Forschungszentrum Jülich

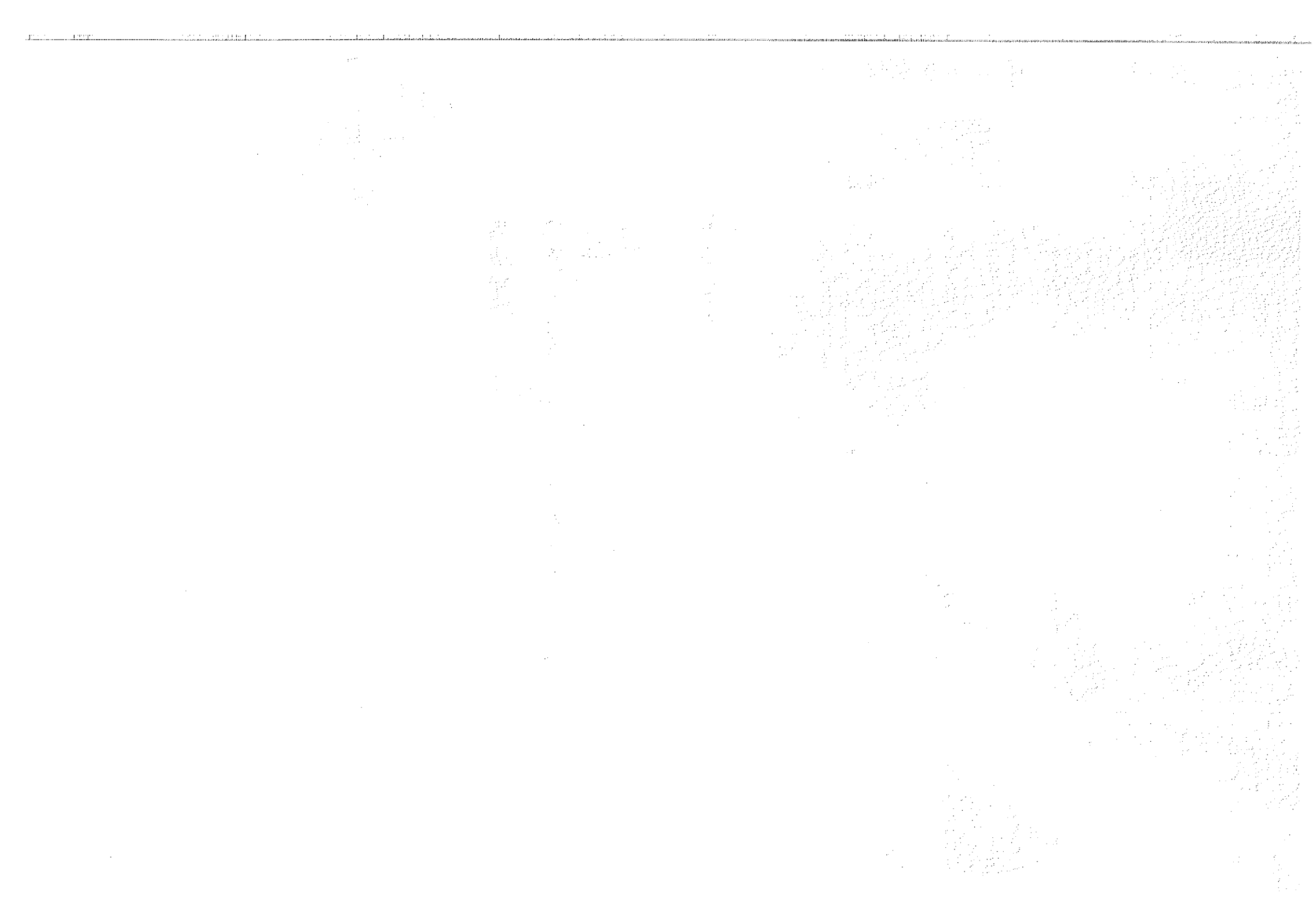
Zentralinstitut für Angewandte Mathematik

EARL

Eine programmierbare Umgebung zur Bewertung paralleler Prozesse auf Message-Passing-Systemen

Felix Wolf

JUEL-3551



EARL

Eine programmierbare Umgebung zur Bewertung paralleler Prozesse auf Message-Passing-Systemen

Felix Wolf

Berichte des Forschungszentrums Jülich ; 3551
ISSN 0944-2952
Zentralinstitut für Angewandte Mathematik Jül-3551

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek
D-52425 Jülich · Bundesrepublik Deutschland
☎ 02461/61-6102 · Telefax: 02461/61-6103 · e-mail: zb-publikation@fz-juelich.de

Kurzfassung

Bei der experimentellen Bewertung des Laufzeitverhaltens paralleler Anwendungen hat sich besonders bei Message-Passing-Systemen die Untersuchung von Ereignisspuren als hilfreiche Methode etabliert.

Diese Arbeit beschreibt die programmierbare Umgebung EARL (**E**vent **A**nalysis and **R**ecognition **L**anguage), welche durch ihre High-Level-Spuranalysesprache eine komfortable Plattform für die einfache und schnelle Erstellung neuer Werkzeuge zur Analyse von Message-Passing-Ereignisspuren anbietet. Die Werkzeuge werden als Skript in der von einem speziellen Spurformat unabhängigen EARL-Sprache verfaßt und durch den EARL-Interpreter ausgeführt.

Als programmierbares Werkzeug ist EARL besonders geeignet zur automatischen Identifikation von Leistungsengpässen, zur Programmvalidierung und zur Berechnung von benutzerdefinierten Leistungsindizes. Ebenfalls wird die automatische Durchführung von Meßreihen mit variierender Prozessoranzahl oder verschiedenen Eingabedaten unterstützt. Ein hohes Maß an Flexibilität erlaubt die Berücksichtigung anwendungsspezifischer Gesichtspunkte bei den genannten Aufgaben.

Die Leistungsfähigkeit von EARL resultiert vor allem aus der durch den Interpreter realisierten abstrakten Sicht auf die Ereignisspur, wodurch die Identifikation komplexer Ereignismuster wesentlich erleichtert wird.

Abstract

Using event traces to analyze the runtime behavior of parallel applications is a well-accepted technique, especially in the case of message passing systems.

This paper describes the programmable environment EARL (**E**vent **A**nalysis and **R**ecognition **L**anguage) which consists of a high-level trace analysis language and its interpreter. EARL enables convenient and fast construction of new trace analysis tools for message passing programs. The tools are written as scripts in the EARL language, independently from a specific trace format, and are then executed by the EARL interpreter.

Because of its programmability, EARL is especially well-suited for automatically identifying bottlenecks, validating programs, and calculating user-defined performance indices. Furthermore, automated execution of repeated measurements with varying number of processors or input data sets is supported. The high degree of flexibility offered by EARL allows for analyzing user programs with respect to domain- or application-specific requirements.

Much of EARL's power comes from its very high-level abstraction of an event trace, allowing for easy identification of complex event patterns.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Einordnung

Mit zunehmender Komplexität von Computeranwendungen steigt auch der Bedarf an Rechenleistung. Vor allem die Simulation komplexer Systeme aus Natur, Technik und Wirtschaft erfordern ein hohes Maß an Rechenkapazität. Aber auch andere Anwendungen, z.B. aus den Bereichen Grafik und Kryptologie, machen in dieser Hinsicht wachsende Ansprüche geltend. Es läßt sich jedoch auch der umgekehrte Trend beobachten, daß die Existenz von entsprechender Computertechnologie der Entstehung neuer Anwendungsfelder Vorschub leistet.

Da der Leistungsfähigkeit einzelner Komponenten einer Rechenanlage technologische Grenzen gesetzt sind, kann eine wesentliche Leistungssteigerung nur durch die Kopplung und gleichzeitige Verwendung mehrerer Komponenten erzielt werden. Aus diesem Grunde ist die Nutzung von Parallelrechnern unabdingbare Voraussetzung für die Gewinnung der erforderlichen Rechenleistung.

Neben der Entwicklung adäquater Algorithmen, der Konstruktion geeigneter Rechnerhardware wirft die parallele Programmiertechnik noch vielfältige Fragen auf. Trotz beachtlicher Errungenschaften auf diesem Gebiet ist u.a. die effiziente Implementierung eines parallelen Algorithmus immer noch mit zahlreichen Problemen behaftet. Eine wesentliche Ursache ist vor allem in der im Vergleich zu sequentiellen Programmen weitaus schwächeren Vorhersagbarkeit des Laufzeitverhaltens zu sehen. Diese resultiert aus der auch angesichts der Komplexität paralleler Rechnerhardware bestehenden Schwierigkeit, das Zusammenwirken der parallelen Prozesse zuverlässig zu modellieren. Die Folgen wichtiger Designentscheidungen können daher oft erst nach Programmausführung richtig beurteilt werden. Zu diesem Zeitpunkt müssen Korrekturen jedoch in der Regel mit sehr hohem Aufwand bezahlt werden.

Insgesamt ergibt sich das Bild eines inkrementellen Entwicklungsprozesses, bei dem das Programm schrittweise in ständigen Wechsel von experimenteller Beobachtung und Modifikation des Quellcodes optimiert wird.

Um dem Prozeß des parallelen *Software Engineerings* an dieser Stelle nachhaltig zu verbessern, d.h. die Häufigkeit solcher Schritte zu verringern, bedarf es neben einer Weiterentwicklung der verwendeten Modelle einer reichhaltigen Unterstützung der experimentellen Bewertung des Laufzeitverhaltens durch die Schaffung geeigneter Werkzeuge zur Sammlung und Evaluation von Laufzeitdaten. In der Vergangenheit wurden auf diesem Sektor bereits zahlreiche Fortschritte erzielt, dennoch kann dieses Problem des Entwicklungsprozesses noch nicht als gelöst betrachtet werden.

Nachwievor ist nicht hinreichend klar, wie man systematisch, von den gemessenen Laufzeitdaten ausgehend, Quellen unerwünschten Verhaltens im Programm geeignet identifizieren kann.

1.2 Motivation

Besonders bei Message-Passing-Systemen hat sich *Tracing* als hilfreiche Methode zur Beobachtung des Laufzeitverhaltens paralleler Anwendungen erwiesen. Dabei werden zur Laufzeit relevante Ereignisse aufgezeichnet und in einer *Ereignisspur* gesammelt. Diese kann nach Beendigung des Programms analysiert werden.

Gegenwärtig existiert eine Vielzahl mächtiger zumeist grafischer Spuranalysewerkzeuge wie z.B. VAMPIR [4], Upshot [15] , SIMPLE [23], Pablo [25] und viele andere. Keines jedoch löst alle der folgenden Probleme in zufriedenstellender Weise:

- Die von Rechenmaschinen heutiger Größe und Geschwindigkeit produzierten Ereignisspuren weisen häufig ein hohes Datenvolumen auf, wodurch sich insbesondere bei grafischen Werkzeugen die Lokalisierung relevanter Laufzeitabschnitte wie z.B. das Auffinden von Leistungsengpässen problematisch gestaltet. In der Regel wird die Ereignisspur als Ganzes eingelesen, und der Benutzer kann das aufgezeichnete Verhalten anschließend im Rahmen eines interaktiven Suchprozesses z.B. durch Zoomen analysieren. Diese Vorgehensweise ist erstens sehr zeitraubend und birgt zweitens die Gefahr, wichtige Informationen zu übersehen. Eine Automatisierung dieses Prozesses wäre von Vorteil.

  Im Extremfall kann die Spur vom Werkzeug aufgrund ihrer Größe allein aus technischen Gründen nicht verarbeitet werden.

- Obwohl die Werkzeuge eine Vielzahl verschiedener Sichten auf die Ereignisspur gewähren, können selten anwendungsspezifische Informationen sichtbar gemacht werden. Wissen über Aufbau und Struktur der Anwendung, welches zur Steuerung des Analysevorgangs benutzt werden könnte, bleibt meistens unberücksichtigt. Hier wäre ein sehr flexibles und einfach zu programmierendes Werkzeug erforderlich.

  Dadurch wäre es auch möglich, neue Analyseverfahren prototypisch zu evaluieren oder mit wenig Aufwand Werkzeuge für anwendungsspezifischen Gebrauch zu erstellen.

- Häufig möchte man Meßreihen durchführen, d.h. Laufzeitdaten in Abhängigkeit von Konfiguration und/oder Eingabedaten ermitteln, wobei im äußersten Fall die nächste Konfiguration erst durch das Ergebnis des letzten Ablaufs bestimmt wird. Neben der Möglichkeit, fremde Prozesse seitens des Werkzeugs zu kontrollieren, wäre auch Unterstützung für vergleichende Analyse mehrerer Ereignisspuren erforderlich.

Daher wurde im Rahmen dieser Arbeit eine programmierbare Umgebung namens EARL (**E**vent **A**nalysis and **R**ecognition **L**anguage) entwickelt, welche die einfache Erstellung neuer Werkzeuge zur Analyse von Message-Passing-Ereignisspuren durch das Schreiben von Skripten in einer High-Level-Spuranalysesprache ermöglicht. Die Skripten können anschließend durch den EARL-Interpreter ausgeführt werden.

Eine kompakte Darstellung der wesentlichen Konzepte dieser Arbeit findet sich in [29].

1.3 Inhalt

Nach dieser Einführung erfolgt in Kapitel 2 zunächst eine Beschreibung des Message-Passing-Programmiermodells sowie eine Darstellung von Tracing als Methode zur experimentellen Untersuchung des Laufzeitverhaltens. Zeitgenössische Spuranalysewerkzeuge werden vorgestellt. Nachdem in Kapitel 3 die Stuktur der EARL-Umgebung skizziert worden ist, wird in Kapitel 4 das von der EARL-Sprache realisierte abstrakte Modell einer Message-Passing-Spur präsentiert, welches eine komfortable Programmierung neuer Analysewerkzeuge gestattet. Eine Darstellung der EARL-Interpreters mit Bemerkungen zu seiner Implementation findet sich schließlich in Kapitel 5, während Kapitel 6 die vollständige Beschreibung der von EARL angebotenen Spuranalysesprache enthält. Die Demonstration der Fähigkeiten von EARL anhand einiger Skriptbeispiele ist Gegenstand von Kapitel 7. Das letzte Kapitel faßt wesentliche Punkte zusammen und liefert einen Ausblick auf mögliche Erweiterungen und Einsatzszenarien.

Kapitel 2

Grundlagen

Dieses Kapitel dient sowohl der Klärung notwendiger Begriffe als auch der Darstellung des *status quo* auf dem Gebiet dieser Arbeit.

Zunächst erfolgt eine Einführung in das Message-Passing Programmiermodell mit einer Abgrenzung von alternativen Modellen. Anschließend werden aktuelle Message-Passing-Systeme vorgestellt. Wegen seiner mittlerweile erlangten Bedeutung wird auf MPI [13] gesondert eingegangen, wodurch neben der exemplarischen Darstellung eines Message-Passing-Systems die Erläuterung wichtiger Konzepte der Programmierung auf der Basis von Message-Passing stattfindet.

Weiterhin wird ein Abriß der experimentellen Bewertung paralleler Programme durch die Bewertung paralleler Prozesse gegeben, und der Zusammenhang mit analytischen Methoden wird skizziert. Es folgt eine Darstellung der wichtigsten Verfahren zur Gewinnung von Laufzeitdaten. Da sich die vorliegende Arbeit speziell mit der Analyse von Ereignisspuren befaßt, wird die Tracing-Methode ausführlich behandelt.

Im letzten Teil werden die gegenwärtig zur Analyse von Message-Passing-Ereignisspuren eingesetzten Werkzeuge klassifiziert und beschrieben. Diese Beschreibung stellt gleichzeitig den Ausgangspunkt der vorliegenden Arbeit dar.

2.1 Message-Passing

Message-Passing ist heute eines der gebräuchlichsten parallelen Programmiermodelle, da es zur Lösung einer großen Klasse von Problemen geeignet ist.

Ein paralleles Programmiermodell ist eine Menge von Abstraktionen, die dem Programmierer eine vereinfachte und transparente Sicht auf das parallele Computersystem gewährt. Es beschreibt die grundsätzliche Art und Weise, wie Parallelität im Programm ausgedrückt wird.

Programmiermodelle existieren auf verschiedenen Ebenen. Die Implementation eines Modells kann selbst auf einem anderen Modell basieren, welches je nach Implementation unterschiedlich sein kann. Genauso kann ein Programmiermodell auf verschiedenen Hardwareplattformen realisiert sein.

2.1.1 Das Message-Passing-Programmiermodell

Das Message-Passing-Modell ist gekennzeichnet durch eine Menge von Prozessen mit lokalem Speicher, die durch das Versenden und Empfangen von Nachrichten miteinander kommunizieren. Eine wesentliche Eigenschaft von Message-Passing besteht darin, daß auf beiden Seiten explizit Operationen zum Senden und Empfangen erforderlich sind, um Daten vom lokalen Speicher eines Prozesses zum lokalen Speicher eines anderern zu transferieren. Man spricht auch von einem *zweiseitigen* Kommunikationsschema.

Beim Message-Passing wird zwischen *synchroner* und *asynchroner* Kommunikation unterschieden. Bei der synchronen Kommunikation müssen Sender und Empfänger gleichzeitig am Nachrichtenaustausch teilnehmen. Ist einer der beiden Prozesse nicht zur Kommunikation bereit, so wird der andere solange blockiert, bis der Partner bereit ist. Dieses Verfahren hat den Vorteil, daß im allgemeinen kein Puffer benötigt wird, aber auch den Nachteil, daß Wartezeiten durch Blockierung entstehen. Bei der asynchronen Kommunikation hingegen darf der Sender die Nachricht senden, gleichgültig, ob der Empfänger bereit ist oder nicht. Ebenfalls kann der Empfänger, ohne blockiert zu werden, seine Empfangsbereitschaft signalisieren. Spezielle Operationen erlauben dann zu prüfen, ob der Nachrichtenaustauch stattgefunden hat.

Im Message-Passing-Modell ist das dynamische Erzeugen und Beenden von Prozessen oder die Ausführung von mehreren Prozessen pro Prozessor nicht ausgeschlossen. Einige Message-Passing-Systeme erzeugen jedoch zum Programmstart eine feste Anzahl identischer Prozesse und verbieten die dynamische Erzeugung von Prozessen während der Laufzeit.

Message-Passing-Systeme werden in der Regel auf massiv parallelen Rechnerarchitekturen mit physikalisch verteiltem Speicher eingesetzt. Ebenfalls üblich ist die Verwendung auf Workstation-Clustern.

Um den Begriff Message-Passing von anderen Programmiermodellen abzugrenzen, seien im nächsten Unterabschnitt drei alternative Modelle dargestellt.

2.1.2 Alternative Programmiermodelle

Datenparalleles Modell

Hierbei wird die Parallelität allein durch Datenstrukturen und die darauf angewandten Operationen ausgedrückt. Strukturen wie z.B. Vektoren oder Matrizen,

bestehend aus eine Vielzahl gleichartiger Elemente, werden so verknüpft, daß auf jedem Element die gleiche Operation ausgeführt wird. Unter Berücksichtigung von Datenabhängigkeiten können diese Operationen häufig parallel ausgeführt werden. Die Parallelisierung dieser Operationen sowie die Partitionierung der zugehörigen Daten wird vom Compiler vollzogen und geschieht in der Regel transparent für den Programmierer. Ein datenparalleles Programm hat also sehr viel Ähnlichkeit mit einem sequentiellen Programm. Der *High Performance Fortran* (HPF) Standard [21] definiert einige Ergänzungen zu Fortran, welche einen Compiler bei der Partitionierung der Daten unterstützen.

Shared Memory

Im Gegensatz zum datenparallelen Modell unterliegt der Parallelismus hier der direkten Kontrolle durch den Programmierer. In diesem Modell greifen mehrere Threads auf ein und denselben Adreßraum zu. Der Zugriff auf von mehreren Threads manipulierte Daten wird durch verschieden Synchronisationsmechanismen, wie z.B. Semaphoren, koordiniert. Manche Implementationen verbergen jedoch den expliziten Gebrauch solcher Mechanismen.

Remote Memory Operationen

Wie beim Message-Passing ist dieses Modell durch eine Menge von Prozessen mit lokalem Speicher charakterisiert. Anders als beim Message-Passing jedoch, wo sowohl der lokale als auch der entfernte Prozeß aktiv an der Kommunikation beteiligt sein müssen, sowie im Gegensatz zum Shared-Memory-Modell, wo Prozesse auf Speicher zugreifen ohne Kenntnis, ob sie auf Hardware-Ebene Kommunikation mit entferntem Speicher auslösen, greift der lokale Prozeß zwar explizit durch spezielle *get* und *put* Operationen auf entfernten Speicher zu, jedoch ohne daß der dem entfernten Speicher zugeordnete Prozeß dazu einen Beitrag leisten muß. Man spricht auch von einem *einseitigen* Kommunikationsschema.

Gelegentlich wird dieses Modell gemeinsam mit Message-Passing verwendet. MPI 2.0 [22] z.B. unterstützt beide Modelle.

2.2 Zeitgenössische Message-Passing-Systeme

Die Menge der gegenwärtig genutzten Message-Passing-Systeme gliedert sich in die Gruppe der proprietären Systeme einerseits und in die Gruppe der hardwareübergreifenden Message-Passing-Bibliotheken andererseits.

Proprietäre Systeme, wie z.B. NX von Intel [17], werden von den Rechnerherstellern speziell für ihre eigenen Architekturen angeboten und können daher die Eigenschaften der Zielplattform optimal ausnutzen. Ein Nachteil entsteht jedoch, wenn das parallele Programm portiert werden soll.

Im Gegensatz dazu existieren eine Reihe von Message-Passing-Bibliotheken, die für mehrere Plattformen gleichzeitig angeboten werden. Zu den bekannteren zählen PICL [11], PVM [7], PARMACS [8] und MPI [13]. Sie sind frei verfügbar.

Gegenwärtig ist davon MPI das am weitesten verbreitete System. MPI ist als inoffizieller Standard definiert und wird ständig fortentwickelt. Seit 1997 existiert bereits die Definition von Version 2.0. Gegenwärtig wird jedoch noch hauptsächlich Version 1.2 genutzt. Wegen seiner inzwischen erlangten Bedeutung, und weil es die meisten Konzepte existierender Systeme subsummiert, soll es hier exemplarisch beschrieben werden. Auf die Darstellung der durch Version 2.0 eingeführten Erweiterungen wird jedoch verzichtet.

2.2.1 MPI

MPI ist ein De-facto-Standard für Message-Passing-Bibliotheken, durch welchen Namen, Aufrufsequenzen und Rückgabewerte für Operationen spezifiziert werden, die zum Versenden und Empfangen von Nachrichten dienen. Es gibt eine Sprachanbindung für Fortran und C.

In vielen MPI-Implementationen wird zu Beginn der Laufzeit eine feste Anzahl von Prozessen mit genau einem Prozeß pro Prozessor erzeugt.

Punkt-zu-Punkt-Kommunikation MPI enthält Operationen zum Versenden und Empfangen von Nachrichten zwischen einzelnen Prozessen, wobei genau ein Prozeß Sender und der andere Empfänger ist. Dabei wird sowohl synchrone Kommunikation durch blockierende Operationen sowie asynchrone Kommunikation durch nichtblockierende Operationen unterstützt.

Kennungen Nachrichten werden durch eine *Kennung (Tag)* gekennzeichnet. Die Kennung wird den Sende- und Empfangsoperationen als Argument übergeben, um diesen die gewünschten Nachrichten zuzuordnen. Auf diese Weise lassen sich zum Beispiel Prioritäten bei der Empfangsreihenfolge von Nachrichten implementieren.

Gruppen Prozesse gehören zu *Gruppen*. Enthält eine Gruppe n Prozesse, so sind die Prozesse von 0 bis $n-1$ durchnumeriert. Die Nummer innerhalb der Gruppe wird auch *Rang* genannt. Durch den Rang kann ein Prozeß innerhalb einer Gruppe eindeutig identifiziert werden. Zum Laufzeitbeginn gibt es nur die Gruppe aller Prozesse. Teilgruppen davon müssen zur Laufzeit etabliert werden.

Kommunikatoren Ein *Kommunikator* spezifiziert einen Kontext für eine Gruppe von Prozessen. Ein *Kontext* ist dabei ein Gültigkeitsbereich für Nachrichtenkennungen. Kommunikatoren ermöglichen Nachrichtenverkehr innerhalb von Modulen

oder Bibliotheken, ohne daß verschickte Nachrichten außerhalb empfangen werden können. Den meisten Kommunikationsoperationen wird ein Kommunikator als Argument übergeben. Auf die diesem Kommunikator zugeordnete Gruppe beziehen sich beispielweise bei Sende- und Empfangsoperationen die Angabe von Quell- bzw. Zielprozeß.

Die einfache Sende-Operation hat in MPI folgende Gestalt:

```
MPI_Send(buf, count, datatype, dest, tag, com)
```

- `(buf, count, datatype)` beschreibt `count` Vorkommen von Objekten des Typs `datatype`, beginnend an der Speicheradresse `buf`.
- `tag` gibt die Kennung der zu versendenden Nachricht an.
- `dest` gibt den Rang des Zielprozesses innerhalb der dem Kommunikator `com` zugeordneten Gruppe an.

Empfangen wird z.B. durch die Operation:

```
MPI_Recv(buf, count, datatype, source, tag, com, status)
```

- `source` gibt den Rang des Quellprozesses bezüglich des Kommunikators `com` an,
- `status` enthält nach Beendigung der Operation Angaben über die empfangene Nachricht.

`MPI_Recv` ist immer blockierend, `MPI_Send` ist je nach Implementation blockierend oder nichtblockierend. Es gibt jedoch explizit blockierende und nichtblockierende Varianten der Sende- und Empfangsoperationen.

Korrespondierende Sende- und Empfangsoperationen, d.h. Operationen, die sich auf ein und dieselbe Nachricht beziehen, besitzen als Argumente die gleiche Kennung und den gleichen Kommunikator. Eine Empfangsoperation empfängt immer die älteste verschickte aber noch nicht empfangene Nachricht mit der Kennung `tag`, die innerhalb des Kommunikators `com` versandt wurde.

Kollektive Kommunikation Neben den Punkt-zu-Punkt Operationen existieren noch sogenannte *kollektive* Operationen, die sowohl Daten zwischen mehr als zwei Prozessen austauschen als auch, damit einhergehend, kollektive Berechnungen wie Summenbildung von verteilten Daten ausführen können. Eine häufig benutzte kollektive Operation ist z.B.

```
MPI_Bcast(buf, count, datatype, root, com)
```

`MPI_Bcast` dient dazu, Daten von einem bestimmten als Argument spezifizierten Prozeß an mehrere andere Prozesse zu verteilen. Die Operation wird von allen beteiligten Prozessen ausgeführt.

- `root` gibt den Rang des Prozesses bezüglich des Kommunikators `com` an, der die zu verteilenden Daten besitzt.
- `(count, datatype)` beschreibt die Anzahl und den Typ der zu verteilenden Daten.
- `buf` gibt für den Prozeß `root` die Startadresse der zu verteilenden Daten und für alle anderen beteiligten Prozesse die Startadresse des Speicherbereichs, in den die Daten von `root` kopiert werden sollen, an.

Nach Beendigung der Operation enthalten alle beteiligten Prozesse an der durch `buf` spezifizierten Adresse die gleichen Daten.

2.3 Bewertung paralleler Prozesse

Das Ziel des Entwicklungsprozesses einer (parallelen) Anwendung ist die Erfüllung einer Anforderungsdefinition.

Gegenüber Anforderungen, die die Entwicklung der Anwendung selbst betreffen wie z.B. hohe Wartbarkeit und geringe Entwicklungskosten, stehen bei parallelen Anwendungen oft Anforderungen bezüglich des Laufzeitverhaltens im Vordergrund. Solche Laufzeitanforderungen betreffen den Ressourcenbedarf wie z.B. Rechenzeit und Speicherbedarf sowie andere Eigenschaften wie z.B. Skalierbarkeit.

Um festzustellen, ob bestimmte Laufzeitanforderungen erfüllt werden, muß man das Laufzeitverhalten bewerten. Dies geschieht mit analytischen Methoden einerseits sowie durch Experimente andererseits.

Analytische Methoden betrachten ein Laufzeitmodell, welches Entwurfsvariablen, Ausführungsparameter und Laufzeitgrößen in Beziehung setzt. Modelle werden bereits vor der Implementierung genutzt, um wichtige Designentscheidungen zu treffen. Je nach Entwicklungsstadium der Anwendung können auch spezielle Eigenschaften der Zielplattform(en) in das Modell integriert werden.

Experimente auf einer konkreten Plattform sind geeignet, das Modell zu verifizieren, zu verfeinern und zu ergänzen. Da bei den experimentellen Bewertungsverfahren Programmläufe betrachtet werden, und zu jedem Programm je nach Ausführungsparametern wie Eingabedaten und Konfiguration unterschiedliche Läufe existieren können, ist es sinnvoll, von der *Bewertung paralleler Prozesse* zu sprechen. Experimentelle Verfahren setzen sich zusammen aus der Beobachtung von Prozessen und der Evaluation der gewonnenen Informationen. Ein Vergleich mit dem Modell kann angestellt werden. Auf letzteren Gesichtspunkt wird hier jedoch nicht weiter eingegangen.

2.3.1 Gewinnung von Laufzeitdaten

Bei der experimentellen Beobachtung des Laufzeitverhaltens werden im wesentlichen drei Techniken verwendet:

- Sampling,
- Counter,
- Tracing.

Sampling wird in der Regel dazu benutzt, sogenannte *Profiles* zu erstellen. Profiles geben Auskunft über die Ressourcen (meistens die Zeit), die in bestimmten Programmteilen verbraucht wurden. Die Zeiten werden z.B. ermittelt, indem zur Laufzeit in regelmäßigen Abständen der Programmzähler abgefragt wird. Unter Zuhilfenahme von Symboltabelleninformationen werden dann die Aufenthaltsdauern in bestimmten Programmteilen geschätzt. Die Daten werden auf einer pro-Prozessor Basis gesammelt. Sampling ist eine einfache aber nicht unbedingt genaue Methode. Die Menge der gesammelten Daten ist gering und unabhängig von der Programmlaufzeit.

Durch Counter werden die Vorkommen bestimmter Ereignisses wie z.B. Seitenfehler oder das Versenden von Nachrichten gezählt. Die verwendeten Zähler sind entweder in Hardware oder Software implementiert. Die erhaltenen Informationen sind genau, die erzeugte Datenmenge ist gering.

Eine nützliche Variante der Verwendung von Countern ist die Kombination mit Interval-Timern. Interval-Timer messen die Zeit, die in einem bestimmten Codeabschnitt des Programms verbracht wurde. Die gemessenen Werte können in Countern akkumuliert werden, um so die insgesamt in dem betrachteten Abschnitt verbrachte Zeit genau bestimmen zu können.

Beide Methoden, d.h. sowohl Sampling als auch der Einsatz von Countern, haben den Vorteil, daß das Laufzeitverhalten des Programms durch die Messung meistens nur in ganz geringem Maße beeinflußt wird. Es wird jedoch nur summarische Information gesammelt, deren Aussagekraft begrenzt ist. Um das Verhalten eines Programms verstehen zu können, ist oft eine detailliertere Sicht auf den Programmlauf erforderlich.

Tracing ist eine Technik, die zur Laufzeit relevante Ereignisse wie z.B. den Aufruf einer Funktion oder das Versenden einer Nachricht aufzeichnet. Das auf diese Weise erstellte Ablaufprotokoll wird auch *Event Trace* oder *Ereignisspur* genannt. Eine Ereignisspur ist eine Datei, die Ereignisrecords mit Zeitstempel und ggf. Angaben über den Typ des jeweiligen Ereignisses enthält. Sie wird zur Laufzeit erzeugt und kann nach Beendigung der Laufzeit, d.h. *post mortem*, analysiert werden. Die Analyse muß daher nicht notwendig auf dem Zielrechner des parallelen Programms stattfinden. Wesentlich für die Erzeugung von Ereignisspuren ist das Vorhandensein einer

hinreichend genauen *globalen* Uhr, damit die Ereignisse unterschiedlicher Prozesse in eine zeitliche Relation gebracht werden können.

Ereignisspuren unterstützen ein breites Spektrum von Möglichkeiten, das Programmverhalten zu studieren. Sie können genutzt werden, um kausale Zusammenhänge innerhalb von Kommunikationsmustern zu untersuchen oder um Wartezeiten und anders beschaffene Leistungsengpässe zu lokalisieren. Außerdem können aus einer Ereignisspur auch diejenigen Informationen extrahiert werden, die man durch Anwendung der zuvor beschriebenen Methoden erhielte.

Die Nachteile des Tracings entstehen in der Hauptsache durch die großen erzeugten Datenmengen. Wird auf einem System mit 128 Prozessorknoten alle 10 Millisekunden auf jedem Knoten ein Ereignisrecord mit einer Länge von 20 Bytes aufgezeichnet, was bei heutigen Taktraten einen durchaus realistischen Wert darstellt, so werden Daten mit einer Rate von ca. 15 MB pro Minute produziert. Dies führt erstens zu einem das Programmverhalten beeinflussenden Overhead, zum anderen macht die Größe der gewonnenen Ereignisspuren deren Verarbeitung u.U. problematisch. Dennoch ist Tracing vor allem wegen seiner vielseitigen Möglichkeiten heute ein weitverbreitetes Hilfsmittel.

Tracing kann natürlich mit den zuvor beschriebenen Methoden kombiniert werden, indem z.B. die mit einem Zeitstempel versehenen Zustände von Countern protokolliert werden.

2.3.2 Tracing auf Message-Passing-Systemen

Die Analyse paralleler Prozesse anhand von Ereignisspuren hat sich insbesondere bei Message-Passing-Systemen als hilfreiche Methode erwiesen, vor allem da sich der Austausch von Nachrichten hierdurch besonders gut beobachten läßt.

Erzeugung von Ereignisspuren Die Erzeugung einer Ereignisspur ist schematisch in Abb. 2.1 skizziert. Das parallele Programm wird zunächst *instrumentiert*, d.h. so präpariert, daß zur Laufzeit relevante Ereignisse aufgezeichnet werden. Relevante Ereignisse sind bei Message-Passing-Programmen in der Regel der Aufruf oder das Verlassen bestimmter Funktionen oder das Versenden und Empfangen von Nachrichten. Kollektive Kommunikationsereignisse werden bei allen gängigen Instrumentierungssystemen auf Punkt-zu-Punkt-Kommunikation abgebildet.

Die Instrumentierung ersetzt die Aufrufe bestimmter Routinen durch Aufrufe sogenannter *Wrapper* (Abb. 2.2). Wrapper enthalten Anweisungen zur Protokollierung von entsprechenden Ereignissen wie z.B. Aufruf und Verlassen der jeweiligen Funktion.

Im Falle einer Kommunikationsfunktion wird u.U. zusätzlich ein Ereignisrecord, welches das Versenden oder Empfangen einer Nachricht anzeigt, generiert. Da der

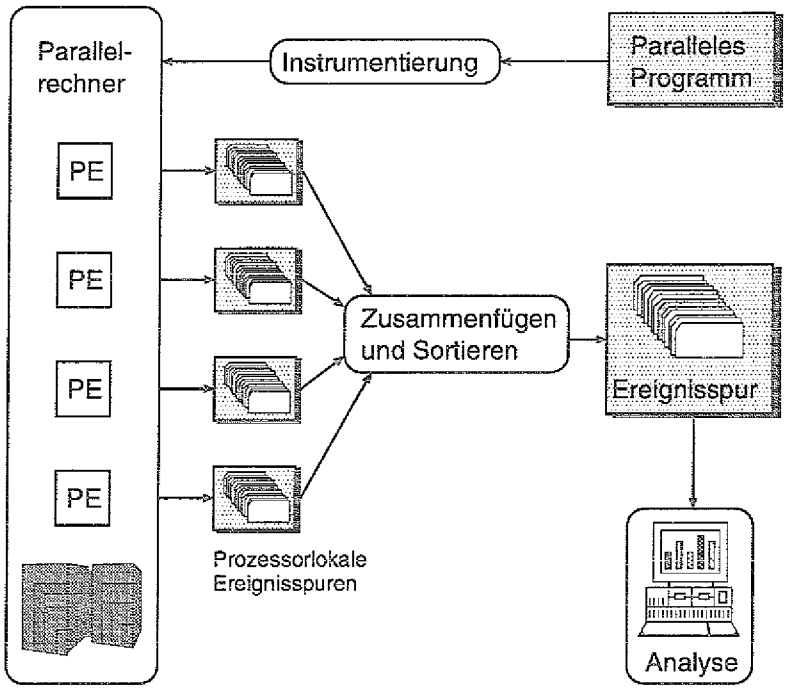


Abbildung 2.1: Erzeugung einer Ereignisspur

Sende- bzw. Empfangszeitpunkt einer Nachricht normalerweise nicht genau bestimmt werden kann und bei asynchroner Kommunikation noch nicht einmal innerhalb der zugehörigen Funktion liegen muß, muß er ggf. geschätzt werden.

Für die Funktionen der Kommunikationsbibliotheken wie MPI werden von Instrumentierungssystemen häufig Wrapper in Form von Bibliotheken zur Verfügung gestellt, die den Aufruf bzw. das Verlassen der entsprechenden Funktionen sowie ggf. zusätzlich Sende- und Empfangsereignisse aufzeichnen. Wrapper für benutzereigene Funktionen werden in der Regel automatisch generiert.

Man unterscheidet im wesentlichen zwei Instrumentierungsverfahren: *Quellcode-Instrumentierung* und *Objektcode-Instrumentierung*.

Quellcode-Instrumentierung wird manuell oder automatisch durchgeführt. Sie hat den Nachteil, daß das Programm nach jeder Instrumentierung neu übersetzt werden muß. Andererseits kann die Instrumentierung auf Quellcode-Ebene bei einer Portierung der Programms in der Regel erhalten bleiben.

Objektcode-Instrumentierung erfolgt nach der Übersetzung und automatisch. Ein hierzu geeignetes System ist z.B. PAT [10]. Ein Vorteil der Objektcode-Instrumentierung besteht in der Unabhängigkeit von der Verwendung einer bestimmten Programmiersprache. Außerdem muß das Programm bei einer Änderung der Instrumentierung nicht neu übersetzt werden. Portierung erfordert natürlich eine Neuinstrumentierung.

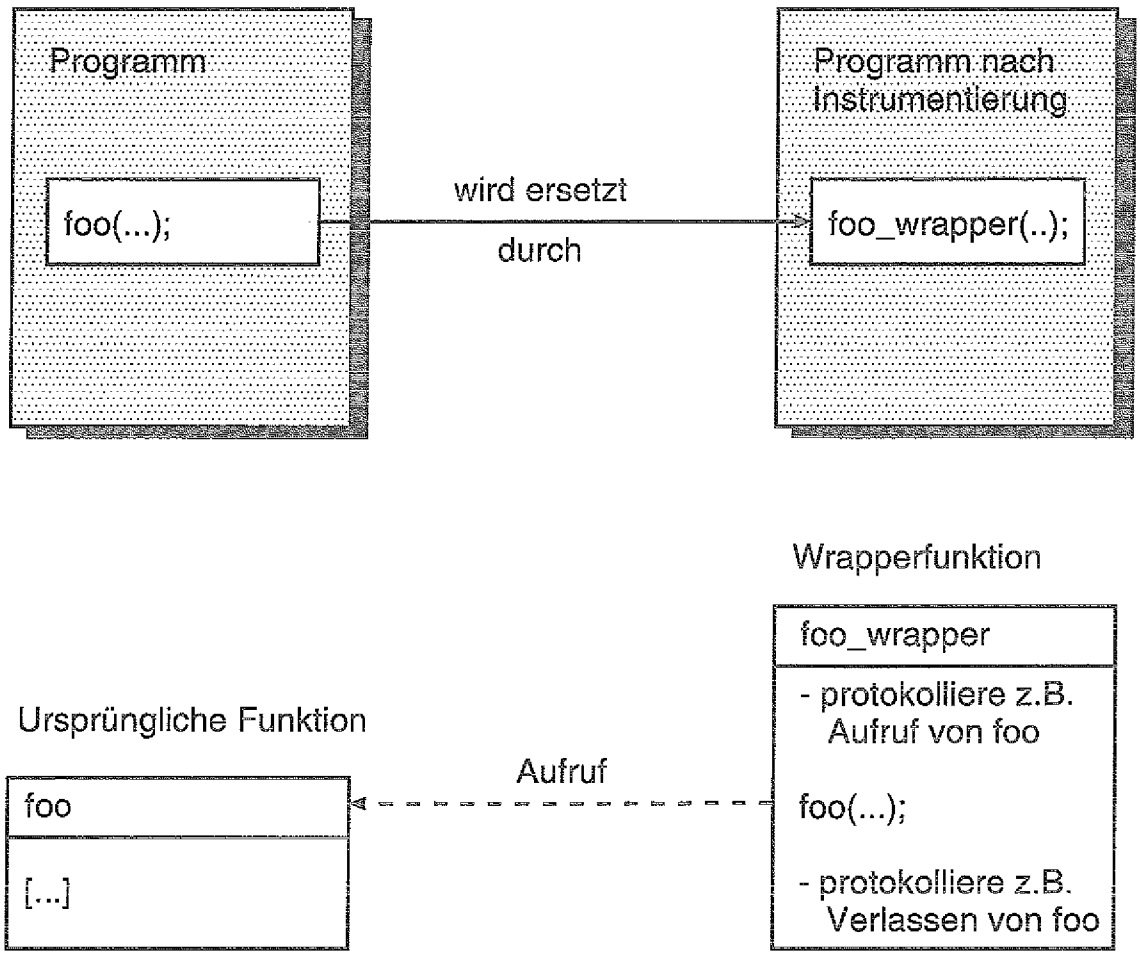


Abbildung 2.2: Instrumentierung eines Programms

Zur Laufzeit des parallelen Programms werden zu den jeweils aufgetretenen Ereignissen Ereignisrecords erzeugt und aus Effizienzgründen zunächst in einen Puffer geschrieben. Ist der Puffer voll, so wird dessen Inhalt in eine Datei kopiert. Da der Dateizugriff sehr zeitaufwendig ist, sollte der Puffer möglichst groß gewählt werden.

Zunächst protokolliert jeder Prozessor für sich alleine und erzeugt eine lokale Ereignisspurdatei. Nach Laufzeitende werden die prozessorlokalen Dateien zu einer einzigen zusammengefügt und die Einzelereignisse ihrer zeitlichen Reihenfolge gemäß sortiert.

Da die Ereignisrecords aus Platzgründen üblicherweise in codierter Form abgespeichert werden, enthält die Spurdatei zusätzlich noch einen Header, der Informationen zur korrekten Dekodierung beinhaltet. Außerdem enthält der Header in der Regel noch globale Informationen wie z.B. die Anzahl der verwendeten Prozessorknoten.

Anschließend kann die Ereignisspur mit einem entsprechenden Werkzeug verarbeitet werden. Zur Zeit gibt es keinen Standard für das Format einer Ereignispur. Die meisten Werkzeuge benutzen ein eigenes Spurformat. Daher bieten viele Werkzeuge

auch ein proprietäres System zur Spurerzeugung an.

2.4 Analysewerkzeuge

Die Werkzeuge zur Analyse von Message-Passing-Ereignisspuren lassen sich nur schwer klassifizieren, da die meisten eine Fülle verschiedener Funktionalitäten gleichzeitig anbieten. Es lassen sich jedoch zwei Familien mit unterschiedlichen Anwendungsschwerpunkten identifizieren:

- Visualisierung der Ereignisspur,
- benutzerdefinierte Transformation der Ereignisdaten.

Zur ersten Familie gehören Werkzeuge wie VAMPIR [4] und Upshot [15] bzw. Nupshot [19]. Eines der frühesten grafischen Werkzeuge dieser Art war Paragraph [14].

Die Ereignisspur wird visualisiert entweder als Animation oder wie bei VAMPIR als zweidimensionale Grafik (Abb. 2.3), wobei für jeden Prozessorknoten eine Zeitlinie gezeigt wird mit farblich markierten Übergängen zwischen den einzelnen Programmregionen. Bei Message-Passing-Spuren sind Nachrichten durch Striche zwischen den Zeitlinien der Knoten symbolisiert. Der Benutzer kann sich bestimmte Ausschnitte durch Zoomen näher ansehen sowie durch Ausblenden irrelevanter Informationen die Übersicht verbessern.

Neben diesen Darstellungsarten existieren noch zahlreiche Varianten. Außerdem werden oft noch zusätzlich Laufzeitstatistiken errechnet. VAMPIR z.B. berechnet Ausführungsprofiles für den jeweils visualisierten Ausschnitt der Ereignisspur. Die Ergebnisse der Statistiken können als Diagramme angezeigt werden.

Die zweite Familie besteht zumeist aus generischen Werkzeugen, die nicht speziell zur Analyse von Message-Passing-Spuren gemacht sind, und denen die Idee zugrunde liegt, durch Abstraktionen höherer Ordnung eine benutzerdefinierte Semantik der Ereignisse festzulegen. Die aktuellen Ereignisdaten werden von den Werkzeugen dann entsprechend umgewandelt.

EDL (**E**vent **D**efinition **L**anguage) [5] war eines der ersten programmierbaren Spuranalysewerkzeuge. EDL erlaubt die Definition komplexer Ereignismuster auf der Basis regulärer Ausdrücke, indem primitive Ereignisse mit Hilfe spezieller Operatoren zu Ereignissen höherer Ordnung kombiniert werden können. Die Konstituenten eines solchen Ausdrucks besitzen zusätzlich Attribute, die sich zueinander in Relation setzen lassen, um das Muster näher zu charakterisieren. Abtraktionsmechanismen gestatten die Wiederverwendung einmal definierter Muster zur Definition beliebiger Ereignishierarchien. Vom System wird aus der Spezifikation der zugehörige Automat generiert, der die Ereignisspur im Anschluß auf Vorkommen des jeweiligen Musters prüft. Zur Analyse von Message-Passing-Spuren ist EDL jedoch weniger geeignet,

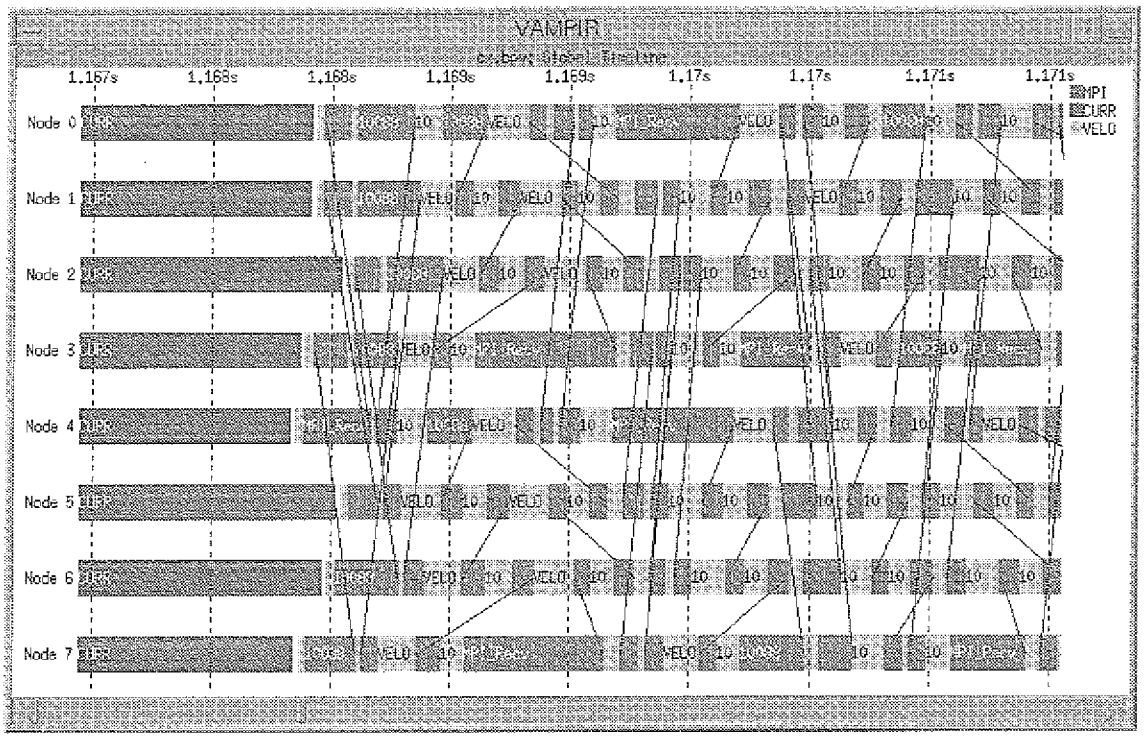


Abbildung 2.3: Grafische Darstellung einer Message-Passing-Ereignisspur mit VAMPIR

da die Handhabung der zur Bewertung interessanten Datenstrukturen wie Keller und Schlangen (Stacks und Queues) problematisch ist.

SIMPLE [23] ist eine aus mehreren Werkzeugen bestehende Umgebung zur Analyse von Ereignisspuren. Jedes Werkzeug verfügt über seine eigene Kommandosprache, um es für anwendungsspezifische Zwecke oder unterschiedliche Spurformate geeignet anzupassen. Für die gewünschte Aufgabe werden die Werkzeuge geeignet hintereinandergeschaltet. SIMPLE ist sehr vielseitig, was seine Bedienung jedoch auch ziemlich komplex gestaltet. Leider ist es nicht möglich, die Aufgaben der einzelnen Werkzeuge effizient zu kombinieren, so daß im schlimmsten Falle für jede Komponente die Ereignisspur vollständig durchlaufen werden muß.

Pablo [25] ist ein programmierbares grafisches Spuranalysewerkzeug. Pablo wird programmiert durch Kombinieren von vordefinierten Modulen für Spureingabe, Spurverarbeitung sowie grafischer Anzeige der transformierten Daten. Die Module selbst sind individuell konfigurierbar. Die Bedienung ist einfach, solange die eingebauten Module dem Analysevorhaben entsprechen. Eine Erweiterung ist problematisch, da sich die Implementation neuer Module sehr kompliziert gestaltet.

Die dargestellten programmierbaren Werkzeuge erfüllen die in der Einführung genannten Forderungen in Bezug auf Mesage-Passing-Systeme jedoch nur zum Teil oder in nicht befriedigender Weise. Hauptsächlich wegen der angestrebten universellen Verwendbarkeit gestaltet sich die Handhabung kompliziert oder bleibt auf eine

begrenzte Anzahl von Anwendungsmöglichkeiten beschränkt.

EARL bietet hingegen speziell für Message-Passing-Systeme eine einfache aber dennoch hochflexible Plattform zur benutzerdefinierten Transformation von Ereignisdaten.

Kapitel 3

Die EARL-Umgebung

Dieses Kapitel gibt einen Überblick über die Struktur von EARL und beschreibt seine Einsatzmöglichkeiten.

EARL ist eine Umgebung zur einfachen Erstellung neuer Werkzeuge für die Analyse von Message-Passing-Ereignisspuren. Man kann EARL daher auch als *Meta-Tool* bezeichnen.

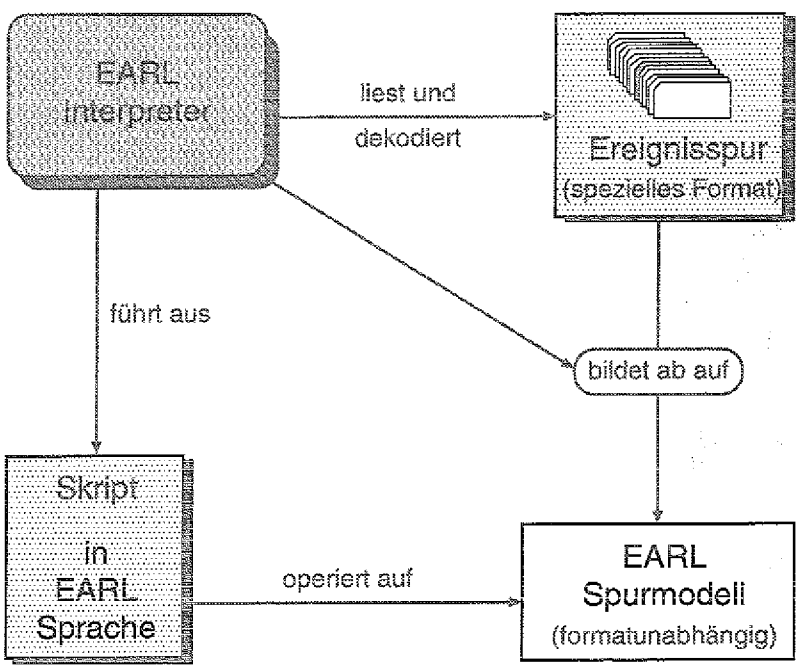


Abbildung 3.1: Der EARL-Interpreter bei der Ausführung eines EARL-Skriptes

Werkzeuge werden in Form von Skripten angefertigt, die in der EARL-Spuranalysesprache geschrieben sind. Die Skripten werden vom EARL-Interpreter ausgeführt. Die zu untersuchende Ereignisspur wird vom Interpreter zur Laufzeit des Skriptes gelesen, dekodiert und auf das EARL-Spurmodell abgebildet, auf welchem das Skript operiert (Abb. 3.1).

Mit EARL erstellte Analysewerkzeuge dienen z.B. der Berechnung von Statistiken, der Suche nach Leistungsengpässen oder der Programmvalidierung.

Je nach verwendeter Instrumentierung liegt die Ereignisspur in einem speziellen Format vor. Die einzelnen Formate besitzten in der Regel jedoch eine ähnliche logische Struktur. Sie definieren Ereignisse zum Betreten und Verlassen von Programmregionen sowie zum Versenden und Empfangen von Nachrichten. Auf dieser gemeinsamen Struktur basiert das EARL-Spurmodell. Da die EARL-Skripten nur auf dem EARL-Spurmodell arbeiten sind sie unabhängig von dem zugrundeliegenden Spurformat und können für jedes von EARL unterstützte Format eingesetzt werden. Zur Zeit sind diese das VAMPIR- [3, 4] und das ALOG-Format [15, 19].

Außerdem definiert das EARL-Spurmodell zusätzlich Abstraktionen, die dem Benutzer eine komfortable Programmierung ermöglichen.

Während die Ereignisspuren normalerweise einfache Dateien sind, die die Ereignisrecords in zeitlicher Reihenfolge enthalten und keine besondere Unterstützung für nichtsequentiellen Zugriff bieten, gestattet der EARL-Interpreter wahlfreien Zugriff auf die einzelnen Ereignisse.

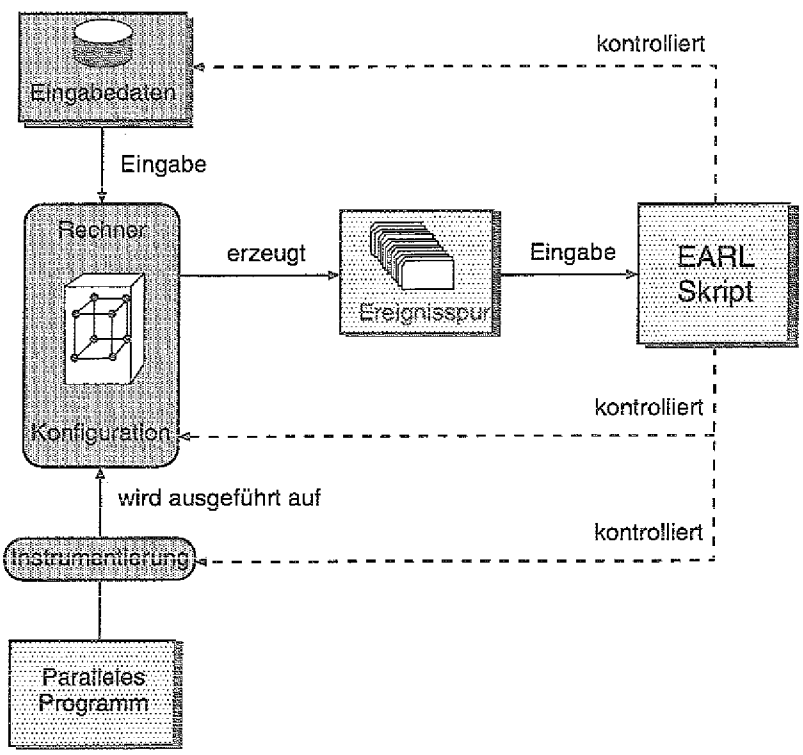


Abbildung 3.2: Experimente mit Feedback

Der EARL-Interpreter wurde implementiert als Erweiterung von Tcl [24]. Die EARL-Sprache besteht also aus dem vollen Umfang der Tcl-Skriptsprache, erweitert um die EARL eigenen Konstrukte.

Zusätzlich können alle für Tcl verfügbaren Erweiterungen für z.B. Grafik wie Tk [24] oder BLT [1] genutzt werden. Hierdurch ergeben sich reichhaltige Möglichkei-

ten zur Integration von mit EARL erstellten Werkzeugen. Eine Darstellung der Tcl-Umgebung erfolgt zusammen mit der Beschreibung des EARL-Interpreters im fünften Kapitel.

Tcl verfügt wie die meisten anderen UNIX-Skriptsprachen über die Möglichkeit, andere Prozesse auszuführen und zu kontrollieren. Dadurch wird die automatisierte Durchführung von Meßreihen mit wechselnden Ausführungs- und Meßparametern ermöglicht.

Das Verhalten eines parallelen Programms wird durch die Rechnerkonfiguration, z.B. die Anzahl der Knoten, und durch seine Eingabedaten beeinflußt. Zur Programmlaufzeit wird eine Ereignisspur erzeugt, die im Anschluß durch ein EARL-Skript analysiert wird. Je nach Ergebnis können nun sowohl die Konfiguration als auch die Eingabedaten variiert werden (Abb. 3.2). Auf Grundlage der neuen Ausführungsparameter kann nun eine zweite Ereignisspur erzeugt werden. Auf diese Weise lassen sich Feedback-gesteuerte Experimente durchführen. Neben den Eingabedaten und der Rechnerkonfiguration kann auch die Instrumentierung beeinflußt werden, um z.B. aufgrund der gewonnene Ergebnisse den Fokus der Messung zu verändern.

Die nächsten Kapitel liefern eine detaillierte Darstellung des EARL-Spurmodells, des EARL-Interpreters und der EARL-Spuranalysesprache.

Kapitel 4

Das EARL-Spurmodell

In diesem Kapitel wird das von EARL verwendete Modell einer Message-Passing-Ereignisspur vorgestellt. Das Spurmodell beschreibt die Sicht des Programmierers von EARL-Skripten auf die Ereignisspur. Es erfüllt im wesentlichen zwei Aufgaben:

- Verbergen von spurformatspezifischen Details,
- Definition von Strukturen über den Ereignissen, die eine vereinfachte Sicht auf die Ereignisspur gewähren.

Das EARL-Spurmodell betrachtet eine Ereignisspur als Ablaufprotokoll eines parallelen Prozesses auf einem Message-Passing-System. Der Prozeß besteht aus einer Menge von Einzelprozessen, die im folgenden *Knoten* genannt werden. Die Knoten sind von 0 bis $n-1$ durchnumeriert, wobei n die Gesamtzahl der vom Programm benutzten Prozesse angibt.

Das EARL-Spurmodell betrachtet eine Ereignisspur als eine endliche Folge von Ereignissen:

$$e_1, e_2, e_3, \ldots, e_m$$

Jedem Ereignis ist ein Zeitstempel zugeordnet, der den Zeitpunkt des Stattfindens angibt. Die Ereignisse sind ihrer zeitlichen Reihenfolge gemäß durchnumeriert, angefangen bei 1. D.h. für zwei Ereignisse e_i und e_j mit $i < j$ gilt: $Zeitstempel(e_i) \leq Zeitstempel(e_j)$. Außerdem ist jedem Ereignis genau ein Knoten als Ort des Geschehens zugeordnet. Gehören die beiden Ereignisse e_i und e_j zu demselben Knoten, so gilt sogar $Zeitstempel(e_i) < Zeitstempel(e_j)$. Durch die Ereignisnummer kann ein Ereignis eindeutig identifiziert werden. Referenzen auf Ereignisse werden deshalb im EARL-Modell durch die Ereignisnummer ausgedrückt.

Ereignisse einer Message-Passing-Spur beziehen sich auf das Betreten bzw. Verlassen von *Regionen* sowie auf das Versenden bzw. Empfangen von *Nachrichten*.

Regionen sind Programmabschnitte wie z.B. Funktionen, Unterprogramme, Schleifen oder einfache Blöcke. Ihnen ist ein Name zugewiesen. Die Ausführung einer Region wird *Regionenaktivierung* genannt. Zur Laufzeit kann es auf jedem Knoten mehrere Aktivierungen derselben Region geben, d.h. eine Region kann zur Laufzeit mehrmals durchlaufen werden. Die Regionenaktivierung beginnt mit der Ausführung der ersten und endet mit der Ausführung der letzten Anweisung innerhalb des der Region entsprechenden Programmabschnitts. Innerhalb einer Regionenaktivierung können Aktivierungen anderer oder, z.B. bei rekursiven Funktionen, derselben Region liegen. Man spricht von *eingeschlossenen* Regionenaktivierungen.

Regionen dürfen nur nach dem LIFO-Prinzip (*Last In First Out*) durchlaufen werden, d.h. die Regionenaktivierungen auf einem Knoten sind zeitlich entweder disjunkt oder es besteht eine Inklusionsbeziehung. Eine echte Überlappung ist hingegen nicht erlaubt. D.h. wird zur Programmlaufzeit auf einem bestimmten Knoten eine Region B betreten, nachdem eine Region A auf demselben Knoten betreten aber noch nicht verlassen worden ist, so darf A erst nach B verlassen werden. Das Betreten einer Region wird auch als *Aufruf* bezeichnet. Die eben formulierte Bedingung stellt sicher, daß die Aufrufstruktur auf jedem Knoten durch einen dynamischen Keller dargestellt werden kann. Dieser Keller wird *Regionenkeller* genannt. Jedem Zeitpunkt und Knoten kann also ein Zustand des zugehörigen Regionenkellers zugeordnet werden.

Regionen können in Gruppen eingeteilt sein, um z.B. Kommunikationsfunktionen von benutzereigenen Funktionen zu separieren.

Eine Nachricht wird zur Programmlaufzeit genau einmal verschickt. Sie hat genau einen Sender und genau einen Empfänger, d.h. die Nachricht wird über eine Punkt-zu-Punkt-Verbindung gesandt. Zu jeder Nachricht gehört also genau ein Sende- und ein Empfangsereignis. Nachrichten besitzten eine Kennung (*Tag*), die Sende- und Empfangsereignis einer Nachricht miteinander assoziiert. Bei MPI Programmen kann die Gültigkeit des Tags zusätzlich durch einen *Kommunikator* eingeschränkt werden. Außerdem haben Nachrichten eine feste Länge. Auf die Zuordnung von Sende- und Empfangsereignis wird später genauer eingegangen.

Da das EARL-Spurmodell nur Punkt-zu-Punkt-Kommunikation kennt, müssen kollektive Kommunikationsoperationen, wie sie z.B. in MPI definiert sind, auf Punkt-zu-Punkt-Kommunikation abgebildet werden.

4.1 Ereignistypen

Ereignisse sind klassifiziert nach Typen. Jeder *Ereignistyp* ist definiert durch eine Menge von *Attributen*. Jede Instanz, d.h. jedes konkrete Ereignis, ist bestimmt durch die den Attributen zugewiesenen Werte. EARL kennt vier vordefinierte Ereignistypen:

- **enter** für das Betreten einer Region,

- exit für das Verlassen einer Region,
- send für das Versenden einer Nachricht,
- recv für das Empfangen einer Nachricht.

Manche Spurformate gestatten die Definition spurspezifischer Typen, d.h. Typen deren Definition in der Spur selbst enthalten ist und für jede Spur unterschiedlich sein kann. Auch um lediglich formatspezifische Typen in Zukunft berücksichtigen zu können, ist im EARL-Spurmodell die Definition zusätzlicher Typen vorgesehen. Für sie ist jedoch eine einheitliche Struktur vorgegeben.

Es gibt fünf Attribute, die allen Ereignistypen gemeinsam sind. Sie werden auch *Basisattribute* genannt. Dazu zählt *num* für die Ereignisnummer, *node* für den Knoten, auf dem das Ereignis stattfand, sowie *time* für den Zeitstempel. Da EARL als Erweiterung von Tcl implementiert ist, und Tcl praktisch untypisiert ist, wird der Typbezeichner bei allen Ereignistypen explizit als Inhalt des Attributs *type* angegeben. Das letzte Basisattribut *enterptr* enthält einen Verweis auf den Beginn der Regionenaktivierung, in der sich der jeweilige Knoten vor Eintritt des Ereignisses befand. Von diesem Attribut wird noch die Rede sein.

enter-Ereignisse zeigen das Betreten oder den Aufruf einer Region an, d.h. den Beginn einer Regionenaktivierung. Der Name der betretenen Region sowie der Name der Gruppe, der die Region angehört, wird durch die beiden Attribute *region* und *group* angegeben. Die Einteilung in Gruppen wird vom Benutzer bei der Instrumentierung des Programms vorgenommen. Sie ist dann normalerweise im Header der erzeugten Spurdatei verzeichnet und muß vom EARL-Interpreter auf das Spurmodell abgebildet werden. Es gibt jedoch Spurformate die keine Gruppeneinteilung unterstützen. In diesem Fall wird das Attribut *group* auf 'All' gesetzt.

exit-Ereignisse beschreiben die Beendigung einer Regionenaktivierung und besitzen die gleiche Struktur wie enter-Ereignisse, nur daß hier *region* und *group* sich auf die gerade verlassene Region beziehen.

send-Ereignisse besitzen ein Attribut *dest* für die Angabe des Zielknotens der Nachricht. Das Attribute *tag* enthält die Kennung, mit der die Nachricht versandt wurde, als ganzzahligen Wert. Bei MPI Programmen wird, vorausgesetzt, das verwendete Spurformat unterstützt dies, der Kommunikator ebenfalls als numerischer Wert in Form einer ganzen Zahl als Inhalt des Attributs *com* angegeben. Anderfalls erhält *com* den Wert '-1'. *len* enthält die Länge der Nachricht in Bytes.

recv-Ereignisse sind ähnlich strukturiert. Anstelle des *dest*-Attributs enthalten sie jedoch ein *src*-Attribut für die Angabe des Knotens, von dem die Nachricht verschickt wurde. Die Knotenangaben der Attribute *src* und *dest* sind immer absolut, d.h. sie sind unabhängig von der Angabe eines Kommunikators - im Gegensatz zu den Kommunikationsfunktionen von MPI, wo immer der Rang des Knotens bezogen auf den aktuellen Kommunikator verlangt wird. *tag*, *com* und *len* sind wie bei send-Ereignissen. recv-Ereignisse verfügen zusätzlich noch über ein Attribut *sendptr*,

welches eine Referenz auf das korrespondierende send-Ereignis als Wert besitzt. Auf dieses Attribut wird noch gesondert eingegangen.

Da bei manchen Message-Passing-Systemen wie z.B. PVM die Länge der Nachricht nicht immer schon zum Sendezeitpunkt bestimmt werden kann, kann bei send-Ereignissen als Inhalt von *len* ggf. auch ein falscher Wert stehen. recv-Ereignisse enthalten jedoch immer den korrekten Wert.

Spur- bzw. formatspezifische Typen weisen ein einziges nicht-Basisattribut *data* auf. Sein Inhalt ist durch das Spurmodell nicht weiter spezifiziert, seine korrekte Interpretation obliegt dem Benutzer.

Die Attribute der EARL-Ereignistypen sind in Tabelle 4.1 noch einmal zusammengefaßt mit genauen Angaben über die Wertebereiche.

Tabelle 4.1: Die EARL-Ereignistypen

Attribute der Ereignistypen
n = Gesamtzahl der verwendeten Knoten m = Gesamtzahl der Ereignisse

Alle Typen	
num	Ereignisnummer als ganze Zahl zwischen 1 und m.
node	Knoten des Ereignisses als ganze Zahl zwischen 0 und $n-1$.
time	Zeitstempel des Ereignisses als Fließkommawert in Sekunden.
type	Explizite Angabe des Typs. Der Inhalt dieses Attributs ist für jeden Typ einzeln aufgeführt.
enterptr	Enthält eine Referenz auf den Beginn der Regionenaktivierung, in welcher sich der Knoten vor Eintritt des Ereignisses befand. Die Referenz wird ausgedrückt durch die Nummer des entsprechenden *enter*-Ereignisses. Fand das Ereignis auf Toplevel statt, so steht hier '-1'.

Betreten einer Region	
type	enter
region	Name der betretenen Region.
group	Name der Gruppe, der die betretene Region angehört. Bei Traceformaten ohne Gruppeneinteilung steht hier 'All'

Verlassen einer Region	
type	exit
region	Name der verlassenen Region.
group	Name der Gruppe, der die verlassene Region angehört. Bei Traceformaten ohne Gruppeneinteilung steht hier 'All'

Senden einer Nachricht	
type	send
dest	Nummer des Zielknotens als ganze Zahl zwischen 0 und $n-1$.
tag	Nachrichtenkennung als ganze Zahl.
com	Kommunikator der Sendeoperation als ganze Zahl. Bei Spuren, die keine Kommunikatoren enthalten, steht hier '-1'.
len	Länge der Nachricht in Bytes.

Empfangen einer Nachricht	
type	recv
src	Nummer des Absenderknotens als als ganze Zahl zwischen 0 und $n-1$.
tag	Nachrichtenkennung als ganze Zahl.
com	Kommunikator der Empfangsoperation als ganze Zahl. Bei Spuren, die keine Kommunikatoren enthalten, steht hier '-1'.
len	Länge der Nachricht in Bytes.
sendptr	Nummer des send-Ereignisses, durch welches die empfangene Nachricht verschickt wurde.

Spur- bzw. formatspezifische Typen	
type	Keine Vorgabe.
data	Keine Vorgabe. Korrekte Interpretation obliegt dem Benutzer.

4.2 Systemzustände

Eine Ereignisspur enthält Ereignisse, die auf einem parallelen System geschehen. Dabei verändert jedes Ereignis den Zustand des Systems. Demzufolge können die Ereignisse auch als Zustandsübergänge angesehen werden.

Die Analyse einer Ereignisspur kann erheblich vereinfacht werden, indem man eine Folge von Ereignissen darstellt durch die Veränderungen, die sie am Systemzustand bewirkt hat.

Eine Ereignisspur beschreibt also auch eine Folge von Zuständen:

$$S_0, S_1, S_2, \ldots, S_m$$

Dabei entspricht S_i dem Systemzustand nach Eintritt des Ereignisses e_i. Der Zustand S_0 ist der *initiale* Systemzustand vor Eintritt des ersten Ereignisses e_1. Zählt man den initialen Zustand mit, so definiert eine Spur mit m Ereignissen genau $m+1$ Zustände.

Ein Systemzustand S_i ist charakterisiert durch den Zustand der *Regionenkeller* sowie durch den Zustand der *Nachrichtenschlange (Message Queue)*. Regionenkeller verwalten die Regionenaufrufstruktur auf den einzelnen Knoten, während die Nachrichtenschlange verschickte aber vom Empfänger noch nicht angenommene Nachrichten aufbewahrt.

Im realen System besteht die Nachrichtenschlange natürlich aus einer Vielzahl einzelner Warteschlangen. Die Gesamtheit dieser Warteschlangen wird im EARL-Spurmodell als Nachrichtenschlange bezeichnet.

Der Zustand der Regionenkeller und der Nachrichtenschlange kann exakt durch Mengen von Ereignissen R und N beschrieben werden, wobei R diejenigen enter-Ereignisse enthält, die den Beginn der augenblicklich existierenden Regionenaktivierungen markieren, während N alle send-Ereignisse enthält, durch die die aktuell im Verkehr befindlichen Nachrichten verschickt wurden. Die Regionenkeller werden im EARL-Spurmodell durch die Menge R, die Nachrichtenschlange hingegen durch die Menge N repräsentiert. Die beiden Mengen N und R zusammen definieren den Systemzustand zu einem gegebenen Zeitpunkt bzw. nach einem gegebenen Ereignis:

$$S_i = (R_i, N_i), \quad 0 \leq i \leq m.$$

Sei $e_k.attr$ der Wert des Attributs *attr* des Ereignisses e_k. Die durch die Ereignisse bewirkten Zustandsübergänge sind wie folgt definiert:

Der initiale Zustand $S_i = (R_0, N_0)$ ergibt sich aus

- $R_0 = \emptyset$,
- $N_0 = \emptyset$.

D.h. der initiale Zustand besteht aus leeren Regionenkellern und leerer Nachrichtenschlange. Alle Folgezustände lassen sich anhand der Ereignisfolge bestimmen:

Für alle $S_i = (R_i, N_i), \quad 1 \leq i \leq m$ gilt

- $$R_i = \begin{cases} R_{i-1} \cup \{e_i\} & \text{gdw} \quad e_i.type = \textsf{enter}, \\ R_{i-1} \setminus \{e_k\} & \text{gdw} \quad e_i.type = \textsf{exit}, \\ & \quad \text{und } e_k \text{ ist das jüngste Ereignis in } R_{i-1} \text{ mit} \\ & \qquad e_k.node = e_i.node, \\ R_{i-1} & \text{sonst}, \end{cases}$$
- $$N_i = \begin{cases} N_{i-1} \cup \{e_i\} & \text{gdw} \quad e_i.type = \textsf{send}, \\ N_{i-1} \setminus \{e_k\} & \text{gdw} \quad e_i.type = \textsf{recv}, \\ & \quad \text{und } e_k \text{ ist das älteste Ereignis in } N_{i-1} \text{ mit} \\ & \qquad e_k.node = e_i.src \quad \text{und} \\ & \qquad e_k.dest = e_i.node \quad \text{und} \\ & \qquad e_k.tag = e_i.tag \quad \text{und} \\ & \qquad e_k.com = e_i.com, \\ N_{i-1} & \text{sonst}. \end{cases}$$

Das jüngste Ereignis einer Menge ist als das Ereignis mit der höchsten Ereignisnummer definiert, während das älteste als das mit der kleinsten Ereignisnummer definiert ist.

Ein exit-Ereignis bewirkt also die Entfernung des korrespondierenden enter-Ereignisses aus der Menge R, d.h. desjenigen Ereignisses, durch welches die gerade verlassene Region betreten wurde. Betrachtet man die Menge R, beschränkt auf Ereignisse eines Knotens, als dynamischen Keller, so veranlassen enter-Ereignisse eine *push*- und exit-Ereignisse eine *pop*-Operation. recv-Ereignisse hingegen entfernen aus der Menge N das korrespondierende send-Ereignis, d.h. das Ereignis, durch welches die empfangene Nachricht verschickt wurde.

Die EARL-Spuranalysesprache gestattet es, den einem Ereigniszeitpunkt entsprechenden Systemzustand abzufragen. Es werden dann Teilmengen von R und N ausgegeben. R kann beschränkt auf einen anzugebenden Knoten (*node*) und N entweder als Ganzes oder beschränkt auf einen bestimmten Quellknoten (*dest*) oder ein bestimmtes Quell- und Zielknotenpaar (*dest* und *src*) abgefragt werden. Anfragen nach anderen Kriterien müssen vom Benutzer selbst durch Sortieren bzw. Zusammenfassen der Ergebnisse direkt unterstützter Anfragen realisiert werden.

Die Definition des Systemzustands durch Mengen von Ereignissen hat den Vorteil, daß sich dieser einerseits auf einfache Weise beschreiben läßt, die Beschreibung aber dennoch ein hohes Maß an Information enthält. Die Struktur des Systemzustands und insbesondere die zeitliche Relation der Ereignisse, die zu seiner Entstehung geführt hat, ist vollständig in den Attributen der Elemente von R und N enthalten.

Die Betrachtung der Regionenkeller ist nützlich, da der Benutzer hierdurch z.B. ein Ereignis einer bestimmten Stelle der Programmausführung zuordnen kann. Die Nachrichtenschlange dagegen gestattet z.B. Einblicke in die Kommunikationsreihenfolge. Insgesamt gewähren diese Strukturen eine vereinfachte Sicht auf die Ereignisspur, indem sie den Kontext beschreiben, in dem ein Ereignis stattfindet. Sie enthalten jedoch nur Informationen, die a priori in der Ereignisspur vorhanden sind.

4.3 Identifikation von Nachrichten und Regionenaktivierungen

Eine zentrale Eigenschaft des EARL-Spurmodells ist die Identifikation von *Regionenaktivierungen* und *Nachrichten* in der Ereignisspur. Dies wird erreicht mit Hilfe der beiden Attribute *enterptr* und *sendptr*.

Eine Nachricht kann aufgefaßt werden als ein Paar von korrespondierenden Ereignissen: das send-Ereigniss, durch welches die Nachricht verschickt wurde, und das recv-Ereignis, durch welches die Nachricht empfangen wurde. So verweist das Attribut *sendptr* eines recv-Ereignisses immer auf das korrespondierende send-Ereignis (Abb. 4.1). Der Verweis wird durch die zugehörige Ereignisnummer ausgedrückt.

Während die Ereignisspur selbst diese Zuordnung in der Regel nicht enthält, ist sie im EARL-Spurmodell von vornherein enthalten. Sie wird vom EARL-Interpreter automatisch hergestellt.

Zwischen Regionenaktivierungen und Ereignissen bestehen zwei Arten von Beziehungen. Einerseits kann man eine Regionenaktivierung wie eine Nachricht als Paar von korrespondierenden Ereignissen auffassen, nämlich einmal das enter-Ereignis, welches den Beginn der Regionenaktivierung markiert, und das exit-Ereignis, welches das Ende der Regionenaktivierung beschreibt. Andererseits findet ein Ereignis auch *innerhalb* einer Regionenaktivierung statt, d.h. das Programm auf einem Knoten befindet sich in einer Region, bevor ein Ereignis stattfindet.

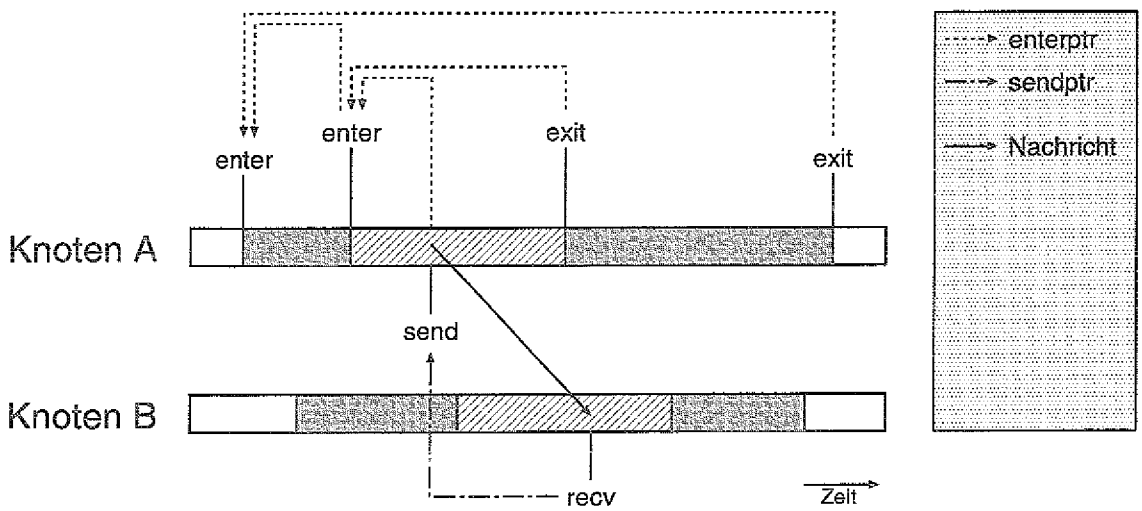


Abbildung 4.1: Identifikation von Nachrichten und Regionenaktivierungen durch die Attribute *enterptr* und *sendptr*

Beide Beziehungen werden durch das Basisattribut *enterptr* ausgedrückt. Der Inhalt von *enterptr* ist daher eine Referenz auf den Aufruf der auf diesem Knoten zuletzt vor Eintritt des Ereignisses betretenen Region. Die Einschränkung 'zuletzt' ist wesentlich, da auf einem Knoten mehrere Regionenaktivierungen gleichzeitig existieren können (Abb. 4.1). Der Aufruf wird durch das enter-Ereignis repräsentiert, die Referenz auf den Aufruf hingegen durch dessen Ereignisnummer.

Insbesondere verweisen exit-Ereignisse durch *enterptr* auf das korrespondierende enter-Ereignis, wodurch eine Regionenaktivierung als Paar korrespondierender enter- und exit-Ereignisse identifiziert werden kann.

Formal zeigt das Attribut *enterptr* eines Ereignisses e_k auf das jüngste Element e_i der Menge R_{k-1} mit der Eigenschaft

- $e_k.node = e_i.node$.

Existiert kein solches e_i, hat das Ereignis bezüglich des knotenlokalen Regionenkellers auf Toplevel stattgefunden. *enterptr* hat in diesem Falle den Wert '-1'.

Das *enterptr*-Attribut ermöglicht bei rekursiver Abfrage von einem Ereignis e_k aus die Navigation durch den Regionenkeller des Knotens $e_k.node$, d.h. via *enterptr* sind alle Elemente $e \in R_{k-1}$ mit $e.node = e_k.node$ erreichbar. Die Ereignisse werden bei rekursiver Abfrage in der umgekehrten Reihenfolge des Regionenaufrufs durchlaufen.

Die durch *enterptr* definierte Relation zwischen Ereignissen wird ebenfalls erst vom EARL-Interpreter hergestellt.

Abb. 4.1 zeigt einen Ausschnitt aus einer Ereignisspur. Auf der linken Seite sind zwei Knoten A und B aufgetragen, die Zeitachse verläuft von links nach rechts. Die Regionenaktivierungen auf den Knoten sind durch farbige Balken markiert. Eingeschlossene Regionenaktivierungen überdecken die einschließenden.

Beide Knoten befinden sich zu Beginn des gezeigten Abschnitts in der weißen Region, es folgt ein Übergang in die graue Region, von dort ein Übergang in die gestreifte Region. Während sich Knoten A in der gestreiften Region befindet, sendet er eine Nachricht an Knoten B, die dieser empfängt, während er sich ebenfalls in der gestreiften Region befindet. Anschließend werden auf beiden Knoten die betretenen Regionen schrittweise verlassen, bis sich beide wieder ausschließlich in der weißen Region befinden.

Ereignisse sind in der Grafik durch senkrechte Striche dargestellt, Nachrichten durch Pfeile. Der Einfachheit halber ist bei Knoten B nur das recv-Ereignis eingezeichnet. Die durch die Attribute *enterptr* und *sendptr* realisierten Verweise sind durch gestrichelte Pfeile gekennzeichnet.

Kapitel 5

Der EARL-Interpreter

Der EARL-Interpreter dient der Ausführung von in der EARL-Sprache geschriebenen Skripten. Die EARL-Sprache enthält u.a. Befehle zum

- Öffnen und Schließen von Ereignisspuren,
- Lesen von Ereignissen,
- Abfragen von Systemzuständen,
- Berechnen von Statistiken.

Beim Öffnen der Ereignisspur wird die zugehörige Datei geöffnet, je nach Spurformat wird der Header gelesen, um die zur Dekodierung der Ereignisrecords notwendigen Informationen zu erhalten. Zusätzlich werden noch Datenstrukturen zur Verwaltung von Symboltabellen und Systemzuständen sowie zur Pufferung von einmal gelesenen Daten angelegt.

Ereignisse können jetzt in beliebiger Reihenfolge unter Angabe der Nummer des gewünschten Ereignisses gelesen werden. Die entsprechenden Ereignisrecords werden vom EARL-Interpreter transparent für den Benutzer von der Spurdatei gelesen, dekodiert und auf die Ereignistypen des EARL-Spurmodells abgebildet. Puffermechanismen unterstützen einen effizienten Zugriff. Außerdem errechnet der Interpreter den dem jeweils gelesenen Ereignis entsprechenden Systemzustand, welcher abgefragt werden kann.

Das Schließen der Ereignisspur bewirkt eine Freigabe aller benutzten Ressourcen sowie die Schließung der entsprechenden Datei.

Wie bereits im dritten Kapitel erwähnt, ist EARL als Erweiterung der Skriptsprache Tcl implementiert. EARL umfaßt den vollen Tcl-Sprachumfang, ergänzt um die für EARL notwendige Funktionalität. Wird vom EARL-Interpreter gesprochen, so ist damit also der erweiterte Tcl-Interpreter gemeint. Zahlreiche Gründe haben dazu beigetragen, Tcl als Basisbaustein für EARL zu wählen. Hierzu zählen:

- einfache Erweiterbarkeit,
- mächtiger Sprachumfang,
- Existenz mannigfaltiger Ergänzungspakete, z.B. für Grafik,
- breite Unterstützung von Kooperation und Integration mehrerer Anwendungen.

Die genannten Möglichkeiten werden im folgenden genauer betrachtet.

5.1 Tcl

Tcl steht für *Tool Command Language* [24] und besteht aus der Tcl-Skriptsprache und dem zugehörigen Interpreter. Tcl ist auf zahlreichen Plattformen frei verfügbar, wie z.B. Unix, Windows und Macintosh. Da EARL in einer UNIX-Umgebung entwickelt wurde, soll hier, soweit plattformspezifische Details eine Rolle spielen, die UNIX-Version des Tcl-Interpreters beschrieben werden.

Tcl ist eine einfache Skriptsprache zur Kontrolle und Erweiterung beliebiger Anwendungen. Tcl verfügt über die üblichen Programmierkonstrukte wie Variablen, Schleifen und Prozeduren. Außerdem enthält die Tcl-Kernsprache Kommandos zur Ein-/Ausgabe, zum Dateizugriff, zur Stringverarbeitung und zur Auswertung arithmetischer Ausdrücke.

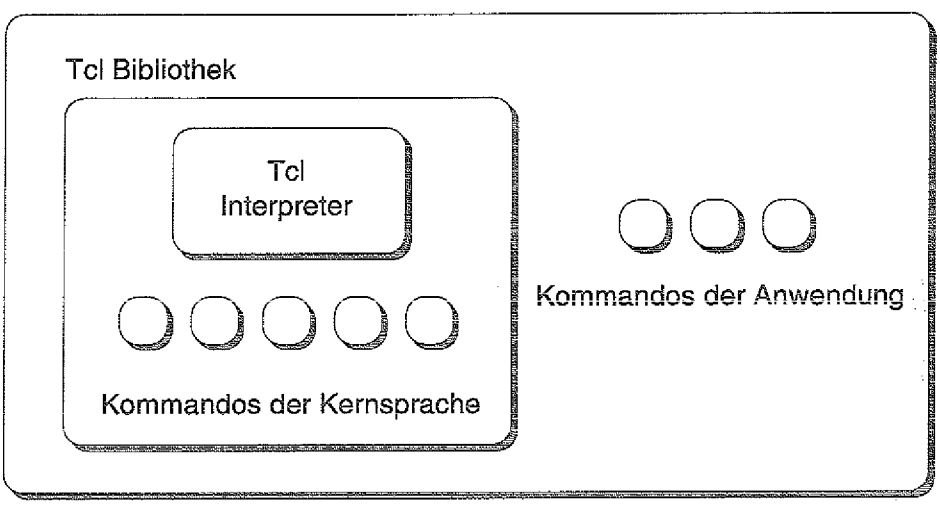


Abbildung 5.1: Eine einfache Tcl-Anwendung, bestehend aus dem Tcl-Interpreter, erweitert um einige anwendungsspezifische Kommandos.

Der Tcl-Interpreter besteht aus einer Bibliothek von C-Funktionen, die auf einfache Weise von einer Anwendungen aus genutzt werden kann. So kann z.B. die Tcl-Kernsprache um zusätzliche in C oder C++ implementierte Kommandos erweitert werden (Abb. 5.1).

Normalerweise wird Tcl als Tcl/Tk zusammen mit der Erweiterung Tk installiert, die Kommandos zur Erstellung von grafischen Benutzeroberflächen und zum Zeichnen einfacher Grafikobjekte anbietet. Tk kann entweder über die Tcl- oder über die C-Schnittstelle der Tk-Bibliothek in eine Anwendungen eingebettet werden. Neben Tk existiert noch eine Vielzahl weiterer nützlicher Ergänzungen.

Die Kommandos einer Tcl-Spracherweiterung können zu einem *Package* zusammengefaßt werden. Das ist vor allem dann von Vorteil, wenn man mehrere Packages zu einer gemeinsamen Anwendung kombinieren will. Packages können entweder statisch zur Linkzeit oder dynamisch zur Laufzeit geladen werden, vorausgesetzt die jeweilige Plattform unterstützt dies (Abb. 5.2).

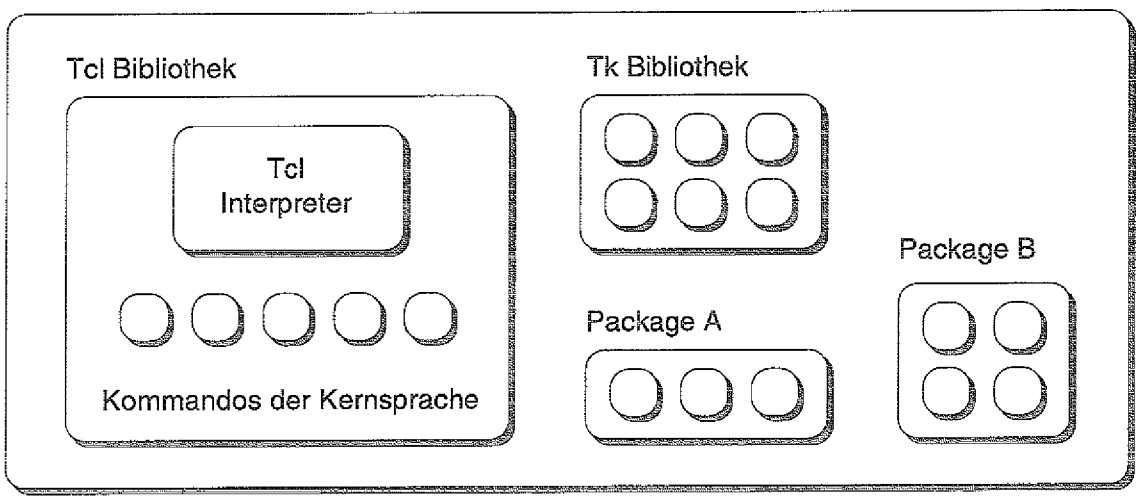


Abbildung 5.2: Eine komplexe Tcl-Anwendung, bestehend aus dem Tcl-Interpreter, erweitert um die Tk-Kommandos sowie mehrere Packages.

Der wesentliche Vorteil von Tcl besteht in der Möglickeit, komplexe Anwendungen aus der Kombination einfacher wiederverwendbarer Primitive aufzubauen. Die Primitive selbst sind in C oder C++ implementiert und können über Tcl-Kommandos angesprochen werden. Ein, gemessen an der beabsichtigten Funktionalität, einfaches Tcl-Skript fügt die Primitive schließlich zu einer übersichtlichen und wartbaren Gesamtanwendung zusammen.

Der Tcl-Interpreter gestattet sowohl die interaktive Eingabe von Kommandos als auch wie bei anderen UNIX-Skriptsprachen die Ausführung von Skripten entweder durch Übergabe des Skriptnamens als Parameter oder durch den direkten Aufruf des ausführbar gesetzten Skriptes, in dessen Kopfzeile der ausführende Interpreter spezifiziert ist.

Der folgende Abschnitt gibt eine kurze Einführung in Tcl. Er dient einerseits der Darstellung der durch Tcl gebotenen Möglichkeiten, andererseits soll er das Verständnis der im siebten Kapitel gezeigten Beispiele ermöglichen. Im Anschluß werden noch einige Bemerkungen zu Grafik und Kooperation bzw. Integration von Anwendungen im Rahmen der Tcl-Umgebung gemacht.

5.1.1 Die Tcl-Sprache

Während die meisten Compiler-Sprachen wie C u.a. durch eine komplexe Grammatik definiert sind, ist Tcl definiert durch einen Interpreter zum Parsen einzelner Tcl-Kommandos zusammen mit einer Menge von Prozeduren zu deren Ausführung. Ein Tcl-Skript besteht aus einer Aneinanderreihung von Kommandos; ein Kommando aus dem Kommandonamen und mehreren Argumenten, die durch *Whitespace*, d.h. Leerzeichen und Tabulatoren, voneinander getrennt sind:

command arg1 arg2 arg3 ...

Der Kommandoname *command* bezeichnet entweder ein internes Kommando oder eine in Tcl definierte Prozedur. Die Ausführung eines Kommandos ergibt ein Resultat in Form eines Strings. In den folgenden Beispielen wird das Resultat durch Voranstellung von '=>' gekennzeichnet. Ein Kommando wird durch *newline* beendet; eine Ausnahme bilden jedoch geklammerte Ausdrücke.

Variablen

Mit Hilfe des `set`-Kommandos kann einer Variablen ein Wert zugewiesen werden. Einer Variablen kann grundsätzlich jeder beliebige String zugewiesen werden, es gibt keine Typisierung. Durch das folgende Anweisung erhält die Variable `a` den String '5' als Wert:

```
set a 5
=> 5
```

Wird an das `set`-Kommando nur der Variablenname übergeben, so ist das Resultat einfach der aktuelle Wert der Variablen:

```
set a
=> 5
```

Arithmetische Ausdrücke

Das `expr`-Kommando evaluiert seine Argumente als arithmetischen Ausdruck und liefert das Ergebnis zurück:

```
expr 7.2 / 3
=> 2.4
```

Substitutionen

Beim Parsen einer Kommandozeile führt der Interpreter verschiedene Substitutionen aus. Elemente einer Kommandozeile, denen ein $-Zeichen vorangestellt ist, werden als Variablennamen interpretiert und durch den Wert der Variablen ersetzt. Ausdrücke in eckigen Klammern werden als Kommandozeile interpretiert und durch das

entsprechende Resultat substituiert. Die folgende Anweisung nimmt den Wert der Variablen a, teilt ihn durch zwei und weist das Ergebnis der Variablen b zu. Der neue Wert von b wird zurückgeliefert.

```
set b [expr $a / 2]
=> 2.5
```

Mehrere Argumente einer Kommandozeile können entweder durch Anführungszeichen oder durch geschweifte Klammern zu einem einzigen Argument gruppiert werden. Anführungszeichen erlauben Substitutionen innerhalb der Gruppe, geschweifte Klammern unterbinden dies:

```
set c "$a $b"
=> 5 2.5
set c {$a $b}
=> $a $b
```

In der ersten Kommandozeile wird der Variablen c der String '5 2.5' zugewiesen, der durch Gruppierung und anschließende Variabliensubtitution von a und b entstanden ist, während in der zweiten Kommandozeile die Variable c mit dem String '$a $b' belegt wird. Hier hat keine Substitution stattgefunden.

Ein Kommando kann natürlich von sich aus Substitutionen an den übergebenen Argumenten durchführen.

Datenstrukturen

Als Mittel zur Aggregation von Daten stehen in Tcl Listen und assoziative Arrays zur Verfügung. Listen sind Strings mit einer speziellen Interpretation, die durch die zu ihrer Verarbeitung bestimmten Kommandos definiert ist. Listen ähneln im Aufbau sehr stark einer Kommandozeile. Sie bestehen aus durch Whitespace getrennten Elementen. Geschachtelte Listen enthalten geklammerte Sublisten. Die Elemente einer Liste sind durchnumeriert, angefangen bei 0. '`lindex list` i' liefert das $(i+1)$te Element einer Liste.

```
lindex {{a b} c d} 0
=> a b
```

Da Listen durch Strings implementiert sind, ist ihre Verarbeitung nicht gerade effizient, insbesondere lange Listen können Probleme verursachen.

Ein assoziatives Array ist eine Variable mit einem stringwertigen Index. Ein Array bildet also Strings auf Strings ab. Intern sind Tcl-Arrays durch eine Hash-Tabelle implementiert, so daß die Kosten für Elementzugriff für alle Indizes in etwa gleich sind. Der Index eines Elements steht hinter dem Arraynamen in runden Klammern. Arrayelemente werden genau wie gewöhnliche Variablen mit `set` definiert:

```
set phone(Gaby) 0180/999
=> 0180/999
```

Tcl unterstützt nur eindimensionale Arrays. Mehrdimensionale können jedoch über zusammengesetzte Indizes simuliert werden.

Kontrollstrukturen

Die Tcl-Kontrollstrukturen sind wie normale Kommandos implementiert, z.B. Rumpf und Bedingung einer `if`-Anweisung werden einfach als Argumente an das `if`-Kommando übergeben:

```
if {$i == 0} {
   # Ausgabe von "i ist gleich 0"
   puts "i ist gleich 0"
}
```

Da Bedingung und Rumpf *Whitespace* enthalten, sind sie mit geschweiften Klammern zusammengefaßt. Das `puts`-Kommando mit einem Argument gibt dessen Inhalt auf der Standardausgabe aus. Kommentare werden mit `#` eingeleitet. Neben gängigen Kontrollstrukturen wie z.B. `for`- und `while`-Schleifen existiert noch ein Kommando zur Iteration durch Listen:

```
set i 0
foreach value {1 2 3 4} {
   set i [expr $i + $value]
}
set i
=> 10
```

Anfangs hat die Variable `i` den Wert `0`. Nacheinander werden die Elemente der Liste zu `i` hinzuaddiert. Das Resultat ist `10`.

Prozeduren

Größere Skripten lassen sich durch Prozeduren mit Hilfe des Kommandos `proc` strukturieren. Parameter werden standardmäßig *by value* übergeben. Jede Prozedur definiert einen lokalen Namensraum für Variablen. Durch bestimmte Kommandos kann der einem Variablennamen zugeordnete Namensraum geändert werden, wodurch Parameter auch *by reference* übergeben werden können. Dies ist insbesondere für Arrays von Vorteil.

Kontrolle von UNIX-Prozessen

Das `exec`-Kommando erlaubt die Ausführung von UNIX-Programmen. Die Standardausgabe des Programms wird zurückgeliefert. Schreibt das Programm auf die

Standardfehlerausgabe oder bricht mit einem Statuscode ungleich Null ab, so wird ein Fehler ausgelöst. Dieser kann jedoch durch geeignete Ausnahmebehandlung abgefangen werden. `exec` unterstützt volle Ein-/Ausgabe-Umlenkung.

Parameterübergabe an Skripten

Tcl-Skripten können Parameter übergeben werden. Nach dem Aufruf eines Skriptes enthält die Variable `argc` die Anzahl der übergebenen Parameter, wobei der Skriptname nicht mitgezählt wird. Die Variable `argv` enthält dann die Parameterliste, der Skriptname steht in einer eigenen Variablen.

Weitere Möglichkeiten

Neben den bisher bereits erwähnten Sprachelementen bietet Tcl u.a. noch reichhaltige Unterstützung für:

- String- und Listenverarbeitung
- Ausnahmebehandlung
- Dateizugriff
- Einfache Kommunikation zwischen Tcl-Prozessen
- *Namespaces* (ab Version 8.0)
- Socketprogrammierung

5.1.2 Grafikerweiterungen

Zu Tcl existieren zahlreiche Erweiterungen für Grafikanwendungen. Dazu zählen z.B. Tk zur Gestaltung grafischer Benutzeroberflächen sowie BLT, wodurch Tk um zusätzliche Möglichkeiten ergänzt wird.

Insbesondere für EARL, welches zur Schaffung neuer Analysewerkzeuge eingesetzt werden soll, ist Tk sehr hilfreich, da die Werkzeuge auf einfache Weise mit einer grafischen Benutzeroberfläche ausgestattet werden können. Außerdem wird durch das Tk-Widget *canvas* die Darstellung einfacher grafischer Objekte wie Bogen, Linien, Ovale und Polygone ermöglicht. Tk kann daher ausgezeichnet als Plattform zur Programmierung grafischer Präsentationen von Analysedaten fungieren.

BLT bietet darüber hinaus noch das Kommando `barchart` zum Zeichnen von Balkendiagrammen, `graph` für x-y-Diagramme und das `vector`-Kommando zur Unterstützung von Vektoren für Fließkommazahlen, wodurch das Anfertigen von Diagrammen weiter vereinfacht wird. Außerdem wird mit dem `bgexec`-Kommando noch

eine Erweiterung des `exec`-Kommandos zur Verfügung gestellt, mit dessen Hilfe sich UNIX-Prozesse im Hintergrund ausführen lassen. Nach deren Beendigung werden Ausgabe und Exit-Status in Tcl-Variablen geschrieben.

5.1.3 Kooperation und Integration von Anwendungen

Da Anwendungen und insbesondere Werkzeuge zur Softwareentwicklung aus mehreren Komponenten aufgebaut sein können, ist es hilfreich, über Möglichkeiten zur Kooperation und Integration von verschiedenen Anwendungen zu verfügen. Dies ist insbesondere im Kontext von EARL bedeutsam, um eine EARL-Anwendung mit anderen Werkzeugen zu kombinieren oder in eine Programmierumgebung einzubetten. Daher werden in diesem Abschnitt die von der Tcl-Umgebung und deren Erweiterungen hierzu angebotenen Konzepte vorgestellt.

Auf der Programmebene werden als Tcl-Kommandos implementierte Primitive in Form eines Tcl-Skriptes zu einer Anwendung kombiniert. Die Primitive können zu Packages zusammengefaßt werden, die einer Anwendung Dienste anbieten. Packages können dann wie Bibliotheken von unterschiedlichen Anwendungen genutzt werden. Ein Package wird entweder statisch zur Linkzeit oder dynamisch zur Laufzeit in die Anwendung integriert.

Bei der statischen Variante wird die Anwendung zur Linkzeit mit dem Package zusammengebunden, und die durch das Package implementierten Kommandos werden direkt nach dem Start beim Tcl-Interpreter registriert. Hierfür sorgt ein zuvor in die Anwendung gesetzter Aufruf einer vom Package bereitgestellten Initialisierungsfunktion. Durch die Registration wird der Interpreter in die Lage versetzt, die Ausführung der zusätzlichen Kommandos zu steuern.

Bei der dynamischen Variante liegt das Package als *shared library* vor und wird zur Laufzeit entweder durch das `load`-Kommando oder durch das Kommando `package require` geladen. Letztere Möglichkeit erfordert, daß das Package zuvor in einen entsprechenden Index der Tcl-Installation aufgenommen wurde. `package require` lädt das Package nicht direkt sondern erst zum Zeitpunkt der ersten Benutzung, d.h. wenn das erste durch das Package implementierte Kommando aufgerufen wird.

In einer komplexen Anwendung findet zur Laufzeit Kooperation zwischen beteiligten Prozessen statt. Dies betrifft Kooperation von Tcl-Prozessen untereinander, zwischen Tcl-Prozessen und fremden Prozessen sowie Kooperation über Rechnergrenzen hinaus.

Kooperation zwischen Tcl- bzw. Tk-Prozessen wird durch das von Tk angebotene `send`-Kommando ermöglicht, welches den Aufruf von Kommandos in anderen Tk-Anwendungen gestattet.

Kooperation zwischen Tcl-Prozessen einerseits und allgemeinen UNIX-Prozessen andererseits kann durch das `exec`-Kommando bzw. durch das `bgexec`-Kommando aus dem BLT-Package erreicht werden.

Über Rechnergrenzen hinaus können Tcl-Prozesse außerdem durch die Socketunterstützung nach dem *Client-Server*-Modell miteinander kommunizieren.

Zusätzlich kann die Kooperation zwischen den Komponenten einer integrierten Anwendung manuell durch den Benutzer gesteuert werden. Dies wird durch mit Tk zu erstellende Benutzeroberflächen wesentlich erleichtert.

5.2 Implementation von Tcl-Kommandos

Um die Darstellung der für EARL vorgenommenen Erweiterung des Tcl-Interpreters vorzubereiten, sollen in diesem Abschnitt zunächst die grundlegenden Mechanismen der Kommandoimplementation skizziert werden.

5.2.1 Kommando-Modelle

Die Tcl-Bibliothek unterstützt zwei Möglichkeiten zur Definition neuer Kommandos, einmal das aktions-orientierte Modell und zum anderen das objekt-orientierte Modell (*action-oriented* und *object-oriented approach* [24]).

Im aktions-orientierten Modell gibt es für jede Aktion ein Kommando, und dem Kommando wird der Name eines Objektes als Argument übergeben. Das `set`-Kommando ist z.B. aktions-orientiert; es manipuliert eine Variable, deren Name als Argument angegeben wird.

Im objekt-orientierten Modell existiert ein Kommando für jedes Objekt, und der Name des Kommandos ist der Name des Objekts. Wenn das Kommando aufgerufen wird, spezifizieren die Argumente die Operation, die am Objekt vollzogen werden soll. In der Regel werden solche Kommandos erst zur Laufzeit zusammen mit den durch sie repräsentierten Objekten erzeugt. Die meisten Tk-Widgets arbeiten nach diesem Prinzip. Z.B. ist `.b` ein einen Button repräsentierendens Kommando, so bewirkt `.b flash` ein Aufflackern des Buttons.

Es muß jedoch ausdrücklich betont werden, daß Tcl trotz der eben beschriebenen Fähigkeit keine objekt-orientierte Sprache ist, da z.B. Konzepte wie Vererbung nicht berücksichtigt sind. Aus diesem Grunde sollen dem objekt-orientierten Modell verpflichtete Kommandos in dieser Arbeit *objekt-assoziierte* Kommandos genannt werden und analog dazu die dem aktions-orientierten Modell entsprechenden *aktions-assoziiert*.

Um die Argumente eines objekt-assoziierten Kommandos, sofern sie Operationen spezifizieren, die an dem entsprechenden Objekt ausgeführt werden sollen, von gewöhnlichen, mit aktions-assoziierten Kommandos verbundenen Operationen abzugrenzen, werden sie als *Methoden* des Objekts bezeichnet.

Das objekt-orientierte Kommando-Modell ist vor allem dann vorteilhaft, wenn die Anzahl der Objekte klein gegenüber der Anzahl der Operationen ist. Wichtig ist,

daß von Tcl intern die Möglichkeit vorgesehen ist, mit einem Kommando Daten beliebiger Gestalt zu assoziieren.

5.2.2 Kommunikation mit dem Tcl-Interpreter

Der Tcl-Interpreter wird intern als Datenstruktur verwaltet, die z.B. Informationen über registrierte Kommandos, definierte Prozeduren oder Werte von Variablen enthält. [1] Kommunikation mit dem Interpreter findet statt, indem der Kommandoprozedur, also der für die Kommandoausführung zuständigen Prozedur, der Interpreter als Aktualparameter übergeben wird. Über entsprechende Bibliotheksfunktionen kann dieser dann manipuliert werden.

Kommandoprozeduren dürfen nicht mit Tcl-Prozeduren verwechselt werden. Kommandoprozeduren sind C- oder C++-Funktionen, die die einzelnen Kommandos implementieren. Tcl-Prozeduren sind selbst Kommandos, die durch das `proc`-Kommando beim Interpreter registriert werden. Ihnen wird eine Kommandoprozedur zur Ausführung des Prozedurrumpfes zugeordnet.

Bei der Initialisierung einer Anwendung wird ebenfalls der Interpreter an die Initialisierungsfunktion übergeben. Dadurch ist es möglich, die zu der Anwendung gehörenden Kommandos beim Interpreter zu registrieren.

EARL kommuniziert mit dem Interpreter zur Bewältigung mehrerer Aufgaben:

- Registration von aktions-assoziierten Kommandos bei der Initialisierung,
- Registration von objekt-assoziierten Kommandos, die von aktions-orientierten Kommandos erzeugt werden,
- Übergabe der Kommandoresultate an den Interpreter,
- Fehlerbehandlung,
- Manipulation von Variablen,

Um eine Vorstellung von den grundlegenden Mechanismen bei der Kommandoausführung zu geben, soll an dieser Stelle die Registration neuer Kommandos sowie die Schnittstelle der Kommandoprozeduren im Detail erklärt werden. Kommandos werden mit Hilfe folgender Bibliotheksfunktion registriert:

```
void Tcl_CreateCommand(Tcl_Interp *interp,
                       char *cmdName,
```

[1] Der Begriff Interpreter wird in dieser Arbeit mit unterschiedlicher Bedeutung verwendet, einmal für die eben erwähnte Interpreter-Datenstruktur, die den Zustand des Interpreters kennzeichnet, sowie für das Programm, welches Tcl- bzw. EARL-Skripten ausführt und natürlich eine Interpreter-Datenstruktur enthält. Aus dem jeweiligen Kontext wird klar, welcher Begriff gerade gemeint ist.

```
                    Tcl_CmdProc *cmdProc,
                    ClientData clientData,
                    Tcl_CmdDeleteProc *deleteProc);
```

- `interp` ist dabei ein Zeiger auf den Tcl-Interpreter, bei dem das Kommando registriert werden soll.
- `cmdProc` ist ein Zeiger auf die C- oder C++-Kommandoprozedur bzw. -funktion, in die beim Aufruf des Kommandos verzweigt werden soll.
- `cmdName` ist ein String mit dem das Kommando künftig in Skripten angesprochen werden kann.
- `clientData` wird nur bei objekt-assoziierten Kommandos verwendet und ist normalerweise ein Zeiger auf die assoziierten Daten, die beim Kommandoaufruf übergeben werden sollen.
- `deleteProc` ist ein Zeiger auf die Löschprozedur, die bei objekt-assoziierten Kommandos das Löschen der assoziierten Daten übernimmt. Sie wird beim eventuellen Löschen des Kommandos aufgerufen. Bei aktions-assozierten Kommandos wird hier `NULL` übergeben.

Die eigentliche Kommandoprozedur `cmdProc` muß genau wie die eingebauten Kommandos über folgende Schnittstelle verfügen.

```
int CmdProc(ClientData clientdata,
            Tcl_Interp *interp,
            int argc,
            char *argv[]);
```

- `clientData` enthält ggf. die assoziierten Daten.
- durch die Übergabe des Interpreters `interp` kann diesem das Kommandoresultat oder eine Fehlermeldung mitgeteilt werden. Genauso können jedoch auch z.B. Variablen mit Werten belegt werden.
- `argc` bezeichnet die Anzahl der übergebenen Aktualparameter des Kommandos einschließlich des Kommandonamens.
- `argv` ist ein Zeiger auf ein Array der Länge `argc` mit den übergebenen Aktualparametern.
- Der Exit-Status der Kommandoprozedur wird als `int`-Wert zurückgeliefert.

Die Löschprozedur `deleteProc` braucht dagegen nur ein Argument:

```
void DeleteProc(ClientData clientdata);
```

5.3 Die Tcl-Erweiterung EARL

Nach der Initialisierung kennt der EARL-Interpreter zunächst nur ein zusätzliches Kommando, das `earl`-Kommando. Es ist aktions-assoziiert und dient hauptsächlich der Erzeugung von objekt-assozierten Kommandos zusammen mit den entsprechenden Objekten. EARL kennt zwei verschieden Objektarten:

- Spurobjekte
- Statistikobjekte

Spurobjekte repräsentieren Ereignisspuren und offerieren Methoden zum Lesen von Ereignissen und Systemzuständen, Abfragen allgemeiner Information sowie zu ihrer Löschung. Statistikobjekte dienen der Verwaltung einer Meßreihe und bieten Methoden zur Aufnahme neuer Werte sowie zur Abfrage von statistischen Standardinformationen.

Wichtig ist, daß zur Laufzeit eines EARL-Skriptes mehrere Spurobjekte zu unterschiedlichen Ereignisspuren (ggf. auch nebeneinander) existieren können, was vergleichende Analyse mehrerer Ereignisspuren gestattet.

Die von EARL zu Tcl hinzugefügten Erweiterungen sind in C++ implementiert. Daher war es naheliegend, die Spur- und Statistikobjekte durch C++-Klassen zu realisieren.

5.3.1 Spurobjekte

Ein Spurobjekt wird zusammen mit einem assoziierten Kommando unter Angabe des Spurdateinamens und des Spurformats erzeugt. Je nach gewähltem Spurformat wird zunächst eine Instanz einer von der abstrakten Klasse `Earl_SeqTrace` abgeleiteten Klasse generiert. Dabei wird durch den Konstruktor auch die Spurdatei geöffnet, und deren Header wird eingelesen. Die Klasse `Earl_SeqTrace` enthält virtuelle Element-Funktionen zum sequentiellen Lesen und Dekodieren der Ereignisrecords. Außerdem wird Zugriff auf den Dateilesezeiger gewährt, um Rücksprünge innerhalb der Datei zu ermöglichen. Die abgeleiteten Klassen heißen `Earl_SeqAlogTrace` für das ALOG-Format sowie `Earl_SeqVmpTrace` für das VAMPIR-Format. Die Abbildung dieser beiden Spurformate auf das EARL-Spurmodell ist im Anhang A bzw. B beschrieben.

Soll EARL zur Bearbeitung alternativer Spurformate erweitert werden, so muß im wesentlichen nur eine neue `Earl_SeqTrace` implementierende Dekoderklasse geschrieben werden. Anhang C enthält eine genaue Darstellung der erforderlichen Maßnahmen.

Das neu erzeugte Dekoderobjekt vom Typ `Earl_SeqTrace` verkapselt die Spurdatei und verbirgt spurformatspezifische Details. Es dient nun als Argument für den

Konstruktor der Klasse `Earl_CmdTraceObj`. Ein Spurobjekt wird intern durch eine Instanz dieser Klasse repräsentiert. Die Element-Funktionen von `Earl_CmdTraceObj` sorgen für die Kommunikation zwischen Spurobjekt und Interpreter. Bei der Registrierung des assoziierten Kommandos wird der Registrierungsfunktion das Spurobjekt als Zeiger auf eine Instanz der Klasse `Earl_CmdTraceObj` übergeben. Zuvor wird der Zeiger jedoch noch *pro forma* in den Typ `clientData` umgewandelt.

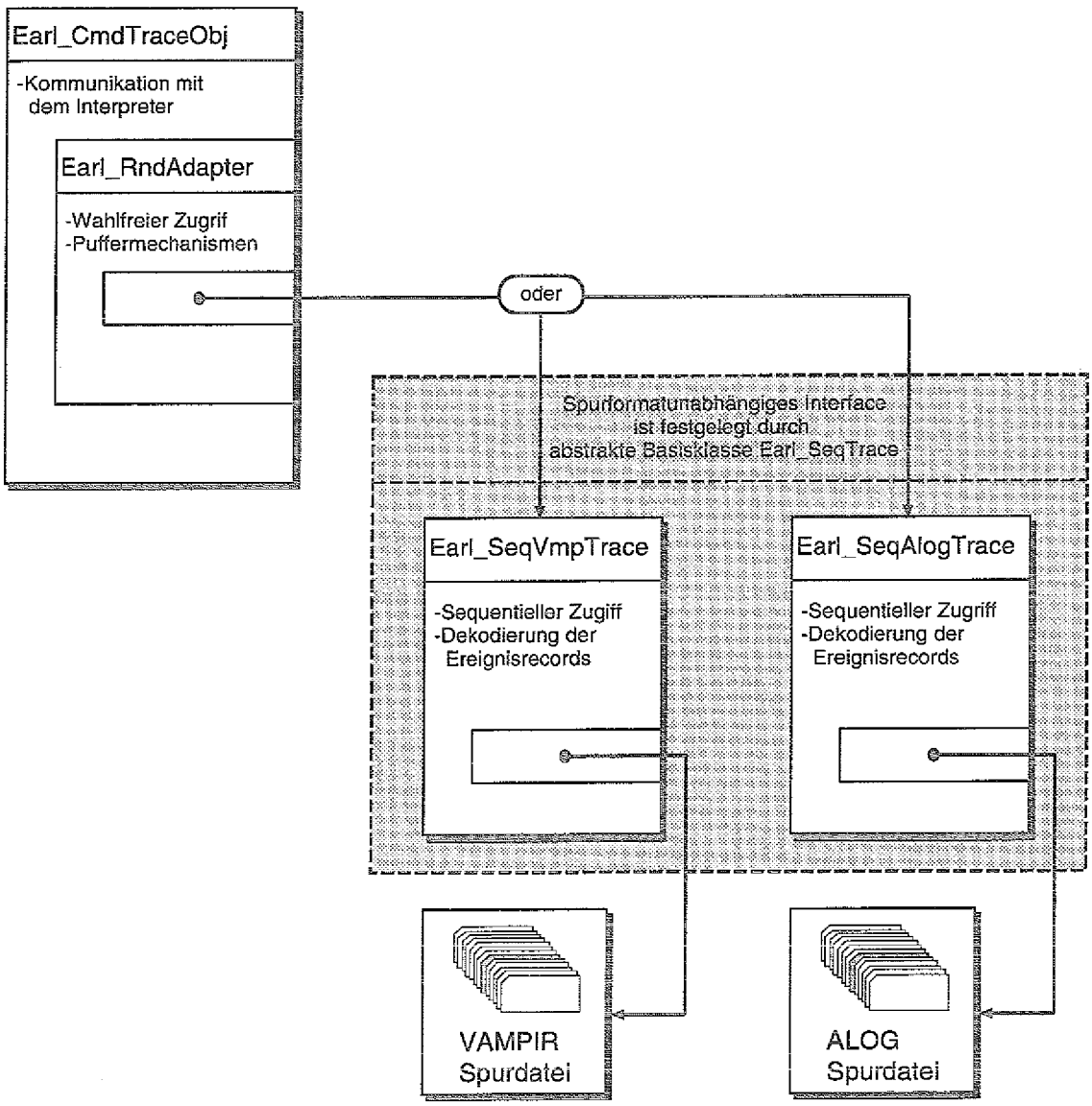


Abbildung 5.3: Spurobjekt als Instanz der Klasse `Earl_CmdTraceObj`

Bei jedem Aufruf des assoziierten Kommandos kann die Kommandoprozedur nun auf das entsprechende Spurobjekt zugreifen. Die Aufgabe der Kommandoprozedur besteht im wesentlichen darin, durch Lesen des ersten Arguments zu entscheiden, welche Methode des Spurobjekts gewählt wurde und daraufhin in die entsprechende Element-Funktion der Klasse `Earl_CmdTraceObj` zu verzweigen. Es gibt nur eine

einzige Kommandoprozedur für alle Spurobjektkommandos, der jedoch für jedes Kommando ein anderes Spurobjekt ausgehändigt wird.

Das Spurobjekt vom Typ `Earl_CmdTraceObj` benutzt das Dekoderobjekt vom Typ `Earl_SeqTrace` nicht direkt. Es verfügt statt dessen über ein Datenelement des Typs `Earl_RndAdapter`, welches das Dekoderobjekt kontrolliert und wahlfreien Zugriff auf die Ereignisse anbietet sowie Puffermechanismen bereitstellt, die diesen Zugriff möglichst effizient gestalten sollen. Die Klasse `Earl_RndAdapter` ist unabhängig von einem speziellen Spurformat und nutzt die ebenfalls formatunabhängige Schnittstelle der abstrakten Klasse `Earl_SeqTrace`. Die Struktur eines Spurobjekts ist schematisch in Abb. 5.3 skizziert.

Zum Lesen des VAMPIR-Spurformates benutzt die Dekoderklasse `Earl_SeqVmpTrace` die in C geschriebene Bibliothek *binlib* [3] von A. Arnold unter Verwendung einer Erweiterungsschnittstelle mit Zugriff auf den Dateilesezeiger. Die Bibliothek ist direkt im EARL-Quelltext enthalten.

Wird die Löschmethode des Spurobjekts aufgerufen, so wird die Löschung des assoziierten Kommandos veranlaßt. Als Folge wird ein Zeiger auf die entsprechende Instanz von `Earl_CmdTraceObj` an die Löschprozedur übergeben, was den Aufruf des Spurobjektdestruktors bewirkt. Dieser wiederum schließt die Spurdatei.

Ereignistypen

Die im EARL-Spurmodell beschriebenen Ereignistypen werden auch durch C++-Klassen implementiert. Alle C++-Ereignisklassen erben von der Basisklasse `Earl_Event`, welche die Basisattribute verwaltet.

EARL unterstützt zwei Arten, auf ein Ereignis zuzugreifen. Entweder wird es vom EARL-Interpreter einem durch die Attributbezeichner indizierten Array zugewiesen oder als Kommandoresultat in Form einer Liste ausgegeben. Die entsprechenden Klassen besitzen Element-Funktionen zur Arrayzuweisung oder zur Erzeugung einer solchen Liste.

Puffermechanismen

Die Ereignisspur wird, um Speicherprobleme zu vermeiden, vom EARL-Interpreter nicht als Ganzes in den Hauptspeicher geladen. Statt dessen wird erst nach Bedarf von der Datei gelesen. Um dennoch effizienten wahlfreien Zugriff auf die Ereignisse zu gewähren, verfügt der EARL-Interpreter über verschiedene Puffermechanismen. Bei der Entwicklung wurde davon ausgegangen, daß eine EARL-Anwendung eine Spurdatei im wesentlichen sequentiell von vorne nach hinten durchliest, dabei jedoch Rücksprünge vollzieht.

Gepuffert wird, indem Ereignisobjekte von Typ `Earl_Event` im Hauptspeicher gehalten werden. Dabei gibt es einen Zentralpuffer, welcher alle gespeicherten Ereignisse

enthält und dafür sorgt, daß kein Ereignis doppelt vorhanden ist. Die einzelnen Mechanismen dagegen enthalten nur Verweise in den Zentralpuffer.

Von den Puffermechanismen oft verwendete Strukturen sind die Systemzustände. Ist ein Systemzustand S_k gepuffert, so befinden sich alle Elemente der Mengen R_k und N_k im Zentralpuffer, während die Zustandsstruktur als Instanz der Klasse `Earl_RndState` Element-Funktionen anbietet, um auf die gespeicherten Ereignisse zuzugreifen. Dabei werden die Regionenkeller separat für jeden Knoten und die Nachrichtenschlange global verwaltet.

Bei den bisher unterstützten Spurformaten ALOG und VAMPIR sind die Ereignisrecords sequentiell in einer Datei abgelegt. Wahlfreier Zugriff ist nicht direkt möglich. Selbst wenn wahlfreier Zugriff auf ein Ereignisrecord möglich wäre, so könnte daraus nicht der Wert von Attributen wie z.B. *num*, *enterptr* oder *sendptr* ermittelt werden, da hierzu Informationen über vorangegangene Ereignisse erforderlich sind. Die Attribute des Ereignisses e_k können jedoch vollständig aus dem k-ten Ereignisrecord und dem Systemzustand S_{k-1} bestimmt werden.

Wurde die Spurdatei im Rahmen der Spurobjekterzeugung geöffnet, so befindet sich die aktuelle Leseposition der Datei am Beginn des ersten Records. Soll nun ein Ereignis e_k gelesen werden, so nimmt der EARL-Interpreter den *leeren* Systemzustand S_0 und liest sukzessive alle Ereignisrecords der Ereignisse e_1 bis e_k. Währenddessen werden aus S_0 und den Records die Ereignisse $e_1, \ldots, e_k$ zusammen mit den Zuständen $S_1, \ldots, S_k$ als Instanzen der entsprechenden Klassen gebildet. Der Zustand S_k kann abgefragt werden, d.h. Regionkeller und Nachrichtenschlange können ausgegeben werden.

Die, um das Ereignis e_k zu lesen, zwischenzeitlich generierten Systemzustände werden in regelmäßigen Abständen zusammen mit der Leseposition innerhalb der Spurdatei als *Lesezeichen* gepuffert, um bei Rücksprüngen nicht immer am Dateianfang beginnen zu müssen.

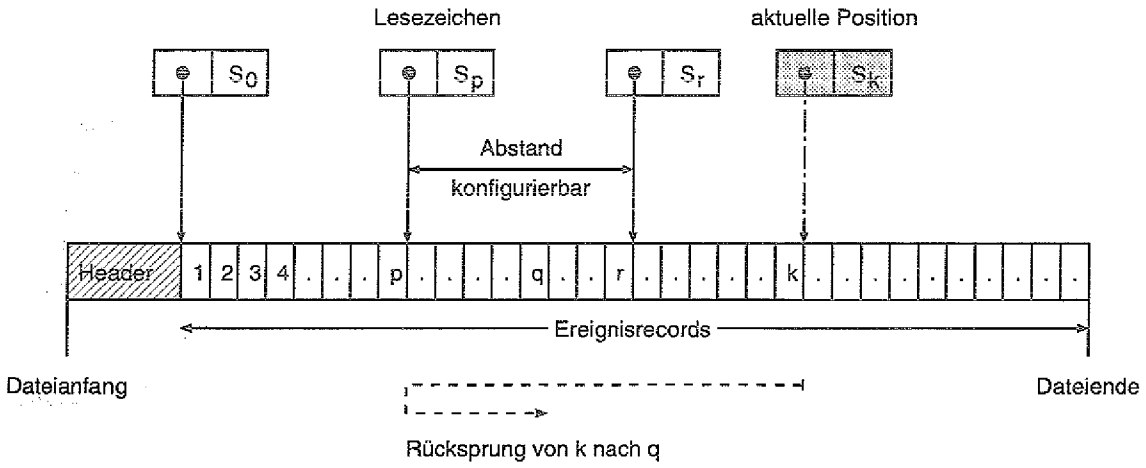


Abbildung 5.4: Spurdatei mit Lesezeichen

Abb. 5.4 zeigt die Situation nach Lesen des Ereignisses e_k. Um nun das Ereignis e_q zu lesen, vollzieht der Interpreter einen Rücksprung innerhalb der Datei. Der gestrichelte Pfeil im unteren Teil der Abbildung zeigt den dabei zurückgelegten Weg, um zum Record q zu gelangen. Der Interpreter wählt den jüngsten abgespeicherten Systemzustand S_p mit $p < q$ und beginnt von der gemeinsam mit dem Zustand gemerkten Dateiposition an zu lesen.

Soll nach e_k dagegen auf ein jüngeres Ereignis e_{k+h}, mit $h > 0$ zugegriffen werden, so wird einfach von der aktuellen Position aus weitergelesen. Der Abstand der Lesezeichen kann bei Erzeugung des Spurobjekts konfiguriert werden.

Wenn, um auf ein bestimmtes Ereignis zuzugreifen, Ereignisrecords von der Datei gelesen werden, werden die dabei zwischenzeitlich gebildeten Instanzen der Klasse `Earl_Event` in einem *History*-Puffer gespeichert, dessen maximale Größe ebenfalls bei der Erzeugung des Spurobjekts spezifiziert werden kann. Der Puffer enthält nur Ereignisse in Folge, bei Sprüngen wird der Puffer ggf. neu aufgebaut. Wird die Maximalgröße überschritten, so werden ältere Ereignisse zugunsten jüngerer verdrängt. Speicherung von entsprechender Systemzustandsinformation erlaubt die Navigation innerhalb der History mit Abfrage von Systemzuständen ohne weiteren Dateizugriff.

Außerdem befinden sich alle von den Ereignissen der History aus über die Attribute *sendptr* und *enterptr* (auch rekursiv) erreichbaren Ereignisse ebenfalls im Hauptspeicher. Dies ist wichtig, da Rückgriffe, wie die Beispiele im siebten Kapitel zeigen werden, vor allem anhand dieser Attribute vorgenommen werden.

Von obigen Mechanismen wird im Zusammenhang mit der Sprachbeschreibung im nächsten Kapitel noch die Rede sein.

5.3.2 Statistikobjekte

Häufig möchte der Benutzer statistische Informationen über aus der Ereignisspur ermittelte Werte gewinnen, wie z.B. Quantile und Durchschnittswerte. Die in Tcl enthaltenen Datenstrukturen Array und Liste sind jedoch nur in begrenztem Umfang zur effizienten Speicherung größerer Datenmengen geeignet. Außerdem kann u.U. die Anzahl der aus einer Ereignisspur gewonnenen Daten so groß werden, daß selbst bei geeigneten Datenstrukturen Speicherplatzprobleme zu befürchten sind. Aus diesem Grunde erlaubt EARL bei konstantem Speicherbedarf die Berechnung statistischer Angaben über eine beliebig große Folge von Fließkommawerten mit Hilfe sogenannter Statistikobjekte.

Statistikobjekte werden ebenfalls über ein objekt-assoziiertes Kommando angesprochen. Statistikobjekte offerieren Methoden zur Aufnahme neuer Werte, zur Reinitialisierung, zur Löschung und zur Abfrage von:

- Summe
- Minimum, Maximum

- Mittelwert
- Varianz
- 0,25, 0,5 und 0,75-Quantil

Das p-Quantil einer Folge von Werten ist derjenige Wert, unter dem 100p% der Folge liegen. Die Quantile werden nach dem P^2-Algorithmus [18] geschätzt, weshalb eine vollständige Speicherung der Folge unnötig wird. Daher ist die Speicherplatzanforderung von Statistikobjekten konstant und deshalb natürlich unabhängig von der Anzahl der Folgenelemente.

Statistikobjekte werden intern durch Instanzen der Klasse `Earl_CmdStatObj` repräsentiert. Die Erzeugung erfolgt analog zu den Spurobjekten.

5.4 EARL-Versionen

EARL wurde auf folgenden Plattformen installiert:

- Cray T3E unter UNICOS
- Sun unter Solaris 2.5 und 2.6
- PC unter Linux 2.0

EARL kann sowohl mit dem C++-Compiler von KAI (KCC) wie auch mit dem GNU-Compiler egcs compiliert werden. Andere Compiler, die dem neuen ISO-C++-Standard gehorchen, sollten jedoch auch zur Übersetzung von EARL eingesetzt werden können.

EARL hat sowohl mit Tcl/Tk 7.5/4.1 und 7.6/4.2 als auch mit 8.0/8.0 zusammengearbeitet. Unter Solaris wurde EARL sowohl als dynamisch ladbares Package compiliert wie auch als statisch gebundene Version mit Tk und BLT.

Kapitel 6

Die EARL-Sprache

Dieses Kapitel enthält die vollständige Darstellung aller Kommandos der EARL-Skriptsprache und ihrer Syntax. Dabei sind Schlüsselwörter und Kommandonamen in Schreibmaschinenschrift gehalten, variable Argumente sind kursiv gesetzt, und optionale Argumente sind von Fragezeichen umgeben. Das Vorkommen von '...' soll andeuten, daß u.U. noch mehrere Argumente folgen können.

Nach der Initialisierung ist zunächst nur das Kommando `earl` registriert. Es ist aktions-assoziiert und dient der Erzeugung von Spur- und Statistikobjekten. Außerdem erfüllt es noch eine Hilfe-Funktion. Welche Operation gerade durch das Kommando `earl` ausgeführt werden soll, wird durch das erste Argument spezifiziert.

> `earl` *operation ...*

Es wurde bewußt nicht jede Operation durch ein eigenes Kommando implementiert, um den Kommandonamensraum so wenig wie möglich zu verändern. Diese Vorgehensweise verringert die Wahrscheinlichkeit von Kommandonamenskonflikten mit anderen Tcl-Erweiterungen.

6.1 Spurobjekte

6.1.1 Erzeugung von Spurobjekten

Neue Spurobjekte werden erzeugt mit:

`earl open` *filename ?traceobj? ?options ...?*

> Die Spurdatei *filename* wird geöffnet und ein assoziertes Kommando namens *traceobj* wird generiert. Existiert bereits ein Kommando mit diesem Namen, so wird eine Fehlermeldung ausgegeben. Wird *traceobj* weggelassen, so wird für das Kommando automatisch ein Name vergeben. Der

Kommandoname wird zurückgeliefert. Das Spurformat kann mit der Option

`-format` *format*

eingestellt werden. Dabei wird der Formatbezeichner klein geschrieben. Mögliche Formatangaben sind '`alog`' und '`vampir`'. Das Defaultformat ist VAMPIR. Die Option

`-hist` *size*

bestimmt die maximale Größe des Historypuffers in Ereignissen. Fehlt die Angabe, so wird ein Defaultwert von 1000 Ereignissen angenommen. Diese Option akzeptiert nur positive ganzzahlige Werte. Andernfalls wird eine Fehlermeldung ausgegeben. Schließlich läßt sich mit Hilfe der Option

`-mark` *distance*

noch der Abstand der Lesezeichen in Ereignissen festlegen. Der Defaultwert ist 10000. Die Option akzeptiert nur nichtnegative ganzzahlige Werte. Bei unerlaubten Werten wird eine Fehlermeldung ausgegeben. Wird 0 angegeben, so wird lediglich das Lesezeichen zu Beginn des ersten Records gesetzt. Wird ein positiver Wert *distance* spezifiziert, so werden Lesezeichen zu Beginn aller Records mit der Nummer $1 + distance * i$ mit $i = 0, 1, 2, \ldots$ abgelegt.

6.1.2 Methoden von Spurobjekten

Lesen von Ereignissen

Ereignisse können gelesen werden einmal unter Angabe einer absoluten Ereignisnummer oder relativ zur Position des *aktuellen Lesezeigers*. Der aktuelle Lesezeiger hat normalerweise die Position des zuletzt gelesenen Ereignisses, zu Beginn steht er auf 0. Wie bereits erwähnt, gibt es zwei Möglickeiten, auf ein Ereignis zuzugreifen:

Ausgabe als Liste Die erste Variante (`get`) liefert das Ereignis in Form einer Liste $attr_1\ value_1\ \ldots attr_n\ value_n$, wobei $attr_i$ jeweils einen Attributbezeichner und $value_i$ dessen Wert repräsentiert. Die Reihenfolge der Attribute entspricht der in Tabelle 4.1. Diese Möglichkeit ist vor allem für interaktiven Gebrauch geeignet. Sei `$to` ein Spurobjekt, d.h. sei `to` eine Variable mit dem Spurobjektbezeichner als Inhalt:

`$to get` *eventnumber* `?-fetchonly?`

Das Ereignis mit der Nummer *eventnumber* wird gelesen und als Liste ausgegeben. Existiert kein Ereignis mit der spezifizierten Nummer, so wird '`-1`' ausgegeben. War die Leseoperation erfolgreich, so hat der aktuelle Lesezeiger nun den Wert *eventnumber*. Wird jedoch die Option `-fetchonly` benutzt, so bleibt der Lesezeiger unbeeinflußt.

Zu dieser Methode gibt es noch zwei Spielarten:

`$to getnext`

Diese Methode liest das relativ zum aktuellen Lesezeiger nächste Ereignis, d.h. hat der aktuelle Lesezeiger die Position i, so wird das Ereignis mit der Nummer $i+1$ gelesen. War die Operation erfolgreich, so wird das Ereignis wie bei `get` als Liste ausgegeben, der aktuelle Lesezeiger auf $i+1$ gesetzt. Andernfalls wird '-1' zurückgeliefert, und der Lesezeiger bleibt unverändert.

`$to getprev`

Die `getprev`-Methode arbeitet analog, außer daß das relativ zum aktuellen Lesezeiger vorangehende Ereignis gelesen wird, also $i-1$ statt $i+1$.

Zuweisung an ein Array Bei der anderen Variante (`set`), auf ein Ereignis zuzugreifen, wird das Ereignis vom EARL-Interpreter einem assoziativen Array *arr* zugewiesen, so daß $arr(attr_1) = value_1, \ldots, arr(attr_n) = value_n$. Diese Methode ist besonders nützlich für Ereignisverarbeitung innerhalb von Programmen (Skripten), da die Attributwerte eines Ereignisses, z.B. Ereignisreferenzen, als Argument für eine weitere Leseoperation dienen können.

`$to set` *arrayname eventnumber* `?-fetchonly?`

Das Ereignis mit der Nummer *eventnumber* wird gelesen und in der beschriebenen Art dem Array mit dem Namen *arrayname* zugewiesen. Hat vorher kein Array mit diesem Namen existiert, so wird es neu erzeugt. Hat es vorher existiert, so wird es aus Effizienzgründen nicht gelöscht und anschließend neu erzeugt, sondern überschrieben. Es werden alle Elemente mit einem Attributbezeichner als Index entweder mit dem entsprechenden Wert oder, falls der Attributbezeichner nicht zu dem gelesenen Ereignis gehört, mit einem Leerstring belegt. D.h. alle Elemente mit den Indizes *num, node, time, type, enterptr, region, group, src, dest, tag, com, len, sendptr* und *data* erhalten einen neuen Wert. War die Operation erfolgreich, so wird die Nummer des gelesenen Ereignisses zurückgeliefert, andernfalls wird '`-1`' zurückgegeben. Die Position des Lesezeigers verhält sich wie bei `get` und kann ebenfalls durch die Option `-fetchonly` vor Veränderung geschützt werden.

Auch hierzu gibt es Spielarten mit relativer Positionsangabe:

`$to setnext` *arrayname*

Verhält sich analog zu `getnext`.

`$to setprev` *arrayname*

Verhält sich analog zu `getprev`.

Re-Initialisierung Durch die folgende Methode läßt sich der aktuelle Lesezeiger reinitialisieren:

`$to reset`

Der Lesezeiger wird auf 0 zurückgesetzt.

Abfrage von Systemzuständen

Ein Spurobjekt verfügt über Methoden zum Abfragen von Systemzuständen. Der Systemzustand kann immer nur relativ zum aktuellen Lesezeiger abgefragt werden, d.h. steht der aktuelle Lesezeiger auf i, so kann man Angaben über S_i erhalten.

Regionenkeller Die `stack`-Methode dient der Abfrage der Regionenkeller:

`$to stack` *node* `?-sym?`

Der Regionenkeller des Knoten *node* wird als chronologisch geordnete Liste von Ereignisnummern ausgegeben. Das jüngste Ereignis, also das mit der größten Ereignisnummer, steht zuerst, d.h. die Ereignisse sind in umgekehrter Regionenaufrufreihenfolge angeordnet. Die durch die Ereignisnummern definierte Menge entspricht der Menge aller $e \in R_i$ mit $e.node = node$. Es sind die `enter`-Ereignisse aller auf diesem Knoten aktuell existierenden Regionenaktivierungen. Die Liste kann natürlich auch leer sein. Wird die Option `-sym` benutzt, so werden anstelle der Ereignisnummern die entsprechenden Regionennamen ausgegeben.

Nachrichtenschlange Mit Hilfe der `queue`-Methode kann man Informationen über die Nachrichtenschlange erfragen:

`$to queue ?`*dest*`? ?`*src*`?`

`queue` ohne optionale Argumente liefert eine Liste der Ereignisnummern aller `send`-Ereignisse, durch die die aktuell im Verkehr befindlichen Nachrichten verschickt wurden. Es sind genau die Elemente der Menge Q_i. Die Liste ist wie bei der `stack`-Methode chronologisch geordnet. Die Menge kann eingeschränkt werden auf `send`-Ereignisse mit einem bestimmten Zielknoten. Dies wird erreicht, indem man ein optionales Argument *dest* in Form einer Knotennummer angibt, die dann von der `queue`-Methode als Zielknoten interpretiert wird. Man erhält die Nummern aller $e \in Q_i$ mit $e.dest = dest$. Will man davon nur die `send`-Ereignisse aller Nachrichten mit bestimmtem Senderknoten, so gibt man diesen als zweites optionales Argument *src* hinter *dest* an. Man erhält die Nummern aller $e \in Q_i$ mit $e.dest = dest$ und $e.node = src$.

Werden bei den Methoden `stack` und `queue` undefinierte Knotennummern als Argument übergeben, so wird eine Fehlermeldung ausgegeben.

Die `info`-Methode

Die `info`-Methode dient der Ausgabe allgemeiner Information zur Ereignisspur, die üblicherweise im Header der Spurdatei enthalten ist. Die allgemeine Syntax lautet:

`$to info` *what* `?`*args*`?`

Dabei spezifiziert *what* die Art der gewünschten Information und `?`*args* ...`?` sind ggf. erforderliche Parameter. Es folgt eine Auflistung der einzelnen Submethoden:

`$to info filename`

Liefert den Namen der Spurdatei.

`$to info format`

Liefert das Format der Spurdatei. Die Formatbezeichner werden in Kleinschrift ausgegeben - gegenwärtig also entweder '`alog`' oder '`vampir`'.

`$to info eventtypes`

Gibt alle definierten Ereignistypen der Ereignisspur als Liste aus. Die Liste enthält die Standardtypen `enter`, `exit`, `send`, `recv`, gefolgt von spur- bzw. formatspezifischen Typen.

`$to info attributes ?`*type*`?`

Liefert die Attribute eines Ereignistyps als Liste. Z.B. für *type* gleich `exit` wird '`node time type enterptr region group`' zurückgeliefert. Fehlt die Angabe *type*, so wird eine Liste der Attribute aller möglichen Ereignisse zurückgeliefert: '`num node time type enterptr region group src dest tag com len sendptr data`'. Ist der Ereignistyp nicht definiert, so wird eine Fehlermeldung ausgegeben.

`$to info nodecount`

Liefert die Anzahl der benutzten Knoten bzw. Prozesse.

`$to info nodesym` *nodenumber*

Sowohl das ALOG- wie auch das VAMPIR-Format gestatten die Definition von symbolischen Knotennamen. Ist für den Knoten mit der Nummer *nodenumber* ein Symbol definiert, so wird dieses zurückgeliefert. Andernfalls erhält man den String '`node_`i' mit i = *nodenumber*. Existiert kein Knoten mit der spezifizierten Nummer, so wird eine Fehlermeldung ausgegeben.

`$to info regions`

Liefert eine Liste mit allen definierten Regionen.

`$to info groups`

Liefert eine Liste mit allen definierten Gruppen. Bei Formaten wie ALOG, die keine Gruppeneinteilung kennen, erhält man nur die globale Gruppe '`All`'.

`$to info group` *groupname*

Liefert eine Liste der zu der Gruppe *groupname* gehörenden Regionen. Ist die spezifizierte Gruppe nicht definiert, so wird eine Fehlermeldung ausgegeben. Bei Formaten ohne Gruppeneinteilung kann man durch Angabe von '`All`' eine Liste aller definierten Regionen erhalten.

Löschen des Spurobjekts

`$to close`

`close` schließt die zu `$to` gehörende Spurdatei und gibt alle benutzten Ressourcen wieder frei. Das Kommando `$to` wird aus dem Interpreter entfernt.

6.2 Statistikobjekte

Statistikobjekte verwalten eine Folge von Fließkommawerten, z.B. Meßwerte. Die Methoden des Objekts erlauben die Abfrage statistischer Standardinformationen.

Die Folge wird intern im allgemeinen nicht in ihrer Gesamtheit gespeichert. Es werden statt dessen Zustandsvariablen verwaltet, die ausreichend sind, um die durch die Methoden angebotenen Informationen zu berechenen. Damit ist es möglich, Statistiken über beliebig viele Werte zu berechnen, ohne diese explizit speichern zu müssen.

Die Zustandsvariablen bestehen aus dreizehn numerischen Werten, die durch eine spezielle Methode ausgegeben werden können, um daraus ein identisches Statistikobjekt zu erzeugen. Auf diese Weise können Statistikobjekte in einer Textdatei gesichert und zu einem anderen Zeitpunkt wiedererzeugt werden.

6.2.1 Erzeugung von Statistikobjekten

`earl stat` ?*statobj*? ?val_1 ... val_{13}?

Erzeugt ein Statistik-Objekt und generiert ein Kommando namens *statobj*, anhand dessen das erzeugte Objekt angesprochen werden kann. Wird *statobj* weggelassen, so wird für das Kommando automatisch ein Name vergeben. Der Kommandoname wird zurückgeliefert.

Werden die dreizehn optionalen Parameter val_1 .. val_{13} angegeben, so wird ein Statistikobjekt erzeugt, welches durch die dreizehn Werte als Zustandsvariablen charakterisiert ist.

6.2.2 Methoden von Statistikobjekten

Sei `$so` ein Statistikobjekt.

Hinzufügen von Werten

`$so addval` *value*

Nimmt den Fließkommawert *value* in die Folge auf.

Berechnung statistischer Informationen

Die Methoden zur Berechnung statistischer Informationen sind in Tabelle 6.1 zusammengestellt. Sie haben haben die allgemeine Syntax:

`$so` *what*

Die Methode `count` liefert die Anzahl der eingelesenen Elemente als nichtnegative ganze Zahl. Alle anderen Methoden liefern Fließkommawerte. Zur Anwendung der Methoden `q25`, `med` und `q75` muß die Folge mindestens fünf Elemente enthalten, sonst wird eine Fehlermeldung ausgegeben. Die zurückgegebenen Werte sind Schätzungen nach dem P^2-Algorithmus [18]. Die Methoden `sum`, `min`, `mean`, `max` und `var` liefern exakte berechnete Werte; damit keine Fehlermeldung ausgegeben wird, muß die Folge bei `var` mindestens zwei Elemente, sonst ein Element enthalten.

Ausgabe des Objektzustands

`$so` *print*

Gibt die Werte der Zustandsvariablen als Liste von dreizehn numerischen Werten aus. Die ausgegebenen Werte können unter Beibehaltung der Reihenfolge zur Erzeugung einer Kopie des Objekts verwendet werden.

Tabelle 6.1: Methoden zum Berechnen statistischer Informationen

`$so count`	Anzahl der aufgenommenen Werte
`$so sum`	Summe der aufgenommenen Werte.
`$so min`	Minimum der aufgenommenen Werte.
`$so mean`	Mittelwert der aufgenommenen Werte.
`$so max`	Maximum der aufgenommenen Werte.
`$so var`	Varianz der aufgenommenen Werte.
`$so q25`	0,25-Quantil.
`$so med`	0,5-Quantil.
`$so q75`	0,75-Quantil.

6.3 Genauigkeit der Fließkommaangaben

Die Genauigkeit der Fließkommawerte, die von den Methoden der Spur- und Statistikobjekte ausgegeben oder an Variablen zugewiesen werden, kann vom Benutzer beeinflußt werden.

Die den Fließkommawerten entsprechenden Strings werden mit einer festen Anzahl von Nachkommastellen (*fixed format* (C++)) formatiert. Die Anzahl der Nachkommastellen richtet sich dabei nach dem Wert der globalen Variablen `earl_precision`. Dieser muß ganzzahlig sein und zwischen 0 und 17 liegen. Hat `earl_precision` einen unzulässigen Wert, so wird eine Fehlermeldung ausgegeben. Existiert keine Variable `earl_precision`, so richtet sich die Genauigkeit nach der globalen Variablen `tcl_precision`. Existiert diese ebenfalls nicht, so wird ein Defaultwert von 9 Nachkommastellen gewählt.

Lediglich die `print`-Methode der Statistikobjekte benutzt immer die maximal verfügbare Stellenzahl, um beim Kopieren eines Objekts höchstmögliche Genauigkeit zu erreichen. Die Formatierung richtet sich nach der C++-Implementation (*general format*).

6.4 Hilfe

Das earl Kommando verfügt noch über eine Hilfe-Funktion:

`earl help formats`

> Liefert eine Liste alle unterstützten Spurformate. Die Formatbezeichner werden in Kleinschrift ausgegeben und sind zulässig als Formatoption für `earl open`.

`earl help version`

Bewirkt Ausgabe der Version des benutzten EARL-Interpreters.

Kapitel 7

Beispiele

Dieses Kapitel demonstriert die Fähigkeiten von EARL anhand von drei Beispielen. Die Beispiele sind generisch in dem Sinne, daß sie für alle von EARL unterstützten Message-Passing-Ereignisspuren genutzt werden können. Obwohl die Beispiele sehr einfach, d.h. kurz, sind, vollführen sie relativ komplexe Berechnungnen. Die Einfachheit wird ermöglicht durch die im EARL-Spurmodell definierten Abstraktionen einerseits sowie die Mächtigkeit der Tcl-Skriptsprache andererseits.

Bei den Beispielen sollen nicht die gezeigten Spuranalyseverfahren sondern die Art und Weise, wie sich diese Verfahren mit EARL implementieren lassen, im Vordergrund stehen.

Außerdem werden Möglichkeiten zur einfachen grafischen Präsentation mit BLT anhand eines vierten Beispiels vorgeführt.

Die folgenden Beispiele weisen alle ein ähnliche Struktur auf (Abb. 7.1)[1]. In Zeile 2 steht ein spezieller Kommentar, der einem UNIX-System mitteilt, mit welchem Programm bzw. Interpreter das Skript ausgeführt werden soll. In Zeile 5 wird die Spurdatei mit `earl open` geöffnet, der Dateiname wird dem Skript als erster Kommandozeilenparameter übergeben. Von der Liste `argv`, die die Kommandozeilenparameter enthält, wird daher mit `[lindex $argv 0]` das erste Element extrahiert. Das Resultat von `earl open`, der Name des mit dem erzeugten Spurobjekt assoziierten Kommandos, wird in der Variablen `t` gespeichert. Der Gebrauch der eckigen Klammern sorgt in Zeile 5 für die Ausführung der beteiligten Kommandos in der richtigen Reihenfolge, d.h. von innen nach außen.

Es wird davon ausgegangen, daß die Spurdatei im VAMPIR-Format vorliegt, für andere Formate muß hinter `earl open` lediglich eine entsprechende Formatoption angegeben werden.

Anschließend erfolgt ggf. die Initialisierung von im Hauptteil benötigten Variablen. Der Hauptteil besteht aus einer Iterationsschleife, im Rahmen derer die Ereignisspur sequentiell von vorne nach hinten durchgelesen wird. Dabei wird jedes neu gelesene

[1] Die Zeilennummern sind nicht Bestandteil des Skriptcodes

```
1   # Angabe der Ausführungsshell
2   #!/usr/local/bin/earl
3
4   # Öffnen der Spurdatei
5   set t [earl open [lindex $argv 0]]
6
7   # Initialisierung von Variablen
8   [...]
9
10  # Iteration entlang der Ereignisspur
11  while {[$t setnext curr] != -1} {
12
13    if {$curr(type) == "typeId"} {
14
15      # Bearbeite Ereignis vom Typ "typId"
16      [...]
17    }
18    [...]
19  }
20
21  # Ausgabe der Ergebnisse
22  [...]
23
24  # Schließen der Spurdatei
25  $t close
```

Abbildung 7.1: Allgemeine Struktur der Beispiele

Ereignis auf seinen Typ hin geprüft, und je nach Typ werden spezielle Aktionen ausgeführt. Die Aktionen beinhalten in der Regel Rücksprünge zu vormals gelesenen Ereignissen.

Die Iteration ist durch eine `while`-Schleife implementiert. Vor jeder Ausführung des Rumpfes wird mit der `setnext`-Methode, angewandt auf das Spurobjekt `$t`, in Zeile 11 das jeweils nächste Ereignis gelesen, angefangen beim ersten Ereignis der Spur. Wird das Dateiende erreicht, so liefert `setnext` '`-1`' zurück, was das Verlassen der Schleife bewirkt. Die Attributwerte des jeweils gelesenen Ereignisses werden von `setnext` dem assoziativen Array `curr` zugewiesen.

Im Schleifenrumpf wird der Typ des aktuellen Ereignisses durch eine `if`-Anweisung in Zeile 13 geprüft, indem das *type*-Attribut durch Auslesen des Arrayelements `curr(type)` mit dem gewünschten Typbezeichner verglichen wird. Ist das Ereignis vom spezifizierten Typ, so können geeignete Aktionen im Rumpf der `if`-Anweisung ausgeführt werden. Ggf. können mehrere solche Typabfragen mit zugeordneten Ak-

tionen vorkommen.

Nach Beendigung der `while`-Schleife werden die Ergebnisse ausgegeben oder weiterverarbeitet. Schließlich wird in Zeile 25 die Spurdatei mit der `close`-Methode geschlossen.

7.1 Regionenstatistik

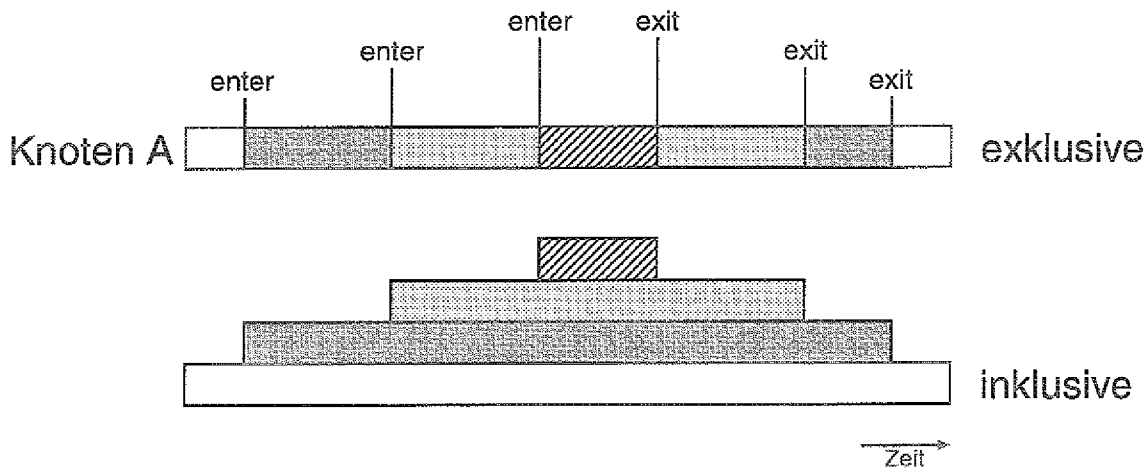


Abbildung 7.2: Regionenverweildauer inklusive/exklusive eingeschlossener Regionenaktivierungen

Das erste Beispiel behandelt die häufig gestellte Aufgabe, die Zeiten, die in den einzelnen Regionen verbracht wurden, über die gesamte Programmlaufzeit hinweg aufzusummieren. Die Zeiten sollen einmal inklusive und einmal exklusive der eingeschlossenen Regionenaktivierungen und außerdem für jeden Knoten einzeln berechnet werden.

Abb. 7.2 zeigt die Zeitlinie für den Knoten A. Zu Beginn befindet sich das Programm in der weißen Region. Während es sich dort befindet, wird schrittweise erst die dunkelgraue, dann die hellgraue und schließlich die gestreifte Region betreten. Dabei wird der Regionenkeller des Knotens A immer größer. Danach werden die Regionen in der Reihenfolge des Eintretens wieder verlassen, bis sich das Programm wieder ausschließlich in der weißen Region befindet.

Die Muster im oberen Teil der Abbildung markieren auch gleichzeitig die Zeit, die während der jeweiligen Regionenaktivierung exklusive eingeschlossener Regionenaktivierungen verbraucht wurde. Im unteren Teil der Abbildung sind die Inklusivzeiten dargestellt. Die Schichtung der Pyramide entspricht auch gleichzeitig dem Zustand des Regionenkellers zu den einzelnen Zeitpunkten.

```
1   #!/usr/local/bin/earl
2
3   set t [earl open [lindex $argv 0]]
4
5   set n [$t info nodecount]
6
7   for {set i 0} {$i<$n} {incr i} {
8
9     foreach r [$t info regions] {
10
11      set excl($i,$r)  0
12      set incl($i,$r)  0
13    }
14  }
15
16  while {[$t setnext curr] != -1} {
17
18    if {$curr(type) == "exit"} {
19
20      $t set enter $curr(enterptr) -fetchonly
21
22      set diff [expr $curr(time) - $enter(time)]
23
24      set index "$curr(node),$curr(region)"
25
26      set incl($index) [expr $incl($index) + $diff]
27      set excl($index) [expr $excl($index) + $diff]
28
29      set p [lindex [$t stack $curr(node) -sym] 0]
30
31      if {$p != ""} {
32        set excl($curr(node),$p) [expr $excl($curr(node),$p) - $diff]
34      }
35    }
36  }
37  [...] # Ausgabe
38  $t close
```

Abbildung 7.3: Berechnung einer Regionenstatistik

Abb. 7.3 zeigt ein Skript, welches die gewünschten Zeiten berechnet. Die Spur wird dabei sequentiell von vorne nach hinten gelesen. Sobald das Skript auf ein exit-Ereignis stößt, wird via Attribut *enterptr* das korrespondierende enter-Ereignis bestimmt, um somit die gerade beendete Regionenaktivierung in Form der beiden

Ereignisse zu erfassen. Der zeitliche Abstand wird nun sowohl zur Inklusivzeit als auch zur Exklusivzeit der entsprechenden Region auf diesem Knoten addiert. Durch eine Abfrage des Regionenkellers wird anschließend die umschließende Regionenaktivierung ermittelt. Von dem Exklusivwert dieser Region wird nun die oben errechnete Zeitdifferenz abgezogen, da diese Zeit in einer eingeschlossenen Regionenaktivierung verbracht wurde.

Die Zeilen 1 bis 3 sind analog zu Abb. 7.1. In Zeile 5 wird die Anzahl der benutzten Knoten bestimmt und der Variablen `n` zugewiesen. In den Zeilen 7 bis 14 werden die beiden assoziativen Arrays `incl` und `excl` fur die Inklusiv- bzw. Exklusivzeiten mit `0` initialisiert. Für jeden Knoten und jede Region wird jeweils ein Element angelegt. Es werden dabei zweidimensionale Arrays simuliert, indem der Index aus der Knotennummer und dem Regionenbezeichner zusammengesetzt wird. Die äußere Schleife iteriert über die Knotennummern, während die innere Schleife über die Liste aller definierten Regionen iteriert. Diese wird mit `[$t info regions]` bestimmt.

Die Zeilen 16 bis 36 beschreiben die sich anschließende Iteration über die Ereignisse der Spur. Sobald ein `exit`-Ereignis gefunden wird, wird in Zeile 20 das korrespondierende `enter`-Ereignis mit der `set`-Methode an das Array `enter` zugewiesen. Die Nummer des `enter`-Ereignisses erhält man mit `$curr(enterptr)`. Die Verwendung der Option `-fetchonly` stellt sicher, daß der aktuelle Lesezeiger erhalten bleibt, und somit in Zeile 16 beim nächsten Schleifendurchlauf auch wirklich das nächste Ereignis gelesen wird.

In Zeile 22 wird die zeitliche Differenz berechnet, indem die Attributwerte des *time*-Attributs voneinander subtrahiert werden. Die Differenz wird der Variablen `diff` zugewiesen. In Zeile 24 wird ein Indexstring aus der Knotennummer und dem Regionenbezeichner der durch das aktuelle Ereignis verlassenen Region gebildet. Schließlich wird die errechnete Zeitdifferenz in den Zeilen 26 und 27 zu den Werten der entsprechenden Arrayelemente hinzuaddiert.

In Zeile 29 wird der Regionenbezeichner zu der umschließenden Regionenaktivierung, d.h. der Name der Region, in die durch das aktuelle `exit`-Ereignis zurückgekehrt wurde, bestimmt, indem der Regionenkeller des aktuellen Knotens durch die `stack`-Methode unter Verwendung der Option `-sym` als Liste von Regionenbezeichnern ermittelt wird. Der gesuchte Regionenbezeichner ist wegen der chronologischen Ordnung der Liste das erste Element und wird mit `[lindex [...] 0]` isoliert und der Variablen `p` zugewiesen.

War der Keller nicht leer, d.h. wurde nicht auf Toplevel zurückgekehrt, so wird die Verweildauer `diff` von der Exklusivzeit des zur umschließendenden Regionaktivierung gehörenden Arrayelements von `excl` abgezogen.

Nach Beendigung der Schleife stehen die gewünschten Werte in den Arrays `excl` und `incl` zur Verfügung.

7.2 Suche nach Leistungsengpaß

Das zweite Beispiel demonstriert die durch EARL gebotenen Mittel zur Bewältigung von Nicht-Standardproblemen – insbesondere zur Erkennung komplexer Ereignismuster.

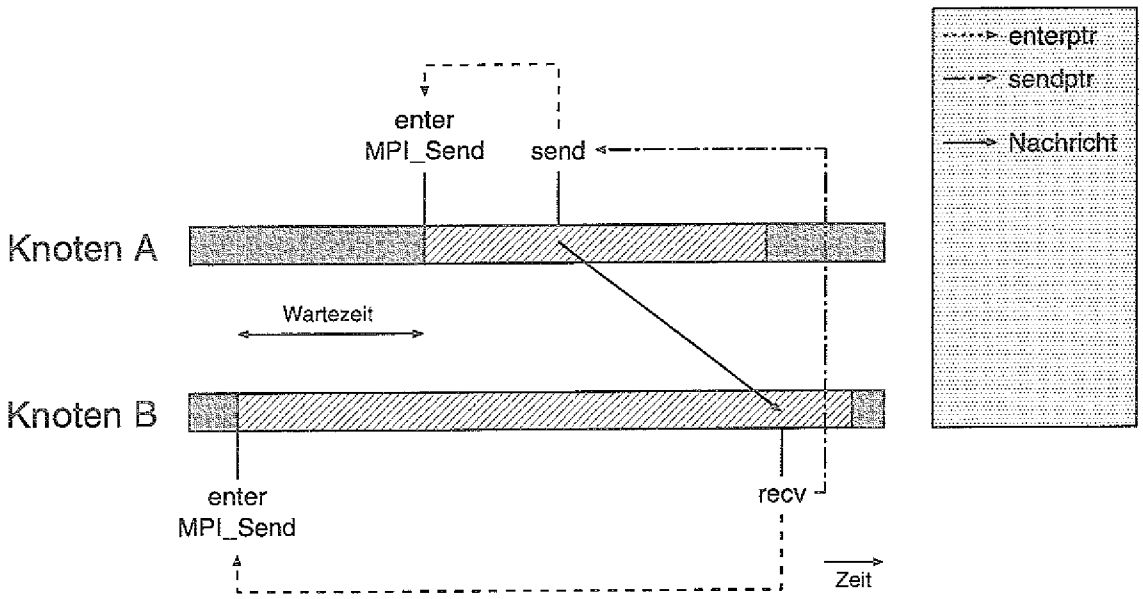


Abbildung 7.4: Verfrühtes MPI_Recv

Es beschreibt die Suche nach einem Leistungsengpaß, der durch ein verfrühtes `MPI_Recv` entsteht (Abb 7.4). Wegen der blockierenden Eigenschaften von `MPI_Recv` muß das Programm auf einem Knoten solange warten, bis die zu empfangende Nachricht durch das korrespondierende `MPI_Send` verschickt wurde. Die verlorene Wartezeit ist dabei mindestens so groß wie die Zeit, um die `MPI_Recv` früher aufgerufen wurde als `MPI_Send`. Dieser Zeitraum wird durch das folgende Beispielskript berechnet (Abb. 7.5).

Sobald das Skript beim Lesen der Spur auf ein `recv`-Ereignis stößt, werden anhand der Attribute *sendptr* und *enterptr* die `enter`-Ereignisse von `MPI_Send` und `MPI_Recv` ermittelt (Abb. 7.4). Die zeitliche Differenz wird berechnet und, falls eine Wartezeit vorliegt, zu einer Summenvariablen hinzuaddiert. Bei Erreichen des Spurendes enthält diese die Summe aller im Programm auf allen Knoten derartig entstandenen Wartezeiten.

Nach Erzeugung des Spurobjekts wird in Zeile 5 die Summenvariable `sum_wasted` mit `0` initialisiert. Von Zeile 7 bis 29 findet wieder die übliche Iteration durch die Ereignisspur statt. Sobald das Skript in Zeile 9 auf ein `recv`-Ereignis stößt, so holt es sich in Zeile 11 mit Hilfe des Attributs *enterptr* das `enter`-Ereignis zu der Regionenaktivierung, in der sich das Programm auf dem entsprechenden Knoten gerade befindet, und weist es dem Array `recv_start` zu. Auch hier unterbindet die Option

```
1   #!/usr/local/bin/earl
2
3   set t [earl open [lindex $argv 0]]
4
5   set sum_wasted 0
6
7   while {[$t setnext curr] != -1} {
8
9     if {$curr(type) == "recv"} {
10
11      $t set recv_start $curr(enterptr) -fetchonly
12
13      if {$recv_start(region) != "MPI_Recv"} {
14        continue
15      }
16      $t set send $curr(sendptr) -fetchonly
17
18      $t set send_start $send(enterptr) -fetchonly
19
20      if {$send_start(region) != "MPI_Send"} {
21        continue
22      }
23      set wasted [expr $send_start(time) - $recv_start(time)]
24
25      if {$wasted > 0} {
26        set sum_wasted [expr $sum_wasted + $wasted]
27      }
28    }
29  }
30  puts "[$t info filename]: $sum_wasted sec wasted."
31  $t close
```

Abbildung 7.5: Suche nach Leistungsengpaß

`-fetchonly` eine Beeinflussung des für `setnext` benötigten aktuellen Lesezeigers. In Zeile 13 wird getestet, ob die Nachricht auch tatsächlich von `MPI_Recv` empfangen wurde. Anderfalls wird der Rest des Schleifenrumpfs übersprungen und die Iteration mit `continue` fortgesetzt.

In Zeile 16 wird anhand des Attributs *sendptr* das **send**-Ereignis der empfangenen Nachricht ermittelt und dem Array `send` zugewiesen. `send(enterptr)` in Zeile 18 schließlich enthält einen Verweis auf den Aufruf der sendenden Region. Das dem Aufruf entsprechende **enter**-Ereignis wird dem Array `send_start` zugewiesen. Das Attribut *region* wird auch hier in Zeile 20 daraufhin getestet, ob die Nachricht tatsächlich von `MPI_Send` und nicht etwa `MPI_Broadcast` ausging.

In Zeile 23 wird schließlich die Wartezeit (Abb. 7.4) als zeitliche Differenz der beiden enter-Ereignisse berechnet und der Variablen `wasted` zugewiesen. Ist sie positiv, d.h. wurde `MPI_Recv` vor `MPI_Send` aufgerufen, so wird sie in Zeile 26 zu `sum_wasted` hinzuaddiert.

In Zeile 30 wird schließlich die errechnete Summe unter Angabe des Spurdateinamens ausgegeben. Der Dateiname ergibt sich dabei aus `[$t info filename]`.

7.3 Programmvalidierung

Das dritte Beispiel zeigt, wie EARL genutzt werden kann, um Programmierfehler in Message-Passing-Programmen zu finden. Das Beispiel wurde der *Grindstone Test Suite for Parallel Performance Tools* [16] entnommen und behandelt das Problem des Nachrichtenaustauschs in falscher Reihenfolge. Das Problem entsteht, wenn ein Empfänger Nachrichten in einer bestimmten Reihenfolge erwartet, der Sender die Nachrichten aber in einer anderen Reihenfolge verschickt.

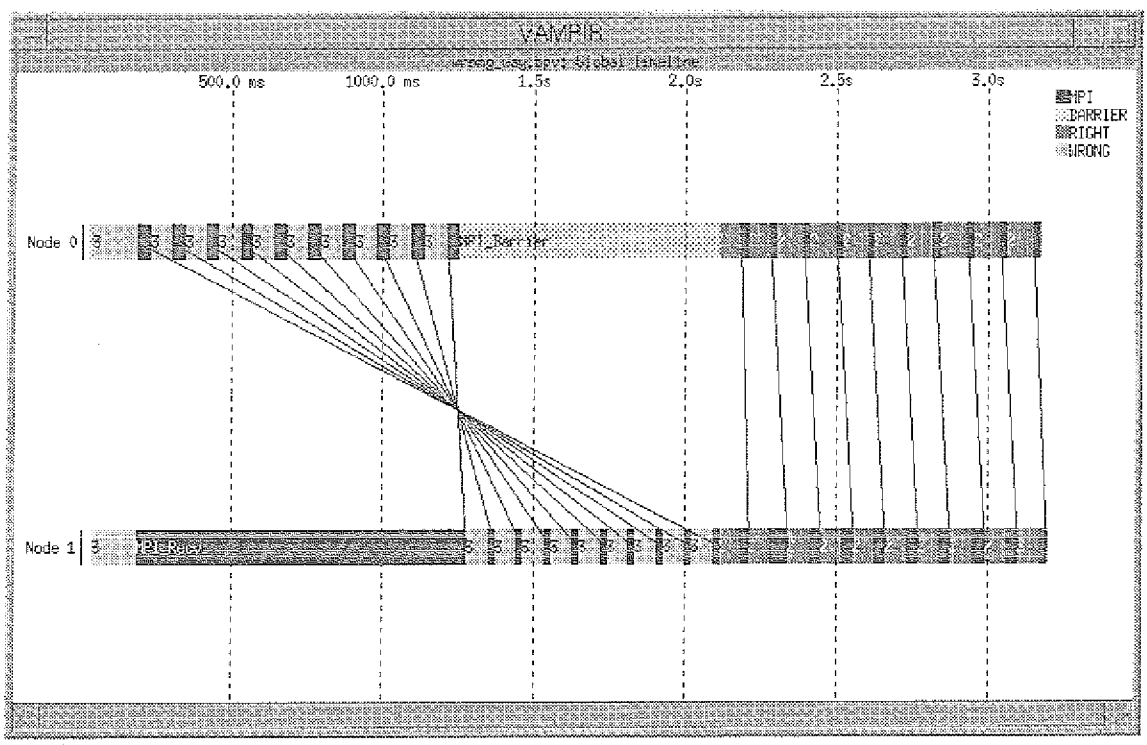


Abbildung 7.6: Nachrichtenaustausch in falscher und in richtiger Reihenfolge

Abb. 7.6 zeigt ein extremes Beispiel als VAMPIR-Grafik. Im ersten Teil des Programms verarbeitet Knoten 1 Nachrichten in genau der umgekehrten Reihenfolge, wie sie von Knoten 0 gesendet werden. Die Nachrichten in der korrekten Reihenfolge zu empfangen, wie im zweiten Teil der Abbildung geschehen, beschleunigt nicht nur

das Programm sondern benötigt auch weit weniger Speicherplatz zur Pufferung von unverarbeiteten Nachrichten.

```
1   #!/usr/local/bin/earl
2
3   set t [earl open [lindex $argv 0]]
4
5   while {[$t setnext curr] != -1} {
6
7     if {$curr(type) == "recv"} {
8
9       foreach send [$t queue $curr(node) $curr(src)] {
10
11        if {$send < $curr(sendptr)} {
12
13          puts "Received message in wrong order"
14          puts "on node $curr(node) at $curr(time) sec."
15          puts "Call Stack: [$t stack $curr(node) -sym] "
16        }
17      }
18    }
19  }
20  $t close
```

Abbildung 7.7: Programmvalidierung

Das zugehörige Skript (Abb 7.7) geht wie folgt vor. Sobald es auf ein recv-Ereignis trifft, wird die Nachrichtenschlange zum Zeitpunkt dieses Ereignisses untersucht. Befindet sich darin eine vom Sender der empfangenen Nachricht an den Knoten des recv-Ereignisses adressierte Nachricht, die älter ist als die gerade empfangene Nachricht, so wird eine Meldung ausgegeben. Das Alter der Nachrichten kann wegen der chronologischen Ordnung innerhalb der Ereignisspur anhand der Ereignisnummern der send-Ereignisse miteinander verglichen werden.

Wird in Zeile 7 ein recv-Ereignis entdeckt, so wird zunächst in Zeile 9 die Nachrichtenschlange, eingeschränkt auf Nachrichten an den Knoten des aktuellen Ereignisses (`$curr(node)`), die vom Quellknoten der gerade empfangenen Nachricht (`$curr(src)`) verschickt wurden, mit Hilfe der `queue`-Methode abgefragt. `queue` liefert die Nachrichtenschlange als Liste von Ereignisnummern der send-Ereignisse, durch die die Nachrichten in Verkehr gebracht wurden. Über diese Liste wird durch `foreach send [...]` von Zeile 9 bis 17 iteriert.

Befindet sich in der Liste ein send-Ereignis, welches älter ist, d.h. eine kleinere Ereignisnummer hat, als das send-Ereignis der gerade empfangenen Nachricht (`$curr(sendptr)`), so wird eine Meldung ausgegeben. Die Meldung enthält den

Knoten und den Zeitpunkt des aktuellen Ereignisses. Außerdem wird der Regionenkeller des aktuellen Knotens als Liste von Regionenbezeichern ausgegeben, um den Zustand der Programmausführung zum Zeitpunkt des Fehlers zu beschreiben.

7.4 Grafik

Das letzte Beispiel soll zeigen, daß man unter Zuhilfenahme des Zusatzpaketes BLT mit einfachen Mitteln eine Präsentation der mit EARL gewonnenen Daten programmieren kann. Es wurde BLT Version 2.3 verwendet.

Das Beispiel (Abb. 7.9) bestimmt die Datenmenge, die einem als Parameter zu spezifizierenden Empfängerknoten von den anderen Knoten zur Laufzeit in Form von Nachrichten zugesandt wurde. Die Menge wird für jeden Senderknoten einzeln berechnet. Sobald das Skript auf ein recv-Ereignis trifft, wird nachgesehen, ob es auf dem angegebenen Empfängerknoten stattgefunden hat. Falls ja, wird der Senderknoten bestimmt und die Länge der Nachricht auf den Wert für diesen Senderknoten aufaddiert. Anschließend werden die Werte für die einzelnen Senderknoten als Balken in einem Diagramm dargestellt (Abb 7.8).

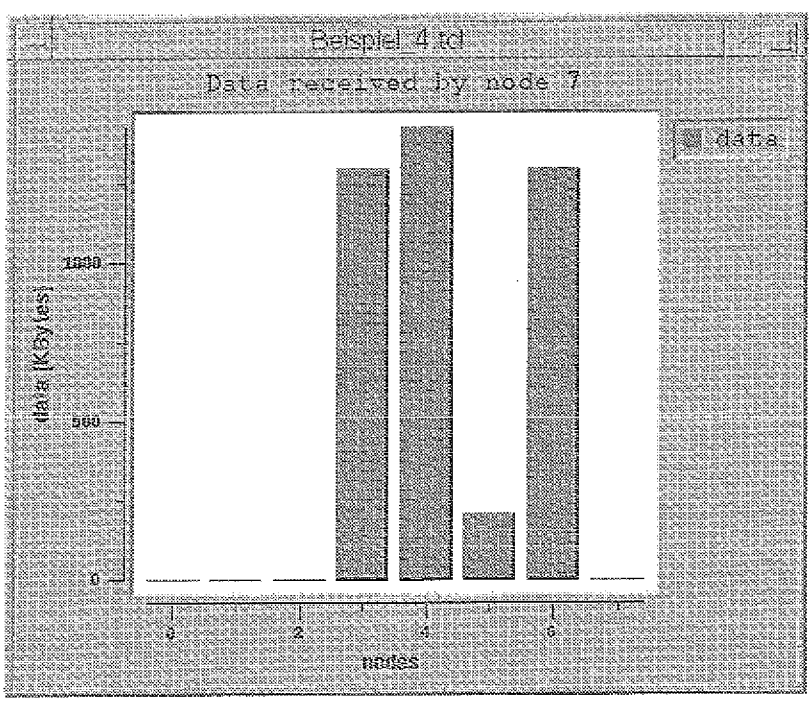


Abbildung 7.8: An Knoten 7 verschickte Datenmengen in KBytes

Anders als bei den vorherigen Beispielen wird hier der Tcl/Tk-Standardinterpreter `wish` als ausführender Interpreter in Zeile 1 angegeben. Die Zeilen 3 und 4 sorgen

dafür, daß die Packages von EARL und ebenso von BLT bei der ersten Benutzung dynamisch geladen werden.

```
1   #!/usr/local/wish
2
3   package require earl
4   package require blt
5
6   set t [earl open [lindex $argv 0]]
7   set receiver [lindex $argv 1]
8
9   set n [$t info nodecount]
10  vector nodeVec($n)
11  vector dataVec($n)
12
13  for {set i 0} {$i<$n} {incr i} {
14    set nodeVec($i) $i
15  }
16  while {[$t setnext curr] != -1} {
17    if {$curr(type) == "recv" && $curr(node) == $receiver} {
18    set dataVec($curr(src)) \
19      [expr $dataVec($curr(src)) + ($curr(len) / 1024.0)]
20    }
21  }
22  barchart .b -plotbackground white
23  .b grid configure -hide true
24
25  .b axis create bytes -title {data [KBytes]}
26  .b axis create nodes -title nodes
27
28  .b xaxis use nodes
29  .b yaxis use bytes
30
31  .b configure -title "Data received by node $receiver"
32
33  .b element create data -xdata nodeVec -ydata dataVec
34  .b element configure data -mapx nodes -mapy bytes
35  .b element configure data -fg gray -bg black
36
37  table . 0,0 .b -fill both
38
39  $t close
```

Abbildung 7.9: Grafikprogrammierung

Das Skript verlangt den Dateinamen der zu untersuchenden Ereignisspur als ersten (Zeile 6) und den Empfängerknoten als zweiten Parameter (Zeile 7). Der Empfängerknoten wird der Variablen `receiver` zugewiesen. In Zeile 9 wird die Variable n mit der Anzahl der benutzten Knoten belegt. In den Zeilen 10 und 11 werden zwei BLT-Vektoren `nodeVec` für die Knotennummern und `dataVec` für die von den Knoten verschickten Datenmengen initialisiert. Von Zeile 13 bis 15 werden die Knotennummern in aufsteigender Reihenfolge in die Vektorelemente `nodeVec(0)` bis `nodeVec(n-1)` eingetragen.

Zeile 16 bis 21 enthält die Iterationsschleife durch die Ereignisspur. Sobald ein passendes recv-Ereignis gefunden wird, wird die Länge der Nachricht (`$curr(len)`), die in Bytes angegeben ist, in Kilobytes umgerechnet und auf das dem Senderknoten entsprechende Vektorelement (`dataVec($curr(src))`) aufaddiert. Der Backslash am Ende von Zeile 18 bewirkt die Fortsetzung des Kommandos in der nächsten Zeile.

In Zeile 22 wird ein Barchartwidget `.b` mit weißem Hintergrund definiert. Ein Hintergrundgitter wird nicht benutzt (Zeile 23). In den Zeilen 25 und 26 werden Koordinatenachsen für Knoten und Datenmengen mit geeigneter Beschriftung definiert. Die Knotenachse wird daraufhin als x-Achse (Zeile 28) und die Datenachse als y-Achse (Zeile 29) deklariert. In Zeile 31 wird für das Balkendiagramm eine Überschrift festgelegt.

Schließlich werden aus dem Vektor mit den Knotennummern `nodeVec` und dem Vektor mit den Datenmengen `dataVec` in Zeile 33 die darzustellenden Balken gebildet. Zeile 34 paßt die Skalierung der Koordinatenachsen entsprechend an, während in Zeile 35 die Balkenfarben festgelegt werden. Zuletzt wird das Barchartwidget `.b` dem `table`-Geometriemanager zur Darstellung des Diagramms übergeben.

BLT bietet noch eine Vielzahl weiterer Möglichkeiten, wie z.B. x-y-Diagramme, die hier der Einfachheit halber nicht gezeigt wurden.

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

Wegen der Schwierigkeit, das Laufzeitverhalten paralleler Anwendungen im voraus zuverlässig zu planen, vollzieht sich der Entwicklungsprozeß in ständigem Wechsel aus experimenteller Beobachtung und anschließender Modifikation, bis das Programm schließlich allen Anforderungen genügt.

Besonders bei Message-Passing-Systemen hat sich Tracing als nützliche Methode zur Bewertung des Laufzeitverhaltens etabliert. Die gegenwärtig zur Analyse der dabei aufgezeichneten Ereignisspuren eingesetzten Werkzeuge bieten jedoch nur unzureichende Unterstützung bei der automatischen Identifikation von Leistungsengpässen, bei der Analyse nach anwendungsspezifischen Gesichtspunkten sowie bei der automatischen Durchführung von Meßreihen.

Die im Rahmen dieser Arbeit enwickelte Umgebung EARL (Event Analysis and Recognition Language) bietet durch ihre High-Level-Spuranalysesprache eine komfortable Plattform für die einfache und schnelle Erstellung neuer Werkzeuge zur Analyse von Message-Passing-Ereignisspuren. Die Werkzeuge werden als Skripten in der EARL-Sprache verfaßt und durch den EARL-Interpreter ausgeführt.

Ein großer Teil der Leistungsfähigkeit von EARL resultiert aus der abstrakten Sicht, die dem Benutzer auf die Ereignisspur gewährt wird. Während sich der Benutzer voll auf seine Bewertungsstrategie konzentrieren kann, sorgt der EARL-Interpreter für die Dekodierung verschiedener Spurformate, wahlfreien Zugriff auf die einzelnen Ereignisse sowie die Abbildung der unterstützten Spurformate auf das einheitliche EARL-Spurmodell, auf welchem die in der EARL-Sprache geschriebenen Skripten operieren. Dadurch wird dem Benutzer ermöglicht, die Skripten unabhängig von einem speziellen Spurformat zu verfassen.

Das EARL-Spurmodell betrachtet eine Ereignisspur als Folge von Ereignissen bestimmten Typs. Ein Typ ist dabei durch eine Menge von Attributen definiert, wobei

bestimmte Attribute wie z.B. für den Zeitpunkt und den Ort des Ereignisses allen Typen gemeinsam sind. Es gibt vier vordefinierte Ereignistypen: das Betreten und Verlassen von Programmregionen (enter, exit) sowie das Senden und Empfangen (send, recv) von Nachrichten. Je nach verwendetem Spurformat kann es noch weitere Typen geben.

Die Konzepte Regionen und Nachrichten werden durch spezielle Attribute unterstützt, anhand derer sich korrespondierende send- und recv-Ereignisse sowie das enter- und exit-Ereignis derselben Regionenaktivierung erkennen lassen. Weiterhin läßt sich das enter-Ereignis derjenigen Regionenaktivierung bestimmen, innerhalb derer ein Ereignis stattfindet. Schließlich definiert EARL noch für jede Position der Ereignisspur den Zustand der Regionenaufrufkeller als Menge von enter-Ereignissen der aktuell existierenden Regionenaktivierungen sowie den Zustand der Nachrichtenschlange als Menge von send-Ereignissen im Verkehr befindlicher Nachrichten. All diese Eigenschaften erlauben die einfache Verarbeitung komplexer Ereignismuster, die durch Regionenaktivierungen, Nachrichten und Systemzustände bestimmt sind.

Der EARL-Interpreter wurde als Erweiterung des Tcl-Interpreters erstellt. Daher umfaßt die EARL-Sprache die volle Mächtigkeit der Tcl-Skriptsprache. Die Erweiterungen wurden in C++ implementiert und unterstützen folgende Funktionalität:

- Öffnen und Schließen von Ereignisspuren,
- Wahlfreier Zugriff auf die Ereignisse und deren Attribute,
- Abfragen von Nachrichtenschlange und Regionenkeller,
- Abfrage allgemeiner Information zur gerade bearbeiteten Ereignisspur wie z.B. Anzahl der verwendeten Rechnerknoten,
- Berechnen von Statistiken.

Ereignisspuren werden von Benutzer an der Programmierschnittstelle als Spurobjekte wahrgenommen, auf die Methoden zur Ausführung entsprechender Operationen angewandt werden. Hierdurch gestaltet sich die Programmierung besonders intuitiv.

EARL ist ein sehr flexibles und einfach zu bedienendes Meta-Tool für Experten. Es erlaubt die einfache Implementation generischer oder anwendungsspezifischer Spuranalysewerkzeuge. Aufgrund seiner Programmierbarkeit kann es in zahlreichen Anwendungsfeldern eingesetzt werden:

- Berechnung von Performance-Indizes und Statistiken aller Art,
- Suche nach Leistungsengpässen,
- Programmvalidierung,

- Automatische Durchführung von Experimenten mit vergleichender Analyse von Ereignisspuren,
- Visualisierung von Leistungsdaten,
- Anwendungsspezifische Varianten dieser Aufgaben.

8.2 Ausblick

Es wird angestrebt, eine Bibliothek generischer EARL-Skripte zu entwickeln, die von Programmierern zu Analyse ihrer parallelen Anwendungen eingesetzt werden kann.

Außerdem ist die Integration weiterer Spurformate durch die Implemenation zusätzlicher Dekoderklassen geplant, um EARL für die Kooperation mit anderen Werkzeugen attraktiver zu machen.

Der im Rahmen dieser Arbeit entwickelte Prototyp konzentriert sich gegenwärtig nur auf die Optimierung von Message-Passing-Programmen. Die Architektur von EARL wurde jedoch bewußt flexibel gehalten, um in Zukunft auch andere Programmiermodelle unterstützen zu können.

Am Zentralinstitut für Angewandte Mathematik des Forschungszentrums Jülich wurde damit begonnen, die Umgebung KOJAK (**K**it for **O**bjective **J**udgement and **A**utomatic **K**nowledge-based detection of bottlenecks) [12] zur automatischen Erkennung von Standardleistungsengpässen in parallelen oder verteilten Anwendungen zu entwickeln. Das Projekt dient vor allem der Erforschung von Methoden zur Identifikation und Repräsentation von Leistungsengpässen. Hierbei soll EARL zur Implementation und Evaluation neuer Verfahrensweisen zum Einsatz kommen.

Anhang A

Das ALOG-Spurformat

Dieser Anhang beschreibt, wie das ALOG-Spurformat durch die Dekoderklasse `Earl_SeqAlogTrace` auf das EARL-Spurmodell abgebildet wird. EARL unterstützt das ALOG-Format in der Ausprägung, wie sie von dem grafischen Spuranalysewerkzeugen Nupshot in der unter [19] verfügbaren Version verwendet wird.

Eine ALOG-Spurdatei ist eine ASCII-Datei, die Konfigurations- und Ereignisrecords enthält. Ereignisrecords stehen für konkrete Ereignisse, während Konfigurationsrecords allgemeine Eigenschaften des protokollierten Programmablaufs beschreiben und zur korrekten Dekodierung der Ereignisrecords notwendige Informationen enthalten. Die Records werden durch Zeilen der Form

```
type process task data cycle timestamp [comment]
```

repräsentiert, wobei das Feld `type` den Recordtyp, als ganze Zahl kodiert, angibt. Die restlichen Felder enthalten recordspezifische Informationen. Eine genaue Darstellung findet sich in der Quelltextdatei `alog.h` von [19].

In den folgenden Abschnitten ist den Erläuterungen zu den einzelnen Recordtypen immer der Inhalt des `type`-Feldes, d.h. die Typnummer, zusammen mit einem kurzen Kommentar vorangestellt (`type` : Kommentar). Die Bezeichner der Recordfelder sind in Schreibmaschinenschrift gesetzt.

A.1 Konfigurationsrecords

In diesem Abschnitt ist die Interpretation aller von EARL berücksichtigten Konfigurationsrecordtypen erklärt. EARL geht davon aus, daß sich alle Konfigurationsrecords vor dem ersten Ereignisrecord im Header der Spurdatei befinden.

-3 : Anzahl der benutzten Prozesse Die Anzahl der benutzten Prozesse wird von EARL als Inhalt des Feldes `data` erwartet.

-9 : Definition eines spurspezifischen Ereignistyps Enthält eine Spurdatei z.B. einen Konfigurationsrecord

```
-9 0 0 5 0 0 start,
```

so wird ein ALOG-Ereignisrecord wie z.B.

```
5 0 0 4 0 2025436865
```

als spurspezifisches EARL-Ereignis mit den Attributen *type* = start und *data* = 4 interpretiert. D.h. Konfigurationsrecords dieses Typs definieren einen spurspezifischen ALOG-Ereignisrecordtyp, dessen Typnummer als Inhalt des Konfigurationsrecordfeldes `data` angegeben wird. Ereignisrecords dieses ALOG-Ereignisrecordtyps werden als Ereignisse eines spurspezifischer EARL-Ereignistyps interpretiert. Die entsprechenden EARL-Ereignisse enthalten den ganzzahligen Inhalt des Ereignisrecordfeldes `data` als Attributwert für das *data*-Attribut. Als Wert des *type*-Attributs, d.h. als Typbezeichner, wird der Inhalt des Feldes `comment` des definierenden ALOG-Konfigurationsrecords verwendet.

-11 : Überrollpunkt der Uhr Gibt den Überrollpunkt der Uhr als Feldinhalt von `timestamp` an, d.h. den Zeitpunkt, nach dessen Erreichen die Uhr wieder bei 0 zu zählen beginnt. Die Verwendung dieses Records ist im Abschnitt über Ereignisrecords dargestellt.

-13 : Definition von enter und exit-Ereignis zu einer Region Enthält eine Spurdatei z.B. einen Konfigurationsrecord

```
-13 0 1 2 0 0 green:boxes Region_A,
```

so werden ALOG-Ereignisrecords wie z.B.

```
1 0 0 0 0 2025437342
2 0 0 0 0 2025438566
```

als enter- und exit-Ereignisse zu der Region mit Namen `Region_A` interpretiert. D.h. dieser Konfigurationsrecord definiert zwei ALOG-Ereignisrecordtypen, die von EARL als enter- und exit-Ereignisse der im Feld `comment` spezifizierten Region angesehen werden. Das `task`-Feld des Konfigurationrecords enthält die ALOG-Typnummer für das enter-, das `data`-Feld die Typnummer für das exit-Ereignis.

Das *group*-Attribut der entsprechenden EARL-Ereignisse wird auf 'All' gesetzt, da das ALOG-Format kein Gruppenkonzept unterstützt. Das *region*-Attribut erhält als Wert natürlich den in `comment` angegebenen Regionenbezeichner. Eine in `comment` ggf. vor den Regionenbezeichner gestellte Angabe zu Darstellungsfarbe und -muster wird von EARL am enthaltenen Doppelpunkt erkannt und ignoriert.

Zu beachten ist, daß ALOG-Ereignistypdefinitionen dieser Art eine spurspezifische Ereignistypdefinition (siehe oben) immer überschreiben, gleichgültig in welcher Reihenfolge die Konfigurationrecords im Header der Spurdatei enthalten sind.

-15 : Definition eines Knotensymbols Dieses Record definiert einen symbolischen Namen für einen Knoten. Z.B. vereinbart der Record

```
-15 7 0 0 0 0 Server
```

für den Knoten 7 den symbolischen Namen 'Server'. Der Name ist als Inhalt des Feldes `comment` angegeben. Er kann von EARL mit der Spurobjektmethode '`info nodesym` *nodenumber*' abgefragt werden.

Alle durch Konfigurationsrecords definierten ALOG-Ereignirecordtypen müssen nichtnegative Typnummern besitzen.

A.2 Ereignisrecords

Bei allen Ereignisrecords werden die Feldinhalte von `process`, `cycle` und `timestamp` auf gleiche Weise interpretiert. `process` gibt den Knoten an, d.h. der Attributwert von *node* des EARL-Ereignisses nimmt den hier angegebenen Wert an. Der Attributwert von *time* wird aus der Summe von `timestamp` und dem Produkt aus `cycle` und dem durch den entsprechenden Konfigurationsrecord spezifizierten Überrollpunkt der Uhr gebildet. Dieser Wert wird als Zeit in Mikrosekunden interpretiert und entsprechend in Sekunden umgerechnet: $time = (\texttt{timestamp} + (\texttt{cycle} * \text{Überrollpunkt})) * 0.000001$.

Der Inhalt des `type`-Feldes bei benutzerdefinierten Ereignisrecordtypen wird auf die im vorigen Abschnitt beschriebene Weise behandelt. Neben diesen werden von EARL jedoch noch zwei vordefinierte Recordtypen berücksichtigt:

-101 : Versenden einer Nachricht Ereignisrecords dieses Typs werden als `send`-Ereignisse interpretiert. Das Attribut *dest* erhält als Wert den Feldinhalt von `data`. Das Feld `comment` enthält nach der Formatbeschreibung die ganzzahlige Nachrichtenkennung gefolgt von der Nachrichtenlänge in Bytes. Diese werden direkt als Attributwerte von *tag* und *len* übernommen. Das *com*-Attribut wird auf '-1' gesetzt, da ALOG keine Kommunikatorangaben unterstützt. Z.B. steht

```
-101 1 0 2 0 2025439346 193 100
```

für das Senden einer Nachricht der Länge 100 Bytes mit der Kennung 193 von Knoten 1 zu Knoten 2.

-102 : Empfangen einer Nachricht Wird analog interpretiert, außer daß der Feldinhalt von `data` als Attributwert für *src* benutzt wird.

Anhang B

Das VAMPIR-Spurformat

In diesem Anhang ist dokumentiert, wie das VAMPIR-Spurformat auf das EARL-Spurmodell abgebildet wird. Die bisher implementierte Dekoderklasse `Earl_SeqVmpTrace` ist in der Lage, VAMPIR-Spuren in der binären Formatvariante [3] zu lesen und zu dekodieren.

Eine binäre VAMPIR-Spurdatei enthält Konfigurations- und Ereignisrecords. Ereignisrecords stehen für konkrete Ereignisse, während Konfigurationsrecords allgemeine Eigenschaften des protokollierten Programmablaufs beschreiben und zur korrekten Dekodierung der Ereignisrecords notwendige Informationen enthalten.

In den folgenden Abschnitten sind die Bezeichner der Recordfelder in Schreibmaschinenschrift gesetzt.

B.1 Konfigurationsrecords

In diesem Abschnitt ist die Interpretation aller von EARL berücksichtigten Konfigurationsrecords erklärt. EARL geht davon aus, daß sich alle genannten Konfigurationsrecords vor dem ersten Ereignisrecord im Header der Spurdatei befinden.

NCPUS Dieser Record legt die Anzahl der benutzten Prozesse bzw. Knoten fest. Außerdem können die Prozesse durch dieses Record für heterogene Anwendungen in Gruppen eingeteilt werden. EARL bestimmt die Anzahl der Knoten als Summe der Gruppengrößen. Die Knotenanzahl errechnet sich aus der Summe der Felder des arraywertigen Recordfeldes `GroupSizes`. Es wird über die Feldelemente 0 bis `GroupCount` − 1 summiert.

DEFCPUSYMBOL Hier wird ein Knotensymbol `CPUName` für den Knoten `CPUNumber` festgelegt. Es kann von EARL mit der Spurobjektmethode '`info nodesym` *nodenumber*' abgefragt werden.

CLKPERIOD Mit dem in diesem Record angegebenen Faktor `ClockPeriod` werden die in der Regel ganzzahligen Zeitstempel der Ereignisrecords, d.h. die Inhalte des Ereignisrecordfeldes `TimeStamp`, multipiziert. Das Produkt wird von EARL als Sekundenwert interpretiert.

TIMEOFFSET Der `TimeOffset` wird zu allen aus dem obigen Produkt gebildeten Zeitstempeln addiert, d.h. er wird als Sekundenoffset interpretiert.

DEFTOKEN Dieser Record deklariert einen Bezeichner für eine Gruppe von Regionen und bildet ihn auf ein Token `Token` ab, anhand dessen die Gruppe in den folgenden Records referenziert wird. Es wird lediglich ein Name festgelegt, die Zuordnung der Regionen zu dieser Gruppe erfolgt später durch Konfigurationsrecords vom Typ DEFSYMBOL. Der deklarierte Gruppenname ist Element der mit der Methode `info groups` erfragbaren Liste.

Dieses Record ist notwendig zur korrekten Interpretation von Records des Typs DEFSYMBOL. Außerdem wird es zur Dekodierung von Ereignisrecords des Typs EXCHANGE benötigt.

DEFSYMBOL Dieser Record ordnet eine Region einer Gruppe zu und legt einen Regionenbezeichner fest. Die Gruppe wird durch das Token `Activity` bestimmt, der zugehörige Gruppenname ist vorher durch ein Record vom Typ DEFTOKEN vereinbart worden. `Symbol_Name` wird von EARL als Regionenname interpretiert. Außerdem wird der Region ein Token `Activity_Number` zugeordnet, durch welches in den Ereignisrecords Referenzen auf diese Region ausgedrückt werden.

B.2 Ereignisrecords

Allen Ereignisrecords gemeinsam sind die Felder `TimeStamp` und `CPU`. Das Feld `TimeStamp` bestimmt den Zeitstempel. Zu dem enthaltenen Wert wird jedoch, nachdem er, wie bereits erwähnt, mit dem Faktor `ClockPeriod` von CLKPERIOD multipliziert wurde, noch der Offset `TimeOffset` addiert. Der auf diese Weise errechnete Wert wird als Attributwert für *time* verwendet. Der Inhalt des Feldes `CPU` wird als Knotennummer interpretiert und direkt als Attributwert für *node* übernommen. Der Attributwert des *type*-Attributs richtet sich nach dem Typ des Ereignisrecords.

EXCHANGE Dieses Record beschreibt ein enter- oder exit-Ereignis. Das Feld `Call_Type` legt fest, um welchen EARL-Ereignistyp es sich handelt. Enthält `Call_Type` den Wert BINFMT_EXTYPE_UNKNOWN, so wird der Ereignistyp aus den Recordfeldern und dem Systemzustand unmittelbar vor Eintritt des entsprechenden Ereignisses inferiert. Das funktioniert jedoch nur bei Programmen, die keine rekursiven Regionenaufrufe enthalten.

Im Falle eines enter-Ereignisses bestimmt das Token `New_Activity` die Gruppe der betretenen Region, während die Region durch das Token

`New_Activity_Number` angegeben ist. Bei einem exit-Ereignis, wird die verlassene Region anhand des Systemzustands vor Eintritt des entsprechenden Ereignisses ermittelt.

Die Dekodierung der Token erfolgt anhand der Zuordnungen durch die Konfigurationrecords vom Typ DEFTOKEN und DEFSYMBOL. Enthält das Feld `New_Activity` bei dem Record eines exit-Ereignisses den Wert BINFMT_NOACT, so wird dies als Rückkehr auf Toplevel interpretiert.

SENDMSG Dieser Record steht für ein send-Ereignis. Das Attribut *dest* bezieht seinen Wert direkt aus dem Feld `TargetCPU`, das Attribut *tag* aus dem Feld `Type`, das Attribut *com* aus dem Feld `Communicator` und schließlich das Attribut *len* aus dem Recordfeld `Length`.

RECVMSG Dieser Record steht hingegen für ein recv-Ereignis. Die Attribute, die recv mit send gemeinsam besitzt, erhalten ihre Werte auf analoge Weise. Das *src*-Attribut bezieht seinen Wert aus dem Feld `SourceCPU`.

Das VAMPIR-Format unterstützt keine weiteren Ereignistypen.

Anhang C

Integration weiterer Spurformate

In diesem Teil des Anhangs wird die Integration weiterer Spurformate in die EARL-Umgebung beschrieben. Eine Erklärung der erforderlichen Modifikationen und Ergänzungen wird anhand des fiktiven Spurformats NEWFORM vorgenommen. Die Einbindung von NEWFORM vollzieht sich in zwei Schritten:

- Implementation einer neuen Dekoderklasse Earl_SeqNewformTrace,
- Änderung der Kommandoprozedur des `earl`-Kommandos, um die Erzeugung entsprechender Spurobjekte zu ermöglichen.

Die Schnittstelle der Dekoderklasse `Earl_SeqNewformTrace` ist in Abb. C.10 gezeigt. Bevor darauf im Detail eingegangen wird, werden noch die benutzten Datentypen zusammen mit den zur Implementation der Dekoderklasse wichtigen Elementfunktionen erläutert.

C.1 Benutzte Datentypen

Die Definitionen der folgenden Klassen befinden sich mit Ausnahme von `Earl_TclError` in gleichnamigen Dateien (`Klassenbezeichner.h`); die Definition `Earl_TclError` befindet sich in `Earl_TclException.h`. Sie können durch entsprechende `#include`-Anweisungen sichtbar gemacht werden.

C.1.1 `Earl_TclString`

Die Klasse `Earl_TclString` basiert auf den dynamischen Strings der Tcl-Bibliothek. Um unnötiges Kopieren zu vermeiden, werden intern Referenzenzähler benutzt, d.h. durch den Kopierkonstruktor oder Zuweisungsoperator entstandene Kopien teilen

sich eine Repräsentation, solange bis eine der Kopien modifiziert wird. Die modifizierte Kopie erhält vor der Modifikation eine eigene Repräsentation. Die für die Implementation der Dekoderklasse wichtigen Elementfunktionen sind in Abb. C.1 dargestellt.

```
class Earl_TclString {

public:

  Earl_TclString();
  Earl_TclString(const char *str);
  Earl_TclString(const Earl_TclString& ts);

  Earl_TclString& operator=(const Earl_TclString& ts);

  const char *value() const;

  Earl_TclString& append(const Earl_TclString& ts);
  Earl_TclString& append_element(const Earl_TclString& ts);

  friend bool operator==(const Earl_TclString& ts1,
                         const Earl_TclString& ts2);
  [...]
};
```

Abbildung C.1: Schnittstelle der Klasse `Earl_TclString`

Die Elementfunktion `value` liefert die Stringrepräsentation als C-String. `append` fügt den String `ts` direkt an, wobei zuvor für das modifizierte Objekt eine eigene Repräsentation angelegt wird. Ebenso bei `append_element`, nur daß hier die Zeichenkette als Listenelement angefügt wird. In den meisten Fällen wird dabei ein Leerzeichen zwischen die ursprüngliche und die anzuhängende Zeichenkette gesetzt. Enthält die anzuhängende Zeichenkette selbst Leerzeichen, so wird sie als Subliste interpretiert und in geschweifte Klammern gesetzt. Hierdurch wird der komfortable Aufbau von Tcl-Listen ermöglicht. Die Semantik von `append_element` entspricht der von `Tcl_DStringAppendElement` aus der Tcl-Bibliothek [24].

Der `operator==` liefert `true` genau dann, wenn die Repräsentationen gleichen Inhalt haben; es ist jedoch nicht erforderlich, daß sich die zu vergleichenden Strings dieselbe Repräsentation teilen.

Die Klasse `Earl_TclString` besitzt noch weitere zur Implemenation der Dekoderklasse nicht notwendige, aber u.U. nützliche Elementfunktionen und Operatoren. Diese sind im Quelltext in der Datei `Earl_TclString.h` dokumentiert.

C.1.2 Earl_Event und abgeleitete Ereignistypen

Die EARL-Ereignistypen werden durch die Klasse `Earl_EventEnter`, `Earl_EventExit`, `Earl_EventSend`, `Earl_EventRecv` und `Earl_EventUserdef` repräsentiert. Letztere steht für alle spur- bzw. formatspezifischen Typen.

Alle Klassen erben von der abstrakten Basisklasse `Earl_Event`. Diese bietet Funktionen zum Zugiff auf die Basisattribute mit Ausnahme des *type*-Attributs (Abb. C.2). *type* und die ereignistypspezifischen Attribute können mit Hilfe der Elementfunktionen der einzelnen Subklassen erfragt werden (Abb C.3, C.4, C.5, C.6 und C.7.

```
class Earl_Event {

public:

  long get_num() const;
  long get_node() const;
  double get_time() const;
  long get_enterptr() const;

  [...]
};
```

Abbildung C.2: Basisklasse der Ereignistypen

Die Konstruktoren der Repräsentationsklassen für die Ereignistypen erwarten den Zeitstempel als `double`-Wert in Sekunden, Knotenangaben für die Attribute *node*, *src* und *dest* als `long`. Ereignisnummern für die Attribute *num*, *enterptr* und *sendptr* werden ebenfalls als `long` übergeben. Regionen- und Gruppenbezeichner für die Attribute *region* und *group* müssen jedoch als `Earl_TclString` spezifiziert werden. Nachrichtenbezogene Attribute wie *tag*, *com* und *len* erhalten ihren Wert wieder als `long`, wobei die Nachrichtenlänge in Bytes verlangt wird.

Bei der Erzeugung von Ereignissen eines spur- bzw. formatspezifischen Typs muß der Typbezeichner explizit dem Konstruktor in Form eines `Earl_TclString` übergeben werden. Das *data*-Attribut erhält seinen Wert wegen der universellen Verwendbarkeit ebenfalls als `Earl_TclString`.

```
class Earl_EventEnter : public Earl_Event {

public:

  Earl_EventEnter(long num, long node, double time, long enterptr,
                  Earl_TclString region, Earl_TclString group);

  Earl_TclString get_type() const;
  Earl_TclString get_region() const;
  Earl_TclString get_group() const;

  [...]
};
```

Abbildung C.3: Repräsentation des enter-Ereignistyps

```
class Earl_EventExit : public Earl_Event {

public:

  Earl_EventExit(long num, long node, double time, long enterptr,
                 Earl_TclString region, Earl_TclString group);

  Earl_TclString get_type() const;
  Earl_TclString get_region() const;
  Earl_TclString get_group() const;

  [...]
};
```

Abbildung C.4: Repräsentation des exit-Ereignistyps

```
class Earl_EventSend : public Earl_Event {

public:

  Earl_EventSend(long num, long node, double time, long enterptr,
                 long dest, long tag, long com, long len);

  Earl_TclString get_type() const;
  long get_dest() const;
  long get_tag() const;
  long get_com() const;
  long get_len() const;

  [...]
};
```

Abbildung C.5: Repräsentation des send-Ereignistyps

```
class Earl_EventRecv : public Earl_Event {

public:

  Earl_EventRecv(long num, long node, double time, long enterptr,
                 long src, long tag, long com, long len, long sendptr);

  Earl_TclString get_type() const;
  long get_src() const;
  long get_tag() const;
  long get_com() const;
  long get_sendptr() const;

  [...]
};
```

Abbildung C.6: Repräsentation des recv-Ereignistyps

```
class Earl_EventUserdef : public Earl_Event {

public:

  Earl_EventUserdef(long num, long node, double time, long enterptr,
                    Earl_TclString type, Earl_TclString data);

  Earl_TclString get_type() const;
  Earl_TclString get_data() const;

  [...]
};
```

Abbildung C.7: Repräsentation von spur- bzw. formatspezifischen Ereignistypen

C.1.3 Earl_RndState

Die Klasse `Earl_RndState` (Abb. C.8) dient der Repräsentation von Systemzuständen. Der Index des Systemzustands läßt sich mit der Funktion `get_index` erfragen, wobei man für den Zustand S_i als Rückgabewert i erhält.

```
class Earl_RndState {

public:

  long get_index() const;

  long stack_size(long nodenumber) const;

  long get_stack_top_num(long nodenumber) const;
  Earl_EventEnter* get_stack_top(long nodenumber) const;
  Earl_EventEnter* get_stack_under_top(long nodenumber) const;

  long get_send_num(long dest, long src,
                    long tag, long com) const;

  [...]
};
```

Abbildung C.8: Repräsentation von Systemzuständen

Mit `stack_size` erhält man die Größe des Regionenkellers auf dem Knoten `nodenumber`, also genau die Länge der Liste, die man von EARL aus mit der

Spurobjektmethode '`stack` *nodenumber*' erhalten würde. Die nächste Funktion `get_stack_top` liefert die Ereignisnummer des obersten `enter`-Ereignisses auf dem Regionenkeller des Knotens `nodenumber`, also die Ereignisnummer des Eintritts in die zuletzt aktivierte Region. Ist der Regionenkeller leer, d.h. befindet sich das Programm auf diesem Knoten auf Toplevel, so wird -1 zurückgeliefert. Die nächsten beiden Funktion liefern das oberste bzw. zweitoberste Element des Regionenkellers auf dem spezifizierten Knoten als Zeiger auf das entsprechende Ereignisobjekt. Existiert das gewünschte Ereignis nicht, da der Keller z.B. leer ist, wird `NULL` zurückgeliefert.

Die letzte Funktion `get_send_num` liefert die Ereignisnummer des ältesten `send`-Ereignisses $e \in N_i$ mit $e.node =$ `src`, $e.dest =$ `dest`, $e.tag =$ `tag` und $e.com =$ `com`.

Es ist durchaus wahrscheinlich, daß nicht alle Elementfunktionen der Schnittstelle benötigt werden. Z.B. `get_stack_under_top` wird nur benutzt, um VAMPIR-Spuren älterer Ausprägung auf das Spurmodell abbilden zu können.

C.1.4 Earl_TclError

Diese Klasse dient der Absetzung von Fehlermeldungen. Ein Ausnahmeobjekt wird unter Angabe einer Fehlermeldung entweder als `Earl_TclString` oder als beliebige mit NULL abgeschlossene Folge von C-Strings erzeugt (Abb C.9).

```
class Earl_TclError : public Earl_TclException {

public:

  Earl_TclError(Earl_TclString msg);
  Earl_TclError(const char *str ...);

  [...]
};
```

Abbildung C.9: Schnittstelle der Ausnahmeklasse `Earl_TclError`

Die Ausnahme kann von einem Dekoderobjekt im Falle eines Fehlers geworfen werden und wird anschließend vom EARL-Interpreter abgefangen. Das entsprechende EARL-Kommando wird daraufhin mit einem Fehler beendet und die Fehlermeldung als Kommandoresultat an den Interpreter übergeben.

C.2 Implementation der Dekoderklasse

In diesem Abschnitt wird die Implementation der Dekoderklasse für das fiktive Spurformat NEWFORM beschrieben. Die Dekoderklasse muß die Schnittstelle in Abb.

C.10 implementieren.

```
class Earl_SeqNewformTrace : public Earl_SeqTrace {

public:

  Earl_SeqNewformTrace(char *filename);
  ~Earl_SeqNewformTrace();

  Earl_Event* read_event(const Earl_RndState& es);
  bool skip_event();

  long get_file_position();
  bool set_file_position(long file_pos);

  Earl_TclString get_file_name();
  Earl_TclString get_format();
  Earl_TclString get_node_sym(long nodenumber);
  Earl_TclString get_regions();
  Earl_TclString get_groups();
  Earl_TclString get_group(const char *groupname);
  long get_node_count();

  const vector<Earl_TclString>& get_userdef_event_types();
};
```

Abbildung C.10: Schnittstelle der Klasse `Earl_SeqNewformTrace`

Dem Konstruktor wird als einziges Argument der Spurdateiname `filename` übergeben. Seine Aufgabe besteht darin, die Spurdatei zu Öffnen und alle zur Dekodierung der Ereignisrecords notwendigen Informationen bereitzustellen. Bei den bisher implementierten Dekoderklassen wird der Header gelesen und es werden Tabellen zur Verwaltung der im Header enthaltenen Informationen angelegt. Nach der Erzeugung eines Dekoderobjekts durch den Konstruktor erwartet EARL, daß der Dateilesezeiger unmittelbar vor dem ersten Ereignisrecord steht.

Sollte beim Öffnen der Spurdatei oder auch später ein Fehler auftreten, so sollte eine Ausnahme des Typs `Earl_TclError` mit einer entsprechenden Fehlermeldung geworfen werden.

Der Destruktor sollte die Spurdatei schließen und alle benutzten Ressourcen wieder freigeben.

Die Elementfunktion `read_event` soll das der aktuellen Position des Dateilesezeigers unmittelbar folgende Ereignisrecord lesen und daraus ein entsprechendes Ereignisobjekt, d.h. eine Instanz eines Subtypen von `Earl_Event`, erzeugen und einen Zeiger

darauf zurückliefern. Als Argument wird eine Referenz auf den Systemzustand als Instanz von `Earl_RndState` mitgeliefert. Soll das Ereignis e_i gelesen werden, so wird der Zustand S_{i-1} übergeben. Auf diese Weise sollten sich alle erforderlichen Attribute mit den Elementfunktionen von `Earl_RndState` berechnen lassen. Insbesondere sollte sich *enterptr* mit `get_stack_top_num` und *sendptr* mit `get_send_num` bestimmen lassen. Läßt sich, weil das Dateiende erreicht ist, ab der Position des Dateilesezeigers kein Ereignisrecord mehr lesen, so muß `NULL` zurückgeliefert werden. Nach erfolgreichem Abschluß der Leseoperation sollte der Lesezeiger vor dem nächsten Ereignisrecord stehen. Falls die Operation wegen Erreichen des Dateiendes nicht erfolgreich beendet werden kann, erwartet EARL, daß der Zustand des Dateilesezeigers, der vor Aufruf der Funktion bestanden hat, restauriert wird.

`skip_event` liest einfach das nächste Ereignisrecord, ohne jedoch ein Ereignisobjekt zu erzeugen. Nach erfolgreichem Abschluß der Leseoperation sollte der Lesezeiger vor dem nächsten Ereignisrecord stehen. Wird das Dateiende erreicht, muß `false` und sonst `true` zurückgeliefert werden; außerdem muß wie bei `read_event` der Zustand des Zeigers restauriert werden.

Die nächsten beiden Operationen `get_file_position` und `set_file_position` gestatten Zugriff auf den Dateilesezeiger. Diese Funktionen müssen in dem Sinne konsistent sein, daß eine mit `get_file_position` gelesene Position durch `set_file_position` später wieder erreicht werden kann. Ob die erfragten Werte dabei dem tatsächlichen Wert des Dateilesezeigers entsprechen, ist irrelevant. Kann `set_file_position` wegen der Angabe einer illegalen Position nicht korrekt ausgeführt werden, so muß, nachdem auch hier der alte Zustand wiederhergestellt worden ist, `false` zurückgegeben werden.

Die nächsten Funktionen implementieren jeweils eine `info`-Spurobjektmethode und müssen daher die in Kapitel 6 definierte Funktionalität bereitstellen. Mit Ausnahme von `get_node_count` wird als Rückgabewert eine Tcl-Liste in Form eines `Earl_TclString` erwartet; die Liste kann mit `Earl_TclString::append_element` aufgebaut werden. Die Zuordnung der Elementfunktionen zu den entsprechenden `info`-Methoden ist in Tabelle C.1 dargestellt.

Tabelle C.1: Implementation von `info`-Methoden

`get_file_name()`	`$to info filename`
`get_format()`	`$to info format`
`get_node_sym(long nodenumber)`	`$to info nodesym` *nodenumber*
`get_regions()`	`$to info region`
`get_groups()`	`$to info groups`
`get_group(const char *groupname)`	`$to info group` *groupname*
`get_node_count()`	`$to info nodecount`

Falls bei `get_node_sym` kein zulässiger Knoten bzw. bei `get_group` keine definierte Gruppe angegeben wird, so muß eine Ausnahme des Typs `Earl_TclError` mit der

Fehlermeldung "undefined nodenumber" bzw. "undefined groupname" geworfen werden.

Die letzte Funktion get_userdef_event_types soll als Rückgabewert ein Referenz auf einen STL-Vektor liefern, der die Bezeichner aller spur- bzw. formatspezifischen Typen enthält. D.h. existieren die spur- bzw. formatspezifischen Typen $type_0, type_1, \ldots, type_{n-1}$, so sollte der Vektor in der Zelle i den Bezeichner des Typs $type_i$ enthalten. Die Vektorgröße (size) muß gleich n sein. Diese Funktion wird für die restlichen info-Methoden benötigt. Es ist gedacht, daß der Konstruktor der Dekoderklasse diesen Vektor als privates Datenelement anlegt und die Funktion lediglich eine Referenz auf das Datenlement zurückliefert.

C.3 Modifikation der earl-Kommandoprozedur

Es müssen die Kommandoprozeduren der Subkommandos earl open und earl help in der Datei Earl_Cmd.C geändert werden. Außerdem ist die Definition von Earl_SeqNewformTrace durch eine entsprechende #include-Anweisung dort sichtbar zu machen.

Die Kommandoprozedur des Subkommandos earl open namens EarlOpenCmd muß um eine zusäztliche Abfrage der Formatoption ergänzt werden. Die Erweiterung ist in Abb. C.11 kursiv dargestellt.

```
int
EarlOpenCmd([...])
{
  [...]

  if((strcmp(format, "vampir")) == 0)
    trPtr = new Earl_SeqVmpTrace(filename);

  else if ((strcmp(format, "alog")) == 0)
    trPtr = new Earl_SeqAlogTrace(filename);

  else if ((strcmp(format, "newform")) == 0)
    trPtr = new Earl_SeqNewformTrace(filename);

  [...]
}
```

Abbildung C.11: Ergänzungen an der Kommandoprozedur von earl open

Die Kommandoprozedur des Subkommandos earl help ist an der Stelle zu erweitern, wo die Ausgabe des Kommandos earl help formats implementiert ist. Dieses

Kommando gibt die Liste aller unterstützten Spurformate aus. Der Bezeichner des neu integrierten Spurformats ist einfach an die Bezeichner der bisher unterstützten Formate anzuhängen (Abb. C.12).

```
int
EarlHelpCmd([...])
{
  [...]

  if((c == 'f') && (strcmp(argv[2], "formats")) == 0) {
    Tcl_SetResult(interp, "alog vampir newform", TCL_STATIC);
    return TCL_OK;
  }

  [...]
}
```

Abbildung C.12: Ergänzungen an der Kommandoprozedur von `earl help`

Bei der Integration mehrerer Spurformate sind die gezeigten Erweiterungen mehrfach auszuführen. Die in den Abb. C.11 und C.12 gezeigten Quelltextstellen aus der Datei `Earl_Cmd.C` sind dort durch entsprechende Kommentare gekennzeichnet.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] BLT Toolkit (Version 2.3). ftp://ftp.tcltk.com/pub/blt/BLT2.3.tar.gz, Online: 26. März 1998. – Quelltextdateien und Dokumentation. (siehe auch http://www.tclconsortium.org/resources/download.html und http://www.tcltk.com/blt.)

[2] ALLEN, W. ; ROBERTS, W.: *Methoden der Statistik.* Rowohlt, 1975

[3] ARNOLD, A.: *VAMPIR Trace Format Specification.* Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathemetik, November 1997. – Interne Dokumentation

[4] ARNOLD, A. ; DETERT, U. ; NAGEL, W.E.: Performance Optimization of Parallel Programs: Tracing, Zooming, Understanding. In: WINGET, R. (Hrsg.) ; WINGET, K. (Hrsg.): *Proc. of Cray User Group Meeting.* Denver, CO, März 1995, S. 252–258

[5] BATES, P.: *Debugging Programs in a Distributed System Environment*, University of Massachusetts, Ph.D. Thesis, Februar 1986

[6] BATES, P. ; WILEDEN, J. C.: High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach. In: *The Journal of Systems and Software* 3 (1983), S. 255–264

[7] BEGUELIN, A. ; DONGARRA, J. ; GEIST, G. A. ; MANCHEK, R. ; SUNDERAM, V.: A user's guide to PVM: Parallel virtual machine / Oak Ridge National Laboratory. 1991 (ORNL/TM-11826). – Technical Report

[8] BOMANS, L. ; ROOSE, D. ; HEMPEL, R.: The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. In: *Parallel Computing* 15 (1990), S. 119–132

[9] FOSTER, I.: *Designing and Building Parallel Programs.* Addison-Wesley, 1995

[10] GALAROWICZ, Jim ; MOHR, Bernd: Analyzing Message Passing Programs on the Cray T3E with PAT and VAMPIR / Forschungszentrum Jülich. Mai 1998 (FZJ-ZAM-IB-9809). – Interner Bericht. Wird erscheinen in: Fourth European CRAY-SGI MPP Workshop, Garching/München, 10. und 11. September, 1998.

[11] GEIST, G. A. ; HEATH, M. T. ; PEYTON, B. W. ; WORLEY, P. H.: PICL: A portable instrumented communications library, C reference manual / Oak Ridge National Laboratory. Juli 1990 (ORNL/TM-11130). – Technical Report

[12] GERNDT, M. ; PANTANO, M. ; MOHR, B. ; WOLF, F.: Automatic Performance Analysis for CRAY T3E / Forschungszentrum Jülich. Mai 1998 (FZJ-ZAM-IB-9808). – Interner Bericht. Erschienen in: Proc. of the 7th Workshop on Compilers for Parallel Computers (CPC'98). Linköping, Schweden, 29. Juni - 1. Juli 1998

[13] GROPP, W. ; LUSK, E. ; SKJELLUM, A.: *USING MPI - Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994

[14] HEATH, M. ; ETHERIDGE, J.: Visualizing the performance of parallel programs. In: *IEEE Software* 8 (1991), September, Nr. 5, S. 29–39

[15] HERRARTE, V. ; LUSK, E.: Studying Parallel Program Behavior with Upshot / Mathematics and Computer Science Division, Argonne National Laboratory. August 1991 (ANL-91/15). – Technical Report

[16] HOLLINGSWORTH, J.K. ; STEELE, M.: Grindstone: A Test Suite for Parallel Performance Tools / University of Maryland. Oktober 1996 (CS-TR-3703). – Computer Science Technical Report

[17] Intel Corporation: *Paragon C System Calls*. Oktober 1993. – Reference Manual

[18] JAIN, R. ; CHLAMTAC, I.: The P2 Algorithm for Dynamic Calculation of Quantiles and Histograms Without Storing Observations. In: *Communcations of the ACM* 28 (1985), Oktober, Nr. 10

[19] KARRELS, E.: Nupshot - Parallel Program Visualization Tool. In: *MPICH-A Portable Implementation of MPI*. Argonne Nationbal Laboratory, Mathematics and Computer Science Division : ftp://ftp.mcs.anl.gov/pub/mpi/mpich.tar.gz, Online: 8. Juni 1998. – Quelltextdateien. (siehe auch http://www.mcs.anl.gov/mpi/mpich/index.html

[20] KERNIGHAN, B. W. ; RITCHI, D. M.: *Programmieren in C*. Carl Hanser Verlag, 1990

[21] KOELBEL, C. H. ; LOVEMAN, D. B. ; SCHREIBER, R. S. ; STEELE, G. L. J. ; ZOSEL, M. E.: *The High Performance Fortran Handbook*. MIT Press, 1993

[22] MESSAGE PASSING INTERFACE FORUM: MPI: Extensions to the Message-Passing Interface. http://www.mpi-forum.org/docs/mpi-20.ps.Z, Juli 1997. – Dokumentation der MPI-Standards 1.2 und 2.0. Online: 22. Juni 1998. (siehe auch http://www.mpi-forum.org/docs/docs.html)

[23] MOHR, B.: Standardization of Event Traces Considered Harmful or Is an Implementation of Object-Independent Event Trace Monitoring and Analysis Systems Possible? In: DONGARRA, J.J. (Hrsg.) ; TOURANCHEAU, B. (Hrsg.): *Proc. CNRS-NSF Workshop on Environments and Tools For Parallel Scientific Computing, Advances in Parallel Computing* Bd. 6. Elsevier, September 1992, S. 103–124

[24] OUSTERHOUT, J. K.: *Tcl and the Tk Toolkit*. Addison-Wesley, 1994

[25] REED, D.A. ; OLSON, R.D. ; AYDT, R.A. ; MADHYASTA, T.M. ; BIRKETT, T. ; JENSEN, D.W. ; NAZIEF, A.A. ; TOTTY, B.K.: Scalable Performance Environments for Parallel Systems. In: *Proc. 6th Distributed Memory Computing Conference*, IEEE Computer Society Press, 1991, S. 562–569

[26] STROUSTRUP, B.: *The C++ Programming Language*. 3. Auflage. Addison-Wesley, 1997

[27] WELCH, B. B.: *Practical Programming in Tcl and Tk*. Prentice Hall, 1995

[28] WELCH, B. B.: *Practical Programming in Tcl and Tk*. 2. Auflage. Prentice Hall, 1997

[29] WOLF, F. ; MOHR, B.: EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs / Forschungszentrum Jülich. April 1998 (FZJ-ZAM-IB-9803). – Interner Bericht

Danksagung

Die vorliegende Arbeit wurde als Diplomarbeit in Informatik am Lehrstuhl für Technische Informatik und Computerwissenschaften der Rheinisch-Westfälischen Technischen Hochschule (RWTH) Aachen erstellt.

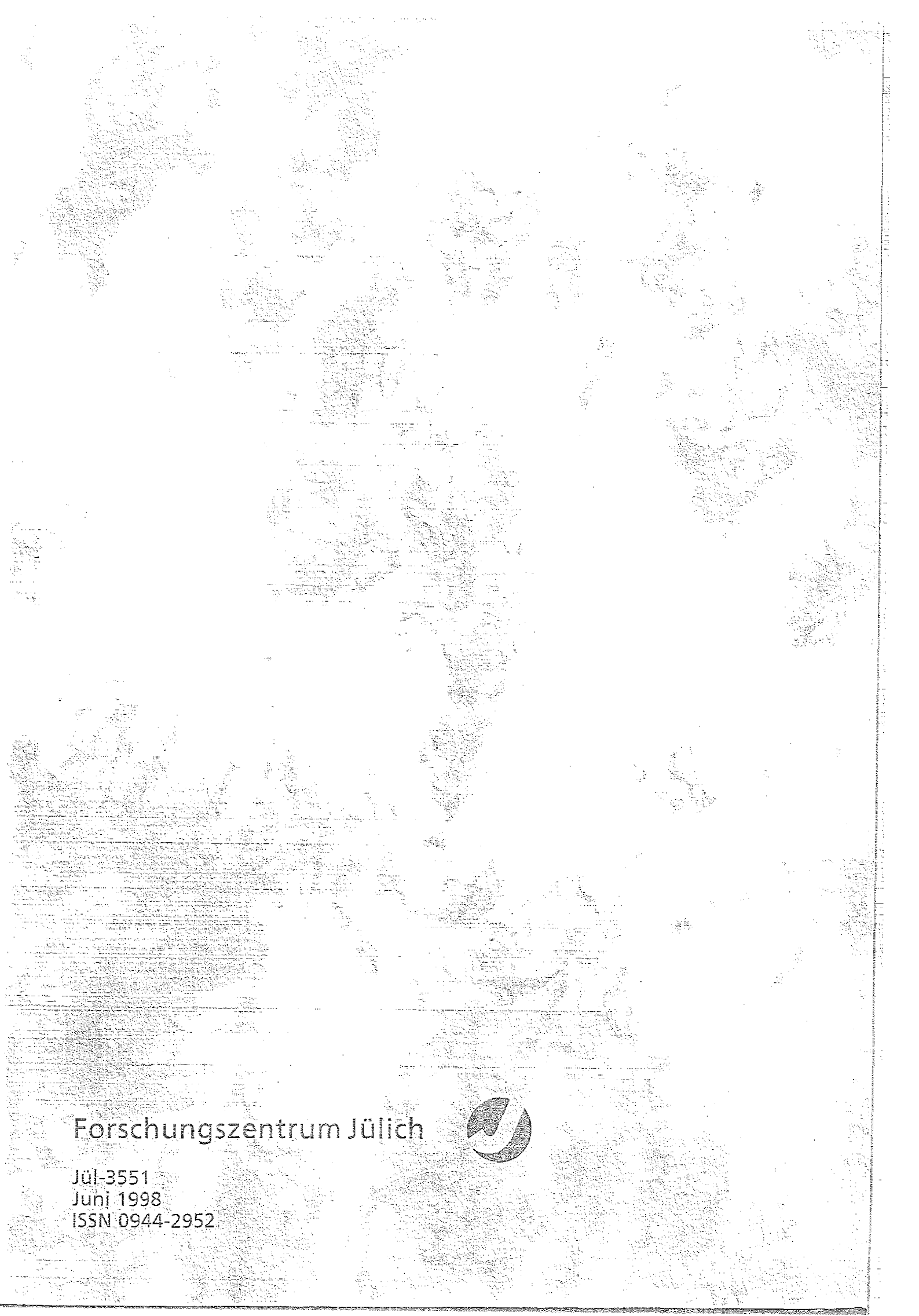
Dem Lehrstuhlinhaber Prof. Dr. Friedel Hoßfeld danke ich für die Möglichkeit, diese Arbeit am Zentralinstitut für Angewandte Mathematik (ZAM) des Forschungszentrums Jülich (FZJ) anfertigen zu können, und für das beständige Interesse, mit welchem er den Fortgang meiner Arbeit begleitet hat. Prof. Dr. Klaus Indermark danke ich für die Übernahme des Zweitgutachtens.

Besonderer Dank gilt Dr. Bernd Mohr für die ausgezeichnete Betreuung, für seine stete Ansprechbarkeit, für die vielen interessanten Diskussionen sowie für die gute Zusammenarbeit beim Verfassen von [29]. Bei Dr. Michael Gerndt, Dr. Bernd Mohr und Mario Pantano, Ph.D., bedanke ich mich für die Möglichkeit, an [12] mitwirken zu können.

Dank gilt auch Dr. Alfred Arnold für die geduldige Beantwortung meiner Fragen zum VAMPIR-Format sowie für die Berücksichtigung meiner Erweiterungswünsche zur *binlib*-Bibliothek. Astrid Göke und Jörg Henrichs gaben mir wertvolle Anregungen, die wesentlich zum Gelingen meines Diplomvortrags beigetragen haben. Susan Poelchau beriet mich in Fragen der englischen Sprache. Marianne Frerichs sowie Franz Josef Schoenebeck halfen mir stets bei Problemen mit dem Betriebssystem, und Wolfgang Frings wußte Rat bei Fragen zur Texverarbeitung.

Vor allem möchte ich mich bei meiner Freundin Ilka und meiner Familie bedanken, die mich mit viel Verständis und Geduld durch die Zeit meiner Diplomarbeit begleitet haben.

Außerdem bedanke ich mich bei allen anderen, die mich beim Anfertigen meiner Arbeit unterstützt haben.



Forschungszentrum Jülich

Jül-3551
Juni 1998
ISSN 0944-2952