Ein symbolischer X Windows Debugger für Modula-2*

Diplomarbeit am
Institut für Programmstrukturen und Datenorganisation

Stefan U. Hänßgen

Betreuer:
Prof. Dr. Walter F. Tichy
Dipl.-Inform. Michael Philippsen

Fakultät für Informatik
Universität Karlsruhe

30. Dezember 1992

Hiermit versichere ich, daß ich diese Diplomarbeit selbständig und ohne unzulässige Hilfsmittel angefertigt habe. Die verwendeten Quellen sind vollständig im Literaturverzeichnis angegeben.

Karlsruhe, den 30. Dezember 1992

(Stefan U. Hänßgen)

Inhaltsübersicht

Thema dieser Diplomarbeit ist der Entwurf und die Implementierung eines symbolischen Debuggers für Modula-2* mit einer X-Windows Benutzeroberfläche.

Es werden mehrere innovative Techniken zur Visualisierung von Daten und Programmablauf eingeführt.

Nach einer Einleitung und Motivation des Projekts werden verschiedene Aspekte des Entwurfs diskutiert, wobei zwischen idealen Anforderungen und praktischer Realisierbarkeit unterschieden wird. Es schließt sich eine Erläuterung der konkret erfolgten Implementierung und ein Vergleich mit verwandten Arbeiten an. Am Ende wird ein Ausblick auf weiterführende Pläne geboten.

Die Anhänge geben eine detailliertere Einsicht in die während der Diplomarbeit erstellten Programme, Hilfsmittel, Benutzerschnittstellen und deren Semantik. Außerdem werden ein aus dieser Diplomarbeit entstandener Konferenzbeitrag und die englische Benutzerdokumentation zum Debugger aufgeführt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation: Paralleles Debuggen

Auf sequentiellen Maschinen gehört ein symbolischer Debugger, eventuell sogar mit grafischer Benutzeroberfläche, schon seit längerer Zeit zum Lieferumfang eines Systems. Kein Anwendungsprogrammierer steigt noch auf Assemblerebene hinab, um Fehler zu suchen[1].

Im Bereich der Parallelrechner war hingegen bis vor wenigen Jahren die Unterstützung für eine Fehlersuche allenfalls rundimentär. Erst seit recht kurzer Zeit sind Produkte auf dem Markt, die auch einem nicht vollständig mit allen Ebenen der Maschine vertrauten Benutzer eine halbwegs effiziente Analyse von Fehlern ermöglichen [Masp91b, Thin91].

Dabei ist gerade bei parallelen Programmen das Debuggen im Vergleich zum sequentiellen Fall in vielerlei Hinsicht erschwert:

- Der parallele Ablauf des Programms auf mehreren tausend Prozessoren ist kaum noch zu überblicken.
- Die Interaktion der parallelen Aktivitäten birgt viele neue Möglichkeiten von Fehlern in sich, wie z.B. Probleme mit dem Zeitverhalten der Prozesse untereinander (Races) oder Verklemmungen in der Kommunikation (Deadlocks).
- Der Benutzer kann sich in der Datenflut, die ein Parallrechner verarbeitet, nicht mehr ohne weitergehende Hilfsmittel zurechtfinden.
- Die Semantik von scheinbar einfachen Hilfsmitteln wie Breakpoints oder Einzelschrittablauf muß im Parallelen neu überdacht werden.
- Den Benutzer interessieren die Vorgänge auf unterer Ebene nicht, er will sich auf dem Niveau seiner Programmiersprache bewegen. Bei bisherigen parallelen Programmiersprachen ist er hingegen gezwungen, auf Besonderheiten der verwendeten Maschine einzugehen.

Daraus ergibt sich die Forderung nach einfach zu bedienenden Debuggern für parallele Programme, die dem Benutzer auf dem für ihn interessanten Abstraktionsniveau bei der Fehlersuche helfen.

Außerdem ist es vorteilhaft, wenn der Debugger auf die jeweilige Sprache zugeschnitten ist und ihre Konstrukte direkt unterstützt, damit man in Konzepten der Programmiersprache arbeiten kann.

[1] Natürlich gibt es immer noch Anwendungen für solches Low-Level Debuggen, wie etwa betriebssystemnahe Programmierung, aber dem normalen Programmierer ist mit solchen Werkzeugen nicht geholfen.

Existierende Debugger für parallele Programme haben zudem dasselbe Problem wie fast alle bisherigen parallelen Programmiersprachen, auf die sie zugeschnitten sind: Der Benutzer ist zu sehr damit beschäftigt, maschinenspezifische Fehler – etwa Kommunikation, Verklemmungen etc. – zu beheben, während Probleme in seinem Algorithmus selbst zu geringe Aufmerksamkeit erhalten. Abstraktion von Vorgängen auf unterster Ebene befreit den Benutzer von dieser Last.

1.2 Zielsetzung

Ziel dieser Arbeit ist die Erstellung eines Debuggers für Modula-2*, der dem Benutzer eine effiziente Fehlersuche in seinem Programm ermöglicht, ohne ihn mit maschinenabhängigen Details zu belästigen. Die Parallelität des Programmablaufs und der Daten soll einfach und intuitiv verständlich dargestellt werden.

Dazu ist es nötig, mit verschiedenen Visualisierungen zu arbeiten, da man geeignete grafische Repräsentationen eines Sachverhalts viel leichter überblicken kann.

Außerdem soll der Debugger die Besonderheiten von Modula-2* direkt unterstützen. Zum Beispiel sollte er die `FORALL`s, die eine zentrale Rolle spielen, auch geeignet darstellen und dem Benutzer näherbringen können.

Von maschinenspezifischen Besonderheiten ist zu abstrahieren, wobei für den interessierten Programmierer – etwa den Übersetzerbauer, der seine Programme austesten möchte – optional eine Anzeige auch der Vorgänge unterhalb der Modula-2*-Ebene zu bieten ist.

1.3 Modula-2*

Modula-2* [TiHe90] ist eine maschinenunabhängige parallele Erweitung von Modula-2 [Chin91] mit den folgenden Eigenschaften[2]:

- Eine beliebige Zahl von Prozessen arbeitet auf einem gemeinsamen Adreßraum, was allerdings nicht unbedingt gleiche Zugriffszeiten bedeutet.
- Synchrone und asynchrone Berechnungen in beliebiger Schachtelung können maschinenunabhängig formuliert werden.
- Alle Abstraktionsmittel von Modula-2 sind auch für paralleles Programmieren verfügbar.

Die Sprache wurde dazu um zwei Konstrukte erweitert:

- Die `FORALL`-Anweisung erzeugt parallele Ausführungseinheiten. Es gibt eine synchrone und eine asynchrone Variante. Die Syntax in EBNF ist:

```
ForallStatement = FORALL ident ":"SimpleType in {PARALLEL | SYNC}
                  [VarDecl + BEGIN] StatementSequence END
```

- Allokatoren geben dem Übersetzer Hinweise für die Verteilung von Feldern auf die realen PEs. Sie werden im weiteren nicht beachtet, da sie nur untergeordnete Bedeutung für den Debugger haben.

[2] Diese Kurzeinführung orientiert sich an [HLP92].

Das neuartige Konzept einer völlig von der unterliegenden Architektur unabhängigen Sprache für Parallelrechner erlaubt einen anderen Schwerpunkt bei der Fehlersuche. Viele Probleme, um die sich herkömmliche Parallelrechner-Debugger kümmern müssen (z.B. Verklemmungen, zeitbedingte Indeterminismen und Kommunikationsmuster auf unterer Ebene), sind für den Modula-2* Programmierer nicht sichtbar, sondern werden vom Übersetzer gelöst. Dadurch kann der Debugger Abstraktionen auf hoher Ebene untersuchen, anstatt maschinenspezifische Probleme zu betonen.

1.4 Voraussetzung und Hilfsmittel

Der praktische Teil der Diplomarbeit wurde auf dem Netzwerk von Sparc Suns des Instituts für Programmstrukturen und Datenorganisation ausgeführt. Als Parallelrechner stand eine MasPar MP-1 zur Verfügung. Zur Implementierung wurden die C-Übersetzer der Hersteller verwendet [Masp91a, Sun90b].

Der Übersetzer für Modula-2* wurde im Rahmen der Diplomarbeit auf die Unterstützung des Debuggers durch Einfügen von Zusatzinformationen eingestellt.

Den Grundstein der Debugger-Oberfläche bildet `xdbx` [Cheu89], der von Po Cheung an der University of Texas entwickelt und für die Allgemeinheit freigegeben wurde. Er setzt auf `dbx` [Sun90c] auf, der auf SUNs und DECstations zur Verfügung steht und die Basis des Debuggers für einen einzelnen Rechner und den Netzwerkübersetzer bildet. Im parallelen Laufzeitsystem wurde `gdb` [StPe92] von der Free Software Foundation verwendet.

Die Visualisierung mehrdimensionaler Daten und die Behandlung mehrerer Quellcode-Fenster stützen sich auf meine Studienarbeit [Haen91]. Implementiert wurde die grafische Oberfläche unter X Windows [AMS91] mit dem Athena Widgets Set [Pete91].

Im Rahmen der Programmierung wurden verschiedene Tools des Unix Workbenchs [Sun90a] sowie Emacs [Stal90] verwendet. Die Ausarbeitung wurde mittels LaTeX [Lamp86, Knut84] erstellt und auf einem HP LaserJet gedruckt.

1.5 Ergebnisse

Im Rahmen der Diplomarbeit wurden mehrere neuartige Konzepte zur Visualisierung und Abstraktion entworfen und im MSDB (**M**odula Two **S**tar **D**e**B**ugger) implementiert:

- Parallele Abläufe werden auf Sprachstrukturen-Ebene nach ihrer dynamischen Vorgeschichte gruppiert und als Baum dargestellt.
- Mehrdimensionale Daten werden, nach verschiedenen Kriterien geordnet, visualisiert.
- Profiling auf Konstruktebene ermittelt die am häufigsten durchlaufenen Programmteile.
- Die Aktivierungen der virtuellen PEs werden visualisiert.
- Über ein Netz von Workstations verteilte Daten und Aktivitäten werden zusammengefaßt und dem Benutzer zentral präsentiert.
- Eine durchgehende grafische Benutzeroberfläche vereinfacht die Bedienung wesentlich.
- Der Debugger wurde für die Fehlersuche auf einer einzelnen Workstation und einem Netzwerk von Workstations implementiert. Dabei wurden auch Konzepte für die Portierung auf eine MasPar und weitere Rechner entwickelt.

Der Debugger abstrahiert dabei von allen maschinenspezifischen Details wie etwa der Kommunikation. Implementiert wurden Versionen für einen einzelnen Prozessor und ein Netzwerk von Workstations. Umsetzungen auf eine MasPar und ein Transputernetz sind geplant.

Ein Konferenzbeitrag zum „3. ACM/ONR Workshop on Parallel and Distributed Debugging“ im Mai 1993 wurde aus Material der Diplomarbeit erstellt und eingereicht.

Kapitel 2

Entwurf

Für den Entwurf eines parallelen Debuggers stellen sich viele Fragen, die im sequentiellen Fall überhaupt nicht vorkommen. Zudem stellt Modula-2* einen Sonderfall unter den heute gebräuchlichen parallelen Sprachen dar, was wiederum Auswirkungen auf die Aufgabenschwerpunkte des Debuggers hat. Schließlich gilt es noch, die Benutzerschnittstelle so zu gestalten, daß die Fehlersuche möglichst effizient vonstatten geht.

Im folgenden werden die Hauptprobleme beschrieben und verschiedene Lösungsansätze diskutiert. Dabei läuft die Reihenfolge von allgemeinen Abschnitten bis hin zu spezifischen Aspekten des zugrundeliegenden Modula-2*-Übersetzers `msc`. Die konkrete Realisierung der einzelnen Punkte wird in Kapitel 3 behandelt, und in Kapitel 4 werden existierende Arbeiten in dieser Richtung analysiert und vergleichend bewertet.

2.1 Grundlegende Entscheidungen

Zwei Entwurfsentscheidungen bestimmen die weiteren Schritte grundlegend: Soll der Debugger interaktiv sein, und auf welcher Sprachebene setzt er an? Außerdem gibt es Kriterien für wirksame Visualisierungen, die beachtet werden sollten.

2.1.1 Interaktivität

Viele parallele Debugger sind post-mortem-Debugger, d.h. sie werden erst nach Ende des Programms eingesetzt, um anhand von Daten, die während des Programmablaufs aufgezeichnet wurden, das Programmverhalten zu analysieren.

Diese Vorgehensweise hat Vorteile, wenn man den Debugger möglichst ohne große Auswirkungen auf das laufende Programm betreiben möchte. Gerade im parallelen Bereich sind die sog. „Heisenbugs“ bekannt. Dies sind Fehler, die gerade durch die Benutzung eines Debuggers und seine Auswirkungen auf die Last und damit das Zeitverhalten etc. indeterministisch auftreten oder verschwinden.

Bei der Fehlersuche in Modula-2* spielen aber solche Probleme mit dem Zeitverhalten keine Rolle: Entweder verlangt der Programmierer eine genaue Synchronisation der Aktivitäten und verwendet ein `FORALL IN SYNC`, oder Indeterminismen in der Ausführungsreihenfolge sind unproblematisch, so daß ein `FORALL IN PARALLEL` benutzt werden kann.[1]

[1] Die Wahl der asynchronen Parallelität an Stellen, die eigentlich eine synchrone Abarbeitung erfordern, ist ein konzeptueller Fehler, der leicht durch die Ersetzung des asynchronen durch ein synchrones `FORALL` nachgeprüft werden kann. Welche konkreten Indeterminismen an der falschen Abarbeitung schuld waren, ist auf der Ebene von Modula-2* nicht sichtbar und daher auch nicht Gegenstand des Debuggers.

Es sollte dem Benutzer also eine interaktive Fehlersuche ermöglicht werden. Die Vorteile liegen auf der Hand: Man kann sofort, wenn ein Fehler auftritt, die relevanten Daten analysieren. Mehr noch, man kann schon vor Auftreten des Fehlers den Debugger aktivieren und sich zeilenweise an die entsprechende Stelle des Problems herantasten und dabei beobachten, welche Vorgänge ablaufen. Außerdem ist es möglich, den Programmablauf interaktiv zu beeinflussen, etwa durch Umsetzen einer Variablen. Das ist bei einem post-mortem-Debugger natürlich nicht möglich oder zu langwierig, da dazu der gesamte Programmablauf wiederholt werden muß.

2.1.2 Hochsprachlichkeit

Ein Schwerpunkt der meisten heute üblichen Debugger für parallele Sprachen liegt auf der Überwachung und Auswertung der Kommunikation aller parallelen Aktivitäten. Den Benutzer interessieren Probleme wie das relative Zeitverhalten der Prozesse, das bei nicht genau kontrollierten Zugriffen zu indeterministischen Fehlern führen kann. Ebenso von Bedeutung ist für ihn der Ablauf der Kommunikation zwischen den Aktivitäten, die sich z.B. verklemmen können.

Alle diese Probleme gibt es per definitionem in Modula-2* nicht, wie in der Einführung bereits erwähnt wurde. Der Programmierer bewegt sich auf einem Abstraktionsniveau, das Begriffe wie Races (zeitbedingte Indeterminismen) oder Deadlocks (Verklemmungen) zumindest auf Ebene der Prozeßkommunikation nicht kennt – die in der Sprachdefinition vorgesehenen Befehle `SEND` und `WAIT` werden in der Praxis nicht verwendet, da sie unzureichend definiert und unhandlich sind. Um die Verteilung der Prozesse und ihre Kommunikation kümmert sich der Übersetzer. Er verdeckt alle maschinenabhängigen Details in diesem Bereich, der Benutzer sieht lediglich Feldzugriffe.

Dementsprechend ist auch die Suche nach solchen Fehlern keine Anforderung an einen Debugger für Modula-2*. Maschinennahe Probleme dieser Art sind nur für den Übersetzerbauer interessant, und dieser kann sich auf die mit jedem Parallelrechner mitgelieferten Werkzeuge zur Low-Level-Analyse stützen. Diese allgemein anzubieten, wäre weder möglich, da sich die verschiedenen Rechner zu sehr unterscheiden, noch erstrebenswert, da die Sprache von diesem Niveau bewußt abstrahiert.

2.1.3 Kriterien für eine Visualisierung

In [Mill92] werden Punkte aufgeführt, auf die man beim Entwurf einer Visualisierung achten sollte, um gute Ergebnisse zu erzielen. Genaueres findet sich im Anhang A, wo auch eine Bewertung der konkret implementierten Visualisierungen anhand dieser Punkte stattfindet. Hier werden nur die wichtigsten Kriterien kurz erläutert:

- Der Benutzer muß auch ohne Vorkenntnisse über den dargestellten Sachverhalt aufschlußreiche Bilder erhalten können.
- Die Visualisierung sollte auf dem Niveau und in Begriffen der Programmiersprache erfolgen.
- Es ist auf Skalierbarkeit zu achten.
- Visualisierungen sollten sich interaktiv untersuchen lassen.
- Von interessanten, aber unwichtigen Details ist zu abstrahieren.
- Die Visualisierung sollte einfach zu handhaben sein.

2.2 Repräsentation der parallelen Ausführung

In diesem Abschnitt werden interessante Gesichtspunkte für die parallele Ausführung von Programmen und verwandte Probleme aufgezeigt.

2.2.1 Gruppierung von Ablaufeinheiten

Nach Ansprechen der Grundproblematik werden die in Modula-2* verwendete Parallelität vorgestellt und Gruppierungsmöglichkeiten aufgezeigt.

Problemstellung

Bei einem parallelen Programm können Millionen von (virtuellen) Prozessen gleichzeitig aktiv sein. Jeden einzelnen davon darzustellen und zu untersuchen, ist nicht sinnvoll, da die Informationsflut den Benutzer überfordern würde. Es gilt also, dem Programmierer verschiedene Abstraktionsebenen zu bieten, auf denen er die Prozesse zu semantisch sinnvollen Gruppen zusammenschließen, aber auch bei Bedarf einen speziellen Einzelprozeß untersuchen kann. Ein ähnliches Problem stellt sich übrigens ebenso bei der Darstellung paralleler Daten im Abschnitt 2.3.

Im folgenden werden die Begriffe „Aktivität“ und „Prozeß“ synonym verwendet (auf Implementierungsebene handelt es sich dabei immer um einen oder mehrere UNIX-Prozesse, die durch Virtualisierungsschleifen alle virtuellen Prozesse simulieren).

Parallelität in Modula-2*

In Modula-2* formuliert man parallele Ausführung durch die `FORALL`s. Abbildung 2.1 stellt die Abarbeitung der synchronen und der asynchronen Variante dar und berücksichtigt auch Virtualisierungsschleifen. Darunter verstehen wir Schleifen, die ein reales PE durchläuft, um seinen Anteil an den virtuellen Prozessen abzuarbeiten, da es i.A. mehr virtuelle Prozesse als reale PEs gibt. Im Falle des einzelnen Rechners arbeitet ein einziger Prozessor in Schleifen alle virtuellen Prozesse ab.

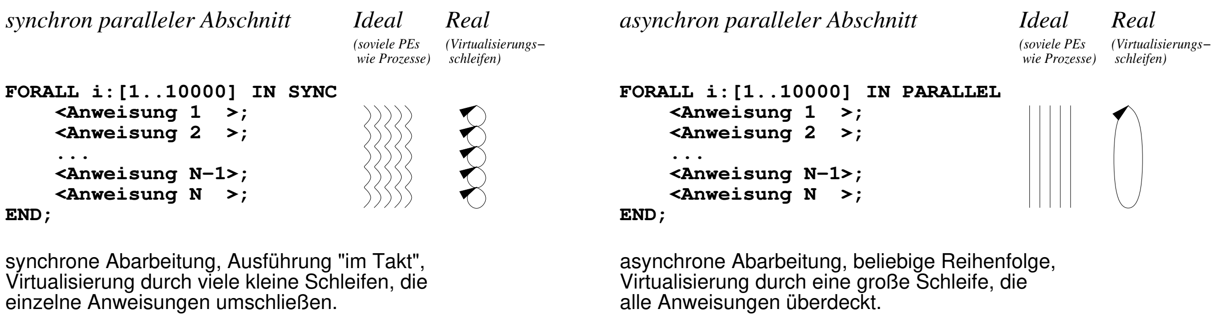


Abbildung 2.1: Abarbeitung der `FORALL`s im idealen Fall und mit Virtualisierungsschleifen.

Die Aktivitäten in einem synchronen `FORALL` sind also so aneinander gebunden, daß man sie als eine einzige großen Gruppe von Prozessen auffassen kann. Zu einem gegebenen Zeitpunkt befinden sich alle Teilprozesse an derselben Stelle des Programms oder sind inaktiv. Eine Ausnahme bilden die Zweige eines `IF–THEN–ELSE` und eines `CASE`, die zueinander asynchron laufen. Allerdings handelt es sich dabei um sehr wenige Zweige im Verhältnis zur Anzahl der Prozesse, so daß der Überblick dadurch nicht gefährdet ist.

Schwieriger ist es innerhalb eines asynchronen `FORALL`s. Die Ablaufreihenfolge ist vollkommen beliebig, so daß sich zur selben Zeit Prozesse desselben `FORALL`s an völlig verschiedenen Stellen des Programms befinden können. Man kann aber kaum jedem dieser Prozesse eine eigene Gruppe zuweisen, da sonst die Übersicht vollständig verloren ginge.

Ein Beispiel für eine mögliche Verteilung der Aktivitäten innerhalb eines synchronen und eines asynchronen `FORALL`s zeigt Abbildung 2.2. Man sieht, daß die Prozesse im synchronen Fall nur in den `THEN`- und den `ELSE`-Teil des `IF`s aufgespalten sind, während sie sich im asynchronen Fall über den gesamten Programmabschnitt verteilen.

Programmcode — *Anzahl der Prozesse je Anweisung*

```
...
FORALL i:[0..9999] IN SYNC
    ...
    V[i] := V[i+1];
    ...
    IF (V[i] > 0) THEN
        V[i] := V[i] - g;        4081
    ELSE
        V[i] := V[i] / h;        5919
    END;
    ...
END;
...
```

Synchrones FORALL, synchrone Abarbeitung:
Aufspaltung der 10000 Prozesse in zwei Gruppen
am IF–THEN–ELSE.

Programmcode — *Anzahl der Prozesse je Anweisung*

```
...
FORALL i:[0..9999] IN PARALLEL
    ...                          543
    V[i] := V[i+1];              1745
    ...                          3201
    IF (V[i] > 0) THEN
        V[i] := V[i] - g;        184
    ELSE
        V[i] := V[i] / h;        2496
    END;
    ...                          1831
END;
...
```

Asynchrones FORALL, beliebige Reihenfolge:
Aufspaltung in viele Prozesse an verschiedenen Stellen
im parallelen Programmabschnitt.

Abbildung 2.2: Verteilung von parallelen Aktivitäten im synchronen und im asynchronen Fall.

Gruppierung

Diese über das Programm verstreuten Aktivitäten gilt es also geschickt zusammenzufassen. Dazu wird der Begriff des (Sprach-) Konstrukts verwendet, der eine nichttriviale Sprachstruktur bezeichnen soll, wie etwa `WHILE`, `FOR`, `IF-THEN-ELSE`, `FORALL` und Funktionsaufrufe, nicht hingegen einzelne Anweisungen.

Bei der Gruppierung werden die Ablaufinformationen der betroffenen Aktivitäten in einer einzigen Informationsstruktur vereinigt, wodurch sich der Überblick wesentlich vereinfacht.

Als Kriterium für eine Gruppierung kann man auf die Blockstruktur des Programms zurückgreifen: Man gruppiert alle Prozesse, die dieselbe Ablaufgeschichte haben (also denselben Vorgänger im Baum besitzen) und sich auf innerster Schachtelungstiefe im selben Konstrukt befinden. Nicht zusammengefaßt werden hingegen Prozesse, die sich zwar an einer ähnlichen Stelle des Programms befinden (etwa im gleichen Konstrukt), aber eine unterschiedliche Vorgeschichte haben. Für diese Gruppierung muß natürlich Information über den Programmablauf mitgeführt werden.

Eine Zusammenfassung lediglich nach Positition des Programmzählers der Prozesse ist hingegen zu grob, weil verschiedene dynamische Aufrufe und Durchläufe eines Konstrukts auf dieselbe statische Zeile des Programms abgebildet werden.

Für MSDB wurde daher die erstgenannte Gruppierung von Aktivitäten gewählt.

2.2.2 Visualisierung des parallelen Ablaufs

Bereits im vorherigen Abschnitt wurde das Problem der Beherrschung vieler paralleler Abläufe angesprochen. Jetzt stellt sich die Frage, wie man dem Benutzer diese Aktivitäten möglichst übersichtlich präsentieren kann. Dabei geht es hier nur um die momentan aktiven Prozesse.

Wenn man durch geeignete Gruppierung die Anzahl der Prozesse reduziert hat, bietet sich die Darstellung als dynamischer Strukturbaum an – also als Baum, dessen Knoten Programmkonstrukte darstellen, und dessen Kanten für eine dynamische Vorgängerschaft im Programmablauf stehen. Dabei befindet sich an der Wurzel die oberste Schachtelungsebene des Hauptprogramms. Jeder Eintritt einer Gruppe in ein neues Sprachkonstrukt erzeugt einen neuen Knoten[2]. Wird durch ein Konstrukt der Programmablauf in weitere Gruppen aufgespalten, so werden diese als verschiedene Söhne des entsprechenden Knotens dargestellt. Durch einen Blick auf die Vorgänger eines Knotens im Baum kann man leicht seine dynamische Ablaufgeschichte erkennen. Ein Beispiel für eine solche Darstellung findet sich in Abbildung 2.3.

Möglich wird diese Darstellung durch die orthogonale Integration des `FORALL`s in Modula-2*, das genau einen Eintritts- und einen Ausgangspunkt hat (One-Entry-One-Exit-Paradigma).

Man kann den Strukturbaum als einen verallgemeinerten Programmkeller auffassen: Ein sequentielles Programm besitzt natürlich nur eine Aktivität auf einmal, hat also einen linearen Aufrufkeller. Durch eine parallele Ausführung können hingegen mehrere Aktivitäten aus einem Konstrukt, z.B. einem `FORALL`, hervorgehen. Es entsteht ein Baum von dynamisch betretenen Sprachkonstrukten. Zudem beinhaltet ein Keller nur Funktionsaufrufe, während der Strukturbaum auf Ebene der Sprachkonstrukte arbeitet.

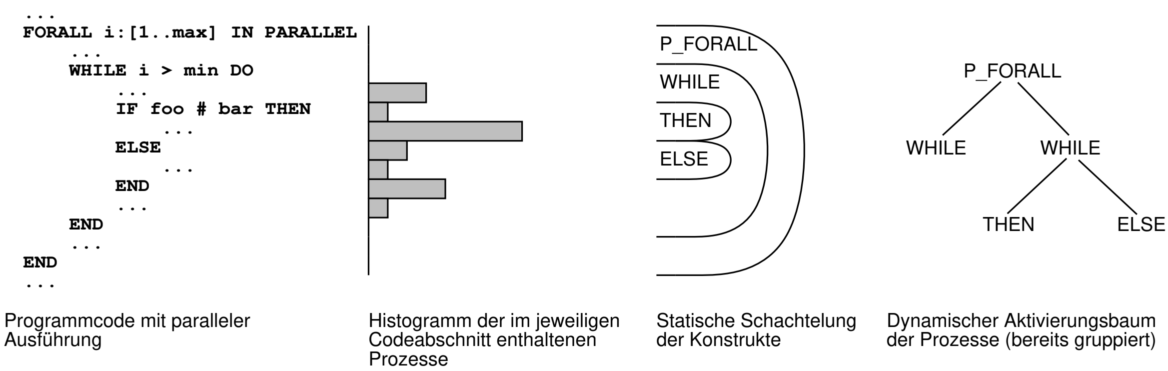


Abbildung 2.3: Möglichkeiten der Visualisierung und Abstraktion, Aufbau des Strukturbaums nach Gruppierung der Aktivitäten

Bei rekursiven Funktionen kann der Baum stark in die Tiefe wachsen, was die Übersicht und die Bildschirmdarstellung erschwert, ohne daß sich der Informationsgehalt wesentlich vergrößert. In diesem Falle könnte man den sich wiederholenden Abschnitt zwischen dem ersten und dem letzten Auftreten der Rekursion durch ein Symbol, etwa eine kurze Reihe von Punkten, ersetzen. Dadurch wäre die Übersicht gewahrt und die Information, daß eine Rekursion auftritt, erhalten.

Eine weitere Möglichkeit der Darstellung ist die einfache Auflistung aller aktiven Prozeßgruppen anhand ihrer momentanen Position im Programmcode. Diese könnte etwa durch Balken neben dem Listing des Quellcodes erfolgen, deren Breite die Zahl der momentan in der entsprechenden Zeile befindlichen Aktivitäten repräsentiert. Allerdings geht bei dieser Art der Darstellung die Information über die dynamische Vorgeschichte der Prozesse

[2]Daraus ergibt sich, daß alle Aktivität sich nur in den Blättern des Baums abspielt.

verloren. Mehrere Gruppen können sich z.B. zufällig an derselben Stelle des Programms befinden, obwohl sie zu völlig unterschiedlichen Aktivitäten gehören. Dies kann etwa vorkommen, wenn eine Funktion von verschiedenen parallelen Aktivitäten aufgerufen wird: Alle diese Prozesse befinden sich dann innerhalb der Funktion, haben aber unterschiedliche dynamische Vorgeschichten.

Eine Vereinfachung dieser Idee wäre die simple Anzeige von mehreren Programmzählern im Code, ohne die Anzahl der beteiligten Prozesse weiter aufzuschlüsseln. Allerdings geht hierbei die Übersicht schnell verloren, wenn nicht gruppiert wird. Einen Kompromiß bildet die Darstellung von mehreren Programmzähler-Pfeilen im Quellcode, deren Dicke die Anzahl der in ihren gruppierten Prozesse darstellt. Ein Anklicken der Pfeile mit der Maus könnte genauere Informationen über die einzelnen Aktivitäten bieten.

Nachteil der drei letzten Möglichkeiten ist, daß sie lediglich die Position im Quellcode zeigen, ohne auf die dynamische Vorgeschichte schließen zu lassen. Andererseits ist ein Strukturbaum allein zu grob, um die Position der Aktivitäten im Programm festzustellen, da er nur konstruktweise aufschlüsselt. Es bietet sich also eine Kombination der Methoden an, um sowohl eine Darstellung der Aufrufdynamik als auch eine genaue Positionsbestimmung zu erreichen.

MSDB verwendet eine Kombination von dynamischem Strukturbaum und verschieden dicken Programmzähler-Pfeilen im Quelltextfenster, um den parallelen Programmablauf zu erfassen.

2.2.3 Profiling-Information

Wichtig für eine Verbesserung der Leistung eines Programms sind Informationen über die Stellen, an denen die meiste Zeit verbraucht wurde. Eine Optimierung im Inneren einer geschachtelten Schleife kann Wunder wirken. Allerdings kann ein dabei eingeführter Fehler auch „Wunder“ ganz anderer Art erzeugen. Besonders frustrierend ist es, wenn sich herausstellt, daß das mühsam optimierte Programmstück praktisch nie durchlaufen wird.

Der Benutzer benötigt also Statistiken, welche Abschnitte seines Programms in der Realität wie oft durchlaufen werden. Erst nach der Analyse dieser Daten lohnt es sich, an eine Leistungssteigerung zu denken.

Im parallelen Fall sagt natürlich die reine Zahl der Durchläufe nichts über den Grad der Auslastung der PEs aus. Ein Konstrukt kann 1 000 000 mal durchlaufen werden, indem 100 000 PEs es je zehnmal abarbeiten, oder indem 10 PEs es 100 000 mal ausführen. Im ersten Fall ist die Auslastung der Maschine gut, im zweiten miserabel.

Zusätzliche Informationen darüber, wieviele Prozesse an einem Konstruktdurchlauf beteiligt sind, sind also angebracht. Andererseits kann aus Aufwandsbetrachtungen kaum für jedes virtuelle PE ein eigener Zähler verwendet werden. Als Kompromiß bietet sich ein Zähler pro realem PE an, der die Durchläufe für alle dort abgearbeiteten virtuellen PEs aufsummiert.

Aber auch die reine Zählerinformation ist schon wertvoll, wenn etwa festgestellt werden soll, welche Schleifen am häufigsten verwendet werden, um dort dann gezielt Optimierungen anzubringen. Außerdem gibt sie einen guten Eindruck von den Ausmaßen der Aktivität an verschiedenen Stellen des Programms.

Zur Sammlung von Statistiken muß der zu untersuchende Code geeignet instrumentiert werden. Dazu bieten sich z.B. die weiter unten eingeführten Debug-Funktionen an. Eine Auswertung der Statistiken läßt sich als Tabelle der am häufigsten durchlaufenen Funktionen mit Zahl ihrer Aufrufe oder grafisch als Histogramm darstellen. Eventuell kann man sie auch gleich in ein Programm-Listing integrieren. Weitergehende Möglichkeiten wie ein

Anspringen der entsprechenden Programmzeile durch einen Mausklick auf ein Histogramm können die Handhabung für den Benutzer noch vereinfachen.

Eine zusätzliche Gewichtung von Konstrukten und in ihnen enthaltenen Anweisungen nach ihrem Rechenaufwand für einen einmaligen Durchlauf könnte die praktische Aussagekraft dieser Aufstellung weiter erhöhen. Ganz genau wäre eine konkrete Messung der in jedem Befehl verbrachten Zeit, jedoch wirkt sich eine so detaillierte Betrachtung sehr negativ auf die Laufzeit des Programms aus.

Der Einfachheit und Laufzeit wegen wurde in MSDB lediglich die Zählung der Konstruktdurchläufe implementiert. Für jedes reale PE exisitert ein eigener Zähler. Weitergehende Überlegungen für mögliche Realisierungen finden sich im Ausblick in Kapitel 5.

2.2.4 Aktivierung der Prozesse

Verwandt mit der Suche nach häufig durchlaufenen Konstrukten ist die Frage nach der Verteilung von Aktivität im Programm, d.h., welche der parallelen Prozesse überhaupt aktiv sind. Es ist z.B. denkbar, daß 90% der Prozesse eines `FORALL`s aufgrund ungeschickter Programmierung desaktiviert sind, während die restlichen 10% alle Rechnungen ausführen. Wünschenswert ist natürlich eine gleichmäßigere Ausnutzung der Prozesse.

Modula-2* verdeckt allerdings vollständig die Maschinenebene, so daß die Auslastungen der realen PEs nicht zu erfassen sind. Von der Sprachidee her sollen sie das auch gar nicht sein. Die genaue Auslastung eines Parallelrechners läßt sich aus den Aktivierungen also nicht direkt ablesen.

Dennoch ist eine Darstellung der momentan aktiven virtuellen Prozesse in einem `FORALL` sinnvoll, z.B. um Muster in der Verteilung zu erkennen, oder bei der Auswertung eines Profiles festzustellen, ob wirklich ein Prozeß die gesamte Arbeit hat oder mehrere Aktivitäten vorhanden sind. Auf einem einzelnen Rechner ist real natürlich immer nur ein Prozeß aktiv, der alle virtuellen Aktivitäten abarbeitet. Auch hier sind aber Muster in der Aktivierung der virtuellen Prozesse interessant.

Für ein einzelnes `FORALL` ist die Darstellung der Aktivierungen recht einfach: Man erzeugt eine Matrix aus sovielen Pixeln, wie es Prozesse im `FORALL` gibt, und färbt die Bildpunkte dann schwarz oder weiß, je nachdem ob der entsprechende Prozeß gerade aktiv ist oder nicht. Aus einer solchen Matrix läßt sich auch schnell ein Eindruck über die Verteilung der Aktivitäten gewinnen. Ein Beispiel dafür findet sich in Abbildung 2.4.

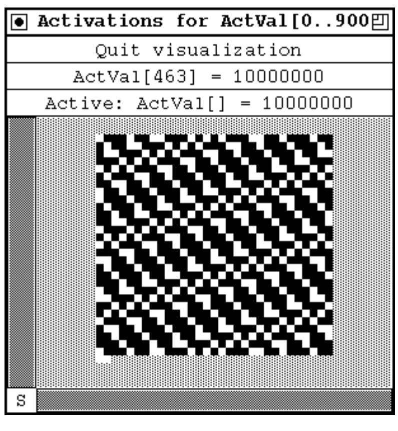


Abbildung 2.4: Beispiel für die Visualisierung der aktiven Prozesse in einem `FORALL`: Die aktiven Prozesse sind als schwarze, die momentan inaktiven als weiße Punkte dargestellt. Es läßt sich leicht ein Muster in der Verteilung erkennen.

Schwieriger wird die Lage bei verschachtelten `FORALL`s. Auf den ersten Blick kann man meinen, daß dadurch das Problem der Darstellung lediglich auf die Anzeige mehrdimensionaler Aktivierungs-Felder zurückgeführt wird. Dem ist aber leider nicht so, denn verschiedene `FORALL`s in derselben Schachtelungstiefe können verschiedene dynamische Grenzen haben.

Denkbar ist hingegen die Aufrundung aller Bereiche in einer `FORALL`-Schachtelung zum Maximum der enthaltenen Längen und eine anschließende Darstellung als n-dimensionaler Quader mit Lücken an den Stellen, wo ein `FORALL`-Bereich kleiner als das Maximum ist.

2.2.5 Semantik im Parallelen

Problemstellung

Praktisch alle aus dem Sequentiellen bekannten Leistungsmerkmale von Debuggern sind auch im Parallelen sinnvoll. Allerdings muß oft die Semantik neu überdacht werden.

Ein klassisches Beispiel dafür ist der Breakpoint, der schon Gegenstand vieler Betrachtungen war [MiCh88, Elsh88, Coop87]. Bei sequentiellen Programmen übergibt das Programm einfach die Kontrolle an den Debugger, wenn es auf einen Breakpoint stößt. Im parallelen Fall ist der Begriff der „Kontrolle“ nicht so einfach: Wird das gesamte Programm mit all seinen Prozessen angehalten? Oder wird nur der Prozeß, der auf den Breakpoint gelaufen ist, gestoppt? Soll der Stop sofort oder nach irgendeiner Synchronisation erfolgen? Was geschieht mit Prozessen in derselben Gruppe? Im folgenden werden verschiedene Semantiken diskutiert, die sich mit diesen Problemen auseinandersetzen.

Semantik in Modula-2*

Durch die Sprachdefinition von Modula-2* werden Rahmenbedingungen gesetzt, die die mögliche Semantik eingrenzen:

Asynchrones `FORALL ... IN PARALLEL`: Es werden n Aktivitäten erzeugt, die unabhängig voneinander parallel oder in beliebiger Reihenfolge ablaufen. Das `FORALL` endet, wenn alle seine Aktivitäten beendet sind.

Nur das Ende des `FORALL`s selbst bildet also eine Synchronisationsbarriere.

Dementsprechend wird die Semantik von Modula-2* nicht beeinträchtigt, wenn eine oder mehrere Aktivitäten an beliebiger Stelle des `FORALL`s angehalten werden. Die restlichen Prozesse, falls es solche gibt, laufen bis zum Ende des `FORALL`s und warten dort auf die angehaltenen Prozesse.

Synchrones `FORALL ... IN SYNC`: Die n erzeugten Aktivitäten laufen synchron zueinander ab, d.h. sie arbeiten jede Anweisung innerhalb des `FORALLS` „gleichzeitig“ ab. Eine Ausnahme sind Aufspaltungen des Kontrollflusses, wie sie etwa bei einem `IF-THEN-ELSE` oder einem `CASE` auftreten. Die Ausführung spaltet sich in mehrere Äste auf, die jeder für sich synchron, aber unabhängig von den anderen Ästen, ausgeführt werden und sich am Ende des Konstrukts wieder zum ursprünglichen synchronen Ablauf vereinigen. Auch das synchrone `FORALL` endet mit Ablauf aller seiner Aktivitäten.

Synchronisationsbarrieren sind demnach alle Anweisungen, das Ende des `FORALL`s, und (bei aufgespaltenem Kontrollfluß) die Wiedervereinigung verschiedener Äste. Sogar innerhalb von Anweisungen können Synchronisationspunkte vorkommen.[3]

Die Semantik vom Modula-2* verlangt also, daß alle Prozesse eines Ausführungsastes gleichzeitig und an derselben Stelle angehalten werden, während die Ausführung eventuell vorhandener parallel ablaufender Äste weitergehen kann. Diese warten dann am Punkt der Wiedervereinigung der Kontrollflüsse.

Mögliche Semantiken eines Breakpoints

Aus den Bedingungen von Modula-2* ergeben sich somit verschiedene Varianten eines Breakpoints, die im folgenden dargestellt und im Hinblick auf Realisierbarkeit und verschiedene Maschinenarchitekturen untersucht werden.

Asynchroner oder synchroner Stop: Der Breakpoint kann entweder asynchron an beliebiger Stelle wirken, oder nur an Synchronisationspunkten erfolgen.

Asynchroner Stop: Der Prozeß, der auf einen Breakpoint trifft, wird sofort und ohne Rücksicht auf andere Prozesse angehalten.

Diese Semantik ist für den asynchronen Fall zulässig, nicht aber für den synchronen.

Dort können nämlich inkonsistente Zustände sichtbar werden, wenn z.B. der Breakpoint mitten in einer Virtualisierungsschleife liegt, so daß nur ein Teil der virtuellen Aktivitäten bereits abgearbeitet ist. Modula-2* verlangt aber, daß alle Aktivitäten synchron laufen.

Synchroner Stop: Der Breakpoint wirkt als Vormerkung, am nächsten Synchronisationspunkt anzuhalten, oder ist selbst ein Synchronisationspunkt.

Diese Variante ist sowohl für den synchronen als auch für den asynchronen Fall erlaubt.

Ein Vormerken eines Breakpoints und Stop an der Stelle der nächsten Synchronisierung ist in der Implementierung sehr aufwendig.

Realisierbar ist der synchrone Stop im einfachsten Fall dadurch, daß nur Synchronisationspunkte als mögliche Breakpoints angenommen werden. Das ist im synchronen `FORALL` kein Problem, aber im asynchronen Falle ist damit das Ende des `FORALL`s der einzige mögliche Breakpoint, was offensichtlich nicht ausreicht. Innerhalb des `FORALL`s kann es nämlich zu Endlosschleifen kommen, wodurch der Breakpoint nie erreicht wird. Außerdem ist eine feinere Granularität der Breakpoints im Programm wünschenswert.

Folglich müssen für die asynchrone Parallelität entweder künstlich potentielle Breakpoints in Form von Synchronisationsstellen eingefügt werden, was Leistungseinbußen mit sich bringt, oder es wird zusätzlich ein asynchroner Stop geboten.

Lokaler oder globaler Stop: Der Breakpoint kann lokal auf den betroffenen Prozeß (oder dessen Gruppe) wirken, oder global alle Prozesse anhalten.

Lokaler Stop: Nur der Prozeß, der auf den Breakpoint trifft, wird angehalten. Alle anderen Prozesse laufen weiter, und warten ggf. an der nächsten Synchronisationsbarriere auf diesen Prozess.

[3] Die vom Übersetzer bei der Optimierung vorgenommene Elimination von Synchronisationsbarrieren wird zur Fehlersuche ausgeschaltet.

Asynchron ist diese Semantik erlaubt. Im synchronen `FORALL` muß sich der Stop auf ganze Prozeßgruppen beziehen, etwa in den Ästen eines `IF-THEN-ELSE`.

Auf einem MIMD-System, wie z.B. einem Netzwerk von Workstations oder einem Transputercluster, ist der lokale Stop der Normalfall, weil die anderen Prozessoren unabhängig weiterlaufen können. Dabei ist allerdings zu beachten, daß auch dort auf jedem realen PE viele virtuelle PEs durch Virtualisierungsschleifen simuliert werden. Eine dieser Aktivitäten unabhängig von den anderen in derselben Schleife anzuhalten, ist kaum möglich.

Im SIMD-Fall oder auf einem einzelnen Rechner ist der lokale Stop aus der gleichen Überlegung heraus nur schwer zu realisieren, da dort alle Prozesse dieselben Anweisungen ausführen bzw. in einer einzigen Virtualisierungsschleife abgearbeitet werden.

Globaler Stop: Alle Prozesse werden angehalten, sobald einer von ihnen auf einen Breakpoint läuft.

Im asynchronen Fall ist diese Variante jedenfalls zulässig. Beim synchronen `FORALL` tritt sie automatisch auf, wenn die einzige Gruppe des `FORALL`s auf einen Breakpoint trifft. Bei mehreren Gruppen müssen diese jeweils am nächsten Synchronisationspunkt angehalten werden.

Bei SIMD-Architekturen oder einzelnen Rechnern ist dies der Normalfall, da dort nur ein realer Prozeß vorliegt.

Im MIMD-Fall müssen alle realen PEs vom Auftreten des Breakpoints benachrichtigt werden und alle ihre Prozesse anhalten. Da in Modula-2* alle Prozesse am Ende eines `FORALL`s aufeinander warten, wird die Semantik nicht beeinträchtigt, wenn die Stop-Anforderung im asynchronen Fall auf manchen PEs erst nach Beendigung aller dortigen Aktivitäten des `FORALL`s eintrifft. Diese befindet sich dann nämlich im Wartezustand befinden und können dort unterbrochen werden.

Zusammenfassend läßt sich sagen, daß aus obengenannten Gründen ein Breakpoint global sein sollte. Innerhalb eines synchronen `FORALL`s muß er synchron erfolgen, also auf einen Synchronisationspunkt fallen, was aber durch die Vielzahl der Synchronisationsbarrieren in diesem Fall nicht schwer ist. Im asynchronen `FORALL` sollte er an beliebiger Stelle asynchron erfolgen können.

Einzelschrittablauf

Für die Verfolgung des Programmablaufs ist auch der Einzelschritt, d.h. die Ausführung lediglich der nächsten Zeile im Programm, von Vorteil. Vom Konzept her entspricht er dem Setzen eines Breakpoints nach der nächsten Zeile, dem Weiterlauf der Ausführung bis dorthin, und dem anschließenden Entfernen des Breakpoints.

Nach den vorherigen Überlegungen wird ein globaler Einzelschritt gewählt, bei dem alle Prozesse die jeweils nächste Programmzeile ausführen. Im synchronen Kontext betrifft das die gesamte Prozeßgruppe, d.h. es wird die die Zeile umgebende Virtualisierungsschleife vollständig durchlaufen. Im asynchronen Fall führt lediglich der momentan abgearbeitete virtuelle Prozeß einen Schritt durch, da die Ausführung einer bestimmten Zeile für alle Indizes der das `FORALL` realisierenden Virtualisierungsschleife kaum machbar wäre.

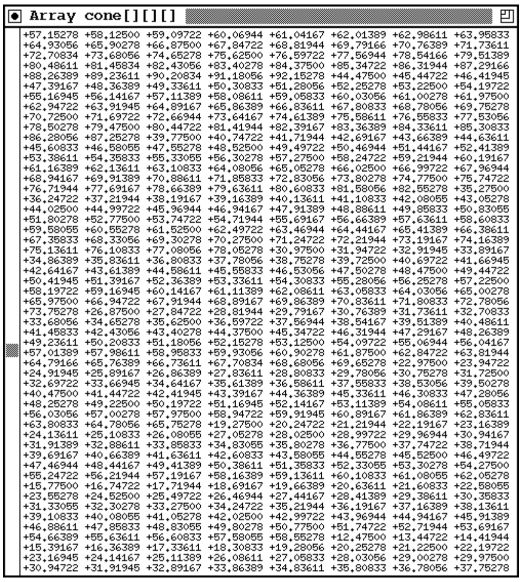
2.3 Repräsentation großer Datenmengen

Nachdem der Ablauf paralleler Programme und die dabei auftretenden Probleme betrachtet wurden, sollen jetzt einige Punkte zur Behandlung der parallelen Daten näher untersucht werden.

2.3.1 Prinzipielle Anforderungen

Wie schon bei den parallelen Aktivitäten, die es übersichtlich zu präsentieren gilt, stellt sich auch hier das Problem, dem Benutzer einen Überblick über die Masse von Daten zu geben, die bei einem parallelen Programm anfällt.

Mit rein textueller Ausgabe ist dabei niemandem geholfen – spätestens beim Anblick eines meterlangen Listings von Zahlenkolonnen (oder noch schlimmer, mehreren unübersichtlichen Bildschirmseiten) wird wohl jedem Programmierer die Suche nach Strukturen darin verleidet. Ein abschreckendes Beispiel zeigt Abbildung 2.5.



+57.15278	+58.12500	+59.09722	+60.06944	+61.04167	+62.01389	+62.98611	+63.95833
+64.93056	+65.90278	+66.87500	+67.84722	+68.81944	+69.79166	+70.76389	+71.73611
+72.70834	+73.68056	+74.65278	+75.62500	+76.59722	+77.56944	+78.54166	+79.51389
+80.48611	+81.45834	+82.43056	+83.40278	+84.37500	+85.34722	+86.31944	+87.29166
+88.26389	+89.23611	+90.20834	+91.18056	+92.15278	+44.47500	+45.44722	+46.41945
+47.39167	+48.36389	+49.33611	+50.30833	+51.28056	+52.25278	+53.22500	+54.19722
+55.16945	+56.14167	+57.11389	+58.08611	+59.05833	+60.03056	+61.00278	+61.97500
+62.94722	+63.91945	+64.89167	+65.86389	+66.83611	+67.80833	+68.78056	+69.75278
+70.72500	+71.69722	+72.66944	+73.64167	+74.61389	+75.58611	+76.55833	+77.53056
+78.50278	+79.47500	+80.44722	+81.41944	+82.39167	+83.36389	+84.33611	+85.30833
+86.28056	+87.25278	+39.77500	+40.74722	+41.71944	+42.69167	+43.66389	+44.63611
+45.60833	+46.58055	+47.55278	+48.52500	+49.49722	+50.46944	+51.44167	+52.41389
+53.38611	+54.35833	+55.33055	+56.30278	+57.27500	+58.24722	+59.21944	+60.19167
+61.16389	+62.13611	+63.10833	+64.08056	+65.05278	+66.02500	+66.99722	+67.96944
+68.94167	+69.91389	+70.88611	+71.85833	+72.83056	+73.80278	+74.77500	+75.74722
+76.71944	+77.69167	+78.66389	+79.63611	+80.60833	+81.58056	+82.55278	+35.27500
+36.24722	+37.21944	+38.19167	+39.16389	+40.13611	+41.10833	+42.08055	+43.05278
+44.02500	+44.99722	+45.96944	+46.94167	+47.91389	+48.88611	+49.85833	+50.83055
+51.80278	+52.77500	+53.74722	+54.71944	+55.69167	+56.66389	+57.63611	+58.60833
+59.58055	+60.55278	+61.52500	+62.49722	+63.46944	+64.44167	+65.41389	+66.38611
+67.35833	+68.33056	+69.30278	+70.27500	+71.24722	+72.21944	+73.19167	+74.16389
+75.13611	+76.10833	+77.08056	+78.05278	+30.97500	+31.94722	+32.91945	+33.89167
+34.86389	+35.83611	+36.80833	+37.78056	+38.75278	+39.72500	+40.69722	+41.66945
+42.64167	+43.61389	+44.58611	+45.55833	+46.53056	+47.50278	+48.47500	+49.44722
+50.41945	+51.39167	+52.36389	+53.33611	+54.30833	+55.28056	+56.25278	+57.22500
+58.19722	+59.16945	+60.14167	+61.11389	+62.08611	+63.05833	+64.03056	+65.00278
+65.97500	+66.94722	+67.91944	+68.89167	+69.86389	+70.83611	+71.80833	+72.78056
+73.75278	+26.87500	+27.84722	+28.81944	+29.79167	+30.76389	+31.73611	+32.70833
+33.68056	+34.65278	+35.62500	+36.59722	+37.56944	+38.54167	+39.51389	+40.48611
+41.45833	+42.43056	+43.40278	+44.37500	+45.34722	+46.31944	+47.29167	+48.26389
+49.23611	+50.20833	+51.18056	+52.15278	+53.12500	+54.09722	+55.06944	+56.04167
+57.01389	+57.98611	+58.95833	+59.93056	+60.90278	+61.87500	+62.84722	+63.81944
+64.79166	+65.76389	+66.73611	+67.70834	+68.68056	+69.65278	+22.97500	+23.94722
+24.91945	+25.89167	+26.86389	+27.83611	+28.80833	+29.78056	+30.75278	+31.72500
+32.69722	+33.66945	+34.64167	+35.61389	+36.58611	+37.55833	+38.53056	+39.50278
+40.47500	+41.44722	+42.41945	+43.39167	+44.36389	+45.33611	+46.30833	+47.28056
+48.25278	+49.22500	+50.19722	+51.16945	+52.14167	+53.11389	+54.08611	+55.05833
+56.03056	+57.00278	+57.97500	+58.94722	+59.91945	+60.89167	+61.86389	+62.83611
+63.80833	+64.78056	+65.75278	+19.27500	+20.24722	+21.21944	+22.19167	+23.16389
+24.13611	+25.10833	+26.08055	+27.05278	+28.02500	+28.99722	+29.96944	+30.94167
+31.91389	+32.88611	+33.85833	+34.83055	+35.80278	+36.77500	+37.74722	+38.71944
+39.69167	+40.66389	+41.63611	+42.60833	+43.58055	+44.55278	+45.52500	+46.49722
+47.46944	+48.44167	+49.41389	+50.38611	+51.35833	+52.33055	+53.30278	+54.27500
+55.24722	+56.21944	+57.19167	+58.16389	+59.13611	+60.10833	+61.08055	+62.05278
+15.77500	+16.74722	+17.71944	+18.69167	+19.66389	+20.63611	+21.60833	+22.58055
+23.55278	+24.52500	+25.49722	+26.46944	+27.44167	+28.41389	+29.38611	+30.35833
+31.33055	+32.30278	+33.27500	+34.24722	+35.21944	+36.19167	+37.16389	+38.13611
+39.10833	+40.08055	+41.05278	+42.02500	+42.99722	+43.96944	+44.94167	+45.91389
+46.88611	+47.85833	+48.83055	+49.80278	+50.77500	+51.74722	+52.71944	+53.69167
+54.66389	+55.63611	+56.60833	+57.58055	+58.55278	+12.47500	+13.44722	+14.41944
+15.39167	+16.36389	+17.33611	+18.30833	+19.28056	+20.25278	+21.22500	+22.19722
+23.16945	+24.14167	+25.11389	+26.08611	+27.05833	+28.03056	+29.00278	+29.97500
+30.94722	+31.91945	+32.89167	+33.86389	+34.83611	+35.80833	+36.78056	+37.75278

Abbildung 2.5: Ungeeignete Darstellung großer Datenmengen

Hier hilft nur die Visualisierung der Daten. Durch eine geeignete Darstellung kann man sich schnell einen Eindruck von großen Datenmengen machen und dann kritische Stellen im Detail untersuchen.

Dabei sollten die Daten anhand einstellbarer Kriterien gruppiert werden können, so daß der Benutzer Strukturen leichter erkennt. Um die Verteilung der Datenwerte zu erfassen, ist eine abgestufte Darstellung, z.B. mit Graustufen oder Farben, von Vorteil.

Natürlich muß es auch möglich sein, den genauen Wert an einer bestimmten Stelle zu erfahren. Die Visualisierung darf die eigentlichen dargestellten Daten nicht völlig unzugänglich machen. Abhilfe schafft etwa die genaue Anzeige des Werts, auf dem sich der Mauszeiger jeweils gerade befindet.

Wie schon zu Anfang des Entwurfskapitels erwähnt, findet sich eine aus [Mill92] entnommene Liste mit Kriterien für eine wirkungsvolle Visualisierung in Anhang A. Dort wird auch ein Vergleich mit der erfolgten Implementierung angestellt.

2.3.2 Visualisierung mehrdimensionaler Felder[4]

Es fällt schwer, sich mehrdimensionale Gebilde räumlich vorzustellen oder einen Überblick über sie zu gewinnen. Leichter wird diese Aufgabe, wenn man ein Gebilde entlang verschiedener Ebenen zerschneiden und sich diese Schnitte dann ansehen kann. Durch Verändern der restlichen Koordinaten ist man dann in der Lage, durch den Körper regelrecht „durchzublättern“ und sich einen Eindruck seiner Strukturen zu verschaffen. Ein Schema dieser Vorgehensweise findet sich in Abbildung 2.6. Dreidimensionale Gebilde kann man natürlich durch Einsatz von farbig schattierter 3D-Grafik noch intuitiver veranschaulichen, doch scheitert dieser Ansatz bei komplexen Gebilden, spätestens jedoch bei Körpern mit mehr als drei Dimensionen. Außerdem ist der damit verbundene Rechenaufwand im Rahmen eines interaktiven Debuggers nicht zu vertreten.

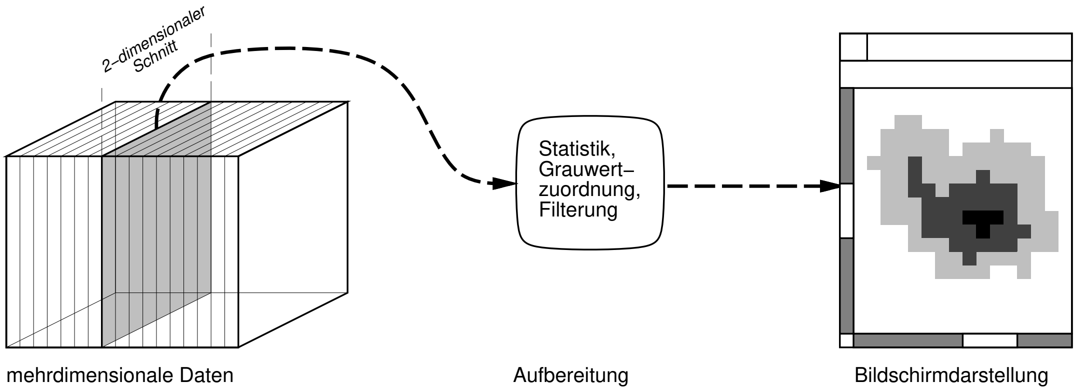


Abbildung 2.6: Schema der Darstellung mehrdimensionaler Felder durch zweidimensionale Schnitte

Aus dieser Überlegung heraus entstand die Visualisierung von Feldern. Sie stellt die Mittel zur Darstellung einer Schnittfläche durch einen mehrdimensionalen Körper zur Verfügung. Dem Benutzer wird der Schnitt als Ebene angezeigt, in der die Punkte nach bestimmten Kriterien markiert sind, wie in Abbildung 2.7 deutlich wird. Die Datenwerte werden hierbei statistisch ausgewertet und bekommen Graustufen zugewiesen, um einen Eindruck ihrer Verteilung zu gewinnen.

Eine weitere Möglichkeit ist die Darstellung von Wahrheitsprädikaten über die Datenelemente. Die Visualisierung könnte etwa alle Elemente hervorheben, die ein bestimmtes Kriterium erfüllen, z.B. `a[i,j,k] > 10`. Dies erlaubt die Klassifizierung in Kategorien, die dem Benutzer sinnvoll erscheinen. Ein Beispiel findet sich in Abbildung 2.8, wo als Kriterium ein Wertebereich gegeben ist. Andere Realisierungen können auch komplexere Gruppierungsmerkmale unterstützen, bis hin zu beliebigen Ausdrücken der verwendeten Programmiersprache.

Man erhält also einen Schnitt durch das mehrdimensionale Gebilde, der sehr aussagekräftig ist. Wird er zu großflächig, etwa weil die Zahl der Datenelemente auch in einer einzigen Schnittebene zu groß für die Anzeige ist, so kann er durch ein Ausschnittsfenster hindurch betrachtet werden, das sich über den gesamten Schnitt bewegen läßt. Zudem ist der Vergrößerungsfaktor, mit dem Punkte des Schnitts dargestellt werden, variabel, um kleinere Datensätze bei Bedarf vergrößert zu betrachten.

Bei MSDB werden verschiedene Arten von Visualisierungen verwendet, darunter auch die hier abgebildeten.

[4] Dieser Abschnitt lehnt sich eng an meine Studienarbeit [Haen91] an.

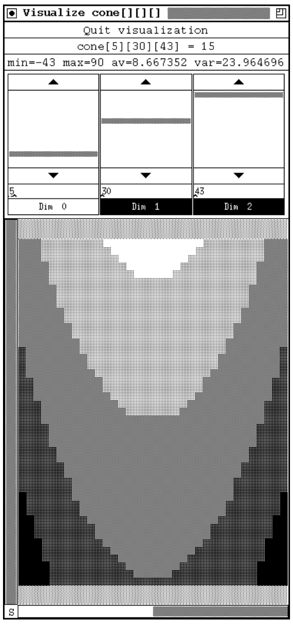


Abbildung 2.7: Visualisierung von mehrdimensionalen Daten: Schnitt durch einen 3-dimensionalen Kegel mit automatischer Zuordnung von Datenwerten zu Graustufen im Bild.

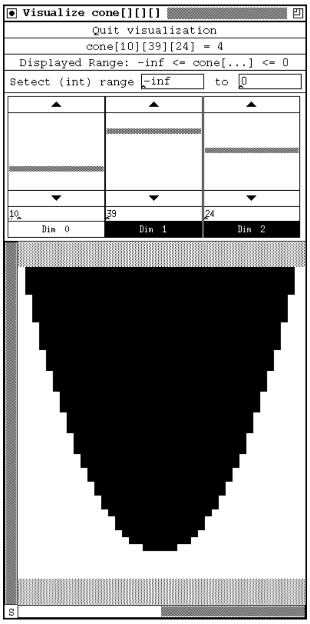


Abbildung 2.8: Visualisierung von mehrdimensionalen Daten: Schnitt durch einen 3-dimensionalen Kegel mit Färbung der Bildpunkte nach benutzerdefinierten Booleschen Kriterien, hier durch Eingabe eines Wertebereiches. Alle Werte im Intervall [$-\infty \ldots 0$] sind schwarz dargestellt.

2.4 Implementierungsentscheidungen

Im Rahmen des Entwurfs wurden auch grundlegende Entscheidungen gefällt, die sich konkreter mit der zu erfolgenden Implementierung befassen:

2.4.1 Gewinnung von Debug-Information

Die übliche Methode zur Gewinnung von Informationen aus einem Programm durch den Debugger ist die genaue Analyse von Datenstrukturen aus dessen Adreßraum, z.B. von Programmzähler oder Aufrufkeller. Der Vorteil ist, daß das Programm zur Fehlersuche im Prinzip nicht verändert werden muß. Allerdings ist auch in diesem Falle für eine sinnvolle Fehlersuche die Neuübersetzung nötig, um wichtige statische Informationen wie Symbolnamen und Korrespondenzen zwischen Maschinencode und Quellcode zu erzeugen, auf die der Debugger dann aufsetzen kann – beim C-Übersetzer unter UNIX muß man z.B. das Debug-Flag „`-g`“ verwenden. In praktisch allen Fällen wird eine gleichzeitige Optimierung des Codes durch den Übersetzer ausgeschlossen, da deren Transformationen eine Abbildung zwischen dem erzeugten Programm und dem Quellcode äußerst schwer machen[5].

Wenn man jedoch sowieso das Programm neu übersetzen muß und auf eine Optimierung verzichtet, wieso sollte man dann nicht gleich zusätzliche nützliche Informationen gewinnen?

Aus dieser Überlegung heraus entstand die Idee der Debug-Funktionen, die im Kapitel 3.5 näher behandelt werden. Beim Übersetzen werden in das Programm an wichtigen Stellen Funktionsaufrufe eingefügt, die Informationen über den aktuellen Zustand des Pro-

[5] Eine Ausnahme bildet z.B. der `gcc` C-Übersetzer der Free Software Foundation in seiner neuesten Version.

gramms (dynamische Aufrufstruktur und Profiling-Daten) während des Ablaufs anlegen. Das Programm zeichnet also selbst die relevanten Daten auf, die der Debugger sich sonst erst mühsam erschließen müßte.

Wird der Programmablauf nun angehalten, so läßt sich aus diesen Daten der Zustand des Programms direkt ablesen. So wird z.B. die dynamische Aufrufstruktur direkt auf einen Strukturbaum abgebildet, den man wie in Abschnitt 2.2.2 vorgeschlagen verwenden kann.

Ein Nachteil dieser Vorgehensweise ist eine Verlangsamung des Programms, da das Anlegen der Zusatzinformation einen Teil der Zeit beansprucht. Außerdem ist es denkbar, daß durch einen Fehler im Programm die aufgezeichnete Information beschädigt wird, da sie sich nicht im geschützten Adreßraum des Debuggers, sondern direkt beim Programm befindet. Allerdings ist, wie oben aufgeführt, der zu untersuchende Code sowieso nicht optimal, also fällt der erste Punkt nicht besonders ins Gewicht. Das zweite Problem läßt sich nicht ganz eliminieren, aber durch kompakte Datenstrukturen der Debug-Information zumindest unwahrscheinlicher machen. Man könnte auch die Informationen schon bei der Gewinnung an den Debugger weiterreichen, um sie dort sicher zu speichern. Dann müßte der Debugger aber während des Programmablaufs permanent aktiv werden und das gesamte System entsprechend große Leistungseinbußen hinnehmen (mehr dazu im Kapitel über die Implementierung).

2.4.2 Abbildung zwischen Modula-2* und C

Da der `msc`-Übersetzer für Modula-2* nicht direkt Code für die jeweilige Maschine erzeugt, sondern ein Programm im parallelen C-Dialekt des Rechners ausgibt, ergeben sich einige Besonderheiten für den Debugger.

Man kann einerseits schon auf einem recht hohen Niveau mit der Fehlersuche ansetzen, indem man einen vorhandenen symbolischen Debugger für C verwendet (siehe dazu auch den nächsten Abschnitt).

Die große Schwierigkeit besteht andererseits darin, eine für den Benutzer transparente Abbildung vom untersuchten C-Programm auf das ursprüngliche Modula-2*-Programm zu leisten. Das ist aufgrund der vielfältigen Transformationen, die `msc` auch schon ohne Optimierung vornimmt[6], leider nicht trivial. Auch hierzu finden sich weitere Details im Implementierungskapitel.

2.4.3 Aufsetzen auf einen vorhandenen Debugger

Um das Rad nicht neu zu erfinden und um die Portabilität zu erhöhen, bietet es sich an, auf einem bereits vorhandenen Standard-Debugger aufzusetzen und dessen Funktionalität anzureichern. Zudem gibt es für C, das wie gerade erläutert die Grundlage des `msc` bildet, bereits ausgereifte Werkzeuge.

Für den einzelnen Rechner bietet sich ein Source-Level-Debugger für C an, der auf praktisch allen UNIX-Systemen vorhanden ist. Durch eine geeignete Schnittstelle kann man von der Low-Level-Implementierung eines Debuggers also ganz absehen, und sich stattdessen auf den vorhandenen C-Debugger verlassen, um die Arbeit auf die höheren Funktionen zu konzentrieren. Dabei stellt sich wieder ein ähnliches Transformations-Problem wie im vorherigen Abschnitt: Die Ausgaben des Debuggers müssen auf Ausdrücke der Ebene von Modula-2* abgebildet werden, und alle Eingaben des Benutzers müssen wieder auf C-Ebene übertragen werden.

[6] Erwähnt seien z.B. Umbenennung von Variablen, Einführung neuer Variablen, Initialisierungsabschnitte, Virtualisierungsschleifen, und Umstellung von u.a. `THEN` und `ELSE` im synchronen Kontext

Auch für ein Netzwerk aus Rechnern kann dieser Ansatz verwendet werden, wenn man für jedes reale PE einen eigenen Debugger startet und diese geeignet koordiniert. In diesem Fall läßt sich ebenfalls ein UNIX-Debugger verwenden.

Allerdings ist auf den eigentlichen Parallelrechnern wie z.B. der MasPar keinerlei Standard für die Fehlersuche vorhanden. Hier ist man also wirklich auf eine eigene Implementierung der unteren Systemebene angewiesen.

Kapitel 3

Implementierung

In diesem Kapitel werden die im Entwurf diskutierten Punkte aufgegriffen und die konkret erfolgte Implementierung beschrieben – die Reihenfolge der Darstellung orientiert sich dabei an der Anordnung im Entwurf. An den entsprechenden Stellen werden auch die Gründe der jeweiligen Entscheidung erläutert, sofern sie von der im Entwurf geplanten Richtung abweicht. Außerdem finden sich hier die maschinenspezifischen Details, von denen vorher abstrahiert wurde.

3.1 Einführung in den MSDB

Um einen Eindruck von den Ergebnissen der Implementierung zu geben, wird im folgenden ein exemplarischer Bildschirmabzug des MSDB in Aktion erklärt, den man in Abbildung 3.1 sieht.

Die Fenster haben im einzelnen folgende Funktionalität:

Hauptfenster mit Quellcode: Links oben ist das Hauptfenster des MSDB, von dem aus die meisten Aktionen gesteuert werden. Es gliedert sich in eine Statuszeile, den Modula-2*-Quellcode, eine Reihe von Knöpfen zur Bedienung des Debuggers und den Abschnitt mit der Ausgabe des Debuggers, in dem man auch direkt Befehle textuell eingeben kann.

Im Quellcodefenster sind ein Breakpoint (dargestellt durch eine stilisierte Hand) und ein Programmzähler-Pfeil zu sehen. Breakpoints werden durch Anklicken der entsprechenden Zeile und Anwählen des stop at Knopfes eingefügt, und durch den delete Knopf wieder gelöscht. next und step führen einen Einzelschritt ohne bzw. mit Durchlaufen von Funktionsaufrufen durch.

Die Untersuchung von Variablen oder Funktionen geschieht durch Markierung im Quelltext, gefolgt von Anklicken eines passenden Knopfes. print gibt z.B. den aktuellen Wert der Variablen aus.

C-Code Fenster: Dieses optionale Fenster in der rechten oberen Ecke stellt den zum Modula-2*-Programm gehörigen C-Quelltext dar, samt Breakpoints und Programmzählern. Ein Schritt im Modula-2*-Programm kann mehreren Schritten im C-Code entsprechen, die dann automatisch am Stück abgearbeitet werden.

Durch Maus-Aktionen in diesem Fenster wird es aktiv und man kann direkt hier Breakpoints setzen oder löschen. Die Aktivierung kann man auch direkt durch den ACTIVE bzw. inactive Knopf auslösen, der bei Klick das entsprechende Fenster aktiviert.

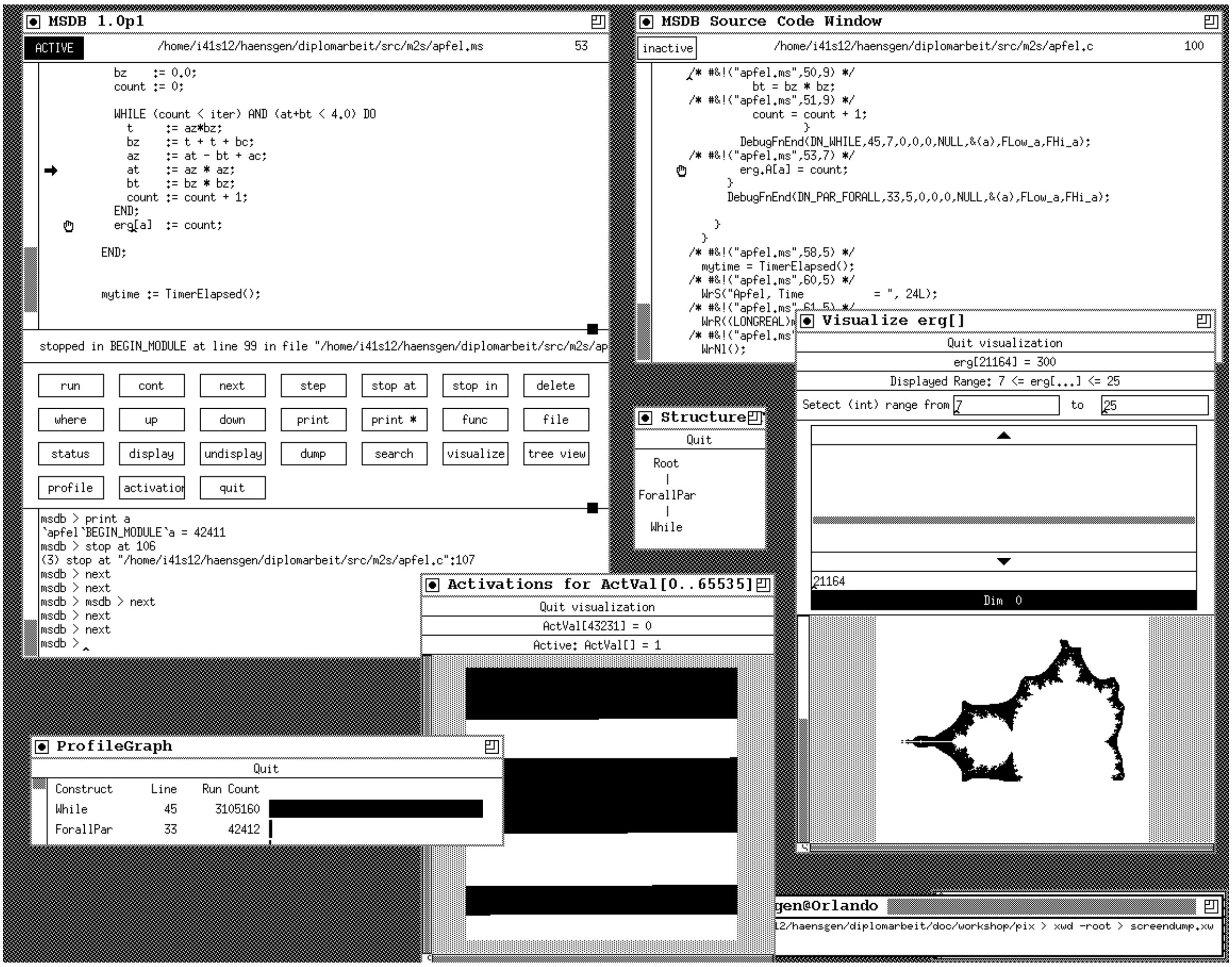


Abbildung 3.1: MSDB in Aktion

Im verteilten Fall kann der interessierte Benutzer für jedes reale PE ein eigenes C-Fenster erhalten, um den Ablauf auf unterer Ebene mitzuverfolgen. Zur eigentlichen Fehlersuche ist es nicht nötig.

Visualisierung: Rechts in der Mitte sieht man eine Visualisierung des Rechenergebnisses. Sie wurde mit dem **visualize** Knopf aufgerufen. Es handelt sich in diesem Beispiel um ein Apfelmännchen, da man dort besonders einfach Ergebnisse grafisch umsetzen kann. Die Berechnung ist zu etwa 3/4 fertig; das untere Viertel steht noch aus.

In diesem Fall wird eine Visualisierung mit Bereichen verwendet. Der Benutzer hat angegeben, daß alle Werte zwischen 7 und 25 schwarz dargestellt werden sollen. Das eindimensionale Feld wurde zur Anzeige auf eine zweidimensionale Fläche abgebildet.

Der Datenwert an der Position 21164, wo der Mauszeiger zuletzt war, ist oben in der Visualisierung angegeben. Darunter befindet sich der Schiebebalken zur Einstellung der Position, der sich bei Bewegung der Maus in der Datendarstellung selbst auch mitbewegt.

Strukturbaum: Der aktuelle Strukturbaum des Programms ist in der Bildmitte angezeigt. Die Ausführung befindet sich gerade innerhalb eines `WHILE`, dessen dynamischer Vorgänger das asynchrone `FORALL` ist. Sichtbar gemacht wurde der Baum durch den tree view Knopf.

Aktivierungen: In der Mitte unten sieht man eine Darstellung der momentanen Aktivierungen der virtuellen Prozesse, angefordert durch den activation Knopf. Prozesse mit Aktivierungswert 1 sind aktiv und schwarz gefärbt, alle anderen sind inaktiv und weiß. Der Aktivierungswert an der letzten Stelle des Mauszeigers (43231) ist 0, wie die zweitoberste Zeile der Visualisierung zeigt.

Profile-Graph: Eine grafische Aufbereitung der Zahl der Konstruktdurchläufe findet man als Histogramm im linken unteren Bildschirmbereich. Mit angegeben sind das jeweilige Konstrukt, die Zeile im Code und die exakte Zahl der Durchläufe.

Die Zeilen im Profile sind nach Zahl der Durchläufe geordnet, so daß die am meisten benutzte Struktur leicht erkennbar oben steht. Erzeugt wurde die Darstellung mit dem profile Knopf.

Damit sollte ein kleiner Eindruck der verschiedenen Möglichkeiten, die MSDB bietet, entstanden sein.

3.2 Repräsentation der parallelen Ausführung

Nach dem Überblick über die Funktionalität des MSDB werden nun die Details der Implementierung unter Beachtung der im Entwurf aufgeführten Gesichtspunkte beschrieben und beurteilt.

3.2.1 Gruppierung von Ablaufeinheiten

Basierend auf den Überlegungen des Entwurfs wurden Debug-Funktionen implementiert, die während des Programmablaufs dynamisch einen die parallele Aufrufstruktur des Programms darstellenden Strukturbaum aufbauen (siehe 3.5).

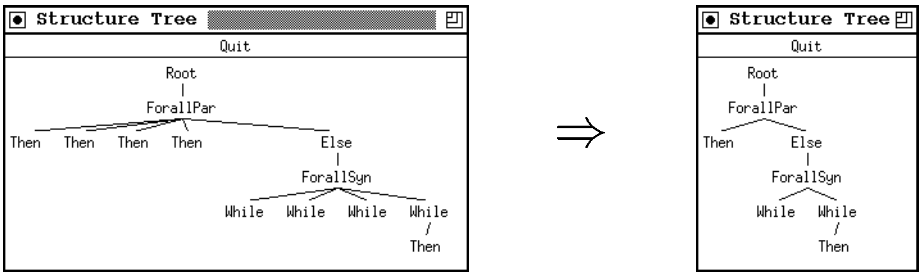


Abbildung 3.2: Strukturbäume vor und nach dem Gruppieren, hier am Beispiel der Zusammenfassung von Prozessen auf 9 realen PEs (9 Blätter im ursprünglichen Baum).

Diese Strukturbäume werden bei der Fehlersuche vom Debugger eingelesen, der dabei ggf. auf mehrere Debugger-Prozesse verteilte Informationen kombiniert. Anschließend werden im resultierende Gesamt-Strukturbaum Knoten zusammengefaßt, die zueinander äquivalent sind. Ein einfaches Beispiel für einen solchen Gruppierungsvorgang findet sich in Abbildung 3.2.

Der Gruppierungsalgorithmus faßt rekursiv Knoten im Baum zusammen, die die folgenden Bedingungen erfüllen:

- Gleicher Vater, d.h. gleiche dynamische Vorgeschichte.

- Gleiches Konstrukt, d.h. beide Knoten sind im selben Kontext, haben die selbe Position im Quelltext und das selbe Konstrukt.
- Gleiche Kinder, d.h. die jeweiligen Kinder der Knoten (falls es solche gibt) sind ihrerseits wieder gruppierbar.[1]

Die Information der einzelnen Knoten geht dabei nicht verloren, sondern wird im aus der Zusammenfassung resultierenden Knoten mit gespeichert. Damit ist sie für eine detailliertere Untersuchung problemlos zugänglich.

3.2.2 Visualisierung des parallelen Ablaufs

Basierend auf dem zusammengefaßten Strukturbaum des parallelen Programms wird eine grafische Darstellung dieses Baumes erzeugt. Einzelne Knoten, die durch Gruppierung aus mehreren Knoten entstanden sind, können expandiert werden, indem man mit der Maus auf sie klickt. Einfache Knoten, die nicht aus Gruppierungen hervorgingen, liefern bei einem Klick Informationen über das zugehörige Konstrukt wie Zeilenbereich, Kontext (sequentiell, synchron oder asynchron) und Zahl der Durchläufe durch dieses Konstrukt.

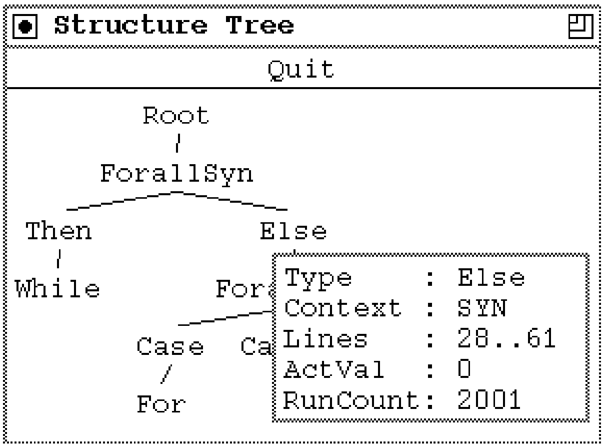


Abbildung 3.3: Beispiel eines Strukturbaumes: Die Grundaktivität spaltet sich in parallele Ausführungen auf. Details über den gerade ausgewählten ELSE-Knoten sind in einem eigenen Informationsfenster dargestellt.

Auf einem einzelnen Rechner degeneriert der Strukturbaum zu einer Strukturliste, da immer nur eine Aktivität auf einmal ausgeführt wird. Dies wäre auch auf einem SIMD-Rechner wie der MasPar der Fall. Dennoch ist er aussagekräftiger als z.B. ein herkömmlicher Keller von Funktionsaufrufen, da er zusätzliche Information auf Ebene der Sprachkonstrukte beinhaltet.

Der aktuelle Strukturbaum kann per Knopfdruck im Debugger in einem eigenen Fenster betrachtet werden. Er wird bei jedem Einzelschritt durch das Programm[2] sowie bei jedem

[1] Diese Bedingung ist nötig, da sonst die Blätter des Baumes nicht mehr alleiniger Ort der Aktivitäten wären: Ein Knoten ohne Kinder könnte mit einem ihm entsprechenden Knoten kombiniert werden, der aber Kinder hat. Dadurch wäre der erste Knoten nach der Zusammenfassung Vater eines Kind-Knotens, obwohl er selbst noch Aktivitäten enthält.

[2] Damit ist hier ein „step"-Befehl gemeint, also ein Einzelschritt im Programm in angehaltenem Zustand. Ein Neuzeichnen bei jeder Änderung im voll laufenden Programm wäre zu zeitaufwendig und von der nötigen Kommunikation zwischen Programm und Debugger her unhandlich.

Erreichen eines Breakpoints automatisch auf den neuesten Stand gebracht. Wird er nicht mehr benötigt, kann die Darstellung des Strukturbaums wieder verborgen werden.

Ein Beispiel für eine grafische Darstellung eines Strukturbaums wird in Abbildung 3.3 gegeben.

Der Strukturbaum wird im Adreßraum des untersuchten Programms dynamisch aufgebaut, muß also zur Anzeige durch den Debugger erst in dessen Adreßraum übertragen werden. Dazu gibt es zum Programm dazugebundene Funktionen, die vom Debugger aufgerufen werden und den Strukturbaum in eine Zeichenkette umwandeln. Diese wird an den Debugger ausgegeben und dort wieder in einen Baum transformiert. Ein direkter Transfer des Baumes am Stück ist praktisch nicht möglich, da es sich um eine verzeigerte Struktur handelt, die dynamisch Speicher anfordert und freigibt und sich daher nicht an einer zusammenhängenden Stelle des Speichers befindet. Implementierungen des Baums durch Tabellen würden zwar eine in einem Block übertragbare Struktur darstellen, wären jedoch zu unflexibel in der Anwendung.

Eine Erkennung von rekursiven Aufrufen und deren verkürzende Darstellung, wie im Entwurf erwähnt, wurde nicht implementiert, da bei den untersuchten Programmen dieses Problem nicht auftrat. Sie kann bei Bedarf aber transparent in die Routine zur Anzeige des Baums eingebaut werden.

3.2.3 Profiling-Information

Die Zahl der Durchläufe durch die Konstrukte des Programms muß geeignet angezeigt werden (eine genauere Beschreibung, wie sie genau gewonnen wird, findet sich später im Abschnitt 3.4.1 bei der Gewinnung der Debug-Informationen).

Die Darstellung der Statistiken erfolgt entweder durch ein sich bei Bedarf öffnendes Fenster mit reinem Text, das die Informationen aus der Profile-Tabelle wiedergibt (siehe Abbildung 3.5), oder durch eine grafische Aufbereitung der Durchlaufzahlen in einem Histogramm, wobei nach Zahl der Durchläufe durch ein Konstrukt sortiert wird. Dadurch kommen die größten Verbraucher sofort ins Blickfeld des Benutzers (Abbildung 3.6). Der zugehörige Programmausschnitt ist in Abbildung 3.4 aufgeführt.

Wie beim Strukturbaum werden die Darstellungen bei jedem Schritt im Programm auf den neuesten Stand gebracht.

Verteilte Profiling-Informationen werden zusammengefaßt und in der Summe ihrer Durchlaufszähler angezeigt.

Eine genauere Aufgliederung des Profiles nach Zeilen oder Anweisungen wäre wünschenswert, aber deutlich aufwendiger zu implementieren. Die Durchlaufzahlen der Konstrukte bieten eine gute Näherung für die Schwerpunkte im Programm und fallen beim Aufbau der Struktur-Information praktisch kostenlos an.

3.2.4 Aktivierung der Prozesse

Wie im Abschnitt 2.2.4 vorgeschlagen wurde, sind die Aktivierungen eines `FORALL`s als Sonderfall der Visualisierung eines mehrdimensionalen Feldes implementiert. Dabei wird ein eindimensionales `FORALL` auf ein möglichst quadratisches zweidimensionales Feld von Aktivierungen abgebildet, um die Darstellung kompakt und übersichtlich zu gestalten. Ein Beispiel findet sich in Abbildung 3.7.

Die Aktivierungen der virtuellen PEs werden dabei von den realen PEs erfragt, d.h. im Falle des Netzwerksystems von jedem PE-Prozeß auf allen verwendeten Maschinen, und bei einem einzelnen Rechner direkt vom einzigen Prozeß. Danach kombiniert der Debugger sie

```
FORALL a:[0..resquare-1] IN PARALLEL     (* Zeile 33 *)
VAR ac,bc,at,bt,az,bz,t : LONGREAL;
    count               : INTEGER;
BEGIN
  ac    := xStart + LONGREAL(a MOD resol)*gap;
  bc    := yStart + LONGREAL(a DIV resol)*gap;
  at    := 0.0;      bt    := 0.0;
  az    := 0.0;      bz    := 0.0;
  count := 0;

  WHILE (count < iter) AND (at+bt < 4.0) DO     (* Zeile 45 *)
    t     := az*bz;
    bz    := t + t + bc;     az := at - bt + ac;
    at    := az * az;        bt := bz * bz;
    count := count + 1;
  END;
  erg[a]  := count;

END;
```

Abbildung 3.4: Programmausschnitt für die Profiles

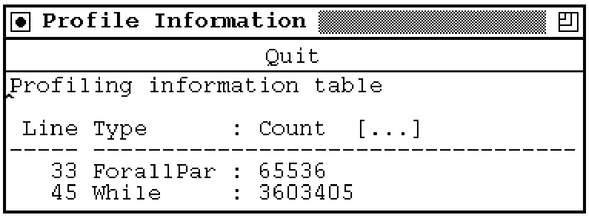


Abbildung 3.5: Beispiel einer textuellen Profile-Tabelle

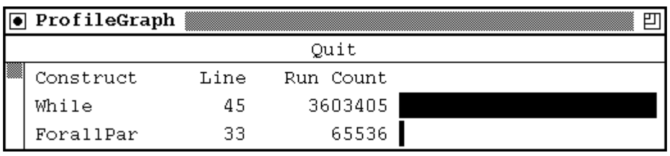


Abbildung 3.6: Beispiel eines sortierten Profile-Histogramms

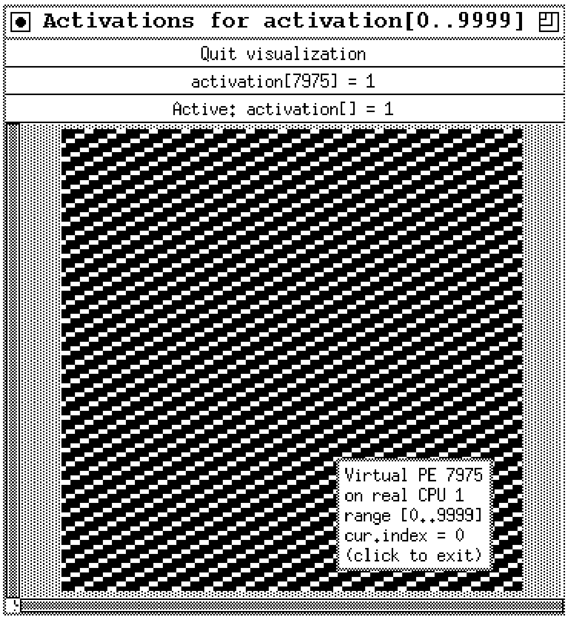


Abbildung 3.7: Beispiel für die Visualisierung der aktiven Prozesse in einem `FORALL`: Die momentan aktiven Prozesse sind als schwarze, die inaktiven als weiße Punkte dargestellt. Es läßt sich leicht ein Muster in der Verteilung erkennen. Die eindimensionalen Aktivierungswerte des `FORALL`s sind auf eine nahezu quadratische Fläche abgebildet, die einen leichten Überblick ermöglicht. Per Mausklick wurde ein Fenster mit genaueren Informationen über einen bestimmten Aktivierungswert geöffnet.

miteinander, um ein Gesamtbild zu erhalten. Auf der MasPar könnte man die Aktivierungen jedes realen PEs verwenden und zu den virtuellen Aktivierungen in Verbindung setzen.

Die Aktivierungswerte für ein synchrones `FORALL` sind im vom Übersetzer erzeugten C-Programm explizit vorhanden, können also direkt dargestellt werden. Schwieriger ist es hingegen bei asynchronen `FORALL`s, da diese lediglich auf Virtualisierungsschleifen ohne eigene Aktivierungsinformationen abgebildet werden.

Hier hilft eine Betrachtung, in welcher Anordnung der Modula-2*-Übersetzer die Virtualisierungsschleifen realisiert: Auf jedem realen PE sind alle Prozesse, deren Schleifenindex unterhalb des momentan abgearbeiteten Indizes liegt, bereits beendet und damit inaktiv, während der momentan laufende und alle ihm folgenden Indizes erst teilweise bzw. noch nicht bearbeitet sind und folglich als aktiv angesehen werden können. Die Visualisierung eines parallelen `FORALL`s besteht also in einer Reihe von inaktiv/aktiv-Punkten, wobei für jedes reale PE ein inaktiver Bereich von einem aktiven gefolgt wird.

Geschachtelte `FORALL`s werden, wie im Entwurf besprochen, als Schnitte durch n-dimensionale Quader dargestellt, wobei jede Dimension des Quaders auf den maximal vorkommenden Bereich der jeweiligen `FORALL`-Variablen mit Leerräumen aufgefüllt wird.

Diese Darstellung kann im Debugger jederzeit aufgerufen werden und bleibt bestehen, bis sich der Kontext im Programm ändert oder der Benutzer sie beendet. Sie wird bei jedem Schritt durchs Programm und bei Erreichen eines Breakpoints aufgefrischt.

3.2.5 Semantik im Parallelen

Als Semantik für Breakpoints und Einzelschritte durch das Programm wurde der globale Stop gewählt, wie im Entwurf beschrieben. Realisiert wird dieses Anhalten wie folgt:

Einzelner Rechner: Der Breakpoint in Modula-2* wird auf einen normalen Breakpoint in der entsprechenden Zeile des C-Programms abgebildet, das der Modula-2*-Übersetzer erzeugt. Sobald die Ausführung diesen Breakpoint erreicht, hält das Programm an, und der Debugger kann aus den vom `dbx` gelieferten Informationen über die Position im C-Code auf die entsprechende Zeile im Modula-2*-Programm schließen.

Ein Einzelschritt läuft ähnlich ab. Er wird auf eine Folge von Einzelschritten im C-Programm abgebildet, die die entsprechende Modula-2*-Zeile verwirklichen. Der Benutzer sieht nur einen einzigen Schritt im Modula-2*-Quellcode. Wenn das optionale C-Fenster aktiviert ist, lassen sich dort die Unterschritte mitverfolgen.

Ein spontanes Anhalten durch den Benutzer wird durch Stoppen des untersuchten Prozesses mittels eines Signals erreicht. In diesem Fall muß erkannt werden, ob gerade ein synchrones `FORALL` abgearbeitet wird, und ggf. die entsprechende Virtualisierungsschleife fertig ausgeführt werden, um eine konsistente Sicht zu gewährleisten. Die Erkennung geschieht durch Betrachten des momentanen Kontexts, der im Strukturbaum sichtbar ist.

Verteiltes Laufzeitsystem: Der Breakpoint wird auf allen PEs des Laufzeitsystems analog zum Fall des Einzelrechners gesetzt. Läuft einer der verteilten Prozesse auf einen Breakpoint, so erfährt das sein zugehöriger Debugger-Prozeß und schickt eine Nachricht an den zentralen Debugger[3]. Dieser fordert alle anderen Debugger-Prozesse auf, ihr Programm anzuhalten. Dadurch wird der globale Stop erreicht.

Der Einzelschritt ist auf jedem PE wie beim Einzelrechner realisiert und wird vom globalen Debugger angestoßen. Die Debugger auf den PEs melden nach ausgeführten Einzelschritten die neuen Positionen im Programm.

Der spontane Stop ist durch Verschicken einer Halte-Aufforderung an alle Debugger-Prozesse verwirklicht. Auch hier ist auf einen eventuellen synchronen Kontext zu achten.

3.3 Repräsentation großer Datenmengen

3.3.1 Anforderungen

Entsprechend den Anforderungen wurden im Rahmen der Diplomarbeit Module zur Visualisierung mehrdimensionaler Daten geschaffen.

An dieser Stelle sei nochmals auf den Anhang A verwiesen, in dem die Implementierung der Visualisierungen anhand von Kriterien für eine wirkungsvolle Visualisierung [Mill92] beurteilt wird.

3.3.2 Visualisierung mehrdimensionaler Felder

Den Überlegungen des Entwurfs folgend wurden Visualisierungen für verschiedene Anwendungen implementiert. Sie alle ermöglichen die Darstellung mehrdimensionaler Felder durch zweidimensionale Schnitte. Eindimensionale Daten werden der besseren Übersicht wegen als ein möglichst quadratisches zweidimensionales Feld dargestellt.

Eine Visualisierung gliedert sich in Unterbereiche:

- Durch eine Betätigung des Schließfelds kann die Visualisierung verlassen werden.

[3] Für die genaue Beschreibung der Prozesse im Laufzeitsystem sei auf den Anhang B und den Abschnitt 3.4.3 verwiesen.

- Der Wert des Feldelements, das sich unter dem Mauszeiger befindet, wird exakt angezeigt.
- Abhängig von der verwendeten Visualisierung werden weiterhin Benutzereingaben angenommen oder Zusatzinformationen dargestellt.
- Eine Reihe von Schiebebalken (einer pro Dimension des Feldes) erlaubt die schnelle Einstellung der Position in der jeweiligen Dimension. Zusätzlich lassen sich mit Knöpfen die Werte genau einstellen oder die gewünschte Zahl textuell eingeben.
- Das eigentliche Visualisierungsfenster stellt den zweidimensionalen Schnitt durch das Feld dar, wobei die Feldelemente durch schwarze, weiße oder graustufige Punkte verschiedener Größe dargestellt sind. Die Darstellung kann mit den Maustasten vergrößert und verkleinert werden, wobei das Element an der aktuelle Mausposition in die Mitte des Fensters rückt. Bei Feldern, die zu groß für das Fenster sind, kann der sichtbare Ausschnitt mit Schiebebalken verschoben werden.

Vergleiche: Alle Datenelemente werden als schwarze oder weiße Punkte dargestellt, je nachdem ob der Vergleich mit einem bestimmten Wert wahr oder falsch ist. Der Benutzer kann den Wert interaktiv verändern und den Vergleich (größer, gleich, kleiner) durch Anklicken wählen. Von der Bedienung her ist diese Visualisierung die einfachste, ein Beispiel findet sich in Abbildung 3.8.

Bereiche: Abbildung 3.9 zeigt eine Visualisierung, die einen benutzerdefinierten Wertebereich als Kriterium zur Färbung der Werte besitzt. Alle Elemente des Feldes, die innerhalb des Bereichs liegen, sind schwarz dargestellt, die anderen weiß. Diese Darstellung ist eine Erweiterung des Vergleichs. Für eine Bereichsgrenze von $\pm\infty$ wird sie zum größer/kleiner-Vergleich, bei Identität der beiden Grenzen ist sie ein Test auf Gleichheit. Der Wert „unendlich“ wird als inf (oder leere Eingabe) angegeben.

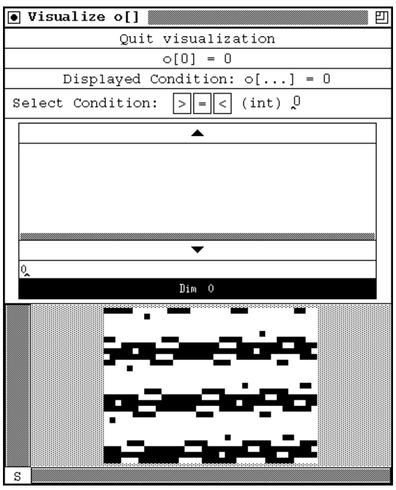


Abbildung 3.8: Visualisierer mit einem 1-dimensionalen Feld – Vergleich mit benutzerdefiniertem Wert als Kriterium

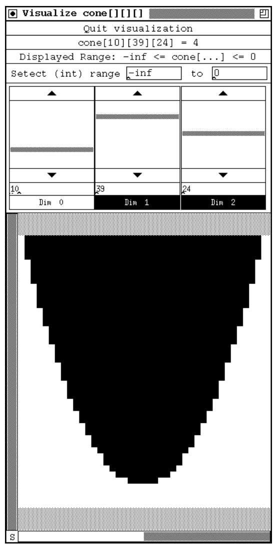


Abbildung 3.9: Visualisierer mit 3-dimensionalem Feld – interaktiv änderbarer Bereich als Kriterium

Werte: Diese Visualisierung errechnet Mittelwert und Standardabweichung der Feldelemente und bildet sie danach auf eine von fünf Graustufen ab. Statistische Ausreißer werden ignoriert, um die Abbildung nicht zu verzerren. Der Benutzer kann hier keine Eingaben tätigen, dafür wird die Statistik der Feldelemente angezeigt. Abbildung 3.10 zeigt eine solche Visualisierung.

Benutzerdefinierte Funktionen: Für den Programmierer, der Visualisierungen anwenden will, ist diese Variante von Interesse. Die Kriterien werden durch eine benutzerdefinierte Funktion definiert, die zur Auswertung der Feldelemente verwendet wird. Sie kann während der Visualisierung geändert werden. Ein Beispiel dafür steht in Abbildung 3.11.

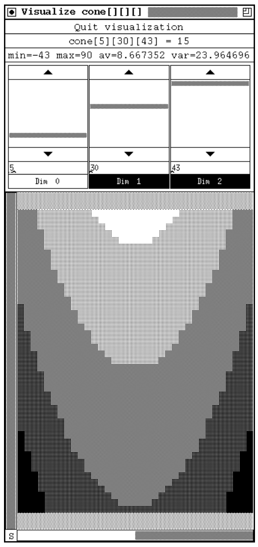


Abbildung 3.10: Visualisierer mit 3-dimensionalem Feld aus Fließkommazahlen – Darstellung mittels Grauwerten, Statistikzeile statt Kriterieneingabe

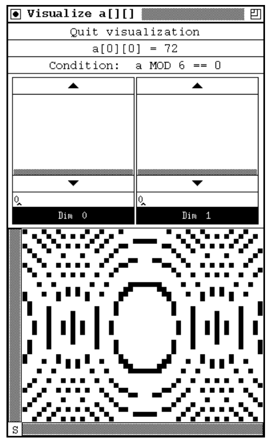


Abbildung 3.11: Visualisierer mit 2-dimensionalem Feld – Kriterium durch benutzerdefinierte Funktion bestimmt

Aktivierungen: Ein Sonderfall der Visualisierung ist die Darstellung von Aktivitäten der virtuellen Prozessoren, wie sie in Abbildung 3.12 zu sehen ist. Hier ist das Kriterium zur Anzeige der aktuelle Aktivierungswert – alle Elemente, die ihm entsprechen, sind schwarz (also aktiv), während der Rest weiß (inaktiv) ist. Mit einem Klick auf einen Punkt kann Information über das entsprechende virtuelle PE abgerufen werden.

Die Darstellung erfolgt dabei bei mittleren Feldgrößen so schnell, daß der Benutzer interaktiv durch das Feld „blättern" kann.

Eine ausführliche Beschreibung der Verwendungsarten findet sich in der Definitionsdatei der Visualisierungen, die als Abschnitt D.3 im Modulführer aufgeführt ist.

3.4 Realisierung der Implementierungsentscheidungen

In den folgenden Abschnitten wird beschrieben, wie die im Entwurf gefällten Implementierungsentscheidungen realisiert wurden. Dabei werden auch die zugrundeliegenden Mechanismen und Datenstrukturen erläutert.

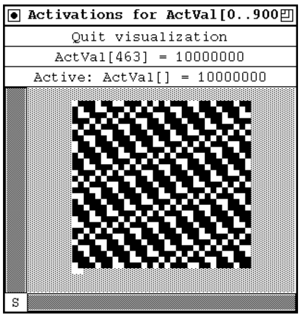


Abbildung 3.12: Visualisierer für ein einzelnes `FORALL` – keine Eingabemöglichkeit, aber ereignisgesteuerte Funktionsaufrufe

3.4.1 Gewinnung der Debug-Information

Die im Entwurf erarbeiteten Überlegungen wurden durch die im Abschnitt 3.5 genauer beschriebenen Debug-Funktionen realisiert.

Das zu untersuchende Programm wird dabei durch den Übersetzer mit Debug-Funktionen instrumentiert, die während des Programmablaufs dynamisch Informationen sammeln. Es gibt zwei Strukturen zur Aufnahme von Informationen:

Der Strukturbaum, der bei Eintritt und Verlassen von Sprachkonstrukten auf- und abgebaut wird, und dessen Knoten Informationen enthalten wie Art des Konstrukts, Kontext des Aufrufs (sequentiell, synchron oder asynchron), Zeilenbereich des Konstrukts, Zahl der Durchläufe sowie Verweise auf Vorgänger und Nachbarknoten.

Ein Knoten wird beim Eintritt in ein Konstrukt neu angelegt und erbt Informationen von seinem dynamischen Vorgänger (z.B. den Kontext, wenn der Knoten selbst nicht ein `FORALL` beschreibt und somit den Kontext ändert).

Bei jedem Durchlauf wird sein Zähler erhöht, wobei der Zusatzaufwand durch einen direkten Zeiger auf den aktuellen Knoten und seinen Zähler gering gehalten wird (für jedes reale PE ist der Strukturbaum eine Liste; nur durch Zusammenfassen der Bäume aller PEs wird er zu einem echten Baum. Daher kann auf jedem PE das jeweils aktuelle Blatt des „Baumes“ durch einen einzigen Zeiger referenziert werden).

Beim Austritt aus dem Konstrukt wird der Knoten wieder entfernt und sein Vorgänger als aktiver Knoten vermerkt.

Der Strukturbaum beinhaltet also die dynamische Information über den Ablauf des Programms.

Die Profile-Information, die für schnelle Zugriffe in einer Hash-Tabelle gehalten wird. Sie zeichnet die Durchläufe durch ein Konstrukt permanent auf, im Gegensatz zu einem Knoten des Strukturbaums, der nach Verlassen des Konstrukts gelöscht wird.

Erreicht wird dies durch Addieren der Durchlaufzähler des Knotens zu dem bereits in der Profile-Tabelle enthaltenen Wert[4]. Ist für das entsprechende Konstrukt noch

[4]Real implementiert ist dieser Punkt durch einen expliziten Verweis vom Knoten im Strukturbaum auf seinen Eintrag in der Profile-Tabelle, dessen Zähler bei jedem Durchlauf direkt erhöht wird. Ansonsten wäre die in der Tabelle enthaltene Information über Durchläufe immer nur aktuell, wenn das entsprechende Konstrukt gerade nicht aktiv ist und damit sein Zähler beim letzten Beenden auf den neuesten Stand gebracht wurde. Der Aufwand für das zusätzliche Inkrementieren eines Zählers ist minimal.

keine Information vorhanden, so wird ein neuer Profile-Eintrag erzeugt, der ab dann alle Durchläufe für dieses Konstrukt speichert.

Außer einem Zähler enthält die Profile-Tabelle noch die Art des Konstrukts, seinen Kontext und Zeileninformation, die sie vom Strukturbaum übernimmt.

Abbildung 3.13 gibt ein Beispiel für das Zusammenwirken dieser beiden Datenstrukturen.

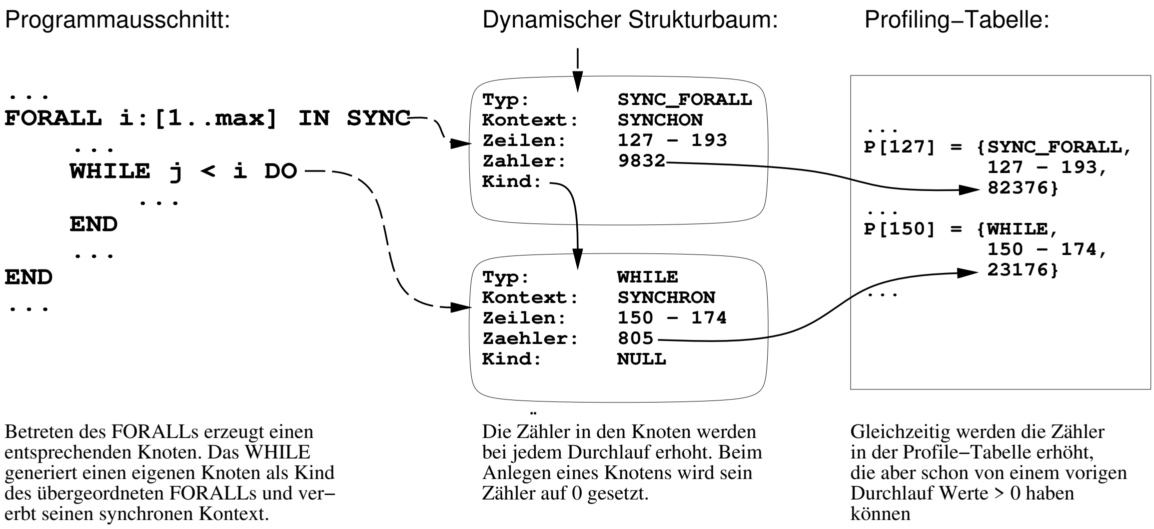


Abbildung 3.13: Zusammenwirken von Strukturbaum und Profiling-Tabelle während des Programmablaufs.

Die Debug-Information wird also durch das laufende Progamm selbst mit aufgebaut. Nachteil dieses Vorgehens ist, daß die Information bei einem schweren Programmabsturz selbst in Mitleidenschaft gezogen werden kann. Die Alternative, nämlich bei jedem Vorgang im Strukturbaum oder der Profiling-Tabelle die entsprechende Information direkt an den Debugger weiterzugeben, scheidet aus Geschwindigkeitsgründen aus. Ansonsten müßte jedesmal ein Prozeßwechsel durchgeführt werden und das untersuchte Programm bräuchte zusätzliche Wege, um spontan Informationen an den Debugger zu übermitteln.

3.4.2 Abbildung zwischen Modula-2* und C

Wie bereits erwähnt, ist die Abbildung zwischen dem ursprünglichen Quellcode eines Modula-2*-Programms und dem erzeugten Code im C-Dialekt der Zielmaschine nicht trivial. Ein beispielhafter Codeausschnitt in Abbildung 3.14 möge dies verdeutlichen.

Zur kurzen Erklärung: Am Anfang des erzeugten C-Codes werden Variablen deklariert, die für die Virtualisierungsschleife nötig sind, wie etwa die obere und die untere Grenze der Schleifenvariablen (`Low_i, Hi_i` etc.). Außerdem finden sich dort die Aktivierungswerte der virtuellen Prozesse (`AB_i`) und Hilfsvariablen für die Erhaltung der Semantik im Synchronen (`H_1_i`). Danach werden die Grenzen dynamisch berechnet (`_InitBounds_`) und alle Aktivierungswerte initialisiert. Es schließt sich eine Virtualisierungsschleife mit Zuweisung des Feldwertes an die Hilfsvariable an, gefolgt von der Zuweisung der Hilfsvariable an das Ziel-Feldelement. All diese Schritte werden durch Synchronisationsanweisungen `_Sync_()` abgesichert.

Um die Abbildung zwischen diesem C-Code und dem ursprünglichen Modula-2*-Programm zu ermöglichen, fügt der Übersetzer zusätzliche Informationen in Form von

```
...
VAR v: ARRAY [0..1000] OF
             LONGINT;
...
FORALL i:[0..999] IN SYNC
    v[i] := v[i+1];
END;
...
```

⟺

```
...
static struct sampleS__1 {
    LONGINT A[999 + 1];
} v;

 ...
{ /* FORALL IN SYNC */
  plural LONGCARD Low_i[1000];
  plural LONGCARD Hi_i[1000];
  plural LONGINT Num_i;
  plural LONGINT AB_i[1000];
  regvar plural LONGCARD i;
  regvar LONGCARD IR_i;
  plural BOOLEAN PEA_i;
  plural LONGINT GrC_i;
  regvar plural LONGCARD FLow_i;
  regvar plural LONGCARD FHi_i;
  plural LONGCARD FNum_i;
  plural BOOLEAN FPEA_i;
  plural LONGINT FGrC_i;
  {/* SyncBlock 2*/
        plural LONGINT H_1_i[1000];
    FLow_i = 0;
    FHi_i = 999;
    _InitBounds_(FHi_i,FLow_i,FNum_i);
    IR_i=0;
    IR_i=0;
    for(i=FLow_i;i<=FHi_i;i++,IR_i++){
      Num_i = 0;
      AB_i[IR_i] = 0;
    }
    _Sync_(0);

    IR_i=0;
    for(i=FLow_i;i<=FHi_i;i++,IR_i++){
      H_1_i[IR_i] = v.A[i + 1];
    }
    _Sync_(0);

    IR_i=0;
    for(i=FLow_i;i<=FHi_i;i++,IR_i++){
      v.A[i] = H_1_i[IR_i];
    }
    _Sync_(0);
  }
}
...
```

Abbildung 3.14: Beispiel eines Modula-2*-Quellcodes und des daraus erzeugten C-Codes (für einzelnen Rechner) besonderen Kommentarzeilen in den Code ein. Beim Einlesen des Quelltextes durch den Debugger werden diese zerteilt und, wie im folgenden beschrieben, aufgearbeitet.

Zeilenabbildung: Für jedes erzeugte C-Konstrukt wird die Modula-2*-Zeile angegeben, aus der das Konstrukt bei der Übersetzung entstanden ist.

Aus diesen Informationen erzeugt sich der Debugger eine Tabelle mit Abbildungen zwischen C- und Modula-2*-Zeilen. Diese Abbildung ist nicht eindeutig, da aus einer Modula-Zeile mehrere C-Zeilen entstehen können (wie das Beispiel eindrucksvoll zeigt). Zudem werden die Zeilen durch Restrukturierungen des Übersetzers in der Reihenfolge vertauscht – so wird z.B. das `ELSE` im synchronen `FORALL` vor dem `IF` ausgeführt.

Es sind also zusätzliche Maßnahmen nötig, um die Abbildung zu glätten, damit der Programmzähler nicht scheinbar unmotiviert im Modula-2*-Quellcode hin– und herspringt. Dies tritt besonders im synchronen `FORALL` mit seinen aufwendigen Virtualisierungsschleifen auf.

Folgende Vorgehensweisen und Heuristiken werden dabei eingesetzt:

- Der C-Quellcode wird sequentiell durchsucht. Wenn eine Zeileninformation gefunden wird, d.h. die Modula-2*-Zeile bekannt wird, aus der die entsprechende C-Zeile entstanden ist, dann werden die C-Zeile und ihre Nachfolger der Modula-2*-Zeile zugeordnet, bis man auf eine neue Zeileninformation trifft. Der Modula-2*-Zeile wird die erste C-Zeile zugeordnet, in der die sie betreffende Information stand.
- Zeileninformationen im C-Code, die auf ein `FORALL ... IN SYNC` verweisen, werden bei der Abbildung ignoriert. Sie treten an verschiedensten Stellen mitten im C-Code des `FORALL`s auf, z.B. immer, wenn Aktivierungswerte initialisiert werden. Dies führt sonst zu irritierenden Sprüngen im Modula-2*-Quellcode.
- Durch die zusätzlich im C-Code vorhandenen Debug-Funktionen ist eine Plausibilitätsprüfung möglich: Wenn ein `DebugFnStart()` angetroffen wird, dann wird die aktuelle C-Zeile der in der Debug-Funktion aufgeführten Modula-2*-Zeile zugeordnet. Eine eventuell schon vorhandene Abbildung von Modula-2* nach C wird dabei überschrieben, da sie sich auf eine Zeile vor der momentanen Zeile bezogen haben muß, also verfrüht war.

  Zusätzlich wird anhand der in den Debug-Funktionen enthaltenen Typen der Konstrukte der gerade aktive Typ festgehalten, der weitere Konsistenzüberprüfungen erlaubt.
- Die endgültige Abbildung von Modula-2* nach C wird erst nach Glätten der C–nach–Modula-2* Abbildung durchgeführt, und zwar durch Übernehmen der C-Zeile, in der die erste Referenz auf die gerade untersuchte Modula-2*-Zeile stattfindet.

Variablenabbildung: Jede C-Variable, die aus einer Modula-2*-Variablen entstanden ist, wird mit ihrem Modula-2*-Namen und ihrem Typ in Modula-2* angegeben.

Dadurch kann der Debugger von einem im Modula-2*-Quellcode ausgewählten Variablennamen auf den entsprechenden Namen in C schließen und sich dessen Typ durch eine Anfrage an den unterliegenden Debugger (`dbx` oder `gdb`) ausgeben lassen.

Eine Besonderheit bilden Felder, die in C in `struct`s eingeschlossen werden, um direkte Zuweisung zu ermöglichen – ein Beispiel ist das Feld `v` in Abbildung 3.14. Um sie zu bearbeiten, z.B. zu visualisieren, sieht der Debugger in den `struct` hinein und stellt die Anzahl und Größe der Dimensionen des Felds fest. Dies geschieht durch Zerteilung der Information, die der `dbx` auf die Anfrage nach dem Typ der entsprechenden Variablen liefert.

Durch diese Maßnahmen läßt sich der Programmablauf vollständig in Ausdrücken der Modula-2*-Ebene untersuchen.

3.4.3 Aufsetzen auf einen vorhandenen Debugger

Einzelner Rechner

Grundlage des Debuggers für einen einzelnen Rechner ist der im Rahmen meiner Studienarbeit [Haen91] modifizierte `xdbx` [Cheu89]. Dieses Programm bildet eine grafische Benutzeroberfläche für den `dbx`-Debugger [Sun90c], der auf der entsprechenden Sun- oder DEC-Workstation läuft.

Der `xdbx` (bzw. MSDB) bildet also eine Schicht zwischen dem Benutzer und dem eigentlichen Debugger, wie in Bild 3.15 gezeigt ist. `xdbx` kontrolliert mittels eines Pseudo-Terminals (`pty`) den `dbx`, indem er dessen Standardeingabe mit Befehlen versorgt, die aus Aktionen des Benutzers erzeugt werden. Umgekehrt empfängt er vom `dbx` alle Statusmeldungen und Ausgaben, um sie aufzubereiten und dann dem Benutzer darzustellen.

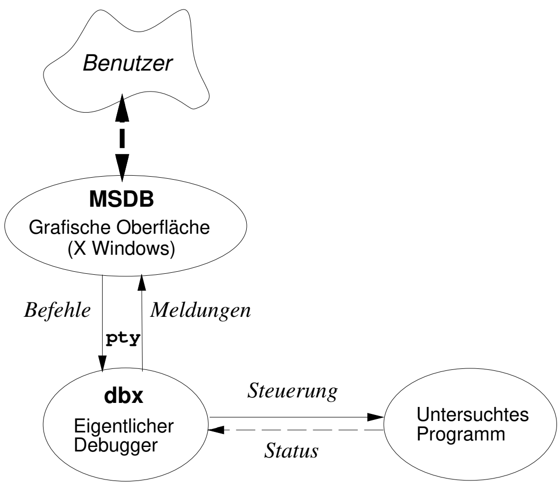


Abbildung 3.15: Interaktion von Benutzer, MSDB, `dbx` und den zu untersuchenden Programm (einzelner Rechner).

Das Absenden von Befehlen an den Debugger erfolgt durch ein `fputs()` auf den File Descriptor des `pty`s, das Lesen von Rückmeldungen analog über ein `fgets()`. Mit der Empfangsseite des `pty`s gekoppelt ist eine ereignisgesteuerte X-Windows-Routine, die bei der Ankunft von Daten am `pty` die Kontrolle übernimmt, die Daten zerlegt, und eine passende Funktion zu ihrer Behandlung aufruft. Dies kann z.B. bei Auftreffen auf einen Breakpoint eine Funktion sein, die die entsprechende Quellcode-Datei lädt und den Programmzeiger auf die Zeile setzt, an der der Breakpoint auftrat. Zusätzlich besteht noch eine Möglichkeit, direkt Ergebnisse vom `dbx` zu erhalten, ohne sie durch die allgemeine Zerteilung zu lassen.

Alle Prozesse, d.h. das Programm selbst, der Debugger und die Benutzeroberfläche laufen dabei in getrennten Adreßräumen ab. Kommunikation findet nur über die Pipes bzw. `pty`s statt.

Zur Übermittlung größerer Datenmengen, z.B. für die Visualisierung eines Feldes, ist der Umweg über eine Darstellung der Daten durch den `dbx` als Text, Übermittlung durch das `pty` und anschließende Rückwandlung von Text in Binärdaten zu aufwendig. Daher wurde ein zusätzlicher Mechanismus vorgesehen. Er basiert darauf, daß im zu untersuchenden Programm zusätzliche Routinen eingebunden werden (dies geschieht schon beim Einfügen der DebugFunktionen, stellt also kein Problem dar). Diese Routinen können dann durch den Debugger explizit aufgerufen werden, wenn das Progamm gestoppt ist, und liefern die gewünschte Information. Es gibt zwei unterschiedliche Implementierungen:

Kernaufruf : Die Routine im untersuchten Programm erhält als Argumente die Adresse der gewünschten Daten, ihre Länge, sowie die Adresse eines Puffers im Debugger und die Prozess-ID des Debuggers. Durch einen `ptrace()`-Systemaufruf hält sie den Debugger an, überträgt die Daten im direkten Zugriff auf seinen Speicher und läßt ihn fortfahren.

Dies entspricht einer kurzen Umkehr der Rollen – der Debugger wird zum Ziel einer Datenmanipulation, die er allerdings selbst in Auftrag gegeben hat. Durch den direkten Weg durch den Betriebssystemkern ist diese Art von Übertragungen recht schnell. Ein Schema dieser Vorgänge findet sich in Abbildung 3.16.

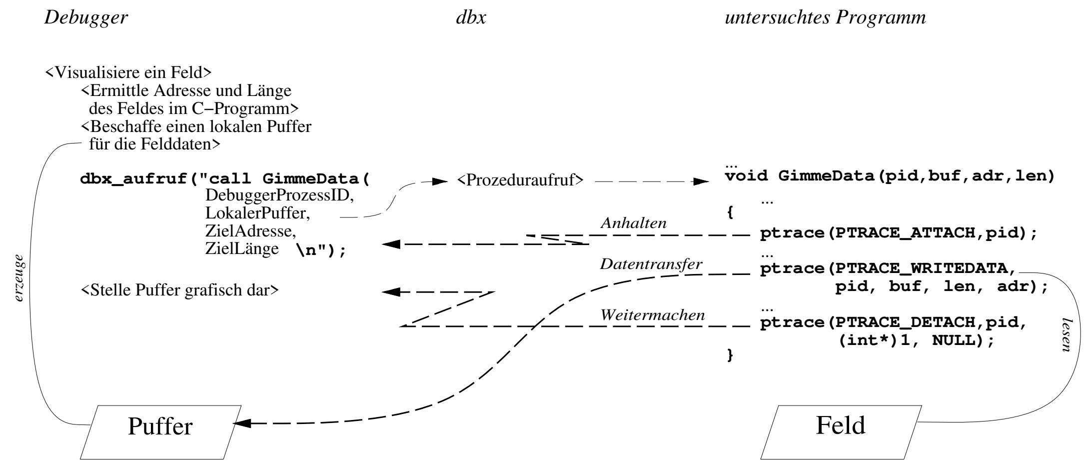


Abbildung 3.16: Ablauf eines Datentransfers mittels `ptrace()` durch eine spezielle Routine im untersuchten Programm.

Dateien : Falls die erste Methode nicht funktioniert (z.B., weil der Debugger selbst mit einem Debugger untersucht wird, wodurch sich kein `ptrace()` mehr durchführen läßt[5]), so wird der Umweg über das Dateisystem gewählt:

Die Routine erhält wiederum Adresse und Länge der zu übertragenden Daten und zusätzlich einen Dateinamen. Sie schreibt die Daten in die angegebene Datei. Der Debugger liest die Datei aus und löscht sie wieder. Natürlich ist dieser Weg zeitaufwendiger als der direkte Zugriff über den Kern, aber immer noch schnell genug für ein interaktives Arbeiten.

[5] Die Debugger unter UNIX basieren alle grundsätzlich auf diesem Systemaufruf, der nur eine Ankoppelung an einen Prozeß auf einmal erlaubt.

LAN Laufzeitsystem

Zusätzlich zu den in Anhang B genauer beschriebenen Komponenten des Laufzeitsystems wie Runner (Speicherzugriffe), Worker (Berechnung) und Syncer (Synchronisation und Start/Terminierung) kommt bei der Fehlersuche noch für jedes PE ein Control-Prozeß hinzu, der den Debugger beinhaltet. Aus technischen Gründen ist dies der `gdb` und nicht der `dbx`, was aber – von einer etwas anderen Befehlssyntax abgesehen – keinen Unterschied macht[6]. Außerdem gibt es einen zentralen Master Debugger, der die Benutzerinteraktion übernimmt und die Control-Prozesse steuert.

Der Aufbau eines Control-Prozesses ist analog zu dem für den einzelnen Rechner geschilderten Fall, d.h. der Control-Prozeß startet selbst einen Debugger, den er über ein `pty` steuert. Der Debugger wird an den Worker des PEs angeschlossen und kann ihn danach vollständig kontrollieren. Der Zustand des zugehörigen Workers wird vom Control-Prozeß ständig überwacht und von sich aus gemeldet, wenn er z.B. auf einen Breakpoint läuft oder ein ungeplanter Programmabbruch auftritt.

Die grafische Darstellung und Kommunikation mit dem Benutzer sowie die Zusammenfassung und Auswertung der von den einzelnen PEs gelieferten Debug-Daten leistet der Master Debugger. Er verteilt auch die Befehle an die einzelnen Control-Prozesse.

Einen Überblick über die Prozesse und ihre Interaktion bietet Abbildung 3.17. Dabei wird von der Aufspaltung des Debugger-Prozesses in den Control-Prozeß und den eigentlichen Debugger abstrahiert. Eine genaue Darstellung eines einzelnen PEs mit allen beteiligten Prozessen bietet Bild 3.18.

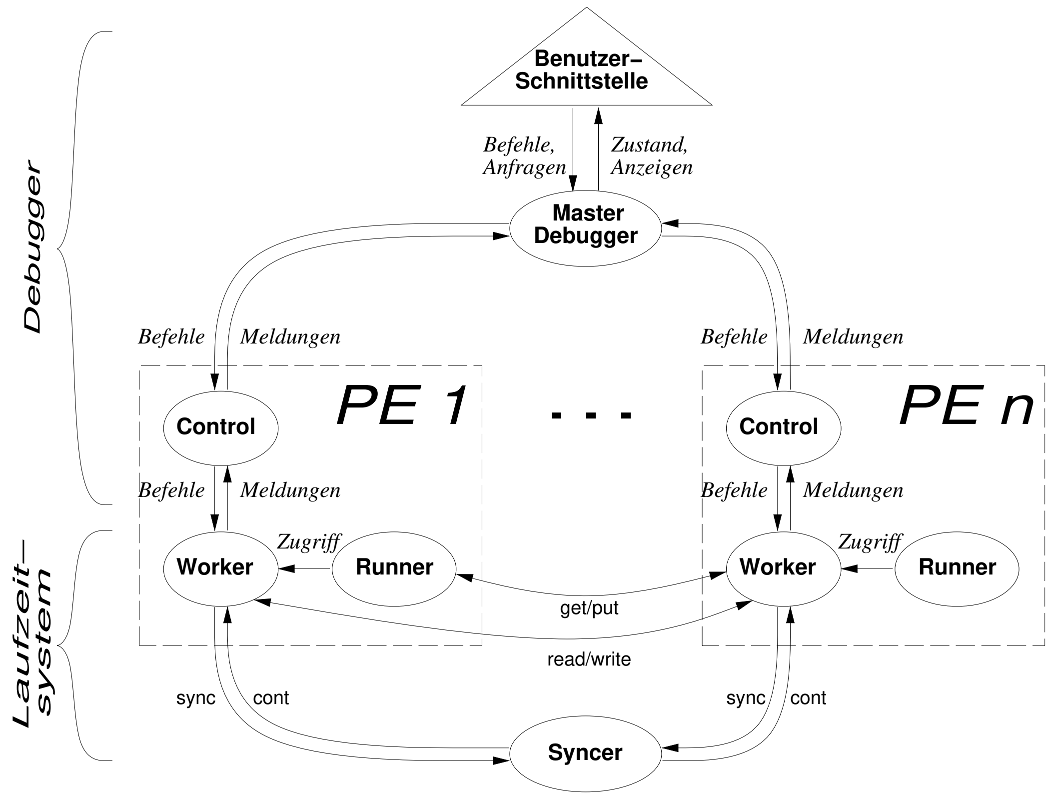


Abbildung 3.17: Prozesse im LAN-Debuggersystem und ihre Interaktion

[6]Im Modulführer wird auf dieses Problem näher eingegangen.

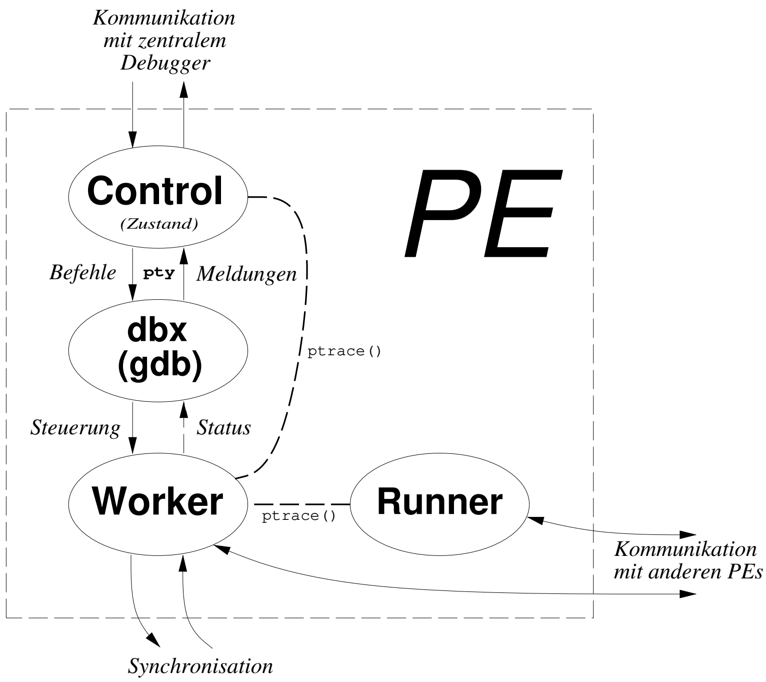


Abbildung 3.18: Prozesse in einem einzelnen PE

Vermeidung von längerem Blockieren

Die Erkennung des Worker-Zustands erfolgt konkret durch Zerteilung der vom Debugger (`gdb`) gelieferten Statusmeldungen. Schlüsselwörter, die etwa einen Breakpoint oder einen Programmabsturz signalisieren, werden ausgewertet und das interne Bild vom Zustand des Workers angepaßt. Wichtig ist, daß der Control-Prozeß nirgends länger blockierend warten darf, da er sonst keine Nachrichten annehmen und z.B. einer Anforderung eines globalen Stops nicht nachkommen könnte. Er sendet also Befehle asynchron an den `gdb` ab und prüft periodisch, ob Rückmeldungen anstehen. Auf Aktionen mit vernachlässigbarem Zeitaufwand, wie dem Stoppen des Programms, kann er auch warten.

Das Problem des langen Blockierens von Nachrichten stellt sich noch verschärfter beim Auftreten von Kommunikationsoperationen im Programm: Ohne besondere Maßnahmen würde der Debugger beim Einzelschritt in eine Lese-Anweisung beliebig lange warten müssen, bis die in der Anweisung erwartete Nachricht eintrifft. Um dies zu verhindern, werden in allen Kommunikationsanweisungen des Programms für den Benutzer unsichtbare Breakpoints gesetzt. Will das Programm nun eine Kommunikation durchführen, so trifft es auf den Breakpoint. Dies merkt der Control-Prozeß und schickt eine entsprechende Nachricht an den Master Debugger, der dadurch ein aktuelle Liste aller anstehenden Lese– und Schreibanweisungen der PEs aufstellen kann. Findet er in dieser Liste eine Übereinstimmung, d.h. PE a will von PE b lesen und PE b will an a schreiben, so gibt er die Kommunikation frei und der Ablauf der beiden PEs geht weiter.

Dieser Ansatz verhindert zuverlässig das Blockieren der Control-Prozesse. Allerdings verlangsamt er den Austausch von Nachrichten im Programm, da jede Nachricht bei Sender und Empfänger zunächst zu einem Breakpoint führt und der Master Debugger die beiden Prozesse explizit zum Weiterlaufen auffordern muß. Im Rahmen einer prototypischen Implementierung ist dieser Mehraufwand aber zu tolerieren. Eine elegantere Lösung wäre z.B. der direkte Anschluß des Debuggers als Unterprogramm, der den Aufruf des `gdb`s über `pty`s ersetzen würde. Dadurch hätte man die volle und direkte Kontrolle über den eigentlichen Debugger, würde aber den Rahmen dieser Diplomarbeit vollends sprengen.

3.5 Debug-Funktionen

Hier werden die oben angesprochenen Debug-Funktionen noch einmal ausführlicher erläutert und ihre Semantik genauer definiert.

3.5.1 Semantik der Debug-Funktionen

Allgemeine Semantik

Jedem Konstrukt in Modula-2*, das für den Programmablauf interessant ist – also nichttrivialen Befehlen wie Kontrollstrukturen oder Funktionsaufrufen – wird eine Debug-Funktion zugeordnet. Diese Funktion wird aufgerufen, wenn das Konstrukt begonnen wird, bei jedem Durchlauf (z.B. in Schleifen), und bei Beendigung des Konstrukts.

Die Debug-Funktionen legen, wenn sie aufgerufen werden, Informationen über den aktuellen Zustand des Programms ab. Dazu gehören die momentane Aufrufstruktur, die bei paralleler Ausführung nicht nur einen Keller sondern einen Baum darstellt, und Profiling-Informationen, d.h. Zähler für die Zahl der Durchläufe durch das jeweilige Konstrukt. Das Zusammenwirken dieser Strukturen wurde in Abschnitt 3.4.1 erläutert.

Im Gegensatz zur Aufrufstruktur, die den dynamischen Ablauf des Programms widerspiegelt, ist die Profiling-Information mit der statischen Struktur, d.h. den Programmzeilen, verbunden. Sie wird anhand der genauen Position im Quellcode abgelegt bzw. auf den neuesten Stand gebracht.

Ein kleines Beispiel in Modula mag diese Konzepte verdeutlichen:

```
     ...
     StartDebugFn(WHILE,Z,...);
Z    WHILE i < 10 DO
         UpdateDebugFn(WHILE,Z,...);
         i := i + 1;
     END;
     EndDebugFn(WHILE,Z,...);
     ...
```

Vor Beginn der `WHILE`-Schleife wird durch *StartDebugFn()* ein neues Blatt des Typs *WHILE* im Strukturbaum erzeugt und initialisiert. Gleichzeitig wird ein Eintrag in die Profiling-Tabelle zur Zeile *Z* mit demselben Typ erzeugt. Wenn solch ein Eintrag schon existiert, dann wird der bereits vorhandene Eintrag übernommen. Das kann etwa vorkommen, wenn das Konstrukt im Programm mehrfach angesprochen wird, weil es in einer Schleife oder in einer Funktion liegt. Jeder Aufruf von *UpdateDebugFn()* in der `WHILE`-Schleife erhöht den Durchlauf-Zähler der Profiling-Information. Schließlich wird nach Ende der Schleife durch *EndDebugFn()* der Knoten im Strukturbaum wieder gelöscht. Sein Profiling-Eintrag bleibt aber erhalten, um bei erneuten Durchläufen weitere Informationen zu sammeln, die dann bei Programmende ausgewertet werden können.

In der Praxis sieht der Aufruf etwas komplizierter aus, weil z.B. innerhalb eines `FORALL`s weitere Informationen über den Ablauf nötig sind. Hierzu gehören etwa die Aktivierungswerte der einzelnen Prozesse oder der innerhalb des Konstrukts gültige Aktivierungswert.

Eine Beschreibung der vollständigen Syntax der Debug-Funktionen findet sich in Abschnitt 3.5.2.

Außer Informationen wie Art des Konstrukts, Aktivierungen und Zeilennummern beinhaltet der Strukturbaum auch den Kontext des jeweiligen Knotens. Dieser kann sequentiell, asynchron parallel oder synchron parallel sein, je nachdem, ob die Programmausführung sich im Moment außerhalb eines `FORALL`s, in einem asynchronen oder in einem synchronen `FORALL` befindet. Der Kontext ändert sich automatisch, wenn ein `FORALL` betreten oder verlassen wird. Am Anfang des Programms ist er sequentiell.

Semantik im Kontext eines `FORALL`s

Bisher wurde hauptsächlich die Wirkung einer Debug-Funktion in sequentiellem Kontext behandelt. Interessant für Modula-2* ist aber vor allem die Semantik innerhalb eines `FORALL`s.

Intuitiv gesehen soll die Debug-Funktion bei jedem Durchlauf eines Konstrukts aufgerufen werden. Da gleichzeitig Profiling-Information gesammelt wird, ist es wünschenswert, wenn die Zahl der Aufrufe mit der Zahl der Aktivitäten übereinstimmt, die das Konstrukt im parallelen Fall durchlaufen. Im Entwurf wurde bereits in Abbildung 2.1 verdeutlicht, wie die synchrone und die asynchrone Parallelität in Modula-2* verwirklicht wird. Daraus ergibt sich folgendes:

Im asynchronen `FORALL ... IN PARALLEL` gilt die geforderte Semantik automatisch, da alle virtuellen Prozesse der Reihe nach in einer Schleife abgearbeitet werden und dementsprechend für jeden dieser Prozesse alle durchlaufenen Debug-Funktionen im `FORALL` aktualisiert werden.

Das synchrone `FORALL ... IN SYNC` ist komplizierter, da es in durch Synchronisationen getrennten Teilabschnitten ausgeführt wird. `UpdateDebugFn()` wird z.B. nicht in der Virtualisierungsschleife aufgerufen, die etwa einen `ELSE`-Zweig realisiert, sondern vor dem Eintritt in sie. Ideal wäre ein Aufruf an einer Stelle, die nur die wirklich aktiven Prozesse durchlaufen, was aber vom Übersetzer her sehr schwer zu realisieren ist. Man könnte durch Abzählen im Feld der Aktivierungswerte die Anzahl der aktiven Prozesse ermitteln, was aber die Ausführung verlangsamen würde.

3.5.2 Syntax der Debug-Funktionen

Eine Debug-Funktion hat das allgemeine Format

`DebugFn`*Art*(*Typ*, *Zeile*, *Spalte*, *BisZeile*, *BisSpalte*,
A-Wert, *A-Feld*, *Bereich*, *Von*, *Bis*)

Die Bedeutung der einzelnen Parameter ist folgende:

Art Die Art der Debug-Funktion, also `Start`, `Update` oder `End`.

Typ Der Typ des Konstrukts, das die Debug-Funktion beschreibt. Beispiele sind `WHILE`, `ELSE` oder `PAR_FORALL`.

Zeile Die Zeile im Modula-2*-Quellcode, in der das Konstrukt anfängt.

Spalte Die Spalte, in der das Konstrukt anfängt.

BisZeile Die Zeile, in der es aufhört.

BisSpalte Die entsprechende Spalte.

A-Wert Der Aktivierungswert, der innerhalb dieses Konstrukts gültig ist, d.h. Prozesse mit anderem Wert werden dort nicht ausgeführt. Wenn ein solcher Wert nicht explizit vorhanden ist, z.B. in sequentiellem Kontext oder parallelem Kontext auf einem einzelnen Rechner, wo alle Prozesse des `FORALL ... IN PARALLEL` nacheinander in einer Schleife ausgeführt werden, dann steht hier eine Null.

A-Feld Ein Zeiger auf das Feld der Aktivierungswerte, falls ein solches vorhanden ist (z.B. innerhalb eines synchronen `FORALL`s), ansonsten ein Null-Zeiger.

Bereich Ein Zeiger auf die Variable, die eine `FORALL`-Anweisung bestimmt, z.B. die Laufvariable einer Virtualisierungsschleife im parallelen `FORALL` auf einem einzelnen Rechner[7]. Wenn keine solche Variable exisitert, dann ist der Bereich gleich dem Null-Zeiger.

Von, Bis Die Bereichsgrenzen dieser Variablen, falls vorhanden, ansonsten Null.

Die Information aus den Debug-Funktionen liefert somit Aufschluß über das momentan aktive Sprachkonstrukt, seine Lage im Programmtext, seinen Kontext (durch Überwachung der dynamischen Vorgänger) und den Stand der Aktivierungen, falls der Kontext parallel ist.

3.5.3 Einfügen der Debug-Funktionen

Im Anhang C werden die Abbildungen von Modula-2* in die C-ähnliche Zielsprache in Hinblick auf die Einfügung der Debug-Funktionen genau definiert. Sie an dieser Stelle aufzuführen, würde den Textfluß stören und keine wesentlichen Erkenntnisse bringen. Im Prinzip wird immer vor Beginn eines Konstrukts die entsprechende `StartDebugFn` aufgerufen, bei einem Durchlauf die `UpdateDebugFn` und am Ende die `EndDebugFn`.

[7] Dort ist ja in einem parallelen `FORALL` immer nur ein einziger Prozeß aktiv, nämlich der, dessen Nummer gerade der Schleifenvariablen entspricht. Alle mit kleinerer Nummer sind bereits durch das `FORALL` gelaufen, und alle mit größerer Nummer sind potentiell aktiv.

Kapitel 4

Verwandte Arbeiten

Auf dem Gebiet des parallelen Debuggens gibt es inzwischen viele verschiedene Ansätze. Im folgenden werden die für den Vergleich mit dieser Diplomarbeit relevanten Debugger kurz beschrieben und bewertet. Nicht aufgeführt sind hier Systeme, die ganz andere Schwerpunkte haben, etwa reine verteilte Debugger.

Zu Beginn werden zwei kommerzielle Debugger für SIMD-Rechner und ein Forschungsprojekt für Transputercluster vorgestellt. Ein Abschnitt über einzelne interessante Ideen anderer Projekte rundet das Kapitel ab.

4.1 PRISM

PRISM ist der Debugger der Connection Machine [Thin91][1]. Er bietet eine X-Windows basierte Umgebung für Entwicklung, Test und Leistungsanalyse von Programmen. Untersucht wurde die Version 1.0 vom November 1991.

Mit PRISM lassen sich Programme in C*, dem parallelen C-Dialekt der Connection Machine, untersuchen. Diese Sprache unterstützt automatische Virtualisierungen von Feldern, bietet ansonsten aber keine Abstraktionen von der Maschine.

Von der Präsentation her ähnelt PRISM dem `xdbx`: Ein Quellcodefenster zeigt das zu untersuchende Programm und Breakpoint-Informationen an, eine Menüleiste stellt Befehle zur Verfügung, eine Leiste mit Knöpfen bietet Abkürzungen für oft benutzte Befehle, und ein Kommandofenster läßt zusätzlich direkte Eingabe von Anweisungen zu. Die Ausgabe verschiedener Befehle wird in eigenen Fenstern angezeigt. Die Kommandos von PRISM stellen eine Erweiterung von `dbx` um parallele Konstrukte und Connection Machine-spezifische Anweisungen dar.

Breakpoints gelten global und können komplexen Ereignissen zugeordnet werden, durch die sie bei deren Auftreten ausgelöst werden. Zur Unterstützung der Parallelität bietet PRISM mehrere Visualisierungen an, nämlich Textdarstellung der Werte, Darstellung durch Farben, Booleschen Vergleich mit einem benutzerdefinierten Wert und Graustufenabbildung. Ähnlich wie bei MSDB werden mehrdimensionale Felder als zweidimensionale Schnittflächen dargestellt.[2]

Außerdem kann PRISM die Zeiten, die das parallele Programm mit Arbeit, Kommunikation und Reduktionen verbracht hat, aufschlüsseln und den einzelnen Zeilen des Programms in Form von Histogrammzeilen zuordnen.

[1] Zur Bewertung stand lediglich das Handbuch des Debuggers zur Verfügung, das allerdings ausführlich und mit vielen Bildschirmabzügen versehen ist.

[2] Die Visualisierungen in MSDB wurden völlig unabhängig von PRISM entwickelt (dessen Handbuch erst später zur Verfügung stand) – interessant ist, daß sich vergleichbare Ansätze zur Datendarstellung ergaben.

Die „Entwicklungsumgebung“ besteht aus einer Möglichkeit zum Aufruf eines Editors und des UNIX `make` Befehls.

Zu allen Fenstern von PRISM gibt es eine On-Line-Dokumentation.

Insgesamt macht PRISM den Eindruck eines professionell auf die Connection Maschine zugeschnittenen `xdbx`, der große Datenmengen durch Visualisierungen darstellen kann. Wie von der SIMD-Architektur her zu erwarten, ist eine Darstellung von parallelem Programmablauf nicht vorgesehen, da alle PEs entweder dieselben Anweisungen ausführen oder inaktiv sind. Allerdings fehlt eine Anzeige der Aktivierungen der einzelnen virtuellen oder realen PEs, die durchaus weitere Einblicke in die parallelen Vorgänge bieten könnte.

4.2 MPPE

Zum Debuggen von `MPL` und FORTRAN-Programmen auf der MasPar steht der MPPE (MasPar Programming Environment [Masp91b]) zur Verfügung[3]. Auch er bietet eine grafische Oberfläche unter X-Windows zur Fehlersuche und Leistungsanalyse von Programmen. Untersucht wurde die Version 3.0.1 von 1992.

Mit MPPE können Programme in FORTRAN und MPL, dem C-Dialekt der MasPar, untersucht werden. MPL kennt keine Virtualisierungen, so daß der Programmierer sämtliche damit verbundenen Probleme selbst lösen muß.

MPPE ist ein eigenständiger Ansatz, der auf keinem sequentiellen Debugger basiert und auch keinerlei textuelle Befehlseingabe kennt. Alle Aktionen werden durch Knöpfe und Menüauswahl und durch Eingabe von Werten ausgeführt. Vorhanden sind ein Kontrollfenster, in dem Meldungen von MPPE erscheinen, ein Quellcodefenster mit zusätzlichem Blick auf globale Variablen, Programmkeller und Breakpoints, sowie bei Bedarf erscheinende Fenster für jede mögliche Aktion wie etwa eine Datenuntersuchung.

Breakpoints sind auch hier global und können mit Bedingungen verknüpft werden. MPPE bietet eine einfache Visualisierung von Werten eines Feldes durch Schwarzfärben aller Elemente, die eine einfache benutzerdefinierte Bedingung erfüllen. Mehrdimensionale Felder können in Schnitten „durchblättert“ werden. Über eine externe Schnittstelle sind komplexere Visualisierungen durch den Benutzer möglich. Die Zustände aller physikalischen PEs können als 2-dimensionales Feld angezeigt werden.

Profiling wird statisch – durch Gewichtung der Befehle durch den Übersetzer – und dynamisch – als Messung der verbrauchten Rechenzeit – angeboten. Die Ergebnisse erscheinen als Histogrammbalken neben dem Programmcode. Eine vielfältig aufbereitete Ausgabe, z.B. auf den Drucker, ist möglich.

Fast alle Objekte eines Programms oder einer Visualisierung können durch Anklicken „vergrößert“ werden und liefern dann genauere Informationen z.B. über Typ der Variablen oder genauen Wert des Feldelements.

Ein On-Line Hilfesystem ist ebenfalls vorhanden.

Zusammenfassend läßt sich MPPE als professioneller und konsequent grafikorientierter Debugger bezeichnen. Die Visualisierungen sind recht eingeschränkt, erfassen aber auch die Aktivierungen der PEs. Da auch die MasPar ein SIMD-Rechner ist, gibt es keine Darstellung des parallelen Ablaufs.

[3] Eine MasPar stand für die Diplomarbeit direkt zur Verfügung, so daß diese Bewertung auf realen Erfahrungen basiert.

4.3 ModulaP

Im Rahmen einer Diplomarbeit am Forschungszentrum Informatik wurde ein Debugger für Modula-P entwickelt, der auf Transputerclustern läuft [Rich91][4].

Er ermöglicht die Untersuchung von Programmen in Modula-P, der parallelen Erweiterung von Modula-2 nach dem CSP-Modell (kommunizierende sequentielle Prozesse). Diese Sprache verwendet eine nachrichtenbasierte Kommunikation zwischen explizit erzeugten Prozessen auf Transputernetzen. Es handelt sich also, im Gegensatz zu den beiden bisher besprochenen Ansätzen, um ein MIMD-System.

Der Debugger bietet eine einfache grafische Oberfläche mit einem Quellcode-Fenster, in dem man Breakpoints setzen kann, sowie einer Reihe von Knöpfen für Aktionen wie Start und Einzelschritt. Eine Hilfsfunktion ist vorhanden.

Breakpoints sind global. Realisiert wird die Debug-Funktionalität durch Instrumentieren des Quellcodes an jeder Stelle, wo ein Breakpoint auftreten kann, und erneuter Übersetzung. Außerdem werden Informationen über Variablentypen und Sichtbarkeit aus dem Quelltext gezogen. Bei der Ausführung wird an jeder möglichen Breakpoint-Position eine globale Variable geprüft und die Ausführung gestoppt, wenn diese gesetzt ist. Zusätzlich wird bei jedem Prozeduraufruf und jeder Prozeßerzeugung eine Nachricht an die zentrale Informationsverwaltung des Debuggers geschickt. Durch diese Maßnahmen verlangsamt sich der Programmablauf gegenüber dem unüberwachten Ablauf um etwa das 15-fache.

Profiling-Informationen werden nicht gesammelt, obwohl sich dies anböte, da der Code bereits instrumentiert wird. Eine Darstellung der Prozeßaktivierungen entfällt, da Modula-P einen anderen Ansatz als Modula-2* wählt und keine virtuellen Aktivitäten kennt.

Insgesamt handelt es sich um eine interessante Forschungsarbeit, die durch eine Codetransformation eine Debug-Funktionalität erreicht, ohne auf weitere Werkzeuge unterer Ebene aufzusetzen.

4.4 Forschungsansätze

Hier werden Ideen aus Forschungsarbeiten vorgestellt, die eine andere Zielsetzung als MSDB haben, aber im Detail interessant sind.

Dynamischer Prozeßbaum: In [RRZ88] wird eine Darstellung des parallelen Ablaufs verschiedener Prozesse durch einen Baum vorgestellt. Außerdem bieten die Autoren weitergehende Visualisierungen, z.B. der Fortschritte der einzelnen Prozesse während der Berechnung, an, die auf einem „von oben gesehenen" Prozeßbaum basieren.

Erweitert wird diese Idee in [ZeRu91], wo verallgemeinert der Begriff der „Arbeit" (Prozessoren, Prozesse oder Speicher) in eine Baumdarstellung eingeht.

In der Dokumentation zum Modula-P Debugger [Rich91] wird ebenfalls ein Prozeßbaum erwähnt.

Allerdings abstrahieren diese Ansätze nicht von Vorgängen auf unterer Ebene, sondern vergrößern noch Details dieser Sicht. Eine Gruppierung von Aktivitäten ist nicht möglich. Die dynamischen Strukturbäume von MSDB hingegen arbeiten auf Ebene der Programmiersprache, unterscheiden einzelne Sprachkonstrukte und fassen äquivalente Äste im Baum zusammen.

[4] Dieser Abschnitt basiert ausschließlich auf der entsprechenden Ausarbeitung der Diplomarbeit.

Profiling: [BeBe92] bildet die theoretische Grundlage der Profiling-Einrichtung des MPPE und beschreibt die Sammlung von Daten auf Befehls- und Prozedurebene sowie ihre grafische Darstellung und interaktive Anzeige.

MSDB hat in dieser Hinsicht weniger zu bieten, da Profiling lediglich auf Ebene der Sprachkonstrukte durchgeführt wird und auch nur die Durchläufe, nicht aber die real verbrauchte Zeit angibt.

Breakpoints: Im verteilten Fall sind Breakpoints Gegenstand vieler Artikel. [Elsh88] beschreibt die Probleme, die sich durch den Übergang vom sequentiellen auf den verteilten Ablauf ergeben, wie Konsistenz, Anhalten von Teilaktivitäten und Zeitbedingungen. Diese Betrachtungen sind auf Betriebssystemebene angesiedelt. Auch [MiCh88] widmet sich dem Anhalten eines verteilten Programms in konsistentem Zustand und führt dazu verschiedene Prädikate ein.

Für die Ebene von Modula-2* sind diese Überlegungen weniger bedeutend. Daten, die im Augenblick des Stops gerade verschickt wurden, werden durch das Laufzeitsystem gepuffert. Die Trennung in Worker und Runner eliminiert somit Probleme wie den Nachrichtenverlust. Da zudem die Breakpoints global sind, können auch keine Puffer überlaufen, da keine weiteren Nachrichten abgeschickt werden können. Lediglich der konsistente Programmzustand ist nicht garantiert, aber auch nur in Hinblick auf z.B. unvollständig durchlaufene Virtualisierungsschleifen, wie im Entwurf erwähnt.

4.5 Bewertung

Viele der in MSDB angewandten Techniken finden sich auch in anderen Arbeiten, wie etwa verschiedene Visualisierungen und grafische Oberflächen. Kommerzielle Debugger bieten darüber hinaus auch ausgefeilte Profiling-Funktionen und große interaktive Hilfssysteme.

Einzigartig sind bei MSDB aber das hohe Abstraktionsniveau und die Gruppierungsmethoden, wie sie bei den dynamischen Strukturbäumen angewandt werden. Vorhandene Debugger orientieren sich an einem bestimmten Maschinenmodell, sei es SIMD oder verteiltes Rechnen, aber keiner versucht, für viele verschiedene Parallrechner gültige Paradigmen zu bieten.

Kapitel 5

Zusammenfassung und Ausblick

5.1 Zusammenfassung

Im Rahmen der Diplomarbeit wurden mehrere neuartige Konzepte zur Visualisierung und Abstraktion entworfen und implementiert:

- Parallele Abläufe werden auf Sprachstrukturen-Ebene nach ihrer dynamischen Vorgeschichte gruppiert und als Baum dargestellt.
- Mehrdimensionale Daten werden, nach verschiedenen Kriterien geordnet, visualisiert.
- Profiling auf Konstruktebene ermittelt die am häufigsten durchlaufenen Programmteile.
- Die Aktivierungen der virtuellen PEs werden visualisiert.
- Über ein Netz von Workstations verteilte Daten und Aktivitäten werden zusammengefaßt und dem Benutzer zentral präsentiert.
- Eine durchgehende grafische Benutzeroberfläche vereinfacht die Bedienung wesentlich.
- Der Debugger wurde für die Fehlersuche auf einer einzelnen Workstation und einem Netzwerk von Workstations implementiert. Dabei wurden auch Konzepte für die Portierung auf eine MasPar und weitere Rechner entwickelt.

Der Debugger abstrahiert dabei von allen maschinenspezifischen Details wie etwa der Kommunikation. Implementiert wurden Versionen für einen einzelnen Prozessor und ein Netzwerk von Workstations. Umsetzungen auf eine MasPar und ein Transputernetz sind geplant.

5.2 Ausblick

Es bieten sich noch mehrere Punkte zur Erweiterung des MSDB an, darunter verschiedene Portierungen, Verbesserungen in der Genauigkeit des Profiles und beim verwendeten Debugger sowie die Integration zu einer kompletten Entwicklungsumgebung. In den folgenden Abschnitten werden mögliche Vorgehensweisen dazu beschrieben.

5.2.1 Portierungen

Viele parallele Rechner sind potentielle Zielarchitekturen für den MSDB:

MasPar: Ursprünglich war die MasPar das erste Ziel für den parallelen Debuggers. Da dort aber keine Werkzeuge auf unterer Ebene bereitstanden, wie etwa ein Äquivalent zum `dbx`, wurde das Laufzeitsystem für Workstations als paralleles System gewählt. Funktionalität auf unterer Ebene beinhaltet u.a. das Setzen und Ausführen von Breakpoints, den Einzelschritt, das spontane Anhalten des Programms und den Zugriff auf Daten anhand ihres Namens.

Wenn man nur eine grobkörnige Kontrolle der Aktivitäten benötigt, ist eine Implementierung auf der MasPar recht einfach: Die Debug-Funktionen werden um eine Abfrage einer globalen Variable erweitert. Wird die Debug-Funktion aufgerufen, was mindestens bei jedem Durchlauf durch eine Schleife der Fall ist, so prüft sie die globale Variable und führt je nach deren Wert eine bestimmte Funktion aus. Das kann etwa der Stop des Programms sein oder die Rückgabe von Datenwerten. Für die Abbildung von Variablennamen auf Adressen kann entweder die Symboltabelle des Programms verwendet werden, oder man erzeugt außer den Debug-Funktionen noch Zugriffsfunktionen für alle vorkommenden Variablen.

Dieses Vorgehen ähnelt dem bei Modula-P [Rich91] gewählten Ansatz.

Da es sich um einen SIMD-Rechner handelt, degeneriert der dynamische Strukturbaum hier zu einer Liste. Er kann daher z.B. zentral auf der ACU gehalten werden, da die einzelnen PEs sehr wenig lokalen Speicher haben.

Eine besondere Behandlung benötigt die Trennung des Programms in Frontend– und Backend-Teil, da das Frontend aus einer DECstation besteht, auf der etwa der `dbx` voll eingesetzt werden kann, während auf dem parallelen Backend nur auf die Debug-Funktionen zurückgegriffen werden kann. Denkbar wäre eine feinkörnige Kontrolle über die sequentiellen Programmteile auf dem Frontend und eine grobkörnigere Behandlung auf dem Backend durch die Debug-Funktionen.

Transputernetz: Hier kann man wieder in Hinsicht auf die Verwendung der Debug-Funktionen ähnlich wie bei der MasPar vorgehen, wobei es sich diesmal um einen MIMD-Rechner handelt. Die Kontrolle muß also analog zum LAN-Debugger verteilt werden. Ebenso sind verteilte Breakpoints global auszulösen etc., was durch eine weitere Aufstockung der Debug-Funktionen möglich wäre.

Da kein grundlegender Debugger wie der `dbx` vorhanden ist, müssen auch mehr Informationen über Variablen, ihre Typen und ihre Lage im Speicher vorhanden sein. Diese könnten etwa durch weitergehende Einfügung von Informationen durch den Übersetzer gewonnen werden.

KSR1: Die Beschaffung einer Kendall Square Research KSR1 ist im Gange. Dieser Rechner ist eine MIMD-Architektur mit globalem verteilten Speicher. Auf den einzelnen Prozessoren läuft ein UNIX-Derivat.

Durch leichte Modifikationen am Laufzeitsystem sollte es sich schnell auf diese Maschine anpassen lassen, indem es sie wie ein Netzwerk von Workstations behandelt, wobei die „Workstations“ eben schnelle Prozessoren sind, die durch ein Hochgeschwindigkeitsnetz verbunden werden. Der Debugger würde exakt wie im verteilten Fall arbeiten, da wiederum auf einen vorhandenen `dbx` zurückgegriffen werden könnte.

Deutliche Effizienzgewinne dürfte eine Neuimplementierung des Laufzeitsystems speziell für die KSR1 bringen, wenn man den globalen Speicher ausnutzt. In diesem Fall könnte der Debugger auch neu überarbeitet werden und auch Gebrauch von direkten Speicherzugriffen etc. machen. Genauere Ideen zur Implementierung sind aber erst nach Eintreffen der Maschine und ihrer Dokumentation sinnvoll.

Es sind also Portierungen auf eine Reihe von verschiedenen Maschinen denkbar, wobei die Feinheit der Kontrolle von verfügbaren Werkzeugen auf Betriebssystemebene abhängt. Durch die Debug-Funktionen läßt sich in jedem Fall eine Fehlersuche auf Konstruktebene erreichen.

5.2.2 Genaueres Profiling

Die einzelnen Konstrukte und der in ihnen enthaltene Code könnten statisch mit Gewichten versehen werden, um den Rechenaufwand abzuschätzen. Bisher wird nur die Zahl der Durchläufe betrachtet, aber verschiedene Anweisungen können sich um Größenordnungen in ihrer Laufzeit unterscheiden. Eine Gewichtung der konkreten Konstrukte wäre aussagekräftiger.

Weiterhin wäre auch eine Erweiterung der Debug-Funktionen zur direkten Zeitmessung denkbar, wodurch die Profiles sehr genau den Verbrauch an Rechenzeit widerspiegeln würden. Allerdings müßte man den Effekt der Debug-Funktionen selbst aus dem Ergebnis herausrechnen. Zudem ist eine Zeitnahme deutlich aufwendiger als das einfache Erhöhen eines Zählers, so daß die Laufzeit der Programme unter Umständen deutlich zunehmen könnte.

5.2.3 Debugger-Kern

Die bisherige Realisierung der unteren Ebene des Debuggers durch Aufsetzen auf `dbx` oder `gdb` hat zwar den Vorteil der weitgehenden Portabilität unter UNIX-Rechnern und der Abstraktion von der Maschinenebene, besitzt aber auch mehrere Nachteile:

- Das Zerteilen der vom Debugger zurückgemeldeten Zeichenketten und das Erzeugen von Kommandos für den Debugger ist ein aufwendiger Umweg. Eleganter wäre eine direkte Schnittstelle auf C-Ebene.

- Die Kommunikation über ein Pseudo-Terminal ist ein Engpaß im Datenaustausch zwischen MSDB und untergeordnetem Debugger. Eine durchsatzstärkere Verbindung wäre wünschenswert.

- Ein eigener Debugger-Prozeß belastet den Rechner stärker und macht den direkten Austausch von Daten zwischen den Adreßräumen des MSDB und des Debuggers unmöglich. Praktischer wäre eine Integration von MSDB und `dbx` oder `gdb`.

Diese Punkte lassen sich durch die Entwicklung eines speziell für den MSDB ausgelegten Debugger-Kerns verbessern, der als Unterfunktion zur Verfügung stünde und damit direkt aufrufbar wäre. Um die Implementierung der entsprechenden Funktionalitäten zu vereinfachen, könnte man auf den frei verfügbaren Quellcode des `gdb` zurückgreifen und dessen Benutzerschnittstelle geeignet modifizieren, um eine direkte Zusammenarbeit mit MSDB zu erreichen.

5.2.4 Hilfestellungen

Um die Akzeptanz und Benutzerfreundlichkeit des MSDB zu erhöhen, wäre ein System von Hilfestellungen wünschenswert, die kontextspezifische Hinweise auf die Bedienung geben. Die momentane Installation des MSDB ist noch experimentell und an manchen Stellen auf eine konkrete Umgebung eines bestimmten Benutzers zugeschnitten. Eine vollständige Verallgemeinerung und genaue Installationshinweise wären von Vorteil.

5.2.5 Entwicklungsumgebung

MSDB könnte zu einer vollständigen Entwicklungsumgebung ausgebaut werden, die außer der Fehlersuche auch die Phase des Programmierens unterstützt. Denkbar wären z.B. eine grafische Darstellung der verwendeten Modula-2*-Module mit ihren Abhängigkeiten, eine Integration eines X-Windows basierten Editors und eine Verbindung zwischen Editor, Übersetzer und Debugger, um einen konsistenten Gesamteindruck zu schaffen.

Anhang A

Kriterien für eine wirkungsvolle Visualisierung

Eine Aufstellung von Kriterien, die eine wirkungsvolle Visualisierung erfüllen muß, findet sich in [Mill92]. Im folgenden werden diese Punkte erläutert und die in der Diplomarbeit verwendeten Visualisierungen – d.h. die Visualisierung der aktiven Prozesse, der Strukturbaum und die Visualisierung von mehrdimensionalen Daten – anhand dieser Kriterien beurteilt.

Keine großen Vorkenntnisse: Um die Visualisierung zu verwenden, darf es nicht notwendig sein, daß der Benutzer bereits eine detaillierte Vorstellung von den Daten hat. Eine Visualisierung, die nur aufgrund genauer Angaben des Benutzers ein aufschlußreiches Bild liefert, macht es sehr schwer, zu neuen Erkenntnissen zu kommen.

Die Implementation im Rahmen dieser Diplomarbeit erfüllt dieses Kriterium, da nicht lange mit komplizierten Parametern experimentiert werden muß, um ein aussagekräftiges Bild zu erhalten. Der Benutzer erhält Einblicke in die Struktur der Daten, ohne schon vorher eine genaue Vorstellung davon besitzen zu müssen.

Eignung für das Programmiermodell: Die Visualisierung soll in Begriffen der untersuchten Programmiersprache erfolgen, d.h. nicht zu weit abstrahieren oder zu sehr in Details gehen, die für die Sprache bedeutungslos sind.

Auch dieses Kriterium ist erfüllt. Der Debugger abstrahiert z.B. von Low-Level-Kommunikation, wie es auch Modula-2* tut, und von der Transformation zwischen Modula-2* und `C`.

Skalierbarkeit: Durch die große Anzahl von Datenelementen ist es im Parallelen wichtig, daß eine Visualisierung aussagekräftig bleibt und nicht unübersichtlich wird, wenn sich die Datenmenge weiter erhöht.

Für die Visualisierung von mehrdimensionalen Daten und Aktivierungen trifft dies zu, sie wurde ja gerade für diesen Zweck entwickelt. Der Strukturbaum wird bei einer großen Anzahl von Knoten zweifellos unübersichtlich, aber durch die Gruppierung von Prozessen wird dieser Tendenz entgegengewirkt.

Die jeweils verwendete Maschine begrenzt natürlich die real handhabbare Größe der Visualisierungen, spätestens durch Überlaufen des Speichers. Sehr große Darstellungen brauchen zudem lange Rechenzeiten, besonders bei der Darstellung in Graustufen, wo über alle Elemente des Felds Mittelwert und Varianz bestimmt werden. Diese Einschränkungen sind jedoch nicht prinzipiell bedingt.

Informative Farbgebung: Bei Verwendung von Farbe soll nicht ein „schönes", sondern ein aussagekräftiges Bild entstehen. Ungeschickter Gebrauch von Farbe verwirrt nur.

Dieser Punkt ist im Rahmen der Diplomarbeit unwesentlich, da von Farbe kein Gebrauch gemacht wird. Vergleichbar sind höchstens die Graustufen bei der Visualisierung von Werten, die aber zweckdienlich und nicht verkünstelt sind.

Interaktivität: Die verschiedenen Aspekte eines parallelen Programms sollten sich interaktiv untersuchen lassen. Sie alle in eine einzige Visualisierung zu pressen, wäre unübersichtlich.

Durch die Verwendung verschiedener Visualisierungen ist dieses Kriterium erfüllt.

Beschriftung: Eine Visualisierung sollte sinnvolle Beschriftungen aufweisen, um den dargestellten Sachverhalt richtig zu bezeichnen.

In dieser Diplomarbeit werden die untersuchten Objekte (Aktivierungen, Felder, Strukturbäume etc.) immer in der Titelzeile des jeweiligen Fensters aufgeführt, sind also problemlos zu erkennen.

Defaulteinstellungen: Durch eine geeignete Defaulteinstellung soll dem Benutzer ein sofortiger Einblick in die dargestellten Sachverhalte ermöglicht werden, ohne daß er lange mit der Einstellung experimentieren muß.

Das ist bei der Visualisierung von Daten und Aktivierungen der Fall. Sollte die angezeigte Datenmenge zu groß für das Fenster werden, kann der Benutzer mittels Scrollbars schnell die Position wechseln oder den Vergrößerungsfaktor ändern.

Abstrahieren: Die komplexe innere Struktur eines Ablaufes zu sehen, kann faszinierend sein. Um aber einen Überblick zu bekommen, ist eine gewisse Abstraktion notwendig.

MSDB gibt lediglich durch das optionale Quellcodefenster mit dem C-Zwischencode einen Einblick auf untere Ebenen, der für Übersetzerbauer und interessierte Benutzer gedacht ist. Zur Bedienung des Debuggers ist er nicht nötig.

Einfache Bedienung: Der Benutzer darf nicht mit einer Vielzahl von verschiedenen Steuerungsobjekten überfordert werden.

Bei den hier verwendeten Visualisierungen ist dies gegeben; es wird für jeden Abstraktionsaspekt ein einziges übersichtliches Objekt geboten. Lediglich bei der Visualisierung von Feldern gibt es mehrere Möglichkeiten, die aber nicht verwirren.

Insgesamt kann man sagen, daß die Kriterien fast ausnahmelos erfüllt sind (und das, obwohl zum Zeitpunkt des Entwurfs und wesentlicher Teile der Implementierung der Visualisierungen der entsprechende Artikel noch gar nicht bekannt war).

Anhang B

Laufzeitsystem für LANs

Dieser Anhang beschreibt Entwurf, Implementierung und Leistungsbewertung des Laufzeitsystems für Modula-2* zum Einsatz auf Workstation-Netzen.

Ein Netzwerk von Workstations ist ein interessanter Extremfall für Modula-2*, das ja für datenparalleles Programmieren entworfen wurde und sich auf der SIMD MasPar gut bewährt hat. Ein LAN hingegen ist ein extremes MIMD System mit äußerst hohen Kosten für Kommunikation und Synchronisation.

Die folgende Ausführung umfaßt nur das eigentliche Laufzeitsystem, ohne den verteilten Debugger selbst. Details über dessen Entwurf und Implementierung finden sich in Abschnitt 3.4.3.

B.1 Entwurf

Ziel des Entwurfs war ein praktikables Konzept für ein einsatzfähiges Laufzeitsystem auf einem Netzwerk von SUN Workstations.

B.1.1 Aufsetzen auf p4

Um nicht auf unterster Betriebssystemebene aufsetzen zu müssen, wurden bereits existierende Laufzeitsysteme für verteilte Programmiersprachen auf ihre Eignung hin untersucht. Ein weiterer Vorteil, der sich aus der Benutzung eines vorhandenen Systems ergibt, ist die vereinfachte Portabilität auf andere Rechner, die vom System schon unterstützt werden.

Zwei solche Systeme sind als Quellcode frei erhältlich und recht weit verbreitet im Einsatz. Beide bieten typisierten Nachrichtenaustausch in heterogenen Computernetzen als Basis für verteilte Programme und stellen Grundfunktionen zum Starten und Beenden verteilter Anwendungen zur Verfügung. Im folgenden werden die Benutzung und die Eigenschaften der beiden Systeme kurz beschrieben:

PVM (Parallel Virtual Machine) wurde am Oak Ridge National Laboratory in Tennessee entwickelt. Untersucht wurde die Version 2.3 vom 1.6.1991.

Vor der Benutzung muß von jedem Benutzer auf jeder zu verwendenden Maschine ein Dämon (also ein Dienstgeber-Prozeß) installiert werden, der das verteilte Programm auf dieser Maschine startet. Die Maschinen, die an einer bestimmten Berechnung teilnehmen sollen, werden in eine Datei eingetragen, die dann einem Dämonen übergeben wird. Danach kann der Benutzer eine PVM-Anwendung starten.

Ein Programm verwendet PVM, indem es sich durch einen Aufruf mit einem beliebigen Namen bei der Kommunikation anmeldet. Es erhält eine Instanznummer zurück und kann danach auf anderen Maschinen weitere PVM-Programme durch Angabe ihres Objektdatei-Namens und der Maschinenarchitektur starten, die sich ihrerseits wieder anmelden müssen. Sind alle PVM-Prozesse aktiviert, so können sie durch Sende– und Empfangsoperationen Nachrichten austauschen.

Nachrichten bestehen aus einem Typ (einer vom Benutzer gewählten Zahl), einem Empfänger (bestehend aus Name und Instanznummer) und der eigentlichen Nachricht. Mehrere verschiedene Datentypen können in einer einzigen Nachricht gebündelt versandt werden. Empfangsoperationen können auf einen bestimmten Nachrichtentyp warten.

Außerdem stehen Funktionen zur Ermittlung der an der Berechnung beteiligten Maschinen und Nachfragen über den Empfang einer Nachricht etc. zur Verfügung. Globale Barrieren, die durch eine eindeutige Zeichenfolge identifiziert werden, sind vorhanden.

Daten werden automatisch in einem einheitlichen Transferformat versandt, wenn die verwendeten Maschinen unterschiedliche Datendarstellungen verwenden.

Nach Ende der Berechnung tragen sich die Prozesse aus der Kommunikation aus und stoppen.

p4 wurde am Argonne National Laboratory in Illinois entwickelt. Untersucht wurde die Version 1.2 vom 10.9.1992.

Anders als bei PVM startet der Benutzer bei p4 nur ein Programm. Dieses startet dann selbst die restlichen Prozesse im Netzwerk anhand einer ihm übergebenen Datei mit Objektdatei-Namen und zu verwendenden Maschinen. Wenn auf den entsprechenden Maschinen ein p4 Server läuft, beschleunigt sich der Start des verteilten Programms, ansonsten wird eine Remote Shell verwendet. Zum Betrieb selbst wird der Server nicht gebraucht.

Alle so gestarteten Programme erhalten eine Kennzahl, anhand der sie miteinander kommunizieren können. Eine Nachricht besteht aus Zielkennung, Nachrichtentyp (Integer) und Nachrichteninhalt (unstrukturierter Puffer mit beliebiger Länge).

Operationen zum blockierenden und nichtblockierenden Senden sowie zum Empfangen und zum Test, ob eine Nachricht ansteht, stehen zur Verfügung. Auf Nachrichten von bestimmten Absendern oder mit bestimmtem Typ kann gewartet werden.

p4 bietet ebenfalls Funktionen zum Ermitteln der Teilnehmerzahl, der eigenen Nummer und weiterer Einzelheiten. Debug-Ausgaben mit verschiedenen Detailgraden sind möglich. Nachrichten an Maschinen mit anderer interner Darstellung können durch besondere Sende– und Empfangsaufrufe bearbeitet werden.

Zusätzlich sind einfache globale Reduktions-Operationen für Summe, Maximum, Minimum etc. sowie globale Synchronisationsbarrieren vorhanden. Shared Memory wird auf Maschinen, die es schon anbieten, unterstützt. Primitive zum Bilden von Monitoren sind ebenfalls vorhanden.

Ist die verteilte Berechnung beendet, gehen die Prozesse in einen Wartezustand über und stoppen, sobald alle Prozesse fertig sind.

Die Wahl fiel auf p4, da es flexibel per Remote Shell oder Dämon aufgerufen werden kann, Debug-Ausgaben direkt unterstützt, globale Reduktions-Operationen bietet und eine engere Kontrolle der maschinennahen Vorgänge zuläßt.

Zudem besitzt PVM in der untersuchten Version mehrere Nachteile:

- Sync-Barrieren können nur einmal pro Identifikator verwendet werden, d.h. verschiedene Sync-Barrieren müssen verschiedene Kennziffern verwenden.
- Der Empfangsbefehl kann zwar auf bestimmte Nachrichtentypen, aber nicht auf Nachrichten von einem bestimmten Sender warten.
- Jeder reale Rechner kann nur ein virtuelles PE beherbergen (!).
- PVM verlangt eine starre Verzeichnisstruktur beim Benutzer, in die die Programme eingebettet sein müssen.
- Die Adressierung der Programme durch Namen und Instanznummer ist von der Anwendung her aufwendiger.
- Ein Dämon kann nur jeweils einen Benutzer bedienen und muß auf jeder einzelnen Maschine von Hand in `/tmp` installiert werden.

Während der Implementierung stellte sich heraus, daß einige der zusätzlichen Eigenschaften von p4 nicht für den geplanten Zweck verwendet werden können, da z.B. an einer Synchronisationsbarriere von p4 alle Prozesse teilnehmen müssen – im Laufzeitsystem sind davon aber nur die eigentlichen Arbeitsprozesse und nicht die des Laufzeitsystems selbst betroffen.

Dennoch war die Wahl von p4 auch im Rückblick kein Fehler; alle gewünschten Funktionen ließen sich ohne große Probleme verwirklichen.

B.1.2 Systemaufbau

Der Übersetzer für Modula-2* auf LANs geht von einer einem Parallelrechner ähnlichen Struktur der Programme aus:

Ein (virtuelles) *Frontend* arbeitet den sequentiellen Code ab und ruft im Falle eines `FORALL`s eine Prozedur auf allen *PE*s auf. Diese Prozedur wird parallel abgearbeitet, wobei sich die PEs bei Bedarf untereinander synchronisieren. Um zu ermitteln, ob noch Prozesse aktiv sind, wird ein globales Oder verwendet. Nach Ende der Berechnung erhält das Frontend wieder die Kontrolle.

Die Befehle zum Starten einer parallelen Prozedur und die Meldung des Prozedurendes an das Frontend sowie Start- und Ende-Befehle werden vom jeweiligen Kommunikationspartner erwartet. Erst nach ihrem Eintreffen fährt er in der Ausführung fort. Diese Befehle entsprechen dem Steuernetzwerk eines Parallelrechners wie etwa dem der MasPar.

Während der Berechnung können die PEs untereinander und auf dem Frontend auf Daten direkt zugreifen, ohne daß das betroffene PE mitwirken muß.

Der Ablauf ist in Abbildung B.1 dargestellt.

B.1.3 Anforderungen an das Laufzeitsystem

Aus den im vorigen Abschnitt geschilderten Sachverhalten ergeben sich folgende Anforderungen, die das Laufzeitsystem erfüllen muß:

Steuerung: Frontend und PEs müssen Botschaften austauschen können, um sich von Start und Ende einer parallelen Aktivität benachrichtigen zu können. Ein Prozeß darf erst weiterlaufen, wenn eine Botschaft vorliegt. Die zugehörigen Operationen sind *read* und *write*.

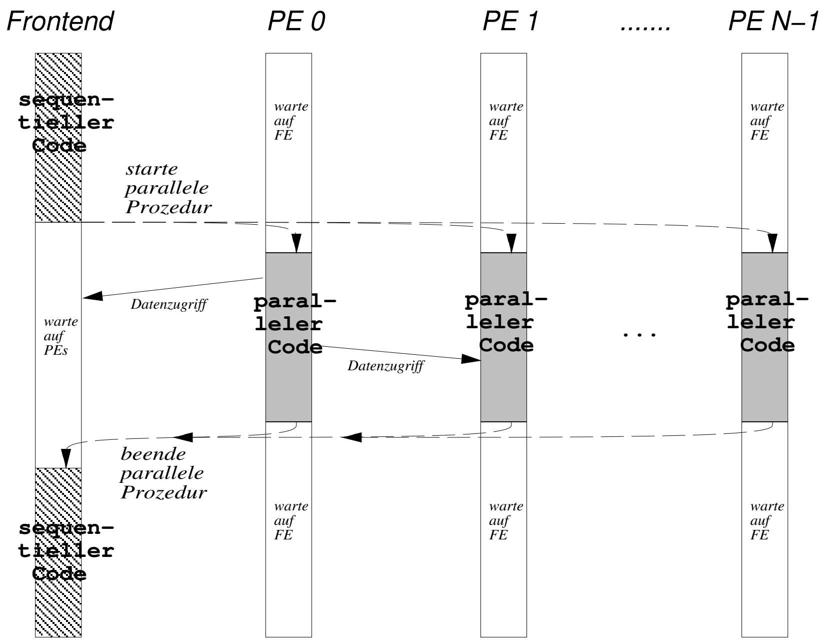


Abbildung B.1: Ablauf eines parallelen Modula-2* Programms auf mehreren Rechnern.

Datentransfer: Es muß möglich sein, Daten von anderen PEs oder dem Frontend zu erhalten, ohne daß diese etwas davon wissen (d.h. ohne einen expliziten Warte– oder Sendeaufruf), da die Datenzugriffe nicht vorab bekannt sind. Diese Operationen heißen im folgenden *put* und *get*.

Synchronisation: Eine Synchronisation der PEs untereinander muß unterstützt werden. Zudem ist ein globales Oder von Vorteil, um festzustellen, ob schon alle Prozesse mit der Abarbeitung fertig sind.

Dazu kommen noch Details wie Ausgabe von Daten für den Benutzer u.ä.

B.1.4 Architektur des Laufzeitsystems

Um diese Anforderungen zu erfüllen, wurde eine geteilte Architektur entworfen, die auf aus mehreren Prozessen aufgebauten „Software-PEs" basiert (diese Benennung soll die Prozeßgruppe im Laufzeitsystem, die zusammen die Funktion eines PEs erfüllt, von einem realen PE abgrenzen). Ein solches PE wird aus jeweils einem Arbeitsprozeß und einem Laufzeitprozeß gebildet. Dazu kommt für alle PEs zusammen ein Prozeß, der für die globale Kommunikation zuständig ist. Das Frontend unterscheidet sich rein von den Prozessen her nicht von einem PE, sondern führt lediglich andere Teile des Programms aus und wirft die Abläufe auf den PEs an.

Es gibt drei Arten von Prozessen:

Ausführung: Der eigentliche Rechenprozeß, der das Modula-2* Programm ausführt. Das Frontend arbeitet den sequentiellen Teil des Programms ab, während die PEs an den `FORALL`s rechnen.

Explizite Kommunikationsaufrufe zur Steuerung sind im Code der Rechenprozesse mit eingebunden, da sie zu bekannten Zeitpunkten im Programm erfolgen.

Datenzugriff: Jedem Rechenprozeß ist ein Laufzeitprozeß zugeordnet, der den Zugriff auf den Speicher des Rechners ermöglicht, ohne daß dieser mitwirken muß.

Um den Wert einer Variablen von einem PE zu erfahren, wendet sich der interessierte Rechenprozeß an den Laufzeitprozeß des Ziel-PEs. Dieser liest den Speicher des Zielrechners aus und schickt das Ergebnis dem anfragenden Rechenprozeß.

Synchronisation: Jedes PE schickt einen Sync-Aufruf an diesen Prozeß, wenn es sich mit den anderen PEs abstimmen will, und wartet danach auf eine Antwort. Die Synchronisation sammelt die Meldungen der PEs. Sobald die Daten aller PEs angekommen sind, schickt sie eine Botschaft an die PEs, die daraufhin in der Ausführung fortfahren.

Auf ähnliche Art wird das globale Oder realisiert.

Außerdem übernimmt die Synchronisation noch die Ausgabe von Daten an den Benutzer und das Starten und Beenden aller Prozesse.

B.1.5 Probleme im verteilten System

Zwischen den verschiedenen Prozessen des Laufzeitsystems wird ein Protokoll benutzt, das alle Ereignisse wie Prozeßstart, Ende, Ausgabe, Datentransfer und globales Oder realisiert.

Durch das Aufsetzen auf p4 werden Probleme wie der Verlust von Nachrichten im Netz vermieden, da sie schon von p4 behandelt werden.

Der Ausfall einer Maschine wirkt sich allerdings fatal aus, da p4 interne Rundrufe durch einen balancierten Baum, der über die Prozeßmenge gelegt wird, realisiert. Fällt ein Knoten weg, so meldet das p4 System einen Fehler und beendet die Ausführung[1]. Die Behandlung dieser Situation würde einen enormen Aufwand an Wiederaufsetzverfahren erfordern, so daß im folgenden von stabilen Maschinen ausgegangen wird.

Ein Engpaß des Laufzeitsystems liegt bei der Synchronisierung, da hierzu alle PEs eine Nachricht schicken müssen. Diese Nachrichten werden zentral gesammelt und nach vollständigem Erhalt wird eine Weiterlauf-Nachricht an alle PEs versandt. Da die Synchronisierung zudem noch die Ein– und Ausgabe übernimmt, kann sie bei Einsatz vieler PEs mit der Bearbeitung vieler gleichzeitiger Anforderungen unter Umständen nicht mehr nachkommen. Allerdings liegen die Laufzeiten von Nachrichten durch das Netz um Größenordnungen über der für ihre Bearbeitung nötigen Zeit.

B.2 Implementierung

Basierend auf den Entwurfsüberlegungen wurde das Laufzeitsystem implementiert. In diesem Abschnitt werden die einzelnen Programmteile und ihr Zusammenwirken beschrieben. Für den an Details interessierten Leser sind im Anhang D die einzelnen Funktionen des Laufzeitystems und die genauen Nachrichtentypen ausführlicher beschrieben.

B.2.1 Nachrichtentypen

Den oben angeführten Anforderungen entsprechend wurden verschiedene Nachrichtentypen zur Kommunikation zwischen den Prozessen festgelegt:

[1] Ein Knoten ist in diesem Fall nicht der entsprechende Prozeß selbst, sondern ein von p4 intern gestarteter Prozeß, der die Kommunikation durchführt. Ein Absturz des eigentlichen Programms wirkt sich also nicht so stark aus.

- Startnachricht, die auch Information über das jeweilige Programm und seine Prozeßidentifikation mitliefert
- Anforderung des Programmendes, geschickt von einem PE
- Befehl zum Beenden, von der Zentrale geschickt
- Anforderung zum Synchronisieren und Weiterlauf-Befehl
- Synchrones und asynchrones Lesen und Schreiben von Daten
- Globales Oder samt Rückgabewert
- Datenausgabe
- Weitere Befehle (z.B. für die Fehlersuche)

B.2.2 Programme

Der Aufteilung des Entwurfs folgend, gibt es drei grundlegende Programme bzw. Programmteile:

Syncer: Behandelt die Synchronisation und das globale Oder sowie Ausgaben des Programms an den Benutzer. Bei der Synchronisation werden die erhaltenen Sync-Nachrichten gezählt und bei Vollständigkeit eine Weiter-Nachricht an die PEs geschickt. Dabei wird geprüft, ob die Sync-Kennzahlen der Nachrichten identisch sind. Ist dies nicht der Fall, so liegt ein Programmfehler vor, und die Ausführung wird abgebrochen.

Zusätzlich ist der Syncer für das Starten aller verteilten Prozesse mittels p4-Aufrufen zuständig. Er schickt eine Start-Nachricht an alle Prozesse. Empfängt er eine Ende-Anfrage-Nachricht von einem Prozeß (typischerweise dem Frontend, das den sequentiellen Programmcode beendet), dann sendet er eine Ende-Nachricht an alle Prozesse, sobald alle noch ausstehenden Synchronisationen und globalen Oders vollständig abgearbeitet sind.

Der Syncer ändert sich für verschiedene Modula-2* Programme nicht, er ist fester Bestandteil des Systems.

Runner: Sorgt für den Zugriff auf Daten des zugeordneten Workers. Sobald er eine entsprechende Anfrage erhält, stoppt er seinen Worker (mittels des UNIX-Systemaufrufs `ptrace()`), liest Daten aus dessen Speicher oder legt sie dort ab, und meldet die gelesenen Daten bzw. den erfolgreichen Schreibzugriff dem anfordernden Worker[2].

Zu Beginn der Ausführung erhält der Runner von seinem Worker genauere Angaben über Programmnamen und Prozeßkennung, damit er später alle zum Zugriff erforderlichen Daten hat.

Auch der Runner ist ein unveränderlicher Systembestandteil.

Worker: Bietet dem Modula-2*-Programm Aufrufe für die Prozeßkommunikation, die Synchronisierung und den Speicherzugriff auf anderen PEs.

[2] Natürlich müssen sich Worker und zugehöriger Runner dazu auf derselben physikalischen Maschine befinden.

Weitere Funktionen beinhalten Rückgabe der PE-Nummer, Zahl aller PEs, Druckausgaben auf dem Syncer und Behandlung des Starts und des Endes des jeweiligen Programms. Außerdem gibt es noch nach Dringlichkeitsgraden geordnete Ausgaben zur Fehlersuche, die in einer dem Prozeß zugeordneten Protokolldatei aufgezeichnet werden.

Der Worker besteht aus einer Objektdatei mit den Routinen für die entsprechenden Aufrufe, die zum jeweiligen Modula-2*-Programm dazugebunden wird.

Die Aufrufe, die das Laufzeitsystem einem Modula-2*-Programm zur Verfügung stellt, sind

- Blockierendes Senden und Empfangen von Daten (*read, write*) durch die Worker selbst
- Direktes Lesen und Schreiben von Daten auf anderen PEs (*get, put*) mit Hilfe der Runner
- Synchronisation durch den Syncer
- Datenausgabe auf dem Syncer
- Ende-Anforderung und Warten auf das globale Ende
- Hilfsfunktionen wie Abbildung zwischen PE-Nummer und realer p4-Adresse sowie Debug-Ausgaben

B.2.3 Verteilung der Prozesse

Ein Programm, das auf n PEs verteilt laufen soll, benötigt n Software-PEs für die Ausführung der `FORALL`s, ein Software-PE für das Frontend und einen Syncer zum Starten und Synchronisieren.

Die physikalische Lokalisierung der Software-PEs ist beliebig. Um ein besseres Laufzeitverhalten zu erreichen, sollten sich verschiedene PEs natürlich auf verschiedenen physikalischen Rechnern befinden. Der Syncer läuft immer auf dem Rechner, auf dem das Programm gestartet wurde. Die Zuordnung der restlichen $(n+1)*2$ Prozesse (n Software-PEs und 1 Frontend mit je einem Worker und einem Runner) erfolgt nach Einträgen in einer entsprechenden Datei, die p4 für jedes verteilte Programm benötigt. Beim verteilten Debugger kommt nochmals ein Control-Prozeß für jedes PE und ein zentraler Master Debugger hinzu.

B.2.4 Abläufe

Bild B.2 gibt die Abläufe und Kommunikationsaufrufe wieder, die sich während einer parallelen Programmausführung zwischen den einzelnen Prozessen im Laufzeitsystem ergeben.

B.3 Messungen und Erläuterungen

Um die Laufzeiten des Systems festzustellen, wurden verschiedene Verteilungs-Konfigurationen eines Testprogramms durchgemessen.

Die Kosten für Kommunikation und Synchronisation sind im LAN sehr hoch. Zum Vergleich: In den 250ms, die eine Synchronisation ohne weiteres dauern kann, führt eine einfache SPARCstation ELC schon über 150 000 Fließkomma-Multiplikationen mit Zuweisung des Ergebnisses an eine Variable durch!

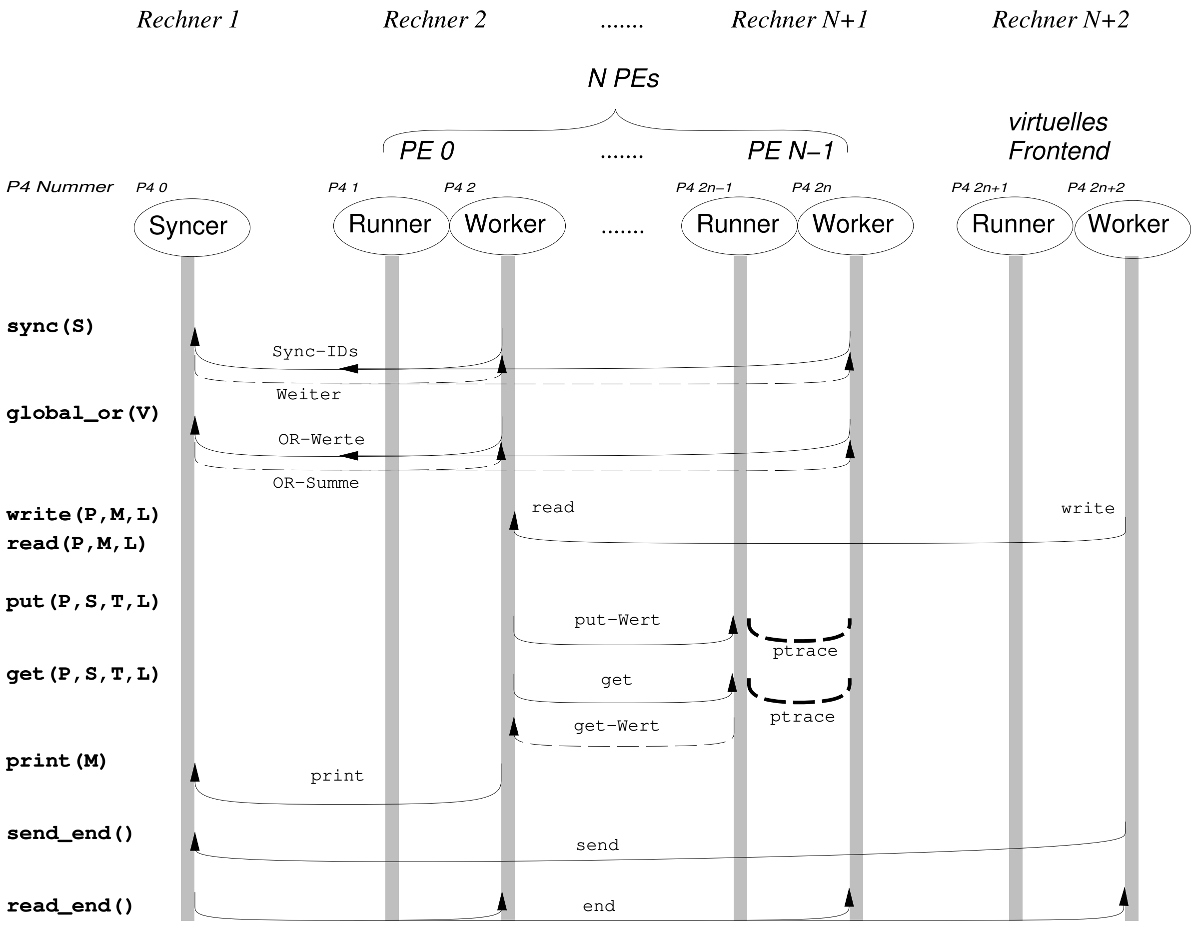


Abbildung B.2: Abläufe zwischen den Prozessen eines Modula-2*-Programms im Laufzeitsystem.

Interessant sind auch die relativen Kosten der einzelnen Kommunikationen in allen möglichen Verteilungsfällen.

Dazu wurden folgende Zeiten gemessen:

sync Zeit für eine Synchronisation, gemessen vom Aufruf des Sync-Befehls bis zur Fortsetzung. Diese Zeit ist wichtig für das Verhalten bei der Synchronisationen, die im Modula-2*-Programm auftreten. Der Engpaß liegt hier, wie schon oben erwähnt, beim Syncer.

circ Zeit für das Schicken einer Nachricht im Kreis über alle PEs von Abschicken bis Ankunft. Diese Zeit gibt einen Eindruck von der Leistungsfähigkeit des gesamten Nachrichtentransportsystems; die Last ist auf alle PEs verteilt.

uniR, uniW Zeiten für das unidirektionale Schicken einer Nachricht, gemessen wurden die Dauer für das Versenden auf einem PE (*uniW*) und das Empfangen auf dem anderen PE (*uniR*). Gesendet wurde jeweils zwischen zwei PEs.

loc Zeit für das lokale Senden und Empfangen einer Nachricht, d.h. Absenden an sich selbst, direkt gefolgt vom Empfang. Diese Zeit mißt den Overhead, den p4 direkt einführt.

get,put Zeit, um von einem andern PE einen Datensatz zu lesen (*get*) oder zu schreiben (*put*). Diese Zeiten messen die Geschwindigkeit, mit der Datenzugriffe vonstatten gehen.

Dabei wurden als Parameter die Anzahl der PEs und die Anzahl der verwendeten physikalischen Rechner variiert.

Es ergaben sich die in Tabelle B.1 aufgeführten Zeiten.[3]

Die Messungen erklären sich wie folgt:

1) Normalfall / Idealfall: Alle PEs sind auf physikalisch verschiedene Rechner verteilt. Die Synchronisation ist die aufwendigste Operation, da sie auf alle PEs warten muß. Beim Schicken im Kreis addieren sich Verzögerungen auf, während beim unidirektionalen Versenden immer nur zwei PEs betroffen sind. Get und Put sind relativ teuer, da sie das Anhalten, Auslesen und Neustarten eines Prozesses beinhalten.

2) Verteilung auf nur noch zwei physikalische Maschinen: Die Laufzeiten steigen allgemein, da jetzt Prozeßumschaltungen zwischen den einzelnen Programmen auf einem physikalischen Rechner nötig werden, die den Ablauf verlangsamen. Nur das Schicken von Nachrichten an sich selbst bleibt schnell, da es keinen Prozeßwechsel erfordert.

3) Volle Last auf einem einzigen physikalischen Rechner: Die Zeiten steigen nochmals um den Faktor 2. Besonders betroffen sind Put und Get, da sie zum Übertragen der Daten und zusätzlich zum eigentlichen Speicherzugriff durch den Runner einen Prozeßwechsel benötigen.

4) Nur noch zwei PEs und das Frontend, jeweils auf verschiedenen Maschinen: Wie erwartet halbiert sich der Aufwand zur Synchronisation und zum Senden im Kreis im Vergleich zu Versuch 1, da nur noch halb so viele PEs beteiligt sind. Die restlichen Zeiten sind unabhängig von der Gesamtzahl der PEs und ändern sich nicht wesentlich.

[3] Die anfangs verwendete Version 1.1 von p4 erreichte übrigens nur etwa ein Zehntel dieser Leistungen, da sie für die Kommunikation aktives Warten einsetzte und dadurch viel Rechenzeit und dauernde Prozeßwechsel benötigte.

Nr	Verteilung der PEs	S	F	1	2	3	4	*sync*	*circ*	*uniR*	*uniW*	*loc*	*get*	*put*
		Maschinen						Zeiten (in ms)						
1	4PEs+FE, alle auf verschiedenen Maschinen	A	A	B	C	D	E	19.5–32.5	3.03–5.72	0.98–1.67	0.93–1.65	0.053–0.113	4.98–5.67	6.86–6.08
2	4PEs+FE auf zwei verschiedenen Maschinen	A	A	B	A	B	A	200.0–205.6	34.8–52.5	0.76–1.07	1.23–1.77	0.053–0.161	26.8–173.3	13.7-135.0
3	4PEs+FE, alle auf derselben Maschine	A	A	A	A	A	A	426.8–437.7	88.9–121.0	3.93–4.47	3.40–3.88	0.052–0.054	216.1–285.6	287.2–328.9
4	2PEs+FE, alle auf verschiedenen Maschinen	A	A	B	C			9.8–21.2	1.81–3.28	0.62	1.01	0.053	4.0	4.9
5	2PEs+FE, alle auf derselben Maschinen	A	A	A	A			260.9–265.8	67.3–79.6	2.25	1.76	0.053	139.0	134.2

Tabelle B.1: Leistungsmessungen im Laufzeitsystem. Unter „Maschinen" ist die Zuordnung der Software-PEs (Frontend *F* und PE *1* bis *4*) und des Syncers *S* auf die physikalischen Maschinen *A* bis *E* angegeben.

5) Wieder alle Prozesse auf demselben Rechner: Durch die Prozeßwechsel steigen die Zeiten wie schon beim Übergang von Versuch 1 auf Versuch 2.

B.4 Ergebnisse

Durch das oben beschriebene Laufzeitsystem ist es jetzt auch möglich, Modula-2*-Programme verteilt auf einem Netzwerk von Workstations auszuführen.

Allerdings engt der hohe Kommunikationsaufwand die Art der Anwendungen ein, die im verteilten Fall eine Geschwindigkeitssteigerung bringen. Jedes synchronisationsintensive oder mit vielen Speicherzugriffen auf beliebig lokalisierte Daten belastete Programm verbringt den größten Teil seiner Ausführungszeit mit Warten auf Synchronisation und Nachrichtenaustausch. Außerdem kommt noch die Zeit zum Start des verteilten Programms hinzu, die im Bereich von 10–200 Sekunden liegt.

Sinnvoll einsetzbar ist das Laufzeitsystem also für Programme, die große, rechenintensive, grobkörnig parallele Probleme mit wenig Synchronisation lösen. Dazu gehören im experimentellen Bereich z.B. Anwendungen wie Raytracer und Mandelbrotberechnungen. Hier finden kaum Synchronisation und nichtlokaler Datenaustausch statt, und der Rechenaufwand überwiegt die Startzeit bei weitem.

Nicht geeignet sind hingegen feinkörnige Sortierverfahren und ähnliche Algorithmen. Hier ist eine einzelne Workstation um Größenordnungen schneller als das durch dauernde Kommunikation abgebremste verteilte System.

Anhang C

Einfügen der Debug-Funktionen

Im folgenden werden die Abbildungen von Modula-2* nach C – bzw. der C-ähnlichen Sprache, für die `msc` compiliert – definiert. Es werden jeweils die Sprachkonstrukte in Modula-2* und die von `domsc` erzeugten C-Konstrukte gegenübergestellt. Die eingefügten Aufrufe an Debug-Funktionen sind hervorgehoben.

Die Abbildungen gelten 1:1 für den sequentiellen und den parallelen Kontext. Im synchronen Kontext werden weitaus mehr Transformationen auf den Code angewandt. So werden z.B. bei einem `IF` zunächst die Aktivierungswerte aller Prozesse entsprechend der Bedingung gesetzt und dann anhand dieser Aktivierungen nacheinander `THEN` und `ELSE` abgearbeitet. Außerdem werden bei untereinander abhängigen Ausdrücken extensiv Hilfsvariablen verwendet.

Solche Besonderheiten würden die Abbildungen sehr unübersichtlich machen und werden daher im folgenden nicht mit aufgeführt. Der Aufruf der Debug-Funktionen erfolgt an derselben Stelle wie im sequentiellen Kontext, also nach der Initialisierung von Aktivierungswerten und ähnlichen Zuweisungen, beim Durchlaufen des Konstrukts und nach dessen Ende.

`IF` *Bedingung*
`THEN`
    Anweisung1
`ELSE`
    Anweisung2
`END`
	$\Longrightarrow$	`if (` *Bedingung* `) {`
		    `DebugFnStart(THEN,...);`
		    Anweisung1
		    `DebugFnUpdate(THEN,...);`
		    `DebugFnEnd(THEN,...);`
		`} else {`
		    `DebugFnStart(ELSE,...);`
		    Anweisung2
		    `DebugFnUpdate(ELSE,...);`
		    `DebugFnEnd(ELSE,...);`
		`};`
			[1]

[1] Der `else`-Teil kann natürlich auch entfallen.

```
CASE Var OF
    Wert1: Anweisung1 |
    ...
    WertN: AnweisungN
   ELSE Anweisung
END;
```

$\Longrightarrow$

```
switch (Var) {
    case Wert1:
        StartDebugFn(CASE,...);

UpdateDebugFn(CASE,...);
        Anweisung1;
        EndDebugFn(CASE,...);
        break;
    ...
    case WertN:
        StartDebugFn(CASE,...);
                                                [2] [3]
UpdateDebugFn(CASE,...);
        AnweisungN;
        EndDebugFn(CASE,...);
        break;
    default:
        StartDebugFn(CASE,...);

UpdateDebugFn(CASE,...);
        Anweisung;
        EndDebugFn(CASE,...);
        break;
}
```

```
WHILE Bedingung DO
    Anweisung
END;
```

$\Longrightarrow$

```
DebugFnStart(WHILE,...);
while ( Bedingung ) {
    DebugFnUpdate(WHILE,...);
    Anweisung
};
DebugFnEnd(WHILE,...);
```

```
REPEAT
    Anweisung
UNTIL Bedingung;
```

$\Longrightarrow$

```
DebugFnStart(REPEAT,...);
do {
    DebugFnUpdate(REPEAT,...);
    Anweisung
} while ( ! Bedingung );
DebugFnEnd(REPEAT,...);
```

[2] Diese weniger als elegante Anordnung der Debug-Funktionen ist nötig, weil bei einem `DebugFnUpdate` gleichzeitig auch die Profiling-Information aktualisiert wird, die beim entsprechenden `DebugFnStart` initialisiert wurde. Würde nur ein `Start` am Anfang des `select` stehen, so könnten die Durchläufe der einzelnen `case`-Teile nicht eigenen Zeilen zugeordnet werden.

[3] Der `ELSE`- bzw. `default`-Teil kann natürlich wieder entfallen.

```
LOOP
    Anweisung1
    ...Bedingung ...EXIT;
    Anweisung2
END;
```

$\Longrightarrow$

```
DebugFnStart(LOOP,...);
for(;;) {
    DebugFnUpdate(LOOP,...);
    Anweisung1
    ...Bedingung ...{
        goto Marke;
    };
    Anweisung2
} Marke: ;
DebugFnEnd(LOOP,...);
```

```
FOR Var = Von TO Bis DO
    Anweisung
END;
```

$\Longrightarrow$

```
DebugFnStart(FOR,...);
for (Var = Von; Var <= Bis ;
    Var += 1) {
    DebugFnUpdate(FOR,...);
    Anweisung
};
DebugFnEnd(FOR,...);
```

[4]

```
FORALL Var:[Von..Bis] IN
PARALLEL
    Anweisung
END;
```

$\Longrightarrow$

```
Initialisierung des FORALLs
DebugFnStart(PAR_FORALL,...);
Eventuelle Virtualisierungsschleife {

DebugFnUpdate(PAR_FORALL,...);
    Anweisung
};
DebugFnEnd(PAR_FORALL,...);
```

```
FORALL Var:[Von..Bis] IN SYNC
    Anweisung
END;
```

$\Longrightarrow$

```
Initialisierung des FORALLs
DebugFnStart(SYN_FORALL,...);
{

DebugFnUpdate(SYN_FORALL,...);
    Anweisung
};
DebugFnEnd(SYN_FORALL,...);
```

[4] Hierbei sind Sonderfälle wie Auf- und Abzählen, verschiedene Semantik bei *Von* > *Bis* etc. der Übersichtlichkeit halber nicht ausgeführt.

PROCEDURE *Name*(*Parameter*) [: *Ergebnis*]
BEGIN
...
END

$\Longrightarrow$

Ergebnis Name(*Parameter*)
Parameter-Deklarationen;
{
    DebugFnStart(FUNCALL,...);
    ...
    DebugFnUpdate(FUNCALL,...);
    DebugFnEnd(FUNCALL,...);
};

Anhang D

Modulführer

Dieser Anhang bietet eine Übersicht der im Laufe der Diplomarbeit implementierten Module und stellt ihre Schnittstellen in Form der C-Definitionen vor.

Im letzten Abschnitt werden zudem interessante Details der Implementierung angesprochen.

D.1 Die implementierten Module im Überblick

Im Laufe der Diplomarbeit wurden mehrere Module implementiert. Einige lassen sich sauber von der restlichen Arbeit trennen, etwa die verschiedenen Visualisierungen, andere hingegen sind eng mit einem Programm verwoben, wie etwa die Quellcodefenster mit dem Debugger. Die Gliederung in dieser Übersicht erfolgt daher nach den bei der Programmierung verwendeten Verzeichnissen, wobei es in einem Verzeichnis auch Referenzen auf ein anderes geben kann, z.B. die Einbindung der Visualisierung in den Debugger selbst.

Die genauen Zusammenhänge und Funktionen der einzelnen Module werden später im einzelnen erläutert. Hier soll zunächst einmal ein Überblick gegeben werden, um einen Kontext zu schaffen, in den die weiteren Details eingeordnet werden können.

Die Tabellen D.2, D.3, D.4 und D.5 zeigen die einzelnen Module. Es werden Angaben über die wichtigsten Dateien, die jeweils gebotene Funktionalität und die Größenverhältnisse gemacht. Die Dateien `counter.*` und `display.*` sind Weiterentwicklungen der in meiner Studienarbeit erstellten Funktionen. MSDB selbst ist ein stark modifizierter `xdbx`, bei dem ca. 60% des Codes im Rahmen der Diplomarbeit neu erstellt oder verändert wurden.

Insgesamt umfaßt die Diplomarbeit mit allen Quelltexten, Bibliotheken und ausführbaren Dateien – ohne die Dokumentation – ca. 12MB. Bei der Programmierung wurde auf Modularität und ausführliche Kommentierung geachtet. Als repräsentatives Beispiel wurden aus den Debug-Funktionen alle Kommentarzeilen entfernt und die resultierenden Größen mit den ursprünglichen verglichen.[1]

[1] Dazu wurden alle Kommentare einer Zeile mittels `sed` und mehrzeiligen Kommentare durch `awk` aus dem Quelltext entfernt und die Zeilen und Zeichen des Ergebnisses aufsummiert:

```
sed -e 's/[ \t]*\/\*[^$]*\*\///' | awk 'BEGIN{REM=0}{if ($1 ~ /\/\*/) REM=1; if (REM == 0)
print; if ($1 ~ /\*\//) REM=0}' | uniq | wc
```

Datei	Mit Kommentaren		Ohne Kommentare		Kommen-taranteil
	Zeilen	Bytes	Zeilen	Bytes	
DebugFn.c	2234	65324	1413	28836	55.9%
DebugFnV.c	919	28545	641	15856	44.5%
DebugFnV.h	62	1689	39	753	55.4%
DebugFn.h	277	9667	154	3805	60.6%

Tabelle D.1: Typisches Verhältnis von Code zu Kommentar

Verzeichnis	Programme	Beschreibung	Gesamtlänge
msdb	msdb.c global.h defs.h	Hauptprogramm des Modula-2*-Debuggers: Initialisie-rung der Grafikumgebung und des verwendeten UNIX-Debuggers (abgeleitet von xdbx), globale Variablen, Definitionen von Datenstrukturen	390 Zeilen 14405 Bytes
	calldbx.c dbx.c signals.c	Start und Kontrolle des externen Debuggers (hier derdbx), Absenden von Befehlen und Lesen von Rückmeldungen über ein pty, Behandeln von Signalen	509 Zeilen 15521 Bytes
	windows.c signs.c source.c swindow.c	X-Windows Fensterbehandlung, Anzeige des Quellcodes in verschiedenen Fenstern, Darstellung der Zeichen (Breakpoint etc.) im Quelltext, Zeilenabbildung zwischen C und Modula-2* sowie Variablenabbildung durch Zerteilen des C-Zwischencodes	2981 Zeilen 89164 Bytes
	parser.c regex.c *_regex.h	Zerteiler für die Auswertung der Rückmeldungen von dbx, reguläre Ausdrücke für verschiedene Maschinen, Zerlegung in Symbole	2801 Zeilen 83931 Bytes
	command.c handler.c	Behandlung der Knöpfe und Kommandotasten, Aufruf von Visualisierung / Profil / Baumdarstellung, Behandlung von zerteilten Rückmeldungen des dbx, Aktualisieren der Fenster	2886 Zeilen 83857 Bytes
	utils.c filemenu.c datadpy.c	Hilfsroutinen (Konkatenation von Zeichenketten etc.), Anzeige einer Dateiauswahl, Anzeigen von Datenstrukturen in eigenen Fenstern	1131 Zeilen 36121 Bytes
	apfel.c msdbtest.c ...	Testprogramme für den MSDB	953 Zeilen 26936 Bytes

Tabelle D.2: Modulübersicht – Der Debugger MSDB (für einen einzelnen Rechner)

Verzeichnis	Programme	Beschreibung	Gesamtlänge
`debugfn`	`DebugFn.h` `DebugFn.c`	Grundfunktionalität der Debug-Funktionen: ADTs Strukturbaum und Profiling-Tabelle, Ein– und Ausgabe, Konvertierung zwischen den ADTs und Strings zum Informationstransfer über ein `pty`	2511 Zeilen 74991 Bytes
	`DebugFnV.h` `DebugFnV.c`	Grafische Ausgabe der durch die Debug-Funktionen gesammelten Daten: Strukturbaum, Profile-Tabelle und Profile-Histogramm als X-Windows Widgets, sowie Funktionen zum Aktualisieren der angezeigten Informationen	981 Zeilen 30234 Bytes
	`testbaum.c` etc.	Programme zum Test der Debug-Funktionen und der grafischen Ausgabe	1121 Zeilen 30874 Bytes
	`libDebugFn.a` `libDebugFnV.a`	Bibliotheken mit den Debug-Funktionen, die zum zu untersuchenden Programm gebunden werden (einmal ohne Grafik, einmal mit)	41804 Bytes 106788 Bytes

Tabelle D.3: Modulübersicht – Debug-Funktionen zum Erfassen von Informationen im Zielprogramm

Verzeichnis	Programme	Beschreibung	Gesamtlänge
`visualizer`	`counter.h` `counter.c`	Zähler und Schiebebalken für die Visualisierung	1236 Zeilen 36771 Bytes
	`display.h` `display.c`	Grafische Anzeige eines Feldes (schwarz/weiß oder Graustufen) für die Visualisierung	1748 Zeilen 57464 Bytes

	`visual.c`
	`VisP.h`
	`Vis.h`
		Visualisierung: Erzeugen, Aktualisieren und Abbauen von Visualisierungen (für Vergleiche, Bereiche, Werte, benutzerdefinierte Funktionen und Aktivierungen), Ablauf im Vorder– oder Hintergrund, Bereitstellung von Widgets mit Visualisierern; dazugehörige Definitionen (privat und öffentlich)	4862 Zeilen
			167696 Bytes

	`testa.c`
	`testf.c`
	...
		Programme zum Test der Visualisierungen (Aktivierungen, Fließkommazahlen, Werte, Widgets, Kommunikation über Sockets etc.)	981 Zeilen
			26043 Bytes

	`libVis.a`	Bibliothek mit den verschiedenen Visualisierungen	318328 Bytes

Tabelle D.4: Modulübersicht – Visualisierer für Daten und Aktivierungen

Verzeichnis	Programme	Beschreibung	Gesamtlänge
`runtime`	`RunWork.h`	Gemeinsame Definitionen für alle Teile des Laufzeitsystems	280 Zeilen 9877 Bytes
	`Runner.c`	Laufzeitsystem für Modula-2*-Programme im LAN, put / get Kommunikation und direkte Speicherzugriffe beim Worker des PEs	533 Zeilen 16105 Bytes
	`Worker.c`	Zum Modula-2*-Programm dazugebundener Teil des Laufzeitsystems für ein PE, read / write Kommunikation	646 Zeilen 17672 Bytes
	`Syncer.c`	Globale Dienstleistungen: Synchronisierung, Ein / Ausgabe, Globales Oder, Start und Terminierung	443 Zeilen 13254 Bytes
	`Control.c`	Debug-Prozeß für ein PE: Externen Debugger (`gdb`) starten, Worker anhalten, Breakpoint setzen, Daten setzen und lesen, Zustand des Workers überwachen und melden, zentrale Befehle ausführen	2007 Zeilen 55071 Bytes
	`msdb`	Zentraler Debugger: Kontrolle der Control-Prozesse, Daten global zusammenfassen, Befehle an Prozesse verteilen, Anzeige über X-Windows, Benutzerschnittstelle (Abwandlung des sequentiellen MSDB, hier die Summe aller Quellcode-Dateien)	20911 Zeilen 645812 Bytes

	`procgroup.`
	`proto`
	`start.proto`
		Prototyp-Dateien für die Prozeßverteilung auf die realen Maschinen im Netz und den Startvorgang	73 Zeilen
			1760 Bytes

	`tif.c`
	`apfel.c`
	`...`
		Programme zum Test des Laufzeitsystems, der Kommunikation, der Anzeige verteilter Daten etc.	386 Zeilen
			9446 Bytes

Tabelle D.5: Modulübersicht – Laufzeitsystem für LANs

D.2 Debug-Funktionen

Es folgen die C-Definitionsdateien für die Debug-Funktionen. Davon gibt es zwei Versionen:

Die erste beinhaltet die Grundfunktionen für den Aufbau des Strukturbaums und der Profiling-Tabelle sowie Funktionen zum Informationstransfer der Datenstrukturen als Zeichenketten. Sie wird zum zu untersuchenden Programm dazugebunden und bringt nur wenig zusätzlichen Code mit sich.

Die zweite Version ergänzt die erste um Grafikfunktionen wie etwa der Anzeige von Strukturbäumen in einem eigenen Fenster und einem grafisch aufbereiteten Profile. Des weiteren beinhaltet sie Funktionen zum Aktualisieren der Anzeigen. Ihr Anwendungsgebiet liegt in Programmen, die die grafische Anzeige der Datenstrukturen verwenden, wie z.B. dem Debugger selbst. Durch die Einbindung von X-Windows ist diese Version im Objectcode deutlich größer als die Grundversion.

```
/*
 * DebugFn.h
 *
 * Headers for the debug functions
 *
 * Stefan Haenssgen  10-Jul-92
 *
 * Updates:
 *
 *      10-Jul-92       First rough implementation:
 *                      Definition of DebugNode
 *                      Definition of types and modes
 *      11-Jul-92       Removed modes - work with different calls instead
 *                      Added Depth of node in tree
 *                      Defined ProfileNode for profiling information
 *      12-Jul-92       Introduced context (sequential, parallel, synchronous)
 *      17-Jul-92       Dropped DN_IF
 *      07-Aug-92       Added Show*Widget() and Tree <-> List stuff
 *                      Introduced DebugFn_header to make double includes OK
 *      12-Aug-92       Added MColumn for the column in which the construct
 *                       starts, to be unique (line+type is not enough!)
 *                      Added MToLine & MToColumn to denote the position of
 *                       the end of the Modula-2* construct in the source
 *                      Also changed ProfileNode accordingly
 *      14-Aug-92       Added {profile,tree}_active to mark number of
 *                       profiles/trees being displayed
 *      24-Aug-92       Added GetLastNode()
 *      22-Oct-92       Created ADT ProfileTableType
 *      25-Oct-92       Added ProfileAsString()
 *      26-Oct-92       Split into DebugFn.h (basic stuff) + DebugFnV.h
 *                       (X11 / Visualizer specific stuff)
 *      27-Oct-92       Made ADT DebugTree
 *                      Introduced NumC & NumK (instead of checksum)
 *                      Updated Headers according to new source, including
 *                       parameters
 *      29-Oct-92       Updated AddListToTree() etc
 *      30-Oct-92       Added AdjustDepths()
 *      31-Oct-92       Made names more consistent ("List" -> "String" etc)
 *                      Replaced DebugFnUpdate by macro (faster!)
 *      04-Nov-92       Added MergedTree(), BuildAndMergeNodes(), CopyNode()
 *      06-Nov-92       Added GroupTree(), GroupSubTree(), JoinNodes()
 *      08-Nov-92       Added NumEntries to Profile Table
 *                      Introduced MakeProfileStat()
 *                      Added ProfileStatAsString()
 *      25-Nov-92       Added NodeAsString() and BuildNodeFromString()
 *                      Added PrintNodeAsString()
 *                      Added GimmeData() and DumpToFile() for data transfer
 *      29-Nov-92       Introduced DebugNode->ProfCount to directly update
 *                       profiling counts
 *                      Rewrote the macro accordingly (check for NULL)
 *      10-Jan-93       Added JoinInfoType to remember infos when joining nodes
 *      13-Jan-93       Added PE to DebugNode
 *                      Call CreateDebugTree with PE
 *      21-Feb-93       Fixed spelling
 *      23-Feb-93       Added AllGroup{Tree,SubTree}() and EquivSubTree()
 *      24-Feb-93       Added AllJoinNodes(), SetPrintStyle() and print
 *                       styles TPS_*
 *
 */

#ifndef DebugFn_header
```

```
#define DebugFn_header

#include <stdio.h>


/*
 * Type of debug node, e.g. for WHILE, IF, THEN, LOOP, FORALL etc
 */

#define DN_ROOT          1
#define DN_WHILE        10
#define DN_REPEAT       11
#define DN_LOOP         12
#define DN_FOR          13
#define DN_SYN_FORALL   20
#define DN_PAR_FORALL   21
#define DN_THEN         31
#define DN_ELSE         32
#define DN_CASE         33
#define DN_OF           34
#define DN_FNCALL       40

/*
 * Context of node
 */

#define DN_SEQ_CONTEXT  1
#define DN_PAR_CONTEXT  2
#define DN_SYN_CONTEXT  3


/*
 * Styles for tree output in PrintNode()
 *
 * Normal / pointers shown as adresses / full output
 */

#define TPS_NORMAL      0
#define TPS_POINTER     1
#define TPS_VERBOSE     2
#define TPS_SHORT       3

/*
 * Abbreviations
 */

#ifndef NULL
#    define NULL (void*)0L
#endif
#ifndef ulong
#    define ulong unsigned long
#endif


/*
 * JoinInfo:
 *
 * Information which would otherwise be lost when joining nodes
 *
 */
typedef struct JoinI {
    long        ID;                     /* ID of joined node            */
    long        RunCount;               /* Counter for the node         */
    int         PE;                     /* PE of this node              */
    struct JoinI*Next;                  /* Pointer to next info         */
} JoinInfo;


/*
 * DebugNode:
 *
 * One node in the tree built at runtime
 *
 */
typedef struct DebN {

    int         Type;           /* Type, e.g. DN_WHILE, DN_IF, ...      */
    int         Context;        /* Context, e.g. DN_PAR_CONTEXT, ...    */
    long        ID;             /* Unique ID number                     */
    int         Depth;          /* Depth of node in tree                */

    struct DebN *Parent;        /* Parent of this node                  */
    struct DebN *LBrother;      /* Next brother to left (ring list)     */
    struct DebN *RBrother;      /* Next brother to right                */
```

```
    struct DebN *Child;         /* First Child                          */

    JoinInfo    *Joined;        /* List of join infos (if any)          */
    long        MLine;          /* Line number on Modula-2* Source      */
    long        MColumn;        /* Dito Column                          */
    long        MToLine;        /* Corresponding ending line of..       */
    long        MToColumn;      /* ..Modula-2* construct                */
    long        ActValue;       /* Current activation value             */
    void        *ActArray;      /* Array of activation values           */

    void        *RangeVar;      /* Variable e.g. of FOR or FORALL       */
    long        RangeLo;        /* Range of Variable e.g. for FORALL    */
    long        RangeHi;        /* Dito                                 */

    long        RunCount;       /* Profiling information                */

    long        *ProfCountP;    /* Pointer to counter in Profile Table  */

    int         PE;             /* PE we're on                          */

} DebugNode;


/*
 * DebugTree:
 *
 * Tree of Debug Nodes representing the dynamic call structure of
 * the program
 *
 */
typedef struct {
    DebugNode   *RootNode;      /* Root of the structure tree           */
    DebugNode   *LastNode;      /* Most recently created node           */
    long         ID;            /* Unique ID for new nodes              */
    long         NumC, NumK;    /* Counters for create/kill (use as a   */
                                /*  checksum: Change when tree changes) */
} DebugTree;


/* Size of hash table and hash function */

#define P_TABLE_SIZE    997
#define phash(x) (x % P_TABLE_SIZE)


/*
 * ProfileNode:
 *
 * One node in profile information list
 *
 */
typedef struct ProN {
    int         Type;           /* Type, e.g. DN_WHILE, DN_IF, ...      */
    int         Context;        /* Most recently used context for this  */
    struct ProN *NextNode;      /* Chain of nodes in one Hash Entry     */
    long        MLine;          /* Line number on Modula-2* Source      */
    long        MColumn;        /* Dito Column                          */
    long        RunCount;       /* Profiling information                */
    long        MToLine;        /* Corresponding ending line of..       */
    long        MToColumn;      /* ..Modula-2* construct                */
} ProfileNode;



/*
 * ProfileTable:
 *
 * One Profile Table with hashed node pointers and line range
 *
 */
typedef struct {
    ProfileNode *ProfileTable[P_TABLE_SIZE];    /* Hash table           */
    long        maxProfileLine;                 /* Line range in Table  */
    long        NumEntries;                     /* Number of entries    */
} ProfileTable;


extern DebugTree        *DTree;         /* Global Debug Structure Tree  */
extern ProfileTable     *PTable;        /* Global Profile Info Table    */

extern int tree_active;         /* Number of trees being displayed now  */
extern int profile_active;      /* Dito: # of currently shown profiles  */
```

```
/*
 * Debug functions visible to user
 */


/* DebugNode, DebugTree: Dynamic call structure tree of program */

extern DebugNode   *NewNode(/*tree,type,parent,         * Create new debug   *
                     mline,mcolumn,mtoline,mtocolumn,   *  information node  *
                     actvalue,actarray,                 *  (internal use!)   *
                     rangevar,rangelo,rangehi*/);
extern DebugNode   *CopyNode(/*tree,par,node*/);/* Copy contents below parent*/
extern void         SetPrintStyle(/*style*/);   /* Set print style for node  */
extern void         PrintNode(/*tree,node*/);   /* Print one node            */
extern void         PrintSubTree(/*tree,node*/);/* Print subtree             */
extern void         PrintTree(/*tree*/);        /* Print whole tree          */
extern DebugTree   *CreateDebugTree(/*PE*/);    /* Create a debug info tree  */
extern DebugNode   *GetLastNode(/*tree*/);      /* Return most recent node   */
extern DebugNode   *AdjustDepths(/*node,sdpt*/);/* Recompute depths w. start */
extern long         GetChecksum(/*tree*/);      /* Changes when tree changes */
extern char        *NodeAsString(/*node*/);     /* String for one node       */
extern void         PrintNodeAsString(/*node*/);/* Print this string         */
extern char        *TreeAsString(/*tree,node*/);/* String describing tree    */
extern void         PrintTreeAsString(/*tree,node*/);/* Print this string    */
extern DebugNode   *BuildNodeFromString(/*s*/); /* Build one node from data  */
extern DebugTree   *BuildTreeFromString(/*s,sID*/);/*Reconstruct tree from s.*/
extern DebugTree   *AddStringToTree(/*t,s,snod*/);/* Merge tree with tree-str*/
extern int          KillNode(/*tree,node*/);    /* Destroy one node          */
extern int          KillSubTree(/*tree,node*/); /* Destroy a subtree         */
extern int          KillTree(/*tree*/);         /* Destroy whole tree        */
extern void         BuildAndMergeNodes(/*tree,parent,
                     node1,node2*/);            /* Recurs.merge subtree nodes*/
extern DebugTree   *MergedTree(/*tree1,tree2*/);/* Merge trees (group nodes) */
extern void         JoinNodes(/*n1,n2*/);       /* Merge infos recursively   */
extern char         EquivSubTree(/*n1,n2*/);    /* Test if subtrees are equiv*/
extern void         GroupSubTree(/*node*/);     /* Group nodes in subtree    */
extern void         GroupTree(/*tree*/);        /* Group infos in tree       */
extern void         AllJoinNodes(/*n1,n2*/);    /* Merge infos, kill n2      */
extern void         AllGroupSubTree(/*node*/);  /* Dito, but only test equiv.*/
extern void         AllGroupTree(/*tree*/);     /*  of nodes, not subtrees!  */


/* ProfileNode, ProfileTable: Profiling information table for program */

extern ProfileTable*CreateProfileTable();       /* Create empty profile table*/
extern void         AddProfileNode(/*pt,dnode*/);/* Add inf of DbgNode to tab*/
extern void         KillProfileNode(/*pt,line,column*/);/* Destroy one entry */
extern void         ClearProfileTable(/*pt*/);  /* Destroy all entries       */
extern void         PrintProfileTable(/*pt*/);  /* Print profile Table       */
extern char         *ProfileAsString(/*pt*/);   /* String describing Table   */
extern void         PrintProfileAsString(/*pt*/);/* Print this string        */
extern void         AddStringToProfile(/*pt,in,replace*/); /* Merge str & tab*/
extern ProfileNode *GetProfileNode(/*pt,line,column*/);  /* Find one node    */
extern ProfileNode**MakeProfileStat(/*pt*/);    /* Sorted array of nodes     */
extern char        *ProfileStatAsString(/*pt,nlines*/);  /* This array as str*/
extern void         KillProfileTable(/*pt*/);   /* Destroy whole table       */


/* DebugFnXXX: Debug Functions, record dynamic structure of program in  */
/*             DebugTree while also collection Profile Information      */

extern void         DebugInitialize();  /* Init debug function environment   */
extern void         DebugEnd();         /* Close & free environment          */
extern void         DebugFnStart(/*type,                * Start a new        *
                       mline,mcolumn,mtoline,           *  block             *
                       mtocolumn,actvalue,actarray,
                       rangevar,rangelo,rangehi*/);

/* Update: Increment RunCount of node and also counter in */
/*        Profile Table entry, if defined                */

#define DebugFnUpdate(type,mline,mcolumn,mtoline,mtocolumn,actvalue,       \
                      actarray,rangevar,rangelo,rangehi)                   \
                      {long*pc=DTree->LastNode->ProfCountP;                \
                       (DTree->LastNode->RunCount)++;if(pc)(*pc)++;}

extern void         DebugFnEnd(/*type,                  * Close the block   *
                       mline,mcolumn,mtoline,           *  and remember     *
                       mtocolumn,actvalue,actarray,     *  profile data etc *
                       rangevar,rangelo,rangehi*/);
```

```
extern int GimmeData(/*pid,buf,adr,len*/); /* Transfer data via ptrace()    */
extern int DumpToFile(/*adr,len,fname*/);  /* Transfer via seperate file    */


#endif /* DebugFn_header */
```

```
/*
 * DebugFnV.h
 *
 * Headers for the debug functions, X11 / Visualizer specific stuff
 *
 * Stefan Haenssgen  10-Jul-92
 *
 * Updates:
 *
 *      26-Oct-92       Split into DebugFn.h (basic stuff) + DebugFnV.h
 *                       (X11 / Visualizer specific stuff)
 *      08-Nov-92       Introduced ProfileGraphWidget()
 *      29-Nov-92       Added UpdateProfileGraph(), renamed ProfileGraphWidget
 *                       to ShowGraphWidget
 */

#ifndef DebugFnV_header
#define DebugFnV_header

#include "DebugFn.h"

#include "Vis.h"

#include <X11/Xlib.h>
#include <X11/StringDefs.h>
#include <X11/Shell.h>
#include <X11/Xaw/Cardinals.h>
#include <X11/Xaw/Label.h>
#include <X11/Xaw/Command.h>
#include <X11/Xaw/Tree.h>
#include <X11/Xaw/AsciiText.h>
#include <X11/Xaw/AsciiSrc.h>
#include <X11/Xaw/Box.h>
#include <X11/Xaw/Paned.h>
#include <X11/Xaw/Viewport.h>

extern char     *_D_TypeString[];       /* Names of types       */
extern char     *_D_ContextString[];    /* Names of contexts    */


/* Additional functions available to the user: */



extern Widget   ShowTreeWidget(/*parent*/);     /* Widget containing a tree  */
extern void     UpdateTreeWidget(/*popup*/);    /* Update shown tree         */

extern Widget   ShowProfileWidget(/*parent*/);  /* Widget with profile infos */
extern void     UpdateProfileWidget(/*popup*/); /* Update shown infos        */

extern Widget   ShowGraphWidget(/*par,pt*/);    /* Widget with profile graph */
extern void     UpdateGraphWidget(/*pop,pt*/);  /* Update shown graph        */

extern void     ShowTree();                 /* Display debug tree & profile infos*/
                                            /* (does its own X event loop etc!)  */

extern void     DebugVisualizeActives(  /* Visualize activation values       */
                   /* AVector,AValue,
                   from,to,name,namelen*/);


#endif /* ifndef DebugFnV_header */
```

D.3 Visualisierungen

Hier werden die öffentlichen Funktionen der Visualisierung aufgeführt. Die privaten Definitionen, die die Visualisierung intern verwendet, sind davon abgekapselt.

Die Anwendung der zur Verfügung stehenden Funktionen wird im Kommentar ausführlich erläutert.

```
/*
 * Vis.h
 *
 * Public definitions for Array Visualizer
 *
 * Stefan Haenssgen 28-Nov-91
 *
 * Updates :    28-Nov-91  Basic Implementation
 *              28-Dec-91  Structured redesign, formed an ADT out of
 *                          the single data structures
 *              30-Dec-91  Added new Widgets "visselect" and "vs_*"
 *                          for user selection of conditions
 *                         Added "vactive{1,2}" for easy access to
 *                          currently active layers
 *              09-Jan-92  Enhanced the documentation
 *                         Changed the interface: Just one call to
 *                          "visualize()" instead of two calls & X Windows init
 *              10-Jan-92  Added generic comparison function "vcmpfn()" and
 *                          array access function "vaccfn()"
 *                         Added new interface "visualize_userfn()"
 *                         Added vrunning for termination check
 *                         Added "{open,close}_visualizer()" and "VIStype"
 *              16-Jan-92  Renamed functions for easier Modula interfacing:
 *                              open_visualizer    -> OpenVisualizer()
 *                              visualize()        -> Visualize()
 *                              visualize_userfn() -> VisualizeUserFn()
 *                              close_visualizer   -> CloseVisualizer()
 *              19-Jan-92  Added Widgets for "from-to range": vftomto, vft_*
 *                         Added Values for range: vfromval, vtoval
 *              28-Jan-92  Added support for sockets and fork()ing:
 *                          VISvis  in VIS,
 *                          vforked, vsocketname, vwritesocket  in visual.type
 *                          also: vbindsocket, vreadsocket
 *              29-Jan-92  Updated documentation
 *              30-Jan-92  Also include time.h for selection timeouts
 *                         Added UPDATE_INTERVAL for regual checks on socket
 *              04-Feb-92  Split visual.h into Vis.h (public) and VisP.h
 *                          (private)
 *                         Made VIS.VISvis void* (instead of visualtype *)
 *              05-Feb-92  Added example usage for library libVis.a
 *                         Added ChangeUserFn()
 *              08-Feb-92  Added coordinates as parameter to OpenVisualizer()
 *                         Added note for one-dimensional arrays
 *              09-Feb-92  libm.a necessary for compilation!
 *              10-Feb-92  Second sync flag for function changes
 *              16-Apr-92  New Visualizer for array values (not only boolean)
 *                         Updated function documentation
 *              03-Jul-92  Added V_FLOAT and V_DOUBLE for floating point
 *                          support
 *              07-Aug-92  Introduced Vis_header to make double includes OK
 *              27-Aug-92  Added VisualizeActivation()
 *              24-Oct-92  Made dimensions (0..n-1 == n) clearer
 *              24-Nov-92  Introduced new calls to utilize Visualizers
 *                          as widgets, not as 100% standalone X applications
 *                          (VisRangeWidget() etc)
 *
 *
 * Example for usage for libVis.a:
 *
 *      cc -o foo foo.o libVis.a -lXaw -lXmu -lXt -lXext -lX11 -lm
 *
 */


/*
 *  Description of functions:
 *  -------------------------
 *
 *
 *  VIStype *OpenVisualizer(int x, int y)
 *
 *  Initializes the X Window basics to make further visualization possible.
 *  The return value is later needed for visualization and finally for
```

```
*  "CloseVisualizer()"
*  "x" and "y" specify the coordinates of the upper left corner of the
*  Visualizer Window when it appears. If "x" or "y" are less than zero,
*  the Visualizer will be positioned with user interaction (unless the
*  user's window manager places windows automatically, anyway).
*
*
*  void CloseVisualizer(VIStype *VIS)
*
*  Destroys the X Window basics connected with VIS.
*  Also kill off background process if there is one.
*
*
*
*  void  Visualize(  VIStype *VIS,
*                    char    *name,
*                    ulong    namelen,         ( unsigned long )
*                    void    *adr,
*                    int     *dims,
*                    ulong    numdims,         ( unsigned long )
*                    int      type,
*                    int      condition,
*                    void    *comparison,
*                    Boolean  forkflag,
*                    Boolean  syncupdateflag)
*
*  Creates a visualization of the array with given starting address
*    "adr" and name "name" of the dimensions [0..dims[i]-1] (whose
*    number is given in "numdims"). The array is of type "type".
*    The length of the name string is passed in "namelen" for compati-
*    bility with Modula calls. It is ignored if it is less than 1.
*  All array elements fulfilling the comparison defined by "condition"
*    and "comparison" are displayed in black, the rest in white.
*  "VIS" is the VIStype* returned by "OpenVisualizer".
*  If "forkflag" is TRUE, the Visualizer will be put to work in the
*    background; the displayed data can be updated using a call to
*    UpdateForkedVisualizer()
*  If "syncupdateflag" is TRUE, a call to UpdateForkedVisualizer() will block
*    until the user presses the sync-button of the Visualzier.
*
*
*  Example call:
*
*    Visualize(VIS,"a", 1, dims, numdims,
*              V_INT, V_GREATER, &value, FALSE, FALSE)
*
*  will display the visualization of the integer array a and highlight
*    all elements that are greater than "value", running in the foreground
*    (i.e. main program being suspended until Visualizer is terminated)
*  Of course the sync-flag is not set here, because there is no background
*    process at all.
*
*
*  void  VisualizeRange(  VIStype *VIS,
*                         char    *name,
*                         ulong    namelen,         ( unsigned long )
*                         void    *adr,
*                         int     *dims,
*                         ulong    numdims,         ( unsigned long )
*                         int      type,
*                         void    *fromvalue,
*                         void    *tovalue,
*                         Boolean  forkflag,
*                         Boolean  syncupdateflag)
*
*  Creates a visualization of the array with given starting address
*    "adr" and name "name" of the dimensions [0..dims[i]-1] (whose
*    number is given in "numdims"). The array is of type "type".
*    The length of the name string is passed in "namelen" for compati-
*    bility with Modula calls. It is ignored if it is less than 1.
*  All array elements in the range "fromvalue" to "tovalue"
*    are displayed in black, the rest in white.
*  "VIS" is the VIStype* returned by "OpenVisualizer".
*  If "forkflag" is TRUE, the Visualizer will be put to work in the
*    background; the displayed data can be updated using a call to
*    UpdateForkedVisualizer()
*  If "syncupdateflag" is TRUE, a call to UpdateForkedVisualizer() will block
*    until the user presses the sync-button of the Visualzier.
*
*  Example call:
*
*    VisualizeRange(VIS,"a", 0, a, dims, numdims, V_INT, &i, &j,
*                   TRUE, FALSE)
*
```

```
*  will display the visualization of the integer array a and highlight
*    all elements in the range [i..j] running in the background (i.e. the
*    main program continues and is able to update the array using
*    UpdateForkedVisualizer(VIS) ), not synchronizing the update calls.
*    Then length of the name (i.e. "a") is computed internally since the
*    value given (i.e. 0) is less than 1.
*
*
*  void  VisualizeUserFn(  VIStype *VIS,
*                          char    *name,
*                          ulong    namelen,          ( unsigned long )
*                          void    *adr,
*                          int     *dims,
*                          ulong    numdims,          ( unsigned long )
*                          int      type,
*                          char    *cmptext,
*                          ulong    cmptextlen,       ( unsigned long )
*                          Boolean  (*compare)(),
*                          void    *(*access)(),
*                          void    *envp,
*                          ulong    envlen,           ( unsigned long )
*                          Boolean  forkflag,
*                          Boolean  syncupdateflag,
*                          Boolean  syncfunctionflag)
*
*  Creates a visualization of the array with given starting address
*    "adr" and name "name" of the dimensions [0..dims[i]-1] (whose
*    number is given in "numdims"). The array is of type "type".
*    The length of the name string is passed in "namelen" for compati-
*    bility with Modula calls. It is ignored if it is less than 1.
*    (the same applies to "cmptextlen")
*  "VIS" is the VIStype* returned by "OpenVisualizer".
*  "envp" points to some user data that may be used in the compare
*    function later on. The length of the data is given in "envlen".
*  The user data is newly allocated and copied so changing it after-
*    wards has no effect. Use ChangeUserFn() to alter it.
*  If "forkflag" is TRUE, the Visualizer will be put to work in the
*    background; the displayed data can be updated using a call to
*    UpdateForkedVisualizer()
*  If "syncupdateflag" is TRUE, a call to UpdateForkedVisualizer() will block
*    until the user presses the sync-button of the Visualzier.
*  If "syncfunctionflag" is TRUE, a call to ChangeUserFn() will block.
*
*  The "cmptext" describes the comparison. It has to contain one
*    occurence of "%s" which is automagically replaced by the array name
*    and the actual coordinates.
*  The user defined comparison and array access functions are used as
*    follows:
*
*  Boolean compare(void *a, int cond, void *envp)
*
*      Compares the value of the array element (given in "a") to some
*        other value, possibly given by the environment pointer "envp"
*        (which is passed at the 1st call to VisualizeUserFn() and may
*        be changed using ChangeUserFn() ). The comparison condition
*        (V_EQUAL, V_LESS etc) is passed in "cond".
*      The user function is free to ignore "cond" as well as "envp"
*        and perform whatever operation it likes.
*
*  void   *access(void *adr, int *x, int *dims, ulong numdims)
*
*      Returns a pointer to the array element at adr[x[0],x[1],...]
*        where the number of dimensions is in numdims and the size of
*        each dimension is given in dims[].
*
*  All array elements fulfilling the comparison defined by the user
*    supplied functions are displayed in black, the rest in white.
*
*  Example call:
*
*    VisualizeUserFn(VIS, "a", 0, a, dims, numdims, V_INT,
*                    "10 <= %s <= 20", 0, my_compare, my_access,
*                    &data, sizeof(data), TRUE, TRUE, TRUE)
*
*  will display the visualization of the integer array a and highlight
*    all elements that fulfill the comparison function "my_compare"
*    using the data in "data" and are accessed via "my_access",
*    running in the background (i.e. the main program continues and is able
*    to update the array using UpdateForkedVisualizer(VIS))
*  The Visualizer is synchronized with the foreground process for both
*    UpdateForkedVisualizer() and ChangeUserFn() calls.
*  The length of both the name of the array and the string describing
*    the user function are computed internally since they are both <1.
*
```

```
*
*
*  void  VisualizeValues(  VIStype *VIS,
*                          char    *name,
*                          ulong    namelen,          ( unsigned long )
*                          void    *adr,
*                          int     *dims,
*                          ulong    numdims,          ( unsigned long )
*                          int      type,
*                          Boolean  forkflag,
*                          Boolean  syncupdateflag)
*
*  Creates a visualization of the array with given starting address
*    "adr" and name "name" of the dimensions [0..dims[i]-1] (whose
*    number is given in "numdims"). The array is of type "type".
*    The length of the name string is passed in "namelen" for compati-
*    bility with Modula calls. It is ignored if it is less than 1.
*  The elements of the array are displayed using grayscale bitmaps,
*    thus giving a quick overview of the distribution of values in
*    the array. However, no boolean operators are used.
*  "VIS" is the VIStype* returned by "OpenVisualizer".
*  If "forkflag" is TRUE, the Visualizer will be put to work in the
*    background; the displayed data can be updated using a call to
*    UpdateForkedVisualizer()
*  If "syncupdateflag" is TRUE, a call to UpdateForkedVisualizer() will block
*    until the user presses the sync-button of the Visualzier.
*
*
*  Example call:
*
*    VisualizeValues(VIS,"a", 1, a, dims, numdims, V_INT, FALSE, FALSE)
*
*  will display the visualization of the integer array a and display
*    all elements as different shades of gray according to their value.
*    No background operation (and thus no sync) is initiated.
*
*
*
*  void  VisualizeActivation(  VIStype *VIS,
*                              char    *name,
*                              ulong    namelen,
*                              void    *adr,
*                              int     *dims,
*                              void    *comparison,
*                              void     (*callfn)(),
*                              Boolean  forkflag,
*                              Boolean  syncupdateflag)
*
*  Creates a visualization of the 1-dimensional array of activation values
*    (longs) with given starting address "adr" and name "name" of the
*    size [0..dims[0]-1]. The length of the array is adjusted so that
*    it fits into a square (but the additional elements are never referenced
*    so no need to worry : function to be called is pointed to by "fn",
* the string describing the function is "str", "dat" points to the
* user data to be transmitted and "dlen" is the length of the user data.
* "str" has to contain one occurence of "%s" which is automatically replaced
* by the name of the array being visualized. Its length is given in "strl"
* which is ignored if it is less than one.
*
* If "fn" is NULL then no function is given and the old one is used further.
* If "str" is NULL the string isn't changed and if "dat" is NULL, no
* data is transmitted.
*
* This function requires that the Visualizer uses a User defined Function
* (i.e. was called using VisualizeUserFn()) and is running in the background.
*
*
*
* void UpdateForkedVisualizer(VIStype *VIS)
*
* Transmit the new contents of the visualized array to the Visualizer
* running in the background
*
*
**
**   NOTES
**   =====
**
** - One-dimensional arrays are now supported, too. To visualize them
**   they are shown as 2-dimensional arrays with the dimension (m,n)
**   where m*n = l and l = size of the one-dimensional array. The Visualizer
**   tries to keep m and n as close to sqrt(l) as possible. However, in
**   the case of a prime length the display may degenerate to 1*l.
**   Note also that an array with n elements uses indices 0..n-1 so
```

```
 **   m*n will try to fit n-1 ! (??? is that ok ???)
 **   Exception: VisualizeActivation() always generates a square display
 **              by stuffing the given array till it fits
 **
 ** - One VIS struct can only be used once at a time, so to use multiple
 **   concurrent Visualizers, you'll have to do an "OpenVisualizer()" for
 **   each of them. After a Visualizer has exited, however, its VIS can
 **   be reused, i.e. sequentially used Visualizers (not in the background)
 **   can use the same VIS.
 **
 ** - Since the Visualizer does its own X Event Loop, it only returns
 **   when the "quit" button is pressed (if it isn't put in the background)
 **
 ** - If you have problems with "undefined stdout" etc, include <stdio.h>
 **   before Vis.h
 *
 *
 * New standalone calls for Widget creation:
 *
 *
 * VIStype *VisWidget(parent, name, namelen, adr, dims, numdims,
 *                    type, condition, comparison)
 *
 * Creates a normal Visualizer Widget with the given parent Widget,
 * returns the complete Visualizer data structure associated with it.
 * Same parameters as Visualize() but no fork/syncupdate flags, no
 * VIS parameter, however an additional "parent" (which must be
 * already initialized!)
 *
 *
 * VIStype *VisRangeWidget(parent, name, namelen, adr, dims, numdims,
 *                    type, fromvalue, tovalue)
 *
 * Dito with Visualizer type "Range"
 *
 *
 * VIStype *VisUserFnWidget(parent, name, namelen, adr, dims, numdims,
 *                    type, condition, comparison)
 *
 * Dito with Visualizer type "User Function"
 *
 *
 * VIStype *VisValuesWidget(parent, name, namelen, adr, dims, numdims,
 *                    type, condition, comparison)
 *
 * Dito with Visualizer type "Range"
 *
 *
 * VIStype *VisActivationWidget(parent, name, namelen, adr, dims,
 *                              comparison, callfn)
 *
 * Dito with Visualizer type "Activations"
 *
 *
 * void UpdateVisWidget(VIStype *VIS)
 *
 * Update a Visualizer's display when the data it's showing have changed
 *
 *
 * NOTE: If one of these Visualizer Widgets is quit, its VIS->VISvis
 *       gets changed to NULL, so it's possible to tell if it has exited
 */

#ifndef Vis_header
#define Vis_header

/* Necessary definitions for X Windows & Maths*/

#include <X11/Intrinsic.h>


/* Definitions of type-tokens */

#define V_CHAR          0
#define V_INT           1
#define V_LONG          2
#define V_FLOAT         3
#define V_DOUBLE        4


/* Definitions of comparison operators */

#define V_GREATER       0
#define V_EQUAL         1
```

```
#define V_LESS          2


/* Common context for visualizer */

typedef struct {
  XtAppContext  VISapp_con;     /* Application Context (X Windows)      */
  Widget        VISparent;      /* Main parent Widget                   */
  void          *VISvis;        /* Visualizer being used (or NULL)      */
} VIStype;


/* For shorter headers :-) */

#ifndef ulong
#    define ulong unsigned long
#endif


/* Declarations of functions that the Visualizer offers */

#ifndef VISUAL_ITSELF
extern VIStype          *OpenVisualizer();
extern void              Visualize();
extern void              VisualizeUserFn();
extern void              VisualizeRange();
extern void              VisualizeValues();
extern void              VisualizeActivation();
extern void              CloseVisualizer();
extern void              UpdateForkedVisualizer();
extern VIStype          *VisWidget();
extern VIStype          *VisRangeWidget();
extern VIStype          *VisUserFnWidget();
extern VIStype          *VisValuesWidget();
extern VIStype          *VisActivationWidget();
extern void              UpdateVisWidget();
#endif

#endif /* Vis_header */
```

D.4 Laufzeitsystem

Diese Definitionsdatei beinhaltet die dem Worker zur Verfügung gestellten Funktionen zur Kommunikation. Außerdem sind die verschiedenen Nachrichtentypen, die das Laufzeitsystem verwendet, aufgeführt.

```
/*
 * RunWork.h
 *
 * Runtime System for Modula-2* on a network of workstations, based on P4
 *
 * Headers: Common functions, global variables
 *
 *
 * Stefan Haenssgen
 *
 * 21-sep-92    Created
 *              Defined message types for slave/runtime/master
 * 22-sep-92    Introduced procinfo_s to contain all process information
 *              New architecture: seperated Runner & Worker, no more
 *               procinfo_s etc necessary
 * 23-sep-92    Added Logfile name
 *              First real communication
 *              Added communication calls as defined by Paul
 * 24-sep-92    Added queues (qentry)
 *              YAR (yet another redesign): No Queues, direc exchange of
 *               read/write between Workers, Sync server (P4 ID 0)
 *              Changed tokens accordingly
 *              Added message buffer handling functions
 * 27-sep-92    Redefined Tokens for put/get
 *              Lowercase all communication functions
 * 28-sep-92    Added M_PRINT for direct output from the Syncer
 *              Added _do_print_() to call this from a Worker
 *              Defined DEBUGPR & DEBUGF to mark debugging output clearly
 * 29-sep-92    Enclosed in #ifndef's
 * 30-sep-92    "Official Version", moved logfiles to /tmp, renamed them
 * 05-oct-92    Renamed functions to be more unique (prefix "_R_")
 * 06-oct-92    Added Global Or and Tokens S_SEND_OR and S_RECV_OR
 * 07-oct-92    Replaced logfile by p4_dprintfl()
 *              Removed open_log, logfile name etc pp
 *              Introduced DBGPRINT as generic macro for outputting debug info
 * 08-oct-92    Reworked DBGPRINT, use varargs
 *              Introduced _R_dbg_lvl
 * 09-oct-92    Introduced new tokens for remote debugging (M_ATTACH_DBG etc)
 * 13-oct-92    Introduced _read_end_() to handle termination for FE
 * 16-oct-92    Added _do_printf_() for vararg-printing
 * 04-nov-92    Introduced S_NAME to get filename from worker
 * 11-nov-92    Introduced S_START to continue Worker from Runner
 * 12-nov-92    Added M_DBG_STOP to stop the running child process
 * 20-nov-92    Added M_DBG_CMD_QUERY to exec command & wait for output
 * 21-nov-92    Added S_NEXT_TYPE for direct Runner/Worker put/get
 * 22-nov-92    Introduced struct GetPutInfo for ptrace from Worker to Runner
 * 09-jan-93    Added M_DISPLAY to get display name from Syncer to Master
 * 17-jan-93    Added D_xxx for master debugger requests and C_xxx for
 *               replies of the control processes
 * 23-jan-93    Added C_POSITION for source code position reports
 * 24-jan-93    Added [CD]_WHATIS and [CD]_ADR for type & address info
 *              Added [CD]_GET_BLOCK
 * 03-feb-93    Added C_BROKEN and C_FAULTED to report problems to M-Dbg
 * 06-feb-93    Added program wait states (WAIT_READ etc)
 *              Added C_NEW_STATE to notify Master of state changes
 *              Added _statename_() for type of wait state of Worker
 *
 *
 */
/*
 * Workers & Runners are started using the following layout:
 *
 *      procgroup entry                 P4ID    PE#     Job             Host
 *
 *      local 0                          0       -      Syncer           A
 *      B 1 Runner                       1       -      Runner (for 2)   B
 *      B 1 Worker                       2       0      Worker (for 1)   B
 *      ...                             ...     ...     ...             ...
 *      Z 1 Runner                      n-1      -      Runner (for n)   Z
 *      Z 1 Worker                       n      n/2-1   Worker (for n-1) Z
 *
 *                                      (n = total number of slaves)
 */

#ifndef _RUN_WORK_H_
```

```
#define _RUN_WORK_H_

#include <p4.h>
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>


/* Communication calls from program inside worker */

extern int  _write_(/*PE,msg,len*/);    /* Send message to other PE     */
extern int  _read_(/*PE,msg,len*/);     /* Receive message from other PE*/
extern int  _put_(/*PE,src,tgt,len*/);  /* Write memory on other PE     */
extern int  _get_(/*PE,src,tgt,len*/);  /* Read memory on other PE      */
extern void _do_sync_(/*id*/);          /* Synchronization with ID      */
extern void _do_print_(/*msg*/);        /* Print string on Syncer       */
extern void _do_printf_(/*fmt,...*/);   /* Print vararg string on Syncer*/
extern int  _getpenum_();               /* Get PE number                */
extern int  _global_or_(/*val*/);       /* Global OR: Add all, give sum */
extern void _send_end_();               /* Send request for end         */
extern int  _read_end_();               /* Receive final M_END message  */
extern int  _R_penr;                    /* PE Number (directly)         */


/* Message buffer handling */

extern char     *_get_msg_buffer_();    /* Allocate reusable buffer     */
extern void      _free_msg_buffer_();   /* Free buffer                  */



/* Common utility functions */

extern int      _R_PEtoRunner();        /* Map P4 IDs <-> PE numbers    */
extern int      _R_PEtoWorker();
extern int      _R_PEtoControl();
extern int      _R_P4toPE();
extern void     _R_do_debugprint();     /* Debug output routine         */
extern void     _R_do_print();          /* Print string on Syncer       */
extern void     _R_do_printf();         /* Print vararg string on Syncer*/
extern void     _R_nil();               /* Do nothing                   */
extern char    *_typename_();           /* Get name of message type     */
extern char    *_statename_();          /* Get name of wait state       */

/*
 * Debug Information output: DBGPRINT(level, format-string, arguments)
 *
 * Levels:
 *
 * 4 - Start and End messages
 * 5 - Worker: Actions
 * 6 -         Return values, OKs
 * 7 - Runner: Actions
 * 8 -         Details, ignored Messages
 *
 */

#ifndef _IN_UTILS_
FILE *_R_log;
extern void _R_debugprintfl();
extern void _R_open_log();
extern void _R_close_log();
#endif

#define DBGPRINT        _R_debugprintfl
#define DBGPRINT_OPEN   _R_open_log             /* In Utils.c   */
#define DBGPRINT_CLOSE  _R_close_log


/* GetPutInfo - tells Worker where to fetch/put data when       */
/*              put/get in debug mode                           */

typedef struct {
    long        type;           /* Type of request (get or put) */
    long        workadr;        /* Address within Worker        */
    long        runadr;         /* Dito within Runner           */
    long        length;         /* Length of data to transfer   */
    long        flag;           /* Flag for busy waiting        */
} GetPutInfo;


/* Name for logfiles (P4 ID is added at the end) + debug output  */
/* NOTE: THIS IS REDEFINED IN THE MAKEFILE HERE JUST AS FALLBACK */
```

```
/*#define LOGFILENAME "/tmp/runtime_log."*/


/* Wait states of program, e.g. waiting for read/write/sync etc */

#define WAIT_NONE       0       /* Not waiting at all           */
#define WAIT_READ       1       /* Want to read                 */
#define WAIT_WRITE      2       /* Want to write (synchron.!)   */
#define WAIT_SYNC       3       /* Want to sync                 */
#define WAIT_SEND_END   4       /* Want to send "end"           */
#define WAIT_READ_END   5       /* Want to receive "end"        */
#define WAIT_GET        6       /* Want to get data             */
#define WAIT_PUT        7       /* Want to put data             */


/* Message types for the PE communications                      */
/*                                                              */
/*      S = message between Worker & Runner                     */
/*      M = message from/to Syncer-Master                       */
/*      D = message from Master Debugger                        */
/*      C = message from Control                                */
/*                                                              */
/* (Syncer = p4 "Master", not to be confused with               */
/*  master debugger!)                                           */


#define S_PID           300     /* Send Unix Pid to runtime     */
#define S_NAME          302     /* Send Unix Filename to Runner */

#define M_START         310     /* Syncer:start the whole stuff */
#define S_START         313     /* Runner: Worker, start now    */
#define S_END           311     /* Request from slave to end    */
#define M_END           312     /* Command to end all activities*/
#define M_DISPLAY       315     /* Display for master debugger  */

#define M_SYNC          320     /* Synchronize workers          */
#define M_CONT          321     /* Continue after sync          */

#define M_DBG_PRINT     328     /* Debug output on Syncer       */
#define M_PRINT         329     /* Print string on Syncer       */

#define S_WRITE         330     /* Send data to PE (data)       */

#define S_GET_INFO      340     /* Give addr where to get from  */
#define S_GET_DATA      341     /* Transmit memory contents     */
#define S_GET_OK        342     /* Acknowledge                  */

#define S_PUT_INFO      350     /* Give address where to put to */
#define S_PUT_DATA      351     /* Transmit memory contents     */
#define S_PUT_OK        352     /* Acknowledge                  */

#define M_OR            360     /* Send value for global Or     */
#define M_OR_VAL        361     /* Receive global sum for Or    */

#define M_START_DBG     370     /* Create debugger process      */
#define M_ATTACH_DBG    371     /* Attach debugger to Worker    */
#define S_ATTACH_INFO   372     /* Status (ok/nope)             */
#define M_DBG_STOP      373     /* Stop Worker                  */
#define M_DBG_CMD       374     /* Send debug command to execute*/
#define M_QUERY_DBG     375     /* Ask for output from debugger */
#define M_DBG_CMD_QUERY 376     /* Send cmd & wait for output   */
#define S_DBG_RESULT    377     /* Result of command (string)   */
#define M_DETACH_DBG    378     /* Detach debugger again        */
#define M_QUIT_DBG      379     /* Exit debugging process       */

#define S_NEXT_TYPE     390     /* Announce type of next message*/

/* Master Debugger - Control communication */

#define D_RUN           400     /* Start running */
#define C_RUN           401     /* Acknowledge */
#define D_STOP          402     /* Stop execution (interrupt) */
#define C_STOP          403
#define D_CONT          404     /* Continue execution */
#define C_CONT          405
#define D_NEXT          406     /* Next statement */
#define C_NEXT          407
#define D_STEP          408     /* Also step in fn-calls */
#define C_STEP          409
#define D_STOP_AT       410     /* Stop at line */
#define C_STOP_AT       411
#define D_STOP_IN       412     /* Stop in function */
#define C_STOP_IN       413
```

```
#define D_DELETE        414     /* Delete breakpoint */
#define C_DELETE        415
#define D_WHERE         416     /* Stack trace */
#define C_WHERE         417
#define D_PRINT         418     /* Print expression */
#define C_PRINT         419
#define D_STATUS        420     /* Status */
#define C_STATUS        421
#define D_DUMP          422     /* Dump local variables */
#define C_DUMP          423
#define D_VISUALIZE     424     /* Combine array */
#define C_VISUALIZE     425
#define D_TREE          426     /* Combine debug tree */
#define C_TREE          427
#define D_PROFILE       428     /* Combine profile */
#define C_PROFILE       429
#define D_ACTIVATIONS   430     /* Combine activations */
#define C_ACTIVATIONS   431
#define D_QUIT          432     /* Quit debugger */
#define C_QUIT          433
#define D_CALL          434     /* Call function */
#define C_CALL          435
#define C_POSITION      437     /* Report postition in source */
#define D_WHATIS        438     /* Get type of expression */
#define C_WHATIS        439
#define D_ADR           440     /* Get address of variable */
#define C_ADR           441
#define D_GET_BLOCK     442     /* Get block of memory */
#define C_GET_BLOCK     443
#define C_BROKEN        445     /* Breakpoint encountered */
#define C_FAULTED       447     /* Program faulted */
#define C_NEW_STATE     449     /* New wait state */
#define D_END_WAIT      450     /* Continue from wait state */
#define C_END_WAIT      451


#endif  /* #ifndef _RUN_WORK_H_ */
```

D.5 Details der Implementierung

In diesem Abschnitt finden sich für den interessierten Leser Feinheiten der Programmierung.

D.5.1 Aufbau des Strukturbaums

Im dynamischen Strukturbaum wird ein allgemeiner Baum aufgebaut, indem alle Söhne eines Knotens untereinander in einer doppelt verketteten Liste angeordnet sind. Der Vater hat einen Verweis auf den ersten Sohn, und alle Söhne kennen ihren Vaterknoten.

Diese Anordnung hat den Vorteil, daß die Datenstruktur im Entartungsfall der Liste kompakt ist, aber dennoch beliebige Bäume unterstützt, ohne in einem Knoten selbst Daten veränderlicher Länge zu halten (z.B. ein Feld mit Verweisen auf die Söhne).

Außerdem läßt sie sich leicht durchlaufen, was die Erstellung einer textuellen Darstellung aus dem Baum und umgekehrt den Aufbau eines Baumes aus einer Zeichenkette erleichtert. Hilfsfunktionen vereinfachen die Handhabung und reduzieren den Aufwand, z.B. durch Bereitstellen einer Prüfsumme, die sich bei Veränderungen im Baum mitverändert. Dadurch muß MSDB den etwas aufwendigen Vorgang des Transfers des Strukturbaums vom untersuchten Programm in den Debugger nur durchführen, wenn sich der Baum wirklich geändert hat.

Ein Beispiel der Abbildung zwischen einem Baum und einer Zeichenkette geben Abbildungen D.1, D.2 und D.3.

D.5.2 Anwendung der Visualisierungen

Es gibt verschiedene Möglichkeiten, die Visualisierungen zu verwenden. Sie lassen sich direkt in einem Programm aufrufen und erzeugen ihre eigene X-Windows Umgebung, oder man kann sie in eine vorhandene X-Umgebung lediglich als Fenster einbinden. Zudem ist ein Ablauf im Vordergrund als Funktion oder im Hintergrund als eigener Prozeß möglich. Ein Visualisierungs-Prozeß ist über einen UNIX `socket` mit dem Hauptprozeß verbunden, über den er sich mit neuen Daten versorgen läßt. Die Darstellung dieser Daten kann sofort erfolgen, oder erst nach Klick auf den `s` Knopf in der linken unteren Ecke des Visualisierungsfensters.

D.5.3 Einsatz schneller Macros

Der Aktualisierungsteil der Debug-Funktionen (`UpdateDebugFn()`) wurde aus Effizienzgründen als Macro ausgelegt, da er im Inneren jeder Schleife aufgerufen wird. Er besteht nur aus dem indirekten Inkrementieren der Zähler im momentanen Blatt des Strukturbaums und in der Profile-Tabelle. Die Zeiger auf diese Zähler werden in der Initialisierung (`StartDebugFn()`) gesetzt.

D.5.4 Probleme auf unterer Ebene

Die folgenden zwei Beispiele mögen exemplarisch für die Art von Problemen auf unterster Ebene stehen, die bei der Entwicklung des MSDB auftraten. Sie brachten keine fundamentalen Ideen ein, beanspruchten aber nicht unwesentliche Arbeitszeit:

Pseudo-Terminals und p4

Wie im Abschnitt über die Implementierung erklärt, verwendet der MSDB für die Arbeit auf C-Ebene den UNIX-Debugger `dbx` oder `gdb`. Die Anbindung geschieht über ein Pseudo-

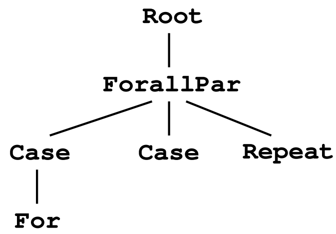


Abbildung D.1: Abstrahierter Strukturbaum des Beispiels.

```
Tree: LastID 5, LastNode 5, NumCreate 6, NumKill 0
SEQ Root (ID=0 l,c=0,0..0,0 count=0, PE=1)
      PAR ForallPar (ID=1 l,c=4,2..25,2 count=1)
            PAR Case (ID=2 l,c=5,4..22,4 count=1)
                  PAR For (ID=5 l,c=8,8..12,8 count=67)
            PAR Case (ID=3 l,c=13,4..13,4 count=6)
            PAR Repeat (ID=4 l,c=19,4..20,4 count=52)
```

Abbildung D.2: Strukturbaum in textueller Darstellung: Die zuletzt vergebene Kennzahl ist 5, der zuletzt angelegte Knoten ist der mit der Kennzahl 5 (das FOR in parallelem Kontext), es wurden 6 Knoten erzeugt und noch keiner gelöscht (Prüfsumme). Für jeden Knoten wird sein Kontext (SEQuentiell oder PARallel-asynchron) sowie seine Kennzahl (ID), seine Startposition und Endposition im Quellcode und sein Durchlaufs-Zähler ausgegeben. Die Einrückung spiegelt die Tiefe im Baum wider. Die Darstellung entsteht durch Tiefensuche im Baum mit gleichzeitiger Ausgabe der angetroffenen Knoten.

$\Updownarrow$

```
"5 5 6 0
 1 1 0 0 0 0 0 0 0 0 0 0 0
 21 2 1 1 4 2 25 2 0 0 0 0 1
 33 2 2 2 5 4 22 4 0 0 0 0 1
 13 2 5 3 8 8 12 8 0 0 0 0 67
 33 2 3 2 13 4 13 4 0 0 0 0 6
 11 2 4 2 19 4 20 4 0 0 0 0 52
 0 0 0 0 0 0 0 0 0 0 0 0 0 "
```

Abbildung D.3: Strukturbaum als Zeichenkette: Als Präambel werden die Kennzahlen und die Prüfziffern ausgegeben. Danach folgt für jeden Knoten sein Typ, sein Kontext, seine Kennzahl, seine Tiefe im Baum, seine Position im Quelltext sowie Angaben über `FORALL`-Variablen (hier Null) und die Zahl der Durchläufe. Die Liste wird mit einer Zeichenkette aus Nullen abgeschlossen. Die Rückwandlung in einen Strukturbaum erfolgt durch Erzeugen von den Angaben entsprechenden neuen Knoten, wobei Gebrauch von der Tiefe des Knotens gemacht wird. Ist sie größer als die des Vorgängers in der Liste, so ist der neue Knoten dessen Sohn. Ist sie gleich, so ist er sein Bruder, und ist sie kleiner, so ist er der Bruder des zuletzt auf dieser Tiefe angelegten Knotens. Da der Baum beim Erzeugen der Zeichenkette mit Tiefensuche durchlaufen wurde, erzeugt dieser Algorithmus wieder den korrekten Ursprungsbaum.

Terminal (`pty`). Sie funktioniert leider nicht so einfach, wie man auf den ersten Blick meinen möchte. Im verteilten Fall z.B. wird auf den PEs die Standard Ein- und Ausgabe von p4 umgelenkt. Das betrifft indirekt auch das `pty`, so daß sich der `dbx` weigert, Befehle auszuführen. Nur aus diesem Grund wird im Laufzeitsystem der `gdb` verwendet, der dieses Problem nicht hat. Ihn konsequent statt des `dbx` zu benutzen, war auch nicht möglich, da der `dbx` zu stark in den MSDB für den einzelnen Rechner integriert ist. Es blieb also nur die Mischlösung.

Dafür wurden die Botschaften, die die verteilten Control-Prozesse austauschen, unabhängig vom zugrundeliegenden Debugger gewählt, so daß der real verwendete Debugger und seine spezifischen Kommandos etc. im Laufzeitsystem verborgen sind. Ausgetauscht werden lediglich Anforderungen von Daten oder Stop-Meldungen, aber keine konkreten Debugger-Befehle.

Signale und p4

Ein weiteres Problem, das durch die Verwendung von p4 beim Debugger auftrat, war ein zunächst unerklärlicher Stillstand des verteilten Programms. Es stellte sich heraus, daß p4 intern das Signal `SIGUSR1` verwendet, um den Prozeß von der Ankunft neuer Daten zu benachrichtigen, und der Debugger dieses Signal normalerweise abfängt. Der Prozeß erhielt also niemals Daten.

Es wurde also eine Anweisung eingefügt, die den Debugger zum Weiterleiten und Ignorieren dieses Signals bewegt, woraufhin das Problem nicht mehr auftrat.

Anhang E

Konferenzbeitrag

Folgender Artikel wurde als Beitrag für den *3rd ACM/ONR Workshop on Parallel and Distributed Debugging* eingereicht:

MSDB – A Parallel Debugger, Profiler, and Visualizer for Modula-2*

Stefan U. Hänßgen
haensgen@ira.uka.de
Universität Karlsruhe,
Department of Computer Science,
Germany

Abstract

Moving the focus of parallel debugging from machine-level problems to the level of a higher, machine-independent programming language leads to new concepts for abstractions and visualizations in debugging. In this paper the novel concepts are presented that we felt to be necessary for debugging an explicitly parallel programming language such as Modula-2*. Prototype implementations of *MSDB* exist for a single CPU machine and for a network of workstations. Work on further platforms is in progress.

E.1 Introduction

The large amount of data and the complex patterns of interaction among thousands of processes still are a great challenge for debuggers of parallel programming.

Existing parallel languages (and therefore, debuggers written for them) largely suffer from machine dependency. The user spends most of the time debugging problems of the machine, not the algorithm.

This extended abstract presents the Modula-2* Debugger *MSDB*. Modula-2* [TiHe90] is a parallel extension of Modula-2 featuring an arbitrary number of processes operating in a single address space and arbitrarily nested synchronous and asynchronous computation expressed in a machine-independent way. Parallelism is introduced by the FORALL construct[1].

---

[1] The FORALL statement has the following syntax:

Due to the high level of abstraction of this language, many of the usual problems of parallel programming are of no concern to the user, e.g. race conditions, deadlocks, and communication operations. These are all taken care of by the compiler.

This property shifts the focus of debugging from machine dependent issues to more abstract problems at the level of the language itself, such as visualization of parallel activities, data and dynamic activation trees. For performance tuning, profiling at the language construct level is also supported.

(The final paper will include more elaborate sections on Modula-2* and related work.)

E.2 Concepts

As a basis for debugging, MSDB provides the usual debugger features such as setting and examining variables, setting breakpoints and single stepping through the program. Breakpoints are global, i.e. all PEs are halted if one of them encounters a breakpoint.[2] All distributed information about the program is integrated on one screen with several windows.

In this chapter, we focus on the concepts that directly support high-level abstractions in terms of the language.

E.2.1 Activation Trees and Grouping

The *activation tree* shows, in terms of the programming language, where the activities inside the program are located at the moment. Each time an activity enters a new statement that affects the control flow, e.g. performs a function call, enters a loop, encounters an IF-statement or spawns a new FORALL, the tree is updated.

The leafs of the tree are the locations of the activities while their predecessors represent their dynamic history, like a stack in a conventional program – only that MSDB shows all constructs, not just function calls, and displays parallel activities instead of a linear stack.

To make this approach feasible even with a large number of concurrent activities at different places in the program, equivalent leafs are *grouped* together, e.g. all activities which are inside the same run of a loop are shown as one node which can be expanded again by clicking on it. A simple example of grouping several nodes in an IF and a WHILE statement executed in parallel is presented in figure E.1.

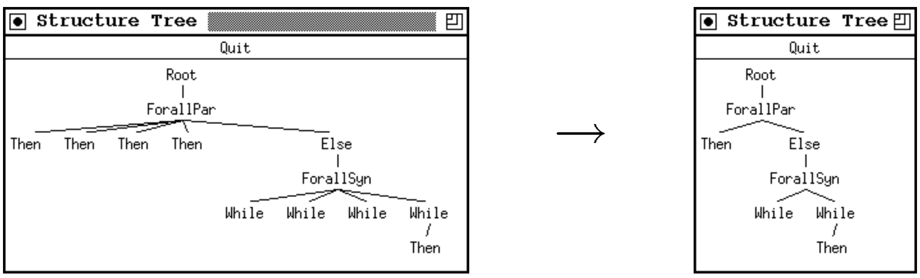


Abbildung E.1: Dynamic activation tree before and after grouping of equivalent nodes

This display looks similiar to the dynamic process trees in [RRZ88], but while they just take a coarse-grained look at processes, MSDB uses terms of the language itself.

E.2.2 Multiple program counters

In the source code window, multiple program counter arrows represent the *grouped activities*, with the size of the arrow indicating the number of processes that are at this position. Again, more detailed information about the original processes can be obtained by clicking on the arrow.

```
ForallStatement = FORALL ident : SimpleType IN {PARALLEL | SYNC}
            [VarDecl + BEGIN] StatementSequence END
```

[2] This is to provide a consistent handling among single processors, SIMD and MIMD machines.

E.2.3 Visualization of data

Multidimensional arrays are visualized in MSDB by viewing 2-dimensional slices through the data (similiar kinds of array data visualization are used in a variety of commercial debuggers, e.g. in [Thin91] and [Masp91b]). 1-dimensional data gets mapped onto a 2-dimensional array to give a better overview.

Different kinds of visualizers are available, depending on the kind of information the user wants to obtain: *Comparison* visualizers – for equality/greater/less relations to a value selected by the user – highlight the array elements for which the comparison is true. *Range* visualizers form a generalization of comparison which highlights all values inside a user-defined range (when one boundary is $\pm\infty$, we have a simple comparison again; if both boundaries are the same, we test for equality). An example is shown in figure E.2. Finally, *Value* visualizers map an array elements to a grayscale representation of the data. The mapping is automatically chosen to represent the median range of all values, compensating for statistical peaks. See figure E.3.

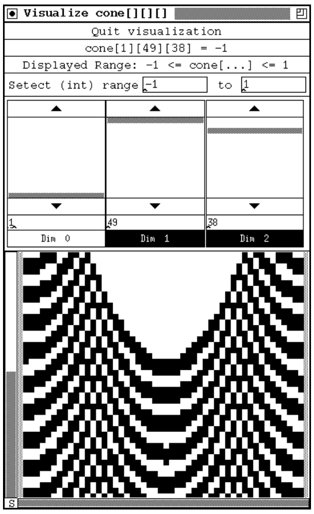


Abbildung E.2: Range visualizer

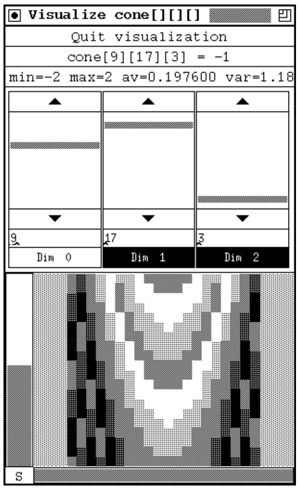


Abbildung E.3: Value visualizer

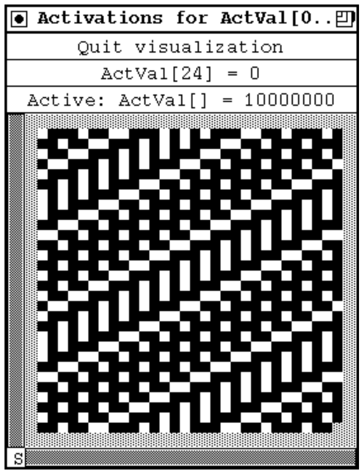


Abbildung E.4: Activity visualizer

E.2.4 Visualization of activities

Abstracting from machine dependent displays of processor activities, such as featured in [Masp91b], MSDB has a specialized visualizer for the current *activities* of the virtual PEs (figure E.4). While they do not directly represent real PEs – which are not visible to the Modula-2* programmer –, a glance at the distribution of activities among the PEs can reveal patterns of idle processes and thus inspire improvements of the program.

E.2.5 Profiling

For performance tuning – and also for debugging –, the user needs to know which parts of the program are executed how often and on which PE. MSDB keeps track of the processes entering, repeating and leaving constructs and thus can provide the user with a display of the most-used *structures*. The number of runs through a statement corresponds roughly to the time processes spent there. Direct timing and a per-line profiling is not performed yet.

E.2.6 Mapping from Modula-2* to C dialects

In the current implementation, the Modula-2* compiler generates code for the parallel C-dialect of the target machine. For the debugger, the use of C as intermediate language has advantages, i.e. the possibility to rely on existing source level debuggers such as `dbx` and `gdb`, but it also introduces the task of transparently mapping Modula-2* to C and vice versa while debugging a program. This is not simple due to the extensive restructuring of the code by the compiler[3] [PhTi91].

In some cases, e.g. for the compiler developer or the interested user, it is useful to see what is going on at the intermediate C level, so MSDB provides an optional display of the C code as well. In the MIMD case, a source code window for every real PE can be opened.

E.3 System Description

Currently, prototype implementations of MSDB exist for a *sequential program* running on a single SUN workstation and for a *network of workstations*. The latter implementation is based on the p4 system [BuLu92] for communication and process control. Both rely on existing UNIX debuggers (`dbx` and `gdb`, namely) to inspect and change both data and program state.

The X–Windows based user interface is derived from `xdbx` [Cheu89]. A representative screen dump of the whole debugger in action is shown in figure E.6.

The runtime system consists of three components per PE: The *Runner* deals with memory access on remote PEs[4], the *Worker* does the distributed computations, and a *Control* process performs debugging activities such as stopping the Worker and modifying it. A central *Syncer* provides global synchronization barriers. The debugger processes of all PEs are coordinated in a *Master Debugger* process which also handles the user interface. Figure E.5 illustrates the setup of the processes and their interaction.

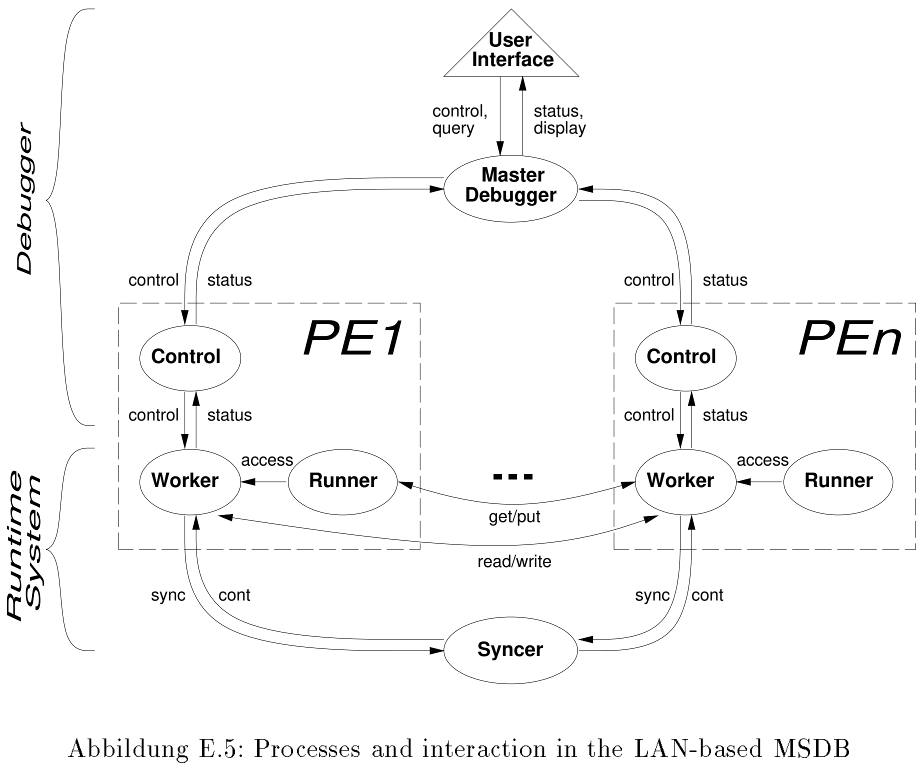


Abbildung E.5: Processes and interaction in the LAN-based MSDB

To support the debugger, the Modula-2* compiler has a switch to place additional debugging information in the code it generates (comparable to the UNIX `cc`'s option "`-g`"), including hints for the line and variable mapping between Modula-2* and C as well as debug functions, which record the dynamic activation tree and also generate the profiling information.

[3] E.g. for the elimination of synchronization barriers

[4] I.e. unpredictable get/put access (like the data network of a parallel machine), while the *Worker* does synchronized read/write access itself (like the control network)

E.4 Current Status and Future Work

The *sequential version* is ready for use. The *distributed runtime system* is fully operational, but the functionality is not yet completely integrated into the graphical user interface.

Work on other platforms, such as MasPar and Transputer Cluster, is in progress.

It is planned to design and implement a general debugging kernel to meet the specific needs of Modula-2*. The current solution with seperate **dbx** and **gdb** processes is slightly inefficient and requires a special interface to the debugger using pseudo terminals.

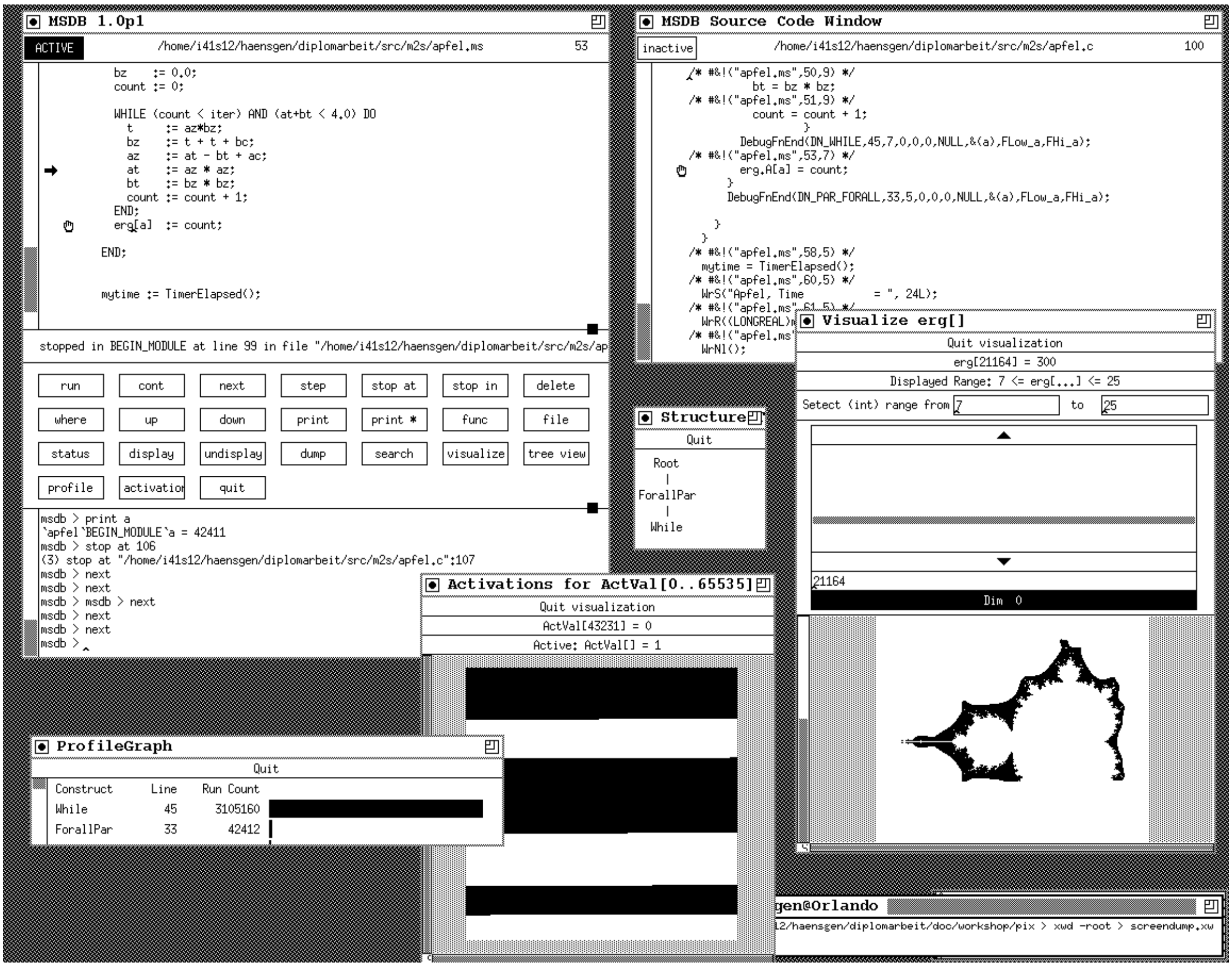


Abbildung E.6: MSDB in action. The program being debugged is a simple fractal computation, the (partial) results of which are shown in a range visualizer. A visualization of the virtual PEs' activities as well as the dynamic activation tree and a profile graph can also be seen. The left window shows the Modula-2* source with a program counter and a breakpoint, the right window displays the corresponding C source.

E.5 Conclusion

In this extended abstract a high-level debugger for Modula-2* with capabilities regarding abstractions and visualizations such as activation trees, array and activity displays, and profiling support was presented. Due to the unique design of Modula-2*, the emphasis could be put on more user-oriented issues while most machine-level problems can be ignored.

Anhang F

Benutzereinführung

Dieser Anhang enthält die englische Benutzereinführung zum Debugger, die mit dem Modula-2* System zusammen ausgeliefert wird.

F.1 Introduction

This documentation briefly describes the features of the Modula-2* Debugger MSDB which offers abstractions and visualizations at the high language level. In section F.2 we present the different kinds of windows and visualizations MSDB provides and discuss their functionality. After that, in section F.3 an overview of the commands is given which also contains more details on how to use MSDB.

The debugger as well as the documentation is still in an experimental state. However, it has already proven useful for examining Modula-2* programs on a single CPU machine. A distributed version of MSDB for the LAN runtime system exists but is still under development. Since MSDB is the result of a single master's thesis, we cannot guarantee continued software support, but we will try to do our best.

Any kind of feedback is welcome, e.g. suggestions for improvement, bug reports, comments, or post cards to the author[1] :-)

Mail to `haensgen@ira.uka.de` or `msc@ira.uka.de` for direct feedback – when reporting bugs or suggesting new features, please give as much detail as possible (just 'MSDB dumps core' hardly helps in finding the problem, but a traceback and a close description of the circumstances under which the error occurred might be valuable).

Happy bug hunting!

[1] Stefan Haenssgen, Nuitsstr. 2c, D-76185 Karlsruhe, Germany (before July 1, 1993: DW-7500 Karlsruhe 21)

F.2 MSDB's Windows

The debugger has a graphical user interface based on **xdbx**, an X Windows version of the UNIX `dbx` debugger. All output is displayed in windows, either as text or as graphic visualization [2]. In this section, we describe all kinds of windows used by MSDB and explain where and when they are used.

After starting MSDB, as described in section F.3.2, two windows will pop up, the *main window* and the additional *C source window*.

F.2.1 Main Window

Fig. F.1 shows a typical MSDB main window:

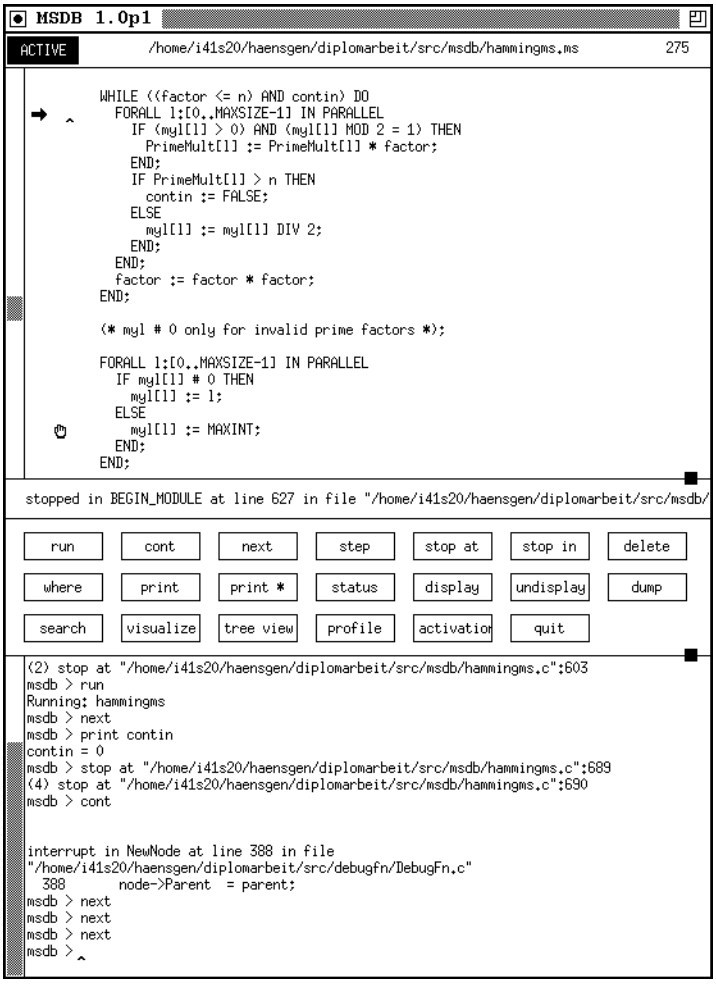


Abbildung F.1: MSDB main window with Modula-2* source code, status line, command buttons and I/O region.

The main window is divided into five regions:

- The *source code information* line shows which file we are currently looking at and in which line the cursor is. It also contains the *active window button* which determines

[2] Additional messages that appear on the terminal window in which MSDB was started are mostly of internal use only and may be ignored.

which source code window is affected by the command buttons. The details of this are described in section F.3.

- In the *source code listing*, we see the Modula-2* source code of the program being debugged. Using the scrollbar, you can look at different parts of the source. An arrow indicates the current position in the program. As we step through the program, the arrow changes position. The hand sign represents a breakpoint.

- The *status line* tells you what has happened lately, e.g. where the program stopped. It also informs you of errors, e.g. if you try to open a visualization window without having selected an array to visualize.

- Using the *command buttons*, you perform different actions such as stepping through the program, examining variables and creating visualizations. They are described in detail in section F.3.

- Finally, the *I/O region* shows MSDB's output and makes it possible to directly enter commands. If the program being debugged performs I/O, it is done in this region, too. However, nearly all debugging operations can be performed simply by pointing and clicking.

F.2.2 C Source Window

The Modula-2* compiler translates Modula-2* to the parallel C derivate of the target machine, which is then processed using their native C compiler. For a better understanding of what is going on at the C level, MSDB also provides a C source code window. It is not necessary for debugging, but may give interesting insights.

Also, it is possible to set breakpoints and examine variables at C level by clicking on the window's activation button in the upper left corner (or by simply clicking inside the window). All actions such as viewing data or setting breakpoints always refer to the currently active window, i.e. Modula-2* or C.

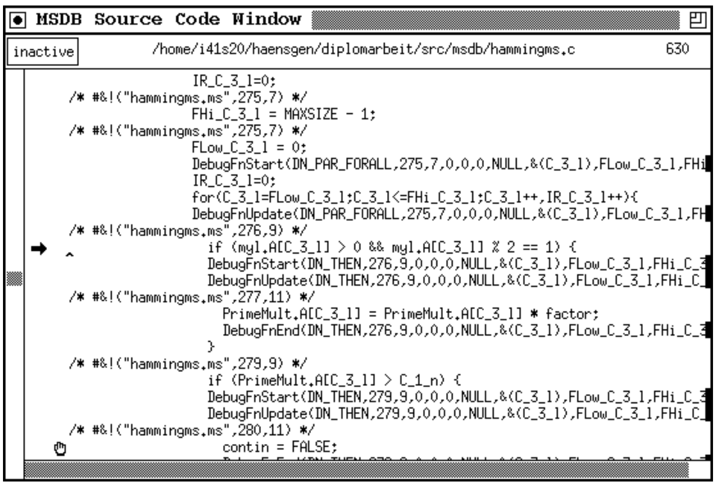


Abbildung F.2: C source main window with corresponding C source code.

F.2.3 Dynamic Structure Tree

The dynamic structure tree may be displayed at any time. It shows the dynamic history of the program and the current locations of execution. Each time the control flow enters a construct such as `WHILE`, `IF`, or `FORALL`, a node corresponding the construct is added to the tree. When the structure is left again, the node is also removed. A `FORALL` creates one node for each of its parallel execution threads. The tree keeps track of those activities and thus gives an overview of the dynamic structure of the program.

In the sequential case, which we are looking at, the tree degenerates to a list, somewhat resembling a call stack. However, the elements in this list correspond to language constructs, not just to a call hierarchy of functions.

By clicking on a node of the tree, more detailed information on the construct it represents can be obtained. Fig. F.3 shows a simple tree with one node enlarged.

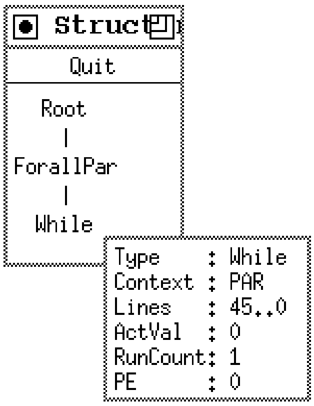


Abbildung F.3: Dynamic structure tree.

If the tree is left up when stepping through the program, it is automatically updated each time it changes. When the program continues and then stops again, it is also updated. Clicking on the quit button pops the tree down again.

F.2.4 Data Visualizers

To create a realistic overview over the large amounts of parallel data, MSDB features several kinds of data visualizers. They display multi-dimensional arrays by showing 2-dimensional slices through them. An example can be seen in fig. F.4.

There are four main areas in a visualizer:

- The *information area* shows the name of the array being displayed and the exact value of the array element under the mouse cursor. Depending on the kind of visualizer used, different parameters may be set here, too. The example is a gray scale visualizer which automatically maps data values onto gray scale pixels, so there is no user control of further parameters.

- Using the *scrollbars*, you can scroll through different slices of the array, either by clicking on the up- and down-button for a single step, or by directly manipulating the scrollbar as you know from other X applications. When you move the mouse inside the display below, the corresponding two dimension values are updated, too.

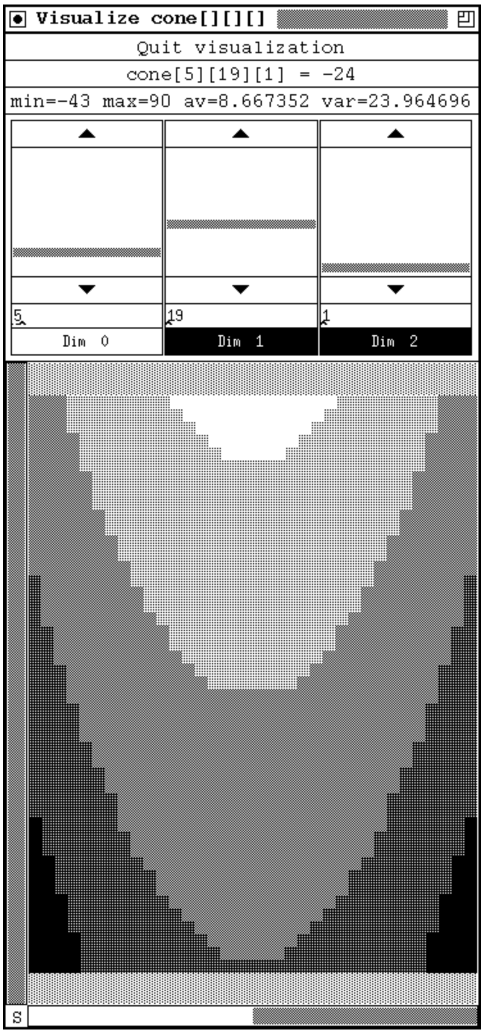


Abbildung F.4: Data Visualizer: information area, scrollbars, dimension selectors, and array display.

- The *dimension selector buttons* allow the selection of the dimensions through which the array is sliced. Two dimensions are active at once, representing the 2-dimensional cut through the array.

  In the example, we see a cut through a 3-dimensional cone. Dimensions 1 and 2 are active. By moving the above mentioned scrollbar for dimension 0, we can view subsequent slices of the array, giving a nice impression how data is structured.

- At the bottom of the visualizer, we see the *array display* itself. It features two scrollbars which may be used to view different regions of the array if it is too large to fit into the window. Clicking the left mouse button on an element zooms in at this element for details, and clicking the right button zooms out of the array.

For one-dimensional data, the array is mapped onto the next possible rectangle with as many elements as the array. This gives a clearer picture than showing one single line of array elements.

Like the tree, a visualizer is also updated on each step[3]. It can be popped down using

[3]This may be time consuming for slow machines and large arrays, especially when using the gray scale

the quit button.

Apart from the gray scale visualizer, there are also other kinds which are discussed later.

F.2.5 Activation Visualizer

The activation visualizer is a specialized array visualizer which gives an picture of the current parallel activations in the program. It shows the virtual processors which are currently active as black and the inactive ones as white:

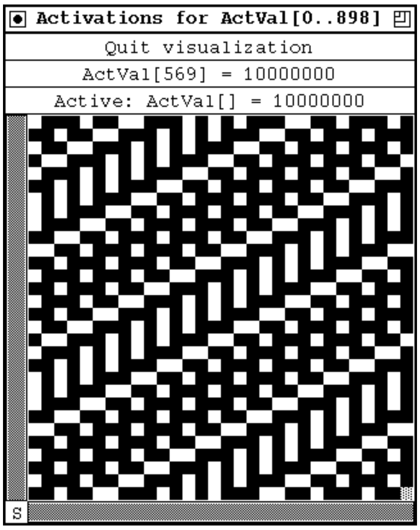


Abbildung F.5: Virtual processor activation display.

On a sequential machine, these are just the virtual activations and only represent potentially parallel activities. E.g., in an asynchronous `FORALL IN PARALLEL`, all values of the `FORALL` variable are sequentially used in a `FOR` loop. Semantically, however, one virtual processor per value executes its own loop. Therefore, when visualizing the activations of such a `FORALL`, all values less than the current value of the index variable are considered inactive, while the rest is still to be executed and thus active.

Similar to the data visualizer, one-dimensional activation information is displayed in a square, but it is filled with blanks to guarantee that it fits into the next bigger square. Visualization of multi-dimensional activity arrays is not supported, yet.

The activation visualizer is also updated on each step through the program. If the context changes, i.e. the `FORALL` it represents is closed, it informs the user and pops down automatically. Of course, it can be manually popped down with the quit button.

F.2.6 Profile Window

Each time a language construct, such as a `WHILE`, an `IF` or a `FORALL`, is executed, MSDB increments a counter for it. Thus, the number of runs through a construct can be examined. The profile window shows each construct encountered until now, together with its line number and its run count:

The run count is a good approximation of the time spent in a construct, although it does not take the different complexity of different statements into account.

---

visualizer.

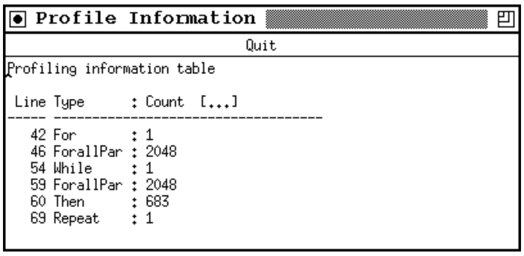


Abbildung F.6: Profiling information display.

Like the windows described before, the profile window is updated at each step or breakpoint. Clicking on the quit button pops it down again.

F.3 Using MSDB

In this section, we show how to prepare a program for debugging and present the debugger's functionality.

F.3.1 Preparing a program

To work with MSDB, a Modula-2* program has to be compiled using the '`-g -S -s0`' options of `msc`. This is similar to using the '`-g`' option of the C compiler for debugging C programs. With the debug flag set, the compiler produces code instrumented with debug functions that keep track of entry and exit of language constructs and perform the profiling counts. In addition, information for mapping the Modula-2* source onto the resulting C source is added in special comments.

F.3.2 Starting the debugger

To debug a program `foo`, which was compiled as described above, you simply enter `msdb foo`. The debugger will start up both the main window and the C source window will appear.

To enable MSDB's use of the X windows system, the environment variable `DISPLAY` has to be set to a valid display and the machine MSDB is running on has to have access to that display. E.g., if you're running MSDB on host A and have the display at host B, you would issue a '`setenv DISPLAY` B`:0`' on A and allow access via '`xhost +`A' on B.

F.3.3 Running and stopping the program

When the debugger pops up, the program is not active yet. To start it, simply click on the `run` button. When an error occurs which would normally lead to a core dump, MSDB gets the control and reports what happened and where in the program.

To stop a running program at once, press Control-C in the I/O window. MSDB will stop the program and report where it was interrupted.

Each time the program is stopped or encounters an error, the source code windows are scrolled to the corresponding position and the program counter arrows are updated.

If you let the program run without interrupting it and no error occurs, MSDB will report when it has finished execution. To be more exact, stops the program just before it completely exits, so that the profiling information is not lost.

F.3.4 Setting and deleting breakpoints

To stop the program at a point in the code where you want to examine its state more closely, you set a breakpoint. This is simply done by clicking on the line in the source code where you want the program to stop (maybe after scrolling the source code window to that line) and then clicking on the `stop at` button. A held up hand icon will appear left of the line to inform you that there is a breakpoint.

When the program reaches the breakpoint, it is stopped and control is returned to the user. You may at this point examine variables or continue in single steps, as shown in the next section.

You can define the breakpoint in the Modula-2* source or in the C window, depending on which one is active at the moment. The corresponding line in the other window gets marked, too. Due to the mapping, a breakpoint set at a certain line may appear at a different line, since breakpoints are only allowed in lines containing executable code.

To delete a breakpoint you no longer want to use, click on its line in the source and then use the `delete` button.

F.3.5 Continuing and stepping

Now that you know every way of stopping the program, let us look at how to continue execution. Note that it is not possible to continue when an error caused the program to stop, but you can continue the program if you interrupted it or it ran into a breakpoint.

To continue the execution of the program, click on the `cont` button. The program will run on until it is stopped again by an error, an interrupt or a breakpoint.

Often you will want to go through the flow of execution in single steps. This is what the `next` button is for: it causes the next line of the Modula-2* program to be executed and then returns control to the user. Note that a single line in Modula-2* may correspond to several lines in the C source, which are automatically stepped over. Due to restructuring of the program by the compiler, sometimes the sequence of execution may seem a bit strange, but a look at the C source explains those effects. Usually, the Modula-2* to C mapping gives smooth results, however.

The button (mis)labeled `step` serves a similar function but steps into functions instead of executing them in a single step like *next* does. Also, it steps according to the C source, not the Modula-2* listing. Its main purpose is the C-level examination of execution, which is rarely needed.

F.3.6 Structure tree, activation and profile

The dynamic structure tree window as well as the virtual process activation and the profile window may be displayed any time the program is stopped, using the buttons `tree view`[4], `activation` and `profile`, respectively. They are updated at each further stop.

For a closer description of the windows, please refer to section F.2.

Note that the tree and the profile are valid any time, but the activation display only makes sense when inside a `FORALL` statement. MSDB will complain when you try to get an activation visualization when outside such a statement. It will also close a display if the `FORALL` it represented is left.

[4] In a distributed environment, pressing the left mouse button to pop up the tree view would display the tree as a combination of all parallel processors, while the right button would group equivalent activities to obtain a clearer picture. On a single machine, however, the tree is a list anyway, so grouping it does not change anything.

F.3.7 Examining data

To show the value of a variable, mark it with the mouse in the source code window and click on the `print` button[5].

Not all identifiers used in the program are known to MSDB, e.g. constant definitions are mapped to C `#define`s which cannot be found in the symbol table.

F.3.8 Visualizing data

The print-it approach is useful for small amounts of data, such as a single variables. When you want to examine large volumes of data, e.g. multi-dimensional arrays, it is nearly impossible to obtain any practical information from a huge listing of the array elements. Instead, the use of a visualizer is recommended. The basic features of visualizers were described in section F.2.4. They can be used for all kinds of numerical arrays but not for compound structures or strings.

To use a visualizer, mark the name of an array in the Modula-2* source[6]. Then, click on the `visualize` button. MSDB will obtain the internal representation of the array and pop up a visualizer for it, which you may use like described in the section on visualizers above. Depending on which mouse button you click, you will get different kinds of visualizers:

- Clicking on the **left** mouse button pops up a *comparison visualizer* which highlights all array elements that fulfill a user defined condition. An example is shown in fig. F.7[7]. You can click on the `<`, the `=` or the `>` button, depending on the kind of comparison you want to be performed. Additionally, you can change the value with which the array elements are compared by moving over it with the mouse and editing the number (using the standard X windows editing keys). Press Return when you want the new value to take effect.

- The **middle** mouse button produces a *range visualizer* which highlights all array values that are inside a user definable range, like shown in fig. F.8. You can change the range by moving the mouse over one of the value entry fields, editing the number and pressing the Return key. In addition to numerical values, *+inf* and *-inf* for positive or negative infinity are also supported. A range of $[x,+\infty]$ is equivalent to a '$>= x$' comparison, $[x,x]$ is a test for equality, and $[-\infty,x]$ is '$<= x$'.

- Finally, using the **right** mouse button pops up a *gray scale visualizer* which automatically maps the array values onto five different gray scale values (that are displayed using different patterns). No user settable parameters exist. Instead, the minimum, maximum, average value and variance are shown. The visualizer compensates for some statistical peaks[8]. This kind of visualizer takes a bit longer to compute for large arrays, but it provides a very useful direct overview over most kinds of data.

For a description of the basic visualizer functionality, see again section F.2.4.

[5] Pointers may be dereferenced using the `print *` button, but this is for use in the C source, mostly.

[6] If the array expression is e.g. '`foo[i+1]`', highlight the '`foo`'.

[7] All three examples show the same set of values (an incomplete fractal computation) using their respective features.

[8] However, if data variance is too big, e.g. 100 times a '9999', followed by '1', '2', and '3', the resolution of the gray scale mapping is not fully utilized.

F.3.9 Miscellaneous functions

In this section, we briefly describe the remaining buttons found in the MSDB's main window:

`stop in` After highlighting a function name, press this button to stop at the first line of this function.

`where` Prints the execution stack, i.e. which functions were called and with which parameters. The dynamic structure tree in combination with the position indication arrow in the source code are another convenient way of obtaining this information.

`status` Shows which breakpoints are set with regard to the C source[9].

`display` Select a variable and click on this button to permanently display the variable's value at the bottom of the I/O window. Slows execution down.

`undisplay` Undo the corresponding variable's display command.

`dump` Print all local variables in terms of the C program. This may contain some strange names which are used by the compiler for the Modula-2* to C mapping

`search` Search for a string in the source code.

`quit` Your guess is as good as mine :-)

F.3.10 The End

This is the end of our little introduction into MSDB. We hope that you found it useful – happy debugging!

[9] Breakpoint 2 is used internally for stopping the program before it exits completely, thus preserving the profile information which is built up in the program's address space.

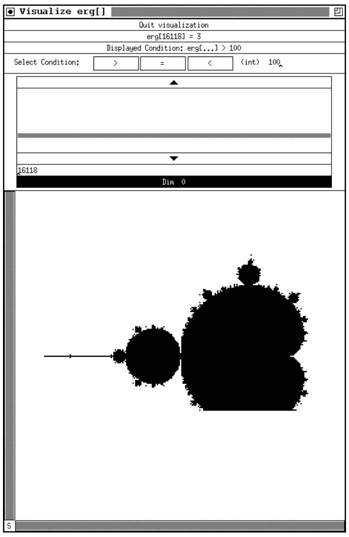


Abbildung F.7: Comparison Visualizer highlighting all values greater than 100.

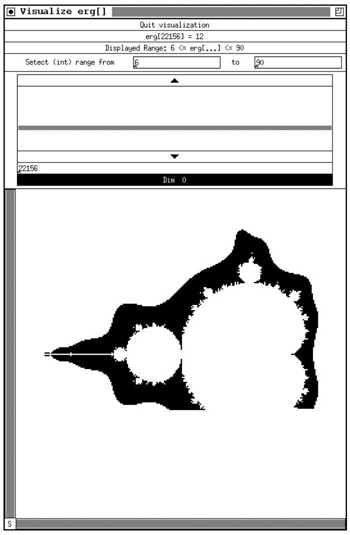


Abbildung F.8: Range Visualizer highlighting all values between 6 and 90.

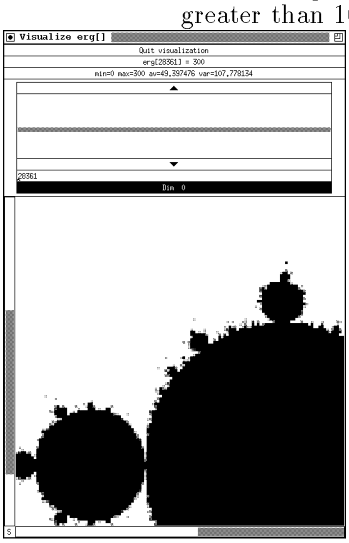


Abbildung F.9: Gray scale Visualizer displaying values as gray scale, zoomed in for details.

Literaturverzeichnis

[AdMu86] E. Adams und S.S. Munchnick, *Dbxtool: A Window-Based Symbolic Debugger for Sun Workstations*, Software – Practice and Experience, Seite 653–669, Juli 1986

[AMS91] P. Asente, J. McCormack und R.R. Swick, *X Toolcit Intrinsics – C Language Interface*, X Version 11 Release 5, MIT, Cambridge, MA, 1991

[BBL91] K.L. Beck, J.D. Becher und Z. Liu, *Integrating Profiling into Debugging*, Proceedings of the 1991 International Conference on Parallel Processing, Seite 284–285, August 1991

[BeBe92] J.D. Becher und K.L. Beck, *Profiling on a Massively Parallel Computer*, MasPar Technical Report TR002.1091, Sunnyvale, CA, 1992

[BBS89] D. Bernstein, A. Bolmarcich und K. So, *Performance Visualization of Parallel Programs on a Shared Memory Multiprocessor System*, Proceedings of the 1989 International Conference on Parallel Processing, Seite 1–7, 1989

[BDGMS91] A. Beguelin, J. Dongarra, A. Geist, R. Mancheck und V. Sunderam, *A User's Guide to the PVM Parallel Virtual Machine*, Oak Ridge National Laboratory ORNL/TM-11826, Oak Ridge, TE, 1991

[BuLu92] R. Butler und E. Lusk, *User's Guide to the p4 Programming System*, Argonne National Laboratory ANL-92/17, Aragonne, IL, 1992

[Cheu89] P. Cheung, *xdbx Manual Page*, University of Texas at Austin, 1989

[Chin91] D. Cin et al, *Programmierung in Modula-2*, 4. Auflage, Teubner, Stuttgart, 1990

[Coop87] R. Cooper, *Pilgrim: A Debugger for Distributed Systems*, Proceedings of the 7th International Conference on Distributed Computing Systems, Seite 458–465, 1987

[CuWi82] R. Curtis und L. Wittie, *Bugnet: A Debugging System for Parallel Programming Environments*, Proceedings of the 3rd International Conference on Distributed Computing Systems, Seite 394–399, 1982

[Elsh88] I.J.P. Elshoff, *A Distributed Debugger for Amoeba*, Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, Madison, WI, Seite 1–10, Mai 1988

[Haen91] S.U. Hänßgen, *Oberfläche eines symbolischen Debuggers für Modula-2**, Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, August 1991

[HHK85] P.K. Harter, D.M. Heimbigner und R. King, *IDD: an Interactive Distributed Debugger*, Proceedings of the 5th International Conference on Distributed Computing Systems, Seite 498–506, 1985

[HLP92] E.A. Heinz, P. Lukowicz und M. Philippsen, *Compiling Machine-Independent, Parallel Programs*, noch nicht veröffentlichter Artikel für die SIGPLAN, Universität Karlsruhe, Oktober 1992

[JoMa91] C.W. Johnson und P. Mackerras, *Architecture of an Extensible Parallel Debugger*, Proceedings of the 1991 International Conference on Parallel Processing, Seite 262–263, August 1991

[KeRi90] B.W. Kernighan und D.M. Ritchie, *Programmieren in C*, 2. Ausgabe ANSI C, Carl Hanser Verlag und Prentice Hall, 1990

[Knut84] D.E. Knuth, *The TeXbook*, Addison-Wesley, Reading, MA, 1984

[Lamp86] L. Lamport, *LaTeX: A Document Preparation System*, Addison-Wesley, Reading, MA, 1986

[Luko92] P. Lukowicz, *Code-Erzeugung für Modula-2* für verschiedene Maschinenarchitekturen*, Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, Februar 1992

[Mart90] M. Martin, *Entwurf und Implemenmtierung eines Übersetzers von Modula-2 nach C*, Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, Februar 1990

[Masp91a] MasPar Computer Corporation, *MasPar MP-1 MPL Programming Manuals*, Sunnyvale, CA, 1991

[Masp91b] MasPar Computer Corporation, *MasPar MP-1 System Overview and MPPE Manuals*, Sunnyvale, CA, 1991

[MiCh88] B.P. Miller und J.D. Choi, *Breakpoints and Halting in Distributed Programs*, Proceedings of the 8th International Conference on Distributed Computing Systems, Seite 316–323, 1988

[Mill92] B.P. Miller, *What to Draw? When to Draw? An Essay on Parallel Program Visualization*, University of Wisconsin-Madison, Madison, WI, 1992

[Pete91] C.D. Peterson, *Athena Widget Set – C Language Interface*, X Version 11 Release 5, MIT, Cambridge, MA, 1991

[PhTi91] M. Philippsen und W.F. Tichy, *Modula-2* and its Compilation*, Proceedings of the First International Conference of the Austrian Center for Parallel Computation, 30. September – 2. Oktober 1991, Seiten 169–183, Salzburg, Österreich, 1991

[Rich91] J. Richter, *Ein Debugger fuer Modula-P*, Diplomarbeit, Forschungszentrum für Informatik, Karlsruhe, 1991

[RRZ88] R.V. Rubin, L. Rudolph und D. Zernik, *Debugging Parallel Programs in Parallel*, Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, Madison, WI, Seite 216–225, Mai 1988

[Stal90] R.M. Stallman, *GNU Emacs Manual*, Free Software Foundation, Cambridge, MA, 1990

[StPe92] R.M. Stallman und R.H. Pesch, Debugging with GDB, Edition 4.06, Cambridge, MA, 1992

[Sun90a] Sun Microsystems, Inc, *Programming Utilities for the Sun Workstation*, Mountan View, CA, 1990

[Sun90b] Sun Microsystems, Inc, *C Programmer's Guide*, Mountan View, CA, 1990

[Sun90c] Sun Microsystems, Inc, *Debugging Tools: dbx*, Mountan View, CA, 1990

[Thin91] Thinking Machines Corporation, *Prism User's Guide V1.0 Beta-2*, Thinking Machines Corporation, Cambridge, MA, November 1991

[TiHe90] W.F. Tichy und C.G. Herter, *Modula-2* – An Extension of Modula-2 for Highly Parallel, Portable Programs*, Interner Bericht Nr. 4/90, Fakultät für Informatik, Universität Karlsruhe, Januar 1990

[ZeRu91] D. Zernik und L. Rudolph, *Animating Work and Time for Debugging Parallel Programs Foundation and Experience*, Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, in ACM SIGPLAN Notices 12/91, Seiten 46–56, Dezember 1991