMS benachrichtigt ein Werkzeug grundsätzlich bei jedem Auftreten eines überwachten Ereignisses. Eine entsprechende Rückruffunktion im Werkzeug muß dann die Aktionsausführung durch Monitoranfragen veranlassen.

Eine Koordination von Anfragen verschiedener Werkzeuge ist in DPCL bisher nicht vorgesehen. Auch für DAMS und FIRST ist kein derartiger Mechanismus in der Literatur dokumentiert. OMIS definiert immerhin einige grundlegende Koordinationsmechanismen. Zunächst werden Aktionen auf jedem Einzelobjekt unteilbar ausgeführt. Ausgenommen sind dabei bestimmte Fehlersituationen (siehe [150], S. 56-58). Zudem kann das gesamte Monitorsystem von einem Werkzeug für die Dauer einer Anfrage gesperrt werden, d.h. die Bearbeitung dieser

Anfrage wird nicht mit der Bearbeitung anderer Anfragen überlappt. Der OCM realisiert zusätzlich zu dieser Spezifikation auch ein *Atomic-Multicast*-Protokoll, das sicherstellt, daß globale Anfragen von allen lokalen Monitoren in derselben Reihenfolge bearbeitet werden (siehe auch § 5.3.1.2, Seite 137). Bei der Implementierung der Fallstudie aus § 3 zeigte sich, daß diese einfachen Mechanismen durchaus ausreichend sind.

Zumindest drei der genannten Monitorsysteme sind erweiterbar. DPCL erlaubt Werkzeugen die völlig freie Programmierung des Instrumentierungscodes. Dadurch können beliebige Ereignisfilter und Aktionen realisiert werden. Ob eine derartige freie Programmierung der Instrumentierung über die Monitor-Schnittstelle auch bei FIRST möglich ist, ist der Literatur nicht zu entnehmen. DAMS gestattet ein Hinzufügen von Modulen zum Monitorsystem, die Dienste an dessen Schnittstelle anbieten. Die Realisierung dieser Dienste erfolgt durch zum Modul gehörige Treiber, die jeweils einen Prozeß des Zielsystems kontrollieren. Da DAMS für diese Treiber keine einheitliche Schnittstelle zum Zielsystem vorsieht, kann im allgemeinen aufgrund von strukturellen Konflikten nur ein Treiber pro Zielprozeß eingesetzt werden, was die Erweiterbarkeit deutlich einschränkt. OMIS erlaubt ebenfalls die Erweiterung der Monitorschnittstelle um Dienste, die in Form von Erweiterungsbibliotheken realisiert werden. Dabei wird zwischen zwei Arten von Erweiterungen unterschieden: Werkzeug-Erweiterungen nutzen ausschließlich bereits vorhandene Dienste, sind also portabel und können keine strukturellen Konflikte mit anderen Erweiterungen auslösen. Monitor-Erweiterungen können dagegen auch direkt auf das Zielsystem zugreifen. Um strukturelle Konflikte möglichst zu vermeiden, realisiert der OCM für diese Erweiterungen eine eigene Zielsystem-Schnittstelle (siehe Abb. 3.4).

5.2.2 Unterstützung direkter Interaktionen

Mit Ausnahme von FIRST unterstützt keine der existierenden Monitorschnittstellen direkte Interaktionen zwischen den angeschlossenen Werkzeugen. FIRST bietet eine Unterstützung lediglich dadurch, daß es auf der Middleware CORBA aufsetzt. Andererseits verfolgen die in § 2.2.3.2 besprochenen Infrastrukturen zur Verhaltensintegration genau das Ziel einer flexiblen Komponenten-Interaktion. CORBA, DCOM und PCTE besitzen dabei den Nachteil, daß die Interaktionen durch direkten Nachrichtenaustausch erfolgt, wobei der Sender den Empfänger explizit kennen und spezifizieren muß. ToolTalk erlaubt die indirekte Kommunikation über einen Broadcast-Nachrichtenserver. Die Werkzeuge müssen aber nach wie vor beim Eintreten eines (möglicherweise) interessanten Ereignisses eine Nachricht versenden – diesmal jedoch an den Broadcast-Nachrichtenserver – und ihr Interesse an Ereignissen durch Definition einer Rückruffunktion bekunden. Dadurch werden die Werkzeuge vom Vorhandensein der ToolTalk-Umgebung abhängig. Problematischer ist, daß die Werkzeuge einheitliche Identifikatoren für alle auftretenden Ereignisse verwenden müssen und auch die Struktur der zugehörigen Ereignisnachrichten von allen Werkzeugen respektiert werden muß. Unter dieser Voraussetzung ist keine unabhängige Entwicklung der Werkzeuge möglich.

Manifold, ToolBus, CHAIMS und TIL vermeiden das zuletzt genannte Problem, da die Zuordnung zwischen den von den einzelnen Werkzeugen unabhängig definierten Ereignissen und Aktionen über eine eigene Koordinations- bzw. Konfigurationssprache außerhalb der Werkzeuge festgelegt wird. Dadurch können nicht nur Namensanpassungen, sondern bei TIL auch beliebige Anpassungen der auszutauschenden Parameter vorgenommen werden. Von den Werkzeugen wird jedoch immer noch gefordert, daß alle eventuell von anderen benötigten Ereignisse und Aktionen explizit über eine von der jeweiligen Infrastruktur abhängige Schnittstelle exportiert werden.

Die Verwendung von DPCL als Monitorsystem könnte es erlauben, die notwendigen Schnittstellen auch dynamisch in die Werkzeuge einzufügen. Eine Kombination von DPCL mit z.B. TIL könnte daher vorteilhaft sein. Die analoge Kombination von OCM und TIL wäre hingegen nicht sinnvoll, da der OCM die Grundfunktionalität zur Verknüpfung von Ereignissen und Aktionen – sowie über die Erweiterungs-Schnittstelle auch für Parameteranpassungen – bereits selbst bereitstellt.

5.2.3 Unterstützung indirekter Interaktionen

Da bei indirekten Interaktionen Ereignisse im Zielsystem eine entscheidende Rolle spielen, müssen sie in jedem Fall direkt durch das Monitorsystem unterstützt werden. DPCL bietet keinerlei derartige Unterstützung an.

Das PDBG-Modul des Monitorsystems DAMS definiert ein Zustandsübergangssystem für den Ausführungsstatus von Prozessen. Änderungen dieser Zustandskomponente können von den Werkzeugen überwacht werden, andere jedoch nicht. Darüber hinaus können Werkzeuge Meldungen über den Abschluß von Aktionen anderer Werkzeuge erhalten. Dadurch sind prinzipiell auch andere Konsistenzprobleme lösbar, da ein Werkzeug die Änderungs-Aktionen anderer Werkzeug sehen kann. Problematisch ist allerdings, daß die genannten Mechanismen nicht im Basissystem von DAMS enthalten sind, sondern von den Dienstmodulen (z.B. PDBG) realisiert werden. Ein Werkzeug, das ein Dienstmodul A benutzt, kann dadurch die Aktionen eines anderen Werkzeugs, das Modul B verwendet, nicht unmittelbar beobachten. Dazu müßte das Werkzeug den entsprechenden Dienst des Moduls B verwenden. Dies widerspricht aber der Idee der Modularisierung und der einfachen Erweiterbarkeit des Monitorsystems.

FIRST und OMIS definieren nur für einige ausgewählte Zustandsänderungen zugehörige Ereignisdienste, insbesondere für Anhalten und Weiterführen von Prozessen. Damit lassen sich zwar die wichtigsten, aber bei weitem nicht alle Konsistenzprobleme lösen.

5.2.4 Mechanismen zur Behandlung logischer Konflikte

Keines der untersuchten Monitorsysteme bietet Mechanismen an, mit deren Hilfe die Reihenfolge der Abarbeitung unterschiedlicher Aktionen desselben Ereignisses festgelegt werden kann. Die Abarbeitungsreihenfolge wird dabei von den Monitorsystemen nicht spezifiziert, kann also prinzipiell beliebig sein. Lediglich `dyninst`, das als Implementierungsbasis von DPCL und FIRST dient, erlaubt beim Einfügen einer Instrumentierung in einen Prozeß eine Festlegung, ob diese vor oder nach bereits vorhandenen Instrumentierungen ausgeführt werden soll. Zur Vermeidung logischer Konflikte ist diese Maßnahme jedoch nicht ausreichend, da z.B. zwei Werkzeuge eine Ausführung nach allen bereits vorhandenen Instrumentierungen fordern können. In diesem Fall ist die letztendliche Ausführungsreihenfolge von der Reihenfolge des Einfügens abhängig.

Die vorhandene Unterstützung zur Meldung von Zustandsmodifikationen durch andere Werkzeuge wurde bereits im letzten Unterabschnitt diskutiert.

Ein Abfangen und Modifizieren von Zugriffen auf den Zielsystemzustand, wie es zur Behandlung von Transparenzproblemen notwendig ist, ist bei keiner existierenden Monitorinfrastruktur möglich.

5.2.5 Fazit

Als Fazit muß daher festgehalten werden, daß keine derzeit existierende Monitoring-Middleware die gestellten Anforderungen erfüllt. Obwohl für direkte Interaktionen die Kombination eines Monitorsystems mit einer Infrastruktur zur Verhaltensintegration möglich erscheint, trägt diese nicht zur Erfüllung der anderen Anforderungen bei. Es ist daher angebracht, ein neues Konzept für eine Middleware zu erstellen, die in durchgängiger Weise sowohl das Monitoring als auch die Interoperabilität von Laufzeit-Werkzeugen unterstützt. Nur so können Werkzeuge, die auf dieser Middleware aufsetzen, mit geringem Aufwand zu interoperablen Umgebungen kombiniert werden.

5.3 Entwurf einer Middleware für interoperable Laufzeit-Werkzeuge

Als Basis für den nachfolgend skizzierten Entwurf einer Middleware für interoperable Werkzeuge dient die Monitorschnittstelle OMIS sowie deren Implementierung OCM, da diese aufgrund ihrer Struktur eine leichte Integration der benötigten Konzepte erlauben. Dabei wird zunächst nur die prinzipielle Umsetzbarkeit der aufgestellten Anforderungen untersucht; implementierungstechnische Details werden in § 6 beschrieben.

5.3.1 Monitorschnittstelle

Die von der Middleware bereitzustellende Monitorschnittstelle kann nahezu unverändert von OMIS übernommen werden. Die Grundstruktur dieser Schnittstelle hat sich über Jahre hinweg entwickelt und im praktischen Einsatz mit unterschiedlichen Werkzeugen bewährt. Dies betrifft insbesondere die verwendeten Konzepte des Ereignis/Aktions-Paradigmas, der Objektbasierung sowie der Ortstransparenz [30].

Das Ereignis/Aktions-Paradigma gewährleistet in Verbindung mit der Objektbasierung ein hohes Maß an Flexibilität der Schnittstelle sowie die Wiederverwendbarkeit der angebotenen Dienste. Es bildet daneben eine natürliche Basis für die Realisierung der Mediator-Konzepts zur Unterstützung direkter und indirekter Interaktionen. Auch die in § 4.3.2.2, Seite 123 skizzierte Behandlung logischer Konflikte läßt sich in natürlicher Weise auf das Paradigma abbilden, da Benachrichtigungen über die Änderung von Zustandskomponenten genau Ereignisse darstellen und das Abfangen von Zugriffen durch ein Ereignis mit angeschlossener modifizierender Aktion realisiert werden kann.

Die Ortstransparenz der Schnittstelle vereinfacht nicht nur ihre Nutzung durch die Werkzeuge erheblich, sondern ist auch eine notwendige Voraussetzung für weitergehende Konzepte zur Erhöhung der Skalierbarkeit, etwa die Anwendung von Diensten auf Mengen von Objekten. Sie erleichtert aber auch die Realisierung des Mediator-Konzepts erheblich, da auf die physische Verteilung der beteiligten Komponenten keine Rücksicht genommen werden muß.

Die OMIS-Schnittstelle bietet somit eine gute Grundlage für die Umsetzung aller von der Middleware zu erfüllenden Anforderungen. Einige Details der Schnittstellenspezifikation, insbesondere der Erweiterungsmechanismus und die Semantik von Monitoranfragen erfordern jedoch noch eine präzisere Betrachtung.

5.3.1.1 Erweiterbarkeit

Die Erweiterbarkeit der Monitorschnittstelle ist eine notwendige Voraussetzung, um unterschiedliche Werkzeuge unterstützen zu können. Die Vorverarbeitung von erfaßten Daten sollte aus Effizienzgründen möglichst nahe bei der Quelle erfolgen, da sie praktisch immer eine Datenreduktion bewirkt, etwa beim Zählen von Ereignissen oder beim Filtern von Information. Wegen der in § 3.2.1.2 beschrieben Struktur des Monitorsystems bedeutet dies, daß Erweiterungen nicht nur in Monitorprozessen sondern auch in den Monitorteilen in den Anwendungsprozessen notwendig werden. Da Werkzeuge möglicherweise spezielle Datenvorverarbeitungen benötigten, sollten sie eigene Erweiterungen in das Monitorsystem einbinden können, ohne daß andere Werkzeuge dies in irgendeiner Weise berücksichtigen müssen. Im Interesse einer einfachen Benutzbarkeit der Werkzeuge sollte dazu möglichst kein Neuübersetzen oder -binden des Monitorsystems erforderlich sein, so daß bei der Installation eines neuen Werkzeugs die vorhandene Installation des Monitors nicht angepaßt werden muß. Idealerweise sollte das Monitorsystem daher eine *Plug-in*-Architektur besitzen, die das dynamische Einbinden von Erweiterungsmodulen zur Laufzeit erlaubt. Erfolgt dies erst bei Bedarf, bleibt das Monitorsystem dabei zudem immer minimal, was die Rückwirkung auf das untersuchte System vermindert.

Realisierbar ist eine solche Architektur durch Nutzung dynamischer Bindetechniken: Ein Erweiterungsmodul besteht dabei aus einem Satz von Bibliotheken, die zu Monitor- bzw. Anwendungsprozessen dynamisch gebunden werden. Die Bibliotheken enthalten neben der Realisierung der Dienste auch eine Beschreibung der Dienst-Schnittstellen, insbesondere deren Namen und Parameter, sowie notwendige Information über deren Realisierung, etwa ob ein Dienst im Monitor- oder Anwendungsprozeß ausgeführt werden kann. Beim Einbinden liest das Monitorsystem diese Beschreibung und aktualisiert seine interne Datenbasis, so daß die neuen Dienste an der Monitorschnittstelle verfügbar werden. Diese Form der dynamischen Erweiterbarkeit erlaubt insbesondere auch die Realisierung von Interaktionen zwischen Werkzeugen. Dazu werden dynamisch Dienste zum Monitorsystem hinzugefügt, die Ereignisse in einem Werkzeug melden bzw. Aktionen in diesem Werkzeug realisieren (siehe § 5.3.2).

Wir sind nun allerdings in einer Situation, in der verschiedene Werkzeuge unabhängig voneinander Erweiterungen des Monitorsystems vornehmen können, was sofort zur Frage nach der Interoperabilität dieser Erweiterungen führt. Es ist leicht zu sehen, daß exakt dieselben Probleme wie sie zwischen Werkzeugen auftreten, auch zwischen zwei Erweiterungsmodulen möglich sind.

Strukturelle Konflikte können hier in erster Linie durch Namenskonflikte von Funktions- oder Variablenbezeichnern sowie durch die Verwendung gemeinsamer Betriebsmittel entstehen. Namenkonflikte werden dabei durch die dynamische Bindetechnik weitestgehend verhindert, da nur sehr wenige, ausgewählte Namen der Module exportiert werden. Die verbleibenden Konflikte müssen durch eine Namenskonvention gelöst werden, die im Fall des OCM in der Verwendung eines eindeutigen Präfixes besteht. Letztendlich bedeutet dies aber, daß jedes Erweiterungsmodul einen eindeutigen Namen besitzen muß, der als Präfix verwendet wird und vor der Realisierung des Moduls von einer zentralen Koordinationsstelle vergeben werden muß. Strukturelle Konflikte durch gemeinsame Betriebsmittel können nur bei Monitor-Erweiterungen auftreten, da Werkzeug-Erweiterungen ausschließlich auf der OMIS-Schnittstelle aufsetzen, die diese Konflikte bereits löst. Zur Unterstützung von Monitor-Erweiterungen definiert der OCM eine interne Zielsystem-Schnittstelle, die strukturelle Konflikte beim Zugriff auf das Zielsystem vermeidet. Da eine derartige Schnittstelle aber niemals vollständig sein kann, ist es unvermeidbar, daß manche Erweiterungsmodule auch direkt auf das Zielsystem zugreifen müssen.

Derartige Erweiterungen können nicht mehr unabhängig von anderen Erweiterungen dieser Art vorgenommen werden. Für den praktischen Einsatz des OCM stellt dies aber kein ernsthaftes Hindernis dar, da solche Erweiterungen nur erforderlich werden, um zusätzliche Zielsystem-Komponenten zu überwachen. Ihre Erstellung ist daher eher mit der zentral durchgeführten Zielsystem- und Monitorentwicklung als mit der verteilt erfolgenden Realisierung von Werkzeugen verbunden.

Logische Konflikte zwischen Erweiterungsmodulen können mit denselben Techniken behandelt werden, die in § 5.1 skizziert wurden. Da diese ebenfalls durch die OMIS-Schnittstelle bzw. eine interne Schnittstelle des OCM zur Verfügung gestellt werden, gilt die obige, für strukturelle Konflikte geführte Diskussion sinngemäß.

Neben neuen Diensten können Erweiterungsmodule auch neue Objekte definieren. Diese sind, wie in § 3.2.1.1 erläutert, in Hierarchien strukturiert, um Konvertierungen (Lokalisierung und Expansion) zwischen Objekttoken zu ermöglichen. Um die Unabhängigkeit der Erweiterungen zu sichern, darf die Einführung einer neuen Objektklasse keine Modifikation der existierenden Objekte notwendig machen. OMIS erlaubt daher die Einführung neuer Objektklassen nur, wenn diese zum Zeitpunkt ihres Einbindens Blätter im Hierarchiegraphen darstellen (siehe Abb. 3.5). In diesem Fall ist es möglich, nicht nur die Lokalisierung von Objekttoken der neuen Klasse O in ein Token der Klasse O', sondern auch die Expansion von Token einer Klasse O'' in solche der Klasse O mit Hilfe rein lokaler Methoden von O zu realisieren. Diese Vorgehensweise wird auch dadurch gerechtfertigt, daß neue Objektklassen i.a. „Verfeinerungen“ existierender Klassen darstellen, bei ihrer Einführung also Blattpositionen im Hierarchiegraphen innehaben. Ein Einfügen neuer Klassen in der Mitte der Hierarchie ist nicht in unabhängiger Weise möglich, da die Implementierung der Tokenkonvertierung für alle Nachfahren in der Hierarchie angepaßt werden müßte.

Es zeigt sich also, daß eine unabhängige, konfliktfreie Erweiterung des Monitorsystems nur eingeschränkt möglich ist. Abstrakt betrachtet haben wir eine induktive Vorgehensweise verwendet: Die Interoperation unabhängig entwickelter Werkzeuge wird durch die Monitorschnittstelle ermöglicht. Diese wird durch interoperable, unabhängige Erweiterungen definiert, die dazu induktiv auf die vorhandene Monitorschnittstelle zurückgreifen. Dieses Vorgehen benötigt jedoch einen Induktionsanfang, d.h. letztendlich gibt es eine Schicht, die nicht mehr in dieser modularen Weise aufgebaut werden kann, sondern monolithisch realisiert werden muß. Beim OCM besteht diese aus dem Basis-System, das z.B. die Behandlung logischer Konflikte realisiert, sowie allen Monitor-Erweiterungen, die direkt auf das Zielsystem zugreifen.

Streng genommen wurde das Problem also nicht wirklich gelöst, sondern nur auf immer tiefere Schichten verschoben. Dies muß jedoch auch so sein, da Interoperation und Unabhängigkeit zwei prinzipiell unvereinbare Ziele darstellen. Interoperation impliziert immer eine gewisse Koordination. Unter der Voraussetzung unabhängig entwickelter Komponenten, die kein Wissen übereinander besitzen, kann diese Koordination letztendlich aber nur durch eine beiden Komponenten in irgendeiner Weise gemeinsame Schicht durchgeführt werden.

Eine derartige Verschiebung von komplexen Problemen auf tiefere Schichten tritt in der Informatik häufiger auf, z.B. auch beim Problem des wechselseitigen Ausschlusses [9]. Abstrakte, komfortable Konstrukte, wie Semaphore, basieren in ihrer Implementierung auf unteilbaren *Read-Modify-Write*-Instruktionen der Maschine, diese letztendlich auf der Arbitrierung des Speicherbusses. Erst auf dieser Ebene wird das Problem also tatsächlich gelöst, wobei die Lösung meist pragmatisch erfolgt, d.h. eine von Null verschiedene Fehlerwahrscheinlichkeit besitzt [90, 190].

Ihre Berechtigung erhält diese Vorgehensweise dadurch, daß auf den höheren Abstraktionsebenen, die für die Programmierung und Nutzung des System die entscheidende Rolle spielen, saubere und klare Konstrukte für die Problemlösung verfügbar sind.

5.3.1.2 Konsistenzmodell

Die Diskussionen in § 3 haben verdeutlicht, daß einige Interoperabilitätsprobleme die Konsistenz von Information innerhalb des verteilten Systems betreffen, das aus den einzelnen Werkzeugen sowie den überwachten Komponenten des Zielsystems besteht. Daher sollten bereits beim Entwurf der Monitorschnittstelle Konsistenzfragen berücksichtigt werden. Aufgrund der Struktur des Systems mit verteilten Werkzeugen einerseits und verteilten Monitoren andererseits ergeben sich zwei zueinander symmetrische Fragen:

1. Angenommen, durch zwei Werkzeuge oder zwei Ereignisse auf unterschiedlichen Knoten wird die Ausführung von je einer Aktionsliste a_1 bzw. a_2 angefordert. Was kann über die Reihenfolge ausgesagt werden, in der die Aktionen von den lokalen Monitoren ausgeführt werden?

2. Angenommen, zwei Werkzeuge A und B überwachen zwei Ereignisse e_1 und e_2 im Zielsystem, die auf verschiedenen Knoten auftreten. Was kann über die Reihenfolge ausgesagt werden, in der die Werkzeuge die beiden Ereignisse zu sehen bekommen?

Es sei angemerkt, daß diese beiden Fragen unabhängig von der in § 3.5.3 und § 4.3.2.2, Seite 123 diskutierten Problematik sind, in welcher Reihenfolge Aktionen unterschiedlicher Werkzeuge für *dasselbe* Ereignis ausgeführt werden sollen.

Reihenfolge der Ausführung lokaler Aktionen. Die hier auftretende Problematik kann am besten an einem praktischen Beispiel demonstriert werden: Ein Werkzeug A sendet eine Anfrage `thread_continue([])`, während Werkzeug B gleichzeitig `thread_stop([])` sendet. Da die beiden Vorgänge kausal unabhängig sind, ist das Resultat dabei selbstverständlich nicht determiniert. Aus Sicht eines Werkzeugentwicklers sind jedoch nur die beiden Resultate „alle Threads laufen“ und „alle Threads sind gestoppt“ akzeptabel. Eine Situation, in der einige Threads gestoppt sind und andere laufen, entspricht keiner der beiden Aktionen, kann aber in einer naiven Implementierung auftreten, wie Abb. 5.1 zeigt.

Um dies zu vermeiden, müssen Anfragen mit Hilfe eines *Atomic Multicast* verteilt werden. Dadurch ist sichergestellt, daß die lokalen Monitore die Teilanfragen alle in derselben Reihenfolge erhalten (logische Gleichzeitigkeit [199]). Arbeiten die lokalen Monitore die ankommenden Anfragen in FIFO-Reihenfolge ab, was sinnvollerweise der Fall ist, ergibt sich auf allen Knoten dieselbe Bearbeitungsreihenfolge der Aktionen, *falls* die lokalen Teile der Aktionslisten nicht-unterbrechbar bearbeitet werden. Aus Effizienzgründen kann eine derartige Nichtunterbrechbarkeit bei einer Schnittstelle wie OMIS jedoch nicht generell gefordert werden, da die Bearbeitung von lokalen Teilen einer verteilten Aktionsliste auch langdauernde Synchronisation beinhalten kann. Zudem ist die Nichtunterbrechbarkeit nur in wenigen Fällen wirklich erforderlich. In diesen Fällen muß sie allerdings von den Werkzeugen explizit spezifiziert werden können. Vorteilhaft ist es, diese Funktion so zu erweitern, daß nicht nur die markierte Aktionsliste nicht von anderen unterbrochen wird, sondern auch selbst keine anderen Aktionslisten unterbricht. Dies erleichtert die Interoperabilität, da es nun ausreicht, wenn eines der Werkzeuge ein mögliches Problem wie im obigen Beispiel vorhersieht und seine Aktionsliste als nicht-unterbrechbar markiert.

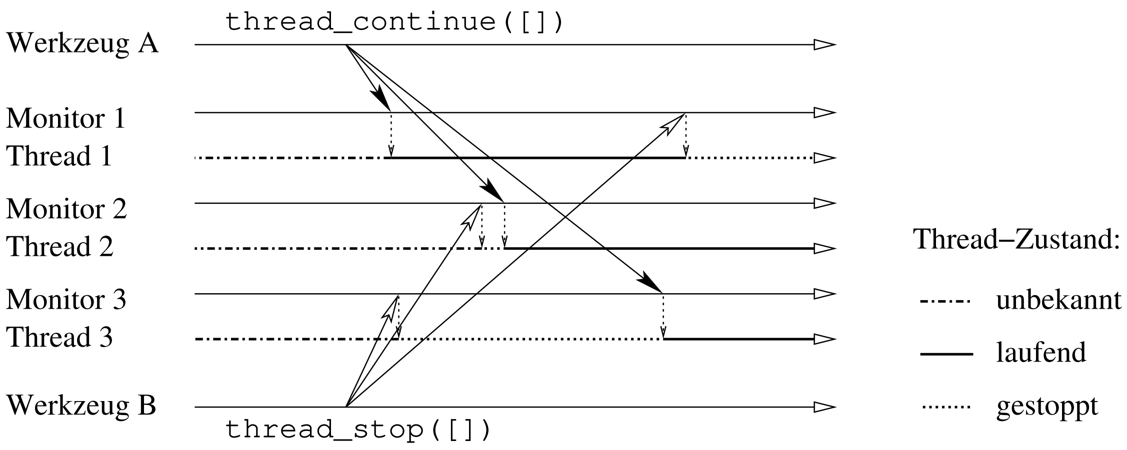


Abbildung 5.1: Mögliches Fehlverhalten bei gleichzeitigen, verteilten Monitoranfragen

Reihenfolge der Beobachtung von Ereignissen. Diese Fragestellung ist analog zu den bekannten Konsistenzfragen in SMPs, d.h. speichergekoppelten Multiprozessoren („*In welcher Reihenfolge sehen die Prozessoren Schreibzugriffe auf den gemeinsamen Speicher?*“) [57] oder Nachrichtensystemen („*In welcher Reihenfolge empfangen Prozesse Broadcast-Nachrichten verschiedener Sender?*“) [199].

Im vorliegenden Fall ist zunächst zu klären, was die Formulierung „ein Werkzeug sieht ein Ereignis“ exakt bedeuten soll. Bei Verwendung von OMIS muß prinzipiell unterschieden werden zwischen der Ausführung der von einem Werkzeug definierten Aktionen durch das Monitorsystem, die unter Umständen verteilt erfolgen muß, und der Ankunft der Antwort beim jeweiligen Werkzeug. Da bei unterbrechbaren Aktionslisten eine Aussage über die Reihenfolge ihrer Abarbeitung nicht immer möglich ist, muß folgende Festlegung erfolgen: Ein Werkzeug sieht ein Ereignis, wenn die Antwort der zugehörigen Aktionsliste bei ihm eintrifft. Für nicht-unterbrechbare Aktionslisten sollte das Monitorsystem sinnvollerweise garantieren, daß die Reihenfolge ihrer Abarbeitung der des Eintreffens der Antworten beim Werkzeug entspricht.

Die strengste mögliche Konsistenzanforderung für diesen Fall würde fordern, daß alle Werkzeuge für alle Ereignisse jeweils dieselbe Reihenfolge wahrnehmen. Dies entspricht der sequentiellen Konsistenz bei SMPs bzw. der logischen Gleichzeitigkeit bei Nachrichtensystemen. Für die oben genannte, in gewisser Hinsicht duale Situation der Bearbeitung von Monitoranfragen wurde dieses Konsistenzmodell als notwendig erkannt. Es führt allerdings zu einem nicht zu unterschätzenden Laufzeitaufwand. Bei der Anfrageabarbeitung kann dieser toleriert werden, da zum einen die Abarbeitung selbst bereits sehr aufwendig ist, zum anderen nur relativ selten Anfragen erfolgen.[2] Ereignisse treten dagegen wesentlich häufiger ein und erfordern meist nur eine sehr kurze interne Bearbeitungszeit. Die Effizienz des Konsistenzprotokolls spielt daher eine wesentlich stärkere Rolle, ebenso seine Skalierbarkeit, da in der Regel wesentlich mehr Rechnerknoten als Werkzeuge im System vorhanden sind.

Für Ereignismeldungen wird daher ein schwächeres, aber immer noch ausreichendes Konsistenzmodell verwendet, das *Weak Ordering* bei SMPs bzw. kausaler Ordnung bei Nachrichtensystemen entspricht. Zwei Werkzeuge sehen dabei zwei Ereignisse nur dann notwendigerweise in derselben Reihenfolge, wenn die Ereignisse im Sinne von Lamports *Happened-Before-*

[2]Da im OCM alle Anfragen über die zentrale NDU laufen ist logische Gleichzeitigkeit hier zudem mit geringerem Aufwand zu realisieren als bei den dezentralen Ereignismeldungen.

Relation [128] voneinander abhängig sind.

Zwei unabhängig (nebenläufig) auftretende Ereignisse $e_1 \parallel e_2$ können somit von einem Werkzeug als e_1, e_2, von einem anderen als e_2, e_1 wahrgenommen werden. Daß die Reihenfolge zweier unabhängiger Ereignisse in verteilten Systemen vom Beobachter abhängen kann, ist eine bekannte Tatsache, die auch in der Physik auftritt.[3] „Inkonsistenzen“ dieser Art sind eine inhärente Eigenschaft verteilter Systeme, auf die ein Benutzer einer Werkzeugumgebung eingestellt sein sollte. Insofern erscheinen sie im Interesse einer ausreichend effizienten Implementierung des Monitorsystems tolerierbar. Bei Bedarf könnte ein Werkzeug die Reihenfolge einiger Ereignisse auch intern korrigieren, wenn es das Monitorsystem dazu veranlaßt, ihnen einen Zeitstempel einer hinreichend genau synchronisierten, globalen Uhr zuzuordnen.

Bei kausal abhängigen Ereignissen $e_1 \mapsto e_2$ müssen alle beteiligten Werkzeuge die Folge $e1, e2$ sehen, da nun e_1 und e_2 sozusagen beweisbar in dieser Reihenfolge aufgetreten sind. Für das Monitorsystem hat dies zwei Konsequenzen:

1. Es muß sichergestellt werden, daß das kausal abhängige Ereignis e_2 von seinem lokalen Monitor erst nach dem Abschluß der Ereignisbehandlung für e_1 bearbeitet wird. Beim OCM wird diese Anforderung erfüllt, indem das auslösende Objekt eines Ereignisses bis zum Abschluß der Ereignisbehandlung angehalten und dadurch das Auftreten von e_2 verzögert wird.

2. Die für die Ereignismeldung notwendige Kommunikation zwischen Monitorprozessen und Werkzeugen muß die kausale Ordnung erhalten, d.h. wenn das Senden einer Nachricht an ein Werkzeug durch Monitor m_1 kausal vor dem Senden an dasselbe Werkzeug durch m_2 erfolgt, wird die Nachricht von m_1 auch als erste empfangen. Je nach Art des Verbindungsnetzes ist dazu eine Rückmeldung an den Sender erforderlich, daß der Empfang bzw. das Schreiben der Nachricht auf das Netz begonnen wurde.

Beide Anforderungen müssen im übrigen auch bereits deshalb erfüllt sein, damit ein einzelnes Werkzeug kausal geordnete Ereignisse in der richtigen Reihenfolge wahrnimmt.

5.3.2 Direkte Interaktionen zwischen Werkzeugen

Wie bereits mehrfach angesprochen, soll in dieser Arbeit die Realisierung direkter Interaktionen durch das Monitorsystem selbst erfolgen. Das Monitorsystem muß daher dazu in der Lage sein, Ereignisse auch in Werkzeugen zu erkennen und Aktionen auch in Werkzeugen auszuführen. Um dies zu erreichen sind lediglich zwei konzeptuelle Erweiterungen von OMIS und OCM erforderlich:

1. Das OMIS Objektmodell muß um die Objektklasse *Werkzeug* erweitert werden. Vorteilhafterweise wird dazu ein einfaches Vererbungskonzept eingeführt, das es gestattet, die Klasse *Werkzeug* von der vorhandenen Klasse *Prozeß* abzuleiten. Dies hat zur Folge, daß alle Prozeßdienste automatisch auch auf Werkzeuge anwendbar werden, insbesondere also auch die Überwachung von Funktionsaufrufen als Ereignisse.

2. Der OCM ist dahingehend auszubauen, daß Aktionen, d.h. Methoden eines Objekts, auch von dem jeweiligen Objekt selbst ausgeführt werden können. Bislang gibt es lediglich die Möglichkeit, Aktionen zu definieren, die in einem beliebigen Kontext, also insbesondere

[3] Relativität der Gleichzeitigkeit in der speziellen Relativitätstheorie [67].

auch in einem beliebigen Anwendungsprozeß ausgeführt werden können. Für die Realisierung direkter Interaktionen werden Aktionen benötigt, die nur im Kontext eines bestimmten „Anwendungsprozesses“ (d.h. Werkzeugs) ausgeführt werden können. Da der Ausführungsort von Aktionen gemäß OMIS für den Benutzer transparent ist, muß dazu zunächst die Anfrageverteilung des OCM entsprechend erweitert werden. Der eigentliche Aufruf einer derartigen Aktion erfolgt dann über einen *inferior RPC* (siehe § 2.1.3.2, Seite 23). Dabei sind Optimierungen möglich, etwa falls die Aktion innerhalb einer Aktionsliste steht, die einem Ereignis zugeordnet ist, das im betroffenen Objekt selbst erkannt wird.

Durch diese Konzepte wird es in Verbindung mit der dynamischen Erweiterbarkeit des OCM möglich, jede Funktion in einem Werkzeug über die Monitorschnittstelle aufzurufen. Da die eigentlichen Werkzeuge keine Information übereinander besitzen sollten, wird in der Regel nur das Rahmenwerkzeug einer Umgebung von dieser Möglichkeit Gebrauch machen. Zudem können Aktionsaufrufe an Ereignisse in Werkzeugen gekoppelt werden, wobei die Relation zwischen diesen über die Monitorschnittstelle frei programmierbar ist. Die Programmierung wird dabei normalerweise wieder vom Rahmenwerkzeug vorgenommen.

5.3.3 Indirekte Interaktionen und logische Konflikte

In diesem Unterabschnitt soll untersucht werden, wie die in § 5.1.4 genannten Mechanismen zur Behandlung logischer Konflikte auf der Basis des in § 4.3.2.2 definierten Modells in einem Monitorsystem realisiert werden können.

5.3.3.1 Modell des Zielsystemzustands

In § 4.3.2.2 wurde das Zustandsmodell des Zielsystems zunächst nur als ein Hilfsmittel zur manuellen Analyse von Konflikten eingeführt. Um die notwendigen Dienste zur Benachrichtigung über Modifikationen und zum Abfangen von Lesezugriffen realisieren zu können, muß dieses Modell nun in eine ausführbare Form gebracht werden. Dies ist deshalb notwendig, da die beiden Dienste voraussetzen, daß zum einen die Zustandskomponenten in definierter Weise bezeichnet werden können, zum anderen Zugriffe auf diese Komponenten für das System als solche erkennbar sind.

Die Umsetzung des Modells wirft eine Reihe von Fragen auf, die im folgenden beantwortet werden:

1. Auf welcher Abstraktionsebene soll die Modellierung erfolgen? Im Falle des OCM bieten sich zwei Ebenen an: die der OMIS-Schnittstelle, sowie die der OCM-internen Zielsystem-Schnittstelle.

2. Wie werden Zustandskomponenten bezeichnet? Welche Zugriffsmethoden sind möglich? Alternativen wären die Einschränkung auf reines Lesen und Schreiben oder das Zulassen von Methoden mit beliebiger Semantik.

3. Wie können Aliasing bzw. indirekte Auswirkungen von Modifikationen (siehe § 4.3.2.2, Seite 125) behandelt werden?

Abstraktionsebene. Damit Werkzeuge die Benachrichtigungs- und Abfang-Dienste, die auf Basis des Objektmodells realisiert werden, möglichst einfach nutzen können, wäre es wünschenswert, daß das Modell die Ebene der OMIS-Schnittstelle beschreibt. Bei einer ausschließlichen Modellierung auf dieser Ebene wird allerdings das mögliche Aliasing zu einem unüberwindbaren Problem. Beispielsweise verändert der OMIS-Dienst `proc_reset` den Speicherinhalt eines Prozesses, während `thread_get_backtrace` diesen liest. Die Abhängigkeit dieser beiden Dienste ist allein auf Ebene der OMIS-Schnittstelle nicht ersichtlich.

Um diese Art des Aliasing korrekt behandeln zu können, muß das Modell genügend feingranular sein. So muß es im Beispiel auch einzelne Zugriffe auf Speicherzellen berücksichtigen können. Im OCM sollte die Realisierung daher auf der Ebene der Zielsystemschnittstelle erfolgen, die diese feine Granularität bietet. Ein weiteres Argument für die Realisierung auf einer tieferen Schicht ist, daß in diesem Fall die Dienste zur Konfliktbehandlung auch innerhalb des Monitorsystems zur Verfügung stehen, z.B. für Monitor-Erweiterungen.

Um trotz der feingranularen Modellierung eine einfache Benutzbarkeit für die Werkzeuge sicherzustellen, wird die Realisierung des Modells eine Hierarchiebildung unterstützen, so daß die Abfang- und Benachrichtigungs-Dienste dann auch wieder auf höheren Abstraktionsebenen genutzt werden können.

Objektmodell. Da das in § 4.3.2.2 eingeführte Zustandsmodell nur für eine manuelle Analyse diente, mußten dort die konkreten Objekttypen und Zustandskomponenten nicht exakt definiert werden. Vielmehr konnten sie im Verlauf der Analyse je nach Situation mit geeigneter „Auflösung“ gewählt werden. Die Struktur eines ausführbaren Modells muß jedoch deutlich statischer sein, da es z.B. möglich sein soll, Zustandskomponenten im Quellcode des Monitors oder eines Erweiterungsmoduls durch a-priori bekannte Namen zu bezeichnen. Eine vollkommen statische Festlegung aller Komponentennamen scheidet andererseits jedoch aus, da das Monitorsystem – und damit auch die Menge der Zustandskomponenten – erweiterbar sein muß.

Als Basis verwenden wir zunächst das durch OMIS definierte Objektmodell mit den Objekttypen *System*, *Knoten*, *Prozeß*, etc. Die Objekte werden wie bisher durch zur Laufzeit generierte Identifikatoren, die Objekttoken, bezeichnet. Jedes Objekt kann eine Hierarchie von Zustandskomponenten besitzen; ein Prozeß beinhaltet so z.B. eine Komponente *Speicher*, die ihrerseits aus *Speicherzellen* besteht. Eine Zustandskomponente wird dann durch die Angabe eines Objekttokens sowie einer Folge von Selektoren identifiziert, wobei als Selektoren Zeichenketten und ganze Zahlen Verwendung finden. Eine Konkretisierung eines derartigen Zustandsmodells für die Basisfunktionen des OCM ist in der Dissertation von J. Trinitis zu finden [235].

Da zur Behandlung von Aliasing die Schreib- und Leseoperationen auf jede einzelne Zustandskomponente exakt bekannt sein müssen, wird ein Zugriff auf Objekte nur über die vordefinierten Methoden *Lesen* und *Schreiben* erlaubt. Diese Methoden sind sowohl für einzelne Komponenten als auch für Aggregate zugelassen, die durch nicht-maximale Folgen von Selektoren bezeichnet werden. Ihre Schnittstellen sind wie folgt definiert:

```
Value read(Token, Selector);
void write(Token, Selector, Value);
```

`Value` ist dabei ein universeller Typ, der beliebige Daten aufnehmen kann.

Bis hierhin entspricht das Zustandsmodell also lediglich der Definition zusammengesetzter Datenstrukturen, allerdings mit expliziten Schreib- und Lesemethoden.

Besonderheiten des Objektmodells. Im Vergleich zu einfachen Datenstrukturen weisen die innerhalb des Monitorsystems benötigten Objekte eine Reihe von Besonderheiten auf. Zunächst sind nur bei Monitorobjekten (siehe § 3.2.1.1), die vom Monitorsystem selbst definiert werden, die zugehörigen Daten direkt in dem Objekt gespeichert. Systemobjekte stellen dagegen lediglich Stellvertreter der „realen" Objekte des Zielsystems dar. In Systemobjekten sind daher nur monitorinterne Verwaltungsdaten gespeichert. Alle anderen Daten werden direkt aus dem Zielsystem gelesen oder in dieses geschrieben, wobei das Systemobjekt lediglich den Namensraum für die einzelnen Zustandskomponenten definiert. Ein Schreibzugriff auf eine Komponente eines Systemobjekts führt daher zu einer Modifikation des Zielsystems, ein Lesezugriff auf eine Komponente zur Ermittlung der entsprechenden Information über das Zielsystem. Ein Zugriff `write(p_16, stopped, True)` würde also beispielsweise zum Anhalten von Prozeß `p_16` führen. Natürlich ist jedoch auch ein *Caching* von Daten eines „realen" Objekts in dem zugeordneten Systemobjekt möglich.

Wichtig ist in diesem Zusammenhang zudem, daß die Systemobjekte nur ein vereinfachtes Modell des Zielsystems darstellen können. Insbesondere bilden sie kein vollständiges Modell, sie beschreiben also in der Regel nur einen Teil der realen Objekte im Zielsystem. Da das Zielsystem während seiner Lebensdauer um neue (Zustands-)Komponenten erweitert werden kann, ist ein vollständiges Modell a-priori auch nicht realisierbar. Unser Objektmodell muß daher ebenfalls dynamisch erweiterbar sein. Dies erlaubt dann nicht nur die Einbeziehung neuer Zustandskomponenten des Zielsystems, sondern auch den Aufbau von Abstraktionen durch Erweiterungsmodule des Monitorsystems. Diese können also neue Komponenten eines Objekts einführen, die eine Abstraktion bereits bestehender Komponenten darstellen. Ein Beispiel ist der Dienst `thread_get_backtrace`, der eine neue Sichtweise auf den Speicher und die Register eines Threads einführt, die als neue Zustandskomponente *Prozeduraufrufkeller* aufgefaßt werden kann. Wie bereits mehrfach erwähnt, müssen derartige Erweiterungen unabhängig voneinander möglich sein, d.h. durch die Verwendung des Objektmodells dürfen keine zusätzlichen Abhängigkeiten zwischen Erweiterungen entstehen. Im Interesse eines modularen Aufbaus des Monitorsystems sollten zudem die Mechanismen zum Melden bzw. Abfangen von Zugriffen generisch sein, ihre Implementierung also unabhängig von den konkreten Objekttypen und eventuellen Erweiterungen sein.

Schließlich müssen diese Mechanismen auch dazu in der Lage sein, Aliasing korrekt zu berücksichtigen. Derartiges Aliasing kann durch die oben erwähnte Einführung neuer Sichten verursacht werden. So kann z.B. eine Modifikation des Speichers eines Prozesses auch dessen Prozeduraufrufkeller ändern, so daß ein entsprechendes Ereignis ausgelöst werden muß, um diese Änderung zu melden. Eine zweite Form des Aliasing wird durch die in § 4.3.2.2, Seite 125 diskutierten indirekten Folgen von Modifikationen verursacht. Um diese berücksichtigen zu können, müssen sie explizit gemacht werden, was am einfachsten durch die Einführung einer dritten Zugriffsmethode auf Zustandskomponenten erfolgen kann. Diese Methode ist ähnlich der Schreibmethode, zeigt aber lediglich eine Veränderung der Komponente an, die durch das System erfolgt ist, anstatt selbst eine derartige Veränderung zu veranlassen. Die Funktion der Methode ist somit analog zu einer *Cache-Update*-Nachricht in Mehrprozessorsystemen mit verteiltem gemeinsamem Speicher.

Melden und Abfangen von Zugriffen. Die prinzipielle, sehr einfache Idee der Methoden zum Melden und Abfangen von Objektzugriffen ist in Abb. 5.2 dargestellt. Über die Methode `notify` wird dem System mitgeteilt, daß bestimmte Schreibzugriffe gemeldet werden sollen. Die Methode `catch` veranlaßt, daß Lesezugriffe abgefangen werden und ihr Ergebnis

```
void notify(Token t, Selector s, Destination d)
{
   speichere (t, s, d) in einer geeigneten Datenstruktur notifylist
}

void catch(Token t, Selector s, Value (*cb)(Value v))
{
   speichere (t, s, cb) in einer geeigneten Datenstruktur catchlist
}

Value read(Token t, Selector s)
{
   Value v = Wert der Komponente s von t
   for each (t,s,cb) ∈ catchlist {
      v = cb(v)
   }
   return v
}

void write(Token t, Selector s, Value v)
{
   schreibe v in die Komponente s von t
   for each (t,s,d) ∈ notifylist {
      Sende Benachrichtigung an d
   }
}
```

Abbildung 5.2: Prinzipielles Vorgehen zum Melden und Abfangen von Objektzugriffen

modifiziert werden kann. Beide Methoden speichern lediglich ihre Parameter in geeigneten Datenstrukturen ab. Bei jedem Zugriff werden dann das Objekttoken und der Selektor mit der gespeicherten Information verglichen und der Zugriff gegebenenfalls gemeldet oder abgefangen. Der genaue Mechanismus der eigentlichen Meldung von Schreibzugriffen ist dabei ebenso unerheblich, wie der konkrete Abfangmechanismus, der in Abb. 5.2 beispielhaft durch eine Rückruffunktion realisiert ist. Die in Abb. 5.2 dargestellten Methoden berücksichtigen nicht, daß ein Lesen und Schreiben sowie das Melden und Abfangen auch für Aggregate möglich sein muß. Die Berücksichtigung von Aggregaten ist durch eine einfache Erweiterung möglich; ebenso einfach kann erreicht werden, daß den Methoden `notify` und `read` anstelle von einzelnen Token und Selektoren Muster bzw. Mengen übergeben werden können. Im Interesse der Klarheit der Darstellung wurde hier jedoch auf diese Erweiterungen verzichtet. Schließlich soll erwähnt werden, daß die Schleife in der Methode `catch` bei Abwesenheit von Abfang-Konflikten höchstens einmal durchlaufen wird (vgl. § 4.3.2.2, Seite 124).

Im folgenden wird gezeigt, wie aufbauend auf diesen Basismethoden die Forderung nach Erweiterbarkeit der Objekte und Berücksichtigung von Aliasing erfüllt werden kann. Zunächst ist unmittelbar ersichtlich, daß ein einfacher Vergleich der Selektoren bei möglichem Aliasing nicht mehr ausreichend ist. Andererseits muß erreicht werden, daß unabhängige Erweiterungsmodule keine Information übereinander benötigen müssen, um das Melden bzw. Abfangen von Zugriffen realisieren zu können. Schließlich müssen Werkzeuge dazu in der Lage sein, Zugriffe auf derselben Ebene abzufangen, auf der sie eine Modifikation vorgenommen haben. Wenn also z.B. ein Werkzeug die oben angesprochene Komponente *Prozeduraufrufkeller* modifiziert, soll es nicht dazu gezwungen werden, Lesezugriffe auf die Komponenten *Speicher* und *Register* abzufangen, um die Änderung vor anderen Werkzeugen zu verbergen.

Um eine Lösungsidee zu entwickeln werden im folgenden die verschiedenen Situationen betrachtet, die bei der Erweiterung eines Objekts um neue Zustandskomponenten auftreten können:

1. **Zustandskomponenten, die von vorhandenen Komponenten abgeleitet sind.**

   (a) Wir betrachten zunächst den Fall, in dem nur ein einziges Erweiterungsmodul betroffen ist. Diese leite eine Zustandskomponente x' von vorhandenen Komponenten $x_1, x_2, ...$ ab. Für den Wert von x' gilt also $x' = F(x_1, x_2, ...)$ mit geeignetem F. Zunächst kann festgehalten werden, daß das Erweiterungsmodul alle Komponenten x_i kennen muß, also in jedem Fall von den Modulen abhängig ist, die die x_i definieren.

   Um den Wert von x' zu liefern, muß das Erweiterungsmodul eine Lesemethode realisieren, die zunächst alle x_i liest. Damit diese Lesezugriffe bei Bedarf auf der Ebene der x_i abgefangen werden können, müssen sie mit Hilfe der überwachten Lesemethoden der x_i erfolgen. Dies führt letztlich, eventuell induktiv, zu Aufrufen der Basis-Lesemethode aus Abb. 5.2. Der zurückgegebene Wert berechnet sich dann zu $x' = F(x_1, x_2, ...)$.

   Falls x' eine schreibbare Komponente darstellt, muß das Erweiterungsmodul zusätzlich eine geeignete Schreibmethode definieren. Diese muß den zu schreibenden Wert x'_{neu} in entsprechende neue Werte für die x_i umformen. Das Schreiben dieser neuen Werte muß dabei wieder durch die überwachten Schreibmethoden der x_i erfolgen, damit ggf. eine Modifikationsmeldung erzeugt werden kann. Im allgemeinen kann der neue Wert für ein x_i sowohl von x'_{neu} als auch von den alten Werten aller x_j abhängen, was als $x_{i,neu} = F_i^{-1}(x'_{neu}, x_1, x_2, ...)$ geschrieben werden kann. Die Funktionen F_i^{-1} müssen dabei offensichtlich die folgenden Gleichungen erfüllen:

   $$\begin{aligned} x' &= F(F_1^{-1}(x', x_1, x_2, ...), F_2^{-1}(x', x_1, x_2, ...), ...) \\ x_i &= F_i^{-1}(F(x_1, x_2, ...), x_1, x_2, ...) \end{aligned}$$

   Sie stellen also in gewissem Sinne die Umkehrfunktion von F dar.

   Zu bemerken ist an dieser Stelle, daß das Erweiterungsmodul F und F_i^{-1} in jedem Fall definieren müßte, auch wenn keine Melde- und Abfangmechanismen zu realisieren wären.

   Die Methode `notify` für x' kann nun durch einen einfachen Aufruf der `notify`-Methode für jedes x_i implementiert werden. Der Umweg über die x_i ist dabei notwendig, da der Wert von x' auch durch Modifikation eines der x_i verändert werden kann. Eine Modifikation von x' führt ebenfalls zu einer Benachrichtigung, da sie auf einen Schreibzugriff auf die x_i zurückgeführt wird. Das Aliasing zwischen x' und den x_i ist somit korrekt berücksichtigt. Allerdings kann es beim Schreiben auf x' zu

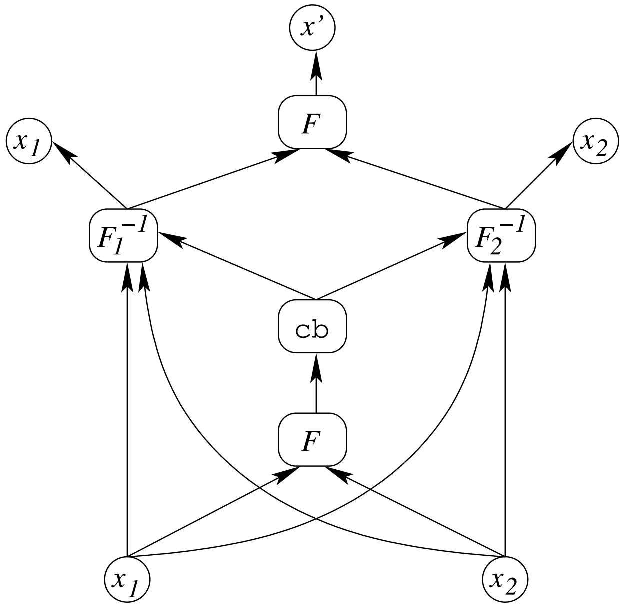


Abbildung 5.3: Datenfluß beim Abfangen von Lesezugriffen

Mehrfachbenachrichtigungen kommen. Diese könnten zum einen durch geeignete Filter wieder zusammengefaßt werden. Zum anderen können sie vermieden werden, indem in der Schreibmethode von x' zunächst geprüft wird, ob Modifikationen von x' gemeldet werden sollen. Falls ja, wird die Meldung erzeugt und die *von der* `notify`*-Methode von* x' angeforderte Überwachung der x_i temporär abgeschaltet.

Beim Abfangen von Lesezugriffen kann analog vorgegangen werden. Um eine Modifikation von x' vor anderen Werkzeugen zu verbergen, müssen auch alle Lesezugriffe auf die x_i abgefangen werden, da letztendlich diese verändert wurden. Die `catch`-Methode von x' ruft daher die `catch`-Methode für jedes der x_i jeweils mit einer Rückruffunktion auf, die den Wert $F_i^{-1}(\texttt{cb}(F(x_1, x_2, ...)), x_1, x_2, ...)$ zurückliefert, wobei `cb` die an die Methode `catch` von x' übergebene Rückruffunktion ist. Um eine endlose Rekursion zu vermeiden, muß dabei während des Lesen der x_i das *von der* `catch`*-Methode von* x' angeforderte Abfangen der Zugriffe auf die x_i temporär abgeschaltet werden. Unter der Voraussetzung, daß keine Abfang-Konflikte (siehe § 4.3.2.2, Seite 124) vorliegen, können die Werte der x_i alternativ auch direkt, also nicht durch die überwachten Lesemethoden, ermittelt werden.

Der entstehende Datenfluß ist in Abb. 5.3 dargestellt. Aus den Werten der x_i wird über F zunächst der entsprechende Wert von x' berechnet und der Rückruffunktion `cb` übergeben. Ihr Ergebnis wird dann durch F_i^{-1} in einen neuen Wert für x_i transformiert, der das Resultat des Lesezugriffs auf x_i bestimmt. Bei einem Lesezugriff auf x' wird aus den so bestimmten Werten der x_i wieder ein Wert für x' berechnet, der wegen der Eigenschaften von F und F_i^{-1} mit dem von der Rückruffunktion gelieferten Wert übereinstimmt. Es kann daher eine Optimierung vorgenommen werden, indem die Lesemethode von x' zunächst prüft, ob der Zugriff auf x' abgefangen werden soll. Falls ja, wird die Rückruffunktion direkt aufgerufen, und ihr Wert als Ergebnis geliefert. Falls nein, wird wie weiter oben beschrieben verfahren.

Anzumerken ist, daß die von der `catch`-Methode benötigen Funktionen F_i^{-1} nur

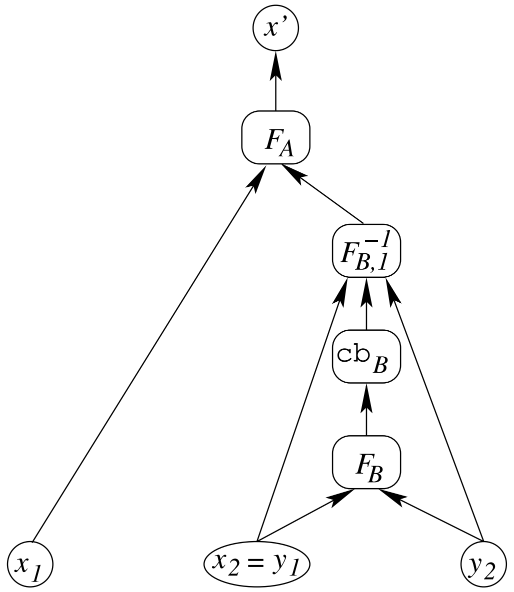


Abbildung 5.4: Datenfluß beim Abfangen von Lesezugriffen überlappender Komponenten

verfügbar sind, falls x' eine schreibbare Komponente darstellt. Ist dies nicht der Fall, ist die Realisierung dieser Methode jedoch auch nicht erforderlich.

(b) Es soll nun der allgemeinste Fall untersucht werden, bei dem Aliasing nur durch abgeleitete Komponenten verursacht wird.

Wir gehen davon aus, daß ein Schreib- bzw. Lesezugriff auf eine Komponente x' erfolgt und eine Meldung bzw. ein Abfangen von Zugriffen auf eine Komponente y' angefordert wurde. Dabei sei x' durch ein Erweiterungsmodul A aus $x_1, x_2, ...$ abgeleitet, y' durch ein Modul B aus $y_1, y_2, ...$ Es gelte also $x' = F_A(x_1, x_2, ...)$ und $y' = F_B(y_1, y_2, ...)$. Es sei angenommen, daß ein Aliasing zwischen x' und y' besteht, d.h. die Modifikation einer der Komponenten auch eine Änderung der anderen bewirkt. Dann muß jedoch auch bereits mindestens eines der x_i ein Alias für ein y_j darstellen, insbesondere kann $x_i = y_j$ gelten.

Ein Schreibzugriff auf x' führt nun zum Schreiben auf x_i. Dieses führt – eventuell induktiv – zur Meldung der Modifikation von y_j, wodurch aufgrund der oben skizzierten Vorgehensweise auch eine Meldung für y' erfolgt[4].

Wurde ein Abfangdienst für y' installiert, und ein Lesezugriff auf x' erfolgt, wird dieser ebenfalls korrekt modifiziert: Das Lesen von x' führt zu einem Lesezugriff auf x_i, dessen Ergebnis – eventuell induktiv – durch den von der `catch`-Methode von y' installierten Abfangdienst für y_j bestimmt wird. Der Datenfluß für eine Beispielsituation ist in Abb. 5.4 dargestellt. Der Lesezugriff auf x' wird also zunächst bis zu den Basiskomponenten propagiert. Dort erfolgt das eigentliche Abfangen, das durch ein Hochtransformieren des Wertes bis zur Abstraktionsebene der Rückruffunktion und anschließender Rücktransformation des Ergebnisses auf die Ebene der Basiskomponenten erfolgt. Aus den so modifizierten Werten wird schließlich der resultierende Wert für x' bestimmt.

[4]Auch hier können evtl. mehrere Meldungen erzeugt werden, was die korrekte Funktion jedoch nicht beeinträchtigt.

Das Melden und Abfangen von Zugriffen kann also trotz der „Überlappung“ der von den Erweiterungsmodulen A und B definierten Komponenten so realisiert werden, daß keinerlei Abhängigkeit zwischen A und B entsteht.

2. **Neue Zustandskomponenten.**

   (a) Zunächst soll der Fall betrachtet werden, in dem ein Erweiterungsmodul A eine neue Komponente eines Objekts einführt, deren Wert ausschließlich in dem Objekt gespeichert wird. Wie auf Seite 142 ausgeführt, ist dies bei Monitorobjekten sowie bei reinen Verwaltungsdaten von Systemobjekten der Fall. Da in diesem Fall für die Komponente ein neuer Speicherplatz reserviert wird, kann kein Aliasing mit bereits existierenden Komponenten auftreten. Demzufolge muß ein anderes Modul B von der Existenz von A nur dann Kenntnis nehmen, wenn es Komponenten aus denen von A ableitet.

   (b) Falls ein Modul A eine neue Komponente einführt, die einen Stellvertreter einer Zustandskomponente des Zielsystems darstellt, ergibt sich eine andere Situation. In diesem Fall lesen bzw. verändern die Zugriffsmethoden auch den Zustand außerhalb des Objektmodells, so daß z.B. ein Schreibzugriff auf eine von A definierte Komponente x indirekt auch eine Komponente y modifizieren könnte, die von einem anderen Modul B definiert wurde (siehe § 4.3.2.2, Seite 125). Damit die oben skizzierten Algorithmen auch in diesem Fall korrekt funktionieren, muß A durch die bereits auf Seite 142 erwähnte *Update*-Methode die indirekte Modifikation von y bekanntgeben. Analog müssen zur Herstellung der Transparenz nach einer Änderung von x explizit auch die Lesezugriffe auf y abgefangen werden. In diesem Fall ergibt sich also eine deutliche Abhängigkeit zwischen den Modulen A und B. Diese ist aber bereits aus den in § 5.3.1.1 genannten Gründen unvermeidbar.

Durch die skizzierten Melde- und Abfangmechanismen werden also keine *zusätzlichen* Abhängigkeiten zwischen Erweiterungsmodulen induziert, so daß sie die eingangs gestellten Forderungen vollständig erfüllen. Ob das skizzierte Objektmodell jedoch mit ausreichender Effizienz in einem Monitorsystem implementiert werden kann, kann derzeit noch nicht entschieden werden.

Zugriffsebenen. In dem bisherigen Entwurf sind die Zugriffsebenen, die in § 4.3.2.2 eingeführt wurden, noch nicht berücksichtigt. Dies würde bedeuten, daß auf Anforderung *alle* Schreibzugriffe gemeldet würden, auch solche, die für das betroffene Werkzeug gar nicht sichtbar sein sollten. Ebenso würden *alle* Lesezugriffe abgefangen, für die ein entsprechender Dienst definiert wurde.

Um ein korrektes Verhalten zu erreichen, müssen daher alle Zugriffe mit ihrer resultierenden Zugriffsebene (vgl. Abb. 4.2) gekennzeichnet werden. Dann kann durch einen einfachen Vergleich der Ebenen entschieden werden, ob tatsächlich eine Meldung bzw. ein Abfangen erfolgen muß:

- Ein Schreibzugriff wird nur dann gemeldet, wenn dessen Zugriffsebene größer oder gleich der Ebene der Benachrichtigungsanforderung ist. Letztere muß identisch mit der Ebene des Lesezugriffs sein, der für das Konsistenzproblem verantwortlich ist, das mit Hilfe der Benachrichtigung gelöst werden soll.

- Ein Lesezugriff wird nur dann abgefangen, wenn seine Zugriffsebene größer als die Ebene der Abfanganforderung ist, die identisch zu der des vorhergegangenen, transparent zu gestaltenden Schreibzugriffs sein muß.

Wie in § 4.3.2.2 bereits erläutert wurde, ist die Gesamtzahl der Schichten jedoch von der konkreten Werkzeugumgebung abhängig. Ein einzelnes Werkzeug kann lediglich zwischen Anwendungs- und Systemebene unterscheiden, deren Grenze jedoch bei verschiedenen Werkzeugen unterschiedlich ist. Das Wissen über die Abbildung der Ebenen eines einzelnen Werkzeugs auf das globale Schichtenmodell der Werkzeugumgebung kann also nur von dem Rahmenwerkzeug geliefert werden, das alle Bestandteile der Umgebung kennt.

Dies führt zur folgenden Vorgehensweise: Beim Start durch das Rahmenwerkzeug erhält jedes Werkzeug zunächst einen eindeutigen Identifikator. Beim Aufruf eines Dienstes spezifiziert das Werkzeug jeweils seine private Zugriffsebene. Die Umsetzung auf die globale Ebene erfolgt dann innerhalb des Monitorsystems durch eine Tabelle, analog zu Tab. 4.2, die diesem während der Initialisierung vom Rahmenwerkzeug bekanntgegeben wird. Die resultierende Ebeneninformation wird nun innerhalb der Dienste bis zu den eigentlichen Zugriffen (bzw. Aufrufen der Methoden `notify` und `catch`) propagiert. Rein monitorinternen Zugriffen kann dabei die Ebene 0 zugeordnet werden, so daß diese nie an Werkzeuge gemeldet werden und von diesen auch nicht abgefangen werden können.

Das skizzierte Vorgehen impliziert, daß alle Zugriffe, die ein von einem Werkzeug angeforderter Dienst ausführt, auf derselben Ebene stattfinden müssen. Dies stellt keine Einschränkung dar, da zum einen diese Bedingung praktisch immer erfüllt ist, zum anderen falls erforderlich ein Erweiterungsmodul für das Monitorsystem erstellt werden kann, das einen betroffenen Dienst in Form mehrerer Teildienste zur Verfügung stellt.

Gäben die Werkzeuge dem Monitorsystem nun auch noch das Ende jedes Zugriffs bekannt, wäre es prinzipiell möglich, dieses so auszubauen, daß Konflikte automatisch erkannt und behandelt werden. Allerdings würde sich durch den verdoppelten Kommunikationsaufwand mit den Werkzeugen sowie durch die erforderliche Analyse aller Zugriffe die Rückwirkung des Monitorsystems auf die überwachte Anwendung deutlich erhöhen, so daß der Nutzen einer derartigen Erweiterung fraglich ist.

5.3.3.2 Reihenfolgesteuerung von Aktionslisten und Ereigniserkennung

Ausführungsreihenfolge von Aktionslisten. Bei einem Monitorsystem, das nach dem Ereignis/Aktions-Schema arbeitet, ist die Abarbeitungsreihenfolge von Aktionslisten, die für dasselbe Ereignis definiert wurden, zunächst nicht festgelegt. Die Diskussion in § 4.3.2.2, Seite 123 hat andererseits gezeigt, daß die Reihenfolge der Aktionsausführung einen großen Einfluß auf die entstehenden Konflikte zwischen den Aktionen verschiedener Werkzeuge hat. Ein Monitorsystem sollte daher Mechanismen vorsehen, mit deren Hilfe ein Werkzeug bei der Definition einer Ereignis/Aktions-Relation festlegen kann, wann die spezifizierte Aktionsliste relativ zu anderen Aktionslisten für dasselbe Ereignis ausgeführt wird. Dies muß möglich sein, ohne daß das Werkzeug dazu Wissen über die anderen Ereignis/Aktions-Relationen oder die Werkzeuge besitzen muß, die diese definiert haben.

Dadurch scheidet ein einfaches Prioritätsschema aus, da ein Werkzeug die von den anderen Werkzeugen gewählten Prioritäten nicht kennen kann. Die Möglichkeit, Aktionen am Anfang bzw. am Ende der globalen Aktionsliste eines Ereignisses einfügen zu können, führt ebenfalls nicht zum Ziel, wie in § 5.2.4 am Beispiel von `dyninst` gezeigt wurde. Natürlich bestünde die Möglichkeit, das Rahmenwerkzeug einer Werkzeugumgebung Prioritäten festlegen zu lassen, indem es etwa bei der Initialisierung an das Monitorsystem eine Tabelle sendet, die jedem Paar (*Werkzeug*, *Ereignis*) eine entsprechende Priorität zuordnet. Die Einbeziehung einer zentralen Instanz ist für diese Aufgabe jedoch nicht notwendig, wie die folgenden Überlegungen zeigen.

Wir betrachten dazu die Situation bei den verschiedenen Klassen von Ereignis/Aktions-Relationen genauer:

1. **Ereignisse für den Beginn oder das Ende eines Bibliotheksaufrufs.**

   (a) **Aktionen, die die Parameter bzw. das Ergebnis dieses Aufrufs lesen bzw. modifizieren.**
   Aufgrund der Überlegungen aus § 4.3.2.2, Seite 123 kann in diesem Fall eine konfliktfreie Reihenfolge durch das Monitorsystem bestimmt werden, sofern dieses Information über die Zugriffsschicht jeder Aktion besitzt. Diese Information muß für das Monitorsystem bereits verfügbar sein, um die Melde- und Abfangmethoden realisieren zu können, wie sie im vorangegangenen Abschnitt diskutiert wurden. Da zwei Werkzeuge nur dann denselben Bibliotheksaufruf überwachen können, wenn sie auf Ebene derselben Schnittstelle arbeiten, folgt, daß zumindest in bezug auf diesen Bibliotheksaufruf und seine Parameter die Schichtenstruktur der Werkzeuge identisch sein muß. Eine Abbildung der lokalen Zugriffsebene auf eine globale Ebene, wie sie in § 5.3.3.1, Seite 147 eingeführt wurde, ist in diesem Fall also nicht erforderlich (siehe auch § 4.3.2.2, Seite 121).

   (b) **Aktionen, die andere Zustandskomponenten lesen bzw. modifizieren.**
   Da diese Zustandskomponenten jederzeit auch durch unbedingte Anfragen anderer Werkzeuge gelesen oder geschrieben werden können, müssen in diesem Fall immer Maßnahmen zur Konsistenz- und Transparenzsicherung getroffen, d.h. betroffene Zugriffe überwacht bzw. abgefangen werden. Damit sind aber auch diejenigen Konflikte aufgelöst, die durch Aktionslisten anderer Werkzeuge für dasselbe Ereignis verursacht werden können, so daß die Ausführungsreihenfolge unerheblich ist.

2. **Andere Ereignisse.** In diesem Fall sind nur Aktionen möglich, wie sie in 1b) bereits diskutiert wurden.

Im folgenden muß also nur noch der Fall 1a) behandelt werden. Eine konfliktfreie Reihenfolge kann hier durch Anwendung von Tabelle 4.3 gefunden werden. Um den Analyseaufwand zur Laufzeit zu verringern und um nicht zwischen den Ereignisklassen *Aufruf* und *Rückkehr* unterscheiden zu müssen, kann auch ein einfacheres Schema verwendet werden, das lediglich zusammenfassende Information über komplette Aktions*listen* nutzt. Jeder Aktionsliste werden dazu zwei Boolesche Attribute zugeordnet: Das Attribut `modifying` gibt an, ob die Aktionsliste den Zielsystemzustand verändert, während das Attribut `need_unmodified` spezifiziert, ob die Aktionsliste den unveränderten Zustand lesen will, der nach Eintreten des Ereignisses vorlag, oder den, der nach Anwendung aller (anderen) modifizierenden Aktionslisten für dieses Ereignis resultiert. Wird ein Funktionsaufruf überwacht, so ist `need_unmodified` gesetzt, wenn die Aktionsliste Lesezugriffe auf Anwendungsebene durchführt. Bei der Überwachung der Rückkehr ist `need_unmodified` dagegen zu setzen, wenn die Lesezugriffe auf Systemebene erfolgen.

Aus den beiden Attributen ergeben sich somit vier Klassen, die unterschiedlichen Prioritäten entsprechen:

1. `modifying = False, need_unmodified = True`

2. `modifying = True,  need_unmodified = True`
3. `modifying = True,  need_unmodified = False`
4. `modifying = False, need_unmodified = False`

Aktionslisten der Priorität 1 müssen dabei als erste abgearbeitet werden. Zu bemerken ist, daß sich in den Klassen 2 und 3 jeweils nur eine Aktionsliste befinden kann. Anderenfalls liegen Schreib/Schreib-Konflikte vor. Da jedoch zwei Klassen für modifizierende Aktionslisten vorhanden sind, können mit Hilfe dieses Schemas immerhin auch einige derartige Konflikte vermieden werden.

Durch die Zusammenfassung der Information über die einzelnen Aktionen unterliegt das Schema andererseits auch gewissen Einschränkungen. So setzt es voraus, daß die Zugriffsart für alle Aktionen bzw. Zustandskomponenten identisch ist. Da zudem nicht mehr zwischen Zugriffen auf verschiedene Zustandskomponenten unterschieden wird, können zwei Aktionslisten in dieselbe Klasse eingeordnet werden,[5] obwohl bei genauerer Betrachtung eine konfliktfreie Reihenfolge gefunden werden könnte. Die bisherige Erfahrung hat jedoch gezeigt, daß das grobgranulare Schema in der Praxis völlig ausreichend ist. Eine Ausweitung des Schemas, bei der die Attribute getrennt für jede Zustandskomponente definiert werden können, wäre jedoch möglich.

Ereigniserkennung. Beim Entwurf der Infrastruktur wurde bisher noch nicht berücksichtigt, daß Transparenzprobleme auch im Zusammenhang mit Ereignissen auftreten können (siehe § 4.3.2.2, Seite 126). Nimmt ein Werkzeug eine Instrumentierung (allgemeiner: eine beliebige Modifikation) am Zielprogramm vor, um ein Ereignis auszulösen, muß es gleichzeitig einen neuen Ereignisfilter im Filterbaum installieren, der erkennt, ob das Ereignis tatsächlich durch diese Instrumentierung ausgelöst wurde (vgl. § 3.2.1.2). Dieser Filter kann entweder bereits im Monitorsystem vorhanden sein oder in einer Werkzeug-Erweiterung neu definiert werden. Er kann zugleich zur Sicherung der Transparenz verwendet werden. Dazu muß lediglich beim Auslösen des Filters das Basisereignis „konsumiert“ werden, so daß es für andere Filter und damit auch andere Werkzeuge nicht mehr sichtbar ist. Analog zu § 5.3.3.1, Seite 147 müssen dabei wieder die Ebenen, auf denen die Werkzeuge die Instrumentierung bzw. Ereignisüberwachung durchführen, entsprechend berücksichtigt werden.

Dieses Konzept kann zusammen mit der zuvor beschriebenen Reihenfolgesteuerung von Aktionslisten in relativ effizienter Weise beim Durchlauf durch den Filterbaum innerhalb des OCM umgesetzt werden. Bei dem Baumdurchlauf werden die Filter zunächst wie bisher ausgewertet. Beim Auslösen eines Filters werden die angehängten Aktionslisten jetzt jedoch entsprechend ihrer Priorität an eine von vier Warteschlangen angefügt. Diese werden erst am Ende des Durchlaufs zusammengefügt und anschließend abgearbeitet. Triggert während des Durchlaufs ein konsumierender Filter, müssen alle Aktionslisten wieder gelöscht werden, die bereits vorher durch einen Filter an eine der vier Warteschlangen angefügt wurden, der einer höheren Beobachtungsebene entspricht. Ebenso muß dafür gesorgt werden, daß beim weiteren Baumdurchlauf alle derartigen Filter ignoriert werden.

Zu beachten ist dabei, daß Filter auch gänzlich neue Ereignisse erkennen können. Ein Beispiel innerhalb des OCM stellt der Filter dar, der die Ankunft einer Nachricht für den OCM erkennt. Er hängt im Filterbaum unter dem Ereignis „*ein SIGIO-Signal trifft ein*“ und prüft bei jedem Aufruf, ob neue Nachrichten vorhanden sind. Derartige „generierende“ Filter stellen eine Grenze für das genannte Vorgehen dar, d.h. der Unterbaum unter einem solchen Filter muß

[5] Kritisch sind hier nur die Klassen 2 und 3.

vollkommen unabhängig vom Rest des Filterbaums betrachtet werden. Im Beispiel in Abb. 5.6 kann sich der konsumierende Filter f daher nicht auf b auswirken.

Eine Optimierung des skizzierten Baumdurchlaufs ist möglich, wenn die Attribute der Filter analog der Attributierung von Syntaxbäumen in Compilern [6] nach oben propagiert werden. Für jeden Knoten des Baums enthält das Attribut `min_consume_level` die minimale Beobachtungsebene, auf der das aktuelle Ereignis von einem Filter des von diesem Knoten definierten Unterbaums konsumiert werden kann, bzw. ∞, falls dies nicht möglich ist. Gesteuert durch dieses Attribut können nun beim Baumdurchlauf Unterbäume eines Knotens in der Reihenfolge aufsteigender `min_consume_level`-Werte abgearbeitet werden, wodurch das nachträgliche Löschen von Aktionslisten weitgehend vermieden werden kann[6]. Obwohl die Attributierung bei jeder Änderung des Baums aktualisiert werden muß, wird in Summe Aufwand eingespart, da Durchläufe durch den Filterbaum erheblich häufiger sind als Änderungen. Die resultierenden Algorithmen sind in Abb. 5.5 dargestellt, Abb. 5.7 zeigt den Ablauf für das Beispiel in Abb 5.6, wenn angenommen wird, daß alle Filter außer h triggern[7].

5.3.3.3 Indirekte Interaktionen und Konsistenzprobleme

Aufbauend auf den skizzierten Benachrichtigungsdiensten des Objektmodells können nun unmittelbar indirekte Interaktionen realisiert und Konsistenzprobleme gelöst werden. Dazu ist die Monitor-Schnittstelle des OCM lediglich noch um einen Ereignisdienst zu erweitern, der auf der `notify`-Methode aufsetzt und somit jede vom Monitorsystem vorgenommene Änderung einer Zustandskomponente überwachen kann. Diesen Dienst sollten die Einzelwerkzeuge bereits von sich aus nutzen, um Konsistenzprobleme bei Lesezugriffen mit einer nichtverschwindenden Dauer von vorneherein zu vermeiden.

Falls ein Werkzeug dies nicht tut, kann und muß das Rahmenwerkzeug einer Werkzeugumgebung entsprechende Ereignis/Aktions-Relationen definieren, wobei die verwendete Aktion zum Aufruf einer Funktion innerhalb des Werkzeugs führt, die den neuen Wert der Komponente liest und bearbeitet. Dabei ist zu beachten, daß OMIS eine synchrone Bearbeitung von Aktionen definiert, d.h. der Auslöser eines Ereignisses wird blockiert, bis die Aktionen abgearbeitet sind. Im vorliegenden Fall würde daher die Aktion, die die Modifikation der überwachten Komponente durchführt, solange blockiert, bis die vom Rahmenwerkzeug definierte Aktion abgeschlossen ist. Dieses Verhalten kann in Einzelfällen notwendig sein, andererseits aber auch zu unakzeptablen Leistungseinbußen führen. OMIS sieht jedoch bereits einen Mechanismus vor, der mit Hilfe benutzerdefinierter Ereignisse auch eine asynchrone, nichtblockierende Abarbeitung von Aktionen ermöglicht [150]. Dadurch kann unterschiedlichen Anforderungen in bezug auf Konsistenz und Leistung Rechnung getragen werden.

5.3.3.4 Transparenzprobleme

Auch die Behandlung von Transparenzproblemen kann über das Ereignis/Aktions-Schema des OCM erfolgen, wenn ein Ereignisdienst vorgesehen wird, der sich auf die `catch`-Methode des vorgestellten Objektmodells abstützt. Eine zugeordnete Aktion übernimmt dann die Aufgabe der Rückruffunktion, modifiziert also das Ergebnis des abgefangenen Lesezugriffs. Wieder

[6] Eine vollständige Vermeidung ist nur möglich, wenn zunächst in einem Baumdurchlauf die Filter ausgewertet und erst anschließend in einem weiteren Durchlauf die Aktionslisten bearbeitet werden. Der Aufwand hierfür wäre letztendlich größer als für das skizzierte Vorgehen.

[7] Diese Annahme dient nur dazu, den Ablauf anhand eines kleinen Filterbaums verdeutlichen zu können. In der Praxis triggert nur ein sehr kleiner Bruchteil der Filter.

```
void attribute(Node n)
{
   for each s ∈ n.children
      attribute(s)
   if (n.filter_type == normal)
      n.min_consume_level =
         MIN { s.min_consume_level | s ∈ n.children }
   if (n.filter_type == generating)
      n.min_consume_level = ∞
   if (n.filter_type == consuming)
      n.min_consume_level =
         MIN ( { s.min_consume_level | s ∈ n.children }
               ∪ { n.level } )
}

int process_event(set of Node S)
{
   int consume_level = ∞
   for each n ∈ S ordered by n.min_consume_level {
      consume_level = MIN ( consume_level,
                            process_filter(n, consume_level))
   return consume_level
}

int process_filter(Node f, int ignore_level)
{
   int consume_level = ∞
   if (f.level <= ignore_level) && (der zu f gehörige Filter triggert) {
      if (f.filter_type == consuming) {
         Entferne alle Aktionslisten aus den Warteschlangen, die seit der
            Bearbeitung des letzten generierenden Filters von einem Filter mit
            Beobachtungsebene > f.level angefügt wurden
         consume_level = f.level
      }
      level = process_events(f.children)
      Füge alle zu f gehörigen Aktionslisten an die ihrer Priorität entsprechene
         Warteschlange an
      if (f.filter_type != generating)
         consume_level = MIN ( consume_level, level )
      }
   }
   return consume_level
}
```

Abbildung 5.5: Algorithmen für Attributierung und Durchlauf durch den Ereignis-Filterbaum

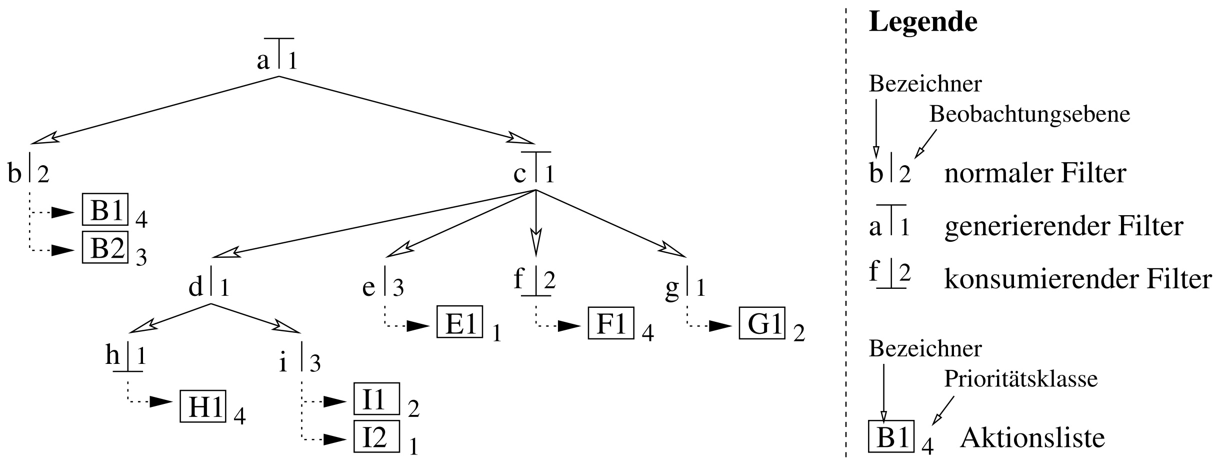


Abbildung 5.6: Beispiel für einen Filterbaum mit konsumierenden Filtern

```
process_events( {a} )
   process_filter( a, ∞ )
      process_events( {b,c} )
         process_filter( b, ∞ )
            Warteschlange für Priorität 3 = [B2]
            Warteschlange für Priorität 4 = [B1]
         process_filter( c, ∞ )
            process_events( {d,e,f,g} )
               process_filter( d, ∞ )
                  process_events( {h,i} )
                     process_filter( h, ∞ )
                        Filter triggert nicht
                     process_filter( i, ∞ )
                        Warteschlange für Priorität 1 = [I2]
                        Warteschlange für Priorität 2 = [I1]
               process_filter( f, ∞ )
                  Konsumiert Ereignis auf Ebebe 2
                  Warteschlange für Priorität 1 = []
                  Warteschlange für Priorität 2 = []
                  Warteschlange für Priorität 4 = [B1,F1]
               process_filter( e, 2 )
                  wird ignoriert
               process_filter( g, 2 )
                  Warteschlange für Priorität 2 = [G1]

Resultierende Warteschlange = [G1,B2,B1,F1]
```

Abbildung 5.7: Ablauf des Algorithmus aus Abb. 5.5 für den Filterbaum aus Abb. 5.6

sollte das Abfangen im Idealfall bereits von den einzelnen Werkzeugen selbst initiiert werden, wenn sie transparent zu haltende Schreibzugriffe durchführen. Es kann jedoch falls nötig auch wieder durch das Rahmenwerkzeug veranlaßt werden. Da in diesem Fall eine synchrone Aktionsabarbeitung zwingend notwendig ist, sollte die notwendige Aktion vollständig in einer Werkzeug-Erweiterung des OCM definiert sein, so daß sie lokal ohne Kommunikation ausgeführt werden kann. Sollte dies immer noch zu einem Leistungsengpaß führen, könnte durch eine spezielle Aktion einer Werkzeug-Erweiterung die `catch`-Methode auch direkt mit einer geeigneten Rückruffunktion aufgerufen werden.

5.3.3.5 Schreib/Schreib-Konflikte

Modifizieren zwei Werkzeuge dasselbe Objekt, kann nicht in allgemeiner Art und Weise entschieden werden, was das Ergebnis sein soll. Schreib/Schreib-Konflikte stellen daher echte Konflikte zwischen Werkzeugen dar, die nicht durch die Werkzeug-Infrastruktur behandelt werden können. Die beschriebenen Algorithmen zum Abfangen von Lesezugriffen und zur Ereigniserkennung setzen vielmehr sogar die Abwesenheit solcher Konflikte voraus, da ein Lesezugriff nur durch ein einziges Werkzeug modifiziert bzw. ein Ereignis nur von einem einzigen Filter konsumiert werden kann.

Bei der Reihenfolgesteuerung von Aktionslisten kann allerdings ein Spezialfall behandelt werden, bei dem der Wert einer Zustandskomponente von einem Werkzeug verändert und anschließend von einem anderen Werkzeug gelesen und nochmals verändert wird. In dem Beispielszenario aus § 3 tritt ein derartiger Fall bei dem Parameter `tid` des Bibliotheksaufrufs `pvm_recv` auf, der zunächst durch `codex` und dann durch `cocheck` verändert wird. Das Schema kann jedoch nur maximal zwei konfligierende Werkzeuge berücksichtigen.

5.3.4 Eine abstrakte Sicht der Infrastruktur

Wir wollen zum Abschluß dieses Kapitels einen Schritt zurücktreten und die skizzierte Infrastruktur noch einmal auf einer sehr abstrakten Ebene betrachten. Die Art und Weise, auf die wir direkte Interaktionen zwischen Werkzeugen realisieren konnten, hat bereits angedeutet, daß diese Infrastruktur eine sehr enge Verwandtschaft zu objektorientierter Middleware wie etwa CORBA besitzt. Im folgenden sollen die Gemeinsamkeiten und auch die Unterschiede vergleichend herausgestellt werden.

5.3.4.1 Gemeinsamkeiten mit objektorientierter Middleware

In beiden Fällen werden verteilte Objektumgebungen unterstützt, d.h. Systeme, die als beliebig verteilte Mengen von Objekten strukturiert sind, für die jeweils ein Satz von Methoden bereitgestellt ist. Diese Methoden können von anderen Objekten aufgerufen werden, unabhängig von der jeweiligen Lokalisierung dieser Objekte im System.

Um neben diesem Koordinationsmodell der festen Verdrahtung (siehe § 2.2.3.2) auch andere realisieren zu können, können Objekte neben Methoden auch Ereignisse definieren, die von der Middleware an andere Objekte weitergemeldet werden. Während CORBA hier nur einen relativ einfachen Dienst zur Verfügung stellt, der implizite Aufrufe ermöglicht, unterstützt neuere Middleware, insbesondere TIL [192], auch das Mediator- bzw. Interaktor-Modell, das es gestattet, in Form expliziter Regeln festzulegen, welche Methoden anderer Objekte beim Eintreten eines Ereignisses ausgeführt werden sollen. Die in diesem Kapitel entworfene Infrastruktur unterstützt ebenso alle genannten Koordinationsformen.

Neben der Weiterleitung von Methodenaufrufen bzw. Ereignissen an die betroffenen Objekte wird in beiden Fällen auch die Erzeugung neuer Objekte kontrolliert und ein globaler Namensdienst für diese zur Verfügung gestellt. Schließlich kann das System zur Laufzeit jeweils nicht nur um neue Objekte, sondern auch um neue Objektklassen (Schnittstellen) erweitert werden.

5.3.4.2 Unterschiede zu objektorientierter Middleware

Trotz dieser deutlichen Gemeinsamkeiten zeigt unsere Werkzeug-Infrastruktur im Vergleich mit üblicher objektorientierter Middleware drei wesentliche Unterschiede, die ausnahmslos Erweiterungen darstellen:

1. Bei üblicher Middleware sind die Methoden immer durch die Objekte selbst realisiert. Ereignisse werden von den Objekten explizit an die Middleware gemeldet. Bei der skizzierten Werkzeug-Infrastruktur ist dies jedoch eher die Ausnahme. Methoden werden hier im allgemeinen durch die Infrastruktur realisiert, z.B. das Anhalten eines Prozesses. In gleicher Weise werden Ereignisse von der Infrastruktur ohne Zutun des Objekts überwacht. Die Realisierung einer Methode im Objekt selbst ist jedoch trotzdem möglich durch eine Aktion, die im Kontext dieses Objekts ausgeführt wird. Auch das explizite Melden von Ereignissen ist realisierbar, indem das Objekt eine vom Monitorsystem instrumentierte, ansonsten aber leere Funktion aufruft.

   Dieser Unterschied ist sehr leicht begründbar: Im Fall der objektorientierten Middleware sind die Objekte speziell dafür entwickelt, in dieser Umgebung mit anderen Objekten zusammenzuarbeiten. Im Fall einer Monitorinfrastruktur sind weder im Zielprogramm Vorkehrungen getroffen, um mit Werkzeugen zu interagieren, noch in den Werkzeugen, um untereinander zu kommunizieren. Die notwendigen Methoden und Ereignisse müssen daher entweder vom Monitorsystem bereitgestellt oder von diesem nachträglich in die Objekte integriert werden.

2. Dieses nachträgliche Bereitstellen von Methoden und Ereignissen für Objekte bedingt, daß die Schnittstelle eines existierenden Objekts im Gegensatz zu üblicher Middleware nicht fest ist, sondern dynamisch erweitert werden kann.

3. Schließlich muß eine Monitorinfrastruktur reflexiv sein, d.h. Objekte müssen Zugriffe der Infrastruktur auf (andere) Objekte überwachen können. Auch dies ist eng verbunden mit der unter 1. genannten Begründung. Da die Objekte sozusagen nicht wissen, daß sie ein Teil einer verteilten Objektumgebung sind, kann ein Objekt bei einer Zustandsänderung die betroffenen Klienten (d.h. Werkzeuge) nicht selbst benachrichtigen, ebensowenig kann es für verschiedene Klienten unabhängige Sichten realisieren. Diese Funktionen müssen daher durch die Infrastruktur bereitgestellt werden.

Ein weiterer, eher implementierungstechnischer Unterschied sei angemerkt: Um die Skalierbarkeit zu erhöhen, erlaubt das Monitorsystem Methodenaufrufe nicht nur für einzelne Objekte, sondern auch für Objektmengen. Diese können dabei explizit durch Aufzählung oder auch implizit, z.B. durch die Tokenkonvertierung in OMIS, gegeben sein.

Zusammenfassend kann also festgestellt werden, daß die wesentlichen konzeptuellen Unterschiede ausnahmslos dadurch bedingt sind, daß bei middleware-basierten Systemen die Interaktion der Objekte bereits bei deren Entwurf berücksichtigt wird, während dies im Kontext des Monitoring nicht der Fall ist. Anwendungen werden ohne Berücksichtigung einer späteren Werkzeugnutzung entwickelt, Werkzeuge bisher nur für die exklusive Verwendung konzipiert. Bei Middleware-Umgebungen integrieren sich die Objekte daher freiwillig in das verteilte Objektsystem, während ihnen dies bei einer Monitor-Umgebung aufgezwungen wird. Die Motivation hierfür ist, daß weder der Code des Zielprogramms modifiziert werden soll, um mit einem Werkzeug zusammenarbeiten zu können, noch der eines Werkzeugs, um mit anderen Werkzeugen zu kooperieren. Die Monitor-Umgebung unterstützt also noch mehr als übliche Middleware die Realisierung einer Kooperation zwischen *Legacy*-Systemen.

Kapitel 6

Realisierung der Monitoring-Middleware OCM

Die folgenden Abschnitte beschreiben die Implementierung einer Monitoring-Middleware für interoperable Werkzeuge auf Basis der Schnittstellenspezifikation OMIS und des Monitorsystems OCM. Dabei wird die technische Umsetzung der Konzepte aus § 5.3 insoweit beschrieben, als eine detailliertere Diskussion erforderlich erscheint. In erster Linie werden jedoch diejenigen Realisierungsaspekte des OCM erläutert, die bisher nicht im Detail ausgeführt wurden.

6.1 Die Schnittstellendefinition OMIS 2.0

Da die Grundideen von OMIS bereits in § 3.2.1.1 vorgestellt wurden, sollen hier nur noch einmal die wichtigsten Eigenschaften der Monitorschnittstelle zusammengefaßt werden:

1. Die Schnittstelle ist **objektbasiert**, das überwachte System wird also als Menge von Objekten strukturiert, die durch abstrakte Bezeichner (Token) referenziert werden. Für jede Objektklasse stellt das Monitorsystem **Dienste** (Methoden) bereit, die **Information** über Objekte dieser Klasse liefern, **Manipulationen** an ihnen vornehmen oder **Ereignisse** überwachen.

2. Gemäß dem **Ereignis/Aktions-Paradigma** können Ereignisdienste mit Listen von Informations- und Manipulationsdiensten (Aktionen) zu bedingten Anfragen kombiniert werden, die das Monitorsystem dann autonom überwacht und ausführt. Ein **Parameterübergabemechanismus** erlaubt dabei das Weiterleiten von Ereignisinformation an die Aktionen (Ereigniskontextparameter). Als Hilfe bei der Anpassung eventuell nicht zusammenpassender Schnittstellen erfolgt eine automatische Konvertierung von Objekttoken. Weitere Mechanismen, um Ereigniskontextparameter vor der Übergabe an Aktionen zu transformieren, sind nicht vorgesehen, um die Schnittstelle nicht aufzublähen und dadurch ihre effiziente Implementierbarkeit zu gefährden.

3. Stattdessen ist die Schnittstelle auf eine modulare **Erweiterbarkeit** hin ausgerichtet. Neue Ereignisse und Aktionen können ohne Wissen über die interne Implementierung des Monitorsystems zu diesem hinzugefügt werden. Sie können sich dabei auf vorhandene Dienste abstützen und so, falls erforderlich, auch Schnittstellenanpassungen vornehmen.

4. Die OMIS-Schnittstelle ist für eine **simultane Nutzung durch mehrere Werkzeuge** vorgesehen. Jedes Werkzeug muß deshalb die Menge der zu überwachenden Objekte selbst definieren; verschiedene Werkzeuge können also eine unterschiedliche Sicht des Zielsystems besitzen.

5. Um eine gute **Skalierbarkeit** sicherzustellen, sind Dienste nicht nur auf Einzelobjekte, sondern auf beliebige Objektmengen anwendbar. Objektmengen können explizit durch Aufzählung oder implizit durch Konvertierung von Objekttoken gegeben sein. Im letzteren Fall kann z.B. ein Knotentoken stellvertretend für die Menge aller (durch das jeweilige Werkzeug) überwachter Prozesse auf diesem Knoten verwendet werden. Derartige Objektmengen sind dynamisch, werden bei bedingten Anfragen also zum Ausführungszeitpunkt, nicht zum Definitionszeitpunkt ausgewertet.

6. Schließlich bietet OMIS eine uneingeschränkte **Ortstransparenz**. Die von einer Anfrage betroffenen Objekte können also völlig beliebig im System verteilt sein, ohne daß das Werkzeug diese Verteilung in irgendeiner Weise berücksichtigen muß.

Eine genauere Beschreibung der Entwurfskonzepte von OMIS findet sich in der Habilitationsschrift von T. Ludwig [142], die exakte Schnittstellenspezifikation wurde in [150] veröffentlicht. Übersichten über die aktuelle sowie frühere Versionen von OMIS finden sich in [144] und [149, 136, 148].

6.2 Aufbau des OMIS-konformen Monitorsystems OCM

Die Grobstruktur des OMIS-konformen Monitorsystems OCM wurde bereits in § 3.2.1.2 beschrieben. An dieser Stelle werden daher nur noch Einzelaspekte der Implementierung diskutiert, die im Zusammenhang mit der Unterstützung interoperierender Werkzeuge stehen. Dies betrifft insbesondere:

1. die transparente Verteilung von Monitoranfragen, die letztendlich die direkten und indirekten Interaktionen zwischen Werkzeugen ermöglicht,

2. die dynamische Erweiterbarkeit des Monitorsystems, die eine Voraussetzung für die Ereignisüberwachung und die Aktionsausführung in Werkzeugen ist,

3. die Maßnahmen, die eine gemeinsame Nutzung der Monitorschnittstelle durch mehrere Werkzeuge unterstützen,

4. die Überwachung und Manipulation von Anwendungsprozessen, Werkzeugen und von Zugriffen des Monitorsystems selbst.

Weitere Details über die Implementierung des OCM für Netze von Arbeitsplatzrechnern unter PVM sind an anderer Stelle veröffentlicht [147, 146, 270, 269]. Informationen zu Portierungen des Monitorsystems auf andere Zielplattformen finden sich in [256, 22, 23, 26]. Der OCM wurde in mehreren Projekten[1] in verschiedensten Umgebungen erfolgreich eingesetzt, was durch zahlreiche Veröffentlichungen dokumentiert wird [144, 250, 249, 28, 24, 25, 18, 20, 21].

[1] U.a. gefördert durch die DFG (SFB 342) und das BMBF (WTZ-Projekte UNG-053-96 und POL 98/011).

6.2.1 Verteilung von Anfragen

Wie erwähnt, spezifiziert OMIS 2.0 eine vollständige Ortstransparenz aller Monitoranfragen. In Verbindung mit der Möglichkeit, Dienste auf (verteilten) Objektmengen anzuwenden, führt dies zu einer sehr mächtigen, komfortablen Schnittstelle. So kann eine einfache bedingte Anfrage verwendet werden, um beim Eintreten eines Ereignisses im Zielsystem oder einem Werkzeug eine Aktion in einem anderen Werkzeug auszuführen, selbst wenn sich dieses auf einem anderen Rechnerknoten befindet. Die korrekte und effiziente Ausführung derartiger Anfragen wird dabei in transparenter Weise durch den OCM realisiert.

6.2.1.1 Problemstellung

Der OCM muß beim Eintreffen einer Anfrage entscheiden, wie deren Ausführung auf die einzelnen Komponenten des Monitorsystems zu verteilen ist, und welche Kommunikation und Synchronisation dabei erforderlich ist. Als Komponenten sind hier nicht nur die NDU und die lokalen Monitorprozesse zu berücksichtigen, sondern auch die Monitorbibliotheken in den überwachten Prozessen (vgl. Abb. 3.8). Betrachtet man eine allgemeine Anfrage der Form

$$\begin{array}{lll} \mathit{Ereignis}\,\texttt{(}\mathit{Objektmenge}_0\texttt{, ...)} & \texttt{:} & \mathit{Aktion}_2\,\texttt{(}\mathit{Objektmenge}_1\texttt{, ...)} \\ & & \mathit{Aktion}_2\,\texttt{(}\mathit{Objektmenge}_2\texttt{, ...)} \\ & & \texttt{...} \end{array}$$

so ergibt sich die folgende Situation:

1. Die Objektmengen können drei verschiedene Arten von Objekten beinhalten. Lokale Objekte sind auf genau einem Knoten des Zielsystems beheimatet. Zu dieser Klasse gehören z.B. Knoten-, Prozeß- und Thread-Objekte. Daneben existieren auch globale Objekte, die im OCM in unterschiedlicher Weise implementiert sein können. Im einfachsten Fall sind sie zentral, d.h. in der NDU lokalisiert. Ein Beispiel ist das Objekt, das die virtuelle Maschine von PVM repräsentiert. Um eine effiziente Ausführung der zugehörigen Dienste zu gewährleisten, können globale Objekte auch verteilt realisiert sein. Beispiele derartiger Objekte sind globale Ereignispuffer [249, 250] sowie Integratoren und Ereigniszähler [24]. Bei letzteren existiert auf jedem Knoten ein lokaler Teilzähler, der durch entsprechende lokale Dienste manipuliert werden kann. Das Auslesen des Zählers ist ein globaler Dienst, der die Summe aller lokalen Teilzähler ermittelt.

2. Das Ereignis kann für jedes spezifizierte Objekt entweder durch den lokalen Monitorprozeß oder durch die Monitorbibliothek innerhalb eines Zielprozesses erkannt werden.

3. Schließlich können die Aktionen im allgemeinen nur in bestimmten Komponenten des Monitorsystems ausgeführt werden. Diese Randbedingung kann für jede Aktion durch zwei Attribute spezifiziert werden: Zum einen durch den Ausführungskontext, der die Art der Komponente festlegt, also NDU, lokaler Monitorprozeß oder Monitorbibliothek innerhalb eines Zielprozesses. Ein zweites Attribut legt fest, ob die Aktion in jeder derartigen Komponente ausgeführt werden kann, oder ob sie auf einen bestimmten Knoten oder Prozeß lokalisiert werden muß. Nicht alle möglichen Kombinationen dieser Attribute sind dabei sinnvoll und erlaubt. Im OCM werden derzeit fünf Klassen unterstützt:

   (a) **Kontext = NDU**
   Diese Aktionen werden zentral in der NDU ausgeführt, z.B. Aktionen, die Information über die virtuelle Maschine von PVM liefern.

(b) **Kontext = NDU oder Monitorprozeß, keine Lokalisierung**
Derartige Aktionen können von allen Monitorprozessen, einschließlich der NDU, mit demselben Ergebnis ausgeführt werden. Ein einfaches Beispiel ist der Dienst `print`, der lediglich seine Argumente als Ergebnis zurückliefert.

(c) **Kontext = NDU, Monitorprozeß oder -bibliothek, keine Lokalisierung**
Aktionen dieser Klasse können in jeder Komponente des OCM abgearbeitet werden. Dazu zählen z.B. Aktionen zur Manipulation von Ereigniszählern.

(d) **Kontext = Monitorprozeß, Lokalisierung auf Knoten**
In diese Kategorie fallen die meisten von OMIS definierten Dienste, die Systemobjekte mit Hilfe des Betriebssystems beobachten bzw. manipulieren und daher nur vom jeweils lokalen Monitorprozeß erbracht werden können.

(e) **Kontext = Monitorbibliothek, Lokalisierung auf Prozeß**
Diese Dienste müssen von der Monitorbibliothek eines bestimmten überwachten Prozesses ausgeführt werden. Sie dienen in der Regel zum Aufruf lokaler Aktionen innerhalb von Werkzeugen.

Es ist leicht zu sehen, daß die Eingruppierung eines Dienstes in diese Klassen nicht völlig unabhängig von der Art der Objekte ist, auf denen der Dienst arbeitet. So muß ein Dienst für ein zentrales Objekt zwangsläufig in der NDU ausgeführt werden.

Die Lokalisierung eines Dienstes wird normalerweise durch die konkreten Objekte festgelegt, für die der Dienst angefordert wurde. Bei der Implementierung kann jedoch auch eine dienstspezifische Lokalisierungsfunktion definiert werden, die auch andere Argumente des Dienstaufrufs berücksichtigt.

Die Information, zu welcher Klasse ein Objekt, ein Ereignis oder eine Aktion gehört, muß bei der Implementierung in einer Schnittstellenbeschreibungsdatei spezifiziert werden (siehe § 6.2.2). Sie liegt daher zur Laufzeit in allen Komponenten des Monitorsystems vor.

6.2.1.2 Vorgehensweise

Entsprechend der Struktur des OCM erfolgt die Verteilung von Anfragen hierarchisch, wobei zunächst die Knotenebene betrachtet wird und dann die Prozeßebene.

Verteilung auf Knoten. Die Verteilung einer Monitoranfrage auf Knotenebene erfolgt durch die NDU unmittelbar nach dem Empfang und der syntaktischen Analyse der Anfrage. Dazu wird zunächst für jeden Dienst in der Anfrage die Menge der Knoten ermittelt, auf denen der Dienst ausgeführt werden muß. Diese Knotenmenge ist bestimmt durch die Klasse und die Lokalisierung aller Objekte, auf die der Dienst angewandt werden soll, sowie durch die Attribute des Dienstes selbst.[2] Als Nebeneffekt ergibt sich ein angepaßter Dienstaufruf für jeden Knoten, der nur noch lokal zugreifbare Objekte referenziert.

Im Fall einer unbedingten Anfrage werden anschließend an jeden betroffenen Knoten die entsprechenden, angepaßten Dienstaufrufe verschickt (vgl. Abb. 6.1). Diese werden dann zueinander nebenläufig abgearbeitet.

[2]Bei Diensten, die keine Lokalisierung erfordern, entsteht ein Freiheitsgrad, der für Optimierungen genutzt wird.

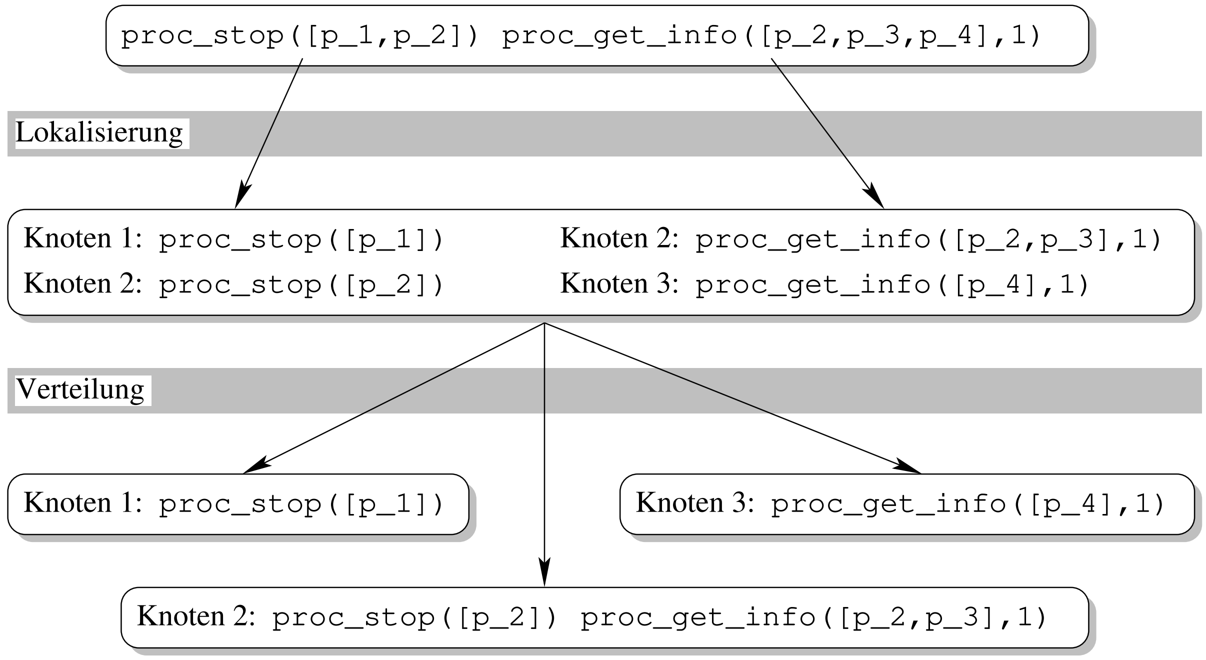


Abbildung 6.1: Beispiel zur Verteilung einer unbedingten Anfrage

OMIS 2.0 erlaubt auch eine Synchronisation durch Einfügen von Barrieren in Aktionslisten [150], die syntaktisch durch ein Semikolon dargestellt werden. Eine derartige Aktionsliste der Form `a1;a2` wird ausgeführt, indem durch die NDU zunächst `a1` verteilt wird. Durch Setzen eines Statusbits in den entstehenden lokalen Anfragen wird dabei die Rücksendung auch eines leeren Resultats erzwungen. Nach dem Eintreffen aller Resultate wird dann `a2` von der NDU bearbeitet.

Kann die NDU bei einer bedingten Anfrage „beweisen“, daß ihre Aktionen unter allen Umständen lokal auf dem Knoten bearbeitet werden können, auf dem das Ereignis jeweils erkannt wird, so ist eine vollständige Verteilung der Anfrage bereits zum Definitionszeitpunkt möglich. Die betroffene Knotenmenge bestimmt sich in diesem Fall allein aus dem Ereignisdienst, wobei jeder Knoten die angepaßte Ereignisdefinition mit einer unveränderten Kopie der Aktionsliste erhält.

Falls Aktionen auch nicht-lokal sein können, erfolgt die Verteilung der Aktionsliste dagegen erst zum Ausführungszeitpunkt. Die ursprüngliche Anfrage wird dazu in

Ereignis `(`*Objektmenge*$_0$`, ...) : gen_suspend($doneinfo) ct_send(`*tag*`)`

abgeändert und auf die einzelnen Knoten verteilt. Nach dem Erkennen eines Ereignisses sorgt die Aktion `gen_suspend` zunächst dafür, daß das auslösende Objekt auch nach der Ausführung der (lokalen) Aktionsliste solange suspendiert bleibt, bis alle ursprünglichen Aktionen bearbeitet wurden. Der spezielle Ereigniskontextparameter `doneinfo` spezifiziert dabei das betroffene Objekt und enthält zusätzliche Verwaltungsdaten. Anschließend wird eine Nachricht mit der eindeutig gewählten Kennung *tag* an die NDU versandt. Diese Nachricht enthält auch alle zu dem Ereignis gehörigen Ereigniskontextparameter, die an die Aktionen weitergegeben werden müssen.

Trifft eine derartige Nachricht bei der NDU ein, so startet sie die Bearbeitung der ursprünglichen Aktionsliste, die unter dem Schlüssel *tag* gespeichert wurde. An diese wird dazu zunächst eine Barriere und die Aktion `gen_resume` angefügt, die die Suspendierung des auslösenden Objekts wieder aufhebt, wenn alle Knoten die Bearbeitung der Aktionsliste abgeschlossen haben. In der resultierenden Aktionsliste

$Aktion_1$ `(`$Objektmenge_1$`, ...)` $Aktion_2$ `(`$Objektmenge_2$`, ...) ... ;`
`gen_resume($node, $doneinfo)`

werden dann alle Ereigniskontextparameter durch die in der Nachricht enthaltenen Werte ersetzt. Anschließend wird sie wie eine unbedingte Anfrage verteilt und ausgeführt.

In [270] und [142] ist eine frühere Version des Verteilungsverfahrens für bedingte Anfragen beschrieben, bei der die Aufteilung bereits vollständig zum Definitionszeitpunkt erfolgt. Jenes Verfahren besitzt zwar keinen geringeren Kommunikationsaufwand, da aufgrund der Konsistenzanforderungen nach wie vor Nachrichten über die NDU verschickt werden müssen, spart jedoch die wiederholte Analyse und Verteilung der Aktionsliste bei jedem Auftreten eines Ereignisses ein. Es setzt jedoch voraus, daß die Verteilung der Aktionsliste bereits bei der Definition der Anfrage bestimmt werden kann. Dies muß aber nicht zwangsläufig der Fall sein, da die Werte von Ereigniskontextparametern zum Definitionszeitpunkt noch nicht bekannt sind. Ein Beispiel für eine derartige Anfrage ist:

```
thread_creates_proc([t_1]): proc_attach([$new_proc])
```

Wenn Thread `t_1` einen neuen Prozeß erzeugt, soll dieser überwacht werden. Da der Prozeß auf einem entfernten Knoten erzeugt werden kann, ist die Anfrage nicht lokal. Andererseits ist bei der Definition der Anfrage der Knoten noch nicht bekannt, auf dem der Dienst `proc_attach` auszuführen ist. Als Alternative zu einer Verteilung zum Ausführungszeitpunkt könnten derartige Anfragen bei der Definition zunächst an alle Knoten versandt werden. Beim Eintreten des Ereignisses würden dann nur die aktuell betroffenen Knoten benachrichtigt. Der notwendige Analyseaufwand ist damit aber genauso hoch wie bei dem hier gewählten Verfahren, das deutlich einfacher zu realisieren ist.

Verteilung auf Prozesse. Anders als auf Knotenebene ist hier keine explizite Verteilung einer Anfrage notwendig, da alle Monitorkomponenten auf einem Knoten über einen gemeinsamen Speicher verfügen und somit direkt auf die Anfrage zugreifen können. Bei der Ausführung von Aktionslisten wird jedoch ein Kontextwechsel erforderlich, falls eine Aktion nicht im aktuellen Kontext ausführbar ist.

Die Bearbeitung einer Anfrage startet zunächst immer im Monitorprozeß, der sie als Nachricht von der NDU empfängt. Bei einer unbedingten Anfrage werden die Aktionen, die im Monitorprozeß ausführbar sind, unmittelbar bearbeitet. Wird eine Aktion angetroffen, die in einem spezifischen Anwendungsprozeß auszuführen ist, wird diese durch einen *Inferior RPC* in diesem Prozeß gestartet. Die Parameterübergabe erfolgt dabei über das gemeinsame Speichersegment. Während der Monitorprozeß auf die Beendigung des *Inferior RPC* wartet, muß weiterhin eine Ereigniserkennung und Aktionsausführung für *andere* Prozesse erfolgen, um die Reaktionszeit des Monitors nicht unzulässig zu verlängern. Zudem könnten sonst Verklemmungen entstehen, falls die Aktion in einem einem Werkzeug auszuführen ist und sie wiederum Dienste des OCM anfordert. Eine Realisierung des Monitorprozesses mit mehreren Threads könnte an dieser Stelle die Programmierung vereinfachen, wurde jedoch mit Rücksicht auf eine gute Portierbarkeit bisher nicht durchgeführt.

Beim Eintreffen einer bedingten Anfrage wird zunächst durch den Monitorprozeß die erforderliche Instrumentierung des Zielsystems durchgeführt und alle notwendige Information in den Filterbaum eingetragen. Dieser befindet sich im gemeinsamen Speichersegment, ist also von allen Monitorkomponenten aus zugreifbar. Wird im weiteren Verlauf das Ereignis durch den Monitorprozeß erkannt, so erfolgt die Ausführung der zugehörigen Aktionsliste wie oben beschrieben. Erkennt jedoch die Monitorbibliothek das Ereignis in einem Anwendungsprozeß, so analysiert sie zunächst, ob alle für dieses Ereignis definierten Aktionen im lokalen Prozeß ausführbar sind. Falls ja, werden sie unmittelbar bearbeitet. Anderenfalls erfolgt ein expliziter Kontextwechsel zum Monitorprozeß. Dazu wird zunächst die Information über das aktuelle Ereignis im gemeinsamen Speichersegment abgelegt. Anschließend sendet die Monitorbibliothek ein Signal an den eigenen Prozeß, das diesen stoppt und gleichzeitig vom Monitorprozeß als Ereignis erkannt und bearbeitet wird.

Durch diese Vorgehensweise werden mehrfache Kontextwechsel weitgehend vermieden. Nach dem Umschalten zum Monitorprozeß kann dieser im allgemeinen alle Aktionen lokal ausführen, da Aktionen, die ausschließlich in einem Anwendungsprozeß ausgeführt werden können und daher zu einem erneuten Kontextwechsel durch einen *Inferior RPC* führen, nur relativ selten vorkommen.

Dynamische Objektmengen. Eine Schwierigkeit bei der Verteilung von bedingten Anfragen stellen implizit definierte Objektmengen dar. Da sie sich dynamisch ändern können, ist die resultierende Menge zum Definitionszeitpunkt der Anfrage noch nicht bekannt. Bei einer Anfrage wie z.B.

```
thread_reached_addr([a_], 0x1234): ...
```

kann daher nicht endgültig entschieden werden, auf welchen Knoten und für welche Prozesse eine Instrumentierung und Überwachung notwendig wird. Die im OCM gewählte Lösung besteht darin, die Objektmenge zunächst zum Definitionszeitpunkt zu bestimmen und die Anfrage gemäß dieser zu verteilen und zu bearbeiten. Gleichzeitig werden Aktionen definiert, die bei einer Änderung der Objektmenge auszuführen sind: Kommt durch die Änderung ein neuer Knoten hinzu, muß die NDU die Anfrage nun auch noch an diesen Knoten senden. Analog muß der lokale Monitorprozeß bei einem neu hinzukommenden bzw. wegfallenden Objekt die erforderliche Instrumentierung für das Ereignis einfügen bzw. löschen.

6.2.2 Dynamische Erweiterbarkeit

In den vorangegangenen Kapiteln wurden zwei Gründe identifiziert, die eine dynamische Erweiterbarkeit von Monitorsystemen um neue Objektklassen und Dienste erforderlich machen: Zum einen können neue Werkzeuge es erfordern, daß neue Aspekte des Zielsystems überwacht werden müssen. Zum anderen setzt die skizzierte Realisierung von Interaktionen zwischen Werkzeugen voraus, daß die Monitorschnittstelle um Dienste erweitert werden kann, die in den jeweiligen Werkzeugen selbst ausgeführt werden. Eine Erweiterbarkeit zur Laufzeit hat in beiden Fällen den Vorteil, daß keine Neuinstallation des Monitorsystems notwendig ist, um die benötigten Funktionen zu integrieren. Zudem verhindert dieses Vorgehen, daß der Monitor direkt oder indirekt von den Werkzeugen abhängig wird, die ihn nutzen. Ein Monitorsystem wie der OCM, das eine derartige *Plug-in*-Architektur realisiert, kann zudem sehr modular aufgebaut werden. Es besteht aus einem Kern, der alle Funktionen zur Verwaltung und Ausführung von Diensten enthält, sowie einer Menge dynamisch nachgeladener Erweiterungsmodule, die die eigentlichen Monitordienste bereitstellen. Im folgenden wird die Realisierung dieser Architektur im OCM detaillierter beschrieben.

6.2.2.1 Schnittstellendefinition

Jedes Erweiterungsmodul ist zunächst durch eine Schnittstellenbeschreibungsdatei (IDL-Datei, vgl. § 2.2.3.2, Seite 33) charakterisiert, die insbesondere die Namen und Parameter seiner Dienste definiert. In der IDL-Datei können zusätzlich Attribute gesetzt werden, die wichtige Eigenschaften der Objektklassen bzw. ihrer Dienste beschreiben und dadurch ihre Einbindung in den OCM steuern. Die wichtigsten Attribute spezifizieren dabei

- **für Objektklassen** die Vaterklasse in der Objekthierarchie (vgl. Abb. 3.5) und die Art der Objekte (lokal, zentral oder verteilt). Zusätzlich können im OCM auch verteilte, assoziative Felder von Objekten deklariert werden, auf die an dieser Stelle nicht näher eingegangen werden kann.

- **für Aktionen** die möglichen Ausführungskontexte (NDU, Monitorprozeß oder Anwendungsprozeß) und die erforderliche Lokalisierung. Daneben kann eine Ziel-Objektklasse für die automatische Konvertierung der übergebenen Objekte durch den OCM-Kern definiert werden. In diesem Fall muß die Aktion nur noch für Einzelobjekte dieser Klasse realisiert werden.

- **für Ereignisdienste** den Erkennungskontext, die Ziel-Objektklasse für die automatische Token-Konvertierung sowie den Instrumentierungszeitpunkt. Das letztgenannte Attribut bestimmt, ob die Instrumentierung bei kurzfristiger Deaktivierung der Ereignisüberwachung entfernt wird. Es erlaubt eine Abwägung zwischen der Beeinflussung durch die Ausführung des Instrumentierungscodes und den Kosten für dessen Einfügung bzw. Löschung.

Abb. 6.2 zeigt einige Beispiele für Definitionen innerhalb dieser IDL-Dateien.

6.2.2.2 Implementierung von Erweiterungsmodulen

Der eigentliche Programmcode eines Erweiterungsmoduls muß alle Funktionen zur Realisierung der Objektklassen, Aktionen und Ereignisdienste zur Verfügung stellen. Aus diesem Quellcode werden zwei dynamische Bibliotheken erzeugt, je eine zum Einbinden in Monitorprozesse und in Anwendungsprozesse. Die notwendige Registrierungs- und Implementierungsinformation wird durch einen IDL-Übersetzer in eine Datenstruktur transformiert, die innerhalb der Bibliothek gespeichert wird. Sie enthält unter anderem eine Tabelle für die Zuordnung von Dienstnamen zu den Funktionen, die den Dienst realisieren. Die nachfolgenden Unterabschnitte gehen näher auf die jeweils erforderlichen Funktionen ein.

Objektklassen. Für jede Objektklasse, die von einem Erweiterungsmodul neu eingeführt wird, muß dieses zunächst einen entsprechenden Konstruktor realisieren, der eine klassenspezifische Schnittstelle besitzen kann. Ebenso ist ein Destruktor zu realisieren. Die Schnittstelle der Destruktoren ist dabei klassenunabhängig, da alle Objekte in einheitlicher Weise über Token referenziert werden. Dies erlaubt es, die Freigabe eines Objekts automatisch durch den OCM-Kern zu veranlassen, sobald keine Referenz auf das Objekt mehr existiert.

Für die Durchführung der Tokenkonvertierung durch den OCM-Kern muß das Erweiterungsmodul eine Lokalisierungsfunktion bereitstellen, die zu einem Token t dieser Klasse und

```
// Prozeß-Objekte
token p_
{
   class = local;                  // lokales Objekt
   parent = n_;                    // Vorgänger in Hierarchie: Knoten
   attachable = true;              // Überwachung muß explizit angefordert werden
};

// Ereignis: Prozeßerzeugung durch einen Thread
// new_proc und new_node sind zusätzliche Ereigniskontextparameter,
// die den neuen Prozeß und dessen lokalen Knoten identifizieren
event thread_creates_proc(token* thread_list) :
                       (token new_node, token new_proc)
{
   context = monitor;              // Ereignis wird vom Monitorprozeß erkannt
   auto_convert_to = t_;           // automatische Konvertierung zu Thread-Token
   instrumentation = on_enable;    // Instrumentierung wird erst bei Aktivierung
                                   // des Ereignisses durchgeführt
};

// Aktion in VISTOP: lösche Ereignisse nach pos
action VISTOP_clear_after_pos(token* tool_list, integer pos)
{
   context = application;          // Aktion muß durch Monitorbibliothek ausge-
                                   // führt werden
   localize = process;             // Aktion nur im lokalen Prozeß ausführbar
};
```

Abbildung 6.2: Beispiele für Definitionen in IDL-Dateien des OCM

einer gegebenen Vorgängerklasse k in der Objekthierarchie ein Token t' der Klasse k liefert, derart, daß t in t' lokalisiert ist. Die Lokalisierungsfunktion kann bei Bedarf rekursiv die entsprechende Funktion ihrer unmittelbaren Vaterklasse aufrufen, so daß sie keine weitere Information über die Objekthierarchie benötigt.

Das Erweiterungsmodul kann optional eine Rückruffunktion definieren, die die erforderlichen Aktionen im Zielsystem ausführt, wenn ein Objekt durch ein Werkzeug zum überwachten System hinzugefügt wurde bzw. nicht länger überwacht wird. Schließlich kann das Modul weitere klassenspezifische Methoden realisieren, die dem Kern des OCM nicht bekannt sind. Die Erweiterung von existierenden Objektklassen um neue Komponenten wird in § 6.2.6 diskutiert.

Aktionen. Zur Realisierung einer neuen Aktion muß ein Erweiterungsmodul lediglich eine Funktion bereitstellen, die die eigentliche Aktion ausführt. Die Schnittstelle dieser Funktion ergibt sich direkt aus der IDL-Beschreibung. Ein Skelett zur Anpassung an die standardisierte Aktions-Schnittstelle innerhalb des OCM wird vom IDL-Übersetzer automatisch generiert.

Dieses Skelett kann zusätzlich auch die notwendige Konvertierung der übergebenen Objekttoken sowie die Behandlung von Objektmengen übernehmen, so daß die Aktion nur noch für Einzelobjekte der jeweiligen Klasse implementiert werden muß.

Ereignisdienste. Ein neuer Ereignisdienst erfordert dagegen die Realisierung von bis zu vier Funktionen, die die folgenden Aufgaben erfüllen:

1. Die Definitionsfunktion bestimmt denjenigen Ereignisknoten im Filterbaum, an den Aktionslisten für das spezifizierte Ereignis angehängt werden müssen (vgl. Abb. 3.6). Die Funktion kann dabei entweder einen schon existierenden Knoten zurückliefern, falls das Ereignis bereits überwacht wird, oder rekursiv die Definitionsfunktion eines Basisereignisses aufrufen. An den so erhaltenen Ereignisknoten wird dann ein Filter angebunden, der einen neu erzeugten Ereignisknoten triggert.

   Die Definitionsfunktion kann also eine Optimierung vornehmen, indem sie Überwachungen von identischen Ereignissen zusammenfaßt, die z.B. durch unterschiedliche Werkzeuge veranlaßt wurden. Der Einsparung von mehrfachen Filterauswertungen bei der Ereigniserkennung steht dabei allerdings ein durch die Zusammenfassung erhöhter Verwaltungsaufwand gegenüber, so daß jeweils im Einzelfall eine Abwägung zu treffen ist.

2. Die Filterfunktion hat die Aufgabe, beim Eintreten eines Basisereignisses dieses genauer zu analysieren und gegebenenfalls die Behandlung des spezifischeren Ereignisses zu veranlassen.

3. Beim Löschen von bedingten Anfragen wird der Ereignisbaum automatisch vom OCM-Kern aktualisiert. Ein Erweiterungsmodul kann jedoch bei Bedarf eine Rückruffunktion bereitstellen, die zusätzliche Aufräumarbeiten, z.B. die Freigabe von Hilfsdatenstrukturen, erledigt.

4. Die zentrale Rolle bei der Realisierung eines Ereignisdienstes spielt schließlich die Instrumentierungsfunktion. Sie fügt dynamische Instrumentierungen in Objekte ein, die notwendig sind, um ein Ereignis beobachten zu können, bzw. löscht diese wieder. Ist eine Instrumentierung bereits statisch in ein Objekt eingebunden, z.B. durch Verwendung vorinstrumentierter Bibliotheken, so wird sie durch diese Funktion lediglich aktiviert bzw. deaktiviert. Die Instrumentierungsfunktion wird vom OCM-Kern automatisch aufgerufen, wenn eine Änderung der Instrumentierung erforderlich werden könnte, etwa weil sich die überwachte Objektmenge geändert hat oder eine Ereignisüberwachung temporär abgeschaltet wurde. Sie muß dabei eine mehrfache, identische Instrumentierung vermeiden und ebenso sicherstellen, daß Instrumentierungen nur dann gelöscht werden, wenn sie nicht mehr zur Überwachung anderer Ereignisse erforderlich sind. Diese Aufgabe kann nicht vom OCM-Kern übernommen werden, da die Abbildung von zu überwachenden Ereignissen auf die erforderlichen Instrumentierungen implementierungsabhängig und nicht notwendigerweise injektiv ist.

6.2.2.3 Einbinden von Erweiterungsmodulen

Nachdem wie oben beschrieben die beiden Bibliotheken für ein Erweiterungsmodul erstellt wurden, können diese zur Laufzeit in den OCM eingebunden werden.

Das Einbinden in die NDU und die lokalen Monitorprozesse erfolgt dabei für die Werkzeuge transparent. Dazu wird die von OMIS definierte Namenskonvention ausgenutzt, nach der die Namen aller Dienste eines Erweiterungsmoduls mit dem Namen dieses Moduls beginnen müssen. Findet der Anfrageparser in der NDU also einen bisher unbekannten Dienst in einer Anfrage, so kann er aus dem Namenspräfix das zugehörige Erweiterungsmodul bestimmen. Die NDU ruft dann über die Systemfunktion `dlopen` den dynamischen Binder auf, um die entsprechende Bibliothek zu laden. Nach dem Laden steht der NDU die Registrierungsinformation des Erweiterungsmoduls zur Verfügung, so daß sie die Korrektheit des Dienstaufrufs (z.B. Parameteranzahl und -typen) prüfen und die Anfrage verteilen kann. Beim Empfang einer Teilanfrage geht der lokale Monitorprozeß analog vor. Es wird also ebenfalls die Bibliothek des Erweiterungsmoduls zum Prozeß hinzugebunden. Die für jeden Dienst aufzurufenden Funktionen und ihre Attribute können dann aus der Registrierungsinformation bestimmt werden.

Das Laden von Erweiterungsmodulen in Anwendungsprozesse erfolgt im Gegensatz dazu nicht automatisch, sondern nur auf Anforderung eines Werkzeugs durch einen speziellen Dienst. Durch diese Vorgehensweise kann ein Werkzeug das relativ zeitaufwendige Laden der benötigten Module zu einem Zeitpunkt veranlassen, an dem die Rückwirkung auf das Zielprogramm minimal bleibt, z.B. unmittelbar vor dessen Start. Im einzelnen führt der Dienst die folgenden Aktionen durch: Zunächst wird der Zielprozeß angehalten. Durch einen *Inferior RPC*, also letztendlich durch eine Modifikation des Befehlszählers, wird der Prozeß dann zum Aufruf des dynamischen Binders gezwungen. Nach Abschluß des Bindevorgangs muß der Monitorprozeß mit Hilfe entsprechender Betriebssystemdienste den benötigten Teil der Registrierungsinformation aus dem Adreßraum des Zielprozesses auslesen.

6.2.3 Gleichzeitige Unterstützung mehrerer Werkzeuge

Um die gleichzeitige, unabhängige Benutzung der Monitorschnittstelle durch mehrere Werkzeuge zu ermöglichen, realisiert der OCM eine Reihe von Konzepten, die zum Teil durch OMIS gefordert werden, zum Teil auch Erweiterungen dieser Spezifikation darstellen.

6.2.3.1 Serialisierbarkeit von Aktionen

Um die von OMIS geforderte Semantik von Anfragen (siehe [150], S. 56 ff) zu garantieren, müssen Aktionen auf einem Objekt serialisierbar zu anderen Aktionen auf demselben Objekt ausgeführt werden.

Im OCM wird dies für lokale und zentrale Objekte primär dadurch erreicht, daß die lokalen Monitorprozesse und die NDU Anfragen streng sequentiell abarbeiten. Allerdings können nebenläufig dazu auch Aktionen durch die Monitorbibliothek in einem Anwendungsprozeß ausgeführt werden. Die betroffenen Objekte müssen also durch Sperren gesichert werden. Als Voreinstellung wird bei Monitoranfragen dabei zunächst nur die Serialisierbarkeit für die Ausführung von Aktionen auf einzelnen Objekten sichergestellt. Verklemmungen durch diese Sperren sind somit ausgeschlossen, da ein Prozeß zu jedem beliebigen Zeitpunkt maximal eine Sperre benötigt.

Bei verteilten Objekten können Aktionen auf den verschiedenen lokalen Teilobjekten nebenläufig ausgeführt werden. Da diese Teilobjekte vollständig unabhängig voneinander sind, ist die Forderung nach Serialisierbarkeit hier automatisch erfüllt, da die Serialisierbarkeit von Aktionen auf demselben Teilobjekt gesichert ist.

6.2.3.2 Sperren globaler Aktionslisten

Muß ein Werkzeug sicherstellen, daß eine Aktionsliste A in globalem Sinne nicht-unterbrechbar abgearbeitet wird, kann es in der entsprechenden Monitoranfrage durch ein Statusbit eine spezielle Sperre anfordern. Diese Sperre stellt sicher, daß sich während der Ausführungzeit von A im gesamten Monitorsystem keine andere Aktionsliste in Ausführung befindet. Es wird also weder A durch andere Aktionen unterbrochen, noch unterbricht A die Abarbeitung schon begonnener Aktionslisten (vgl. § 5.3.1.2, Seite 137).

Die Behandlung gesperrter Aktionslisten im OCM geschieht wie folgt: Zunächst wird jede derartige Aktionsliste als global betrachtet, ihre Ausführung also immer durch die NDU veranlaßt, gegebenenfalls nach Empfang einer Ereignisnachricht (vgl. § 6.2.1.2, Seite 160). Wird die Bearbeitung einer derartigen Aktionsliste A bei der NDU angefordert, wartet diese zunächst, bis alle eventuell noch in Bearbeitung befindlichen, globalen Aktionslisten beendet sind. Während dieser Zeit werden eintreffende Anfragen nicht bearbeitet, sondern lediglich in einer Warteschlange zwischengespeichert. Anschließend belegt die NDU bis zur vollständigen Abarbeitung von A eine lokale Sperre. Da die NDU vorher auf die Beendigung *aller* globaler Aktionslisten gewartet hat, ist die Sperre an dieser Stelle des Ablaufs garantiert frei. Solange die Sperre gesetzt ist, führt die NDU nur noch Schritte aus, die der Bearbeitung von A dienen. Andere Anfragen werden wiederum in der Warteschlange gespeichert. Die Aktionsliste A wird nun um knoten- und prozeßlokale Sperren erweitert:

```
node_lock([a_]) proc_lock([a_]);
A;
proc_unlock([a_]) node_unlock([a_])
```

Gleichzeitig wird jede Aktion durch ein Statusbit markiert, um sie als Teil der momentan ausgeführten, gesperrten Aktionsliste kenntlich zu machen. Das Resultat wird dann wie in § 6.2.1.2, Seite 160 beschrieben verteilt.

In einem Monitorprozeß bewirkt die Aktion `node_lock` lediglich das Setzen einer lokalen Sperre. Wegen der sequentiellen, nicht-unterbrechbaren Abarbeitung von Aktionslisten durch den Monitorprozeß und des Sperrprotokolls in der NDU ist diese Sperre bei Ausführung von `node_lock` wiederum garantiert frei. Bis sie durch Ausführung von `node_unlock` wieder freigegeben wird, arbeitet der Monitorprozeß nur noch Aktionen ab, die durch das oben erwähnte Statusbit gekennzeichnet sind. Dieser Sperrmechanismus ist notwendig, da Barrieren innerhalb von A bewirken können, daß von der NDU nacheinander mehrere zu A gehörige Teilanfragen an einen Knoten geschickt werden müssen.

Nun muß noch der wechselseitige Ausschluß mit Aktionslisten sichergestellt werden, die von den Monitorbibliotheken direkt in Anwendungsprozessen ausgeführt werden. Die Aktion `proc_lock` setzt dazu zunächst eine Sperre, die sich im gemeinsamen Speichersegment befindet und bewirkt, daß die Monitorbibliotheken keine neuen Aktionslisten mehr starten, bis die Sperre durch `proc_unlock` wieder aufgehoben wird. Der Monitorprozeß wird durch `proc_lock` dann so lange blockiert, bis alle eventuell noch in Bearbeitung befindlichen Aktionslisten in Anwendungsprozessen beendet sind. Jede Monitorbibliothek setzt dazu ein Statusbit im gemeinsamen Speichersegment, wann immer sie Aktionslisten abarbeitet.

Beim Erreichen der Barriere, die sich in der oben gezeigten erweiterten Aktionsliste unmittelbar vor A befindet, ist also sichergestellt, daß keine Komponente des Monitorsystems mehr Aktionslisten bearbeitet. Gleichzeitig sind alle Komponenten für andere Aktionslisten als A gesperrt. Es ist leicht nachzuprüfen, daß das skizzierte Sperrprotokoll verklemmungsfrei ist, falls die Monitorkomponenten alle Aktionslisten in endlicher Zeit ausführen. Dies kann unter den folgenden beiden Voraussetzungen garantiert werden:

1. Außer den speziellen Synchronisationsaktionen warten keine anderen Aktionen blockierend auf das Eintreten von Ereignissen. Ein derartiges Verhalten von Aktionen ist aufgrund des Ereignis/Aktions-Paradigmas des OCM weder notwendig noch sinnvoll.

2. Anwendungsprozesse werden vom Monitorprozeß nicht angehalten, während ihre Monitorbibliothek Aktionslisten bearbeitet. Diese Bedingung wird durch ein spezielles Synchronisationsprotokoll der Aktion `thread_stop` gesichert.

6.2.3.3 Konsistenzprotokolle

In § 5.3.1.2 wurde begründet, daß die Kommunikationsprotokolle innerhalb des OCM zwei Anforderungen bezüglich der Konsistenz in dem verteilten System erfüllen müssen, das durch die Monitorkomponenten und die Werkzeuge gebildet wird:

1. Bei der Verteilung von Anfragen muß sichergestellt sein, daß die Monitorprozesse die Teilanfragen logisch gleichzeitig empfangen.

2. Beim Zurücksenden von Ergebnissen bedingter Anfragen an Werkzeuge muß die kausale Ordnung der Nachrichten erhalten bleiben.

Die erste Forderung wird dabei im OCM bereits ohne zusätzliche Maßnahmen erfüllt. Da die zentral implementierte NDU die einzige Instanz innerhalb des OCM darstellt, die Anfragen an die lokalen Monitorprozesse verteilt, und die einzelnen Kommunikationskanäle die Nachrichtenreihenfolge erhalten, treffen die lokalen Teilanfragen bei allen Monitorprozessen logisch gleichzeitig ein. Für eine gesperrte Aktionsliste bedeutet dies zusätzlich, daß die lokalen Teile der Aktionsliste auch logisch gleichzeitig abgearbeitet werden. Ansonsten ist dies nur bis zur ersten Barriere innerhalb der Aktionsliste sichergestellt, da diese eine Unterbrechung der Abarbeitung bewirkt.

Auch eine Erhaltung der kausalen Ordnung von Ergebnisnachrichten wird durch die Server-Architektur des OCM in Verbindung mit der zentralen NDU vereinfacht. Diese Struktur hat zur Folge, daß alle Nachrichten von lokalen Monitorprozessen an Werkzeuge über die NDU vermittelt werden. Zur Sicherstellung der kausalen Ordnung müssen in diesem Fall zwei Bedingungen erfüllt werden:

- Nach dem Absenden des Ergebnisses einer bedingten Aktionsliste darf die Ereignisabarbeitung erst dann abgeschlossen, d.h. die Suspendierung des auslösenden Objekts wieder aufgehoben werden, wenn die Ergebnisnachricht bei der NDU angekommen ist. Dies erfordert im allgemeinen die Rücksendung einer Bestätigung.

- Die NDU muß Nachrichten in FIFO-Reihenfolge weiterleiten. Diese Bedingung wird durch die sequentielle Arbeitsweise der NDU sichergestellt.

Das Protokoll erfordert also einen spürbaren Mehraufwand. Allerdings wird dieser Aufwand nur dann notwendig, wenn Werkzeuge über bedingte Anfragen eine unmittelbare Benachrichtigung über das Auftreten eines Ereignisses bzw. den Systemzustand zu diesem Zeitpunkt anfordern. Derartige Monitoranfragen sind jedoch wegen der in jedem Fall entstehenden hohen Kommunikationskosten eher selten. Bei häufig auftretenden Ereignissen werden in der Regel nur manipulierende Aktionen definiert, die z.B. einen Zähler oder eine Ereignisspur aktualisieren. Gemäß der OMIS-Spezifikation erfolgt in diesem Fall – abgesehen von möglichen Fehlermeldungen – bei der Ausführung der Aktionsliste keine unmittelbare Rückmeldung an das Werkzeug.

6.2.3.4 Reihenfolgesteuerung der Ereignisbearbeitung

Zur Vermeidung logischer Konflikte sind im OCM zwei Mechanismen realisiert, mit deren Hilfe die Reihenfolge von Aktionen festgelegt werden kann, die demselben Ereignis zugeordnet sind. Zum einen wurde eine Testimplementierung des bereits in § 5.3.3.2 im Detail beschriebenen Verfahrens zur Reihenfolgesteuerung bei dynamisch definierten Ereignis/Aktions-Relationen durchgeführt, um dessen Tauglichkeit nachzuweisen. Zum anderen wurde das zum OCM gehörige Hilfsprogramm zur statischen Instrumentierung von Programmierbibliotheken so erweitert, daß es eine koordinierte, mehrfache Instrumentierung unterstützt. Dadurch kann auch die Reihenfolge von bedingten Aktionen festgelegt werden, die durch ein Werkzeug, z.B. `Codex`, nicht über die Ereignis/Aktions-Verwaltung des OCM definiert werden, sondern direkt im Instrumentierungscode implementiert sind.

Das Hilfsprogramm unterstützt dazu die mehrfache Anwendung des *Binary Wrapping* (vgl. § 2.1.3.2, Seite 22). Dazu werden zunächst neue Namen für die zu instrumentierenden Bibliotheksfunktionen erzeugt, wobei Konflikte zu bereits existierenden Namen automatisch erkannt und aufgelöst werden. Anschließend werden alle Funktionsdefinitionen und -aufrufe in der Bibliothek konsistent umbenannt. Die neuen, dynamisch erzeugten Namen werden dann dem einzufügenden Instrumentierungscode über *C*-Makros zur Verfügung gestellt, so daß dieser die umbenannten Funktionen weiterhin über a-priori bekannte (Makro-)Namen aufrufen kann. Schließlich wird der Instrumentierungscode übersetzt und der Bibliothek hinzugefügt. Durch die dynamische und eindeutige Namensgenerierung kann das Instrumentierungswerkzeug in identischer Weise wieder auf die entstandene Bibliothek angewandt werden, so daß eine zwiebelschalenartige Struktur wie in Abb. 3.25 erzeugt werden kann. Die Reihenfolge der Aktionsausführung ergibt sich dabei aus der Reihenfolge, in der die Instrumentierungen durchgeführt wurden.

6.2.4 Überwachung und Manipulation von Anwendungsprozessen

Die Realisierung der Überwachung und Manipulation normaler Anwendungsprozesse sei an dieser Stelle nur der Vollständigkeit halber kurz diskutiert. Für Details wird auf die Literatur verwiesen, z.B. [270, 205, 55].

Die Überwachung beruht letztendlich auf nur zwei Mechanismen:

- Der lokale Monitorprozeß kann über die beiden Betriebssystem-Schnittstellen `ptrace` und `/proc` Signale an Anwendungsprozesse abfangen. Falls erforderlich kann ein derartiges Signal an definierten Stellen der Programmausführung durch Einfügen von *Trap*-Instruktionen in den Code erzeugt werden.

- Durch Einwickeln von Funktionen der von der Anwendung genutzten Programmierbibliotheken wird erreicht, daß vor bzw. nach deren Aufruf Codestücke der Monitorbibliothek ausgeführt werden. Diese Codestücke können über Boolesche Variablen im Adreßraum des jeweiligen Anwendungsprozesses einzeln an- und ausgeschaltet werden. Die Variablen werden durch die Instrumentierungsfunktion des zugehörigen Ereignisdienstes (siehe § 6.2.2.2, Seite 166) vom Monitorprozeß über Betriebssystem-Dienste gesetzt.

Analog werden auch Manipulationen der Anwendungsprozesse entweder vom lokalen Monitorprozeß über Betriebssystem-Schnittstellen vorgenommen oder durch Code aus der Monitorbibliothek, der direkt im jeweiligen Anwendungsprozeß ausgeführt wird. Zum Aufruf dieses Codes durch den Monitorprozeß wird ein *Inferior RPC* benutzt. Diese Variante ist in erster Linie für komplexere Manipulationen vorteilhaft, wie z.B. dem Löschen einer Nachricht aus dem Empfangspuffer eines Prozesses. Im Prinzip können solche Manipulationen auch direkt über die vom Betriebssystem bereitgestellte Schnittstelle realisiert werden, jedoch erlaubt diese nur einfache Lese- und Schreiboperationen, was zu einer sehr geringen Effizienz führt.

6.2.5 Überwachung und Manipulation von Werkzeugen

Da für das Monitorsystem ein zu überwachendes Werkzeug lediglich einen speziellen Anwendungsprozeß darstellt, kommen die oben genannten Techniken grundsätzlich auch hier zum Einsatz. Bei der Überwachung und Manipulation von Werkzeugen sind jedoch etwas unterschiedliche Randbedingungen zu berücksichtigen, die nachfolgend diskutiert werden.

6.2.5.1 Überwachung von Funktionsaufrufen

Funktionsaufrufe können effizient durch Einwickeln der betroffenen Funktionen in Code der Monitorbibliothek beobachtet werden. Ein Nachteil dieser statisch eingefügten Instrumentierung ist allerdings, daß sie ein erneutes Binden der zu überwachenden Programme erfordert. Beim eigentlichen Zielprogramm stellt dies kein großes Problem dar, da die hier betrachteten Werkzeuge vorwiegend in der Entwicklungsphase des Programms verwendet werden. In dieser Phase wird das Zielprogramm in der Regel bereits standardmäßig mit der instrumentierten Bibliothek gebunden. In jedem Fall liegt jedoch der Programmcode vor, der für den Bindevorgang mit dieser Bibliothek erforderlich ist, da die Nutzung des Monitorsystems durch den Entwickler des Zielprogramms erfolgt.

Anders ist die Situation, wenn das Monitorsystem Werkzeuge überwachen muß, um deren Interoperation zu realisieren. Hier kann nicht vorausgesetzt werden, daß alle zu beobachtenden Funktionen eines Werkzeugs bereits instrumentiert sind, zumal die im Einzelfall interessierenden Funktionen bei der Erstellung des Werkzeugs noch nicht unbedingt abzusehen sind. Genausowenig kann vom Vorhandensein des notwendigen Codes für ein erneutes Binden ausgegangen werden.

Für die Beobachtung der Werkzeuge sind daher dynamische Instrumentierungstechniken erforderlich, die die Notwendigkeit einer speziellen Vorbereitung vollständig vermeiden. In einer Kooperation mit der University of Wisconsin, Madison[3] wurde zu diesem Zweck eine Monitorerweiterung für den OCM erstellt, die eine Überwachung von Funktionsaufrufen mit Hilfe der Instrumentierungsbibliothek `dyninst` (siehe § 2.1.3.2, Seite 23) realisiert. Da sowohl die `dyninst`-Bibliothek als auch der Kern des OCM nun die (weitgehend) exklusiven Betriebssystem-Schnittstellen zum Zugriff auf die Zielprozesse verwenden und beide Komponenten Signale in diesen Prozessen überwachen müssen, führte die Erweiterung zunächst zu strukturellen und logischen Konflikten. Zur Lösung dieser Konflikte konnten aus den in § 5.3.1.1 genannten Gründen zwar nicht die vom OCM realisierten Mechanismen, wohl aber die in § 4 definierten Methoden verwendet werden. Konkret wurde zur Aufrechterhaltung der Konsistenz eine Benachrichtigung von `dyninst` bei der Erkennung einiger Ereignisse im OCM-Kern realisiert. Zudem mußten implizite Annahmen über die Einstellung einiger Parameter der

[3] Gefördert durch die Deutsche Forschungsgemeinschaft, Forschungsstipendium WI 1346/1-1 [259]

Prozeßkontroll-Schnittstelle des Betriebssystems beseitigt werden. Ein Erfahrungsbericht über einen Teil der entstandenen Probleme und deren Lösung ist in [257] enthalten.

6.2.5.2 Ausführung von Aktionen

Die Technik des *Inferior RPC* ermöglicht den Aufruf von Aktionen in einem Anwendungsprozeß unabhängig vom aktuellen Zustand dieses Prozesses. Dies ist insbesondere bei der Anwendung des Monitorsystems zur Fehlersuche notwendig. Ein Problem kann aber dadurch entstehen, daß eine Aktion zu einem Zeitpunkt aufgerufen wird, zu dem sie nicht korrekt ausgeführt werden kann. Beispielsweise kann ein Prozeß durch einen *Inferior RPC* direkt oder indirekt zum Aufruf einer Funktion veranlaßt werden, während er sich (zufällig) bereits in dieser Funktion befindet. Viele Funktionen, insbesondere auch in Standard-Programmierbibliotheken, sind nicht in diesem Sinne wiedereintrittsfähig. Beim Monitoring des eigentlichen Zielprogramms stellt dies wieder kein gravierendes Problem dar, da zum einen nur sehr spezielle Aktionen Verwendung finden, die auf das Wiedereintrittsproblem Rücksicht nehmen. Zum anderen können das Werkzeug bzw. der Anwender den Aufrufzeitpunkt der Aktionen exakt kontrollieren.

Bei der Überwachung von Werkzeugen rufen die Aktionen dagegen fast ausschließlich Funktionen des Werkzeugs auf, deren Interna nicht im Detail bekannt sind. Zudem sind die Aktionen regelmäßig mit Ereignissen in *anderen* Werkzeugen verknüpft, die völlig asynchron zum Ablauf des ersten Werkzeugs auftreten. Wir benötigen daher eine gewisse Synchronisation, die sicherstellt, daß Aktionen in einem Werkzeug immer an einer sicheren Stelle in dessen Ablauf aufgerufen werden.

Im OCM wurde dazu die folgende Lösung gewählt: Gemäß der OMIS-Spezifikation muß jedes Werkzeug sicherstellen, daß beim (asynchronen) Eintreffen einer Nachricht an gewissen Dateideskriptoren in die von OMIS bereitgestellte Behandlungsfunktion `omis_handler` verzweigt wird (siehe [150], S. 51ff). Da `omis_handler` im Zuge seiner Ausführung beliebige, für Monitoranfragen definierte Rückruffunktionen aktivieren kann, muß das Werkzeug beim Installieren dieser Behandlungsfunktion bereits dafür Sorge tragen, daß sie nur an Stellen im Ablauf ausgeführt werden kann, die nicht zu Wiedereintrittsproblemen führen. Zur sicheren Aktionsausführung wurde daher in diese Behandlungsfunktion gewissermaßen ein Trojanisches Pferd eingebaut. Zum Aufruf einer Aktion innerhalb eines Werkzeugs wird nun vom OCM eine spezielle Nachricht an dieses Werkzeug gesandt, die zur Aktivierung von `omis_handler` führt. Die Funktion erkennt und analysiert diese Nachricht und ruft die entsprechende Aktion direkt auf. Der Trojaner kann dabei fest in die OCM Werkzeug-Bibliothek eingebaut sein oder mit Hilfe der `dyninst`-Erweiterung des OCM auch erst zur Laufzeit eingefügt werden.

6.2.6 Überwachung und Manipulation von Zugriffen des Monitorsystems

Wir haben in § 3.5 gesehen, daß das Monitorsystem neben dem Zielprogramm und den Werkzeugen letztendlich auch sich selbst überwachen und manipulieren können muß. Da eine vollständig allgemeine Überwachung des Monitorsystems durch sich selbst weder notwendig noch möglich ist, beschränken wir uns auf das in § 5.3.3.1, Seite 142 beschriebene Erkennen bzw. Abfangen von Objektzugriffen. Um die technische Umsetzbarkeit des dort entwickelten Objektmodells zu überprüfen, wurde eine Testimplementierung durchgeführt, an die folgende Anforderungen gestellt wurden:

- Realisierung aller Eigenschaften des Objektmodells, insbesondere:

– dynamische Erweiterbarkeit der Objekte um neue Komponenten,
– Möglichkeit spezieller Lese- und Schreibmethoden für Komponenten von Systemobjekten, die eine Stellvertreterfunktion besitzen,
– Methoden zur Beobachtung und zur Manipulation von Schreib- bzw. Lesezugriffen.

- Effizienz: Zugriffe auf Zustandskomponenten müssen ohne Suchaufwand in der Zeit $\mathcal{O}(1)$ durchgeführt werden können. Diese Zugriffszeit ist kritisch, da ein Monitorsystem naturgemäß sehr häufig Objektzugriffe durchführt und ihr Aufwand direkt in die Beeinflussung des Zielsystems eingeht.

- Typsicherheit, d.h. die Existenz von Komponenten sowie die korrekte Verwendung ihres Datentyps sollte trotz der dynamischen Erweiterbarkeit möglichst bereits zur Übersetzungszeit getestet werden können.

- Da der OCM in der Sprache *C* programmiert ist, sollte die Implementierung schließlich in Standard-*C* erfolgen.

Die Realisierung der Objekte selbst basiert auf folgenden Konzepten: Ein Objekt ist im wesentlichen durch ein dynamisches Feld von Zeigern auf Strukturen realisiert. Jede dieser Strukturen faßt all die Komponenten zusammen, die vom selben Erweiterungsmodul definiert werden. Die Objekte werden also nicht um jede Komponente einzeln erweitert, sondern um ganze Gruppen von Komponenten, die jeweils in einer Struktur zusammengefaßt sind. Der Typ dieser Struktur wird dabei vom Erweiterungsmodul statisch definiert. Als Konvention erhält der Typ den Namen *Module_Name*_`data`, wobei *Module_Name* der (eindeutige) Name des Erweiterungsmoduls ist (siehe § 5.3.1.1). Die Typdefinition ist somit zur Übersetzungszeit allen anderen Modulen bekannt, die auf die neuen Komponenten zugreifen wollen.

Zur Laufzeit wird neben den eigentlichen Objekten auch Information über deren Klasse gespeichert. Ein derartiges Klassenobjekt besteht wieder aus einem dynamischen Feld von Zeigern auf Strukturen, die nun den vom Erweiterungsmodul zu definierenden Typ *Module_Name*_`class` besitzen. Diese Strukturen enthalten unter anderem Zeiger auf spezifische Lese- und Schreibmethoden für alle Komponenten der zugehörigen Objekte. Der Selektor des jeweiligen Funktionszeigers ergibt sich aus dem Komponentennamen, indem ihm das Suffix `_r` bzw. `_w` angefügt wird.

Beim dynamischen Einbinden eines Erweiterungsmoduls, das Objekten neue Komponenten hinzufügt, wird dann zunächst mit Hilfe des Klassenobjekts der nächste freie Platz in den dynamischen Feldern bestimmt, an dem die Zeiger auf die entsprechenden Erweiterungsstrukturen eingetragen werden können. Anschließend erfolgt die Erweiterung des Klassenobjekts und aller bereits existierender Objekte dieser Klasse. Bei nachfolgend erzeugten Objekten wird die Erweiterungsstruktur dann aufgrund der erweiterten Klasseninformation bereits bei der Erzeugung angelegt. Durch diese Vorgehensweise ist sichergestellt, daß die Erweiterungsstruktur eines Moduls in jedem Objekt an derselben Position des dynamischen Feldes gespeichert ist[4].

Lese- und Schreibzugriffe auf die so realisierten Objekte erfolgen ausschließlich über die beiden *C*-Makros

```
read(object_token, module_name, selector)
write(object_token, module_name, selector, new_value)
```

[4] Dies gilt nur lokal auf jeweils einem Knoten, was ausreichend ist, da alle Zugriffe lokal erfolgen.

Durch Aufnahme des Namens des Erweiterungsmoduls in die Makros wird dieser Name in dem Objektmodell aus § 5.3.3.1, Seite 142 de facto zum ersten Teil des Komponentenselektors, was die Allgemeinheit des Modells jedoch in keiner Weise einschränkt.

Bei der Implementierung der Makros wird das sogenannte *Token Pasting* [59] des ANSI-*C*-Präprozessors ausgenutzt. Dadurch können aus den übergebenen Parametern durch den Präprozessor gültige *C*-Identifikatoren für die oben genannten Typen sowie die Selektoren der Lese- und Schreibmethoden zusammengebaut werden. Für den eigentlichen Zugriff müssen die Makros den Namen des Erweiterungsmoduls auf den zugehörigen Index abbilden, unter dem der Zeiger auf die Erweiterungsstruktur in einem Objekt gespeichert ist. Da sich diese Abbildung nach dem Einbinden einer Erweiterung nicht mehr ändert, wird der Index bereits während des Einbindens in einer globalen Variable gespeichert, deren Name wieder aus dem des Erweiterungsmoduls abgeleitet ist. Der Zeiger auf die Erweiterungsstruktur, in der sich die selektierte Komponente befindet, kann somit in konstanter Zeit ermittelt werden. Diesem Zeiger wird in den Makros nun durch eine *Type-Cast*-Operation der korrekte Typ zugewiesen, so daß die eigentliche Komponentenselektion durch die normalen Sprachmittel von *C* erfolgen kann. Dadurch erfolgt die Existenz- und Typprüfung bereits zur Übersetzungszeit durch den Standard-*C*-Compiler, ebenso wie die Umsetzung des Selektors in die zugehörige Relativadresse.

Vor dem eigentlichen Zugriff wird wie in Abb. 5.2 angedeutet ein Vergleich mit den zu meldenden bzw. abzufangenden Zugriffen durchgeführt. Der Vergleich von Modulnamen und Selektoren kann dabei sowohl über den zugeordneten Index bzw. die Relativadresse als auch – durch eine weitere Funktion des ANSI-*C*-Präprozessors, das sogenannte *Stringizing* [59] – mit Hilfe von Zeichenketten erfolgen. Letzteres ermöglicht dann auch einen Mustervergleich über reguläre Ausdrücke, wie er in § 5.3.3.1, Seite 142 erwähnt wurde.

Der Laufzeitaufwand dieser Implementierung ist sehr gering: Wenn für das betroffene Objekt keine Überwachung definiert wurde, entsteht beim Zugriff lediglich ein Mehraufwand von sechs Speicherzugriffen und drei bedingten Sprüngen. Ein nennenswerter Einfluß auf die Rückwirkung des Monitorsystems kann damit ausgeschlossen werden.

Die durchgeführte Implementierung hat somit gezeigt, daß das vorgeschlagene, dynamische Objektmodell prinzipiell mit akzeptablem Laufzeitaufwand realisiert werden kann. Wegen der Realisierung durch Makros kann im praktischen Einsatz allerdings die resultierende Codegröße ein Problem darstellen. Da der durch die Makros erzeugte *C*-Code hochgradig von deren Aktualparametern abhängt, ist eine Komprimierung dieses Codes, z.B. durch eine im Makro-Rumpf aufgerufene Funktion, nicht möglich. Der eigentliche Faktor, der eine Integration des implementierten Objektmodells in den OCM bisher verhindert hat, ist jedoch schlicht und einfach die Tatsache, daß die übliche Syntax der Objektzugriffe im Code nicht beibehalten werden kann. Der Aufwand für die notwendige Anpassung jedes einzelnen Zugriffs ist daher enorm.

Die Frage, ob und in wieweit die dynamische Erweiterbarkeit und Überwachbarkeit von Objekten durch existierende *objektorientierte* Sprachen oder entsprechend erweiterte Sprachmittel unterstützt werden kann, stellt einen interessanten Gegenstand für zukünftige Forschungen dar. Diese Konzepte sind letztendlich nicht nur für Monitorsysteme, sondern für jegliche Art von erweiterbarer Software relevant.

Kapitel 7

Realisierung der Werkzeugumgebung

Das folgende Kapitel beschreibt die Realisierung der in § 3 konzipierten Werkzeugumgebung zur Fehlersuche in parallelen Programmen mit hoher Ausführungszeit. Die Umgebung, die mit dem Akronym `RAD`[1] bezeichnet wird, ist auf der Basis des erweiterten Monitorsystems OCM realisiert. Sie besteht wie in § 3 beschrieben aus dem Debugger `DETOP`, dem Sicherungspunktgenerator `CoCheck`, dem Visualisierer `VISTOP` und dem Werkzeug `Codex` zur deterministischen Ablaufsteuerung. Neben dem eigentlichen Zweck für Anwendungsprogrammierer dient sie vor allem zur Demonstration und Bewertung der im Rahmen dieser Arbeit konzipierten Infrastruktur. Wir werden daher nach einem kurzen Blick darauf, wie sich `RAD` einem Anwender darstellt, diskutieren, welche Schritte erforderlich waren, um die Werkzeuge auf Basis der Infrastruktur interoperabel zu machen. Aus der Erfahrung, die bei der Realisierung von `RAD` gesammelt wurde, leiten wir in § 8 eine qualitative Bewertung der Infrastruktur und der durch sie induzierten Vorgehensweise zur Integration von Laufzeit-Werkzeugen ab.

7.1 Die Werkzeugumgebung aus Anwendersicht

Wie Bild 7.1 zeigt, präsentiert sich `RAD` für den Anwender durch die graphischen Benutzerschnittstellen der einzelnen Werkzeuge[2], die durch ein zusätzliches Fenster zur Steuerung der Gesamtumgebung ergänzt werden.

7.1.1 Überblick

Die gesamte Umgebung wird durch ein einziges Kommando `rad` gestartet, das als Argumente die Projektdatei für das Werkzeug `Codex` sowie den Namen und die Parameter des zu untersuchenden Programms erhält. Das auf diese Weise aufgerufene Rahmenwerkzeug stellt zunächst die Verbindung mit dem Monitorsystem OCM her und startet dann das zu untersuchende Anwendungsprogramm. Anschließend werden die drei Werkzeuge `DETOP`, `VISTOP` und `Codex` so aufgerufen, daß sie sich an die bereits existierende Zielanwendung ankoppeln. Da `CoCheck` als Erweiterung des OCM realisiert ist, muß dieses Werkzeug nicht explizit gestartet werden.

Wird das zu Beginn gestoppte Anwendungsprogramm mit Hilfe des Debuggers fortgesetzt, blockiert dieses zunächst, bis im Werkzeug `Codex` das zu verwendende Kontrollmuster spezifiziert wird. Nun kann der durch dieses Muster determinierte Programmablauf gleichzeitig

[1] `RAD` = *Random Access Debugger* = Debugger mit wahlfreiem Zugriff auf die Programmzustände.

[2] Der Sicherungspunktgenerator `CoCheck` besitzt keine Benutzerschnittstelle.

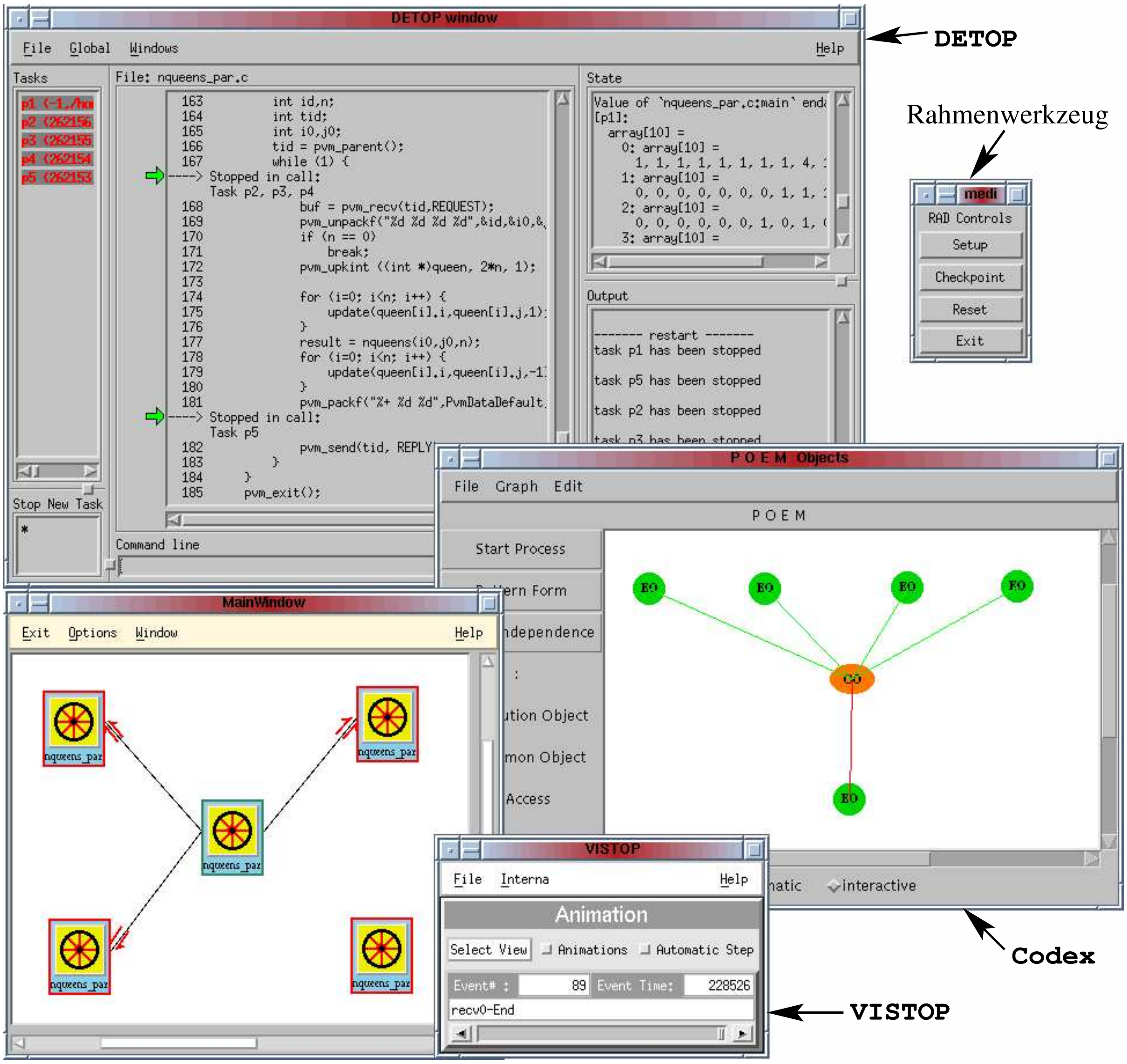


Abbildung 7.1: Die Werkzeugumgebung RAD in Aktion

durch DETOP und VISTOP untersucht werden, wodurch dem Programmierer sowohl Information über den internen Ablauf einzelner Prozesse als auch über deren globale Interaktion zur Verfügung steht. Dabei ist die Konsistenz der dargestellten Information im Rahmen der in § 3.3.2 festgelegten Kriterien gesichert.

Die Zusatzfunktionen zum Sichern und Wiederherstellen von Programmzuständen sind über die zusätzliche Bedienoberfläche erreichbar, die vom Rahmenwerkzeug realisiert wird. Der Anwender kann Sicherungspunkte bei Bedarf manuell auslösen oder vom System automatisch in periodischen Abständen erstellen lassen (Schaltflächen *Checkpoint* bzw. *Setup*, vgl. § 3.4.1). Das *Reset*-Kommando setzt das Zielprogramm auf den Zustand zurück, der aktuell von VISTOP angezeigt wird (vgl. § 3.4.2). Über den Knopf *Exit* kann schließlich die gesamte Werkzeugumgebung koordiniert beendet werden.

7.1.2 Sicherungspunkterstellung

Die Wahl der Sicherungsfrequenz bleibt in der Werkzeugumgebung dem Anwender überlassen, da die optimale Frequenz eine Abwägung erfordert zwischen dem Laufzeitaufwand, der durch die Erzeugung der Sicherungspunkte entsteht, und der Zeit, die für das Rücksetzen benötigt wird. Der Aufwand für einen einzelnen Sicherungspunkt ist dabei von der Anzahl der Anwendungsprozesse und deren Datensegmentgröße abhängig. Um die Größenordnung zu verdeutlichen sei ein Ergebnis von Trinitis aus [235], Kap. 7.1 zitiert. Für das parallele Strömungssimulationsprogramm `ParTfC`, das auf 4 Knoten eines SUN Sparc Clusters die Umströmung eines Zylinders auf einem Gitter mit 1.360.775 Elementen berechnet, dauert die Erstellung eines Sicherungspunkts knapp 3 Sekunden. Die Größe der einzelnen Datensegmente liegt dabei bei 13 MByte. Bei einer Sicherung im Minutentakt entsteht also ein Mehraufwand von lediglich 5%, der allerdings linear mit der Anwendungsgröße wächst. Die Sicherungsfrequenz muß daher bei größeren Anwendungen gegebenenfalls reduziert werden, um deren Ablauf nicht zu sehr zu verlangsamen. Für die Fehlersuche ist wichtig, daß sich die Verzögerung des Anwendungsprogramms auch bei noch so häufiger Sicherung wegen der deterministischen Steuerung durch `Codex` keinesfalls auf das logische Programmverhalten auswirken kann.

Die Zeit, die für das Rücksetzen des Programms auf einen gegebenen Zielzustand benötigt wird, ist in erster Näherung durch den Zeitabstand der Sicherungspunkte gegeben, da der Zielzustand durch Wiederausführung ab dem unmittelbar vorangehenden[3] Sicherungspunkt erreicht wird. Solange dieser Zeitabstand also mindestens etwa eine Größenordnung kleiner als die Laufzeit des Programms ist, ergibt sich ein deutlicher Gewinn. Für das oben erwähnte Programm `ParTfC` ist in [135], Kap. 7.8.2.7 für die Zylinderumströmung mit einem wesentlich kleineren Gitter von 81.375 Elementen auf 4 Knoten eines Clusters bereits eine Rechenzeit von 1.045 Sekunden für 10 Zeitschritte angegeben. Für eine vollständige Simulation werden in der Regel etwa 70 Zeitschritte benötigt, so daß sich für das oben erwähnte große Gitter eine Rechenzeit von ca. 34 Stunden abschätzen läßt. In diesem Fall wäre also selbst eine stündliche Sicherung noch äußerst vorteilhaft.

Neben dem Laufzeitaufwand beschränkt vor allem der Speicherplatzbedarf die maximal mögliche Sicherungsfrequenz, da im Gegensatz zur Erstellung von Sicherungspunkten zur Fehlertoleranz nicht nur die letzte Sicherung wichtig ist, sondern alle. In dem `ParTfC`-Beispiel würden bei minütlicher Sicherung nach einer Stunde insgesamt bereits 3,12 GByte Plattenplatz benötigt. In dem typischen Anwendungsszenario von `RAD`, nämlich lang laufenden Anwendungen, in deren *Endphase* ein Fehler auftritt, kann ein Anwender dieses Problem etwas entschärfen, indem er die automatische Sicherungspunkterzeugung erst kurz vor der fehlerverdächtigen Phase aktiviert. Realisierbar ist auch ein adaptives Verfahren, das bei Bedarf einige, aber nicht alle, der ältesten Sicherungspunkte wieder löscht. Dadurch dauert das (weniger wahrscheinliche) Rücksetzen auf sehr frühe Zustände zwar entsprechend länger, ist aber immer noch schneller als eine vollständige Neuausführung.

7.2 Notwendige Anpassungen der Werkzeuge

Das erklärte Ziel dieser Arbeit ist es, eine Infrastruktur zu schaffen, die es erlaubt, existierende Werkzeuge mit möglichst wenigen Änderungen interoperabel zu machen, im Idealfall sogar

[3] Vorausgesetzt ist dabei, daß das Zielprogramm ausreichend häufig überwachte Ereignisse produziert; vgl. die Diskussion in § 3.4.2.

gänzlich ohne eine Modifikation. Dieses Ideal konnte im vorliegenden Fall zwar nicht ganz erreicht werden, zur Realisierung von RAD waren aber tatsächlich nur drei Änderungen an zwei der beteiligten Werkzeuge erforderlich.

Zunächst mußten VISTOP und Codex so erweitert werden, daß sie sich an bereits existierende Zielprozesse ankoppeln können, zumindest dann, wenn sichergestellt ist, daß sich diese noch am Anfang ihrer Ausführung befinden. Dazu wurden die Werkzeuge um eine Kommandozeilenoption erweitert, die als Parameter den Identifikator des initialen Prozesses erhält[4]. VISTOP fordert vom OCM dann den Dienst proc_attach anstelle von proc_create an. Analog überspringt Codex den normalerweise ausgeführten PVM-Aufruf zur Prozeßerzeugung. Es sei erwähnt, daß diese Änderungen im Prinzip nicht fest in den Code der Werkzeuge hätten eingebaut werden müssen. Stattdessen könnte auch das Monitorsystem zur Laufzeit die entsprechenden Aufrufe der Funktionen omis_request bzw. pvm_spawn abfangen und geeignet modifizieren. Auf diese Möglichkeit wurde verzichtet, da zum einen die Fähigkeit des Ankoppelns an existierende Prozesse eine Grundvoraussetzung für jegliche Interoperabilität ist und somit vom Werkzeug selbst realisiert werden sollte. Zum anderen ist Codex in der Sprache *Java* realisiert und kann deshalb zur Zeit nicht vom OCM überwacht werden.

Eine größere Änderung wurde im Werkzeug VISTOP notwendig, da die von VISTOP vorausgesetzte FIFO-Eigenschaft der PVM-Nachrichtenwarteschlangen durch Codex zerstört wird. Um dieses Problem zu lösen wurde die Instrumentierung der betroffenen Ereignisse wie in § 3.5.6.2 skizziert abgeändert.

Neben diesen Modifikationen sind in der derzeitigen Realisierung zusätzlich auch relevante Ereignisse in DETOP und VISTOP durch sogenannte *Hooks* gekennzeichnet. Dies sind Aufrufe leerer Funktionen, die sich in einer eigenen Bibliothek befinden und sich somit problemlos durch die im OCM als Standard vorgesehene Instrumentierungstechnik des *Binary Wrapping* beobachten lassen. Sie wurden eingeführt, damit während der Entwicklungs- und Testphase der OCM-Infrastruktur auf deren dyninst-Erweiterung verzichtet werden konnte, um die Komplexität der Infrastruktur überschaubar zu halten. Es hat sich jedoch die in § 2.3.4.2 geäußerte Vermutung bestätigt, daß die relevanten Ereignisse durchwegs mit dem Aufruf oder der Rückkehr einer im Werkzeug bereits vorhandenen Funktion assoziiert sind. Ihre Überwachung ist also mit Hilfe von dyninst auch ohne eine statische Vorbereitung der Werkzeuge möglich. Eine entsprechend angepaßte Version von RAD, die die dyninst-Dienste des OCM nutzt, ist verfügbar.

7.3 Dynamisch eingebundene Aktionen

Bis auf die eben erwähnten Modifikationen erfolgen alle anderen notwendigen Erweiterungen der Werkzeuge dynamisch zur Laufzeit. Dazu wurde je eine OCM-Erweiterung für DETOP und VISTOP definiert, die Aktionen in diesen Werkzeugen als Dienste an der OMIS-Schnittstelle zur Verfügung stellt. Die Schnittstellen dieser insgesamt zehn Dienste sind in den Abb. 3.17 und 3.23 definiert. Abgesehen von get_events konnten alle diese Dienste mit je etwa 5 Zeilen *C++*-Code realisiert werden, da sie lediglich bereits innerhalb der Werkzeuge existierende Funktionen und Methoden aufrufen. Die Realisierung von get_events ist etwas aufwendiger, weil sie die zurückgelieferte Ereignisspur in ein von VISTOP unabhängiges Datenformat konvertiert.

[4]Die Umgebung RAD wurde für PVM-Programme realisiert. Diese starten zunächst mit genau einem Prozeß.

Eine kleinere technische Schwierigkeit ergab sich durch die Implementierung der Werkzeuge in *C++*. Durch die Schutzmechanismen von *C++* können dem Code hinzugefügte, externe Funktionen nicht notwendigerweise auf alle Methoden bzw. Komponenten von *C++*-Objekten unmittelbar zugreifen. Um einen Zugriff auch auf geschützte Klassenmitglieder möglich zu machen, kann die Erweiterung abgeleitete Klassen definieren, die die nötigen Funktionen als öffentliche Methoden bereitstellen. Bei den hier betrachteten Werkzeugen `DETOP` und `VISTOP` erwies sich diese Methode bereits als ausreichend. Bei nicht sorgfältig entworfenen Klassen kann jedoch bei einer Erweiterung auch ein Zugriff auf Mitglieder der Klasse notwendig werden, die als privat deklariert wurden. Technisch ist dies ebenfalls ohne große Probleme möglich, da die Klassendefinition dem Übersetzer nur durch eine in den Quellcode der externen Funktion eingebundene *Header*-Datei bekannt ist. Durch eine entsprechende Anpassung dieser Datei – im einfachsten Fall durch Löschung des Schlüsselworts `private`, etwas sauberer durch Einfügen einer Zugriffsmethode in die Klasse – kann der Schutz daher umgangen werden. Voraussetzung dabei ist allerdings, daß sich durch die Anpassung das Speicherlayout der Objekte nicht ändert, was bei allen untersuchten *C++*-Übersetzern der Fall ist. Letztendlich macht sich in dieser Diskussion die Tatsache bemerkbar, daß die Sprache *C++* kein Konzept zur dynamischen Erweiterung von Klassen bzw. Objekten unterstützt (vgl. § 6.2.6), wie es für eine universelle *Plug-in*-Architektur notwendig wäre.

7.4 Das Rahmenwerkzeug

Das Rahmenwerkzeug für `RAD` wurde als generisches Modul entwickelt, das zur Laufzeit für eine spezielle Werkzeugumgebung konfiguriert werden kann. Dadurch können durch dieses Rahmenwerkzeug, das wegen seiner Mediator-Rolle den Namen `MEDI` trägt, in einfacher Weise auch andere OCM-basierte Werkzeugumgebungen unterstützt werden. Die Konfiguration erfolgt dabei zum einen durch ein einfaches Skript, zum anderen durch ein *Plug-in* in Form einer dynamischen Bibliothek.

Bild 7.2 zeigt das vollständige, für `RAD` verwendete Skript. Die Anweisung `load` bewirkt zunächst das Laden der für `RAD` spezifischen dynamischen Bibliothek in das Rahmenwerkzeug. Nach dem Laden dieser Bibliothek wird deren Initialisierungsfunktion gerufen, die die graphische Bedienoberfläche erzeugt.

Der Rest des Skripts besteht ausschließlich aus OMIS-Anfragen, die um zwei Konstrukte erweitert sind: Zum einen können Ergebnisse von Anfragen an Variable zugewiesen und in nachfolgenden Anfragen wieder eingesetzt werden. Einige Variablen, z.B. `app_name` sind dabei von `MEDI` vordefiniert. Zum anderen können für bedingte Anfragen Rückruffunktionen in der dynamischen Bibliothek spezifiziert werden, die beim Eintreffen eines Ergebnisses aktiviert werden. Diese Erweiterungen sind in Abb. 7.2 durch Kursivdruck hervorgehoben.

Die in dem Skript definierten OMIS-Anfragen werden – nach entsprechender Variablensubstitution – direkt an den OCM gesandt. Das Skript für `RAD` erzeugt also zunächst den initialen Anwendungsprozeß, sowie die Werkzeuge `DETOP` und `VISTOP`. `CoCheck` ist als Teil des OCM zu diesem Zeitpunkt bereits aktiv, während `Codex` erst zu einem späteren Zeitpunkt gestartet werden kann (siehe unten). Anschließend werden die in Abb. 3.17 und 3.23 definierten Aktionen in `DETOP` und `VISTOP` eingebunden und die darauf basierenden Verhaltensregeln programmiert (Anfragen `csr1` ... `csr4`). Die verbleibenden Anfragen dienen für die Funktion des Rahmenwerkzeugs selbst.

Nach der Ausführung des Skripts arbeitet das Rahmenwerkzeug vollständig ereignisorien-

```
! Laden der Bibliothek für RAD
load rad.so

! Erzeugen der Anwendung und der Werkzeuge
appl = :proc_create([#local_node], "#app_name",[#app_args],[],[])
detop = :tool_create([#local_node], "detop",["-a","--","#appl"],[],[])
:proc_load_extension([#detop],"detopio")
vistop = :tool_create([#local_node], "Vistop",["-a","#appl"],[],[])
:proc_load_extension([#vistop],"vistopio")

! Verhaltensregeln der Werkzeuge
csr1 = thread_has_been_stopped([])\
       : vistopio_read_events([#vistop])\
         vistopio_show_last_state([#vistop])
csr2 = thread_has_started_lib_call([#vistop],"VisualizeCurrentState")\
       : detopio_activate_state_display([#detop],1)
csr3 = thread_has_started_lib_call([#vistop],"VisualizePastState")\
       : detopio_activate_state_display([#detop],0)
csr4 = thread_has_started_lib_call([#detop],"DetopIsActive")\
       : vistopio_show_last_state([#vistop])

! Anfragen für das Rahmenwerkzeug selbst
csr5 = vistop_beatask([#appl])\
       : print([$tid,$proc]) > beataskCb
csr6 = thread_creates_proc([])\
       : proc_attach([$new_proc])\
         proc_get_info([$new_proc], 0x1) > creationCb
csr7 = thread_has_been_stopped([]) : print([$proc]) > stopCb
csr8 = thread_has_been_continued([]) : print([$proc]) > contCb

! Aktivierung aller bedingten Anfragen
:csr_enable(#csr1,#csr2,#csr3,#csr4,#csr5,#csr6,#csr7,#csr8])
```

Abbildung 7.2: Skript zur Programmierung der Verhaltensregeln in RAD

tiert, wobei Ereignisse vom Zielsystem oder von der graphischen Benutzerschnittstelle kommen können. Neben dem Sichern und Rücksetzen des Zielprogramms, das gemäß den Algorithmen in Abb. 3.18 und 3.22 realisiert sind, implementiert das Rahmenwerkzeug drei weitere Hilfsfunktionen:

1. Start des Werkzeugs `Codex`. Im Gegensatz zu den anderen Werkzeugen basiert `Codex` nicht auf dem Monitorsystem OCM, sondern interagiert direkt mit einer entsprechenden Instrumentierung der PVM-Bibliothek im Zielprogramm. Da diese Interaktion durch PVM-Nachrichten erfolgt, benötigt `Codex` zum Ankoppeln an einen Prozeß dessen PVM-Task-Identifikator (TID). Dieser existiert jedoch noch nicht, wenn ein neuer Prozeß durch den OCM erzeugt wurde, sondern wird ihm erst beim Aufruf der ersten PVM-Funktion zugewiesen. Das Rahmenwerkzeug überwacht daher dieses Ereignis (`csr5`) und startet `Codex` entsprechend verzögert. `Codex` wird also erst aktiv, nachdem die

Anwendung bereits mit der Ausführung begonnen hat. Da der von `Codex` eingebrachte Instrumentierungscode die Anwendung jedoch solange blockiert, bis das Kontrollmuster übertragen wurde, ist die korrekte Funktion sichergestellt.

Eine für den Anwender befriedigendere Lösung wäre, `Codex` gleichzeitig mit den anderen Werkzeugen zu starten und ihm später durch eine vom OCM ausgelöste Aktion den TID bekanntzugeben. Daß dies derzeit nicht realisierbar ist, liegt lediglich an der bereits erwähnten Tatsache, daß `Codex` aufgrund seiner Implementierung in *Java* nicht durch den OCM überwacht werden kann.

2. Überwachung der Prozeßerzeugung. Damit das Rahmenwerkzeug bei der Ausführung der Algorithmen zum Sichern und Rücksetzen alle Prozesse der Zielanwendung steuern kann, koppelt es durch die Anfrage `csr6` an alle neu erzeugen Prozesse an. Zugleich wird mit dieser Anfrage der TID ermittelt und zusammen mit dem entsprechenden Prozeßtoken gespeichert. Diese Zuordnung ist zur Umsetzung der beim Rücksetzen von `VISTOP` erfragten Ereignisspur erforderlich, in der Prozesse durch deren TID identifiziert werden.

3. Schließlich wird das Anhalten und Weiterführen von Prozessen überwacht, um die Uhr für die periodische Sicherungspunkterstellung in Ausführungspausen anhalten zu können. Dadurch wird eine unnötige Sicherung verhindert, wenn das Zielprogramm längere Zeit angehalten ist, was bei der Fehlersuche durchaus häufiger vorkommt.

Kapitel 8

Bewertung

Aus den Erfahrungen, die bei der Realisierung der Werkzeugumgebung RAD gewonnen wurde, wird im folgenden eine erste qualitative Bewertung der vorgeschlagenen Konzepte sowie deren Umsetzung in der Werkzeug-Infrastruktur OCM abgeleitet. Die Bewertung versucht sich dabei soweit möglich von der konkret implementierten Umgebung zu lösen, so daß sie prinzipiell auch für andere Anwendungsfälle gültig ist.

Zunächst muß festgehalten werden, daß die Wunschvorstellung, in der ein Anwender nach Belieben unabhängig entwickelte Werkzeuge starten kann, und diese dann konsistent zusammenarbeiten, nicht erreicht werden kann. Dies liegt letztendlich daran, daß eine derartige Zusammenarbeit nur möglich ist, wenn Wissen über mögliche positive oder negative Beeinflussungen der Werkzeuge vorhanden ist. Dies setzt aber ein globales Wissen über *alle* an der Umgebung beteiligten Werkzeuge voraus. Wenn wir dieses Wissen nicht in den Werkzeugen selbst realisieren wollen, um eine gegenseitige Abhängigkeit zu vermeiden, muß es in einer eigenständigen Komponente – dem Rahmenwerkzeug – gesammelt werden. Eine Kombination von unabhängigen Werkzeugen ist somit nur möglich, wenn ein derartiges Rahmenwerkzeug bereits vorliegt. Ad-hoc zusammenstellbare Umgebungen sind damit ausgeschlossen.

Weitere Voraussetzungen müssen durch die Werkzeuge selbst erfüllt werden. Zum einen müssen sie über kompatible Monitorsysteme verfügen, die zumindest frei von strukturellen Konflikten sind. Im Idealfall sollten die Werkzeuge auf einer gemeinsam genutzten Monitorschnittstelle aufsetzen. Obwohl derartige Schnittstellen existieren (siehe § 2.3.4.1) und weitere entwickelt werden (z.B. [196]), verwenden nahezu alle kommerziellen Werkzeuge noch proprietäre Monitorsysteme. Eine Umstellung auf standardisierte Schnittstellen wäre zwar technisch oft problemlos möglich, wird aber aus finanziellen und auch aus produkt-strategischen Überlegungen unterlassen. Bei modernen Werkzeugen meist erfüllt ist dagegen die Voraussetzung, daß sich diese auch an eine bereits laufende Anwendung ankoppeln können. Wie das Beispiel von VISTOP und Codex gezeigt hat, muß schließlich auch sichergestellt sein, daß implizit über das Zielsystem getroffene Annahmen eines Werkzeugs nicht von den anderen verletzt werden. Interoperabilität läßt sich daher nur äußerst schwer erreichen, wenn diese nicht schon bei der Werkzeugentwicklung als eine prinzipielle Möglichkeit mit eingeplant wurde. Die Details der Kooperation mit konkreten anderen Werkzeugen können und sollten dagegen erst später bei Bedarf realisiert werden.

8.1 Bewertung der Methodik

Werden die angesprochenen Voraussetzungen von den Werkzeugen erfüllt, können mit Hilfe der in § 4 definierten Methode die auftretenden Konflikte analysiert und mit Hilfe des Monitorsystems aufgelöst werden.

Der Analyseprozeß erfordert dabei trotz der formalen Hilfsmittel ein hohes Maß an Hintergrundwissen über die Implementierung der einzelnen Werkzeuge sowie an Intuition, um mögliche Konflikte vorauszusehen, die dann näher zu überprüfen sind. So konnte in dem Beispielszenario `RAD` nicht verhindert werden, daß ein Konflikt bei der Analyse übersehen und erst beim Test der Umgebung erkannt wurde[1]. Weiterhin muß bekannt sein, inwieweit die Werkzeuge bereits intern eine Auflösung möglicher Konflikte durchführen, etwa durch Überwachung kritischer Ereignisse zur Konsistenzsicherung (siehe § 4.3.2.2, Seite 123). Für die Realisierung eventuell zusätzlich benötigter Aktionen innerhalb der Werkzeuge und die Festlegung relevanter Ereignisse ist ebenfalls detailliertes Implementierungswissen erforderlich. Dies bedeutet, daß in der Regel die Konfliktanalyse und -auflösung nur in Zusammenarbeit mit den jeweiligen Werkzeug-Entwicklern möglich sein wird, bzw. zumindest eine detaillierte Dokumentation der Implementierung voraussetzt.

Zur Umsetzung der notwendigen Interaktionen zwischen den Werkzeugen hat sich das Mediator-Schema voll bewährt, da es eine logische Trennung der lokalen Aktionen und Ereignisse einerseits und deren globaler Verknüpfung andererseits erlaubt. Neben der Vermeidung von Abhängigkeiten unterstützt diese Trennung eine schnelle Prototypenentwicklung, bei der inkrementell mehr und mehr Verknüpfungen zwischen den Werkzeugen realisiert und getestet werden. Häufige Regressionstests sind andererseits bei Verwendung dieses ereignisorientierten Schemas auch notwendig. Die Erfahrung während der Realisierung von `RAD` hat gezeigt, daß wegen der völlig asynchronen und (zumindest logisch) dezentral gesteuerten Interaktionen die entstehenden globalen Abläufe teilweise sehr überraschend und äußerst schwer nachzuvollziehen sind. Als Beispiel sei genannt, daß bei der Aktivierung von `DETOP` nicht nur `VISTOP` wieder den aktuell letzten Zustand visualisiert, sondern dadurch wiederum die Zustandsanzeige in `DETOP` aktiviert wird. Ist das Verhalten hier durchaus korrekt und gewünscht, so ist das nächste bereits deutlich problematischer: Stoppt ein Zielprozeß, so wird `VISTOP` veranlaßt, die Ereignisspur vom OCM zu lesen. Hält jedoch währenddessen ein weiterer Prozeß an, so wird `VISTOP` erneut, d.h. rekursiv zum Aufruf der Lesemethode veranlaßt, was zu einem Fehlverhalten führt. Da dies ein generelles Problem ist, wurden bereits im OCM Maßnahmen getroffen, die zeitlich überlappende Aufrufe von Aktionen in Werkzeugen verhindern. Genaugenommen wäre zur Programmierung der Verhaltensregeln ein Meta-Werkzeug erforderlich, das eine Beobachtung ihrer Auswirkungen erlaubt. Ein derartiges Werkzeug sollte auf der Basis des OCM leicht zu realisieren sein. Dies wurde in der vorliegenden Arbeit jedoch nicht näher untersucht.

Das Einbinden der neuen Aktionen in die Werkzeuge zur Laufzeit hat gegenüber einer statischen Modifikation des Quellcodes Vor- und auch Nachteile. Abgesehen davon, daß ein Einbau in den Quellcode an der Verfügbarkeit dieses Codes, der benötigten Programmierbibliotheken oder auch der Compiler scheitern kann, bewirkt das Einbinden zur Laufzeit zwangsläufig eine klare Trennung zwischen dem eigentlichen Werkzeugcode und Code, der nur zur Unterstützung der Interoperabilität dient. Dies ist aus dem Blickwinkel des Software-Entwurfs und auch der Wartbarkeit des Werkzeugs sehr vorteilhaft. Positiv ist auch, daß durch das dynamische Laden nur diejenigen Aktionen zum Werkzeug hinzugefügt werden, die in der aktuellen Umgebung

[1] Modifikation der *Hook Arrays* durch das Restaurieren Programmzustands aus einem Sicherungspunkt, vgl. § 3.5.2.2, Seite 100.

wirklich notwendig sind. Wie bei jeder Modularisierung muß aber sichergestellt sein, daß die Schnittstellen der Module konsistent zueinander bleiben. Da im vorliegenden Fall die Aktionen jedoch interne, möglicherweise undokumentierte Schnittstellen des Werkzeugs nutzen, kann bei einer Weiterentwicklung des Werkzeugs ein erheblicher Wartungsaufwand zur Anpassung dieser Aktionen entstehen. Es sollte daher durchaus angestrebt werden, Aktionen fest in ein Werkzeug zu integrieren, falls

1. ihre Nutzung auch für andere Werkzeugumgebungen von Interesse sein könnte,

2. sie relativ stark mit Interna des Werkzeugs verkoppelt sind, also nicht auf einer wohldefinierten Schnittstelle des Werkzeugs basieren,

3. dadurch keinerlei Abhängigkeiten zu anderen Werkzeugen erzeugt werden.

Im Beispielszenario treffen diese Eigenschaften auf die Funktion zum Ankoppeln an existierende Prozesse zu, was ein Grund dafür war, diese fest in `VISTOP` und `Codex` zu integrieren.

8.2 Bewertung der Infrastruktur

Die während der technischen Realisierung der Werkzeugumgebung gewonnenen Erfahrungen erlauben auch eine erste Bewertung der erweiterten OMIS/OCM-Infrastruktur.

Zunächst hat sich bestätigt, daß die Idee, ein verteiltes Monitorsystem auch als Middleware für Werkzeug-Interaktionen zu verwenden, tragfähig ist und in der Praxis funktioniert. Die dynamische Erweiterbarkeit des Monitorsystems vereinfacht dessen Benutzbarkeit dabei extrem. Dies trifft ebenso auf die Skriptsteuerung des universellen Rahmenwerkzeugs `MEDI` zu, mit dessen Hilfe die Verhaltensregeln sehr einfach spezifiziert und verschiedene Alternativen sehr schnell überprüft werden können.

Trotz dieser recht deutlichen Bestätigung der entwickelten Konzepte wurden bei der Erstellung der Umgebung `RAD` auch Schwächen der momentanen Realisierung des OCM deutlich. Zunächst ist der OCM derzeit bezüglich der unterstützten Zielsysteme noch zu eingeschränkt, was z.B. die Überwachung von `Codex` verhindert hat. An entsprechenden Anpassungen wird daher derzeit im Rahmen von mehreren Diplomarbeiten und einer Dissertation intensiv gearbeitet.[2] Insbesondere sollen dabei Anwendungen auf der Basis von MPI, Java und CORBA, threadbasierte Programme auf SMP-Systemen sowie Metacomputing-Anwendungen unterstützt werden.

Als deutliches Problem hat sich die rein sequentielle Realisierung der lokalen Monitorprozesse des OCM herausgestellt. Durch eine Implementierung mit mehreren Threads würde insbesondere das in § 6.2.1.2, Seite 162 erwähnte Problem bei der Durchführung eines *Inferior RPC* vereinfacht, daß nämlich beim Warten auf dessen Rückkehr gewisse andere Aktionen bearbeitet werden müssen. Geschieht dies nicht, sind Verklemmungen möglich, wenn eine Aktion in einem Werkzeug wiederum Anfragen an den OCM stellt. Andererseits dürfen keine Aktionen ausgeführt werden, die zu einem weiteren *Inferior RPC* in denselben Prozeß führen (vgl. das in § 8.1 erwähnte Wiedereintrittsproblem in `VISTOP`). Dies wäre erheblich einfacher zu erreichen, wenn jede Aktionsliste grundsätzlich durch einen eigenen Thread abgearbeitet würde. Der *Inferior RPC* muß dann lediglich durch ein Semaphor geschützt werden. Andererseits ist

[2] Kooperation mit Dr. Marian Bubak von der *University of Metallurgy and Mining*, Krakau, Polen; gefördert durch das BMBF, WTZ-Projekt POL 98/011.

damit zu rechnen, daß eine derartige Implementierung neue technische Probleme aufwirft, die zur Zeit nicht abzusehen sind.

Als hilfreich hätte sich auch ein erweitertes Objekt-Konzept der OMIS-Schnittstelle erwiesen, das Vererbung und Tokenkonvertierung konsistenter unterstützt. Das bisher realisierte einfache Vererbungsschema hat den Nachteil, daß im Gegensatz zur ursprünglichen OMIS-Spezifikation bei der Expansion eines Tokens die Zielklasse nicht mehr eindeutig ist. Dies ist am besten an einem Beispiel zu verdeutlichen: Angenommen, das Rahmenwerkzeug möchte die Beendigung der Einzelwerkzeuge der Umgebung überwachen. Eine Anfrage der Form

```
proc_has_terminated([a_]) : ...
```

kann diesen Zweck nicht erfüllen, da zunächst nicht klar ist, ob `a_` hier für „alle Werkzeuge“ oder „alle Prozesse“ stehen soll. Der OCM legt daher derzeit die mit OMIS konforme Bedeutung „alle Prozesse“ fest. Hier wäre eine Erweiterung wünschenswert, die bei der Token-Expansion eine Spezifikation der Zielklasse erlaubt. Ein allgemeineres Konzept wäre die Definition dynamischer Objektmengen über Eigenschaften der Objekte. Dies würde dann unter anderem auch den Aufruf von Aktionen in allen Werkzeugen mit bestimmten Eigenschaften erlauben, etwa die Hervorhebung eines Prozesses in allen Visualisierern einer Werkzeugumgebung.

8.3 Fazit

Alles in allem hat die Fallstudie `RAD` gezeigt, daß man trotz vieler Detailprobleme dem eingangs genannten Ideal letztendlich doch ziemlich nahe kommen kann: Wenn die Einzelwerkzeuge einige Grundvoraussetzungen erfüllen und eine geeignete Infrastruktur vorhanden ist, können sie zu beliebigen Umgebungen integriert werden, ohne daß sie dazu modifiziert werden müßten. Die gesamte notwendige Koordination und letztlich auch beliebige andere Erweiterungen der Einzelwerkzeuge[3] können zur Laufzeit durch ein relativ kleines Rahmenwerkzeug realisiert werden. Die dynamische Erweiterbarkeit ermöglicht dabei eine einfache, inkrementelle Integration, bei der zunächst problematische Konflikte aufgelöst und anschließend mehr und mehr komfortable Zusatzfunktionen realisiert werden.

Als Kritik könnte nun angeführt werden, daß trotz des modularen Aufbaus der Kern des OCM im Vergleich zu den eigentlichen Monitordiensten relativ groß ist. Hat man sich also die Möglichkeit interoperabler Werkzeuge durch einen großen, monolithischen Monitorsystem-Kern erkauft, im Endeffekt also gar nichts erreicht? Die Antwort auf diese Frage ist eindeutig „Nein“. Zum einen dient tatsächlich nur ein relativ kleiner Teil des Kerns ausschließlich der Unterstützung der Interoperabilität. Zum anderen könnten mit demselben Argument auch Betriebssysteme und jede andere Middleware in Frage gestellt werden. Sie stellen ebenfalls komplexe, zum Teil monolithische Software-Infrastrukturen dar, die dafür Sorge tragen, daß andere Programme unabhängig voneinander, aber in koordinierter Weise ausgeführt werden können. Ihre Berechtigung erhalten sie letztendlich dadurch, daß sie immer wieder benötigte Funktionen der Anwendungen kapseln, und dadurch deren Programmierung erheblich vereinfachen, genauso wie ein Monitorsystem dies für Laufzeit-Werkzeuge tut.

[3] Beispielsweise kann in der Umgebung `RAD` ohne größere Probleme der Knopf für das Rücksetzen in die Bedienoberfläche von `VISTOP` eingebaut werden, wie es in § 3.3.5 als Alternative genannt wurde.

Kapitel 9

Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war, zu untersuchen, wie und unter welchen Voraussetzungen mehrere unabhängig entwickelte Laufzeit-Werkzeuge gleichzeitig auf dasselbe Zielprogramm anwendbar sind und dabei konfliktfrei zusammenarbeiten. Die Aufrechterhaltung der Unabhängigkeit wurde dabei als fundamental erachtet: Die Werkzeuge sollen zwar ohne Modifikation ihres Quellcodes mit anderen Werkzeugen kooperieren können, jedoch nicht von diesen abhängig werden. Insbesondere müssen sie nach wie vor auch einzeln verwendbar sein. Für ihre Implementierung bedeutet dies, daß keinerlei Wissen über konkrete andere Werkzeuge notwendig sein bzw. ausgenutzt werden darf.

Es ist leicht einsichtig, daß diese Anforderungen nach einer wie auch immer gearteten, koordinierenden Infrastruktur verlangen. Sie muß zum einen die für die Kooperation notwendigen Interaktionen zwischen den Werkzeugen regeln. Zum anderen entstehen im Regelfall Konflikte zwischen simultan verwendeten Werkzeugen, die deren korrektes Funktionieren verhindern und deshalb aufgelöst werden müssen. Die vorliegende Arbeit hat detaillierte Anforderungen an eine solche Infrastruktur definiert, ein Entwurfskonzept und eine Realisierung erarbeitet und deren Nutzbarkeit anhand einer konkreten Werkzeugumgebung demonstriert.

9.1 Ergebnisse

Zunächst wurden anhand eines **Fallbeispiels** die notwendige Koordination zwischen den einzelnen Werkzeugen einer Werkzeugumgebung sowie die auftretenden Konflikte näher untersucht. Eine Verallgemeinerung dieser Resultate führte zu einer genaueren Klassifizierung und Modellierung. Dazu wurde die Werkzeugumgebung auf zwei verschiedenen Abstraktionsebenen betrachtet.

Eine Analyse der von der Umgebung auf der Ebene der Benutzerschnittstelle zu erbringenden Dienste deckt die erforderlichen **Interaktionen zwischen den Werkzeugen** auf. Von den dabei prinzipiell in Frage kommenden Koordinationsmodellen ist wegen der Forderung nach der Unabhängigkeit der Werkzeuge nur der **Mediator-Ansatz** geeignet. Er basiert auf einem Ereignis/Aktions-Modell, d.h. Ereignisse in einem Werkzeug lösen Aktionen in anderen Werkzeugen aus, wobei die Verknüpfung zwischen diesen durch den Mediator definiert wird. Neben diesen direkten Interaktionen werden auch indirekte erforderlich, bei denen das Ereignis nicht in einem Werkzeug, sondern im Zielsystem auftritt, wo es durch eine Aktivität eines Werkzeugs ausgelöst wurde.

Die Betrachtung auf der Realisierungsebene der Werkzeug führte zu einer **Methodik zur**

systematischen Ermittlung und Behandlung von Konflikten. Strukturelle Konflikte entstehen durch simultane Zugriffsversuche auf exklusive Betriebsmittel, die zwangsläufig fehlschlagen müssen. Diese Konflikte sind nur durch Nutzung von Schnittstellen vermeidbar, die die betroffenen Betriebsmittel für mehrere Nutzer zur Verfügung stellen. Die Realisierung dieser Schnittstellen kann durch ein universelles Monitorsystem erfolgen. Logische Konflikte treten auf, wenn Werkzeuge den Zielsystemzustand nicht nur beobachten, sondern auch modifizieren. Um sie systematisch analysieren zu können, wurde ein einfaches Modell für den Zielsystemzustand sowie für die von den Werkzeugen durchgeführten Zugriffe entwickelt. Das Modell unterscheidet Lese- und Schreibzugriffe auf Komponenten des Systemzustands, wobei die Zugriffe jeweils mit einer Zeitdauer versehen sind. Diese Zeitdauer beschreibt, wie lange ein Werkzeug davon ausgeht, daß die gelesene Information gültig bleibt, bzw. wie lange sich eine Modifikation auf das Zielsystem auswirkt. Konflikte sind dann als zeitlich überlappende Zugriffe zweier Werkzeuge A und B auf identische Zustandskomponenten charakterisiert. Dabei können drei Arten unterschieden werden:

1. Konsistenzprobleme treten auf, wenn B eine Komponente ändert, nachdem A diese gelesen hat und annimmt, daß sich ihr Wert nicht ändert.
2. Transparenzprobleme entstehen, falls B eine Komponente ändert, die Änderung von A aber nicht wahrgenommen werden darf.
3. Schreib/Schreib-Probleme entstehen, wenn A und B dieselbe Komponente modifizieren.

Während für das letztgenannte Problem keine allgemeine Strategie zur Behebung existiert, können die ersten beiden systematisch durch Benachrichtigungen über Schreibzugriffe und das Abfangen von Lesezugriffen behandelt werden. In speziellen Situationen sind sie auch von vorneherein vermeidbar, indem die Reihenfolge der Zugriffe geeignet festgelegt wird.

Unter Verwendung dieser Modellierung und der mit dem Fallbeispiel gewonnenen Erfahrung wurde eine allgemeine **Vorgehensweise zur Erstellung von Umgebungen interoperabler Werkzeuge** entwickelt. Dabei werden zunächst die notwendigen Interaktionen und Zusatzfunktionen auf der Ebene der Benutzerschnittstelle ermittelt. Sie ergeben sich aus Konsistenzanforderungen an dieser Schnittstelle und der erwünschten Kooperation zwischen den Werkzeugen. Anschließend werden die auf der Implementierungsebene entstehenden Konflikte analysiert und eine geeignete Konfliktbehandlung definiert. Die Realisierung der Interaktionen, der Konfliktbehandlung, sowie eventueller Zusatzfunktionen der Werkzeugumgebung erfolgt durch ein Rahmenwerkzeug, d.h. eine von den eigentlichen Werkzeugen getrennte Komponente. Sie kapselt alle Aspekte der Realisierung, die die Werkzeug*umgebung* betreffen und folgt damit der Leitidee des Mediator-Konzepts.

Aus der definierten Vorgehensweise folgt unmittelbar eine Reihe von **Anforderungen an eine unterstützende Infrastruktur**. Sie muß zunächst eine gemeinsam nutzbare Monitorschnittstelle bereitstellen, die strukturelle Konflikte bei Zugriffen auf das Zielsystem vermeidet. Dies ist eine Grundvoraussetzung dafür, daß überhaupt mehrere Werkzeuge simultan angewandt werden können. Logische Konflikte sind dadurch jedoch noch nicht ausgeschlossen. Die Infrastruktur muß daher zusätzliche Konzepte zur Behandlung dieser Konflikte realisieren. Dabei erwiesen sich folgende Mechanismen als notwendig:

1. die Steuerung der Reihenfolge von bedingten Aktionslisten,
2. Ereignismeldungen bei Modifikationen des Zielsystemzustands,
3. die Möglichkeit, Lesezugriffe auf den Zielsystemzustand abzufangen.

Schließlich muß die Infrastruktur auch noch die direkten und indirekten Interaktionen zwischen den Werkzeugen ermöglichen.

Auf der Basis dieses Anforderungskatalogs wurden zunächst existierende Monitorinfrastrukturen evaluiert. Da diese die Anforderungen derzeit nur unzureichend erfüllen, wurden ein Entwurf und eine Implementierung einer vollständigen **Infrastruktur für interoperable Laufzeit-Werkzeuge** erarbeitet. Als Ausgangspunkt für die Implementierung diente das Monitorsystem OCM, das bereits eine flexible Umsetzung des Ereignis/Aktions-Modells bot. Es wurde um Mechanismen zur Behandlung von Konflikten und zur Realisierung der Interaktionen zwischen Werkzeugen erweitert. Letztere wurden mit Hilfe dynamischer Instrumentierungs- und Bindetechniken so realisiert, daß benötigte Ereignis- und Aktions-Schnittstellen bei Bedarf zur Laufzeit in die Werkzeuge eingebunden werden können.

Die entwickelte Infrastruktur wurde schließlich in der Praxis eingesetzt, um die **Implementierung der im Fallbeispiel untersuchten Werkzeugumgebung** durchzuführen. Sie ermöglicht eine komfortable und effiziente Fehlersuche in parallelen Programmen mit hoher Ausführungszeit, indem sie eine schnelle Rückkehr zu beliebigen früheren Programmzuständen gestattet. Daneben erlaubte die Realisierung der Umgebung auch eine qualitative Bewertung der Infrastruktur und ihrer zugrundeliegenden Konzepte. Eine Bewertung der Werkzeugumgebung selbst durch ihren Einsatz an Anwendungen repräsentativer Größe konnte im dem begrenzten zeitlichen Rahmen dieser Arbeit jedoch noch nicht vorgenommen werden.

9.2 Bewertung

Der verfolgte Ansatz hat sich als praktikabel und erfolgreich erwiesen. Er erlaubt die Interoperation von Laufzeit-Werkzeugen, ohne daß Abhängigkeiten zwischen diesen entstehen und ohne daß sie auf die Verwendung einer bestimmten Middleware, wie z.B. CORBA oder ToolTalk festgelegt werden. Zudem müssen aufgrund der speziellen Realisierung der Werkzeug-Interaktionen die dazu benötigten Ereignis- und Aktions-Schnittstellen nicht bereits bei der Erstellung der Werkzeuge festgelegt werden. Vielmehr kann der hierfür notwendige Code in die Werkzeuge zur Laufzeit eingebaut werden. Sullivan selbst bezeichnet in seiner Originalarbeit über das Mediatorkonzept eine a-priori Festlegung dieser Schnittstellen als „unverhältnismäßig schwierig“ ([229], Kap. 8.10), sieht jedoch die hier verwendete Lösung nicht, da er Programme als statische, zur Laufzeit unveränderliche Objekte betrachtet. In der Praxis ermöglicht die vorliegende Arbeit damit erstmals die Erstellung von Werkzeugumgebungen auf Basis bereits existierender Werkzeuge, die nicht speziell für den Einsatz in diesen Umgebungen entwickelt wurden. Die Werkzeuge können dabei ohne Modifikation ihres Quellcodes wiederverwendet werden. Zusätzlicher Code wird nur in geringem Umfang in Form des Rahmenwerkzeugs erforderlich, das die erforderlichen Interaktionen zwischen den Werkzeugen vermittelt und zudem die Behandlung möglicher Konflikte steuert. Im Fallbeispiel besteht dieses Rahmenwerkzeug aus 700 Zeilen *C*-Code verglichen mit etwa 150.000 Zeilen der Werkzeuge selbst.

Die vorgeschlagene Vorgehensweise ist natürlich pragmatisch. Das dynamische Einbinden der notwendigen Schnittstellen wird propagiert, weil sie in existierenden Werkzeugen schlichtweg noch nicht vorhanden sind. Das bedeutet aber nicht, daß diese Vorgehensweise auch für den Entwurf neuer Werkzeuge bevorzugt werden sollte. Hier sollten vielmehr die für die Kooperation mit anderen Werkzeugen sinnvollen und vorhersehbaren Ereignis- und Aktions-Schnittstellen von Anfang an vorgesehen werden. Selbst in dem Fall, in dem dies vollständig gelingt, ist jedoch ein dynamisches Einbinden des Codes für das Weiterleiten der Ereignisse bzw. den entfernten Aufruf der Aktionen vorteilhaft, da dadurch eine Abhängigkeit der Werkzeuge von der Kommunikationsmiddleware vermieden wird.

Eine ad-hoc Zusammenstellung völlig beliebiger Werkzeuge zu einer funktionierenden Umgebung ist aber auch mit dem in dieser Arbeit entwickelten Ansatz nicht möglich. Die Werkzeuge müssen immer noch zumindest zwei Voraussetzungen erfüllen:

1. Die Monitorkomponenten der Werkzeuge dürfen keine strukturellen Konflikte aufweisen. Bis auf einzelne Ausnahmen heißt das derzeit, daß sie ein gemeinsames Monitorsystem verwenden müssen, wie es z.B. mit dem ursprünglichen OCM entwickelt wurde [270, 142]. Diese Voraussetzung stellt zur Zeit sicher noch das größte Hindernis für Umgebungen interoperabler Werkzeuge dar. Langfristig könnte diese Einschränkung mehr und mehr entfallen, wenn die für das Monitoring benötigten Schnittstellen in Betriebs- und Laufzeitsystemen verbessert werden, d.h. insbesondere eine simultane Nutzung durch mehrere Prozesse erlauben. Anzeichen für eine solche Entwicklung sind z.B. in der Verdrängung von `ptrace` durch das `/proc`-Dateisystem unter UNIX zu sehen.

2. Die Werkzeuge dürfen keine für ihre Funktion fundamentalen Annahmen beinhalten, die durch andere Werkzeuge verletzt werden können. Anderenfalls würde eine Zusammenarbeit nur mit massiven Änderungen in einem der Werkzeuge zu erreichen sein. Letztendlich bedeutet dies, daß bis zu einem gewissen Grad bereits bei der Entwicklung eines Werkzeugs das prinzipiell mögliche Vorhandensein anderer Werkzeuge bedacht werden muß.

Es sei angemerkt, daß es die entwickelte Infrastruktur technisch durchaus erlauben würde, selbst diese beiden Voraussetzungen durch entsprechende Modifikation des Werkzeugcodes zur Laufzeit herzustellen. Dazu müßte das Rahmenwerkzeug allerdings wesentliche Teile der Funktion eines Werkzeugs neu definieren, was keineswegs sinnvoll ist. Zudem widerspricht es dem Modularisierungsgedanken, nach dem das Rahmenwerkzeug nur Aspekte beinhalten soll, die mit der Werkzeugumgebung als ganzem, nicht aber mit einzelnen Werkzeugen verknüpft sind.

9.3 Weiterentwicklung

Die prinzipielle Tragfähigkeit der in der vorliegenden Arbeit entwickelten und implementierten Konzepte wurde bereits anhand einer relativ komplexen Werkzeugumgebung demonstriert. Bei der Weiterführung des Projekts sollen dennoch zunächst weitere praktische Erfahrungen gesammelt werden. Im Sinne eines Vorgehens, das die Anwender in den Mittelpunkt stellt, betrifft dies zuerst die Realisierung der konkreten Werkzeugumgebung aus dem Fallbeispiel. Sie muß zunächst anhand praxisrelevanter Anwendungen durch die Anwendungsentwickler erprobt und bewertet werden. Da in der Umgebung bisher nur wenige, repräsentative Interaktionen zwischen den Werkzeugen realisiert wurden, wird erwartet, daß bei ihrer Anwendung in der Praxis insbesondere Forderungen nach erweiterten Interaktionen laut werden. Als ein Beispiel seien werkzeugübergreifende Selektions- und Markierungsmechanismen für Prozesse in `DETOP`, `VISTOP` und `Codex` genannt. Die Realisierung derartiger neuer Funktionen der Umgebung wird dann wiederum zeigen, ob die entwickelten Konzepte zur Interoperabilität allgemein genug sind, oder ob wider Erwarten Erweiterungen notwendig werden.

Daneben sollen auch weitere Umgebungen mit anderen Werkzeugen realisiert werden. In Zusammenarbeit mit der *University of Mining and Metallurgy* in Krakau entstand beispielsweise ein auf dem OCM basierendes Online-Leistungsanalysesystem [24, 25, 19], das vorteilhaft mit dem Debugger `DETOP` kombiniert werden kann. Erste Versuche dazu, die allerdings noch nicht die hier entwickelte Infrastruktur verwenden, wurden bereits unternommen [18].

Vor einem noch breiteren Einsatz dieser Infrastruktur müßten jedoch die mir ihr verbundenen Sicherheitsaspekte sowie die rechtlichen Fragen der dynamischen Modifikation von Software geklärt werden. Sie wurden in dieser forschungsorientierten Arbeit außer Acht gelassen, in der die prinzipielle Machbarkeit im Vordergrund stand. Die Sicherheitsfragen betreffen primär das Verhindern von Abfragen bzw. Modifikationen des Zielsystemzustands – einschließlich der angeschlossenen Werkzeuge – durch nicht-autorisierte Benutzer. Dies ist insofern relativ einfach, da die betroffenen Dienste nur über die zentrale Schnittstelle des OCM zu erreichen sind. Im einfachsten Fall ist also eine Authentifizierung bei der Anmeldung, z.B. durch Kerberos [167] ausreichend, eventuell kombiniert mit der Verschlüsselung von Anfragen und Ergebnissen durch *Public-Key*-Verfahren. Ein differenzierterer Zugriffsschutz, bei dem für unterschiedliche Benutzer bzw. Werkzeuge festgelegt werden kann, welche Dienste sie ausführen dürfen, ist durch *Capabilities* [219] zu erreichen.

Längerfristig wäre zu untersuchen, inwieweit die beschriebene Methodik und die realisierte Infrastruktur neben Laufzeit-Werkzeugen auch die Interoperabilität anderer Softwarekomponenten unterstützen kann. In Frage kämen hier vor allem Werkzeugumgebungen für den computerunterstützten Software-Entwurf (*CASE*) [12, 232], bzw. auch für den Entwurf mechanischer oder elektronischer Produkte (*CAD,CAE*) [143]. In derartigen Szenarien ist zwar aufgrund des grundsätzlich verschiedenen Datenmodells die Datenintegration anders zu lösen, die Verhaltensintegration ist aber dennoch vorteilhaft mit den skizzierten, dynamischen Methoden realisierbar.

Letztendlich führt dies zu der Frage, die bereits in § 5.3.4 angeklungen ist, ob nämlich der entwickelte Ansatz die Grundlage für eine allgemeine, ereignisorientierte Middleware darstellen kann. Die fundamentalen Prinzipien einer derartigen Middleware wären das Mediator- bzw. Interaktorkonzept sowie das dynamische Einbinden von Schnittstellen und/oder Schnittstellenadaptern in die beteiligten Komponenten. Sie könnte vor allem eine Chance für das *Megaprogramming* [271] in verteilten, heterogenen und evolutionären Systemen darstellen.

9.4 Weiterführende Forschung

Bei der Durchführung dieses Forschungsprojekts wurden einige wissenschaftliche Fragestellungen aufgeworfen, die über die soeben skizzierte, geradlinige Weiterentwicklung hinausgehen. Sie konnten in dem zur Verfügung stehenden Zeitrahmen bisher nicht bearbeitet werden, stellen aber interessante, allgemeine Forschungsziele dar, für die die vorliegende Arbeit zumindest zum Teil bereits Grundlagen bereitstellen konnte.

Zunächst wurde in § 5.3.3.1, Seite 148 die Möglichkeit der automatischen Konflikterkennung und -behandlung skizziert. Sie wurde bisher nicht weiter verfolgt, da für das hier untersuchte Gebiet der Laufzeit-Werkzeuge der im Monitorsystem entstehende Mehraufwand für die Abarbeitung von Anfragen zu hoch wird. Bei dem oben angeregten Einsatz der Infrastruktur in anderen Anwendungsgebieten, in denen die Effizienzanforderungen nicht so hoch ausfallen, wäre eine derartige Fähigkeit aber von großem Vorteil. Die Herausforderung ist dabei eine mathematisch so exakte Definition des Endes von Zugriffen, daß eine systematische oder im Idealfall sogar automatische Erkennung und Meldung des Zugriffsendes erfolgen kann.

Für die Erstellung des Rahmenwerkzeugs einer Umgebung wären Abfragedienste günstig, die die in einem Werkzeug vorhanden Ereignis- und Aktions-Schnittstellen dokumentieren.

Notwendig werden derartige Dienste spätestens dann, wenn die derzeitige Infrastruktur zu einer allgemeinen Middleware ausgebaut werden soll. Einen Ansatz bietet z.B. *Java Beans* unter dem Schlagwort *Introspection* [66]. Über spezielle Klassen und Methoden kann dabei die Signatur aller verfügbaren Methoden einer *Bean* zur Laufzeit erfragt werden. Das eigentliche Problem ist allerdings nicht so sehr die Syntax der Schnittstelle, sondern die Semantik der angebotenen Methoden. Bei *Java Beans* ist diese nur für Lese- und Schreibmethoden für einfache Datenkomponenten einer *Bean* anhand einer Namenskonvention ersichtlich. Für die Erstellung einfacher Eigenschafts-Editoren, wie sie z.B. die *BeanBox* bereitstellt [230], ist dies zwar ausreichend, für eine Verhaltensintegration von Softwarekomponenten ist aber deutlich mehr Information über die Semantik erforderlich. Ob diese in maschinell bearbeitbarer und dennoch praktikabler Form spezifiziert werden kann, ist derzeit völlig offen.

Die Implementierung des zur Konfliktbehandlung benötigten Objektmodells des Zielsystems im OCM hat schließlich die Notwendigkeit dynamisch erweiterbarer Objektsysteme aufgezeigt. Herkömmliche objektorientierte Systeme ermöglichen zwar oft das dynamische Hinzufügen neuer, möglicherweise abgeleiteter Klassen und somit die Erzeugung erweiterter Objekte. Sie erlauben jedoch nicht die Erweiterung bereits existierender Objekte um neue Komponenten. Die letztere Eigenschaft eines Objektsystems wäre jedoch nicht nur für Monitorsysteme, sondern für jegliche andere Art von objektorientierter Software wünschenswert, die eine dynamische Erweiterbarkeit unterstützt. Als Beispiel sei das Fenstersystem X-Windows unter UNIX genannt, das freilich derzeit nicht mit Hilfe einer objektorientierten Sprache realisiert ist. In diesem System kann ein *Window Manager* die Fenster um neue Teile erweitern, z.B. um eine Titelleiste und um Schaltflächen zum Verschieben und Vergrößern des Fensters. In einer solchen Situation ist es mit Blick auf die Datenkonsistenz viel sinnvoller und auch eher dem objektorientierten Paradigma entsprechend, die notwendigen Informationen in neuen Komponenten der Objekte selbst zu speichern, als externe Verknüpfungen zwischen Objekten und dieser Information zu realisieren. Derartige dynamische Objektmodelle werden bereits vereinzelt untersucht, z.B. in der Sprache BETA [210]. Angesichts der stetig wachsenden Bedeutung adaptiver und erweiterbarer Softwaresysteme sollte diesem Thema jedoch mehr Aufmerksamkeit gewidmet werden.

Literaturverzeichnis

[1] F. Arbab, P. Ciancarini und C. Hankin. Coordination Languages for Parallel Programming. *Parallel Computing*, 24(7):989–1004, 1998.

[2] Requirements for Ada Programming Support Environments: STONEMAN. Ada Joint Program Office, US Dept. of Defence, Washington D.C., USA, Februar 1980. `http://www.adahome.com/History/Stoneman/`

[3] F. Arbab, I. Herman und P. Spilling. An Overview of Manifold and its Implementation. *Concurrency: Practice and Experience*, 5(1):23–70, Februar 1993.

[4] W. Akkerman. Strace Home Page. WWW-Seite der Univ. Leiden, Niederlande. `http://www.wi.leidenuniv.nl/˜wichert/strace/`

[5] F. Arbab. Specification of MANIFOLD, Version 1.0. Techn. Bericht CS-R9220, Centrum voor Wiskunde en Informatica, Amstardam, Niederlande, 1992. `http://www.cwi.nl/pub/CWIreports/IS/CS-R9220.ps.Z`

[6] A. V. Aho, R. Sethi und J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison Wesley, 1986. ISBN 0-201-10194-7.

[7] AT&T. *UNIX System V Programmer's Guide*, 1987. Prentice-Hall, Englewood Cliffs, NJ.

[8] R. A. Aydt. The Pablo Self-Defining Data Format. Techn. Bericht, Department of Computer Science, Univ. of Illinois, Urbana, März 1992. Zuletzt aktualisiert im Februar 2000. `ftp://vibes.cs.uiuc.edu/pub/Pablo.Release.5/SDDF/Documentation/SDDF.ps.gz`

[9] M. Ben-Ari. *Grundlagen der Parallel-Programmierung*. Hanser, 1984.

[10] P. Bates. Distributed Debugging Tools for Heterogeneous Distributed Systems. In *Proceedings of the 8th IEEE Conference on Distributed Computing Systems*, Seiten 308 – 315, San Jose, CA, USA, Juni 1988. IEEE-CS Press.

[11] R. Bagwill, J. Barkley et al. Security in Open Systems. Special Publication SP 800-7, National Technical Information Service (NTIS), Springfield, VA, USA, Juli 1994. `http://csrc.nist.gov/nistpubs/800-7/`

[12] A. W. Brown, D. J. Carney, E. J. Morris, D. B. Smith und P. F. Zarrella. *Principles of CASE Tool Integration*. Oxford University Press, 1994.

[13] F. Bodin, Z. Chamski, E. Rohou und A. Seznec. *Functional Specification of* SALTO*: A Retargetable System for Assembly Language Transformation and Optimization*. INRIA, Rennes, Frankreich, 1997.
`http://www.irisa.fr/caps/PROJECTS/Salto/PUBLIS/SALTO.PS`

[14] M. Beck, J. J. Dongarra, G. E. Fagg, G. A. Geist, P. Gray, J. Kohl, M. Migliardi, K. Moore, T. Moore, P. Papadopoulous, S. L. Scott und V. Sunderam. HARNESS: A Next Generation Distributed Virtual Machine. *Future Generation Computer Systems*, 15(5/6), 1999.
`http://www.netlib.org/utk/people/JackDongarra/PAPERS/harness2.ps`

[15] R. Borgeest, B. Dimke und O. Hansen. A Trace Based Performance Evaluation Tool for Parallel Real Time Systems. *Parallel Computing*, 21(4):551–564, April 1995.

[16] S. Browne, J. Dongarra und K. London. Review of Performance Analysis Tools for MPI Parallel Programs. WWW-Seite der University of Tennessee, Knoxville, TN, USA.
`ttp://www.cs.utk.edu/˜browne/perftools-review/`

[17] T. Bemmerl. *Programmierung skalierbarer Multiprozessoren*. Habilitationsschrift, Fakultät für Mathematik und Informatik, Technische Universität München, Mai 1991.

[18] M. Bubak, W. Funika, B. Baliś und R. Wismüller. Interoperability of OCM-based On-line Tools. In J. Dongarra, P. Kacsuk und N. Podhorszki, Herausgeber, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proc. 7th European PVM/MPI Users Group Meeting*, Band 1908 aus *Lecture Notes in Computer Science*, Seiten 242–249, Balatonfüred, Ungarn, September 2000. Springer-Verlag.

[19] M. Bubak, W. Funika, B. Baliś und R. Wismüller. Performance Measurement Support for MPI Applications with PATOP. In *Proc. PARA 2000 Workshop on Applied Parallel Computing*, Bergen, Norwegen, Juni 2000.
`http://www.ii.uib.no/para2000/program/marian.ps`

[20] M. Bubak, W. Funika, B. Baliś und R. Wismüller. Towards Extension of an OCM-based Tool Environment. In M. Bubak, J. Mościński und M. Noga, Herausgeber, *Proc. SGI Users' Conference*, Seiten 263–271, Krakau, Polen, Oktober 2000. CYFRONET AGH.
`http://wwwbode.in.tum.de/˜wismuell/pub/sgi00.ps.gz`

[21] M. Bubak, W. Funika, B. Baliś und R. Wismüller. On-line OCM-based Tool Support for Parallel Applications. In C.-K. Yuen, Herausgeber, *Annual Review of Scalable Computing*, Band 3, Kapitel 2. World Scientific Publishing Co. und Singapore University Press, 2001. Im Druck.

[22] M. Bubak, W. Funika, R. Gembarowski, P. Hodurek und R. Wismüller. Enhancing OCM to support MPI applications. In P. Sloot, M. Bubak, A. Hoekstra und B. Hertzberger, Herausgeber, *High Performance Computing and Networking, Proc. of HPCN'99*, Band 1593 aus *Lecture Notes in Computer Science*, Seiten 1274–1277, Amsterdam, Niederlande, April 1999. Springer-Verlag.
`http://wwwbode.in.tum.de/˜wismuell/pub/hpcn99a.html`

[23] M. Bubak, W. Funika, R. Gembarowski und R. Wismüller. OMIS-compliant monitoring system for MPI applications. In R. Wyrzykowski et al., Herausgeber, *Proc. 3rd International Conference on Parallel Processing and Applied Mathematics - PPAM'99*, Seiten 378–386, Kazimierz Dolny, Polen, September 1999. Technical University of Czestochowa.
`http://wwwbode.in.tum.de/˜wismuell/pub/ppam99.ps.gz`

[24] M. Bubak, W. Funika, K. Iskra, R. Maruszewski und R. Wismüller. Enhancing the Functionality of Performance Measurement Tools for Message Passing Environments. In J. Dongarra, E. Luque und T. Margalef, Herausgeber, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proc. 6th European PVM/MPI Users Group Meeting*, Band 1697 aus *Lecture Notes in Computer Science*, Seiten 67–74, Barcelona, Spanien, September 1999. Springer-Verlag.
`http://wwwbode.in.tum.de/˜wismuell/pub/europvm99.html`

[25] M. Bubak, W. Funika und R. Wismüller. Experience with Development of Performance Monitoring Tools for Parallel Applications. In R. Tadeusiewicz, S. Białas, T. Szmuc und M. Szymkat, Herausgeber, *Computer Methods and Systems in Research and Engineering, CCATIE'99*, Seiten 369–372, Krakau, Polen, Oktober 1999.
`http://wwwbode.in.tum.de/˜wismuell/pub/ccatie99.ps.gz`

[26] M. Bubak, W. Funika, D. Zbik, D. v. Albada, K. Iskra, P. Sloot, R. Wismüller und K. Sowa-Pieklo. Performance Measurement, Debugging and Load Balancing for Metacomputing. In *Proc. ISThum 2000 – Research and Development for the Information Society*, Seiten 409–418, Poznan, Polen, April 2000.
`http://wwwbode.in.tum.de/˜wismuell/pub/poznan.ps.gz`

[27] B. Brügge, T. Gottschalk und B. Luo. A Framework for Dynamic Program Analyzers. In *Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'93*, Band 28(10) aus *ACM SIGPLAN Notices*, Seiten 65–82, Washington D.C., USA, September 1993.

[28] P. Brezany, S. Grabner, K. Sowa und R. Wismüller. DeHiFo – An Advanced HPF Debugging System. In *7th Euromicro Workshop on Parallel and Distributed Processing, PDP'99*, Seiten 226–232, Madeira, Portugal, Februar 1999. IEEE CS Press.

[29] J. A. Bergstra und P. Klint. The TOOLBUS Coordination Architecture. In P. Ciancarini und C. Hankin, Herausgeber, *Coordination : languages and models : First International Conference COORDINATION '96*, Band 1061 aus *Lecture Notes in Computer Science*, Seiten 75–88, Cesena, Italien, April 1996. Springer-Verlag.

[30] A. Bode, W. Karl, T. Ludwig und R. Wismüller. Monitoring Technologies for Parallel On-Line Tools. In *SFB 342 Final Colloquium: Methods and Tools for the Efficient Use of Parallel Systems*, Seiten 1–29. Technische Universität München, München, August 2000.
`http://wwwbode.in.tum.de/˜wismuell/pub/sfb00.ps.gz`

[31] P. Becker und R. Müller. Achieving Interoperability between the Distributed Model-Manament-Systems MMM and Progress. In *Proc. 4th ISDSS Conference*, Lausanne,

Schweiz, Juli 1997.
http://mmm.wiwi.hu-berlin.de/˜rmueller/paper/matse-interop.html

[32] A. Bode. Werkzeuge zur Entwicklung paralleler Programme. In K. Waldschmidt, Herausgeber, *Parallelrechner: Architekturen - Systeme - Werkzeuge*, Kapitel 13, Seiten 517–562. B. G. Teubner, Stuttgart, 1995. ISBN 3-519-02135-8.

[33] P. Braun. *Visualisierung des Ablaufverhaltens paralleler Programme*. Doktorarbeit, Fakultät für Informatik, Technische Universität München, Dezember 1994.
http://wwwbode.in.tum.de/archiv/disserta/braun/diss.ps.gz

[34] S. Browne. Cross-Platform Parallel Debugging and Performance Analysis Tools. In V. Alexandrov und J. Dongarra, Herausgeber, *Recent Advances in Parallel Virtual Machine and Message Passing Interface – 5th European PVM/MPI Users' Group Meeting*, Band 1497 aus *Lecture Notes in Computer Science*, Seiten 257–264, Liverpool, UK, September 1998. Springer-Verlag.

[35] B. Brügge. A Portable Plattform for Distributed Event Environments. In B. P. Miller und C. E. McDowell, Herausgeber, *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, May 20-21, 1991*, Band 26(12) aus *ACM SIGPLAN Notices*, Seiten 184–193, Santa Cruz, CA, USA, Dezember 1991.

[36] I. Ben-Shaul, A. Cohen, O. Holder und B. Lavva. HADAS: A Network-Centric Framework for Interoperability Programming. *Intl. J. of Cooperative Information Systems*, 6(3&4):293–314, 1997.

[37] D. Beringer, C. Tornabene, P. Jain und G. Wiederhold. A Language and System for Composing Autonomous, Heterogeneous and Distributed Megamodules. In *DEXA International Workshop on Large-Scale Software Composition*, Wien, Österreich, August 1998.
http://www-db.stanford.edu/CHAIMS/Doc/Papers/DEXA/Composition_DEXA_1998.ps

[38] T. Bemmerl und R. Wismüller. On-line Distributed Debugging on Scalable Multiprocessor Architectures. In W. Gentzsch und U. Harms, Herausgeber, *High-Performance Computing and Networking, Proceedings, Volume II: Networking and Tools*, Band 797 aus *Lecture Notes in Computer Science*, Seiten 394–400, München, April 1994. Springer-Verlag.
http://wwwbode.in.tum.de/˜wismuell/pub/hpcn94.ps.gz

[39] T. Bemmerl und R. Wismüller. On-line Distributed Debugging on Scalable Multiprocessor Architectures. *Future Generation Computer Systems*, 11(4-5):375–385, November 1995.
http://wwwbode.in.tum.de/˜wismuell/pub/fgcs95.ps.gz

[40] P. Braun und R. Wismüller. Visualization of Parallel Program Execution. In A. Bode, T. Ludwig, V. Sunderam und R. Wismüller, Herausgeber, *Workshop on PVM, MPI, Tools, and Applications*, Seiten 33–43, München, November 1995. Technische Universität München. Bericht Nr. TUM-I9535, SFB-Bericht Nr. 342/18/95 A.
http://wwwbode.in.tum.de/˜wismuell/pub/vis95.ps.gz

[41] J.C. Cunha und V. Duarte. Monitoring PVM Programs Using the DAMS Approach. In *Recent Advances in Parallel Virtual Machine and Messag Passing Interface, Proc. EuroPVM/MPI'98*, Band 1497 aus *Lecture Notes in Computer Science*, Seiten 273–280, Liverpool, UK, September 1998. Springer-Verlag.

[42] C. Clémencon, A. Endo, J. Fritscher, A. Müller, R. Rühl und B. J. N. Wylie. The "Annai" Environment for Portable Distributed Parallel Programming. In *Proc. of the 28th Annual Hawaii International Conference on System Sciences, Volume II*, Seiten 242–251. IEEE, Januar 1995.
`ftp://ftp.cscs.ch/pub/CSCS/papers/hicss28.ps.gz`

[43] C. Clémençon, A. Endo, J. Fritscher, A. Müller, R. Rühl und B. J. N. Wylie. Annai: An integrated parallel programming environment for multicomputers. In A. Zaky und T. Lewis, Herausgeber, *Tools and Environments for Parallel and Distributed Systems*, Kapitel 2, Seiten 33–59. Kluwer Academic Publishers, Februar 1996.
`ftp://ftp.cscs.ch/pub/CSCS/papers/pdtepds.ps.gz`

[44] C. Clémençon, A. Endo, J. Fritscher, A. Müller und B. J. N. Wylie. Annai Scalable Run-time Supprot for Interactive Debugging and Performance Analysis of Large-scale Parallel Programs. In *Proc. 2nd International Euro-Par Conference, Volume I*, Band 1123 aus *Lecture Notes in Computer Science*, Seiten 64–69. Springer-Verlag, August 1996.
`ftp://ftp.cscs.ch/pub/CSCS/papers/EuroPar96.ps.gz`

[45] The Joint CSCS-ETH/NEC Collaboration in Parallel Processing. WWW-Seite des Centro Svizzero di Calcolo Scientifico, Manno, Schweiz.
`http://www.cscs.ch/Official/Project_CSCS-NEC.html`

[46] Welcome to CSCS. WWW-Heimatseite des Centro Svizzero di Calcolo Scientifico, Manno, Schweiz.
`http://www.cscs.ch`

[47] F. Carter und A. Fekete. Cerberus - A Tools for Debugging Distributed Algorithms. In *Proc. IFIP Workshop on Software Engineering for Parallel and Distributed Systems*, Seiten 110–121, Berlin, März 1996. Chapman and Hall publishers.

[48] J. Cownie und W. Gropp. A Standard Interface for Debugger Access to Message Queue Information in MPI. In J. Dongarra, E. Luque und T. Margalef, Herausgeber, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proc. 6th European PVM/MPI Users Group Meeting*, Band 1697 aus *Lecture Notes in Computer Science*, Seiten 51–58, Barcelona, Spanien, September 1999. Springer-Verlag.
`ftp://info.mcs.anl.gov/pub/tech_reports/reports/P754.ps.Z`

[49] D. Cheng und R. Hood. A Portable Debugger for Parallel and Distributed Programs. In *Proc. Supercomputing'94*, Seiten 723–732, Washington D.C., USA, November 1994. IEEE.
`http://wk122.nas.nasa.gov:80/NAS/Tools/Projects/P2D2/sc94.ps`

[50] P. Ciancarini und C. Hankin, Herausgeber. *Coordination Languages and Models – First International Conference, COORDINATION '96*, Band 1061 aus *Lecture Notes in Computer Science*, Cesena, Italien, April 1996. Springer-Verlag.

[51] B. Chapman. The ESPRIT Project PPPE – Project Summary. Bericht der Universität Wien, Österreich.
http://www.vcpc.univie.ac.at/activities/projects/PPPE/ps/MoreInfo.ps

[52] J. C. Cunha, J. Lourenço und T. Antão. A Debugging Engine for a Parallel and Distributed Environment. In *Proc. 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems, DAPSYS'96*, Miscolc, Ungarn, Oktober 1996.

[53] J. C. Cunha, J. Laurenço und V. Duarte. Tool Integration Issues for Parallel and Distributed Debugging. In *Proc. 3rd SEIHPC Workshop*, Madrid, Spanien, Januar 1998.
ftp://ftp.cpc.wmin.ac.uk/pub/seihpc/madrid/unl.ps.gz

[54] J. C. Cunha, J. Louren co, J. Vieira, B. Moscão und D. Pereira. A Framework to Support Parallel and Distributed Debugging. In P. Sloot, M. Bubak und B. Hertzberger, Herausgeber, *High-Performance Computing and Networking – Proceedings HPCN'98*, Band 1401 aus *Lecture Notes in Computer Science*, Seiten 708–717, Amsterdam, Niederlande, April 1998. Springer-Verlag.
http://www-asc.di.fct.unl.pt/˜jml/papers/1998/hpcn98.ps.gz

[55] J. Cargille und B. P. Miller. Binary Wrapping: A Technique for Instrumenting Object Code. *ACM SIGPLAN Notices*, 27(6):17–18, Juni 1992.

[56] Cray Research Inc. *UNICOS CDBX Debugger User's Guide*, 1996. Cray Publication SG-2094 8.0.

[57] D. E. Culler und J. P. Singh. *Parallel Computer Architecture – A Hardware/Software Approach*. Morgan Kaufman Publishers, 1999. ISBN 1-55860-343-3.

[58] P. Ciancarini und A.L. Wolf, Herausgeber. *Coordination Languages and Models – Third International Conference, COORDINATION '99*, Band 1594 aus *Lecture Notes in Computer Science*, Amsterdam, Niederlande, April 1999. Springer-Verlag.

[59] P. Davies. *The Indispensable Guide to C : With Engineering Applications*. Addison Wesley, 1995. ISBN 0-201-62438-9.

[60] J. Dongarra, G. Fagg, A. Geist, J. A. Kohl, P. M. Papadopoulos, S. L. Scott, V. Sunderam und M. Magliardi. HARNESS: Heterogeneous Adaptable Reconfigurable NEtworked SystemS. In *The Seventh IEEE International Symposium on High Performance Distributed Computing*, Chicago, IL, USA, Juli 1998. IEEE.
http://dlib.computer.org/conferen/hpdc/8579/pdf/85790358.pdf

[61] S. K. Damodaran-Kamal und J. M. Francioni. *mdb*: A Semantic Race Detection Tool for PVM. In *Proc. of SHPCC'94, Scalable High Performance Computing Conference*, Seiten 702–709, Knoxville, TN, USA, Mai 1994.

[62] J. C. de Kergommeaux, M. Ronsse und K. De Bosschere. MPL*: Efficient Record/Replay of Nondeterministic Features of Message Passing Libraries. In J. Dongarra, E. Luque und T. Margalef, Herausgeber, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proc. 6th European PVM/MPI Users Group Meeting*, Band 1697 aus

Lecture Notes in Computer Science, Seiten 141–148, Barcelona, Spanien, September 1999. Springer-Verlag.

[63] J. Dongarra, S. W. Otto, M. Snir und D. Walker. An Introduction to the MPI Standard. Technical Report CS-95-274, Univ. of Tennessee, Knoxville, TN, USA, Januar 1995.
`http://www.netlib.org/tennessee//ut-cs-95-274.ps`

[64] European Computer Manufacturers Association (ECMA). Portable Common Tool Environment (PCTE) – Abstract Specification. Standard ECMA-149, 4. Ausgabe, Dezember 1997.
`http://www.ecma.ch/stand/ECMA-149.HTM`

[65] European Computer Manufacturers Association (ECMA). Portable Common Tool Environment (PCTE) – C Programming Language Binding. Standard ECMA-158, 4. Ausgabe, Dezember 1997.
`http://www.ecma.ch/stand/ECMA-158.HTM`

[66] G. Hamilton (Ed.). *Java Beans, Version 1.01*. Sun Microsystems, Mountain View, CA, USA, Juli 1997.
`http://java.sun.com/products/javabeans/docs/beans.101.ps`

[67] A. Einstein. *Relativity – The Special and the General Theory*. Wing Books, 1961. ISBN 0-517-02961-8.

[68] Etnus, Inc. WWW-Heimatseite der Etnus Inc, Framingham, MA, USA.
`http://www.etnus.com/`

[69] TotalView Multiprocess Debugger. WWW-Seite der Etnus Inc., Framingham, MA, USA, 1999.
`http://www.etnus.com/products/totalview/index.html`

[70] EuroTools Tools Database. WWW-Seite der ESPRIT Arbeitsguppe EuroTools.
`http://www.irisa.fr/cgi-bin/cgiwrap/pazat/pazat_dir/swish/tools/Tools.pl`

[71] T. Fahringer. The Weight Finder – An Advanced Profiler for Fortran Programs. In *Automatic Parallelization, New Approaches to Code Generation, Data Distribution, and Performance Prediction*, Vieweg Advanced Studies in Computer Science, Seiten 7–31, Wiesbaden, März 1993. Vieweg Verlag.
`http://www.par.univie.ac.at/~tf/papers/p3t/wf.ps.gz`

[72] S. I. Feldman und C. B. Brown. IGOR: A System for Program Debugging via Reversible Execution. In *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, Band 24(1) aus *ACM SIGPLAN Notices*, Seiten 112–123, Madison, WI, USA, Mai 1988. ACM.
`http://www.acm.org/pubs/articles/proceedings/onr/68210/p112-feldman/p112-feldman.pdf`

[73] T. Fahringer, M. Gerndt, G. Riley und J. L. Träff. Knowledge Specification for Automatic Performance Analysis – APART Technical Report. Techn. Bericht, ESPRIT IV Working

Group on Autommatic Performance Analysis, November 1999.
http://www.par.univie.ac.at/˜tf/apart/wp2/report-wp2.ps.gz

[74] G. E. Fisher. A CSL View of Applications, Portability, Scalability, and Interoperability. *StandardView*, 4(2):107, Juni 1996.

[75] M. Frey und M. Oberhuber. Testing and Debugging Parallel and Distributed Programs with Temporal Logic Specifications. In *Proc. of 2nd Workshop on Parallel and Distributed Software Engineering 1997*, Seiten 62–72, Boston, MA, USA, Mai 1997.
http://wwwbode.in.tum.de/˜oberhube/PUBS/PS/pdse97.ps.gz

[76] APART – Esprit Working Group on Automatic Performance Analysis: Resources and Tools. WWW-Seite des Forschungszentrums Jülich.
http://www.fz-juelich.de/apart/

[77] Index of Base Modules and Related Work. WWW-Seite des Forschungszentrums Jülich.
http://www.fz-juelich.de/apart/wp3/modules.html

[78] The MPI Forum. MPI: A Message-Passing Interface Standard. Techn. Bericht Version 1.1, Univ. of Tennessee, Knoxville, TN, USA, Juni 1994.
http://www.mpi-forum.org/docs/mpi-11.ps.Z

[79] GNU Emacs. WWW-Seite der Free Software Foundation, Boston, MA, USA.
http://www.gnu.org/software/emacs/emacs.html

[80] International Organization for Standardization. Industrial automation systems and integration – Product data representation and exchange – Part 1: Overview and fundamental principles. ISO Standard 10303-1:1994, 1994.

[81] J. Fenlason und R. Stallman. *GNU gprof – The GNU Profiler*. Free Software Foundation, November 1998.
http://www.gnu.org/manual/gprof-2.9.1/gprof.html

[82] J. Gait. A Probe Effect in Concurrent Programs. *Software – Practice and Experience*, 16(3):225–233, März 1986.

[83] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck und V. Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[84] D. Gelernter und N. Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97–107, Februar 1992.

[85] ADAPTOR – GMD's High Performance Fortran Compilation System. WWW-Seite der GMD - Forschungszentrum Informationstechnik GmbH, St. Augustin.
http://www.gmd.de/SCAI/lab/adaptor/adaptor_home.html

[86] W. Gu, G. Eisenhauer, K. Schwan und J. Vetter. Falcon: On-line Monitoring and Steering of Parallel Programs. *Concurrency: Practice and Experience*, 10(9):699–736, August 1998.
http://www.cc.gatech.edu/systems/papers/GuCPE.ps

[87] I. Glendinning, V. S. Getov, S. A. Hellberg, R. W. Hockney und D. J. Pritchard. Performance Visualisation in a Portable Parallel Programming Environment. Techn. Bericht, Department of Electronics and Computer Science, The University of Southampton, Southampton, UK, 1993.
`ftp://ftp.ecs.soton.ac.uk/pub/benchmarks/pvpppe.ps.Z`

[88] A. Geist, J. Kohl und P. Papadopoulos. Visualization, Debugging, and Performance in PVM. In M. L. Simmons, A. H. Hayes, J. S. Brown und D. A. Reed, Herausgeber, *Debugging and Performance Tuning for Parallel Computing Systems*, Seiten 65 – 77. IEEE CS Press, 1996.
`http://www.epm.ornl.gov:80/~geist/CapeCod.ps`

[89] D. Garlan und D. Le Metayer, Herausgeber. *Coordination Languages and Models – Second International Conference, COORDINATION '97*, Band 1282 aus *Lecture Notes in Computer Science*, Berlin, September 1997. Springer-Verlag.

[90] E. Haseloff. Metastabiles Verhalten von TTL-Bausteinen. *Elektronik*, (3):63–66, Februar 1986.

[91] R. Hood und D. Cheng. Accomodating Heterogeneity in a Debugger - A Client-Server Approach. In *Proc. of the 28th Annual Hawaii International Conference on System Sciences, Volume II*, Seiten 252–253. IEEE, Januar 1995.
`http://wk122.nas.nasa.gov:80/NAS/Tools/Projects/P2D2/hicss.ps`

[92] M. T. Heath und J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8(5):29–39, September 1991.

[93] M. T. Heath und J. E. Finger. ParaGraph: A Performance Visualization Tool for MPI. WWW-Seite der University of Illinois at Urbana-Champaign, IL, USA.
`http://www.csar.uiuc.edu/software/paragraph/`

[94] D. Heller und P. M. Ferguson. *Motif Programming Manual*, Band 6A aus *The Definitive Guides to the X Window System*. O'Reilly, Februar 1994. ISBN 1-565-92016-3.

[95] C. T. Howie, J. C. Kunz und K. H. Law. Software Interoperability. Techn. Bericht, Center for Integrated Facility Engineering, Stanford University, Stanford, CA, November 1996.
`http://www.dacs.dtic.mil/techs/interop/title.shtml`

[96] R. Hockauf, W. Karl, M. Leberecht, M. Oberhuber und M. Wagner. Exploiting Spatial and Temporal Locality of Accesses: A New Hardware-Based Monitoring Approach for DSM Systems. In D. Pritchard und J. Reeve, Herausgeber, *Euro-Par'98 Parallel Processing, 4th International Euro-Par Conference*, Band 1470 aus *Lecture Notes in Computer Science*, Seiten 206–215, Southampton, UK, September 1998. Springer-Verlag.

[97] J. K. Hollingsworth, B. P. Miller und J. Cargille. Dynamic Program Instrumentaiton for Scalable Performance Tools. In *Proc. Supercomputing'94*, Seiten 841–850, Washington D.C., USA, November 1994. IEEE.
`ftp://grilled.cs.wisc.edu/technical_papers/dyninst.ps.Z`

[98] J. R. Hollingsworth, B. P. Miller, M. J. R. Gonçalves, Z. Xu, O. Naim und L. Zheng. MDL: A Language and Compiler for Dynamic Program Instrumentation. In *Proc. International Conference on Parallel Architectures and Compilation Techniques*, San Francisco, CA, USA, November 1997.
`ftp://grilled.cs.wisc.edu/technical_papers/mdl.ps.gz`

[99] J. Hollingsworth. An Application Program Interface for Runtime Code Patching. Techn. Bericht, Computer Science Department, Univ. of Maryland, College Park, MD, USA, März 1996.
`http://www.cs.umd.edu/~hollings/papers/dyninstAPI.ps`

[100] R. Hood. The *p2d2* Project: Building a Portable Distributed Debugger. In *Proc. of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, Seiten 127–136, Philadelphia, PA, USA, Mai 1996. ACM Press.

[101] A. H. Hayes, M. L. Simmons, J. S. Brown und D. A. Reed. *Debugging and Performance Tuning for Parallel Computing Systems*. IEEE CS Press, Los Alamitos, CA, 1996.

[102] IEEE Standard Glossary of Software Engineering Terminology. The Institute of Electrical and Electronics Engineers Inc. (IEEE), New York, NY, USA, 1990. IEEE Standard 610.12-1990.

[103] B. Irvin und B. P. Miller. The Super-Symbol Table Project. Briefing im Rahmen des *1995 ARPA CSTO Joint PI Meeting*, 1995.
`http://www.cs.wisc.edu/~paradyn/CSTOtalk/index.html`

[104] The Salto Project. WWW-Seite des INRIA Forschungsinstituts, Rennes, Frankreich.
`http://www.irisa.fr/caps/PROJECTS/Salto/SALTO.HTM`

[105] Intel Corporation, Mt. Prospect, IL, USA. *Pentium Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual*, 1995. ISBN 1-55512-247-7.

[106] J. Irion. High Performance Computing Applications for Industrial Embedded Systems. In P. P. Spies, Herausgeber, *EuroArch-93, Europäischer Informatik-Kongress Architektur von Rechensystemen*, München, Oktober 1993. Springer-Verlag.

[107] ISO/IEC. Information technology – Portable Common Tool Environment (PCTE) – Part 1: Abstract Specification. Standard ISO/IEC 13719-1:1998, Oktober 1998.

[108] ISO/IEC. Information technology – Portable Common Tool Environment (PCTE) – Part 2: C programming language binding. Standard ISO/IEC 13719-2:1998, Oktober 1998.

[109] A. M. Julienne und B. Holtz. *ToolTalk & Open Protocols: Inter-Application Communication*. SunSoft Press, 1994. ISBN 0-13-031055-7.

[110] GUP Linz. WWW-Seite der Johannes Kepler Universität Linz, Österreich.
`http://www.gup.uni-linz.ac.at:8001`

[111] MAD – Monitoring And Debugging environment. WWW-Seite der Johannes Kepler Universität Linz, Österreich.
`http://www.gup.uni-linz.ac.at:8001/research/debugging/mad/`

[112] C. Royster Jr. DoD Strategy on Open Systems and Interoperability. *StandardView*, 4(2):104–106, Juni 1996.

[113] P. Kacsuk. Macrostep-by-macrostep Debugging of Message Passing Parallel Programs. In *Proc. Parallel and Distributed Computing and Systems PDCS'98*, Seiten 527–531, Las Vegas, USA, 1998.

[114] P. Kacsuk, G. Dózsa, T. Fadgyas und R. Lovas. GRADE: A Graphical Programming Environment for Multicomputers. *Computers and Artificial Intelligence*, 17(5):417–427, 1998.

[115] P. Kacsuk, G. Dózsa, T. Fadgyas und R. Lovas. The GRED Graphical Editor for the GRADE Parallel Program Development Environment. In P. Sloot, M. Bubak und B. Hertzberger, Herausgeber, *High-Performance Computing and Networking – Proceedings HPCN'98*, Band 1401 aus *Lecture Notes in Computer Science*, Seiten 728–738, Amsterdam, Niederlande, April 1998. Springer-Verlag.

[116] J. A. Kohl und G. A. Geist. The PVM 3.4 Tracing Facility and XPVM 1.1. In H. El-Rewini und B. D. Shriver, Herausgeber, *Proc. 29th Hawaii International Conference on System Sciences (HICSS-29)*, Band 1, Seiten 290–299, Maui, Hawaii, Januar 1996.

[117] D. Kranzlmüller, S. Grabner und J. Volkert. Event Graph Visualization for Debugging Large Applications. In *Proc. of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, Seiten 108–117, Philadelphia, PA, USA, Mai 1996. ACM Press.

[118] R. Koppler, S. Grabner und J. Volkert. Visualization of Distributed Data Structures for HPF-like Languages. *Scientific Programming*, 6(1):115–126, 1997.

[119] D. Kranzlmüller, S. Grabner und J. Volkert. Debugging with the MAD Environment. *Parallel Computing*, 23(1-2):199–217, April 1997.

[120] W. Kim, Herausgeber. *Modern Database Systems: The Object Model, Interoperability, and Beyond*. ACM Press, 1995.

[121] P. Kacsuk, R. Lovas und J. Kovacs. Systematic Debugging of Parallel Programs in DIWIDE Based on Collective Breakpoints and Macrosteps. In *Proc. 19th Intl. Conf. on Software Engineering*, Seiten 90–97, Toulouse, Frankreich, August 1999.

[122] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr. und M. E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1993. ISBN 0-262-61094-9.

[123] J. Kohl. XPVM: A Graphical Console and Monitor for PVM. WWW-Seite der University Tennessee, Knoxville, TN, USA.
`http://www.netlib.org/utk/icl/xpvm/xpvm.html`

[124] J. M. Kewley und R. Prodan. A Distributed Object-Oriented Framework for Tool Development. In Q. Li, D. Firesmith, R. Riehle, G. Pour und B. Mayer, Herausgeber, *Proc. 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 34)*, Seiten 353–362, Santa Barbara, USA, Juli 2000. IEEE Computer Society Press.
`http://www.cscs.ch/~kewley/pubs/TOOLS2000.ps.gz`

[125] J. Krammer. Performance Analysis of HPF Programs. In A. Bode, Herausgeber, *Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM) – Annual Report 1995*, Band 3 aus *LRR TUM Research Report Series*, Kapitel 2.9, Seiten 43–48. Shaker-Verlag, Aachen, Juni 1996. ISBN 3-8265-1809-8.

[126] D. Kranzlmüller und J. Volkert. Debugging Point-To-Point Communication in MPI and PVM. In *Recent Advances in Parallel Virtual Machine and Messag Passing Interface, Proc. EuroPVM/MPI'98*, Band 1497 aus *Lecture Notes in Computer Science*, Seiten 265–272, Liverpool, UK, September 1998. Springer-Verlag.

[127] D. Kranzlmüller und J. Volkert. NOPE: A Nondeterministic Program Evaluator. In P. Zinterhof, M. Vajtersic und A. Uhl, Herausgeber, *Parallel Computation, 4th International ACPC Conference*, Band 1557 aus *Lecture Notes in Computer Science*, Seiten 490–499, Salzburg, Österreich, Februar 1999. Springer-Verlag.
`http://link.springer.de/link/service/series/0558/bibs/1557/15570490.htm`

[128] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, Juli 1978.

[129] J. Lourenço und J. C. Cunha. The PDBG Process-level Debugger for Parallel and Distributed Programs. In *Proc. 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, Seite 154, Welches, OR, USA, August 1998. ACM Press.
`http://www-asc.di.fct.unl.pt/~jml/papers/1998/spdt98.ps.gz`

[130] M. Lenke. *Ein Gestaltungskonzept für den Anwendungszyklus massiv paralleler Strömungssimulationsalgorithmen*, Band 4 aus *LRR-TUM Research Report Series*. Shaker-Verlag, Aachen, Oktober 1996. ISBN 3-8265-1858-6.

[131] J. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers, 1999. ISBN 1-55860-496-0.
`http://iecc.com/linker`

[132] M. Lindermeier. *Konzepte zur Lastverwaltung in verteilten, objektorientierten Systemen*. Doktorarbeit, Fakultät für Informatik, Technische Universität München, 2001. In Vorbereitung.

[133] F. Long und E. Morris. An Overview of PCTE: A Basis for a Portable Common Tool Environment. Techn. Bericht CMU/SEI-93-TR-1, Software Engineering Institute, Carnegie Mellon Univ., Pittsburgh, PA, USA, März 1993.
`http://www.sei.cmu.edu/pub/documents/93.reports/pdf/tr01.93.pdf`

[134] T. J. LeBlanc und J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, C-35(4):471–481, April 1987.

[135] P. Luksch, U. Maier, S. Rathmayer, M. Weidmann, F. Unger, P. Bastian, V. Reichenberger und A. Haas. SEMPA – Software Engineering Methods for Parallel Applications in Scientific Computing. Abschlußbericht des BMBF-Projekts SEMPA, Technische Universität München, August 1998.

[136] T. Ludwig, M. Oberhuber und R. Wismüller. An Open Monitoring System for Parallel and Distributed Programs. In L. Bougé, P. Fraigniaud, A. Mignotte und Y. Robert, Herausgeber, *Euro-Par'96 – Parallel Processing*, Band 1123 aus *Lecture Notes in Computer Science*, Seiten 78–83, Lyon, Frankreich, August 1996. Springer-Verlag.
`http://wwwbode.in.tum.de/~wismuell/pub/europar96.ps.gz`

[137] J. Livingston und K. Spohrer. DWARF: A Debugging Standard. In *Proceedings of the Supercomputing Debugging Workshop 1991*, Albuquerque, NM, USA, November 1991.

[138] J. R. Larus und E. Schnarr. EEL: Machine-Independent Executable Editing. In *SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, Band 30(6) aus *ACM SIGPLAN Notices*, Seiten 291–300, La Jolla, CA, USA, Juni 1996. ACM.
`ftp://ftp.cs.wisc.edu/wwt/pldi95_eel.ps`

[139] M. Litzkow, T. Tannenbaum, J. Basney und M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Environment. Techn. Bericht 1346, Comp. Sci. Dept., Univ. of Wisconsin-Madison, Madison, WI, USA, April 1997.
`http://www.cs.wisc.edu/condor/doc/ckpt97.ps`

[140] T. Ludwig, J. Trinitis und R. Wismüller. Synergetic Tool Environments. In V. Malyshkin, Herausgeber, *Parallel Computing Technologies, Proc. PaCT'99*, Band 1662 aus *Lecture Notes in Computer Science*, Seiten 248–257, St. Petersburg, Russland, September 1999. Springer-Verlag.
`http://wwwbode.in.tum.de/~wismuell/pub/pact99.html`

[141] T. Ludwig. *Automatische Lastverwaltung für Parallelrechner*, Band 94 aus *Reihe Informatik*. B.I. Wissenschaftsverlag, 1993. ISBN 3-411-16501-4.

[142] T. Ludwig. *Methoden zur effizienten Entwicklung paralleler Werkzeuge*. Habilitationsschrift, Technische Universität München, April 1998.
`http://wwwbode.in.tum.de/~ludwig/LN/papers/tl9804/PS/book.ps.gz`

[143] P. Luksch. *Increased Productivity in Computational Prototyping with the Help of Parallel and Distributed Computing*, Band 21 aus *LRR-TUM Research Report Series*. Shaker-Verlag, Aachen, April 2000. ISBN 3-8265-8021-4.

[144] T. Ludwig und R. Wismüller. OMIS 2.0 – A Universal Interface for Monitoring Systems. In M. Bubak, J. Dongarra und J. Wasniewski, Herausgeber, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proc. 4th European PVM/MPI Users' Group Meeting*, Band 1332 aus *Lecture Notes in Computer Science*, Seiten 267–276, Krakau, Polen, November 1997. Springer-Verlag.
`http://wwwbode.in.tum.de/~wismuell/pub/europvm97b.ps.gz`

[145] T. Ludwig, R. Wismüller, R. Borgeest, S. Lamberts, C. Röder, G. Stellner und A. Bode. THE TOOL-SET – An Integrated Tool Environment for PVM. In J. Dongarra, M. Gengler, B. Touracheau und X. Vigouroux, Herausgeber, *Proceedings of EuroPVM'95 Short Papers*, Lyon, Frankreich, September 1995. Ecole Normale Supérieure de Lyon. Technischer Bericht 95-02.
`http://wwwbode.in.tum.de/~wismuell/pub/europvm95.ps.gz`

[146] T. Ludwig, R. Wismüller und M. Oberhuber. OCM — A Monitoring System for PVM. In H. Matthies und J. Schüle, Herausgeber, *Paralleles und Verteiltes Rechnen — 4. Workshop über Wissenschaftliches Rechnen*, Seiten 107–114, Aachen, Oktober 1996. Shaker-Verlag.
`http://wwwbode.in.tum.de/~wismuell/pub/wwr96.ps.gz`

[147] T. Ludwig, R. Wismüller und M. Oberhuber. OCM – An OMIS Compliant Monitoring System. In A. Bode, J. Dongarra, T. Ludwig und V. Sunderam, Herausgeber, *Parallel Virtual Machine – EuroPVM'96*, Band 1156 aus *Lecture Notes in Computer Science*, Seiten 81–90, München, Oktober 1996. Springer-Verlag.
`http://wwwbode.in.tum.de/~wismuell/pub/europvm96b.ps.gz`

[148] T. Ludwig, R. Wismüller, M. Oberhuber und A. Bode. An Open Interface for the On-Line Monitoring of Parallel and Distributed Programs. *Intl. Journal of Supercomputer Applications and High Performance Computing*, 11(2):160–174, Sommer 1997.

[149] T. Ludwig, R. Wismüller, V. Sunderam und A. Bode. OMIS – On-line Monitoring Interface Specification. Techn. Bericht TUM-I9609, SFB-Bericht Nr. 342/05/96 A, Technische Universität München, Februar 1996.
`http://wwwbode.in.tum.de/~omis/OMIS/Version-1.0/version-1.0.ps.gz`

[150] T. Ludwig, R. Wismüller, V. Sunderam und A. Bode. *OMIS — On-line Monitoring Interface Specification (Version 2.0)*, Band 9 aus *LRR-TUM Research Report Series*. Shaker-Verlag, Aachen, 1997. ISBN 3-8265-3035-7.
`http://wwwbode.in.tum.de/~omis/OMIS/Version-2.0/version-2.0.ps.gz`

[151] E. Maillet. Issues in Performance Tracing with Tape-PVM. In *EuroPVM'95*, Band 5 aus *Parallèlisme, réseaux et répartition*, Seiten 143–148, Lyon, Frankreich, September 1995. Hermès, Paris.

[152] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam und T. Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11), November 1995.
`http://www.cs.wisc.edu/paradyn/papers/overview.ps.gz`

[153] W. Mehl. Deutsche Bahn erwägt, Schadensersatz zu fordern – Siemens-Rechner legt Stellwerk in Hamburg für zwei Tage lahm. *Computerwoche*, (12):27, 24. März 1995.
`http://www.computerwoche.de/info-point/heftarchiv/index.cfm?id=16000&cfid=552886&cftoken=61822906&nr=1`

[154] C. E. McDowell und D. P. Helmbold. Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4):593–622, Dezember 1989.

[155] Distributed Component Object Model (DCOM). WWW-Seite der Microsoft Corporation, Redmond, USA.
`http://www.microsoft.com/com/tech/DCOM.asp`

[156] J. Menapace, J. Kingdon und D. MacKenzie. *The "stabs" Debug Format*. Cygnus Support, Oktober 1999.
`http://www.delorie.com/gnu/docs/gdb/stabs_toc.html`

[157] A. Malony und S. Shende. Performance Technology for Complex Parallel and Distributed Systems. In G. Kotsis und P. Kacsuk, Herausgeber, *Distributed and Parallel Systems: From Concepts to Applications – Proc. Third Austrian-Hungarian Workshop on Distributed and Parallel Systems, DAPSYS 2000*, Seiten 37–46, Balatonfüred, Ungarn, 2000. Kluwer.
`http://www.cs.uoregon.edu/research/paracomp/papers/dapsys2k.ps.gz`

[158] GRADE – A Graphical Parallel Programming Environment. WWW-Seite des MTA SZTAKI Forschungsinstituts, Budapest, Ungarn.
`http://www.lpds.sztaki.hu/projects/grade/grade.html`

[159] MTA SZTAKI Laboratory of Parallel and Distributed Systems. WWW-Seite des MTA SZTAKI Forschungsinstituts, Budapest, Ungarn.
`http://www.lpds.sztaki.hu/`

[160] NAS Home Page. WWW-Heimatseite der N.A.Software Ltd, Liverpool, UK.
`http://www.nasoftware.co.uk/`

[161] PTLib Parallel Tools Library. WWW-Seite der National HPCC Software Exchange (NHSE).
`http://www.nhse.org/ptlib/`

[162] The STEP Project. WWW-Seite des National Institute of Standards and Technology, Gaithersburg, MD, USA.
`http://www.nist.gov/sc4/`

[163] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe und K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer 63*, 12(1):69–80, Januar 1996.
`http://www.kfa-juelich.de/zam/PTdocs/vampir/vampir.html`

[164] S. G. Nekolla, T. Ibrahim, T. Balbach, C. Klein, F. Bengel und M. Schwaiger. Coregistration and Fusion of Cardiac MRI and PET Studies. In P. Marzullo, Herausgeber, *Proc. NATO Advanced Research Workshop "Understanding Cardiac Imaging Techniques from Basic Pathology to Image Fusion"*, Pisa, Italien, Dezember 2000. CNR Institute of Clinical Physiology, Pisa, IOS Press. Im Druck.

[165] R. H. B. Netzer und B. P. Miller. Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs. In *Proc. Supercomputing'92*, Seiten 502–511, Minneapolis, MN, USA, November 1992. IEEE CS Press.

[166] R. H. B. Netzer und B. P. Miller. What are Race Conditions? Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, März 1992.

[167] B. C. Neuman und T. Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications*, 32(9):33–38, September 1994.
`http://www.isi.edu/gost/publications/kerberos-neuman-tso.html`

[168] M. Oberhuber. Benutzergesteuerte Elimination des Nichtdeterminismus in parallelen Programmen. Diplomarbeit, Fakultät für Informatik, Technische Universität München, 1993.

[169] M. Oberhuber. *Methoden zur kontrollierten Ausführung nichtdeterministischer kommunizierender Programme*, Band 13 aus *LRR-TUM Research Report Series*. Shaker-Verlag, Aachen, März 1999. ISBN 3-8265-4791-8.

[170] Object Management Group Home Page. WWW-Seite der Object Management Group Inc., Framingham, MA, USA.
`http://www.omg.org/`

[171] Object Management Group Inc., Framingham, MA, USA. *CORBA 2.3.1 / IIOP Specification*, Oktober 1999.
`http://www.omg.org/library/c2indx.html`

[172] TAU Monitoring Framework. WWW Seite der Universität von Oregon, Eugene, OR, USA.
`http://www.cs.uoregon.edu/research/paracomp/tau/monitor/`

[173] TAU: Tuning and Analysis Utilities. WWW Seite der Universität von Oregon, Eugene, OR, USA.
`http://www.cs.uoregon.edu/research/paracomp/proj/tau/`

[174] M. Oberhuber und R. Wismüller. *DETOP Users Manual*. Technische Universität München, November 1994.
`http://wwwbode.in.tum.de/˜wismuell/pub/detopman.ps.gz`

[175] M. Oberhuber und R. Wismüller. DETOP - An Interactive Debugger for PowerPC Based Multicomputers. In P. Fritzson und L. Finmo, Herausgeber, *Parallel Programming and Applications*, Seiten 170–183. IOS Press, Amsterdam, Mai 1995.
`http://wwwbode.in.tum.de/˜wismuell/pub/zeus95.ps.gz`

[176] Pallas Products: PARMACS and TraceGen. WWW-Seite der Pallas GmbH, Brühl.
`http://www.pallas.de/pages/parmacs.htm`

[177] Pallas Products: VAMPIR. WWW-Seite der Pallas GmbH, Brühl.
`http://www.pallas.de/pages/vampir.htm`

[178] Welcome to Pallas. WWW-Heimatseite der Pallas GmbH, Brühl.
`http://www.pallas.de/`

[179] C. Pancake. Supercomputing'95 Ptools Questionnaire on Parallel Tools. WWW-Seite der Oregon State University, OR, USA, 1996.
`http://www.cs.orst.edu/˜pancake/surveys/sc95quest.html`

[180] Dynamic Probe Class Library. WWW-Seite des Parallel Tools Consortium.
`http://www.ptools.org/projects/dpcl/`

[181] MQM Working Group Home Page. WWW-Seite des Parallel Tools Consortium.
`http://www.ptools.org/projects/mqmi/`

[182] Parallel Tools Projects Around the World. WWW-Seite des Parallel Tools Consortium.
`http://www.ptools.org/projects.world.html`

[183] *Proc. 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, Welches, OR, USA, August 1998.

[184] D. Pase. Dynamic Probe Class Library (DPCL): Tutorial and Reference Guide, Version 0.1. Techn. Bericht, IBM Corp., Poughkeepsie, NY, USA, 1998.
`http://www.ptools.org/projects/dpcl/tutref.ps`

[185] D. Pase. Dynamic Probe Class Library (DPCL): Class Reference Manual, Version 0.3. Techn. Bericht, IBM Corp., Poughkeepsie, NY, USA, 1999.
`http://www.ptools.org/projects/dpcl/ref.ps`

[186] S. Petri. *Lastausgleich und Fehlertoleranz in Workstation-Clustern*. Shaker-Verlag, Aachen, 1997. ISBN 3-8265-2471-3.

[187] R. Prodan und J. M. Kewley. FIRST: A Framework for Interoperable Resources, Services, and Tools. In *Proc. of PDPTA'99*, Las Vegas, NV, USA, Juni 1999.

[188] R. Prodan und J. M. Kewley. A Framework for an Interoperable Tool Environment. In A. Bode, T. Ludwig, W. Karl und R. Wismüller, Herausgeber, *Euro-Par 2000 – Parallel Processing*, Band 1900 aus *Lecture Notes in Computer Science*, Seiten 65–69, München, August 2000. Springer-Verlag.
`http://link.springer.de/link/service/series/0558/papers/1900/19000065.pdf`

[189] D. Z. Pan und M. A. Linton. Supporting reverse execution for parallel programs. In *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, Band 24(1) aus *ACM SIGPLAN Notices*, Seiten 124–129, Madison, WI, USA, Mai 1988. ACM.
`http://www.acm.org/pubs/articles/proceedings/onr/68210/p124-pan/p124-pan.pdf`

[190] W. W. Plummer. Asynchronous Arbiters. *IEEE Transactions on Computers*, (C21):37–42, 1972.

[191] G. Paul und K.-U. Sattler. MediatorService - Integration von verteilten Objekten durch Beschreibung der Interaktionen. In *Proc. Int. Workshop Trends in Distributed Systems*, Aachen, Oktober 1996.
`http://wwwiti.cs.uni-magdeburg.de/~sattler/papers/trevs.ps`

[192] G. Paul, K. Sattler und M. Endig. An Integration Framework for Open Tool Environments. In H. König, K. Geihs und T. Preuß, Herausgeber, *Distributed Applications and Interoperable Systems*, Seiten 193–200, Cottbus, September 1997. IFIP, Chapman & Hall.
`http://wwwiti.cs.uni-magdeburg.de/~sattler/papers/dais97.ps`

[193] L. Perrochon, G. Wiederhold und R. Burback. A Compiler for Composition: CHAIMS. In *Fifth International Symposium on Assessment of Software Tools and Technologies (SAST'97)*, Pittsburgh, PA, USA, Juni 1997.
`http://www-db.stanford.edu/CHAIMS/Doc/Papers/97sast/97sast.ps`

[194] R. Rabenseifner. Automatic Profiling of MPI Applications with Hardware Performance Counters. In J. Dongarra, E. Luque und T. Margalef, Herausgeber, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proc. EuroPVM/MPI'99*, Band 1697 aus *Lecture Notes in Computer Science*, Seiten 35–42, Barcelona, Spanien, September 1999. Springer-Verlag.
`http://www.hlrs.de/people/rabenseifner/publ/profiling_pvmmpi99_29.ps.gz`

[195] G. Rackl. Multi-Layer Monitoring in Distributed Object-Environments. In L. Kutvonen, H. König und M. Tienari, Herausgeber, *Distributed Applications and Interoperable Systems II – IFIP TC 6 WG 6.1 Second International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99)*, Seiten 265–270, Helsinki, Finnland, June 1999. Kluwer Academic Publishers.

[196] G. Rackl. *Monitoring and Managing Heterogeneous Middleware*. Doktorarbeit, Fakultät für Informatik der Technischen Universität München, 2001. In Vorbereitung.

[197] PureCoverage: Code Coverage Analysis Tool, Automated Software Testing, UNIX. WWW-Seite der Rational Software Corporation, Cupertino, CA, USA.
`http://www.rational.com/products/purecoverage/index.jtmpl`

[198] Purify: Fast Detection of Memory Leaks and Access Errors. Whitepaper der Rational Software Corporation, Cupertino, CA, USA.
`http://www.rational.com/products/purify_unix/prodinfo/whitepapers/dynamic.jtmpl?doc_key=319`

[199] M. Raynal. Logical Instantaneity and Causal Order: Two "First Class" Communication Modes for Parallel Computing. In A. Bode, T. Ludwig, W. Karl und R. Wismüller, Herausgeber, *Euro-Par 2000: Parallel Processing – 6th International Euro-Par Conference*, Band 1900 aus *Lecture Notes in Computer Science*, Seiten 35–42, München, August 2000. Springer-Verlag.

[200] William Rubin und Marshall Brain. *Understanding DCOM*. Prentice Hall, 1998. ISBN 0-13-095966-9.

[201] High Performance Fortran. WWW-Seite der Rice University, Houston, TX, USA.
`http://www.crpc.rice.edu/HPFF/home.html`

[202] S. Rathmayer und M. Lenke. A Tool for On-Line Visualization and Interactive Steering of Parallel HPC Applications. In *11th International Parallel Processing Symposium*, Seiten 181–186, Genf, Schweiz, 1997. IEEE Computer Society.

[203] R. R. Raje, Z. Liu, S. Chinnasamy, J. Williams, W. Mascarenhas und M. Zhong. Distributed-Object Computing. In R. Buyya, Herausgeber, *High performance cluster computing – Volume 2 : Programming and applications*, Kapitel 12, Seiten 249–273. Prentice Hall, 1999.

[204] C. Röder. *Load Management Techniques in Distributed Heterogeneous Systems*, Band 11 aus *LRR-TUM Research Report Series*. Shaker-Verlag, Aachen, September 1998. ISBN 3-8265-4151-0.

[205] J. B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley & Sons, 1996.

[206] S. Rottegger. Entwicklung eines Visualisierungswerkzeugs für PVM. Diplomarbeit, Fakultät für Informatik, Technische Universität München, Mai 1995.

[207] S. Sistare, D. Allen, R. Bowker, K. Jourdenais, J. Simons und R. Title. A Scalable Debugger for Massively Parallel Message-Passing Programs. In *Proc. of SHPCC'94, Scalable High Performance Computing Conference*, Seiten 825–832, Knoxville, TN, USA, Mai 1994.

[208] K.-U. Sattler. *Tool-Komposition in integrierten Entwurfsumgebungen*. Doktorarbeit, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, 1998. Erschienen als Fortschritt-Bericht Nr. 543 im VDI-Verlag, Düsseldorf.

[209] S. P. Schach. *Software Engineering*. Aksen Associates, 2 Ausgabe, 1993.

[210] R. W. Schmidt. Dynamically Extensible Objects in a Class-Based Language. In *Proc. 23rd Technology of Object Oriented Languages and Systems*, Santa Barbara, CA, USA, Juli 1997. IEEE Computer Society Press.
`http://home4.inet.tele.dk/rws/papers/dynslotpaper.htm`

[211] J. Siegel. *CORBA Fundamentals and Programming*. John Wiley & Sons, 1996. ISBN 0471-12148-7.

[212] Developer Central - ProDev Workshop. WWW-Seite der Silicon Graphics Inc., Mountain View, CA, USA.
`http://www.sgi.com/developers/devtools/tools/prodev.html`

[213] K. J. Sullivan, I.J. Kalet und D. Notkin. Evaluating the Mediator Method: Prism as a Case Study. *IEEE Transactions on Software Engineering*, 22(8):563–579, August 1996. auch Techn. Bericht CS-95-29 der Univ. Virginia, Charlottesville, VA, USA.
`ftp://ftp.cs.virginia.edu/pub/techreports/CS-95-29.ps.Z`

[214] R. Schwarz und F. Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing*, 7(3):149–174, 1994.
`http://www.inf.ethz.ch/department/IS/vs/publ/papers/holygrail.ps`

[215] S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman und S. Karmesin. Portable Profiling and Tracing for Parallel, Scientific Applications using C++. In *Proc. 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, Seiten 134–145, Welches, OR, USA, August 1998. ACM Press.

[216] T. Sterling, P. Messina und J. Pool. Findings of the Second Pasadena Workshop on System Software and Tools for High Performance Computing Environments. Techn. Bericht, Center of Excellence in Space Data and Information Sciences, NASA Goddard Space Flight Center, Greenbelt, MD, USA, 1995.
`http://hpcf.nersc.gov/about/NUG/workinp/greenbook/node78.html`

[217] T. Sheehan, A. Malony und S. Shende. A Runtime Monitoring Framework for the TAU Profiling System. In *Proc. 3rd Intl. Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'99)*, San Francisco, CA, USA, Dezember 1999.
`http://www.cs.uoregon.edu/research/paraducks/papers/iscope99.ps.gz`

[218] G. Stellner und J. Pruyne. Resource Management and Checkpointing for PVM. In *EuroPVM'95*, Band 5 aus *Parallèlisme, réseaux et répartition*, Seiten 131–136, Lyon, Frankreich, September 1995. Hermès, Paris.

[219] J. H. Saltzer und M. D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.
`http://www.mediacity.com/~norm/CapTheory/ProtInf/`

[220] C. Scheidler und L. Schäfers. TRAPPER: A Graphical Programming Environment for Industrial High-Performance Applications. In A. Bode, M. Reeve und G. Wolf, Herausgeber, *Proc. PARLE'93, Parallel Architectures and Languages Europe*, Band 694 aus *Lecture Notes in Computer Science*, Seiten 403–413, München, Juni 1993. Springer-Verlag.

[221] G. Stellner und J. Trinitis. Load Balancing Based on Process Migration for MPI. In C. Lengauer, M. Griebl und S. Gorlatch, Herausgeber, *Euro-Par'97 – Parallel Processing, Proceedings*, Band 1300 aus *Lecture Notes in Computer Science*, Seiten 150 – 157, Passau, August 1997. Springer-Verlag.

[222] CHAIMS: Compiling High-level Access Interfaces for Multi-site Software. WWW-Seite der Stanford University, CA, USA.
`http://www-db.stanford.edu/CHAIMS/`

[223] Dataflows in CHAIMS. WWW-Seite der Stanford University, CA, USA.
`http://www-db.stanford.edu/CHAIMS/Doc/Details/controldata.html`

[224] R. M. Stallman. *Debugging with GDB: The GNU Source-Level Debugger for GDB version 4.18.* Free Software Foundation, 3 Ausgabe, Februar 1999. ISBN 1-88211-476-0.

[225] G. Stellner. *Methoden zur Sicherungspunkterzeugung in parallelen und verteilten Systemen*, Band 2 aus *LRR-TUM Research Report Series*. Shaker-Verlag, Aachen, November 1996. ISBN 3-8265-1808-X.

[226] W. R. Stevens. *UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI.* Prentice Hall, zweite Ausgabe, 1998. ISBN 0-13-490012-X.
`http://www.kohala.com/start/unpv12e.html`

[227] W. R. Stevens. *UNIX Network Programming, Volume 2: Interprocess Communications*. Prentice Hall, zweite Ausgabe, 1999. ISBN 0-13-081081-9.
`http://www.kohala.com/start/unpv22e/unpv22e.html`

[228] M. Stitt. *Debugging: Creative Techniques and Tools for Software Repair.* Wiley Professional Computing. John Wiley & Sons, 1992.

[229] K. J. Sullivan. *Mediators: Easing the Design and Evolution of Integrated Systems*. Doktorarbeit, Dept. of Computer Sciences and Engineering, Univ. of Washington, USA, 1994. Techn. Bericht 94-08-01.
`ftp://ftp.cs.washington.edu/tr/1994/08/UW-CSE-94-08-01.PS.Z`

[230] JavaBeans – The BeanBox. WWW Seite von Sun Microsystems, Inc., Palo Alto, CA, USA.
`http://java.sun.com/products/javabeans/software/beanbox.html`

[231] Sun Microsystems, Inc., Mountain View, CA, USA. *ToolTalk User's Guide, Revision A*, Juni 1995.
`http://www.unix.digital.com/faqs/publications/base_doc/DOCUMENTATION/V50_HTML/SUPPDOCS/TT_UGD/TITLE.HTM`

[232] D. Schefström und G. van den Broek, Herausgeber. *Tool Integration: Environments and Frameworks*. John Wiley & Sons, 1993. ISBN 0-471-93554-9.

[233] I. Thomas. PCTE Interfaces: Supporting Tools in Software-Engineering Environments. *IEEE Software*, Seiten 15–23, November 1989.

[234] A. Tamches und B. P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Third Symposium on Operating Systems Design and Implementation (OSDI'99)*, Seiten 117–130, New Orleans, LA, USA, Februar 1999. USENIX.
`http://www.cs.wisc.edu/paradyn/papers/kerninst.ps.gz`

[235] J. Trinitis. *Interoperability of Distributed Checkpointing and Debugging Tools*, Band 19 aus *LRR-TUM Research Report Series*. Shaker-Verlag, Aachen, 2000. ISBN 3-8265-1858-6.

[236] J. Trinitis, V. Sunderam, T. Ludwig und R. Wismüller. Interoperability Support in Distributed On-line Monitoring Systems. In M. Bubak, H. Afsarmanesh, R. Williams und B. Hertzberger, Herausgeber, *High Performance Computing and Networking, 8th International Conference, HPCN Europe 2000*, Band 1823 aus *Lecture Notes in Computer Science*, Seiten 261–269, Amsterdam, Niederlande, Mai 2000. Springer-Verlag.

[237] R. H. Thyler und M. C. Thyler. Glossary of Software Engineering Terms. In R. H. Thyler und A. P. McGettrick, Herausgeber, *Software Engineering: A European Perspective*, Seiten 587–648. IEEE CS Press, 1993.

[238] Programming Languages and Compiler Technology. WWW-Seite der Universität Wien, Österreich.
`http://www.par.univie.ac.at/research/lang-comp/lang-comp.html`

[239] The PPPE Project Homepage. WWW-Seite der Universität Wien, Österreich.
`http://www.vcpc.univie.ac.at/activities/projects/PPPE/`

[240] SDDF: Pablo Project Data Metaformat. WWW-Seite der Univ. of Illinois, Urbana, USA.
`http://vibes.cs.uiuc.edu/Project/SDDF/SDDFOverview.htm`

[241] An Application Program Interface (API) for Runtime Code Generation. WWW-Seite der University of Maryland, College Park, USA.
`http://www.cs.umd.edu/projects/dyninstAPI/`

[242] EEL: An Executable Editing Library. WWW-Seite der University of Wisconsin, Madison, USA.
`http://www.cs.wisc.edu/~larus/eel.html`

[243] UNIX International Programming Languages SIG, Parsippany, NJ, USA. *DWARF Debugging Information Format, Revision 2.0.0*, Juli 1993.
`ftp://sgigate.sgi.com/pub/dwarf/dwarfFeb97.tar.Z`

[244] University of Wisconsin-Madison, Madison, WI, USA. *Condor Version 6.1.15 Manual*, August 2000.
`http://www.cs.wisc.edu/condor/manual/v6.1/ref.html`

[245] A. H. Veen und M. de Lange. Overview of the PREPARE Project. PREPARE Deliverable ACE-2026-ProjectOverview, ACE Assoviated Computer Experts bv, Amsterdam, Niederlande, Oktober 1993.

[246] C. Viravan. *Enhancing Debugging Technology*. Doktorarbeit, Purdue University, West Lafayette, IN, USA, März 1994.
`http://www.serc.net/TechReports/files/TR151P.PS`

[247] K. Waldschmidt, Herausgeber. *Parallelrechner: Architekturen - Systeme - Werkzeuge*. B. G. Teubner, Stuttgart, 1995. ISBN 3-519-02135-8.

[248] A. Wasserman. Tool Integration in Software Engineering Environments. In F. Long, Herausgeber, *Software Engineering Environments*, Band 467 aus *Lecture Notes in Computer Science*, Seiten 137–149. Springer-Verlag, 1990.

[249] R. Wismüller, G. Dózsa und D. Drótos. Using OMIS for On-line Monitoring in the GRADE Programming Environment. In *Proc. Austrian-Hungarian Workshop on Distributed and Parallel Systems, DAPSYS'98*, Seiten 177–184, Budapest, Ungarn, September 1998. Institut für angewandte Informatik und Informationssysteme, Universität Wien.
`http://wwwbode.in.tum.de/~wismuell/pub/dapsys98.ps.gz`

[250] R. Wismüller, G. Dózsa und D. Drótos. Enhanced Monitoring in the GRADE Programming Environment by using OMIS. *Future Generation Computer Systems*, 16(6):637–648, April 2000.
`http://www.elsevier.nl/gej-ng/10/19/19/41/29/31/article.pdf`

[251] Peter Wegner. Tutorial Notes: Models and Paradigms of Interaction. Techn. Bericht CS-95-21, Department of Computer Science, Brown University, Providence, RI, USA, September 1995.
`http://www.cs.brown.edu/publications/techreports/reports/CS-95-21.html`

[252] A. Weinand, E. Gamma und R. Marty. ET++ – An Object Oriented Application Framework in C++. In *OOPSLA'88 Conference Proceedings*, Band 23(11) aus *ACM SIGPLAN Notices*, Seiten 168–182, San Diego, CA, USA, September 1988.

[253] A. Weinand, E. Gamma und R. Marty. Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured Programming*, 10(2):63–87, 1989. http://www.ubs.com/e/index/about/ubilab/ext/publications/e_wei89.htm

[254] R. Wismüller. *Quellsprachorientiertes Debugging von optimierten Programmen*. Reihe Informatik. Shaker-Verlag, Aachen, 1995. ISBN 3-8265-0841-6. http://www.shaker.de/Online-Gesamtkatalog/Details.idc?ISBN=3-8265-0841-6

[255] R. Wismüller. State Based Visualization of PVM Applications. In A. Bode, J. Dongarra, T. Ludwig und V. Sunderam, Herausgeber, *Parallel Virtual Machine – EuroPVM'96*, Band 1156 aus *Lecture Notes in Computer Science*, Seiten 91–99, München, Oktober 1996. Springer-Verlag. http://wwwbode.in.tum.de/~wismuell/pub/europvm96.ps.gz

[256] R. Wismüller. On-Line Monitoring Support in PVM and MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (Proc. EuroPVM/MPI'98)*, Band 1497 aus *Lecture Notes in Computer Science*, Seiten 312–319, Liverpool, UK, September 1998. Springer-Verlag. http://wwwbode.in.tum.de/~wismuell/pub/europvm98.ps.gz

[257] R. Wismüller. Experience Report on Using the dyninstAPI. Technischer Bericht für das *Dept. of Computer Science, Univ. of Wisconsin-Madison*, Dezember 1999. http://wwwbode.in.tum.de/~wismuell/pub/madison.ps.gz

[258] R. Wismüller. Interoperability Support in the Distributed Monitoring System OCM. In R. Wyrzykowski et al., Herausgeber, *Proc. 3rd International Conference on Parallel Processing and Applied Mathematics - PPAM'99*, Seiten 77–91, Kazimierz Dolny, Polen, September 1999. Technical University of Czestochowa. http://wwwbode.in.tum.de/~wismuell/pub/ppam99b.ps.gz

[259] R. Wismüller. Interoperable Werkzeuge zur Laufzeituntersuchung und -steuerung paralleler und verteilter Systeme. Abschlußbericht für das DFG Forschungsstipendium WI 1346/1-1, LRR-TUM, Technische Universität München, Januar 2000. http://wwwbode.in.tum.de/~wismuell/pub/dfg00.ps.gz

[260] L. D. Wittie. Debugging distributed C programs by real time reply. In *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, Band 24(1) aus *ACM SIGPLAN Notices*, Seiten 57–67, Madison, WI, USA, Mai 1988. ACM. http://www.acm.org/pubs/articles/proceedings/onr/68210/p57-wittie/p57-wittie.pdf

[261] J. C. Wileden und A. Kaplan. Software Interoperability: Principles and Practice. In *Proc. 19th Intl. Conf. on Software Engineering*, Seiten 631–632, Boston, MA, USA, Mai 1997. ACM Press.

[262] R. Wismüller und T. Ludwig. THE TOOL-SET – An Integrated Tool Environment for PVM. In H. Lidell, A. Colbrook, B. Hertzberger und P. Sloot, Herausgeber, *Proc. High-Performance Computing and Networking*, Band 1067 aus *Lecture Notes in Computer Science*, Seiten 1029–1030, Brüssel, Belgien, April 1996. Springer-Verlag.

[263] R. Wismüller und T. Ludwig. Interoperable Run-Time Tools for Distributed Systems - A Case Study. In H. R. Arabnia, Herausgeber, *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Application, PDPTA'99, Volume IV*, Seiten 1763–1769, Las Vegas, NV, USA, Juni 1999. CSREA Press.

[264] R. Wismüller und T. Ludwig. Interoperable Run-Time Tools for Distributed Systems - A Case Study. *The Journal of Supercomputing*, 17(3):277–289, November 2000.
`http://www.wkap.nl/oasis.htm/277527`

[265] R. Wismüller, T. Ludwig, A. Bode, R. Borgeest, S. Lamberts, M. Oberhuber, C. Röder und G. Stellner. THE TOOL-SET Project: Towards an Integrated Tool Environment for Parallel Programming. In Xu De, K.-E. Großpietsch und C. Steigner, Herausgeber, *Proceedings of Second Sino-German Workshop on Advanced Parallel Processing Technologies, APPT'97*, Seiten 9–16, Koblenz, September 1997. Verlag Dietmar Fölbach.
`http://wwwbode.in.tum.de/~wismuell/pub/appt97.ps.gz`

[266] R. Wismüller, M. Oberhuber, J. Krammer und O. Hansen. Interactive debugging and performance analysis of massively parallel applications. *Parallel Computing*, 22(3):415–442, März 1996.
`http://wwwbode.in.tum.de/~wismuell/pub/pc95.ps.gz`

[267] P. H. Worley. A New PICL Trace File Format. Techn. Bericht ORNL/TM-12125, Oak Ridge National Laboratory, Oak Ridge, TN, USA, September 1992.
`http://www.epm.ornl.gov/~worley/papers/traceformat.ps`

[268] P. H. Worley. MPICL (Instrumentation Library for MPI). WWW-Seite des Oak Ridge National Laboratory, TN, USA, 1999.
`http://www.epm.ornl.gov/picl/mpicl.html`

[269] R. Wismüller, J. Trinitis und T. Ludwig. A Universal Infrastructure for the Run-time Monitoring of Parallel and Distributed Applications. In *Euro-Par'98, Parallel Processing*, Band 1470 aus *Lecture Notes in Computer Science*, Seiten 173–180, Southampton, UK, September 1998. Springer-Verlag.
`http://wwwbode.in.tum.de/~wismuell/pub/europar98.ps.gz`

[270] R. Wismüller, J. Trinitis und T. Ludwig. OCM — A Monitoring System for Interoperable Tools. In *Proc. 2nd SIGMETRICS Symposium on Parallel and Distributed Tools SPDT'98*, Seiten 1–9, Welches, OR, USA, August 1998. ACM Press.
`http://wwwbode.in.tum.de/~wismuell/pub/spdt98.ps.gz`

[271] G. Wiederhold, P. Wegner und S. Ceri. Towards Megaprogramming. *Communications of the ACM*, 35(11):89–99, November 1992.
`http://www-db.stanford.edu/CHAIMS/Doc/Papers/92cacm/cacm-final.ps`

[272] X/Open Event Services Subgroup. Event Services for System Management – A White Paper. White paper, X/Open Company Limited, Reading, UK, September 1994.

[273] X/Open Group. *System Management: UMA Data Capture Interface (DCI)*. X/Open Company Limited, Reading, UK, März 1995. X/Open Dokument Nr. P434.

[274] X/Open Group. *System Management: UMA Data Pool Definitions (DPD)*. X/Open Company Limited, Reading, UK, März 1995. X/Open Dokument Nr. P435.

[275] X/Open Group. *System Management: Universal Measurement Architecture Guide*. X/Open Company Limited, Reading, UK, März 1995. X/Open Dokument Nr. G414.

[276] X/Open Group. *Systems management: UMA Measurement Layer Interface (MLI)*. X/Open Company Limited, Reading, UK, März 1995. X/Open Dokument Nr. P426.

[277] J. C. Yan, S. R. Sarukkai und P. Mehra. Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit. *Software Practice & Experience*, 25(4):429–461, April 1995.
`http://www.nas.nasa.gov/Groups/Tools/Projects/AIMS/papers/swpe.ps`

[278] M. Zagha, B. Larson, S. Turner und M. Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proceedings Supercomputing '96*, Pittsburgh, PA, USA, November 1996. IEEE Computer Society.
`http://www.sgi.com/processors/r10k/timing/perf_count.html`

[279] V. C. Zandy, B. P. Miller und M. Livny. Process Hijacking. In *The Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC'99)*, Seiten 177–184, Redondo Beach, CA, USA, August 1999.
`http://www.cs.wisc.edu/paradyn/papers/hijack.ps.gz`

Index

Fett gedruckte Seitennummern weisen auf Begriffsdefinitionen hin.

R

S

T

U

V

W

Z