Die approbierte Originalversion dieser Dissertation ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

DISSERTATION

Zeitanalyse von Echtzeitprogrammen

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften

Eingereicht an der Technischen Universität Wien,
Technisch-Naturwissenschaftliche Fakultät

von

Dipl.-Ing. Peter Puschner
Laxenburgerstr. 41/31, 1100 Wien
Matr.Nr. 8325870
geboren am 27. August 1965 in Wien

Wien, im Dezember 1993 .....

Zeitanalyse von Echtzeitprogrammen

Kurzfassung

Die Kenntnis der Abarbeitungszeiten von Programmen ist beim Entwurf und der Verifikation von Echtzeitsoftware von großer Bedeutung. Es werden daher Verfahren benötigt, mit denen genaue Aussagen über das Zeitverhalten von Programmen bzw. Programmteilen gemacht werden können.

Diese Arbeit stellt eine Methode zur Ermittlung knapper Schranken für die maximalen Abarbeitungszeiten von Programmen vor. Programmstücke werden in Form von Graphen repräsentiert, die die Struktur des Codes, das Zeitverhalten von sequentiellen Anweisungsfolgen und zusätzliche Informationen über das dynamische Ablaufverhalten, d.h. wie oft bestimmte Programmteile maximal abgearbeitet werden können, widerspiegeln. Schranken für die maximale Abarbeitungszeit werden aus dieser Repräsentation dadurch ermittelt, daß ein entsprechendes ganzzahliges lineares Optimierungsproblem generiert und gelöst wird. Die Zielfunktion beschreibt die Abarbeitungszeit, die sich aus der Summe aller mit der Anzahl der Ausführungen gewichteten Kantenausführungszeiten ergibt. Sie wird unter Restriktionen, die die Programmstruktur und Zusatzinformationen über das dynamische Verhalten des Programmes beschreiben, maximiert.

Das hier vorgestellte Verfahren zeichnet sich dadurch aus, daß durch die Verwendung von Benutzerinformation Zeitschranken einer sehr hohen Qualität ermittelt werden können. Weiters liefert es nicht nur die maximale Abarbeitungszeit von Programmen selbst, sondern auch detaillierte Informationen über die Abarbeitungszeiten aller Teilkonstrukte, sowie darüber, wie oft jeder einzelne Programmteil im Worst Case abgearbeitet wird. Letztere Informationen sind für die genaue Dokumentation des Zeitverhaltens und für Programmoptimierungen von großem Interesse.

Timing Analysis for Real-Time Programs

Abstract

The knowledge of program execution times is crucial for the development and the verification of real-time software. Therefore, there is a need for methods and tools which allow to predict the timing behavior of pieces of program code and entire programs.

This thesis presents a novel method for the analysis of program execution times. Programs are represented by graphs which reflect the structure and the timing behavior of the code and can be annotated with user-supplied information about infeasible paths. The graphs are searched for those execution paths which take the maximum time to execute. The search problem is transformed into a linear programming problem: The goal function, which has to be maximized, yields the execution time. Its constants represent the execution times of program parts, the variables the number of executions of these pieces of code. The restrictions of the programming problem describe the code structure and the infeasible paths.

The new timing analysis method is special in that it can be used to compute execution time bounds of high quality, i.e., very tight bounds. It does not only calculate execution time bounds, but also derives detailed information about the execution time and the number of executions of every single program construct.

Danksagung

Diese Arbeit entstand im Rahmen meiner Forschungs- und Lehrtätigkeit am Institut für Technische Informatik, Abteilung für Echtzeitsysteme und Softwaretechnologie, an der Technischen Universität Wien.

Besonders danken möchte ich dem Betreuer meiner Dissertation, Herrn o. Univ. Prof. Dr. Hermann Kopetz, der mir die Forschungstätigkeit am Institut für Technische Informatik ermöglichte. Er unterstützte meine Arbeit durch wertvolle Anregungen und stimulierende Diskussionen und prägte so meinen wissenschaftlichen Werdegang.

Weiters danke ich Prof. Dr. Alan Shaw von der University of Washington in Seattle, Washington, USA, und Prof. Dr. Al Mok von der University of Texas at Austin, Texas, USA, für ihr Interesse an meiner Arbeit und zahlreiche fruchtbare Diskussionen, die wir im Rahmen meiner Forschungsaufenthalte an deren Universitäten führten.

Ich möchte allen meinen Kollegen und Freunden am Institut für Technische Informatik für ihren Beitrag zur angenehmen und motivierenden Arbeitsatmosphäre danken. Besonderer Dank ergeht dabei an meinen Studienkollegen, langjährigen Zimmerkollegen und Freund Gerhard Fohler. Er stand mir stets als Ansprechpartner für neue Ideen zur Verfügung, lieferte mir zahlreiche Anregungen für meine Arbeit und teilte im privaten Bereich einen großen Teil der Zeit mit mir, in der wir gemeinsamen Interessen nachgingen und die Welt bereisten. Weiters bedanke ich mich bei Alexander Vrchoticky für die gute, intensive Zusammenarbeit während einiger Phasen meiner Arbeit, bei Emmerich Fuchs und Anton Schedl für wertvolle Anregungen und die Implementierung von Tools und bei Christian Koza für die interessanten fachlichen Gespräche, die wir gemeinsam führten. Zudem danke ich Gerhard Fohler, Anton Schedl und Alexander Vrchoticky für das Korrekturlesen einer ersten Fassung der Dissertation.

Meinen Eltern gebührt ein großer Teil meines Dankes. Sie ermöglichten mir das Studium und unterstützten mich während meiner gesamten Ausbildung. Meinen Freunden aus der Pfarre St. Johann in Wien 10, besonders meiner Freundin Marianne Schlögelhofer, danke ich dafür, daß sie mir besonders während der Erstellung meiner Dissertation Rückhalt boten und halfen, auch während des Dissertierens nicht auf die anderen wesentlichen Seiten des Lebens zu vergessen.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Ziel der Arbeit 2

1.2 Verwandte Arbeiten 3

1.3 Aufbau der Arbeit 7

2 Statische Tasklaufzeitanalyse **9**

2.1 Zeitanalyse 9

2.1.1 Schedulability Test und Laufzeitanalyse 10

2.1.2 Abgrenzung der Codezeitanalyse 11

2.2 Die Abarbeitungszeit 12

2.2.1 Hardwareeinflüsse auf die Abarbeitungszeit 13

2.2.2 Softwareeinflüsse auf die Abarbeitungszeit 15

2.3 Die maximale Abarbeitungszeit 15

2.3.1 Schranken für die maximale Abarbeitungszeit 16

2.3.2 Anforderungen an eine Methode zur Ermittlung einer $MAXT_C$. 18

2.4 Zusammenfassung 19

3 Verwandte Arbeiten **20**

3.1 Real-Time Euclid 20

3.2 Zeitanalyse mit Timetool und TAL 22

3.3 Berechnung von Abarbeitungszeiten mittels “Timing Schema” 24

3.4 Zeitanalyse für MARS 27

3.4.1 Untersuchung des Zeitanalyseansatzes 27

3.4.2 Integration der Zeitanalyse in eine Taskentwicklungsumgebung . 28

3.4.3 Bezug zur vorliegenden Arbeit 29
3.5 Zusammenfassung 30

4 Ermittlung von Schranken für die Abarbeitungszeit 31
4.1 Coderepräsentation für die Zeitanalyse 31
4.1.1 Der Zeitanalysegraph 32
4.1.2 Abarbeitungspfade und Abarbeitungszeiten 35
4.2 Die MAXT und Zirkulationen 36
4.2.1 T-Graphen und Zirkulationen 37
4.2.2 Flußrestriktionen auf Zirkulationen 40
4.2.3 Eigenschaften von Erweiterten T-Graphen mit Restriktionen . . 44
4.3 Zusammenfassung 48

5 Die Berechnung der Maximalen Abarbeitungszeit mittels ganzzahliger linearer Programmierung 50
5.1 Kurzer Überblick über die Lineare Programmierung 51
5.2 Die Berechnung der maximalen Kosten einer Zirkulation mit Restriktionen — Ein ganzzahliges lineares Programmierungsproblem 52
5.2.1 Formulierung des Zirkulationsproblems als Programmierungsproblem 53
5.2.2 Ganzzahligkeit von Lösungen 56
5.3 Ergebnisse der Programmierungsprobleme für die MAXT-Analyse . . . 56
5.3.1 MAXT-Berechnung und Sensitivitätsanalysen 58
5.4 Zusammenfassung 59

6 Verwendung der Theorie zur MAXT-Berechnung 60
6.1 Die MAXT-Analyse von Assemblerprogrammen 60
6.1.1 Annahmen 61
6.1.2 Umwandlung in ein Programmierungsproblem 62
6.2 Die MAXT-Analyse von Sprachkonstrukten der strukturierten Programmierung 63
6.2.1 Die Beschreibung der Transformation 67
6.2.2 Übersetzung der Konstrukte 70
6.2.3 Beispiel für die Übersetzung einer Prozedur 78

6.2.4 Rückführung von Zeitinformation in die Ausgangsrepräsentation 79
6.3 Berücksichtigung der Laufzeitüberprüfungen von Restriktionen 82
6.3.1 Knappere Schranke versus höhere Abarbeitungszeit 83
6.4 Zusammenfassung 85

7 Experimente **86**
7.1 Vorgehen bei den Experimenten 86
7.2 Beispiele und Daten 88
7.3 Ergebnisse der MAXT$_C$-Berechnungen 93
7.4 Laufzeitüberprüfungen von Restriktionen 98
7.5 Zusammenfassung 102

8 Zusammenfassung und Ausblick **104**

Literaturverzeichnis **107**

A Transformation eines Programmstückes in ein Programmierungsproblem **113**

B Sprache zur Beschreibung von Codestruktur und Zeitverhalten **126**

Kapitel 1

Einleitung

Bedingt durch die immer höher werdende Leistungsfähigkeit und fallende Kosten steigt der Einsatz von Computern in Meß-, Steuerungs- und Regelungsanwendungen. Diese Anwendungen sind dadurch gekennzeichnet, daß an das Computersystem nicht nur funktionale Anforderungen, sondern auch harte Anforderungen im Zeitbereich gestellt werden. Die zeitlichen Anforderungen werden von den Zeitkonstanten und der Dynamik der Umgebung des Computersystems bestimmt.

Man nennt Computersysteme, die zeitlichen Anforderungen gehorchen müssen, *Echtzeitsysteme*. Beispiele für Echtzeitsysteme sind Überwachungsanlagen für Atomkraftwerke, Flugzeugsteuerungen, Steuerungen von Lebenserhaltungssystemen, Antiblockiersysteme in Kraftfahrzeugen, sowie automatisierte Fertigungsstraßen. In Abhängigkeit von den Folgen eines Fehlers im Zeitverhalten unterscheidet man zwischen harten bzw. kritischen und nicht harten Echtzeitsystemen. Ein Echtzeitsystem heißt *hartes Echtzeitsystem*, wenn die Kosten eines Fehlers den Nutzen des Systems um Größenordnungen übersteigen können. Zu den Kosten zählen dabei sowohl finanzielle Einbußen und Folgekosten, als auch der Verlust von Menschenleben.

Die hohen Anforderungen an Echtzeitsysteme machen es notwendig, daß das Zeitverhalten dieser Computersysteme vorhersehbar und das Einhalten von Zeitschranken garantierbar ist. Um das zu gewährleisten kommt dem Faktor Zeit eine zentrale Rolle während des gesamten Softwareentwicklungsprozesses zu. Dies gilt einerseits für Design und Implementierung, andererseits aber auch für die Analyse und Verifikation der Korrektheit des Systems.

Wenn man Aussagen über das Zeitverhalten eines Computersystems machen will, muß man genau über die zeitlichen Abläufe in diesem System Bescheid wissen. Man muß wissen, in welcher Reihenfolge Prozesse abgearbeitet werden (bzw. werden können), wie oft und wann sie aktiviert werden, wie lange Prozesse Ressourcen benötigen und wann Prozesse miteinander bzw. mit der Umgebung kommunizieren. Nur wenn all diese zeitlichen Zusammenhänge im System bekannt sind, können Garantien über das Zeitverhalten des Gesamtsystems abgegeben werden.

Eine Grundvoraussetzung für die Bestimmung der zeitlichen Abläufe in einem Computersystem ist die Kenntnis der Abarbeitungszeiten der einzelnen Tasks sowie der System Calls und Systemroutinen. Im speziellen sind es die Worst Case Abarbeitungszeiten, d.h. die maximalen Abarbeitungszeiten, dieser Codestücke, die von zentralem Interesse sind. Kann man die Worst Case Abarbeitungszeit für alle Programmteile bestimmen und das geforderte Zeitverhalten des Gesamtsystems unter diesen Abarbeitungszeiten garantieren, so kann das geforderte Zeitverhalten in jeder von der Spezifikation abgedeckten Situation garantiert werden.

1.1 Ziel der Arbeit

In dieser Arbeit wird ein Verfahren zur Analyse des Zeitverhaltens von Codesegmenten vorgestellt. Die Idee dieses Ansatzes ist, durch eine Untersuchung des Programmcodes und die Verwendung von Zusatzinformationen über das dynamische Verhalten gute, *knappe Schranken* für die maximalen Abarbeitungszeiten (MAXTs) der Codestücke zu berechnen. Dabei geht es in erster Linie um den Softwareaspekt, d.h. wie bestimmte Sprachkonstrukte bzw. Zusammenhänge im Programmcode bei der Zeitanalyse berücksichtigt werden können. Wir nehmen eine einfache Hardwarestruktur mit flachem Speicher (keine Caches) und ohne parallele oder überlappend arbeitende Verarbeitungseinheiten (Parallele Recheneinheiten, Instruction Pipeline, etc.) an.

Zentrales Ziel ist es, eine hohe Qualität der MAXT-Analyse zu erreichen. Durch alleinige Analyse der statischen Programmstruktur, die durch die Anordnung der Konstrukte entsprechend den syntaktischen Regeln festgelegt wird, kann dies im allgemeinen nicht geschehen. Dazu fehlt dieser Beschreibung die semantische Information. Es ist daher notwendig, Wissen über das dynamische Verhalten des Codes, das sich aus den erlaubten bzw. möglichen Eingabedaten ableiten läßt, in die Zeitanalyse einzubringen. Mechanismen zur Beschreibung solcher Zusammenhänge werden bereitgestellt.

Das hier vorgestellte Verfahren ermittelt Schranken für die maximale Abarbeitungszeit mittels *ganzzahliger linearer Optimierung*: Ein Programm wird als eine Menge von Blöcken interpretiert, denen Abarbeitungszeiten zugeordnet sind. Restriktionen des Optimierungsproblems beschreiben die Struktur des Codes dadurch, daß sie die Anzahl der Abarbeitungen der einzelnen Blöcke zueinander in Beziehung setzen. Ebenfalls durch Restriktionen werden die oben genannten Zusammenhänge, welche die Codesemantik mit sich bringt, ausgedrückt. Die Berechnung der maximalen Abarbeitungszeit erfolgt durch Lösung der dem Problem zugeordneten Optimierungsaufgabe.

Das Verfahren zur Berechnung der maximalen Abarbeitungszeit läßt sich durch die folgenden Punkte charakterisieren:

- Abarbeitungszeiten werden aus Information über die Struktur des zu analysierenden Programmstückes und die maximalen Abarbeitungszeiten von einfachen,

sequentiellen Codeblöcken berechnet. Diese Vorgehensweise ist nicht darauf angewiesen, daß das zu analysierende Programmstück in einer speziellen Form vorliegt. Sie ist für unterschiedliche Darstellungen des Quellcodes, von der strukturierten Programmiersprache, über Zwischendarstellungen, bis zum Assemblercode reichend, anwendbar.

- Neben der statischen Codestruktur kann Zusatzwissen über das dynamische Ablaufverhalten in die MAXT-Analyse einbezogen werden. Auf diese Art kann eine hohe Qualität der Resultate erzielt werden.

- Die Notation zur Beschreibung der statischen Struktur und der zusätzlichen Restriktionen, die die datenabhängigen Einschränkungen für die Abarbeitungen ausdrücken, ist dieselbe. Zusatzinformationen über das dynamische Verhalten bedürfen daher bei der Berechnung der Zeitschranken keiner gesonderten Behandlung.

- Durch die Verwendung der linearen Optimierung zur Lösung des Problems kann der Aufwand für die Berechnung der MAXT gering gehalten werden. Im Vergleich zu den bisher verwendeten Verfahren [Pus93] entfällt die mühsame Suche nach dem Worst Case Pfad. Außerdem brachten die Arbeiten auf dem Gebiet des Operations Research leistungsfähige Softwarepakete zur Lösung von Optimierungsaufgaben hervor.

- Neben der Schranke für die maximale Abarbeitungszeit liefert das Verfahren sehr detaillierte Information über das Worst Case Zeitverhalten des analysierten Codes. Es liefert die Abarbeitungszeiten für einzelne Programmteile und dokumentiert den Beitrag einzelner Statements zum gesamten Worst Case.

1.2 Verwandte Arbeiten

Die Arbeiten auf dem Gebiet der Tasklaufzeitanalyse lassen sich in zwei Gruppen unterteilen. Auf der einen Seite gibt es Ansätze, die versuchen mit Hilfe von formalen Beweistechniken zu Aussagen über das Zeitverhalten von Codestücken zu gelangen. Dem gegenüber stehen Arbeiten, die weniger formal argumentieren und durch die Konstruktion und Anwendung von Tools die Anwendbarkeit ihrer Verfahren zeigen.

Formale Ansätze

Zu den Arbeiten, die formale Methoden verwenden, zählen [**?**, Sha79, Zed88]. Alle drei Autoren verwenden Dijkstras *Weakest Precondition* Calculus [Dij76], um Programmcode mit Zusicherungen zu versehen und diese Zusicherungen zu beweisen. Die drei Arbeiten sind von der grundlegenden Idee her sehr ähnlich, auch wenn sie sich in der Art der Präsentation und vom beschriebenen Detail her unterscheiden. Sie gehen davon

aus, daß ein Programmstück in einer Sprache vorliegt, deren funktionale und zeitliche Semantik mit Hilfe von Weakest Preconditions (*wp*) beschrieben werden kann (Haase verwendet die Guarded Command Notation, Shaw die Sprache Pascal und Zedan die Programmiersprache Occam). Der Faktor Zeit wird durch Hinzufügen einer eigenen Zeitvariable zum ursprünglichen *wp*-Modell eingeführt. Die eigentliche Analyse des Zeitverhaltens erfolgt mittels eines formalen Beweises, durch die Anwendung von definierten Prädikatentransformationsregeln auf das gegebene Programmstück.

Das Problem der Abschrankung von Schleifen wird sowohl in [**?**], als auch in [Sha79] behandelt. Haase unterscheidet zwei Fälle. Ist die maximale Anzahl von Iterationen einer Schleife aus dem Programmcode ableitbar, geht dieser Wert automatisch in den Beweis durch Prädikatentransformation ein. Ist die Iterationsanzahl jedoch von den Eingabedaten abhängig, so muß diese entweder in Form einer Funktion auf den Eingabedaten oder durch die direkte Angabe einer oberen Schranke der Analyse zur Verfügung gestellt werden. Shaw verlangt bei der Führung des Beweises in jedem Fall, daß die maximale Iterationsanzahl bzw. eine Approximation dieses Wertes vom Benutzer eingegeben wird.

Keiner der drei beschriebenen formalen Ansätze wurde weiter verfolgt. Ihre Anwendbarkeit wurde nur durch kleine Beispiele veranschaulicht. Eine umfassende Evaluation in der Praxis wurde von keinem der genannten Autoren durchgeführt.

Programmiersprachen und Tools

Während die oben beschriebenen Arbeiten Zeitanalyse als formale Verifikation betrachten, gibt es weniger formale Ansätze, die zur Entwicklung von Programmiersprachkonstrukten zur Beschreibung des Ausführungsverhaltens von Code, sowie Tools zur Codezeitanalyse führten.

Wei [Wei81] entwickelte ein Softwarewerkzeug, das zur Compilezeit obere Schranken für die Abarbeitungszeit von *Path Pascal* Programmen ermittelt. Diese Arbeit stützt sich auf [Sch78], wo Pascal-Schleifenkonstrukte um das Schlüsselwort *limit* zur Angabe der maximalen Anzahl von Iterationen einer Schleife erweitert wurde. Das Tool berechnet Schranken für die Abarbeitungszeiten von Prozeduren und Funktionen, die nicht rekursiv aufgerufen werden und keine goto-Statements enthalten.

Rehm [Reh87] untersucht Assemblercode von übersetzten C-Programmen auf sein Zeitverhalten. Dabei stellt er fest, daß es oft unbefriedigend ist, Schleifen nur durch ein Iterationslimit abzuschranken. Das Verfahren versucht daher schon zur Übersetzungszeit festzustellen, welche Teile des Codes zur Laufzeit tatsächlich zur Ausführung gelangen. Schleifen werden durch sogenannte *hints*, Variablen eines bestimmten Namens, denen die maximale Iterationsanzahl als Wert zugewiesen wird, abgeschrankt. Dasselbe gilt für Rekursionen, die bei Rehm durchaus verwendet werden dürfen. Die Zeitanalyse liefert bei Rehm nicht nur eine Schranke für die maximale Abarbeitungszeit, sondern detailliertere Informationen über die Ausführungszeiten von einzelnen

Funktionen oder Schleifenrümpfen. Über die Qualität der ermittelten Schranken wird allerdings keine Aussage gemacht.

Die für diese Arbeit richtungsweisendsten Arbeiten sind die von Kligerman und Stoyenko [Kli86, Sto87], Mok und Forschungsgruppe [Ame85, Che87, Mok89], Shaw und Park [Sha89, Par89, Par90, Par92, Par93] und der MARS-Gruppe [Pus89, Pus90, Kop91, Pus91, Pos92, Vrc92, Vrc93, Kop93, Pus93, Sch93]. Diese Arbeiten werden in Kapitel 3 genauer vorgestellt.

Berücksichtigung spezieller Hardware

Während die bisher genannten Arbeiten von einer einfachen Prozessorarchitektur ohne Caches, Pipelines und Prefetchingmechanismen ausgehen und sich primär auf den Softwareaspekt der Zeitanalyse konzentrieren, gibt es auch Bestrebungen, die speziellen Features moderner Prozessoren in die Zeitanalyse miteinzubeziehen. Obwohl die Berücksichtigung spezieller Hardware bei der Zeitanalyse nicht Zielsetzung dieser Arbeit ist, sollen hier doch die Arbeiten, die dieses Ziel haben, kurz erwähnt werden.

Im Rahmen des Spring-Projekts [Sta87, Sta91] stellt Niehaus ein Zeitanalyseverfahren vor, das mögliche Cache Hits bzw. Cache Misses eines Instruktionscaches bei der Zeitanalyse berücksichtigt [Nie91b, Nie91a]. Es wird angenommen, daß die analysierten Codeeinheiten zur Laufzeit nicht unterbrochen werden können, sodaß keine Cache Misses durch konkurrierende Tasks untersucht werden müssen. Cache Misses werden nur beim erstmaligen Zugriff bzw. dadurch, daß verschiedene Codeteile eines Tasks auf denselben Adreßbereich im Cache abgebildet werden, verursacht. Durch die Ausnutzung des Wissens über das Hit/Miss-Verhalten des Cache können Schranken für die Abarbeitungszeit berechnet werden, die näher an der wahren maximalen Abarbeitungszeit liegen, als wenn man pessimistischerweise annimmt, daß jeder Zugriff auf einen Instruktionsblock auf den langsamen Hauptspeicher erfolgen muß.

Ebenfalls mit der Problematik von Instuktionscaches setzt sich [Fuc93] auseinander. Diese Arbeit basiert auf der in [Sch93] und dieser Arbeit vorgestellten Methode zur Berechnung von Schranken der maximalen Abarbeitungszeit. Es wird ein Problem der ganzzahligen linearen Optimierung konstruiert, das neben der Struktur und dem Zeitverhalten von einzelnen Instruktionen auch die Eigenschaften eines Instruktionscaches repräsentiert. Im Gegensatz zur Arbeit [Sch93] liegt der Schwerpunkt in [Fuc93] auf der Berücksichtigung von Hardwareeigenschaften. Auf der Softwareseite werden nur die einfachsten Konstrukte von Programmiersprachen, Sequenzen, If-Statements und Schleifen, berücksichtigt.

Harmon [Har91, Har92] stellt eine Zeitanalysemethode vor, bei der parallele Verarbeitungseinheiten einer CPU, Instruction Pipelining und Caching betrachtet werden. Die Berechnung der Schranken für die maximale Abarbeitungszeit erfolgt in zwei Teilschritten. Zunächst werden für alle Instruktionssequenzen eines Programmes die maximalen Abarbeitungszeiten ermittelt. Dabei werden die Eigenschaften der genannten

Hardwaremechanismen berücksichtigt: Alle Instruktionen werden in Mikroinstruktionen, die den verschiedenen Hardwareeinheiten zugeordnet werden können, unterteilt. Die Ausführung dieser Mikroinstruktionen wird dann unter Berücksichtigung möglicher Parallelausführungen simuliert und so das Zeitverhalten für die Sequenzen ermittelt. Im zweiten Schritt wird mit dem Wissen über den Kontrollfluß zwischen den sequentiellen Blöcken eine Schranke für die Gesamtabarbeitungszeit berechnet.

Zhang, Burns und Nicholson [Zha93] berechnen Zeitschranken für Programme, die auf dem 80C188 Prozessor ablaufen. Der Prozessor besitzt eine Instruktionspipeline, die parallel zur Instruktionsausführung mit bis zu vier Bytes gefüllt werden kann. Der Performancegewinn, der durch das parallele Ausführen und Laden von Instruktionen erzielt werden kann, wird in der Zeitanalyse auf Basic Block Ebene berücksichtigt. Vergleiche der Ergebnisse dieser Zeitanalysemethode mit tatsächlichen Abarbeitungszeiten bzw. mit Analysemethoden, bei denen die Parallelität nicht in die Berechnung eingeht, dokumentieren die Qualität der Ergebnisse des beschriebenen Verfahrens.

Neue Hardwarekonzepte

Das Problem bei der Vorhersage des Zeitverhaltens von Systemen mit Caches, Pipelines, etc. liegt in der Größe des Zustandsraumes von möglichen Szenarien bzw. in ereignisgesteuerten Computersystemen auch in der Unvorhersehbarkeit der Zeitpunkte, zu denen Unterbrechungen (Interrupts) zu einer Veränderung des Zustands bzw. Inhalts dieser Hardwarebausteine führen.

In den letzten Jahren entstanden einige Arbeiten, die versuchen, dieser schweren Analysierbarkeit des Verhaltens von Computersystemen durch die Entwicklung spezieller Hardwarearchitekturen beizukommen. Als Vertreter dieser Arbeiten seien hier [Kir89, Kir90, Cog91, Lin91, Age93] genannt.

Die Arbeiten von Kirk et al. [Kir89, Kir90] bzw. Cogswell und Segall [Cog91] streben eine Trennung der Kontexte verschiedener Tasks an. Kirk et al. versuchen, das Verhalten von Zugriffen auf hierarchischen Speicher deterministischer zu machen, indem sie den Tasks eigene Cachesegmente zuweisen, die von anderen Tasks nicht verändert werden können. Cogswell und Segall verwenden einen flachen, aufgeteilten Speicher, mit einer einheitlichen Zugriffszeit. Sie erreichen eine Erhöhung des Durchsatzes bei vorhersagbarem Zeitverhalten dadurch, daß ihr Prozessor zyklisch einen von mehreren Kontexten bearbeitet, während die anderen Kontexte auf den Abschluß von Speicheroperationen warten.

Neue Ideen zur Verwaltung von hierarchischem Speicher werden in [Age93] präsentiert. Es werden zwei Verfahren vorgestellt, die Tasks bzw. Teile davon bereits vor deren Abarbeitung vom langsamen Hauptspeicher in einen kleinen, schnellen Pufferspeicher laden. Damit kann im Gegensatz zu Cachespeichern, wo die Ladeentscheidung für Blöcke erst zur Laufzeit getroffen wird, ein deterministisches Zeitverhalten erreicht und eine leichtere Analysierbarkeit erzielt werden.

Lindh und Stanischewski [Lin91] entwickelten einen eigenen Prozessor, auf dem die zentralen Betriebssystemfunktionen in Hardware realisiert sind. Mit diesem Prozessor können kleine Task Sets unter deterministischen Bedingungen abgearbeitet werden.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit ist in acht Kapitel gegliedert. Das folgende Kapitel grenzt den Aufgabenbereich der statischen Tasklaufzeitanalyse ein. Es behandelt Einflußfaktoren, Voraussetzungen und Annahmen für die Berechnung von Schranken für die maximale Abarbeitungszeit. Die Anforderungen an die Tasklaufzeitanalyse werden formuliert.

Kapitel 3 diskutiert zur vorliegenden Arbeit verwandte Arbeiten im Hinblick auf die in Kapitel 2 gemachten Annahmen und Anforderungen. Damit werden dem Leser Umfeld und Motivation für diese Arbeit nähergebracht.

In der Folge werden die theoretischen Grundlagen für den hier vorgestellten Ansatz der Berechnung von Abarbeitungszeiten (bzw. deren Schranken) erörtert. Die für die Zeitberechnung relevanten Informationen über die Struktur und das Zeitverhalten eines Programmstückes werden durch Graphen repräsentiert, die Zeitberechnung zuerst auf das graphentheoretische Problem der Bestimmung einer maximalen Zirkulation auf einem gerichteten Graphen zurückgeführt. Hierauf wird eine allgemeinere Methode zur Beschreibung der Flüsse auf Kanten vorgestellt. Diese erlaubt es, Flüsse verschiedener Kanten zueinander in Relation zu setzen. Damit wird es möglich, das Verhalten von Programmen, die maximale Anzahl von Iterationen von Schleifen und sonstige Zusammenhänge, so genau zu beschreiben, daß grundsätzlich für jedes Stück Programmcode nicht nur eine obere Schranke für die maximale Abarbeitungszeit, sondern die maximale Abarbeitungszeit selbst, berechnet werden kann.

Kapitel 5 zeigt, wie das beschriebene Zirkulationsproblem mit Hilfe der ganzzahligen linearen Optimierung gelöst werden kann. Die optimalen Lösungen einer Optimierungsaufgabe liefern die Schranke für die maximale Abarbeitungszeit und erlauben Aussagen über das Zeitverhalten des durch den Graphen repräsentierten Codestückes.

Programme oder Programmstücke, deren Zeitverhalten analysiert werden soll, liegen meist nicht unmittelbar in der geeigneten Repräsentation vor. In Kapitel 6 wird daher gezeigt, wie man aus Assemblercode bzw. aus einer Repräsentation, die das Zeitverhalten auf der Source Code Ebene einer höheren Programmiersprache ausdrückt, Graphen und entsprechende ganzzahlige lineare Optimierungsaufgaben konstruiert. Dabei wird sowohl die Information über die statische Struktur, als auch Information über das dynamische Verhalten in Form von Scopes, Markern und Restriktionen auf Markern, transformiert. Außerdem wird die Möglichkeit diskutiert, Iterationsschranken für Schleifen und Restriktionen auf Markern zur Laufzeit zu überprüfen. Der zusätzliche Zeitaufwand für diese Kontrolle muß dann ebenfalls in die Zeitanalyse eingehen.

Kapitel 7 evaluiert den beschriebenen Ansatz mittels Experimenten. Die Abarbei-

tungszeiten von Prozeduren werden analytisch und durch Simulationen ermittelt und verglichen. Die Zielsetzung dieser Experimente geht in zwei Richtungen. Einerseits wird untersucht, wie gut man in der Praxis das Laufzeitverhalten eines Programmstückes mit den vorhandenen Mitteln beschreiben kann, d.h. wie knapp berechnete Abarbeitungszeiten an die tatsächliche maximale Abarbeitungszeit herankommen. Auf der anderen Seite wird errechnet, wie hoch Kosten für die Laufzeitüberprüfungen von Iterationsgrenzen und Restriktionen sind.

Den Abschluß der Arbeit bildet Kapitel 8. Dieses Kapitel faßt die Arbeit und ihre zentralen Aussagen zusammen und gibt einen Überblick über mögliche, auf dieser Arbeit aufsetzende Forschungsschwerpunkte.

Kapitel 2

Statische Tasklaufzeitanalyse

Jeder strukturierte Softwareentwicklungsprozeß kann in mehrere Phasen, bei denen sich konstruktive Schritte (design creation) und Evaluierungsschritte (design verification) abwechseln, unterteilt werden [?, ?, ?]. Ein wesentliches Merkmal der Entwicklung von Echtzeitsoftware ist, daß die Zeiteinen zentralen Punkt sowohl bei der Planung und Realisierung, als auch bei der Verifikation der Funktionsfähigkeit der Software spielt. Die Zeitanalyse ist daher ein fester Bestandteil der Validierung eines Echtzeitsystems, die statische Tasklaufzeitanalyse wiederum ein Teil davon.

In diesem Kapitel soll zunächst der Aufgabenbereich der statischen Tasklaufzeitanalyse innerhalb der globalen Zeitanalyse definiert werden. Es folgt eine Diskussion der Einflußfaktoren und Voraussetzungen für die Berechnung von Schranken für Taskabarbeitungszeiten. Den Abschluß dieses Kapitels bildet die Formulierung von Anforderungen an die statische Tasklaufzeitanalyse.

2.1 Zeitanalyse

Bei der Zeitanalyse eines Echtzeitsystems wird untersucht, ob das System alle spezifizierten Zeitanforderungen erfüllt. Es geht darum zu verifizieren, daß das System auf alle Beobachtungen von relevanten Ereignissen bzw. Stimuli aus der Umgebung innerhalb eines von der Außenwelt vorgegebenen Zeitintervalls auf geeignete Art reagiert. Mit relevanten Ereignissen bzw. Stimuli sind dabei all jene Ereignisse gemeint, die laut Spezifikation eine Reaktion des Systems erfordern. Wie das System beim Auftreten dieser Ereignisse zu reagieren hat, wird durch die Anforderungen an das System klar definiert.

Die Verifikation des Zeitverhaltens des Systems dient dazu, die Vorhersehbarkeit (predictability) des Computersystems im Zeitbereich zu zeigen [Sta90] und Aussagen über die Zuverlässigkeit (dependability [Lap92]) machen zu können. Für harte Echtzeitsysteme, wie sie hier betrachtet werden sollen, bedeutet das, daß gezeigt werden

muß, daß alle harten Zeitschranken (deadlines) garantiert eingehalten werden können, d.h. daß Antworten rechtzeitig erfolgen. Dabei sei explizit darauf hingewiesen, daß die geforderte Rechtzeitigkeit auf keinen Fall mit hoher Geschwindigkeit bzw. Performance gleichgesetzt werden darf, wie das immer wieder getan wird [Sta88]. Rechtzeitig heißt "innerhalb der vorgegebenen Zeitschranken" und nicht "so schnell wie möglich".

Prinzipiell lassen sich zwei Ansätze der Verifikation unterscheiden: statische Verifikation und dynamische Verifikation [Lap92]. Die beiden Ansätze unterscheiden sich dadurch, daß bei der dynamischen Verifikation das untersuchte System exekutiert wird, bei der statischen Verifikation nicht. Wichtiger Vertreter der dynamischen Verifikation des Zeitverhaltens ist das Testen [Sch92]. Darauf soll in dieser Arbeit allerdings nicht näher eingegangen werden. Zu den statischen Verifikationsverfahren für Echtzeitsysteme zählen alle "Tests", die anhand der Beschreibung von Zeitparametern des Systems (Tasks, Taskperioden, Deadlines, etc.) bzw. statischer Codeanalyse feststellen, ob das geforderte Zeitverhalten eingehalten werden kann.

2.1.1 Schedulability Test und Laufzeitanalyse

Unterscheidet man zunächst einmal nicht zwischen Schedulability Tests (bzw. globaler Zeitanalyse) und der Codelaufzeitanalyse, so stellt sich das Problem der statischen Zeitanalyse wie folgt:

> *Gegeben* ist die Beschreibung von (komplexen) zeitlichen Anforderungen an ein Kontrollsystem, sowie detaillierte Information über die Hardware und Software (einschließlich programmiertem Code bzw. Codebeschreibung), mit der dieses Kontrollsystem realisiert werden soll.
> *Gesucht* ist die Aussage, ob das Computersystem die gestellten Anforderungen erfüllt.

Obwohl es denkbar ist, die Zeitanalyse für ein Computersystem in einem einzigen, umfassenden Schritt durchzuführen, der sowohl die Abarbeitungszeiten für Codestücke in allen möglichen Situationen als auch Abhängigkeiten und Zusammenhänge zwischen Codestücken berücksichtigt, wird das Problem meistens in zwei unabhängige Teilaufgaben aufgespalten: die Laufzeitanalyse für Codestücke und den Schedulability Test. Bei der Laufzeitanalyse werden Stücke von Programmcode, die frei von blockierenden Statements sind, auf ihr Worst Case Zeitverhalten untersucht. Der Schedulability Test prüft, ob die Tasks mit ihren berechneten Abarbeitungszeiten, Precedence Constraints und Ressourcenanforderungen auf dem gegebenen Computersystem so abgearbeitet werden können, daß das geforderte Zeitverhalten garantiert werden kann.

Neben Überlegungen des Software Engineering und der Komplexität der Analyse sind es auch "historische" Gründe, die zur Aufteilung der Zeitanalyse führten. Als

man begann, sich mit der Zeitanalyse von Echtzeitsystemen auseinanderzusetzen, konzentrierte man sich ausschließlich auf die Problematik des Scheduling. Die Abarbeitungszeiten für Tasks bzw. Codesegmente wurden als bekannt angenommen, ohne darauf einzugehen, wie diese ermittelt würden [Liu73, Lei80, Lei82, Lei86, Mok84]. Bei der Zeitanalyse von Real-Time Euclid Programmen [Kli86] beschäftigte man sich erstmals damit, wie man die Laufzeitanalyse von Codesegmenten in die globale Zeitanalyse einbeziehen kann. Erst in den folgenden Jahren begannen dann vermehrt Forschungsgruppen auf dem Gebiet der Tasklaufzeitanalyse zu arbeiten (siehe z.B. [Mok89, Pus89, Sha89, Har91]). Die wichtigsten Forschungsarbeiten werden in Kapitel 3 näher vorgestellt und verglichen.

2.1.2 Abgrenzung der Codezeitanalyse

Wir haben oben eine grobe Unterteilung der Zeitanalyse in die Analyse der Abarbeitungszeit von Codestücken und den Schedulability Test bzw. das Scheduling vorgenommen. Die Einheiten des Codes, die bei der Codelaufzeitanalyse untersucht werden, sollen hier genauer identifiziert werden.

Wir betrachten ein Computersystem als eine Funktionseinheit aus Hardware und Software, die mit der Umgebung kommuniziert (siehe Abbildung 2.1). Die Hardware setzt sich aus einem Prozessor, dem Speicher, dem Adreß- und/oder Datenbus, sowie Bausteinen für das Memory Management, die Ein-/Ausgabe, usw. zusammen. Zur Software zählen das Betriebssystem, sowie alle System- und Anwendungsprogramme.

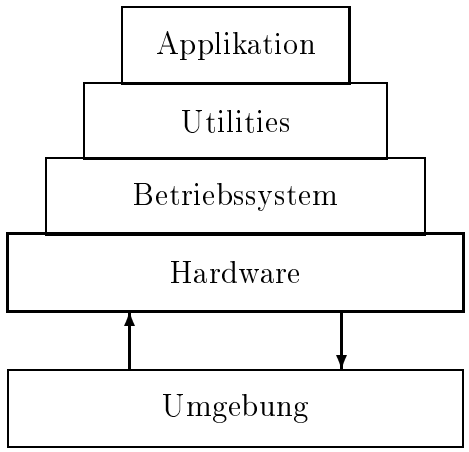


Abbildung 2.1: Hardware-/Softwaremodell eines Computersystems (modifiziert aus [Sta92])

Das dynamische Verhalten des Computersystems läßt sich durch die Aktivitäten, die auf den unterschiedlichen Bausteinen des Computersystems ablaufen, charakteri-

sieren. Sie werden auf der einen Seite durch die Instruktionen der verschiedenen Softwarekomponenten (Betriebssystem und Programme) und auf der anderen Seite durch die Aktivitäten der Hardware, die zum Teil durch Ereignisse der Umgebung getriggert werden (Interrupts, DMA, Memory Refresh, etc.), bestimmt.

Verschiedene Aktivitäten der Hardware und Software des Systems stehen miteinander in Wechselwirkung bzw. beeinflussen sich gegenseitig:

- *Explizite Wechselwirkungen* ergeben sich durch Interprozeßkommunikation, zeitliche Abhängigkeiten oder Anforderungen für die Exekution eines Codestückes, bzw. beim Datenaustausch mit der Umgebung.
- *Implizite Wechselwirkungen* resultieren aus Konflikten um Ressoucen (CPU, Bus, Memory), wenn diese von mehreren Aktivitäten gleichzeitig angefordert werden.

Die Wechselwirkungen zwischen den Aktivitäten führen dazu, daß gestartete Aktivitäten unterbrochen und verzögert werden. Außerdem müssen für die Auflösung von Ressourcekonflikten, egal ob diese in Hardware oder Software realisiert wird, zusätzliche Aktivitäten (Scheduling bzw. Arbitration) gestartet werden, wodurch das Zeitverhalten des Gesamtsystems weiter beeinflußt wird.

Die Betrachtung des beschriebenen Modells führt zur Unterscheidung der folgenden Faktoren, die sich auf das Zeitverhalten eines Computersystems auswirken:

- Hardwareaktivitäten (exklusive Verzögerungen oder Wechselwirkungen),
- Softwareaktivitäten (exklusive Verzögerungen oder Wechselwirkungen),
- Verzögerungen und Unterbrechungen von Aktivitäten,
- Arbitration.

Die Codelaufzeitanalyse beschäftigt sich genau damit, Aussagen über das Zeitverhalten der Softwareaktivitäten (Programmstücke von Anwendungsprogrammen bzw. Betriebssystemtasks) zu machen. Sie orientiert sich dabei ausschließlich am Code des untersuchten Programmstückes. Jegliche Verzögerungen durch Unterbrechungen, Kommunikation, oder Ressourcekonflikte sind von diesem Teil der Zeitanalyse ausgeschlossen. Es wird angenommen, daß letztere getrennt, bei der globalen Zeitanalyse des Gesamtsystems, im Rahmen des Schedulability Tests berücksichtigt werden.

2.2 Die Abarbeitungszeit

Entsprechend der im letzten Abschnitt vorgenommenen Eingrenzung ist die folgende Definition der *Abarbeitungszeit* zu verstehen, die etwas modifiziert aus [Vrc89] übernommen wurde.

Def.: Die *Abarbeitungszeit* eines Codestückes auf einem Computersystem ist jene Zeit, die der Prozessor des Computersystems mit der Abarbeitung dieses Codestückes zubringt.

Die Abarbeitungszeit eines Programmstückes hängt von einer Vielzahl von Parametern der verwendeten *Hardware* und *Software* ab. Außerdem bestimmen die *Eingabedaten* die jeweilige Exekutionszeit einer Ausführung des Programms. Die wichtigsten Einflußfaktoren von Hardware und Software werden hier aufgelistet.

2.2.1 Hardwareeinflüsse auf die Abarbeitungszeit

Die folgenden Parameter der Hardware eines Computersystems bestimmen die Abarbeitungszeit von Codestücken:

- Der *Prozessor*. Die Entwicklung der letzten Jahre hat eine enorme Steigerung der Leistungsfähigkeit von Mikroprozessoren gebracht. Moderne Technologien, mit hohen Taktraten und einem hohen Maß an Parallelität bei der Instruktionsverarbeitung, ermöglichen es, Codestücke mit großer Geschwindigkeit abzuarbeiten.

- *Pipelining*. Das Pipelining nutzt die Tatsache, daß die Ausführung von Instruktionen in mehrere Teiloperationen zerlegt werden kann. Diese Operationen – Laden und Dekodieren der Instruktion, Laden der Operanden, sowie die eigentliche Befehlsausführung – werden von verschiedenen spezialisierten Verarbeitungseinheiten des Prozessors überlappend durchgeführt.

- *Instruction Prefetching*. Beim Instruction Prefetching werden Instruktionen aus dem Speicher geladen, bevor diese tatsächlich vom Prozessor angefordert werden. Dies dient dazu, die Wartezeit auf Instruktionen zu verkürzen und die Abarbeitung zu beschleunigen.

- *Zeitverhalten des Speichers*. Das Verhältnis der Geschwindigkeiten von Prozessor und Speicher bestimmt, ob der Prozessor bei einem Speicherzugriff warten muß, bis der entsprechende Datentransfer abgeschlossen ist. Bei der Verwendung eines langsamen Speichers muß der Prozessor sogenannte *Wait States* einfügen, ehe er mit der Verarbeitung fortfahren kann.

- *Speicherverwaltung*. Das Memory Management dient dazu, virtuelle Adressen in physikalische Adressen zu übersetzen, ggf. das Laden eines Speicherbereichs vom Sekundärspeicher zu initiieren, und die Zugriffsberechtigung auf Speicherbereiche zu überprüfen. Die Speicherverwaltung kann sowohl softwaremäßig, als auch hardwaremäßig realisiert werden. In jedem Fall schlägt sie sich auf das Zeitverhalten von Speicherzugriffen nieder.

- *Cache-Speicher*. Caches sind kleine, schnelle Speicher, die einen kleinen Teil des gesamten Speicherinhalts aufnehmen. Die Zugriffszeiten auf den Cachespeicher sind um einiges kürzer als die des Hauptspeichers. Caches werden verwendet, um die Abarbeitungszeiten von Programmen zu senken.

- *Ein-/Ausgabebausteine*. Bedingt durch Unterschiede in der verwendeten Technologie und der Implementierung der Zugriffsoperationen auf interne Register unterscheiden sich die Zugriffszeiten von Ein-/Ausgabebausteinen. Entsprechend der Anzahl der Operationen wirken sich daher Ein-/Ausgaben auf die gesamte Ausführungszeit aus.

- *Direct Memory Access* (DMA). Beim Datentransfer über DMA wird die Kontrolle über den Bus zeitweise von der CPU an eine andere Einheit übergeben. Diese Einheit führt dann einen Datentransfer mit höherem Durchsatz durch, als dies mit den Lese- oder Schreiboperationen des Prozessors möglich wäre. DMA-Transfers können sowohl synchron, vom Prozessor initiiert, als auch asynchron, durch einen Interrupt getriggert, erfolgen.

Annahmen für diese Arbeit

In dieser Arbeit geht es weniger darum, alle Finessen moderner Prozessoren bei der Berechnung der Abarbeitungszeit zu berücksichtigen, als darum, möglichst viel Information über die zulässigen Pfade in den Prozeß der Ermittlung einer Schranke für die maximale Abarbeitungszeit einfließen zu lassen. Es werden daher folgende Annahmen über die Hardwareeigenschaften gemacht:

- Instruktionen werden streng sequentiell, ohne Überlappung, ausgeführt. Bezogen auf die oben genannten Hardwareeigenschaften heißt das, daß superskalare Prozessorarchitekturen mit mehreren parallel arbeitenden Verarbeitungseinheiten, Pipelining, Prefetching und Caching nicht berücksichtigt werden.

- Für jede Instruktion des Prozessors kann aus dem Wissen über die Operation und die Operanden die Ausführungszeit, bzw. eine obere Schranke für die Ausführungszeit bestimmt werden.

- Die Ausführungszeiten von Instuktionen sind unabhängig von Vorgänger- oder Nachfolgeinstruktionen. Die Abarbeitungszeit einer Folge von Instruktionen erhält man durch Summation der Abarbeitungszeiten der einzelnen Instruktionen.

- Die Zugriffszeiten für den gesamten adressierten Speicherbereich und alle verwendeten Ein-/Ausgabebausteine sind nach oben hin abschrankbar und bekannt.

- Die Zeiten, die für die Speicherverwaltung benötigt werden, sind bekannt. Es wird angenommen, daß Programme, die abgearbeitet werden sollen, vorher zur Gänze in den Hauptspeicher geladen werden (keine virtuelle Speicherverwaltung).

- DMA erfolgt ausschließlich synchron. Die Protokollausführungszeiten für die Übergabe der Kontrolle über den Bus, die Übertragungsgeschwindigkeit und die Menge der bei jedem einzelnen Transfer zu übertragenden Daten sind bekannt.

2.2.2 Softwareeinflüsse auf die Abarbeitungszeit

Auch auf ein und derselben Hardware kann ein Programmstück, das eine gegebene Aufgabe mit einer bestimmten Menge von Eingabedaten lösen soll, mit sehr unterschiedlichen Abarbeitungszeiten ausgeführt werden. Abhängig von der Abstraktionsebene gibt es eine Zahl von Softwarefaktoren, die die Abarbeitungzeit eines Codestückes beeinflussen.

- Betrachtet man ein Programm auf der untersten Softwareebene, auf der Ebene der Maschineninstruktionen, so bestimmen die Anzahl und Art der *Instruktionen*, die verwendet werden, um die gewünschte Funktionalität zu erzielen, die Ausführungszeit des entsprechenden Programmstückes.

- Bei Codestücken, die in einer höheren Programmiersprache implementiert werden, wirken sich der Sprachumfang der *Programmiersprache* bzw. die Regeln, nach denen der verwendete *Compiler* Maschinencode erzeugt, auf das Laufzeitverhalten aus.

- Der *Algorithmus*, der zur Lösung einer Aufgabe gewählt wird, bzw. dessen *Implementierung* bestimmen ebenfalls die Abarbeitungszeit eines Codestückes.

  Die Wahl des Algorithmus zur Lösung eines Problems orientiert sich an Annahmen, die über die Eingabedaten gemacht werden, sowie qualitativen Anforderungen (Speicherplatzbedarf, Abarbeitungszeit, etc.), die eingehalten werden sollen. Es sei in diesem Zusammenhang noch einmal betont, daß das zentrale Ziel bei der Lösung von Echtzeitproblemen nicht das Erreichen einer hohen Performance (kurzen durchschnittlichen Abarbeitungszeit) ist. Vielmehr kommt es darauf an, daß die maximalen Abarbeitungszeiten von Codestücken kurz und knapp abschrankbar sind.

2.3 Die maximale Abarbeitungszeit

Ziel der statischen Zeitanalyse für Codestücke ist es, Aussagen über die maximalen Abarbeitungszeiten dieser Programmteile zu machen. Die Faktoren, die die Ausführungs-

zeit von Programmcode beeinflussen, wurden im vorigen Abschnitt diskutiert. Wir definieren nun die maximale Abarbeitungszeit.

Def.: Die *Maximale Abarbeitungszeit* (*MAXT* ... **Ma**ximum e**X**ecution **T**ime) eines Codestückes auf einem Computersystem ist jene Zeit, die der Prozessor des Computersystems bei gegebenem Eingaberaum maximal mit der Abarbeitung dieses Codestückes zubringt.

Der Einfluß von Hardware und Software auf die Ausführungszeit wurde bereits erläutert. Die Eingabedaten beeinflussen die Abarbeitungszeit dadurch, daß sie den Kontrollfluß der Ausführung eines Programmstückes über die Wahrheitswerte von Bedingungen in Verzweigungen und Schleifen bestimmen. Das heißt, die maximale Abarbeitungszeit eines Codestückes ist in sehr hohem Maße applikationsspezifisch. Für ein und dasselbe Stück Code können beispielsweise Unterschiede in der Datenmenge (Größe eines Arrays, etc.), der Verteilung von Datenwerten oder der Werte von Variablen (z.B. in Abbruchbedingungen von iterativen Näherungsverfahren) zu sehr verschiedenen maximalen Abarbeitungszeiten führen.

2.3.1 Schranken für die maximale Abarbeitungszeit

Obwohl es wünschenswert wäre, maximale Abarbeitungszeiten für Codestücke einer Applikation exakt berechnen zu können, ist dies in der Praxis nur selten möglich. Die Gründe dafür sind folgende:

- Das Wissen über das Verhalten der Hardware ist unvollständig. So variieren z.B. die Abarbeitungszeiten von einigen Maschineninstruktionen der betrachteten Klasse von Prozessoren in Abhängigkeit von den Operanden; der Zusammenhang zwischen den Werten der Operanden und der Ausführungszeit ist nicht dokumentiert (siehe Multiplikation und Division auf dem M68000 [**?**]).
- Die Abbildung von Code einer höheren Programmiersprache auf Maschinencode ist nicht komplett nachvollziehbar. Will man dennoch eine Zeitanalyse auf Programmiersprachenebene durchführen, wie dies in [Par90, Par92] erfolgt, so muß man die Zeiten zum Teil sehr defensiv abschranken.
- Die Einflüsse des Eingaberaumes auf die Abarbeitungszeiten können nur zum Teil berücksichtigt werden. Das Problem liegt dabei einerseits darin, den Eingaberaum genau zu beschreiben, andererseits darin, dem Zeitanalyseprogramm das Wissen über die Eingabedaten oder deren Auswirkungen auf mögliche bzw. unmögliche Abarbeitungspfade [Par90] in geeigneter Form zur Verfügung zu stellen.

Auf Grund dieser Probleme muß man in der Praxis bei der Berechnung von Abarbeitungszeiten mit beschränktem bzw. approximiertem Wissen das Auslangen finden.

Was man damit berechnen kann sind obere Schranken für maximale Abarbeitungszeiten.

Def.: Die *Berechnete Maximale Abarbeitungszeit* ($MAXT_C$ ... **C**omputed Maximum Execution Time) eines Codestückes ist eine rechnerisch ermittelte obere Schranke für die maximale Abarbeitungszeit dieses Codestückes.

Die berechnete maximale Abarbeitungszeit hängt ab von Hardware, Software und Eingaberaum, sowie dem über diese Faktoren verfügbaren Wissen und davon, inwieweit diese Informationen bei der Berechnung genutzt werden.

Da es, außer durch Simulation der möglichen Abarbeitungen, im allgemeinen nicht möglich ist, aus einer Beschreibung der Eingabedaten direkt auf das Verhalten eines Programmstückes zu schließen, gehen wir davon aus, daß das Wissen über die Eingabedaten nicht direkt, sondern durch eine Beschreibung der Auswirkungen der Eingabedaten auf das Programmverhalten in die Berechnung von Schranken für die maximalen Abarbeitungszeiten eingeht. Das bedeutet, daß zum Programmstück, das analysiert werden soll, Information über mögliche bzw. unmögliche Programmabarbeitungen in Form von *Loop Bounds* oder sonstigen Konstrukten [Pus89] hinzugefügt wird.

Die folgenden Anforderungen müssen erfüllt werden, damit eine Schranke für die maximale Abarbeitungszeit berechnet werden kann:

- Die Abarbeitungszeiten der primitiven Einheiten der Zeitanalyse, seien es Ausführungszeiten von Maschineninstruktionen [Mok89, Pus89] oder von Programmblöcken [Par90, Par92], sind nach oben hin abschrankbar und die Schranken sind bekannt.

  Die Qualität des Resultats der Zeitanalyse hängt natürlich davon ab, wie knapp die Abarbeitungszeiten der genannten primitiven Einheiten abgeschrankt werden bzw. werden können. So ist es zum Beispiel bei einigen Maschineninstruktionen der betrachteten Klasse von Prozessoren der Fall, daß die Abarbeitungszeit der Instruktion nicht nur von den Operandentypen bestimmt wird, sondern auch in Abhängigkeit vom Wert der Operanden variiert (z.B. bei den Operationen Shift und Rotate des M68000 [**?**]). Sind der Zusammenhang zwischen Wert der Operanden und Ausführungszeit der Instruktion und die tatsächlichen Operandenwerte zur Zeit der Analyse bekannt, so kann die Abarbeitungszeit der Instruktionen genau angegeben werden. Anderenfalls kann sie nur mit der maximalen Befehlsausführungszeit nach oben abgeschrankt werden.

  Ähnliches gilt für die Analyse auf höherem Programmiersprachniveau. Sehr knappe Schranken für die Basisblöcke der Zeitanalyse können angegeben werden, wenn sehr viel Information über die Arbeitsweise des Compilers vorhanden ist. Ist diese Information schwer zugänglich, muß man die Abarbeitungszeiten sehr vorsichtig, d.h. pessimistisch, abschranken.

- Die Anzahl der Ausführungen aller Teile des untersuchten Codestückes ist nach oben hin begrenzt. Diese Grenze, bzw. eine obere Schranke dafür, ist bekannt.

  Diese Forderung findet man bereits in [Kli86, Pus89]. Speziell wollen wir annehmen, daß die Anzahl der Iterationen aller Schleifen nach oben hin abgeschrankt ist und daß die analysierten Programmstücke frei von Rekursionen sind. Für die Qualität der Zeitanalyse gilt in Analogie zum vorigen Punkt: Je genauere Schranken für die maximale Anzahl von Iterationen von Schleifen angegeben werden können, desto knappere Schranken für die maximale Abarbeitungszeit können ermittelt werden.

Die obigen Forderungen bilden gerade die Minimalforderungen, die erfüllt sein müssen, damit eine Schranke für die maximale Abarbeitungszeit berechnet werden kann. Um, wie oben beschrieben, Schranken berechnen zu können, die knapp an der maximalen Abarbeitungszeit liegen, ist es für den allgemeinen Fall notwendig, weitere Beschreibungsmöglichkeiten zur Verfügung zu haben. Eine Vorstellung solcher Erweiterungen wird im Rahmen dieser Arbeit erfolgen.

2.3.2 Anforderungen an eine Methode zur Ermittlung einer $\mathbf{MAXT_C}$

Neben den oben genannten Forderungen stellen wir noch weitere Ansprüche an die MAXT-Analyse. Für eine Methode bzw. ein Softwaretool zur Berechnung der MAXT_C gelten folgende Anforderungen [Pus93]:

- *Berechenbarkeit*: Es muß genügend Information über Hardware, Software und das durch den Eingaberaum bedingte dynamische Verhalten des Codes vorhanden sein, sodaß eine ober Schranke für die maximale Abarbeitungszeit berechnet werden kann. Dieser Punkt wurde bereits oben behandelt.

- *Qualität*: Die Differenz zwischen MAXT_C und MAXT sollte klein sein, d.h. der berechnete Wert soll eine gute Näherung für die tatsächliche Abarbeitungszeit sein. Dies ist notwendig, um eine gute Auslastung der Ressourcen des Zielcomputersystems zu erreichen.

  Die Gründe, die dafür verantwortlich sind, daß die MAXT_C von der MAXT abweicht, wurden bereits in Abschnitt 2.3.1 Punkt für Punkt erläutert. Der dritte Punkt aus dieser Liste ist in dieser Arbeit von zentralem Interesse: Durch die Angabe von Informationen über das dynamische Verhalten eines Programms, das sich in den möglichen Abarbeitungspfaden widerspiegelt, kann die Qualität von MAXT-Berechnungen gesteigert werden (siehe auch [Pus89, Pus91, Par92, Par93]).

- *Detaillierte Information*: Das Resultat der Zeitanalyse soll das Zeitverhalten eines Programmstückes detailliert dokumentieren. Es reicht nicht aus, sich auf einen

einzigen Wert für die MAXT_C zu beschränken. Nur genau aufgeschlüsselte Information über das Zeitverhalten und den Beitrag jedes Programmteils zur MAXT_C erlaubt es dem Programmierer, das zeitliche Verhalten des Codes genau zu beurteilen und hilfreiche Schlüsse für mögliche Optimierungen zu ziehen.

- *Modellierbarkeit*: Diese Anforderung betrifft nicht die Methode, sondern Tools zur Zeitanalyse. Diese Tools sollten die Möglichkeit bieten, die Auswirkungen von Veränderungen des Zeitverhaltens bestimmter Codeteile auf das gesamte Zeitverhalten zu untersuchen. Das Modellieren soll es ermöglichen, mit hypothetischen Abarbeitungszeiten für Programmteile zu experimentieren und so die Auswirkungen möglicher Änderungen zu testen, bevor diese Änderungen tatsächlich programmiert werden.

2.4 Zusammenfassung

Dieses Kapitel diente zur Abgrenzung des Aufgabenbereichs der statischen Zeitanalyse zur Bestimmung der maximalen Abarbeitungszeit von Codestücken. Es wurden die Faktoren untersucht, die die Abarbeitungszeit eines Programmstückes bestimmen. Die Hardware des Zielsystems, sowie die Software und die möglichen Eingangsdaten eines Programmstückes bestimmen die maximale Abarbeitungszeit eines Programmstückes. In der Praxis ist das Wissen über diese bestimmenden Faktoren oft nicht vollständig. Dennoch können mit der vorhandenen Information unter den genannten Voraussetzungen Schranken für maximale Abarbeitungszeiten ermittelt werden. Eine Liste von Anforderungen an Methoden bzw. Tools zur Ermittlung von Schranken für die maximale Abarbeitungszeit von Codestücken schließt das Kapitel ab.

Kapitel 3

Verwandte Arbeiten

3.1 Real-Time Euclid

In [Kli86, Sto87] stellen die Autoren die Programmiersprache Real-Time Euclid vor, die entworfen wurde, um die Erstellung von Echtzeitprogrammen mit garantiertem Antwortzeitverhalten zu ermöglichen. Die Arbeiten beschreiben die Programmiersprache und setzen sich mit den Voraussetzungen für die Erreichung eines vorhersagbaren Zeitverhaltens, sowie der Realisierung entsprechender Konstrukte auseinander. Außerdem wird erklärt, wie das Zeitverhalten von sequentiellen Programmstücken und von ganzen Programmen, die aus parallel arbeitenden, miteinander interagierenden Prozessen bestehen, ermittelt wird.

Eine Entwicklungsumgebung für Real-Time Euclid Programme besteht aus drei Teilen: dem Laufzeitsystem, dem Compiler und dem *Schedulability Analyzer*. Das Laufzeitsystem muß Möglichkeiten zur Verwaltung von parallelen Prozessen, zur Interprozeßkommunikation und Synchronisation, sowie für die Behandlung von Timeouts und das Scheduling enthalten. Der Compiler übersetzt Programme und liefert Informationen über die Codestruktur, Ressourcenanforderungen von Tasks und die Ausführungszeiten von sequentiellen Anweisungsfolgen, sogenannten *Segmenten*. Der Schedulability Analyzer stellt aufgrund der vom Compiler generierten Informationen über die Tasks einer Applikation fest, ob die Applikation unter den gegebenen zeitlichen Anforderungen abgearbeitet werden kann.

Real-Time Euclid Programme bestehen aus Modulen, die Prozeduren und Prozeßdefinitionen enthalten und getrennt übersetzt werden können. Um Aussagen über die maximale Last und das Zeitverhalten einer Applikation machen zu können, werden Prozesse statisch kreiert. Die Sprache enthält einerseits ein Konstrukt für die periodische Aktivierung von Prozessen, andererseits eine Klausel, mit der Prozesse beim Auftreten bestimmter Ereignisse aktiviert werden können. Prozesse laufen dann parallel ab. Zur Synchronisation zwischen Prozessen stehen die Semaphoroperationen *wait* und *signal* zur Verfügung, Interprozeßkommunikation erfolgt über Monitore. Die Sprache bietet

weiters die Möglichkeit, Exception Handler zu definieren. Diese werden entweder durch die Überschreitung von Timeouts oder durch Verletzung von Laufzeitbedingungen aktiviert.

Das Sprachdesign von Real-Time Euclid ist ganz wesentlich von der Zielsetzung, das Worst Case Zeitverhalten von Prozeduren und Prozessen vor der Laufzeit bestimmen zu können, geprägt. Die Berechenbarkeit bzw. Einhaltung des geforderten Zeitverhaltens wird durch folgende Punkte garantiert:

- Abschrankung der Abarbeitungszeit aller Anweisungen. Dies gilt einerseits für Konstrukte, die lokal Prozeßdaten manipulieren, aber auch für Ein-/Ausgabe und Semaphoroperationen.
- Ausschließliche Verwendung von abgeschrankten Schleifen, sogenannten *Bounded Loops*. In Real-Time Euclid sind zwei Arten von Schleifen zugelassen: *for*-Schleifen, deren maximale Anzahl an Iterationen aus der Zuweisung von Werten an die Laufvariable abgeleitet werden kann, und zeitlich abgeschrankte Schleifen.
- Verbot von Rekursionen.
- Alle Datenstrukturen haben bekannte Größe und es gibt keine dynamischen Datenstrukturen.
- Alle zeitlichen Bedingungen (Zeitschranken von Schleifen, Wartezeiten bei Semaphoroperationen, etc.) werden zur Laufzeit durch Timeouts überprüft. Im Falle eines Timeouts wird eine Ausnahmebehandlung (Exception) gestartet.

Analyse des Zeitverhaltens

Durch die *Schedulability Analysis* wird sichergestellt, daß eine Applikation das geforderte Zeitverhalten einhält. Diese Analyse erfolgt in zwei Schritten. Zunächst werden die Abarbeitungszeiten von Basic Blocks bestimmt. Diese Zeiten werden vom Compiler zusammen mit Information über die Ablaufstruktur, Kommunikation und Synchronisation generiert und als *Segment Tree* abgelegt. Im zweiten Schritt analysiert der Schedulability Analyzer die Segment Trees aller zur Applikation gehörigen Tasks. Dabei werden in einer Simulation alle möglichen Szenarien von Wechselwirkungen (inklusive Exceptions) zwischen allen Prozessen der Applikation, generiert und im Hinblick auf ihr Antwortzeitverhalten untersucht. Die Simulation liefert als Ergebnis die Aussage, ob für die Applikation die gegebenen zeitlichen Anforderungen garantiert werden können oder nicht.

Der erste Schritt der Schedulability Analysis entspricht dem Teil der Zeitanalyse, der im Rahmen dieser Arbeit von Interesse ist. Soweit möglich werden dort Segmente zusammengefaßt. Teile des Segment Trees werden durch einzelne Segmente ersetzt,

wobei als Abarbeitungszeit des neuen Segments eine obere Schranke für die Abarbeitungszeit des ersetzten Teilbaumes eingesetzt wird. Die Regeln zur Vereinfachung des Segment Trees erlauben es, Verzweigungen bzw. Schleifen mit sequentiellem Rumpf zu reduzieren. Eine aufwendigere Code-Zeitanalyse wird nicht durchgeführt.

Bewertung

Im Hinblick auf die statische Analyse der Abarbeitungzeiten von Codestücken läßt sich Real-Time Euclid wie folgt bewerten:

+ Das Zeitverhalten von Codesegmenten ist abschrankbar und analysierbar. Schranken für die Abarbeitungszeiten von Segmenten werden einerseits berechnet, andererseits zur Laufzeit auf ihre Einhaltung überprüft.

– Die Berechnung von Codelaufzeiten ist mit der Analyse des Zeitverhaltens der gesamten Applikation verwoben. Eine klare Trennung von Taskzeitanalyse und Analyse von Wechselwirkungen zwischen Tasks ist nicht möglich.

– Die einzige Möglichkeit, die maximale Anzahl von Iterationen von Schleifen anzugeben, besteht durch die Verwendung von *for*-Schleifen. Schleifen mit allgemeinen Abbruchbedingungen (*while*, *repeat*) können nur mit Zeitschranken versehen werden.

– Es gibt keine Konstrukte oder Hilfsmittel, um detaillierte Informationen über mögliche bzw. unmögliche Abarbeitungspfade anzugeben. Dies geht zu Lasten der Qualität der berechneten Zeitschranken.

3.2 Zeitanalyse mit Timetool und TAL

Im Rahmen der Forschungsarbeiten von Al Mok und Gruppe an der University of Texas at Austin wurde eine Menge von Softwarewerkzeugen entwickelt, mit der Worst Case und Best Case Abarbeitungszeiten von Prozeduren berechnet werden können [Mok89, Ame85, Ame88, Che87]. Die Eingabe für die Zeitanalyse bilden C-Programme. Diese Programme werden in einer Folge von Schritten untersucht, die als Ergebnis Informationen über das Zeitverhalten liefert.

- Das Tool *annotate* ist ein Preprozessor, der C-Programme mit sogenannten *Event Markern* versieht. Event Marker sind Annotationen im Programm, die den Beginn und das Ende von Programmabschnitten, die für die Berechnung von Abarbeitungszeiten von Bedeutung sind, kennzeichnen. Dazu zählen z.B. der Beginn und das Ende von Prozeduren, sowie der Beginn und das Ende von Schleifen. Für jede Schleife wird zusätzlich ein Defaultwert für die maximale Anzahl von Iterationen angegeben.

- Mit einem modifizierten *Portablen C-Compiler* [Ame88] wird das annotierte C-Programm in ein Assemblerprogramm übersetzt. Dieses Assemblerprogramm enthält dieselben Event Marker wie das C-Programm. Außerdem kann es vom Benutzer um weitere Annotationen, sogenannte *Split Points*, erweitert werden.

- Das annotierte C-Programm bildet auch die Eingabe für *talgen*. Das Programm *talgen* generiert ein TAL-*Programm* (TAL ... *Timing Analysis Language*, [Che87]), ein Skript zur Berechnung von Zeitschranken aus Programmteilen, die durch Event Marker gegliedert sind.

- Als letztes Werkzeug ermittelt *timetool* die Schranken für die Worst Case und Best Case Abarbeitungszeit des Programms. Zunächst baut es einen Flußgraphen auf, der die Struktur des annotierten C-Programmes widerspiegelt. Die Knoten des Flußgraphen, die den Basic Blocks, bzw. den durch Split Points begrenzten Blöcken entsprechen, werden mit den Ausführungszeiten der zugehörigen Instruktionen, die mit einem Hardware Simulator [Ame85] ermittelt werden, versehen. Anschließend wird das TAL-Programm interpretiert; mit Hilfe der Blockausführungszeiten aus dem Flußgraphen werden die Abarbeitungszeiten berechnet.

Die Ergebnisse der Zeitanalyse sind meist unbefriedigend, wenn die von den Tools generierten Ausgaben unverändert für die Zeitanalyse verwendet werden. Erstens ist die Defaultannahme für die maximale Anzahl von Iterationen für Schleifen in den meisten Fällen nicht adäquat. Weiters können oft nur sehr lose Zeitschranken von Programmen, deren Flußgraph außerdem reduzibel sein muß, ermittelt werden.

Um diesen Schwächen beizukommen, können sowohl die annotierten Assemblerprogramme, als auch TAL-Skripts händisch modifiziert werden. In Assemblerprogrammen können weitere Referenzpunkte für die Zeitanalyse, die oben genannten Split Points, eingefügt werden. TAL-Programme können beliebig ergänzt, geändert bzw. überhaupt händisch aufgesetzt werden, wodurch jegliches Maß an semantischer Information über den Programmablauf in die Berechnung der Abarbeitungszeiten eingebracht werden kann.

Bewertung

Die Vor- und Nachteile der von Mok und Gruppe verwendeten Methode sollen hier kurz zusammengefaßt werden.

+ Durch die Möglichkeit, Annotationen in Assemblerprogramme einzufügen und TAL-Skripts händisch zu modifizieren bzw. zu schreiben, ist eine Berechnung von knappen Schranken für die Abarbeitungszeit möglich.

+ Die Tools erlauben es, Abarbeitungszeiten für beliebige Teile von Prozeduren bzw. Programmteilen zu ermitteln.

– Um bei der Analyse von Abarbeitungszeiten ein gutes Ergebnis zu erzielen, müssen TAL-Skripts *manuell erstellt* werden. Dieser Prozeß ist fehleranfällig. Er verlangt, daß der Programmierer sein Wissen über semantische Zusammenhänge im Code in Form eines Berechnungsskripts ausdrückt.

– Die Analyse erfolgt auf Assemblersprachniveau. Um Wissen über das Zeitverhalten von Teilen eines C-Programmes zu erhalten, muß man die entsprechenden Passagen im Assemblercode suchen und markieren.

– Die angegebenen Iterationsgrenzen von Schleifen und Informationen über Zusammenhänge von Programmteilen bei der Codeabarbeitung werden zur Laufzeit nicht überprüft. Damit können Fehler zur Laufzeit nicht erkannt, das berechnete Zeitverhalten nicht garantiert werden.

3.3 Berechnung von Abarbeitungszeiten mittels "Timing Schema"

Prof. Alan Shaw und ChangYun Park haben Konzepte und Tools für die Ermittlung von Programmabarbeitungszeiten entwickelt, die viele Parallelen mit dem in dieser Arbeit vorgestellten Ansatz aufweisen [Sha89, Par89, Par90, Par92, Par93]. Sie machen Vorhersagen über die minimalen und maximalen Abarbeitungszeiten von Programmen mit Hilfe sogenannter *Timing Schemas*, das sind Formeln zur Berechnung von Abarbeitungszeiten durch Analyse des Programmcodes.

Es wird angenommen, daß jedes Programm/jeder Prozeß auf einem eigenen Prozessor läuft. Daher gibt es keine wechselseitige Beeinträchtigung von Prozessen durch Betriebssystemaktivitäten wie Scheduling, Interruptbehandlung oder die Behandlung von Ein-/Ausgabeoperationen. Interferenzen, die durch Konflikte um Hardwareressourcen entstehen (Zugriffe auf Shared Memory oder Bus), werden in den Ausführungszeiten der jeweiligen Zugriffsstatements berücksichtigt.

Um Abarbeitungszeiten berechnen zu können, gibt es für jedes Programmiersprachkonstrukt eine Regel, *Timing Schema*, die angibt, wie aus den Zeiten für die einzelnen Teile dieses Konstrukts die Abarbeitungszeiten für das gesamte Konstrukt berechnet wird. Diese Regeln werden bei der Zeitanalyse entsprechend der Programmstruktur auf den Programmcode angewendet. Die unterste Ebene von Konstrukten, die nicht weiter unterteilt werden, bilden die *Atomic Blocks*. Atomic Blocks sind einzelne Statements bzw. auch ganze Statement Sequenzen, deren minimale und maximale Abarbeitungszeiten aus dem Wissen über Code, Codeübersetzungsregeln des Compilers und die Hardwareparameter des Zielsystems bestimmt werden [Par90, Par92].

Die Konzepte wurden in einem *Timing Tool* realisiert, das für Programme, die in einer vereinfachten Version der Programmiersprache C geschrieben sind, Schranken für Abarbeitungszeiten ermittelt. Dieses Tool, das selbst auf dem Parser eines GNU

C-Compilers aufbaut, analysiert das Verhalten von Programmen, die mit dem GNU C-Compiler für den MC68010 Prozessor übersetzt werden. Der maschinenabhängige Teil des Tools ermittelt die Abarbeitungszeiten der Atomic Blocks, die mit Hilfe des Parsers identifiziert werden. Der maschinenunabhängige Teil wendet die Timing Schema auf den Output des ersten Teils an und berechnet die Abarbeitungszeiten. Um die Timing Schema vollständig auflösen zu können, müssen dem Timing Tool interaktiv Schranken für die Anzahl der Iterationen von Schleifen angegeben werden.

Die Funktionstüchtigkeit des Tools wurde in Experimenten getestet. Diese Tests ergaben, daß Vorhersagen dann gut funktionieren, wenn die zur Laufzeit tatsächlich auftretenden Programmabarbeitungspfade der Menge der Pfade entspricht, die durch die Konstrukte des Programmes beschrieben werden. In vielen Programmen werden auf Grund der erlaubten Eingabedaten bzw. der Semantik des Programmes nicht alle von der statischen Programmstruktur her möglichen Pfade auch tatsächlich exekutiert. In diesem Fall kommt es dann vor, daß sogenannte *Infeasible Paths*, "unmögliche Pfade", in die Zeitberechnung eingehen und zur Berechnung von Schranken führen, die sehr pessimistisch sind. Diese Beobachtung führte zur Erweiterung des Ansatzes.

Pfadmodell für die Berechnung von Abarbeitungszeiten

Das Pfadmodell geht von folgender Beobachtung aus: Die statische Struktur eines Programmes beschreibt eine Menge von Pfaden. Einige davon können bei der Abarbeitung des Programmes in einer Applikation auftreten (*Feasible Paths*), andere nicht (*Infeasible Paths*). Durch Hinzufügen von Information, die die Menge von Feasible Paths geeignet von den Infeasible Paths abgrenzt und damit die Gesamtmenge der zu berücksichtigenden Pfade reduziert, kann mehr Wissen in die Zeitanalyse gebracht werden. Die Berechnung besserer Schranken für Abarbeitungszeiten wird ermöglicht.

Die Menge der durch die Programmstruktur implizierten Pfade wird in Form von regulären Ausdrücken dargestellt. Dasselbe gilt für die Zusatzinformation über mögliche Pfade, die vom Benutzer bereitgestellt wird. Letztere muß die Menge der möglichen Pfade nicht notwendiger Weise vollständig eingrenzen. Bildet man die Schnittmenge der so beschriebenen Pfade, so erhält man eine Obermenge der tatsächlich möglichen Pfade (die gleichzeitig eine Teilmenge der von der Programmstruktur her ermittelten Pfadmenge ist), die auf ihr Zeitverhalten hin untersucht wird.

Für die endgültige Berechnung der Zeitschranken wird die Menge der möglichen Pfade in Untermengen zerteilt (Pathwise Decomposition). Die Untermengen werden getrennt auf ihr Zeitverhalten untersucht, das Minimum der minimalen Abarbeitungszeiten der Untermengen gibt die minimale Abarbeitungszeit des gesamten Programms, das Maximum der maximalen Zeiten die resultierende maximale Abarbeitungszeit.

Reguläre Ausdrücke würden zwar (theoretisch) eine vollständige Beschreibung der möglichen Abarbeitungspfade durch den Benutzer erlauben, sind aber kein adäquates Ausdrucksmittel für den Benutzer. Aus diesem Grund wurde die Sprache IDL (*Informa-*

tion Description Language) entwickelt, mit der Zusammenhänge zwischen Programmteilen in Bezug auf ihre Abarbeitung beschrieben werden können [Par93]. Die Zusammenhänge werden als Annotationen zum zu analysierenden Programm geschrieben. Sie können z.B. ausdrücken, daß ein Programmteil A immer genau dann durchlaufen wird, wenn auch B durchlaufen wird, oder daß eine Schleife L zehn Mal durchlaufen wird, wenn ein Statement A nicht ausgeführt wird, etc. IDL ist zwar benutzerfreundlicher zu verwenden als dies bei regulären Ausdrücken der Fall wäre, eine vollständige Beschreibung von möglichen Pfaden ist aber mit IDL nicht möglich.

Um Fehler bei der Charakterisierung von Pfaden zu erkennen, schlagen Park und Shaw vor, die Korrektheit der Pfadbeschreibung formal zu verifizieren. Dies könnte z.B. durch eine Verifikation mit Hilfe der Hoare Logik bzw. mit Hilfe von Dijkstras *Weakest Preconditions* erfolgen. Die genannten Verifikationsmöglichkeiten wurden nicht realisiert. Die Autoren meinen [Par93], daß eine automatisierte Verifikation einerseits sehr aufwendig, andererseits auf Grund der Annahmen, die über System und Umgebung der zu analysierenden Software extrahiert werden müßten, in der Praxis nur schwer durchführbar wäre.

Bewertung

Für die abschließende Bewertung lassen sich folgende Beobachtungen zusammenfassen:

+ Der beschriebene Ansatz erlaubt die Berechnung knapper Schranken für Programmabarbeitungszeiten.
+ Programme können unmittelbar auf Programmiersprachenebene mit Annotationen über Abarbeitungspfade versehen werden.
+ Ein Konzept zur Verifikation des beschriebenen Abarbeitungsverhaltens wird vorgestellt, auch wenn dessen Realisierung mit erheblichem Aufwand verbunden wäre.
– Eine vollständige Beschreibung von möglichen bzw. unmöglichen Pfaden ist mit den vorhandenen Mitteln (IDL) nicht möglich.
– Annotationen und Schleifengrenzen werden zur Laufzeit nicht überprüft.
– Durch die Verwendung von regulären Ausdrücken ist die Analyse mit exponentiellem Aufwand verbunden.

3.4 Zeitanalyse für MARS

Die MARS-Gruppe an der Technischen Universität Wien beschäftigt sich seit Jahren mit Themenstellungen, die für die Realisierung von Echtzeitsystemen von zentralem Interesse sind. Auf dem Gebiet der Tasklaufzeitanalyse gehen die Forschungsarbeiten in zwei Richtungen. Auf der einen Seite werden Konzepte und Methoden entwickelt, die die Ermittlung von Schranken für die maximalen Abarbeitungszeiten von Programmen bzw. Programmstücken ermöglichen [Pus89, Sch93]. Auf der anderen Seite werden Prototypen von Hilfsmitteln und Tools entwickelt, mit denen detaillierte Informationen über das Zeitverhalten von Programmcode ermittelt bzw. die Programmierung von zeitkritischen Tasks unterstützt wird [Kop91, Pos92, Kop93].

[Pus89] beschreibt zunächst die Forderungen an Hardware und Software, die erfüllt sein müssen, damit die maximale Abarbeitungszeit eines Programmstückes ermittelt werden kann. Wie in [Kli86] werden Rekursionen verboten. Außerdem wird die Verwendung von Variablen, die Funktionen referenzieren, und von *goto*-Statements untersagt. Für jede Schleife wird verlangt, daß sie entweder durch Angabe der maximalen Iterationsanzahl oder eines Zeitlimits abgeschrankt wird (zeitlich abgeschrankte Schleifen werden in der Folge ebenfalls nicht mehr zugelassen, da die Überprüfung der Zeitbedingung den Replikadeterminismus von redundanten Komponenten im MARS-System verletzen würde).

Die Formeln zur Berechnung der Schranken für die Abarbeitungszeit ermitteln rekursiv für jedes Konstrukt aus den Abarbeitungszeiten der einzelnen Teile des Konstrukts (Bedingung, Aktionsfolgen, Zeiten für das Verzweigen) die gesamte maximale Abarbeitungszeit. Die neu eingeführten Konstukte, *Marker und Scopes*, sowie *Loop Sequences*, ermöglichen es, neben dem Wissen über die statische Programmstruktur auch Informationen über den durch die Programmsemantik implizierten Kontrollfluß in die Taskzeitanalyse mit einzubeziehen.

3.4.1 Untersuchung des Zeitanalyseansatzes

Um die Berechnungsmethode für die maximale Abarbeitungszeit zu evaluieren, wurden die berechneten maximalen Abarbeitungszeiten mit experimentell ermittelten Abarbeitungszeiten verglichen. Dazu wurde zunächst eine Meßumgebung aufgebaut. Nach einer Menge von Tests, die dazu dienten, die (z.T. unerwarteten) Eigenheiten des Zielsystems kennenzulernen [Vrc91], wurden Tasks einerseits analytisch auf ihre maximale Abarbeitungszeit hin untersucht, andererseits mit Testdaten, die auch Daten zur Generierung des Worst Case enthielten, auf dem Zielsystem exekutiert und deren Abarbeitungszeiten gemessen [Pus91].

Die Beispieltasks wurden für die analytische Zeitberechnung mit einem unterschiedlichen Maß an Information über Eigenschaften des Kontrollflusses, die sich aus der Semantik des Codes ableiten lassen, versehen. Für jedes dieser Szenarien wurde die

Abarbeitungszeit berechnet. Anschließend wurden die maximale Laufzeit aus dem Experiment ermittelt und mit den berechneten Schranken verglichen.

Die Ergebnisse der Experimente zeigten einerseits, daß der gewählte Ansatz prinzipiell gut für die Vorhersage maximaler Abarbeitungszeiten geeignet ist. Gleichzeitig konnte festgestellt werden, daß es die Möglichkeit geben muß, Wissen über das dynamische Verhalten eines Programmstückes zu beschreiben, will man knappe Schranken für maximale Abarbeitungszeiten berechnen. In den durchgeführten Experimenten konnte die berechnete Schranke durch die Berücksichtigung solcher Information in einer händischen Analyse für ein Programm bis auf 25% des ursprünglichen, ohne Zusatzinformation ermittelten Wertes gesenkt werden.

3.4.2 Integration der Zeitanalyse in eine Taskentwicklungsumgebung

Eine zweite Stoßrichtung bildet die Schaffung einer Taskentwicklungsumgebung für das MARS-System. Diese Entwicklungsumgebung soll das Programmiermodell von MARS [Kop93] und damit die Programmierung von Tasks, die strikte, vorgegebene Zeitschranken einhalten, unterstützen.

Die Programmiersprache

In einem ersten Schritt wurde für die Programmierung von MARS-Tasks eine eigene Programmiersprache, Modula/R [Vrc92], konzipiert. Modula/R ist eine Echtzeitprogrammiersprache, die das Taskmodell von MARS unterstützt und eine geeignete Schnittstelle zum MARS-Betriebssystem [Rei93] hat. Die Sprache zeichnet sich einerseits durch ihr strenges Typenkonzept und die statische Speicherallokation, andererseits durch die zeitliche Abschrankbarkeit aller Konstrukte, sowie die Möglichkeit, Informationen über unmögliche Pfade anzugeben (Scopes, Marker, Loop Sequences), aus. Damit kann für jeden Task sowohl der maximale Speicherplatzbedarf, als auch eine Schranke für die maximale Abarbeitungszeit mittels statischer Analyse ermittelt werden.

Die Programmierumgebung

Für die Entwicklung der Programmierumgebung [Pos92] spielte die folgende Beobachtung eine zentrale Rolle: Bei der Programmierung von Echtzeittasks muß der Programmierer Software so schreiben, daß diese die vom Designer vorgegebenen Zeitschranken garantiert einhält. Um dieses Ziel zu erreichen, ist es unumgänglich, daß der Programmierer jederzeit während des gesamten Softwareentwicklungvorganges detaillierte Informationen über das Zeitverhalten des implementierten Codes bekommen kann. Nur so kann er Programmteile erkennen, die zuviel Zeit konsumieren und daher optimiert werden müssen. Die MARS-Taskprogrammierumgebung berücksichtigt diese Anforderung.

Sie gibt dem Programmierer nicht nur einen detaillierten Einblick in das Zeitverhalten des programmierten Codes, sondern erlaubt es ihm auch, die Auswirkungen von Änderungen im Code auf das Zeitverhalten von Programmteilen und das Zeitverhalten des gesamten Programms zu evaluieren.

Die Programmierumgebung besteht aus drei zusammenarbeitenden Funktionseinheiten, dem *Editor*, der sich aus einem Texteditor und dem sogenannten *Time Editor* zusammensetzt, dem *Modula/R Compiler* und dem *Zeitanalysetool*. Diese drei Softwarewerkzeuge kommunizieren miteinander über den *Timing Tree*, eine Datenstruktur, in der die Information über die Struktur und das Zeitverhalten eines Programmstückes dargestellt wird.

Der Texteditor unterscheidet sich von anderen Texteditoren dadurch, daß er sehr eng an den Time Editor gekoppelt ist. Der Time Editor zeigt für Programme und ihre einzelnen Konstrukte Schranken für die maximalen Abarbeitungszeiten an. Diese Zeiten werden neben dem entsprechenden Programmtextstück angezeigt.

Der Compiler hat zwei Aufgaben. Er übersetzt die Quellprogramme in Maschinencode und generiert Output für das Zeitanalysetool [Vrc93]. Letzterer wird durch einen Timing Tree repräsentiert, der die Codestruktur widerspiegelt und die Abarbeitungszeiten aller sequentiellen Programmstücke in den Blättern enthält.

Das Zeitanalysetool (*MAXT-Analyzer*) [Pus93] untersucht das zeitliche Verhalten von Codestücken anhand ihrer Timing Trees. Es berechnet Schranken für die Abarbeitungszeiten von einzelnen Konstrukten und ganzen zu analysierenden Codestücken. Dabei können die Eingangsdaten entweder in Form eines vom Compiler generierten Timing Trees oder eines Timing Trees, der hypothetische Abarbeitungszeiten enthält (durch den Time Editor erzeugt), übergeben werden [Pos92].

3.4.3 Bezug zur vorliegenden Arbeit

Die Berechnung von Schranken für die maximale Abarbeitungszeit von Programmen bzw. Programmstücken bildet nach wie vor einen Schwerpunkt im Rahmen des MARS-Projekts. Ziel der gegenwärtigen Bestrebungen ist es, mit Hilfe von Informationen über Datenabhängigkeiten der Programmausführung, die sich aus der Semantik des Codes und dem Wissen über die möglichen Eingabedaten ableiten lassen, möglichst knappe Schranken für die maximale Abarbeitungzeit zu berechnen.

Die im Abschnitt 3.4.1 erörterten Erkenntnisse bildeten die Motivation für die vorliegende Arbeit. Es wurde dort gezeigt, daß umfassenderes Wissen über das dynamische Verhalten eines Programmstückes für eine Verbesserung des Ergebnisses der MAXT-Analyse genutzt werden kann. Dies führte zum hier behandelten Ansatz, bei dem die maximale Abarbeitungszeit durch Lösung eines entsprechenden linearen Optimierungsproblems ermittelt wird. Eine erste Dokumentation dieses Verfahrens, bei dem ein Weg zur MAXT-Berechnung für ein in Form eines Timing-Trees beschriebenen Codestückes

vorgestellt wird, ist in [Sch93] zu finden.

3.5 Zusammenfassung

In diesem Kapitel wurden die wesentlichen Arbeiten, die sich mit dem Thema statische Codelaufzeitanalyse auseinandersetzen, vorgestellt. Den Arbeiten ist gemeinsam, daß sie das Ziel haben, Schranken für die Abarbeitungszeit von Programmen oder Programmteilen zu berechnen. Auf der anderen Seite unterliegen die präsentierten Verfahren Einschränkunen, die in dieser Arbeit überwunden werden sollen: Die Entwickler von Real-Time Euclid zielen nicht darauf hin, Zeitschranken einer hohen Qualität zu ermitteln. Bei der Verwendung von TAL und *timetool* muß man Scripts für die Berechnung der Abarbeitungszeit händisch modifizieren, wenn man knappe Zeitschranken berechnen will. Der Ansatz von Park und Shaw erlaubt zwar die Berechnung von Zeitschranken, die nahe an der tatsächlichen Abarbeitungszeit liegen, erlaubt aber keine vollständige Beschreibung aller unmöglichen Abarbeitungspfade. Außerdem arbeitet das Analyseverfahren mit hohem Aufwand.

Kapitel 4

Ermittlung von Schranken für die Abarbeitungszeit

In diesem Kapitel werden die theoretischen Grundlagen für den in dieser Arbeit präsentierten Zeitanalyseansatz vorgestellt. Es wird eine Repräsentation von Codestücken in Form von gerichteten Graphen, die die Codestruktur sowie die Ausführungszeiten von sequentiellen Codeeinheiten beschreiben, eingeführt. Diese Repräsentation erlaubt es, den Kontrollfluß und das Zeitverhalten von Programmabarbeitungen zu beschreiben und bildet die Grundlage für die Berechnung der maximalen Abarbeitungszeit. Die vorgestellte Methode führt die Ermittlung der maximalen Abarbeitungszeit eines Codestückes auf die Berechnung einer maximalen Zirkulation in einem gewichteten Flußnetzwerk (Graph mit Kantenbewertungen), zurück. In diesem Flußnetzwerk dürfen die Flüsse durch Kanten nicht nur durch Kapazitätsgrenzen für jede einzelne Kante, sondern auch durch allgemeinere Restriktionen, die den Fluß mehrerer Kanten betreffen und in Relation setzen, abgeschrankt werden.

4.1 Coderepräsentation für die Zeitanalyse

Gesucht ist eine Darstellung, die genau die Information über das Programm enthält, die für die Ermittlung der maximalen Abarbeitungszeit von Codestücken von Bedeutung ist. Um mehr Information über das Worst Case Zeitverhalten von Programmen bzw. Programmstücken berechnen zu können als nur die maximale Abarbeitungszeit (siehe Kapitel 2), kann man das Problem der Codezeitanalyse wie folgt formulieren:

> Ermittle den Pfad mit der längsten Abarbeitungszeit durch das gegebene Programmstück und berechne für diesen Pfad die maximale Abarbeitungszeit.

Um diese Aufgabe zu lösen, muß eine Darstellung gewählt werden, mit der man einerseits feststellen kann, welche Abarbeitungspfade bei der Ausführung des Programmstückes möglich sind und andererseits, welche Abarbeitungszeit den einzelnen Pfaden zugeordnet werden kann. Eine Darstellungsform, die im Compilerbau zur Beschreibung der Kontroll- und Ablaufstruktur von Programmen verwendet wird, ist die Darstellung in Form eines Graphen, des Programmflußgraphen [**?**]. Der Flußgraph stellt sequentielle Folgen von Programmstatements durch Kanten und Verzweigungen im Kontrollfluß durch entsprechende Verzweigungen an den Knoten des Flußgraphen dar. Eine Graphenrepräsentation, die von der Struktur dem Flußgraphen eines Codestückes sehr ähnlich ist, eignet sich für die Berechnung von Abarbeitungszeiten. Allerdings müssen für die Zeitanalyse die im Graphen repräsentierten Informationen an die Problemstellung angepaßt werden.

Man kann annehmen, daß der für die Zeitanalyse verwendete Graph von einem Compiler oder mit Hilfe eines Codeanalysewerkzeuges aus einem Programmstück generiert wird. Die Anordnung seiner Kanten repräsentiert die Kontrollstruktur des Quellprogrammes, Kantenbewertungen beschreiben die Abarbeitungszeiten der einzelnen Programmteile und weitere Informationen beschreiben die Anzahl der Abarbeitungen von Kanten. Außerdem werden Zusatzinformationen über das dynamische Verhalten des Programms, die im Quellprogramm durch Marker und andere Konstrukte [Pus89] angegeben werden, in Restriktionen übersetzt, die die Anzahl der Exekutionen von Kanten, die später als ganzzahlige Flüsse repräsentiert werden, entsprechend beschränken.

Im Rest dieses Kapitels werden Aussehen und Eigenschaften der für die Zeitanalyse verwendeten Graphen näher beschrieben. Wie solche Graphen aus Programmen erzeugt und die maximale Abarbeitungszeit mittels ganzzahliger linearer Optimierung berechnet werden kann, wird in den nachfolgenden Kapiteln ausgeführt.

4.1.1 Der Zeitanalysegraph

Ein Codestück, dessen maximale Abarbeitungszeit berechnet werden soll, wird durch einen *zusammenhängenden gerichteten Graphen* $G = (V, E)$ repräsentiert. In G bezeichnet $V = \{v_i : 0 \leq i < |V|\}$ die Menge der Knoten und $E = \{e_i : 0 \leq i < |E|\}$ die Menge der gerichteten Kanten. Die Kanten werden auch als geordnete Paare in der Form $e_i = (v_j, v_k)$ geschrieben, wobei v_j den Startknoten und v_k den Endknoten der Kante e_i angibt. Weiters soll G die folgenden Eigenschaften haben:

1. G besitzt genau eine Quelle ($v_0 = s$), d.h. einen Knoten von dem nur Kanten weg führen, $\{e_i : e_i = (v_j, s)\} = \emptyset$.
2. G besitzt genau eine Senke ($v_{|V|-1} = t$), d.h. einen Knoten zu dem nur Kanten hin führen, $\{e_i : e_i = (t, v_j)\} = \emptyset$.
3. Jede Kante $e_i \in E$ ist Teil mindestens eines gerichteten Kantenzuges von der Quelle s zur Senke t.

4. Es existiert eine Abbildung t, die jeder Kante e_i eine nicht negative ganze Zahl $t_i = t(e_i)$ zuordnet.

Da die eben charakterisierten Graphen den Ausgangspunkt für die weitere Beschreibung und Berechnung des Zeitverhaltens von Programmstücken bilden, wollen wir zunächst diesen Graphen einen Namen geben.

Def.: Ein *T-Graph* (*Timing Analysis Graph* oder *Zeitanalysegraph*) ist ein zusammenhängender, gerichteter Graph, der die oben angeführten Eigenschaften 1 bis 4 hat.

Ein T-Graph, bzw. der Zusammenhang zwischen einem T-Graphen und dem Programmstück, das er repräsentiert, wird wie folgt interpretiert:

- Jede Kante des Graphen steht für eine Folge von Statements bzw. Instruktionen des Codes, der zu untersuchen ist. Das können entweder Sequenzen von Maschineninstruktionen sein, wenn Assemblercode analysiert werden soll, können aber auch Folgen von Statements einer höheren Programmiersprache oder einer Aktionsfolge sein, die im Pseudocode angegeben ist. Die hier angegebene Repräsentation kann aus jeder dieser Darstellungen abgeleitet werden, wenn nur eine maximale Abarbeitungszeit für alle sequentiellen Programmteile angegeben werden kann (siehe unten). Um die Vollständigkeit der Zeitanalyse garantieren zu können, muß weiters sicher gestellt werden, daß auch jeder Teil des Originalprogrammes auf eine entsprechende Kante abgebildet wird.

- Die Knoten von G stehen für Punkte, an denen sich der Kontrollfluß des Codes verzweigt, vereinigt, bzw. fortsetzt. Sie haben selbst keine unmittelbare Entsprechung im Code. Es muß allerdings gelten, daß die Kanten aller Codesequenzen, die unmittelbar hintereinander ausgeführt werden können auch im Graphen an einem Knoten aufeinanderfolgen.

- Die Eigenschaften 1 und 2 des zusammenhängenden Graphen sollen sicherstellen, daß das vorliegende Codestück eine sinnvoll zu analysierende Einheit mit genau einem Startpunkt und einem Endpunkt, z.B. Prozeduranfang und -ende, bildet.

- Eigenschaft 3 garantiert zudem, daß alle Sequenzen Teile gültiger Programmpfade vom Codebeginn zum Ende sind, d.h. Programmteile, die von ihrer Struktur her nie abgearbeitet oder verlassen werden können, werden ausgeschlossen (siehe Abbildung 4.1).

- Die Abbildung t, siehe Punkt 4, ordnet jeder Kante die maximale Abarbeitungszeit des entsprechenden sequentiellen Codestückes zu.

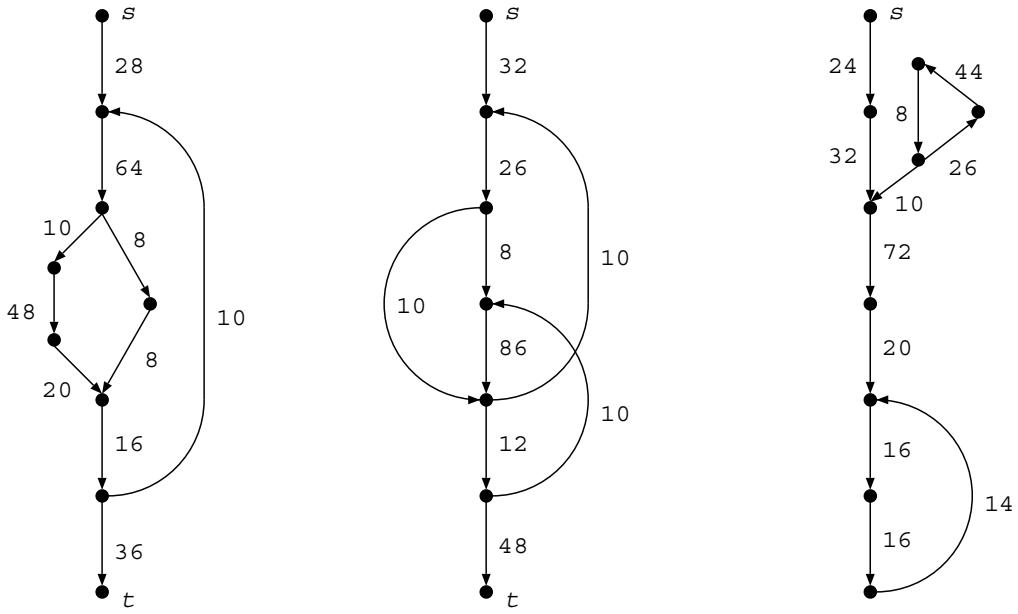


Abbildung 4.1: Zwei gültige T-Graphen (links und in der Mitte) und ein ungültiger T-Graph (rechts), der Eigenschaft 3 verletzt

T-Graphen müssen keineswegs reduzibel [**?**] sein, sondern können eine viel allgemeinere Struktur aufweisen (siehe etwa Abbildung 4.1 mittlerer Graph). Es ist daher möglich, allgemeinere Programmstrukturen als solche, die sich durch höhere Programmiersprachen beschreiben lassen, darzustellen und zu untersuchen.

Weiters sei darauf hingewiesen, daß in der Interpretation der Semantik des T-Graphen nicht verlangt wird, daß die Kanten Statement- oder Instruktionssequenzen maximaler Länge repräsentieren. Es ist also durchaus möglich, eine durchgehende Sequenz als lineare Folge von Kanten darzustellen, die jede für sich Teile dieser Sequenz repräsentieren. Im Extremfall kann jedes einzelne Statement bzw. jede einzelne Instruktion als eigene Kante dargestellt werden.

Weiters können Teile des Codes durchaus durch mehr als eine Kante repräsentiert werden, solange der beschriebene Kontrollfluß gültig ist. Dies ist z.B. bei bedingten Sprunginstruktionen sinnvoll, wenn die Instruktion in Abhängigkeit davon, ob der Sprung durchgeführt wird oder nicht, unterschiedliche Ausführungszeiten haben kann. Im T-Graphen kann diese Instruktion durch mehrere Kanten mit verschiedenen Zeiten und den entsprechenden Nachfolgekanten beschrieben werden.

Abbildung 4.2 zeigt einen Ausschnitt aus einem Assemblerprogramm für den Motorola MC68000 Prozessor und einen entsprechenden T-Graphen. Im Programmstück ist für jede Instruktion die maximale Anzahl von CPU-Zyklen, die für die Abarbeitung der Instruktion benötigt werden, angegeben. Die bedingten Sprunginstruktionen benötigen 10 Zyklen für das Verzweigen bzw. 8 Zyklen, wenn die Abarbeitung bei der

Folgeinstruktion fortsetzen soll.

```
        moveq #19, d2          |  4
        cmpw d2, d3            |  4
        bgts L16               | 10; 8
    L14:
        movew d2, d0           |  4
        extl d0                |  4
        movel d0, d1           |  4
        asll #2, d1            | 12
        lea a1@(-4,d1:l), a0   | 12
        movel a0@, d4          | 12
        cmpl a1@(d1:l), d4     | 20
        bles L12               | 10; 8

        movel a0@, d0          | 12
        movel a1@(d1:l), a0@   | 26
        movel d0, a1@(d1:l)    | 18
    L12:
        subqw #1, d2           |  4
        cmpw d2, d3            |  4
        bles L14               | 10; 8
    L16:
        addqw #1, d3           |  4
        cmpw #19, d3           |  8
```

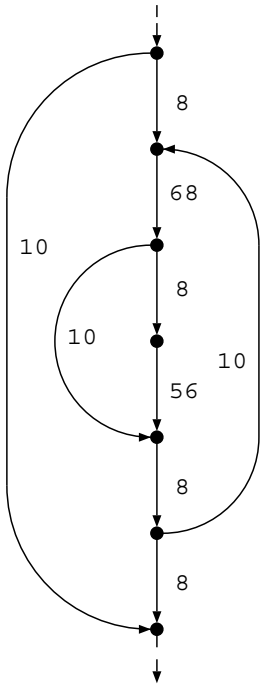


Abbildung 4.2: Beispiel für ein Assemblerprogrammstück und einen entsprechenden T-Graphen

4.1.2 Abarbeitungspfade und Abarbeitungszeiten

Da nun Programmstücke in Form von T-Graphen dargestellt werden können, sollen die Begriffe *Abarbeitungspfad* und *Abarbeitungszeit* definiert werden. Unter einer Abarbeitung eines Programmstückes versteht man die Ausführung einer Folge von Aktionen, die durch dieses Programmstück beschrieben werden, wobei diese Aktionsfolge beim Startpunkt beginnt, dem Kontrollfluß gehorcht und beim Endpunkt des Codes endet. Einer Abarbeitung kann nun ein Kantenzug im T-Graphen zugeordnet werden, der dem Kontrollfluß entspricht. Dieser Kantenzug beginnt in s und führt nach t.

Def.: Ein Kantenzug $(e_{j_1}, e_{j_2}, \ldots, e_{j_m})$ in einem T-Graphen G, der mit Startknoten s beginnt und mit dem Endknoten t endet heißt *Abarbeitungspfad* in G.

Wir werden Abarbeitungspfade im Rest der Arbeit auch kurz *Pfade* nennen. Jeder Abarbeitungspfad setzt sich aus einer endlichen Menge von Kanten zusammen und jedem Pfad entspricht genau eine Abarbeitungszeit. Diese ist gleich der Summe der Abarbeitungszeiten der Kanten, die auf dem Pfad liegen.

Def.: Die *Abarbeitungszeit* eines Pfades $P = (e_{q_1}, e_{q_2}, \ldots, e_{q_m})$ ist die Summe der Abarbeitungszeiten aller Kanten des Pfades.

$$xt(P) = \sum_{i=1}^{m} t(e_{q_i}) \tag{4.1}$$

Wegen der Kommutativität der Addition ist die Reihenfolge, in der die Kanten im Kantenzug auftreten, für die Berechnung der Abarbeitungszeit ohne Bedeutung. Bestimmt man daher für jede Kante e_i aus dem T-Graphen, wie oft sie in P enthalten ist, so erhält man ein n-Tupel $F(P) = (f_{p0}, f_{p1}, \ldots, f_{p|E|-1})$, für das gilt, daß f_i die Anzahl der Vorkommnisse von e_i in P ist. Eine zur Formel 4.1 äquivalente Formulierung ist daher:

$$xt(P) = \sum_{i=0}^{|E|-1} f_{p_i} t(e_i) = \sum_{i=0}^{|E|-1} f_{p_i} t_i \tag{4.2}$$

Wir haben in Kapitel 2 festgestellt, daß ein Programm bzw. Programmstück einer Applikation in Abhängigkeit vom Kontext, in dem es abläuft, nur eine endliche Anzahl von Aktionsfolgen zuläßt. Diesen Aktionsfolgen entsprechen endliche Mengen von Pfaden in T-Graphen. Betrachtet man die Abarbeitungszeiten aller möglichen Pfade, so kann man die maximale Abarbeitungszeit ermitteln.

Def.: Die *Maximale Abarbeitungszeit (MAXT)* einer Menge von Pfaden π in einem T-Graphen G ist:

$$maxt(\pi) = \max_{P \in \pi}(xt(P)) = \max_{P \in \pi} \sum_{i=0}^{|E|-1} f_{p_i} t_i \tag{4.3}$$

In der Praxis ist es mühsam, bzw. in vielen Fällen mit unvertretbarem Aufwand behaftet, alle möglichen Abarbeitungspfade eines Programmstückes einzeln aufzuzählen, um den exakten Wert der maximalen Abarbeitungszeit zu berechnen. Wir wollen daher hier eine Repräsentationsform verwenden, die gemeinsame Eigenschaften von Pfaden beschreibt, anstatt einzelne Pfade aufzuzählen. Die MAXT-Berechnung wird mit dieser Darstellung einfacher.

4.2 Die MAXT und Zirkulationen

Wie oben gezeigt wurde, kann man mit T-Graphen die statische Struktur von Programmen bzw. Programmabschnitten beschreiben. Damit kann man aus dem Graphen zwar ablesen, in welcher Reihenfolge die den Kanten entsprechenden Aktionsfolgen von der Programmstruktur her abgearbeitet werden dürfen, es ist aber nicht gesagt,

daß jeder Kantenzug im Graphen auch tatsächlich Teil irgendeines im der Anwendung auftretenden Abarbeitungspfades bzw. eines Abarbeitungspfades mit maximaler Abarbeitungszeit ist. Im Rahmen dieser Arbeit ist es zwar nicht von Interesse, die Menge der möglichen Pfade ganz exakt einzugrenzen, aber es soll der MAXT-Analyse doch soviel Zusatzwissen über die Worst Case Pfade verfügbar sein, daß man die maximale Abarbeitungszeit des Codestückes berechnen kann. Wir wollen daher dem Graphen zusätzliche Information hinzufügen: Restriktionen, die angeben, wie oft Kanten im Graphen maximal durchlaufen werden.

Im Rest dieses Kapitels wird gezeigt, wie die MAXT-Berechnung für einen T-Graphen mit Restriktionen auf das Problem der Berechnung einer maximalen Zirkulation in einem Graphen zurückgeführt werden kann. Wir wollen uns dem Problem allerdings in zwei Schritten nähern: Zunächst sollen nur einfache Restriktionen auf T-Graphen verwendet werden. Diese legen für jede Kante eine Obergrenze an Abarbeitungen fest. Im Anschluß wird gezeigt, wie mit verallgemeinerten Restriktionen für die Anzahl von Kantenabarbeitungen beliebige zulässige Pfadmengen eines Programmstücks charakterisiert werden können.

4.2.1 T-Graphen und Zirkulationen

Die bisherige Problemstellung für die MAXT-Analyse soll nochmals in der graphentheoretischen Terminologie zusammengefaßt werden: Gegeben ist ein gerichteter, zusammenhängender Graph mit bewerteten Kanten ("Kosten" der Kanten sind ihre Abarbeitungszeiten), der eine Quelle und eine Senke besitzt. Außerdem gibt es eine Menge von Restriktionen auf Kanten. Gesucht ist ein Kantenzug mit maximalen Kosten von der Quelle zur Senke.

Dieses Problem läßt sich in ein klassisches graphentheoretisches Problem, die Ermittlung einer Zirkulation maximaler Kosten in einem gerichteten Graphen, abbilden. Zirkulationen sind eine Verallgemeinerung von Flüssen auf Graphen. Die folgenden Definitionen sind im Zusammenhang mit Zirkulationen von Bedeutung [Jun90].

Def.: Eine Abbildung $f : E \to \mathbf{R}$ heißt *Zirkulation* auf einem Graphen G, wenn sie dem Erhaltungssatz (Flußerhaltungssatz) genügt (e^+ steht dabei für den Anfang, e^- für das Ende einer gerichteten Kante):

$$\forall v \in V : \sum_{e:e^+=v} f(e) = \sum_{e:e^-=v} f(e). \tag{4.4}$$

Def.: Es seien zwei Abbildungen, *Kapazitätsfunktionen*, $b : E \to \mathbf{R}$ und $c : E \to \mathbf{R}$ für alle Kanten definiert, mit $b(e) \leq c(e)$. Eine Zirkulation f heißt dann *zulässig*, wenn für alle Kanten e gilt:

$$b(e) \leq f(e) \leq c(e). \tag{4.5}$$

Def.: Gegeben sei eine Abbildung, die *Kostenfunktion*, $\gamma : E \to \mathbf{R}$. Die *Kosten* $\gamma(f)$ einer Zirkulation f sind dann definiert durch:

$$\gamma(f) := \sum_e \gamma(e) f(e). \tag{4.6}$$

Zur Bestimmung der maximalen Abarbeitungszeit wird ein T-Graph mit Restriktionen wie folgt auf das Zirkulationsproblem transformiert:

1. Der T-Graph G wird in einen *Erweiterten T-Graphen* $G' = (V', E')$ umgewandelt, indem eine *Rückkehrkante* $e_{|E|} = (t, s)$ mit Abarbeitungszeit $t(e_{|E|}) = 0$ hinzugefügt wird ($V' = V$, $E' = E \cup \{e_{|E|}\}$).

2. Die Werte der Kapazitätsfunktionen b und c werden für alle Kanten aus E' definiert.

$$b(e) := \begin{cases} 1 & e = e_{|E|} \text{ (Rückkehrkante)} \\ 0 & \text{sonst} \end{cases} \tag{4.7}$$

$$c(e) := \begin{cases} 1 & e = e_{|E|} \\ r(e) & \text{sonst} \end{cases} \tag{4.8}$$

   Dabei steht $r(e)$ für die oben genannte Obergrenze für die Anzahl von Abarbeitungen der Kante e, die das Maximum an Abarbeitungen dieser Kante durch die erlaubten Pfade beschreibt.

3. Die Kosten der Kanten im erweiterten T-Graphen entsprechen den Abarbeitungszeiten der Kanten im T-Graphen, $t(e_{|E|})$ wurde mit 0 definiert.

$$\forall e \in E' : \gamma(e) := t(e). \tag{4.9}$$

Folgerungen:

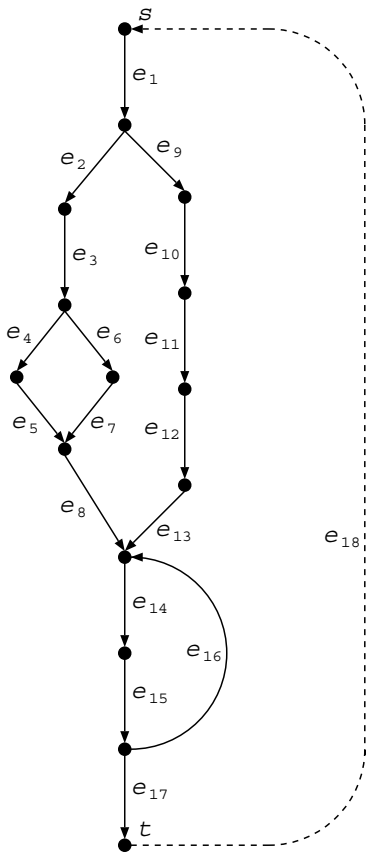
1. Einer Abarbeitung im T-Graphen entspricht nun ein geschlossener Kantenzug $((s, v_{i_1}), (v_{i_1}, v_{i_2}), \ldots, (t, s))$. Dieser Kantenzug induziert einen ganzzahligen Fluß auf dem Graphen, wobei jedes Abarbeiten einer Kante den Beitrag 1 zum Fluß dieser Kante liefert. Da es sich um einen Kantenzug handelt, wird für jede Abarbeitung einer Kante (v_i, v) auch eine Nachfolgekante (v, v_j) abgearbeitet. Somit gilt tatsächlich das Flußerhaltungsgesetz.

2. Der erweiterte T-Graph soll zunächst für den Mindestfluß jeder Kante außer Vorgaben, die durch das Flußerhaltungsprinzip impliziert werden, nur verlangen, daß dieser nicht negativ wird. Eine Ausnahme bildet die Rückkehrkante: $b((t, s)) = c((t, s)) = 1$ stellt sicher, daß der von s wegführende bzw. zu t hinführende Fluß gleich 1 ist (Flußerhaltung). Dies entspricht genau einem Pfad von s nach t in G (siehe auch den vorigen Punkt).

Für die maximale Kapazität der Kanten nimmt man sinnvollerweise an, daß $c(e) \geq 1$ für alle e. Aus den Eigenschaften des ursprünglichen T-Graphen folgt dann sofort, daß es auf jeden Fall mindestens eine zulässige Zirkulation und somit auch eine Zirkulation maximaler Kosten gibt.

3. Den Kosten einer Kante wird deren Abarbeitungszeit im T-Graphen zugeordnet. Für einen Abarbeitungspfad P in G, dem die Zirkulation f in G' entspricht, kann daher wegen den oben beschriebenen Zusammenhängen die Abarbeitungszeit durch die Kosten der Zirkulation, $\gamma(f)$, berechnet werden.

Folgerung 3 impliziert, daß die maximale Abarbeitungszeit eines Abarbeitungspfades in G den maximalen Kosten einer gültigen Zirkulation in G' entspricht. Die Berechnung der maximalen Abarbeitungszeit entspricht dann der Ermittlung einer Zirkulation maximaler Kosten. Dies soll anhand eines Beispiels veranschaulicht werden (siehe Abbildung 4.3).



e	$c(e)$	$\gamma(e)$
e_1	1	36
e_2	1	8
e_3	1	86
e_4	1	8
e_5	1	112
e_6	1	10
e_7	1	30
e_8	1	46
e_9	1	10
e_{10}	1	12
e_{11}	1	26
e_{12}	1	32
e_{13}	1	24
e_{14}	10	44
e_{15}	10	20
e_{16}	9	10
e_{17}	1	56
e_{18}	1	0

Abbildung 4.3: Beispiel für einen Erweiterten T-Graphen.

Abbildung 4.3 zeigt einen Erweiterten T-Graphen. Für jede Kante sind sowohl die maximale Kapazität $c(e)$ als auch die Kosten $\gamma(e)$ angegeben. Für die minimalen

Kapazitäten gilt, $b(e_{18}) = 1$ (Rückkehrkante) und für alle anderen Kanten $b(e_i) = 0$ (siehe oben). Die maximale Abarbeitungszeit für den Graphen G' ist

$$maxt(G') = \max \sum_{e \in E'} f(e)\gamma(e)$$

$$\text{wobei } \forall e \in E' : b(e) \leq f(e) \leq c(e) \text{ und } \forall v \in V' : \sum_{e:e^+=v} f(e) = \sum_{e:e^-=v} f(e).$$

Auch wenn noch kein Algorithmus zur Bestimmung einer Zirkulation maximaler Kosten auf einem Erweiterten T-Graphen vorgestellt wurde, so ist diese hier doch leicht zu finden: Für e_1 und e_{17} gilt auf jeden Fall $f(e) = 1$. Um eine zulässige Zirkulation zu erhalten kann nur entweder $f(e_2)$ oder $f(e_9)$ gleich 1 sein. Der Fluß durch die andere Kante muß auf jeden Fall gleich Null sein. Da beide zulässigen Kantenfolgen im linken Teilgraphen $(e_2, \ldots, e_8)$ höhere Kosten als die Kantenfolge des rechten Teilgraphen $(e_9, e_{10}, e_{11}, e_{12}, e_{13})$ haben, ist der Fluß im rechten Teilgraphen gleich Null. Auf der linken Seite hat der Kantenzug $(e_2, e_3, e_4, e_5, e_8)$ die höheren Kosten. Der maximale Fluß aller Kanten des folgenden Kreises gleicht deren maximalen Kapazitäten. Als maximaler Fluß bzw. maximale Abarbeitungszeit ergibt sich somit:

$$\gamma(e_1)+\gamma(e_2)+\gamma(e_3)+\gamma(e_4)+\gamma(e_5)+\gamma(e_8)+10\gamma(e_{14})+10\gamma(e_{15})+9\gamma(e_{16})+\gamma(e_{17})+\gamma(e_{18}) =$$

$$36 + 8 + 86 + 8 + 112 + 46 + 440 + 200 + 90 + 56 = \underline{1082}.$$

4.2.2 Flußrestriktionen auf Zirkulationen

Bevor das obige Ergebnis weiter diskutiert wird, soll hier der Begriff des von einer Zirkulation induzierten Zirkulationssubgraphen eingeführt werden.

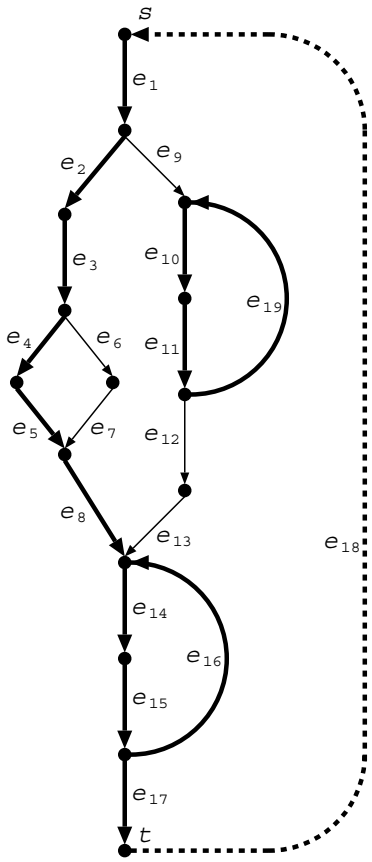
Def.: Ein Subgraph $G_f = (E_f, V_f)$ eines erweiterten T-Graphen G' mit Zirkulation f, für den gilt

$$\begin{aligned} E_f &= \{e : e \in E' \text{ und } f(e) > 0\} \\ V_f &= \{v : v \in V' \text{ und } \exists e \in E_f : e^+ = v\} \end{aligned} \tag{4.10}$$

wird von f induzierter *Zirkulationssubgraph* von G' genannt.

Im vorigen Beispiel konnte die maximale Abarbeitungszeit des Graphen durch die Bestimmung der Zirkulation maximaler Kosten ermittelt werden. Dies funktioniert aber nur in Sonderfällen, speziell immer dann, wenn der ursprüngliche T-Graph kreisfrei ist. Ist der T-Graph G nicht kreisfrei, so ist nicht garantiert, daß die Zirkulation maximaler Kosten in G' einen zusammenhängenden Zirkulationssubgraphen induziert, d.h. die Zirkulation liefert keinen zulässigen Programmpfad, siehe Abb. 4.4.

Der in Abbildung 4.4 gezeigte Graph entsteht aus dem Graphen des vorhergehenden Beispiels durch Hinzufügen der Kante e_{19} und Modifikation der Kapazitätsgrenzen.



e	$c(e)$	$\gamma(e)$	$f(e)$
e_1	1	36	1
e_2	1	8	1
e_3	1	86	1
e_4	1	8	1
e_5	1	112	1
e_6	1	10	0
e_7	1	30	0
e_8	1	46	1
e_9	1	10	0
e_{10}	8	12	7
e_{11}	8	26	7
e_{12}	1	32	0
e_{13}	1	24	0
e_{14}	10	44	10
e_{15}	10	20	10
e_{16}	9	10	9
e_{17}	1	56	1
e_{18}	1	0	1
e_{19}	7	10	7

Abbildung 4.4: Eine Zirkulation maximaler Kosten, die keinem gültigen Abarbeitungspfad entspricht. Die fetten Kanten kennzeichnen den zugehörigen Zirkulationssubgraphen.

Betrachtet man den Zirkulationssubgraphen der Zirkulation maximaler Kosten, so erkennt man, daß es in diesem zwar einen Kantenzug $(s, \ldots, t)$ gibt, daß aber nicht alle Kanten auf diesem Kantenzug liegen. Es gilt nicht einmal, daß der Abarbeitungspfad maximaler Kosten nur aus Kanten besteht, die in der Kantenmenge des Zirkulationssubgraphen sind. Der Worst Case Pfad enthält die Kanten e_9, e_{12} und e_{13}, dafür aber nicht den Kantenzug $(e_2, \ldots, e_8)$. Die einzige Aussage, die man treffen kann, ist, daß die maximalen Kosten einer Zirkulation eine obere Schranke für die maximale Abarbeitungszeit bilden.

Das Problem für die MAXT-Berechnung ist, daß zulässige Zirkulationen nicht "zusammenhängend" sein müssen, d.h. der von ihnen induzierte Zirkulationssubgraph muß *nicht stark zusammenhängend* sein. Der Grund dafür liegt darin, daß die betrachteten Graphen G nicht kreisfrei sein müssen. Klarerweise erfüllt jeder in diesen Kreisen zirkulierende Fluß für sich die Anforderungen der Zirkulationen. Die Kreise mit positivem Fluß müssen daher nicht auf einem Programmpfad liegen, um Teile gültiger Zirkulationen zu sein.

Um die maximale Abarbeitungszeit trotzdem mit Hilfe von Zirkulationen ermitteln zu können, muß man die Anforderungen an diese Zirkulationen verschärfen. Es wird die zusätzliche Forderung eingeführt, daß der von der Zirkulation induzierte Subgraph stark zusammenhängend sein muß.

Satz: Für einen Graphen G, dem der Erweiterte T-Graph G' mit den Restriktionen b und c entspricht, ist die maximale Abarbeitungszeit gleich den Kosten einer Zirkulation maximaler Kosten, deren induzierter Zirkulationssubgraph stark zusammenhängend ist.

Beweis: Es ist zu zeigen, daß es zu jedem Zirkulationssubgraphen G_f mindestens einen Programmpfad gibt, bzw. daß jedem Programmpfad genau eine Zirkulation mit zusammenhängedem Zirkulationssubgraphen entspricht. Dann folgt wegen der Definition der maximalen Abarbeitungszeit und dem Zusammenhang von Abarbeitungszeiten und Zirkulationskosten sofort obiges.

Nach Folgerung 1 von Seite 38 entspricht jedem Kantenzug $(s, \ldots, t)$ in G eine Zirkulation in G'.

Umgekehrt enthält jeder Zirkulationssubgraph wegen $b((t,s)) = c((t,s)) = f((t,s)) = 1$ die Knoten s und t. Wegen des Zusammenhangs zwischen Flüssen und Kantenzügen kann der Zirkulation genau die Menge von Programmpfaden zugeordnet werden, für die die Anzahl der Vorkommnisse jeder Kante e dem Fluß durch die Kante $f(e)$ entspricht. Alle diese Pfade haben laut Definition dieselbe Abarbeitungszeit. □

Die Beschreibung eines erweiterten T-Graphen soll nun so modifiziert werden, daß nur mehr Zirkulationen zulässig sind, die einen Zirkulationssubgraphen mit starkem Zusammenhang induzieren. Zu diesem Zweck soll es möglich sein, Flüsse verschiedener Kanten zueinander in Relation zu stellen.

Def.: Eine Ungleichung bzw. Gleichung der Form

$$\sum_{e_i \in E'} a_i f(e_i) \circ \sum_{e_i \in E'} a_i' f(e_i) + k,$$

mit $a_i, a_i' \in \mathbf{Z_0}$, $k \in \mathbf{N_0}$ und $\circ \in \{<, \leq, =\}$, die die Flüsse von Kanten eines erweiterten T-Graphen G' in Beziehung setzt, heißt *Flußrestriktion* oder kurz *Restriktion* auf G'.

Bemerkung: Jede Restriktion der obigen Form kann ohne Einschränkung durch die Subtraktion von Termen der linken Seite ($b_i = a_i - a_i'$) in die "Normalform"

$$\sum_{e_i \in E'} b_i f(e_i) \circ k$$

gebracht werden. Diese Form wird in der Folge noch öfters verwendet werden.

Um einen stark zusammenhängenden Zirkulationssubgraphen zu erhalten, dürfen die Kanten jedes Kreises nur dann einen Fluß größer Null aufweisen, wenn es im Zirkulationsgraphen einen Kantenzug gibt, der vom Startknoten zum Kreis führt, d.h. es existiert ein Fluß größer Null zu diesem Kreis und auch von diesem Kreis weg (Flußerhaltung). Die modifizierte Beschreibung des T-Graphen soll sicher stellen, daß nur solche Flüsse auf erweiterten T-Graphen zulässig sind.

Def.: Seien e_i eine Kante und E_j eine Teilmenge der Kantenmenge eines T-Graphen G. Dann gilt: e_i *impliziert* E_j genau dann, wenn jeder Abarbeitungspfad der e_i enthält auch mindestens eine Kante aus E_j enthält.

Ein Kantenzug $(e_{i_1}, \ldots e_{i_m})$ der ausschließlich Kanten enthält, die E_j implizieren, heißt E_j *implizierender Kantenzug.*

Diese Definition wird im folgenden Satz verwendet:

Satz: Ist G' ein erweiterter T-Graph, in dem für jeden Kreis mit Kantenmenge E_c und implizierenden Kanten E_i der Fluß mindestens einer Kante e_c des Kreises durch eine Restriktion der Form

$$f(e_c) \leq \sum_{e_j \in E_i} a_j f(e_j)$$

eingeschränkt ist, so induzieren alle zulässigen Zirkulationen stark zusammenhängende Zirkulationssubgraphen, d.h. es sind nur Zirkulationen zulässig, denen Programmabarbeitungen entsprechen. Die Kosten einer Zirkulation maximaler Kosten sind gleich der maximalen Abarbeitungszeit eines Programmes, dessen Pfade den gegebenen Restriktionen gehorchen.

Beweis: Daß den zusammenhängenden Zirkulationssubgraphen Programmabarbeitungen entsprechen wurde bereits gezeigt. Bleibt zu beweisen, daß die genannten Bedingungen den starken Zusammenhang implizieren.

Beweis indirekt: Es wird angenommen, daß der induzierte Zirkulationssubgraph nicht stark zusammenhängend ist. Es gibt dann einen Kreis mit Kanten E_c und Restriktionen R, der im Zirkulationssubgraphen nicht von s erreichbar ist. Das bedeutet für alle Kanten des Kreises e_c, daß $f(e_c) > 0$ und für die E_c implizierenden Kanten $e_i \in E_i$, daß $f(e_i) = 0$. Da nun der Fluß mindestens einer Kante e_c des Kreises durch eine Restriktion obiger Form begrenzt ist und wegen $f(e_i) = 0$ für alle $e_i \in E_i$ erhält man $f(e_c) = 0$. Daraus folgt sofort, daß der Fluß im ganzen Kreis gleich Null sein muß. Dies steht im Widerspruch zur Annahme und beweist den Satz. □

4.2.3 Eigenschaften von Erweiterten T-Graphen mit Restriktionen

Es wurden bisher zwei Mechanismen zur Beschränkung der zulässigen Flüsse in Kanten von erweiterten T-Graphen beschrieben:

- In Abschnitt 4.2.1 wurden die Kapazitätsfunktionen b und c eingeführt, die für jede Kante e den zulässigen Fluß definiert ($\forall e : b(e) \leq f(e) \leq c(e)$).
- Oben wurden Restriktionen eingeführt, um den Fluß in Kreisen derart zu beschränken, daß nur zusammenhängende Zirkulationen zugelassen werden.

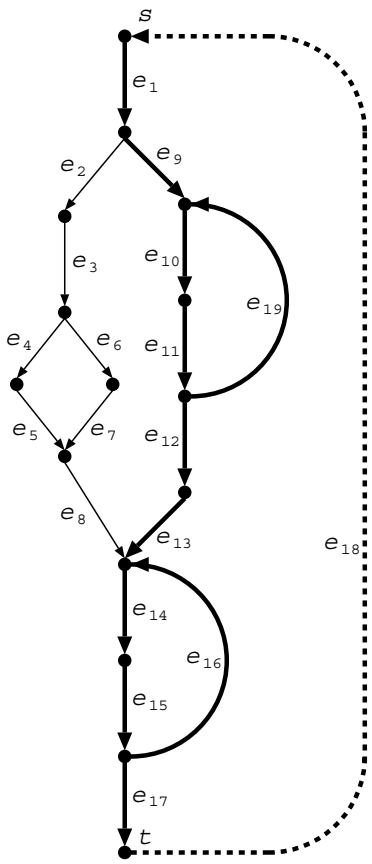
In der Folge sollen nur mehr Restriktionen verwendet werden, um den Fluß in Kanten zu begrenzen. Wie bereits oben gezeigt wurde, können mit diesen allgemeinere Zusammenhänge als mit Kapazitätsfunktionen ausgedrückt werden. Sind die zulässigen Flüsse für einen Graphen dennoch durch Kapazitätsfunktionen b und c gegeben, so kann auf einfache Art eine äquivalente Menge von Restriktionen erzeugt werden: Für jede Kante e mit Funktionswerten $b(e)$ und $c(e)$ werden die Ungleichungen $b(e) \leq f(e)$ und $f(e) \leq c(e)$ in die Menge der Restriktionen aufgenommen.

Eine weitere Vereinfachung betrifft die Anzahl der Restriktionen, die für einen T-Graphen angegeben werden muß. Im speziellen sollen die Restriktionen betrachtet werden, die den Fluß von Kanten nach oben hin abschranken. Für die Abschrankung nach unten hin gilt ohnehin $f((t,s)) \geq 1$ bzw. $f(e) \geq 0$ für die restlichen e, sofern es nicht ausdrücklich anders angegeben wird. Bei der Beschreibung der Funktionen c war notwendig, daß jeder Kante des T-Graphen ein Funktionswert zugewiesen wurde. Viele dieser Funktionswerte sind in der Regel redundant, da sie aufgrund des Flußerhaltungssatzes ohnehin aus den Werten der benachbarten Kanten abgeleitet werden können (siehe Beispiel 4.3). Es soll daher angegeben werden, welche bzw. wieviele Restriktionen ausreichen, um die Zirkulationen, die Programmpfaden entsprechen, zu beschreiben:

1. Da die ermittelten Zirkulationen genau einem Abarbeitungspfad entsprechen sollen, muß auf jeden Fall gelten, daß $f((t,s)) = 1$. Damit ist aufgrund des Flußerhaltungssatzes für die Erweiterung eines kreisfreien T-Graphen bereits der maximale Fluß aller Kanten bestimmt.
2. Für jeden erweiterten T-Graphen G' mit zugehörigem, nicht kreisfreiem T-Graphen G muß gelten, daß für jeden Kreis in G eine Flußrestriktion bezüglich der implizierenden Kanten (siehe Satz) existiert. Wegen des Flußerhaltungssatzes beschränkt dann die Restriktion des Flusses der Kante den Fluß im gesamten Kreis. Laut dem Satz entsprechen außerdem alle Zirkulationen Programmpfaden.

Abbildung 4.5 zeigt ein Beispiel für einen erweiterten T-Graphen mit Restriktionen. Dieser Graph wurde aus dem T-Graphen von Abbildung 4.4 erzeugt, indem die

Kapazitätsfunktionen durch Flußrestriktionen ersetzt wurden ($f(e_i) > 0$ ist nicht extra angeführt). Die Restriktionen für die Kanten e_{10} bzw. e_{14} schranken den Fluß dieser Kanten jeweils relativ zum Fluß von Kanten, die deren Kreise implizieren, ab. Auf diese Weise sind nur zusammenhängende Flüsse zulässig. Der dem eingezeichneten maximalen Fluß entsprechende Kantenzug (fette Kanten) ist gleichzeitig der Worst Case Abarbeitungspfad, dessen Durchlaufen 1262 Zeiteinheiten benötigt.



e	$\gamma(e)$	$f(e)$
e_1	36	1
e_2	8	0
e_3	86	0
e_4	8	0
e_5	112	0
e_6	10	0
e_7	30	0
e_8	46	0
e_9	10	1
e_{10}	12	8
e_{11}	26	8
e_{12}	32	1
e_{13}	24	1
e_{14}	44	10
e_{15}	20	10
e_{16}	10	9
e_{17}	56	1
e_{18}	0	1
e_{19}	10	7

$f(e_{18}) = 1$
$f(e_{10}) \leq 7f(e_9)$
$f(e_{14}) \leq 10f(e_8) + 10f(e_{13})$

Abbildung 4.5: Eine Zirkulation maximaler Kosten auf einem erweiterten T-Graphen mit Restriktionen.

Vollständige Pfadbeschreibung durch Restriktionen

Zum Abschluß dieses Kapitels soll noch eine zentrale Aussage über die Mächtigkeit des hier eingeführten Verfahrens zur Beschreibung des Verhaltens von Programmen gemacht werden. Das dynamische Verhalten eines Codestückes kann mit Hilfe von erweiterten T-Graphen und Restriktionen so charakterisiert werden, daß die Berechnung der maximalen Abarbeitungszeit möglich ist.

Satz: Mit erweiterten T-Graphen und Restriktionen kann die Menge von erlaubten Abarbeitungspfaden eines Codestückes so beschrieben werden, daß die maximale

Abarbeitungszeit (nicht bloß eine obere Schranke) für dieses Codestück berechnet werden kann (*Vollständigkeit* der Beschreibung in Bezug auf die Berechenbarkeit der maximalen Abarbeitungszeit).

Beweis: Gegeben sei ein erweiterter T-Graph G'. Daß es durch geeignete Wahl von Restriktionen R möglich ist, eine obere Schranke für die maximale Abarbeitungszeit zu berechnen wurde bereits gezeigt. Hier ist zu zeigen, daß die Angabe einer Menge von Restriktionen die genaue Berechnung der maximalen Abarbeitungszeit erlaubt. Der Beweis wird durch Induktion nach Anzahl der Pfade geführt.

1. Betrachtet man einen einzelnen Abarbeitungspfad $P_1 = (e_{p_1}, \ldots, e_{p_m})$, so kann diesem ein n-Tupel $F(P) = (f_{p_0}, \ldots, f_{p_{|E|}})$ zugeordnet werden, für das gilt, daß f_{p_i} jeweils die Anzahl der Vorkommnisse von e_i in P_1 ist. Daraus konstruiert man eine Menge von Restriktionen R_1,

   $$R_1 := \{f(e_i) \leq f_{p_i} : 1 \leq i \leq |E|\}.$$

   Maximiert man die Kosten von möglichen Zirkulationen unter diesen Einschränkungen so erhält man:

   $$maxt(P_1) = xt(P_1) = \sum_{i=1}^{|E|} f_{p_i} t(e_i) = \max_{R_1} \sum_{i=1}^{|E|} f(e_i) t(e_i).$$

   Damit stimmt der obige Satz wenn ein Programmpfad zulässig ist.

2. Die Induktionsannahme lautet, daß für n Pfade, die durch den erweiterten T-Graphen und die Restriktionen R_n beschrieben sind, die maximale Abarbeitungszeit berechnet werden kann.

   Induktionsbehauptung: Eine Menge von $n+1$ Pfaden kann auch so charakterisiert werden, daß man die maximale Abarbeitungszeit berechnen kann.

   Sei R_n die Menge von Restrikionen für n Pfade, die laut Annahme die Berechnung der maximalen Abarbeitungszeit erlaubt. Weiters sei $\overline{R}_{n+1}$ die Menge von Restriktionen, die dem weiteren Pfad P_{n+1} zugeordnet ist. Da P_{n+1} nicht in $\{P_1, \ldots, P_n\}$, kann man zwei Fälle unterscheiden: entweder P_{n+1} oder ein Pfad aus $\{P_1, \ldots, P_n\}$ hat maximale Abarbeitungszeit. Es gelten daher entweder die Restriktionen R_n oder $\overline{R}_{n+1}$.

   Mit einem Trick, der in der Linearen Programmierung verwendet wird [Hil88], kann man R_n und $\overline{R}_{n+1}$ wie folgt disjunktiv zusammenfassen. Man wählt eine Konstante M, die sehr groß ist (größer als alle bisherigen rechten Seiten der Ungleichungen). Dann muß entweder

   $$f(e_i) \leq f_{p_i} \text{ für } R \in R_n \quad \text{und} \quad f(e_i) \leq f_{p_i} + M \text{ für } R \in \overline{R}_{n+1},$$
   $$\text{oder}$$
   $$f(e_i) \leq f_{p_i} + M \text{ für } R \in R_n \quad \text{und} \quad f(e_i) \leq f_{p_i} \text{ für } R \in \overline{R}_{n+1}.$$

gelten. Man kann diese Mengen von Ungleichungen nach Einführung einer binären Variable y und einer zusätzlichen Restriktion für y wie folgt zu einer einzigen Menge von Restriktionen verknüpfen:

$$\begin{array}{ll} f(e_i) \leq f_{p_i} + yM & \text{für } R \in R_n \\ f(e_i) \leq f_{p_i} + (1-y)M & \text{für } R \in \overline{R}_{n+1} \\ y \leq 1 & \end{array}$$

Durch entsprechendes Umschreiben der Ungleichungen (die yM müssen auf die linke Seite gebracht werden) erhält man wieder ein System von Ungleichungen der obigen Form. Es gilt:

$$maxt(\{P_1, \ldots, P_{n+1}\}) = \max(maxt(\{P_1, \ldots, P_n\}, maxt(P_{n+1})) =$$

$$\max(\max_{R_n} \sum_{i=1}^{|E|} f(e_i)t(e_i), \max_{\overline{R}_{n+1}} \sum_{i=1}^{|E|} f(e_i)t(e_i)) = \max_{R_{n+1}=R_n \oplus \overline{R}_{n+1}} \sum_{i=1}^{|E|} f(e_i)t(e_i).$$

Der Operator $\oplus$ steht dabei für die obige disjunktive Verknüpfung der Restriktionen.

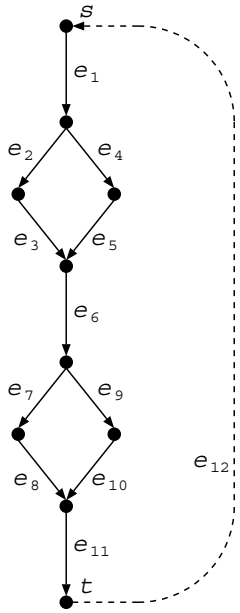
Damit ist gezeigt, daß beliebig große Mengen von Abarbeitungspfaden mit Hilfe von Restriktionen so beschrieben werden können, daß die maximale Abarbeitungszeit berechnet werden kann. □

Geht man bei der Beschreibung von Restriktionen für Abarbeitungspfade so vor, wie dies im Beweis geschieht, so erhält man sehr schnell eine große Menge an Restriktionen. Dies würde die praktische Durchführbarkeit der MAXT-Analyse für eine beliebige Analysemethode in Frage stellen. In der Praxis wird man daher erstens nicht sämtliche Restriktionen angeben (daß dies auch nicht notwendig ist, wurde bereits oben gezeigt) und zweitens die Beschreibung nicht auf die elementaren Restriktionen beschränken. Es sind beliebige lineare Ungleichungen, die durch Zusammenfassen und Vereinfachung von elementaren Ungleichungen entstehen können, für die Charakterisierung des Verhaltens eines Codestückes zulässig. Diese Vorgehensweise wurde bereits im Beispiel von Abbildung 4.5 verwendet.

Das Beispiel aus Abbildung 4.6 veranschaulicht das Gesagte. In dem Programmstück seien drei Abarbeitungspfade erlaubt:

$$\begin{array}{l} (e_1, e_2, e_3, e_6, e_7, e_8, e_{11}), \\ (e_1, e_2, e_3, e_6, e_9, e_{10}, e_{11}), \\ (e_1, e_4, e_5, e_6, e_9, e_{10}, e_{11}). \end{array}$$

Der Pfad $(e_1, e_4, e_5, e_6, e_7, e_8, e_{11})$ soll nicht zulässig sein. Dieser Sachverhalt wird durch die zweite Restriktion ausgedrückt, indem die Summe der Flüsse durch die Kanten e_5 und e_8 mit 1 beschränkt wird. Weitere Restriktionen zur Beschreibung des



e	$\gamma(e)$	$f(e)$
e_1	44	1
e_2	8	1
e_3	80	1
e_4	10	0
e_5	132	0
e_6	56	1
e_7	8	1
e_8	82	1
e_9	10	0
e_{10}	12	0
e_{11}	46	1
e_{12}	0	1

$f(e_{12}) = 1$
$f(e_5) + f(e_8) \leq 1$

Abbildung 4.6: Ein erweiterter T-Graphen mit Restriktionen, die sich nicht vollständig durch Kapazitätsschranken ausdrücken lassen.

Sachverhalts sind aufgrund des Flußerhaltungssatzes nicht notwendig. Man beachte außerdem, daß mit Hilfe der obigen Beschreibung die maximale Abarbeitungszeit des Programmstückes durch Ermittlung einer Zirkulation maximaler Kosten mit 324 Zeiteinheiten berechnet werden kann. Hätte man nur Kapazitätsfunktionen zur Verfügung, so ließen sich bestenfalls für alle Kanten getrennt die maximalen Flüsse (gleich 1 für alle Kanten) beschreiben. Der obige Zusammenhang könnte nicht ausgedrückt werden. Es könnte nur eine obere Schranke für die maximale Abarbeitungszeit, die Abarbeitungszeit des unzulässigen Pfades, von 378 Zeiteinheiten berechnet werden.

4.3 Zusammenfassung

In diesem Kapitel wurde gezeigt, wie Programmstücke für die Zeitanalyse in Form von gerichteten, zusammenhängenden Graphen präsentiert werden können. Diese Graphen haben zwei ausgezeichnete Knoten, eine Quelle und eine Senke. Eine Programmabarbeitung entspricht einem gerichteten Kantenzug von der Quelle zur Senke.

Um die maximale Abarbeitungszeit für eine Menge von Pfaden berechnen zu können, ist es notwendig, die möglichen Abarbeitungen zu beschreiben. Da eine enumerative Darstellung der Pfade praktisch nicht handhabbar ist, wurde nach einer alternativen Repräsentation gesucht, die ebenfalls erlaubt, Aussagen über das Zeitverhalten der Pfade zu machen. Die ursprünglichen Graphen wurden auf zyklische Graphen, Erweiterte T-Graphen genannt, abgebildet und die Berechnung der maximalen Ab-

arbeitungszeit auf die Ermittlung einer Zirkulation maximaler Kosten im erweiterten T-Graphen zurückgeführt.

Die Kapazitätsfunktionen, die zur Abschrankung des Flusses in Zirkulationen verwendet werden, haben nur lokale Auswirkung. Komplexere Zusammenhänge, wie sie zwischen Teilen von Programmpfaden auftreten, können mit ihnen nicht dargestellt werden. Aus diesem Grund wurden Flußrestriktionen in Form von Gleichungen bzw. Ungleichungen eingeführt. Mit ihnen kann man Flüsse verschiedener Kanten in Beziehung setzen. Es wurde gezeigt, daß jedes Codestück und seine möglichen Abarbeitungspfade mit erweiterten T-Graphen und Restriktionen so beschrieben werden kann, daß auch tatsächlich die Berechnung seiner maximalen Abarbeitungszeit mit Hilfe von Zirkulationen möglich ist.

Kapitel 5

Die Berechnung der Maximalen Abarbeitungszeit mittels ganzzahliger linearer Programmierung

Im vorigen Kapitel wurde eine Methode vorgestellt, um das dynamische Ablaufverhalten von Programmen zu beschreiben. Programme wurden in Form von Erweiterten T-Graphen dargestellt und die Menge von möglichen Pfaden durch Restriktionen für Zirkulationen auf diesen Graphen beschrieben. Es wurde gezeigt, daß die Berechnung der Maximalen Abarbeitungszeit dem Problem der Ermittlung einer Zirkulation maximaler Kosten auf dem T-Graphen mit Restriktionen entspricht.

In diesem Teil der Arbeit soll nun gezeigt werden, wie das oben erläuterte Zirkulationsproblem als Lineares Programmierungsproblem [Hil88, Nem89] repräsentiert und gelöst werden kann. Den Anfang des Kapitels bildet eine kurze Charakterisierung von Linearen Programmierungsproblemen. Der nächste Teil zeigt, wie die Aufgabe, eine Zirkulation maximaler Kosten auf einem T-Graphen zu finden, als Lineares Optimierungsproblem (Programmierungsproblem) formuliert und mit einem entsprechenden Softwarepaket gelöst werden kann. In der Folge wird diskutiert, wie man die Ergebnisse der Lösung des Programmierungsproblems in Hinblick auf die Zeitanalyse verwenden kann. Den Abschluß bildet ein Exkurs darüber, wie weit man Ergebnisse einer Sensitivitätsanalyse für Aussagen über Abarbeitungszeiten verwenden kann.

5.1 Kurzer Überblick über die Lineare Programmierung

Ein lineares Programmierungsproblem ist ein mathematisches Problem, bei dem es darum geht, den Wert einer Zielfunktion unter Einhaltung einer Menge von Restriktionen zu maximieren. Die in [Hil88] angeführte Standardform eines linearen Programmierungsproblems hat folgendes Aussehen:

Maximiere

$$Z = c_1x_1 + c_2x_2 + \ldots + c_nx_n \tag{5.1}$$

unter den Restriktionen

$$\begin{array}{rcrcrcrcl} a_{11}x_1 & + & a_{12}x_2 & + & \ldots & + & a_{1n}x_n & \leq & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \ldots & + & a_{2n}x_n & \leq & b_2 \\ & & & & \vdots & & & & \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \ldots & + & a_{mn}x_n & \leq & b_m \end{array} \tag{5.2}$$

und

$$x_i \geq 0 \ f\ddot{u}r\ 1 \leq i \leq m. \tag{5.3}$$

In linearen Programmierungsmodellen wird die folgende Terminologie verwendet. Es ist $c_1x_1 + \ldots + c_nx_n$ die zu maximierende Funktion, die *Zielfunktion*. Die Ungleichungen aus 5.2 sind die (funktionellen) *Nebenbedingungen* oder *Restriktionen*. Weiters werden noch die *Nichtnegativitätsbedingungen* 5.3 gefordert. Die x_i heißen *Entscheidungsvariablen* oder kurz *Variablen* und die a_{ij}, b_i und c_j *Parameter* des Modells.

Jedes n-Tupel von Werten für die Variablen $x_1, \ldots, x_n$ wird als *Lösung* bezeichnet. Eine Lösung heißt *zulässig*, wenn sie alle Nebenbedingungen erfüllt. Eine zulässige Lösung, die den besten (maximalen) Zielfunktionswert Z ergibt, heißt *optimale Lösung*.

Nicht jedes Modell muß in der oben angegebenen Standardform vorliegen. Es gibt einige erlaubte Abweichungen, die sich leicht wieder in eine Formulierung mit obigem Aussehen bringen lassen [Hil88] (vor allem die Abweichung aus Punkt 3 wird in der Folge die Schreibweise der Restriktionen vereinfachen):

1. Die Zielfunktion soll nicht maximiert, sondern *minimiert* werden.
2. Es treten Nebenbedingungen mit umgekehrten Vergleichsoperator, mit dem Aussehen $a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n \geq b_i$, auf. Statt "$\geq$" kann auch "$>$" stehen.
3. Einige funktionale Nebenbedingungen haben Gleichungsform.
4. Für einige Entscheidungsvariablen ist die Nichtnegativitätsbedingung aufgehoben.

Damit eine Aufgabe als lineares Programmierungsproblem gelöst werden kann, muß sie erst so beschrieben werden, daß sie den vier grundlegenden Annahmen der linearen Programmierung gehorcht. Diese vier Annahmen sind:

- *Proportionalität:* Betrachtet man eine Variable x_i, so ist der Beitrag dieser Variable in den Ungleichungen, $a_{ji}x_i$, und ihr Beitrag zum Wert der Zielfunktion, c_ix_i proportional zu x_i.
- *Additivität:* Die Additivität besagt, daß die einzelnen Werte x_i voneinander unabhängig sind. Die Werte $a_{ji}x_i$ und c_ix_i gehen additiv in die Nebenbedingungen bzw. die Zielfunktion ein. Es gibt keine wechselseitigen Abhängigkeiten von Variablen, die zu multiplikativen Zusammenhängen in Termen der Zielfunktion oder der Ungleichungen führen.
- *Teilbarkeit:* Um die optimale Lösung eines linearen Programmierungsproblems berechnen zu können, müssen die Variablen nicht ganzzahlige Werte annehmen können. Da es Probleme gibt, in denen ganzzahlige Lösungen gesucht werden (dies gilt auch für den Aufgabenbereich dieser Arbeit), wurden spezielle Verfahren zur *Ganzzahligen Programmierung* entwickelt. Es gibt jedoch auch Gruppen von Problemstellungen der linearen Programmierung, die immer zu ganzzahligen Lösungen führen.
- *Bestimmtheit:* Diese Annahme besagt, daß alle Parameter des Modells, a_{ij}, b_i und c_j, bekannt sind und konstante Werte haben.

Ein weit verbreitetes Verfahren zur Lösung von linearen Programmierungsproblemen ist das *Simplexverfahren*. Dieses Verfahren läßt sich leicht auf Computern implementieren. Es existiert heute eine Vielzahl von zum Teil sehr leistungsfähigen Softwarepaketen, die auch zur Lösung großer Probleme herangezogen werden [Hil88]. Den Programmen werden Zielfunktionen und Restriktionen entweder in Matrixform oder in der Form von Gleichungs-/Ungleichungssystemen übergeben. Das Resultat besteht aus dem ermittelten besten Zielfunktionswert Z, sowie den entsprechenden Belegungen der Variablen x_i.

5.2 Die Berechnung der maximalen Kosten einer Zirkulation mit Restriktionen — Ein ganzzahliges lineares Programmierungsproblem

Das Verfahren der linearen Programmierung soll nun verwendet werden, um die Maximale Abarbeitungszeit eines Programmstückes zu berechnen. Dazu muß die in Abschnitt 4.2 hergeleitete Beschreibung des zur MAXT-Berechnung äquivalenten Pro-

blems — der Ermittlung einer Zirkulation maximaler Kosten auf einem erweiterten T-Graphen mit Restriktionen — als Programmierungsproblem formuliert werden.

Die zu lösende Problemstellung soll hier nochmals kurz zusammengefaßt werden:

> *Gegeben* ist ein erweiterter T-Graph $G' = (V', E')$. Das ist ein Graph, der aus einem zusammenhängenden gerichteten Graphen mit genau einer Quelle s, einer Senke t und der Eigenschaft, daß jede Kante Teil eines Kantenzuges von der Quelle zur Senke ist, dadurch entsteht, daß man eine zusätzliche gerichtete Kante von der Senke zur Quelle einfügt. Dieser Graph besteht aus Knoten v_i und Kanten e_i, wobei jeder Kante eine Abarbeitungszeit $t_i = t(e_i)$, auch genannt Kosten der Kante, zugeordnet ist.
>
> Weiters sind auf dem Graphen Zirkulationen definiert, die dem Flußerhaltungssatz gehorchen, und es existiert eine Menge von Restriktionen R, die die zulässigen Flüsse in den Kanten begrenzen.
>
> *Gesucht* ist eine zulässige ganzzahlige Zirkulation f maximaler Kosten auf dem erweiterten T-Graphen.

5.2.1 Formulierung des Zirkulationsproblems als Programmierungsproblem

Man kann die gegebene Problemstellung nun in den folgenden Schritten in eine Beschreibungsform für ein lineares Programmierungsproblem bringen (die hier angegebene Reihenfolge der Schritte orientiert sich an der Beschreibung eines Programmierungsproblems):

1. Die zu maximierende *Zielfunktion* resultiert aus der Kostenfunktion der Zirkulation:

$$Z = \sum_{i=1}^{|E'|} f_i t_i.$$

   Die t_i sind die Abarbeitungszeiten der Kanten. Die f_i, die den Fluß in den Kanten beschreiben, bilden die Variablen des Programmierungsproblems.

2. Die *funktionalen Nebenbedingungen* ergeben sich aus der Struktur des Graphen und den zusätzlichen Einschränkungen für die Zirkulation.

   - Der Graph wird durch eine Menge von Nebenbedingungen repräsentiert, die die Stuktur des Graphen in Form von Flußgleichungen beschreiben (eine

ähnliche Vorgangsweise ist in [McH90] für die Lösung des "Shortest Path" Problems zu finden). Für jeden Knoten v_i wird eine Gleichung der Form[1]

$$\sum_{e_j^+ = v_i} f_j = \sum_{e_k^- = v_i} f_k$$

erzeugt. Der Fluß durch die zum Knoten v_i führenden Kanten e_j^+ muß dem Fluß der von Knoten wegführenden Kanten e_k^- gleichen.

- Die Restriktionen R werden in das Programmierungsproblem übernommen. Die Gleichungen/Ungleichungen vom Aussehen

$$\sum_{e_j \in E'} a_{ij} f_j \circ \sum_{e_j \in E'} a'_{ij} f_j + k,$$

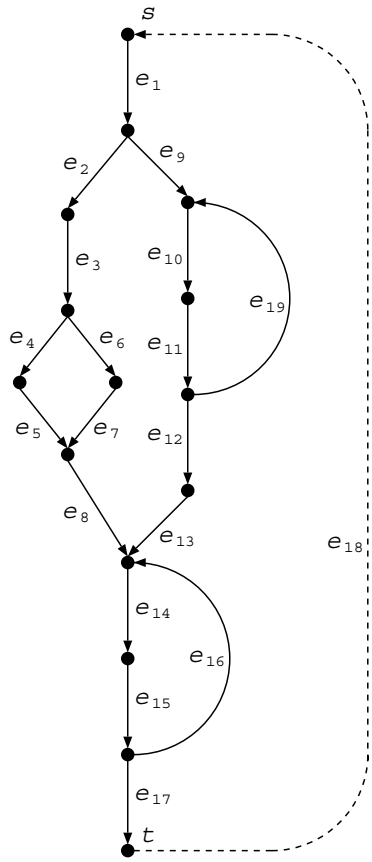
mit $a_{ij}, a'_{ij} \in \mathbf{Z_0}$, $k \in \mathbf{N_0}$ und $\circ \in \{\leq, <, =, >, \geq\}$ beschreiben Restriktionen für Kreise (Schleifen) und sonstige Restriktionen und Zusammenhänge von Programmteilen. Außerdem enthalten sie die Restriktion $f_{|E'|} = 1$.

3. Für die Flüsse aller Kanten wird die *Nichtnegativitätsbedingung*, $f_j \geq 0$, in die Menge der Restriktionen aufgenommen. Bei vielen Softwarepaketen brauchen diese Bedingungen nicht extra angegeben werden. Die Nichtnegativität wird in diesen Programmen implizit angenommen (siehe auch [WR92]).

Die eben beschriebene Umwandlung soll anhand von Abbildung 5.1 illustriert werden. Die obere Hälfte des Bildes zeigt den erweiterten T-Graphen mit Restriktionen, der bereits in Abbildung 4.5 von Seite 45 verwendet wurde. Im unteren Teil ist das entsprechende Programmierungsproblem zu sehen. Die Zielfunktion ist die Summe der Ausführungszeiten aller Kanten, jeweils mit der vor der Lösung des Problems noch unbekannten Anzahl ihrer Ausführungen gewichtet. Die Graphbeschreibung enthält für jeden Knoten genau eine Gleichung zur Beschreibung des Flusses. Man beachte, daß Kanten, die Teile von Kreisen sind, auf die gleiche Art wie alle anderen Kanten in diese Beschreibung eingehen. Eine Beschränkung der in Kreisen zirkulierenden Flüsse erfolgt durch zwei zusätzliche Ungleichungen (siehe rechte Seite), die unmittelbar aus den Restriktionen des T-Graphen übernommen werden. Die Gleichung $f_{18} = 1$ skaliert das Problem, sodaß die Anzahl der Pfade von s nach t gleich eins ist. Für den Fluß auf allen Kanten gilt die Nichtnegativitätsbedingung.

Löst man das Programmierungsproblem aus Abbildung 5.1, z.B. mit dem Simplexverfahren, so erhält man die optimale Lösung aus Abbildung 5.2. Man erhält einen maximalen Zielfunktionswert von 1262 Zeiteinheiten und die gezeigten Variablenbelegungen (Flüsse auf den Kanten). Dieses Ergebnis entspricht der bereits in Abschnitt 4.2.3 händisch ermittelten Lösung.

[1] Die Form der Gleichungen entspricht nicht der oben angegebenen Standardform, kann aber durch Subtraktion der f_k von beiden Seiten leicht auf diese gebracht werden.



e	$\gamma(e)$
e_1	36
e_2	8
e_3	86
e_4	8
e_5	112
e_6	10
e_7	30
e_8	46
e_9	10
e_{10}	12
e_{11}	26
e_{12}	32
e_{13}	24
e_{14}	44
e_{15}	20
e_{16}	10
e_{17}	56
e_{18}	0
e_{19}	10

$f(e_{18}) = 1$
$f(e_{10}) \leq 7f(e_9)$
$f(e_{14}) \leq 10f(e_8) + 10f(e_{13})$

Zielfunktion des Programmierungproblems:

$$\begin{aligned} Z \quad = \quad & 36f_1 + 8f_2 + 86f_3 + 8f_4 + 112f_5 + 10f_6 + 30f_7 + 46f_8 + 10f_9 + 12f_{10} + \\ & 26f_{11} + 32f_{12} + 24f_{13} + 44f_{14} + 20f_{15} + 10f_{16} + 56f_{17} + 10f_{19}. \end{aligned}$$

Graphbeschreibung:

$$\begin{aligned} f_{18} &= f_1 \\ f_1 &= f_2 + f_9 \\ f_2 &= f_3 \\ f_3 &= f_4 + f_6 \\ f_4 &= f_5 \\ f_6 &= f_7 \\ f_5 + f_7 &= f_8 \\ f_9 + f_{19} &= f_{10} \\ f_{10} &= f_{11} \\ f_{11} &= f_{19} + f_{12} \\ f_{12} &= f_{13} \\ f_8 + f_{13} + f_{16} &= f_{14} \\ f_{14} &= f_{15} \\ f_{15} &= f_{16} + f_{17} \\ f_{17} &= f_{18}. \end{aligned}$$

Rückkehrkante und Kreise:

$$\begin{aligned} f_{10} &\leq 7f_9 \\ f_{14} &\leq 10f_8 + 10f_{13} \\ f_{18} &= 1. \end{aligned}$$

Nichtnegativitätsbedingungen:

$$f_i \geq 0 \text{ für } 1 \leq i \leq 18.$$

Abbildung 5.1: Beschreibung eines MAXT-Problems als Programmierungsproblem

Zielfunktionswert:

$$Z = 1262.$$

Belegungen der Variablen:

$$\begin{array}{lllllll} f_1 = 1, & f_2 = 0, & f_3 = 0, & f_4 = 0, & f_5 = 0, & f_6 = 0, & f_7 = 0, \\ f_8 = 0, & f_9 = 1, & f_{10} = 8, & f_{11} = 8, & f_{12} = 1, & f_{13} = 1, & f_{14} = 10, \\ f_{15} = 10, & f_{16} = 9, & f_{17} = 1, & f_{18} = 1, & f_{19} = 7. & & \end{array}$$

Abbildung 5.2: Optimale Lösung des Programmierungsproblems aus Abbildung 5.1.

5.2.2 Ganzzahligkeit von Lösungen

Während man der hier behandelten Problemstellung schnell entnehmen kann, daß sie die Eigenschaften der Proportionalität, Additivität und Bestimmtheit erfüllt, soll hier kurz auf die Teilbarkeitseigenschaft eingegangen werden.

Aus dem ursprünglichen Problem heraus läßt sich ableiten, daß nur solche optimalen Lösungen von Interesse sind, die ganzzahlige Variablenbelegungen haben. Nicht ganzzahlige Variablenbelegungen können sicher keiner sinnvollen Programmabarbeitung entsprechen.

Oben wurde erwähnt, daß es Klassen von linearen Programmierungsproblemen gibt, die so beschaffen sind, daß sie auf jeden Fall ganzzahlige Lösungen liefern. Wie gezeigt werden kann, zählt die hier behandelte Problemstellung nicht zu diesen. Bereits das einfache Beispiel eines Graphen, der genau zwei verschiedene, mit gleichen Kosten behaftete Pfade, (e_1, e_2, e_4) bzw. (e_1, e_3, e_4), zuläßt, hat neben zwei ganzzahligen optimalen Lösungen ($f_1 = f_4 = 1$ und $f_2 = 1$, $f_3 = 0$ bzw. $f_2 = 0$, $f_3 = 1$) unendlich viele nicht ganzzahlige optimale Lösungen ($f_1 = f_4 = 1$ und $f_2 = 1 - f_3$ mit $0 < f_2 < 1$).

Ein allgemeines MAXT-Problem, das durch einen Erweiterten T-Graphen mit Restriktionen ausgedrückt und mittels linearer Programmierung analysiert wird, liefert also nicht automatisch eine ganzzahlige Lösung. Um dennoch die Ganzzahligkeit der Resultate zu erreichen ist es notwendig, das Programmierungsproblem mit den Methoden der ganzzahligen Programmierung [Kor71, Bur72, Nem88] zu lösen.

5.3 Ergebnisse der Programmierungsprobleme für die MAXT-Analyse

Es wurde nun gezeigt, wie man Programme zur Lösung von Programmierungsproblemen für die Berechnung der maximalen Abarbeitungszeiten von erweiterten T-Graphen mit Restriktionen einsetzen kann. Aus der optimalen Lösung eines Programmierungs-

problems zur MAXT-Berechnung mit Hilfe des Simplexverfahrens lassen sich die folgenden Informationen gewinnen:

- Die *Maximale Abarbeitungszeit*, d.h. die maximale Zeit, die ein Programmpfad durch den T-Graphen, der den Restriktionen gehorcht, zur Abarbeitung benötigt.

- Eine *Menge von Pfaden maximaler Abarbeitungszeit*. Die Problemlösung liefert nicht nur die MAXT, sondern auch Information darüber, durch welche Abarbeitungen, auf welchen Pfaden, dieser Wert zustande kommt. Diese Information kann aus den Werten der Entscheidungsvariablen am Ende der Analyse abgelesen werden. Die Belegung der Variablen charakterisiert eine Menge von Abarbeitungspfaden (alle gültigen Pfade, auf denen die Kanten e_i genau f_i mal abgearbeitet werden) mit maximaler Abarbeitungszeit.

  Die Werte der Entscheidungsvariablen liefern *eine* Menge von Pfaden maximaler Abarbeitungszeit. Diese Menge kann, muß aber nicht *alle* Pfade mit dieser Abarbeitungszeit enthalten. Es können durchaus noch weitere Pfadmengen existieren, die anderen, ebenfalls optimalen Lösungen des Programmierungsproblems entsprechen. Das Simplexverfahren liefert allerdings in einem Durchlauf nur (irgend)eine optimale Lösung.

  Will man alle Mengen von Pfaden, die maximale Abarbeitung haben, so muß man, abhängig von der verwendeten Software, entweder von der gefunden Lösung ausgehend nach weiteren suchen, oder die Analyse unter explizitem Ausschluß der bisher gefundenen optimalen Lösungen erneut starten.

- Den *Beitrag von Programmabschnitten zur MAXT*. Mit Hilfe der Variablenbelegungen kann leicht festgestellt werden, wieviel Zeit in bestimmten Progammteilen im Fall der ermittelten Worst Case Abarbeitungen verbraucht wird. Diese Zeit wird durch jene Teilsumme der Zielfunktion berechnet, die nur die Beiträge der Kanten des untersuchten Programmabschnitts enthält.

  Es gilt wieder derselbe Vorbehalt wie im vorigen Punkt. Der so berechnete Beitrag eines Programmteiles zur gesamten maximalen Abarbeitungszeit bezieht sich nur auf die betrachtete optimale Lösung. Für verschiedene optimale Lösungen kann der Beitrag ein und desselben Programmteiles ganz unterschiedlich aussehen.

- Information, *wie oft einzelne Kanten an einem Pfad maximaler Abarbeitungszeit beteiligt sind*. Diese Information entspricht genau den Werten der Entscheidungsvariablen am Ende der Analyse. Sie ist vor allem für sehr lokale Optimierungen von Teilen des Codes von Interesse, wenn diese Verbesserungen eine Reduzierung der maximalen Abarbeitungszeit zum Ziel haben. Grob kann man sagen: Je öfter eine Kante im Worst Case ausgeführt wird, desto mehr wirkt sich eine Verkürzung ihrer Abarbeitungszeit im Gesamtresultat aus. Diese Aussage ist nicht unbeschränkt gültig. Erstens gilt wieder der in den letzten beiden Punkten gemachte Vorbehalt. Außerdem kann es vorkommen, daß ein Programmteil ab

einer gewissen Verkürzung seiner Abarbeitungszeit nicht mehr Teil eines Pfades mit maximaler Abarbeitungszeit ist. Alle Reduktionen der Abarbeitungszeit, die über diese Grenze gehen, wirken sich nicht mehr auf die MAXT aus.

Die obigen Aussagen gehen davon aus, daß das zu analysierende Programm vollständig beschrieben ist (zur Vollständigkeit siehe Kapitel 4.2.3). Dies bezieht sich vor allem auf die Nebenbedingungen, die die Menge der durch den Graphen beschriebenen Pfade auf die tatsächlich erlaubten einschränken. Diese Restriktionen lassen sich nicht automatisch aus dem erweiterten T-Graphen ableiten und müssen daher explizit vom Benutzer zur Verfügung gestellt werden.

Sind die Restriktionen zur Beschreibung der auftretenden Programmpfade nicht vollständig, erlauben aber doch nach den minimalen Anforderungen aus Abschnitt 4.2.3 den Fluß in allen Kreisen abzuschranken, so gelten die obigen Aussagen nur in abgeschwächter Form. Es kann dann nur garantiert werden, daß die optimalen Lösungen des Programmierungsproblems eine *obere Schranke* für die maximale Abarbeitungszeit ist. Die den Lösungen entsprechenden Variablenbelegungen sind in diesem Fall Schätzungen in Bezug auf diese Schranke. Das bedeutet auch für die oben gemachten Aussagen über die Abarbeitungszeiten von Programmteilen und deren Beiträge zur gesamten Abarbeitungszeit, daß sie sich auf die berechneten Schranken beziehen und deshalb nur Richtwerte für Analysen bzw. Verbesserungen sein können.

5.3.1 MAXT-Berechnung und Sensitivitätsanalysen

Sensitivitätsanalysen werden verwendet, um herauszufinden, wie sich Veränderungen von Parametern eines Programmierungsmodells auf die Lösungen auswirken. Diese Analysen kommen aus typischen Operations Research Anwendungen, wo Modelle oft mit geschätzten Parametern erstellt und bewertet werden. Um Aussagen über die Stabilität dieser Lösung auch bei geänderten, von der Schätzung abweichenden Parametern zu erhalten, variiert man die Größen des Modells und untersucht die entsprechenden Resultate.

Für die Berechnung der maximalen Abarbeitungszeit wurde angenommen, daß alle Informationen — Abarbeitungszeiten und Restriktionen — bekannt sind, also nicht geschätzt werden müssen. Der einzige Punkt, der zu einer "Unschärfe" in den Ergebnissen führen kann, ist das Fehlen von Restriktionen, die der Benutzer zu liefern hat und die unmögliche Pfade von der Analyse ausschließen sollen. Dennoch ist es interessant, einen kurzen Blick auf die vier in [Hil88] genannten Fälle von Modellmodifikationen für Sensitivitätsanalysen im Hinblick auf das Problem der MAXT-Analyse zu werfen:

- *Veränderungen von Koeffizienten von Variablen*: Für die MAXT-Analyse selbst wird angenommen, daß die Koeffizienten der Variablen, c_i bzw. a_{ij}, bekannt und unveränderlich sind. Die Untersuchung von Veränderungen der c_i könnte jedoch

für die Abschätzung der Auswirkungen lokaler Programmveränderungen auf die gesamte Abarbeitungszeit dienen (siehe [Pos92]). Die c_i entsprechen den Abarbeitungszeiten der Kanten. Diese wiederum repräsentieren sequentielle Programmteile. Geplante Änderungen innerhalb dieser sequentiellen Programmteile könnten mit Hilfe der Sensitivitätsanalyse in Bezug auf ihren globalen Effekt bewertet werden.

- *Veränderungen der b_i*: Nachdem angenommen wird, daß Restriktionen, die den Erweiterten T-Graphen bzw. Einschränkungen der Programmpfade beschreiben, korrekt ermittelt werden können, scheint eine Veränderung der b_i im Kontext der MAXT-Analyse nicht sinnvoll.

- *Hinzufügen von neuen Variablen*: Variablen repräsentieren den Fluß auf Kanten bzw. dienen als binäre Hilfsgrößen.

  Während die Struktur des Codestückes gegeben ist, kann es durchaus sinnvoll sein, neue binäre Variable einzuführen, um zusätzliches Wissen über die gültige Pfadmenge in die Analyse einzubringen. Dies ist gleichzeitig mit der Einführung neuer Restriktionen, bzw. mit der Veränderung von Restriktionen verbunden.

- *Hinzufügen neuer Restriktionen*: Werden nach der Berechnung der maximalen Abarbeitungszeit neue Abhängigkeiten in einem Codestück erkannt, so wird man untersuchen, ob diese Abhängigkeit zur Berechnung einer kleineren Schranke für die MAXT führt: Erfüllt die gefundene optimale Lösung die Restriktion, so bringt diese keine Änderung im Ergebnis. Ist die aktuelle Lösung in Gegenwart der neuen Restriktion nicht zulässig, so muß eine neue Lösung berechnet werden. Daß dies gleichzeitig zu einer Verringerung des Ergebniswertes der MAXT-Analyse führt ist nur dann garantiert, wenn die bisherige optimale Lösung die einzige war. Andernfalls kann es weitere optimale Lösungen geben, die die hinzugefügten Restriktion erfüllen. Dann ändert sich der Ergebniswert nicht.

5.4 Zusammenfassung

In diesem Kapitel wurde gezeigt, wie das Problem der Berechnung der maximalen Abarbeitungszeit in erweiterten T-Graphen mit Restriktionen als Programmierungsproblem realisiert und gelöst werden kann. Die T-Graphen werden in eine Zielfunktion und eine Menge von Nebenbedingungen, die die Struktur des Graphen beschreiben, übersetzt. Nachdem auch die Restriktionen des T-Graphen an die Graphenbeschreibung angepaßt worden sind, wird das Programmierungsproblem mit dem Simplexverfahren gelöst. Als Ergebnis erhält man nicht nur die maximale Abarbeitungszeit des untersuchten Codestückes, sondern auch Informationen über Pfade maximaler Abarbeitungszeit, den Beitrag von einzelnen Codestücken zur Abarbeitungszeit, sowie Aussagen darüber, wie oft die einzelnen Teile des Codes zum ermittelten Worst Case beitragen.

Kapitel 6

Verwendung der Theorie zur MAXT-Berechnung

Bisher wurde die Berechnung der maximalen Abarbeitungszeit auf erweiterten T-Graphen als Programmierungsproblem beschrieben. Diese Repräsentation eignet sich zwar gut als Beschreibungsmodell, ist aber in der Praxis, wo ein konkretes Programm in einer Programmiersprache oder sonstigen Repräsentation vorliegt, nicht unmittelbar verwendbar. Dieses Kapitel soll daher die Beziehung zwischen der bisher behandelten Theorie und gängigeren Formen der Coderepräsentation herstellen.

Die Erzeugung eines T-Graphen wird für zwei Arten der Coderepräsentation beschrieben. Zunächst wird gezeigt, wie Assemblerprogramme abgebildet werden. Im Anschluß werden Regeln angegeben, wie Konstrukte höherer Programmiersprachen als T-Graphen bzw. Programmierungsproblem beschrieben werden können und wie auf diese Art die MAXT ermittelt werden kann.

Spezielle Beachtung wird den Restriktionen geschenkt, die vom Benutzer zur Beschreibung von Eigenschaften der erlaubten Programmpfade angegeben werden. Die Richtigkeit dieser Restriktionen kann nicht anhand der Programmstruktur überprüft werden. Um nun trotzdem sicherzustellen, daß Restriktionen zur Laufzeit nicht (unbemerkt) verletzt werden, wird vorgeschlagen, Code für Laufzeitüberprüfungen zu erzeugen. Es wird diskutiert, wie Restriktionen und deren Überprüfung in die Zeitanalyse einbezogen werden können.

6.1 Die MAXT-Analyse von Assemblerprogrammen

In diesem Abschnitt soll gezeigt werden, wie die maximale Abarbeitungszeit von Assemblerprogrammen mittels des oben vorgestellten Verfahrens berechnet werden kann.

Die Motivation für diesen Abschnitt liegt nicht darin, daß wir meinen, daß Assemblersprachen gut für die Echtzeitprogrammierung geeignet sind. Es geht vielmehr darum, zu zeigen, daß mit der in den letzten Kapiteln vorgestellten Methode für Codestücke ganz unterschiedlicher Repräsentationsformen, darunter auch Maschinensprache, die maximale Abarbeitungszeit relativ einfach ermittelt werden kann.

6.1.1 Annahmen

Assemblerprogramme bestehen aus einer Menge von *Basic Blocks*, das sind Instruktionfolgen maximaler Länge, die sequentiell durchlaufen werden. Jede Abarbeitung eines Basic Blocks beginnt mit seiner ersten Instruktion und endet mit seiner letzten Instruktion. Ein Hineinspringen in diesen Block, bzw. ein Herausspringen aus diesem Block ist nicht möglich. Am Ende eines Basic Blocks steht meist eine Sprunginstruktion (bedingt oder unbedingt). Mit Ausnahme des letzten Blocks hat jeder Block mindestens einen Nachfolger. Der oder die Nachfolger werden entweder implizit durch die Abfolge der Blöcke im Programm, oder explizit durch Sprungadressen definiert.

Für die Analyse wird angenommen, daß ein Codestück genau eine der folgenden zwei Bedingungen erfüllt:

- Es gibt einen ausgezeichneten Anfangsblock und einen ausgezeichneten letzten Block (Dabei beziehen sich "Anfang" und "Ende" nicht auf die sequentialisierte Anordnung der Blöcke in der Codedarstellung sondern auf die logische Abfolge der Blöcke). Für alle anderen Blöcke gilt, daß sie (transitiv) Nachfolger des Anfangs- und Vorgänger des Endblocks sind.
- Es gibt einen Anfangsblock und mehrere ausgezeichnete Endblöcke, deren letzte Instruktion jeweils eine *Return*-Instruktion (Return from Subroutine, Return from Exception, etc.) ist. Alle anderen Knoten sind (transitiv) Nachfolger des Anfangsblocks und Vorgänger eines Endblocks.

Diese Bedingungen sollen sicherstellen, daß nur Codeabschnitte analysiert werden, die einen eindeutigen Anfang und ein eindeutiges Ende haben und keinen unerreichbaren Code (Dead Code) beinhalten. Jedes Haupt- oder Unterprogramm erfüllt z.B. sofort die zweite Bedingung.

Über die Ausführungszeiten von einzelnen Instruktionen bzw. Folgen von Instruktionen werden folgende Annahmen gemacht:

- Die *maximalen Abarbeitungszeiten der Instruktionen* sind bekannt. Im Befehlssatz einiger Prozessoren gibt es Instruktionen, deren Ausführungszeit von den Operanden abhängt. Kann aufgrund der Operanden die exakte Abarbeitungszeit bzw. eine knappe Schranke für diese Zeit ermittelt werden, schrankt man die Abarbeitungszeit der Instruktion durch diesen Wert ab. Anderenfalls nimmt man den Maximalwert.

- Die *Kontrollkosten für einen bedingten Sprung* hängen oft davon ab, ob der Sprung durchgeführt wird oder nicht. Dieser Unterschied in den Ausführungszeiten der Befehle kann bei der MAXT-Analyse berücksichtigt werden.
- Die maximale Abarbeitungszeit einer Folge von Instruktionen ist gleich der Summe der maximalen Abarbeitungszeiten der einzelnen Instruktionen (*Additivität*).
- Die *maximalen Ausführungszeiten von Unterprogrammen*, die vom zu analysierenden Programmstück aufgerufen werden, sind bekannt.
- Für alle im Codestück enthaltenen Schleifen existieren Restriktionen, die die *Anzahl der Iterationen* der Schleife abschranken.

Sind diese Bedingungen erfüllt, kann die MAXT durch Lösung eines entsprechenden Programmierungsproblems abgeschrankt werden. Um knappe Schranken bzw. die exakte MAXT zu berechnen, sind gegebenenfalls noch weitere Restriktionen anzugeben.

6.1.2 Umwandlung in ein Programmierungsproblem

Durch die Anwendung eines einfachen Regelschemas kann aus einem Assemblerprogramm ein erweiterter T-Graph mit Restriktionen konstruiert werden. Jener wiederum kann dann als Progammierungsproblem geschrieben, die maximale Abarbeitungszeit durch Lösung des Programmierungsproblems ermittelt werden.

- Ein Basic Block, der genau einen Nachfolger hat, wird durch eine Kante repräsentiert, deren Abarbeitungszeit die Abarbeitungszeit der Instruktionen des Blocks ist. Die Kante wird in den T-Graphen so eingefügt, daß Vorgänger- und Nachfolgekante die dem unmittelbarem Vorgänger- bzw. Nachfolgeblock zugeordneten Kanten sind. Ist eine dieser Kanten noch nicht im Graphen eingetragen, so bleibt das entsprechende Kantenende zunächst unverbunden.
- Ein Basic Block, der mehr als einen Nachfolger hat, wird in zwei Teile gegliedert, die nacheinander umgewandelt werden. Der erste Teil, der alle Instruktionen des Blocks bis auf die letzte Instruktion, die Sprunginstruktion, enthält, wird wie ein Basic Block, der genau einen Nachfolger hat, in den T-Graphen eingefügt. Der zweite Teil besteht nur aus dem Sprung. Die Sprunginstruktion wird durch eine Menge von Pseudoinstruktionen — je eine für jedes Spungziel bzw. die Nachfolgeinstruktion — ersetzt.

  Jeder Pseudoinstruktion wird die passende Ausführungszeit zugewiesen, dann wird sie wie im vorigen Punkt beschrieben in den Graphen aufgenommen.
- Der erste und der letzte Basic Block werden zunächst wie alle anderen Blöcke behandelt. Am Schluß werden ihre Kanten durch eine Rückkehrkante verbunden, um einen erweiterten T-Graphen zu erhalten.

- Die Restriktionen auf Basic Blocks werden in Restriktionen auf Kanten übersetzt.

Die angegebenen Regeln stellen nur eine Möglichkeit der Übersetzung eines Assemblerprogramms in einen erweiterten T-Graphen mit Restriktionen dar. Es sind auch andere Zuordnungen denkbar, solange diese den Kontrollfluß und das Zeitverhalten des Programmes richtig beschreiben. Es ist z.B. möglich, einen einzelnen Knoten durch zwei Knoten, die mit einer Kante mit Ausführungszeit Null verbunden sind, zu ersetzen. In diesem Fall müssen nur die zum Knoten hinführenden bzw. vom Knoten wegführenden Kanten mit dem neuen Knoten am Anfang bzw. am Ende der Kante verbunden werden. Ein solches Vorgehen kann bei der automatischen Umwandlung von Assemblerprogrammen in T-Graphen hilfreich sein.

Ein Beispiel

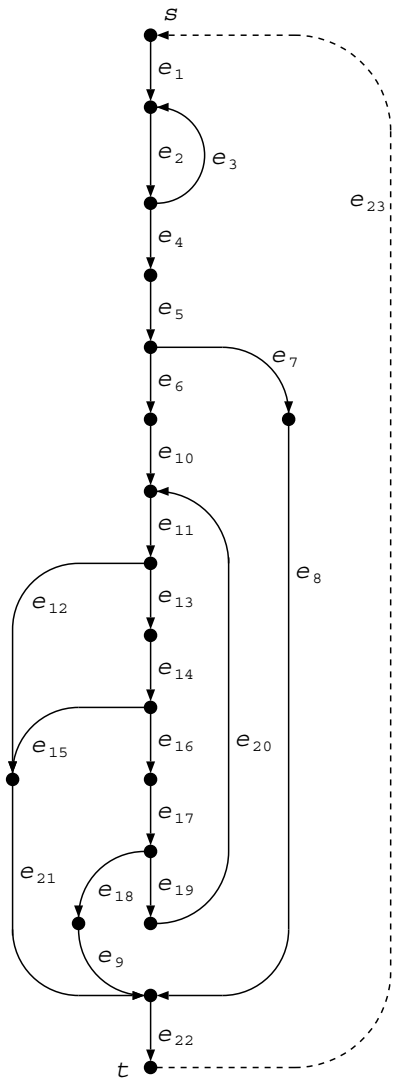
Abbildung 6.1 zeigt das Source Listing eines Assemblerprogramms und einen entsprechenden erweiterten T-Graphen mit Restriktionen. Neben dem Programmlisting sind die maximalen Abareitungszeiten der Instruktionen bzw. die Kanten, die den Instruktionssequenzen oder Instruktionen zugeordnet werden, angeführt. Die Abarbeitungszeiten der Kanten sind die Summen der Abarbeitungszeiten der entsprechenden Instruktionen. Bedingte Sprunginstruktionen werden, wie oben beschrieben, in zwei Pseudoinstruktionen mit den entsprechenden Abarbeitungszeiten aufgespalten. Jeder Pseudoinstruktion entspricht eine eigene Kante.

Das Listing enthält zwei Schleifen. Eine beginnt bei Label `L2`, die andere bei Label `L6`. Die Iterationszahlen dieser Schleifen müssen nach oben abgeschrankt werden. Es soll hier nicht festgelegt werden, wie diese Schranken zur Verfügung gestellt werden. Das kann zum Beispiel durch eine Annotation des Programms wie in [Mok89] oder durch eine interaktive Eingabe des Benutzers bei der Ausführung des Graphengenerierungsprogramms erfolgen. Unter der Annahme, daß die erste Schleife maximal 15 Mal und die zweite Schleife maximal zehn Mal durchlaufen wird, erhält man den rechts abgebildeten erweiterten T-Graphen und die dazugehörenden Restriktionen.

6.2 Die MAXT-Analyse von Sprachkonstrukten der strukturierten Programmierung

In diesem Teil soll gezeigt werden, wie Programmstücke, die durch Konstrukte der strukturierten Programmierung und entsprechende grundlegende Informationen über das Zeitverhalten und Restriktionen bezüglich der erlaubten Programmpfade charakterisiert sind, durch Umformung in ein Programmierungsproblem auf ihr Zeitverhalten hin untersucht werden können. Die hier gezeigte Vorgehensweise ist an keine spezielle Programmiersprache oder Repräsentationsform gebunden. Eine Beschreibung

Assembler Code	*maxt*	*Kante*
`_getop:`		
`link a6, #0`	16	
`moveml #0x3020, sp@-`	32	e_1
`movel a6@(8), a2`	16	
`movel a6@(12), d3`	16	
`L2:`		
`jsr _getchar`	220	e_2
`moveq #32, d1`	4	
`cmpl d0, d1`	6	
`beq L2`	10 8	e_3 e_4
`movel d0, a1`	4	
`lea a1@(-48), a0`	8	e_5
`moveq #9, d1`	4	
`cmpl a0, d1`	6	
`bcc L4`	10 8	e_6 e_7
`bra L11`	10	e_8
`L12:`		
`moveq #2, d0`	4	e_9
`bra L11`	10	
`L4:`		
`moveb d0, a2@`	8	e_{10}
`moveq #1, d2`	4	
`L6:`		
`jsr _getchar`	220	e_{11}
`moveq #47, d1`	4	
`cmpl d0, d1`	6	
`bge L7`	10 8	e_{12} e_{13}
`moveq #57, d1`	4	e_{14}
`cmpl d0, d1`	6	
`blt L7`	10 8	e_{15} e_{16}
`cmpl d2, d3`	6	e_{17}
`ble L12`	10 8	e_{18} e_{19}
`moveb d0, a2@(d2:l)`	14	
`addql #1, d2`	8	e_{20}
`bra L6`	10	
`L7:`		
`moveq #1, d0`	4	e_{21}
`L11:`		
`moveml a6@(-12), #0x40c`	40	e_{22}
`unlk a6`	12	
`rts`	16	



$$\begin{aligned} f_2 &\leq 15 f_1 \\ f_{11} &\leq 10 f_{10} \\ f_{23} &= 1 \end{aligned}$$

Abbildung 6.1: Ein Assemblerprogramm mit entsprechendem erweiterten T-Graphen

der Programmstruktur, Zeitinformationen und Restriktionen, wie sie hier verwendet wird, kann auf einfache Weise aus einer Vielzahl von Repräsentationen (z.B. den in [Pus89, Sha89, Par92, Pos92] verwendeten) gewonnen werden. Es ist auch möglich, den zum Programmstück gehörigen Flußgraphen zusammen mit Zeitinformationen über die Ausführungszeiten von Instruktionssequenzen als Ausgangsdarstellung zu nehmen. Ei-

ne solche Vorgehensweise wird in [Mok89] beschrieben.

Auf die Details der möglichen Ausgangsdarstellungen soll hier nicht näher eingegangen werden. Wir nehmen in dieser Arbeit an, daß ein zu analysierendes Codestück in einer einfachen Beschreibungssprache vorliegt. Mit dieser Beschreibungssprache werden die statische Kontrollstruktur des Codestückes, das Zeitverhalten und die Restriktionen rekursiv geschachtelt dargestellt. Die Grundelemente bilden einfache Strukturelemente für Statements, Sequenzen, Verzweigungen und Schleifen, sowie Konstrukte für Return, Break und Continue Statements. Zur Beschreibung von Zusammenhängen zwischen verschiedenen Programmteilen wird eine verallgemeinerte Form der in [Pus89] vorgestellten *Marker und Scopes* verwendet. Außerdem werden Scopes herangezogen, um Returns, Breaks und Continues einheitlich darzustellen.

Einfache Anweisungen, Sequenzen, Verzweigungen und Schleifen

Die einfachsten Einheiten, die zu beschreiben sind, bilden Codeteile, die streng sequentiell abgearbeitet werden. Diese Teile werden in der Beschreibungssprache durch ihre Abarbeitungszeit repräsentiert. Sequentiell aufeinanderfolgende Kontrollkonstrukte (einfache Anweisungen, Verzweigungen, Schleifen, etc.) werden unter Beibehaltung der Reihenfolge sequentiell abgebildet.

Verzweigungen und Schleifen erfordern eine umfassendere Beschreibung ihrer Kontrollsemantik sowie der einzelnen Konstituenten. Reservierte Schlüsselwörter werden daher zur Unterscheidung der Konstrukte bzw. zur Dokumentation der dargestellten Struktur- und Zeitinformation verwendet. Dies soll an dieser Stelle nur anhand eines Beispiels gezeigt werden (siehe Abbildung 6.2). Eine vollständige Beschreibung der Beschreibungssprache findet sich in Anhang B.

```
if
    condition 36
    oh_true   10
    oh_false   8
    then      40
              24
              86
    else      56
endif
```

Abbildung 6.2: Beispiel für die Darstellung eines *if*-Statements

Marker, Scopes und Restriktionen

Bisher bildeten erweiterte T-Graphen den Ausgangspunkt für die Zeitanalyse. Restriktionen zur Beschreibung von Zusammenhängen zwischen Pfaden bezogen sich jeweils

auf den gesamten zu analysierenden Graphen.

Will man das Zeitverhalten von Unterprogrammen oder Programmstücken untersuchen, so kann es vorkommen, daß der gesamte zu analysierende Bereich unübersichtlich groß ist. Für diesen Fall soll es möglich sein, Restriktionen anzugeben, die sich nur auf einen festgelegten Codeabschnitt und nicht auf den gesamten Code beziehen. Ein solcher Programmabschnitt kann mit dem Scope-Konstrukt gekennzeichnet werden.

Bereits in [Pus89] wurden Marker und Scopes dazu verwendet, um in Programmen Informationen über das dynamische Verhalten anzugeben. In diesem Ansatz konnte man allerdings nur Aussagen über die maximale Anzahl von Abarbeitungen einzelner Programmabschnitte innerhalb eines Scopes machen. Restriktionen, die Zusammenhänge zwischen mehreren Programmteilen beschreiben (d.h. die entsprechenden Restriktionen enthalten mehr als eine Variable), konnten nicht formuliert werden. Um nun allgemeinere Restriktionen für die Zeitanalyse verfügbar zu machen, modifizieren wir das Marker/Scope-Modell wie folgt:

- Ein *Scope* ist ein Sprachkonstrukt, das dazu dient, einen Teil eines Codestückes speziell zu kennzeichnen. Mit jedem Scope werden beliebig viele Restriktionen assoziiert. Diese Restriktionen beschreiben die Eigenschaften von Abarbeitungspfaden, die innerhalb des Scopes Gültigkeit haben.

- Ein *Marker* ist eine Stelle am Beginn oder in der Mitte einer Anweisungssequenz, die in einem Scope enthalten und mit einem symbolischen Namen gekennzeichnet ist.

- Die mit einem Scope assoziierten *Restriktionen* beschreiben die dynamischen Zusammenhänge von Abarbeitungspfaden innerhalb dieses Scopes durch Gleichungen bzw. Ungleichungen (siehe Kapitel 4). Die Anzahl der Abarbeitungen von bestimmten Punkten im Code werden in den Restriktionen durch den symbolischen Namen des entsprechenden Markers referenziert.

Abbildung 6.3 veranschaulicht die Verwendung des verallgemeinerten Marker/Scope-Modells anhand eines kurzen Codefragments. Die Programmteile, die die Marker *Marker1* und *Marker2* enthalten, werden zusammen innerhalb des Scopes nicht öfter als 25 Mal exekutiert.

Darstellung von Return, Break und Continue

Die drei Anweisungen, Return, Break und Continue, können als Sprünge zum Verlassen eines Programmabschnittes angesehen werden. Mit einem Return-Statement wird eine Prozedur verlassen. Dies kann als Sprung zum Ende der Prozedur interpretiert werden. Ein Break dient zum Verlassen einer umschließenden Schleife, ein Continue zum Verlassen des Rumpfes einer Schleife. Prozeduren, Schleifen bzw. Schleifenrümpfe werden

```
scope S1

    ... Statements ...

    Marker1

    ... Statements ...

            Marker2

    ... Statements ...


    ... Restriktionen ...

    Marker1 + Marker2 <= 25

endscope S1
```

Abbildung 6.3: Codefragment mit Markern und Restriktionen in einem Scope

daher als implizit definierte Scopes behandelt, die mit den entsprechenden Sprüngen an ihr Ende verlassen werden. Wir verwenden in unserer Beschreibungssprache die Namen *Procedure*, *Loop* und *LoopBody* für diese Scopes und die Anweisung *exit*, um die Sprunganweisungen zu realisieren. Die Anweisung *exit Procedure* entpricht einem Return, *exit Loop* einem Break und *exit LoopBody* einem Continue. Dabei beziehen sich die Scopenamen jeweils auf den innersten, die Sprunganweisung umgebenden Scope mit entsprechendem Namen.

6.2.1 Die Beschreibung der Transformation

Die in der Beschreibungssprache vorliegende Information über Struktur, Abarbeitungszeit und Restriktionen wird konstruktweise abgearbeitet und in das Programmierungsproblem für das gesamte Codestück eingebunden. Wie diese Transformation im Detail geschieht, soll hier gezeigt werden. Für jedes Konstrukt der Beschreibungssprache wird gezeigt, wie der entsprechende Teilgraph im T-Graphen aussieht und welchen Beitrag es zum Programmierungsproblem liefert. Dies geschieht in drei Teilen:

- Eine Ableitungsregel beschreibt die *Syntax* des Konstrukts. Diese Regel spiegelt die Kontrollstruktur sowie die mit dem Konstrukt verknüpfte Zeitinformation wieder. Es geht bei dieser Darstellung hauptsächlich darum, die anfallenden Zeitinformationen und den Kontrollfluß des Konstrukts zu veranschaulichen. Die syntaktische Struktur (Schlüsselwörter, Reihung der Komponenten, etc.) wurde mit Bedacht auf eine gute Lesbarkeit gewählt. In der Praxis ist sie allerdings von

untergeordneter Bedeutung, da angenommen werden kann, daß die Informationen ohnehin maschinell (eventuell ausschließlich durch interne Datenstrukturen repräsentiert) verarbeitet werden.

- Eine *graphische Darstellung* zeigt, wie der dem Konstrukt entsprechende Teilgraph eines T-Graphen aussieht. Dies illustriert den Einfluß des Konstrukts auf die statische Kontrollstruktur des T-Graphen.

- Eine Menge von Regeln gibt an, wie die Information über das Konstrukt zur *Generierung der Zielfunktion bzw. Restriktionen* des Programmierungsproblems herangezogen wird.

Die *syntaktische Beschreibung* der Konstrukte wird durch Produktionsregeln einer LR(1) Grammatik angegeben. Auf der rechten Seite dieser Regeln stehen kursive Zeichenketten (z.B. *Stmts'*) mit Ausnahme der t_X, wobei X für einen Kleinbuchstaben steht, für Nonterminale. Alle anderen Zeichenketten sind Terminalsymbole. Die Apostrophe am Ende der Namen von Nonterminalen gehören nicht zur syntaktischen Beschreibung. Sie dienen als Hilfsmittel bei der Beschreibung der Transformation, um bei den Regeln den Bezug zum entsprechenden Nonterminal herzustellen. Terminalsymbole im Teletype-Font (z.B. `else`) stehen für Zeichenketten, die unverändert lexikalische Einheiten der Beschreibungssprache bilden. Zeichenketten in Großbuchstaben stehen für Namen (IDENT) bzw. nicht negative ganze Zahlen (NUMBER). Die t_X, wobei X ein Kleinbuchstabe ist, repräsentieren nicht negative ganze Zahlen, die für eine Anzahl von Zeiteinheiten stehen.

Bei der *graphischen Veranschaulichung* der T-Graphen gelten folgende Konventionen: Durchgehende oder strichlierte gerichtete Kanten, mit e_X, X Kleinbuchstabe, benannt, repräsentieren sequentielle Codeteile, denen unmittelbar Abarbeitungszeiten zugeordnet werden können bzw. die Rückwärtskante des T-Graphen. Kanten mit Kreisen bilden weiter strukturierte Konstrukte ab, Kanten mit Rechtecken Sequenzen von strukturierten Konstrukten. Die Kantenmengen dieser Konstrukte werden mit E' bzw. E'' bezeichnet. Schließlich werden strichlierte Rechtecke, die Teilgraphen umschließen, verwendet, um Scopes darzustellen. Jeder Scope ist mit seinem Namen beschriftet.

Die Regeln zur Umwandlung der Codebeschreibung in ein lineares Programmierungsproblem geben für jedes Konstrukt an, wie die Information über Struktur und Zeitverhalten der einzelnen Konstituenten des Konstrukts zur Beschreibung des vorliegenden Konstrukts verwendet wird. Das Resultat bildet selbst ein Teilergebnis des gesamten Programmierungsproblems, das in der Folge wiederum entsprechend den Regeln in die Analyse des umschließenden Konstrukts einbezogen wird. Der Beitrag eines Konstrukts wird dabei als Sextupel $D = (G, C, E_{in}, E_{out}, X, M)$ geschrieben. Bei der Beschreibung, wie der Beitrag eines Konstrukts zum Programmierungsproblem ermittelt wird, benennen wir das Sextupel des aktuellen Konstrukts mit D, die Beiträge von den in dieses Konstrukt eingebetteten Teilkonstrukte mit $D' = (G', C', E'_{in}, E'_{out}, X', M')$

bzw. $D'' = (G'', C'', E''_{in}, E''_{out}, X'', M'')$. Die Komponenten eines Sextupels haben folgende Bedeutung:

G ist der vom Konstrukt gelieferte Beitrag zur Zielfunktion in der Form

$$G = \sum_{e_i \text{ im Konstrukt}} f_i t_i.$$

Die Summation läuft über alle Kanten, die im Konstrukt enthalten sind.[1] Die Variable f_i repräsentiert wieder den Fluß, die Konstante t_i die maximale Ausführungszeit der Kante.

C steht für die Menge der Restriktionen, die im Konstrukt gelten.

E_{in} ist die Menge aller Kanten, die vom Startknoten des Konstrukts wegführen.

E_{out} ist die Menge aller Kanten, die zum Endknoten des Konstrukts hinführen. E_{in} und E_{out} werden benötigt, um das aktuelle Konstrukt in die umliegenden Konstrukte einzubetten, d.h. im umschließenden Konstrukt die Flußgleichungen für den Anfangsknoten bzw. Endknoten generieren zu können.

X ist eine Menge von geordneten Paaren, (s_i, e_{i_j}). s_i ist der symbolische Name eines Scopes und e_{i_j} der Name einer Kante, die ans Ende dieses Scopes führt. Die Liste X wird verwendet, um die Kantennamen von Return-, Break- und Continue-Statements zu speichern, bis die Konstruktion des Subgraphs des zugehörigen Scopes abgeschlossen wird. Dann werden die Namen für diese Kanten in die Liste der Kanten, die zum Endknoten des Scopes führen, aufgenommen und in der Folge in die Restriktion zur Beschreibung des Flusses in diesem Knoten einbezogen.

M ist eine Menge von geordneten Paaren, (m_i, f_i), wobei m_i ein symbolischer Name für einen Marker und f_i die Variable ist, die den Fluß durch die dem Marker zugeordnete Kante beschreibt. Diese Zuordnung von symbolischen Namen zu Kanten wird benötigt, um Restriktionen, die sich auf symbolische Namen in der ursprünglichen Beschreibung der Programmteile beziehen, in entsprechende Restriktionen auf dem erweiterten T-Graphen umzuformen.

Zum Selektieren des Flusses, der durch die einem Marker zugeordnete Kante fließt, definieren wir den Selektionsoperator σ. Ist M eine Menge von geordneten Paaren $\{(m_i, f_i)\}$ und m_j der Name eines Markers, so gilt:

$$\sigma(M, m_j) = \begin{cases} f_j & (m_j, f_j) \in M \\ undef. & sonst. \end{cases}$$

[1] In der Beschreibung der Generierung eines Programmierungsproblems werden die Kanten mit symbolischen Konstanten indiziert. Bei einer Implementierung der Transformationsregeln ist darauf zu achten, daß ein Mechanismus zur Verfügung gestellt wird, der es erlaubt, eindeutige Kantennamen zu generieren.

6.2.2 Übersetzung der Konstrukte

In diesem Abschnitt werden die Regeln zur Übersetzung von Konstrukten der strukturierten Programmierung sowie von verallgemeinerten Markern und Scopes angegeben. Es wird davon ausgegangen, daß die Codeeinheiten für die MAXT-Analyse Prozeduren sind. (Es können auch Teile von Prozeduren analysiert werden, wenn das Eingabeformat der beschriebenen Syntax entspricht.) Die Startregel für die Beschreibung von Prozeduren lautet:

Procedure → `procedure` IDENT
Stmts'
Restr'
`end` IDENT

Die Startregel wird beim Lesen der Beschreibung als letztes reduziert. Die ihr entsprechenden Aktionen liefern das der Eingabe zugeordnete Programmierungsproblem. Das Sextupel des Nonterminals *Stmts'* wird um die Rückkehrkante e_r zum Anfangsknoten und die entsprechenden Flußrestriktionen erweitert. In der Gleichung für den Fluß im letzten Knoten werden alle Return-Statements (Kanten zum Ende des Scopes *Procedure*) berücksichtigt. Außerdem werden die Restriktionen, die sich auf das gesamte zu analysierende Programmstück beziehen, *Restr'* in das Programmierungsproblem übertragen. Dementsprechend sieht D wie folgt aus:

$$\begin{aligned} D \;=\; & (G', C' \cup \{f_r = \sum_{e_i \in E_{in}} f_i,\ \sum_{e_i \in E_{out}} f_i + \sum_{(Procedure, e_j) \in X'} f_j = f_r\} \cup \\ & \{\sum a_{ji}\sigma(M', m_i) \circ c_j : \sum a_{ji} m_i \circ c_j \in Restr'\} \cup \{f_r = 1\}, \emptyset, \emptyset, \emptyset, \emptyset). \end{aligned}$$

Der dazugehörige erweiterte T-Graph ist in Abbildung 6.4 zu sehen. Die Kante mit dem Rechteck steht für den Subgraphen, der *Stmts'* entspricht, die beiden Kanten an der linken Seite stellen Return Statements dar. Der strichlierte Kasten, der den Graphen umschließt, symbolisiert den *Procedure* Scope.

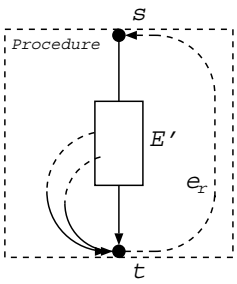


Abbildung 6.4: Erweiterter T-Graph für eine *Procedure*.

Statement-Sequenzen

Eine Statement-Sequenz besteht aus einer Folge von ein oder mehreren Statements. Die zugehörige Ableitungsregel lautet:

$$\begin{array}{lll} \text{Stmts} & \rightarrow & \mathit{Stmt'} \\ & | & \mathit{Stmt'}\ \mathit{Stmts''} \end{array}$$

Besteht die Sequenz nur aus einem Statement, so ist das Tupel für die gesamte Sequenz gleich dem dieses Statements. Tragen mehrere Statements zur Sequenz bei, so wird das neue Sextupel durch die Verknüpfung der Sextupel des ersten Statements und der restlichen Sequenz gebildet. Im ersten Fall gilt:

$$D = D'.$$

Im zweiten Fall wird das neue Sextupel wie folgt gebildet:

$$D \quad = \quad (G' + G'', C' \cup C'' \cup \{ \sum_{e_j \in E'_{out}} f_j = \sum_{e_k \in E''_{in}} f_k \}, E'_{in}, E''_{out}, X' \cup X'', M' \cup M'').$$

Die neue Zielfunktion ist die Summe der Zielfunktionen der beiden Teile der Sequenz. Die Menge der Restriktionen entsteht durch Vereinigung der beiden Restriktionsmengen und Hinzufügen einer zusätzlichen Restriktion, die den Fluß in dem Knoten, an dem die beiden Teilgraphen verbunden sind, beschreibt. Die Kanten vom Startknoten der Sequenz sind die Anfangskanten von *Stmt'*. Die Kanten zum Endknoten werden von *Stmts"* übernommen. Die Mengen der Paare mit den Exitkanten bzw. Markern entstehen durch Vereinigung der der entsprechenden Mengen von D' und D''. Abbildung 6.5 veranschaulicht die Bildung einer Sequenz. Die Kante mit dem Kreis repräsentiert dabei ein Statement.

Statements

$$\begin{array}{lll} \text{Stmt} & \rightarrow & \mathit{Simple'} \\ & | & \mathit{If'} \\ & | & \mathit{Loop'} \\ & | & \mathit{Exit'} \\ & | & \mathit{Scope'} \end{array}$$

Stmt leitet nach einem der fünf gezeigten Konstrukte ab. Der Beitrag zum Programmierungsproblem ist gleich dem Beitrag des jeweiligen Konstrukts auf der rechten

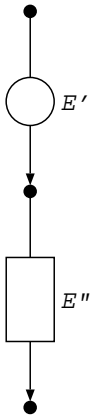


Abbildung 6.5: Graphische Darstellung einer Statement-Sequenz

Seite:

$$D = D'.$$

Einfache Statements

Einfache Statements bilden die einfachsten Bausteine der Analyse. Sie entsprechen einer Sequenz von Statements im zu analysierten Programm, der eine maximale Abarbeitungszeit, t_i, zugeordnet werden kann, d.h. sie umfassen in der Regel mehr als ein Statement der verwendeten Ausgangsdarstellung des untersuchten Programmstücks.

$$\text{Simple} \quad \rightarrow t_i$$

Ein einfaches Statement entspricht einer einzelnen Kante in der graphischen Darstellung (siehe Abbildung 6.6).

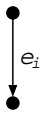


Abbildung 6.6: Graphische Darstellung eines einfachen Statements

Ein einfaches Statement liefert einen einzelnen Term zur Zielfunktion. Die Menge der vom Anfangsknoten wegführenden bzw. zum Endknoten hinführenden Kanten besteht jeweils nur aus der einzelnen Kante, die für das Statement steht. Die anderen Komponenten des Sextupels bleiben leer.

$$D = (f_i t_i, \emptyset, \{e_i\}, \{e_i\}, \emptyset, \emptyset).$$

Verzweigungen

In modernen Programmiersprachen, z.B. Modula-2, gibt es verschiedenartige Verzweigungskonstrukte. Das meist verbreitete Konstrukt hat die Semantik eines *if-then-else-endif*, wobei die *else*-Klause optional ist. Es gibt aber auch Konstrukterweiterungen, die ein *elsif* erlauben, bzw. Mehrfachverzweigungen, wie z.B. das *case*-Statement.

In dieser Arbeit werden nur Verzweigungen in Form von If-Statements, die sowohl mit, als auch ohne *else*-Klause geschrieben werden können, behandelt. Alle anderen Verzweigungskonstrukte können durch geeignete Kombination meherer If-Statements beschrieben werden.

If → `if`
`condition` t_c
`oh_true` t_t
`oh_false` t_f
`then` *Marker′ Stmts′*
Else
`endif`

Else → ϵ
| `else` *Marker″ Stmts″*

Marker → ϵ
| IDENT

Der Wert t_c steht für die Zeit, die zum Auswerten der Bedingung gebraucht wird. t_t bzw. t_f stehen für die Kontrollkosten für das Verzweigen im Konstrukt, die zumindest die Zeiten für das Verzweigen zum Beginn des *then*- bzw. *else*-Zweiges beinhalten. Die Zeiten für Sprünge an das Ende des Kontrukts können entweder den Kontrollkosten oder dem *then*- bzw. *else*-Zweig selbst zugerechnet werden. Hier werden die Kontrollkosten so verstanden, daß t_t die Zeit für das Verzweigen zur *then*-Klause, t_f die Zeit für das Verzweigen zur *else*-Klause bzw. an das Ende des Konstrukts ist. Eine graphische Interpretation ist in Abbildung 6.7 zu sehen.

Then- und *else*-Zweig bestehen aus einem optionalen Marker und einer Sequenz von Statements. Mit den Markern wird die Anzahl der Ausführungen des gesamten *then*- respektive *else*-Zweiges in Restriktionen, die innerhalb eines Scopes gelten (siehe Ende des Abschnitts), beschrieben.

$\overline{M}$ sei die Menge der Markerzuordnungen, die durch die Verzweigung definiert sind, d.h. die Teilmenge von $\{(\text{IDENT}', f_t), (\text{IDENT}'', f_f)\}$, die nur die Paare enthält, für die ein Marker existiert. Dann ist das Resultat eines *if-then-else*:

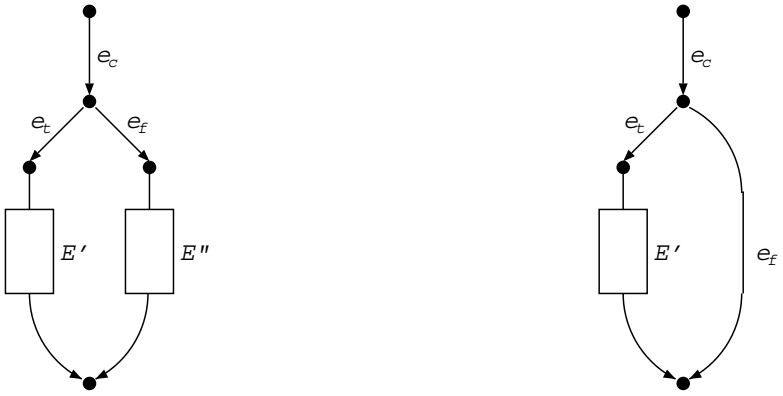


Abbildung 6.7: Graphische Darstellung eines If-Statements mit bzw. ohne Else-Zweig

$$\begin{aligned} D \;=\; & (G' + G'' + f_c t_c + f_t t_t + f_f t_f, C' \cup C'' \cup \{f_c = f_t + f_f,\ f_t = \sum_{e_j \in E'_{in}} f_j, \\ & f_f = \sum_{e_j \in E''_{in}} f_j\}, \{e_c\}, E'_{out} \cup E''_{out}, X' \cup X'', M' \cup M'' \cup \overline{M}). \end{aligned}$$

Gibt es keinen *else*-Zweig lautet das Ergebnis:

$$\begin{aligned} D \;=\; & (G' + f_c t_c + f_t t_t + f_f t_f, C' \cup \{f_c = f_t + f_f,\ f_t = \sum_{e_j \in E'_{in}} f_j\}, \\ & \{e_c\}, E'_{out} \cup \{e_f\}, X', M' \cup \overline{M}). \end{aligned}$$

Schleifen

Ähnlich wie bei Verzweigungen gibt es auch bei den Schleifen unterschiedliche Konstrukte (*for*, *while*, *repeat*, etc.). Wir wollen hier die Übersetzung eines einfachen Schleifenkonstrukts, das von der Struktur her einer Repeat-Schleife entspricht, illustrieren. Dieses Konstrukt enthält die wesentlichen Elemente, die eine Schleife ausmachen, sodaß auch andere Schleifenkonstrukte, wie z.B. While-Schleifen darauf aufbauend gebildet werden können.

```
Loop      → loop
              maxcount NUMBER
              body Marker' Stmts'
              condition t_c
              oh_back t_b
              oh_exit t_e
            endloop
```

Die wesentlichen Elemente einer Schleife bilden der Rumpf (*body*), die Bedingung (*condition*), sowie ein bedingter Sprung zum Beginn des Rumpfes der durch die Pseudoinstruktionen e_b und e_e repräsentiert wird. Jede Schleife ist mit einem *maxcount* versehen, der angibt, wie oft der Rumpf der Schleife bei jeder Ausführung maximal abgearbeitet werden kann.[2] Der *maxcount* geht in einer Restriktion in das Programmierungsproblem ein.

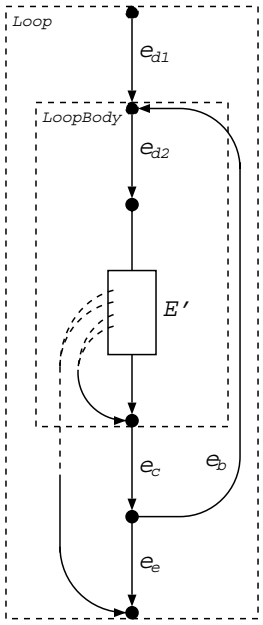


Abbildung 6.8: Graphische Darstellung einer Schleife

Abbildung 6.8 veranschaulicht, wie Schleifen in einen T-Graphen umgewandelt werden. Damit selbst geschachtelte Schleifen in erweiterten T-Graphen einfach beschrieben werden können, wird am Anfang von Schleifen eine Dummykante, e_{d1}, hinzugefügt. Die zweite Hilfskante, e_{d2}, wird verwendet, um eine Bezugskante für den Marker und die

[2] Dieses Wissen könnte ebensogut mit Hilfe des Markers im Schleifenrumpf und einer Restriktion beschrieben werden. Da die Angabe einer maximalen Iterationsanzahl aber eine zentrale Rolle für die Abschrankbarkeit der Abarbeitungszeit eines Programmteils spielt, wurde diese speziell ins Schleifenkonstrukt aufgenommen.

Iterationsschranke zu haben. Neben den beschriebenen Kanten und dem Rumpf der Schleife sind die beiden Default-Scopes *Loop* und *LoopBody* eingezeichnet (strichlierte Rechtecke). Exit-Statements im Rumpf der Schleife werden als Sprünge zum Ende des *Loop*-Scopes, Continue-Statements als Sprünge zum Ende des *LoopBody*-Scopes behandelt (Kanten an der linken Seite des Graphen). Eine Schleife wird wie folgt als Sextupel geschrieben:

$$
\begin{aligned}
D \;=\; & (G' + f_c t_c + f_b t_b + f_e t_e, C' \cup \{ f_{d2} \leq \text{NUMBER}' f_{d1},\ f_{d1} + f_b = f_{d2}, \\
& f_{d2} = \sum_{e_j \in E'_{in}} f_j,\ \sum_{e_j \in E'_{out}} f_j + \sum_{(LoopBody, e_k) \in X'} f_k = f_c,\ f_c = f_b + f_e \}, \{e_{d1}\}, \\
& \{e_e\} \cup \{e_j : (Loop, e_j) \in X'\}, X' \setminus \{(s,e) : s = Loop \text{ oder } s = LoopBody\}, \\
& M' \cup \overline{M}).
\end{aligned}
$$

Die Flüsse der beiden Dummykanten werden nicht in die Zielfunktion aufgenommen, da die Ausführungzeit dieser Kanten ohnehin gleich Null ist. Die Restriktionen für eine Schleife werden wie folgt aus zwei Teilen zusammengesetzt: Restriktionen, die die Beziehung zwischen Flüssen auf Kanten im Rumpf der Schleife beschreiben, werden unverändert aus der Beschreibung des Rumpfes übernommen. Weiters werden jene Gleichungen und Ungleichungen hinzugefügt, die die Flüsse der in der Schleife neu hinzukommenden Kanten und der Kanten, die einem *Continue*-Statement entsprechen, in die Menge der Flußrestriktionen einbinden.

Die Kante e_{d1} bildet die Anfangskante in der Schleifenbeschreibung, e_e und sämtliche Kanten, die aus *Break*-Statements generiert wurden, die Menge der Kanten, die ans Ende des Konstrukts führen. Die Menge X, in der die noch "offenen" Sprünge an die Enden von Scopes vorgemerkt werden, wird aus X' erzeugt, indem alle Paare, die die Schleife oder deren Rumpf referenzieren, entfernt werden.

Für $\overline{M}$ gilt: Ist der Schleifenrumpf mit einem Marker versehen, so ist $\overline{M} = (\text{IDENT}', f_{d_2})$, anderenfalls $\overline{M} = \emptyset$.

Return, Break und Continue

Wie bereits oben erwähnt, werden Return-, Break,- und Continue-Statements als Sprünge an das Ende der Scopes *Procedure*, *Loop* bzw. *LoopBody* interpretiert. Sie werden in einer einzigen Syntaxregel zusammengefaßt.

Exit → `exit` IDENT

Ident bezeichnet den Default Scope, an dessen Ende gesprungen werden soll. Theoretisch wäre es möglich, das Exit-Statement auf beliebige, auch selbst definierte Scopes

zu erweitern. Wir wollen uns hier allerdings auf existierende Konstrukte der strukturierten Programmierung beschränken.

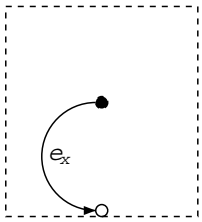


Abbildung 6.9: Graphische Darstellung eines Exit-Statements

Eine Exit-Kante ist nicht weiter strukturiert. Die Ausführungszeit der Kante e_x kann als die Zeit, die zur Ausführung der Verzweigung gebraucht wird, gesehen werden. t_x kann aber auch Null gesetzt werden, wenn diese Zeit bereits bei den vorhergehenden Statements berücksichtigt wurde. Das Wesentliche am Exit Statement ist der Sprung ans Ende eines Scopes.

Die Übersetzung eines Exit Statements ist der eines einfachen Statements sehr ähnlich. Im Unterschied zum einfachen Statement ist aber die Menge der zum Ende des letzten Knoten des Statements führenden Kanten leer. Die Kante wird stattdessen im Tupel (IDENT, e_x) in der Menge X "vorgemerkt" und wird später bei der Übersetzung des entsprechenden Scopes in den Flußrestriktionen berücksichtigt.

$$D = (f_x t_x, \emptyset, \{e_x\}, \emptyset, \{(\text{IDENT}, e_x)\}, \emptyset).$$

Scopes und Restriktionen mit Markern

Die Aufgabe eines Scopes ist es, Statements zu gruppieren, um die Restriktionen, die auf diesen Statement gelten, zu beschreiben.

Scope → `scope` IDENT
Stmts′
Restr′
`endscope` IDENT

Ein Scope ist selbst kein Kontrollkonstrukt. Er liefert als Subgraphen eine Dummykante gefolgt von dem von *Stmts′* repräsentierten Graphen.

Die Dummykante dient ähnlich wie bei den Schleifen dazu, die Beschränkungen, die durch die Restriktionen des Scopes angegeben werden, zur Anzahl der Aktivierungen des Scopes, d.h. zum Fluß in den Scope, in Beziehung zu setzen. Entsprechend seiner Semantik, liefert der Scope ein Sextupel, das wie folgt aussieht (es wird angenommen, daß die Restriktionen in Normalform vorliegen):

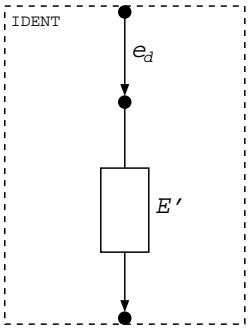


Abbildung 6.10: Graphische Darstellung eines Scopes

$$D = (G', C' \cup \{f_d = \sum_{e_j \in E'_{in}} f_j\} \cup \{\sum a_{ji}\sigma(M', m_i) \circ c_j f_d : \sum a_{ji} m_i \circ c_j \in Restr'\}, \{e_d\}, E'_{out}, X', M').$$

6.2.3 Beispiel für die Übersetzung einer Prozedur

Die folgenden Abbildungen illustrieren die Transformation der Beschreibung des Zeitverhaltens einer Prozedur in ein Programmierungsproblem. Abbildung 6.11 zeigt das Modula/R Listing für eine Implementierung des Bubble Sort Algorithmus, mit dem 7 Elemente sortiert werden können.

Die Struktur und die Basisinformationen über das Zeitverhalten der Prozedur sind in Abbildung 6.12 gezeigt. Diese Informationen entsprechen in etwa denen, die in einem Timing Tree [Pos92] repräsentiert sind. Um die Zeitinformationen für die einzelnen Programmteile zu bekommen, muß die Prozedur zuerst übersetzt werden. Dann können die Zeiten für die einzelnen Statements extrahiert und in die Struktur eingetragen werden.

Die so erhaltene Beschreibung des Zeitverhaltens der Bubble Sort Implementierung wird nach den oben beschriebenen Regeln in ein ganzzahliges Programmierungsproblem umgewandelt. Das Ergebnis sowie die Lösung dieses Programmierungsproblems sind in Abbildung 6.13 zu sehen. Das Programmierungsproblem ist in der Tupeldarstellung angegeben. Die Zielfunktion bildet die erste Komponente, die Menge der Restriktionen die zweite. Die restlichen Elemente werden für das endgültige Resultat der Umwandlung nicht benötigt, sie sind daher leere Mengen.

Die einzelnen Schritte der Umwandlung sind vollständig in Appendix A dokumentiert. In diesem Anhang ist auch der dem Programmierungsproblem zugeordnete erweiterte T-Graph abgebildet.

```
const N = 7;

procedure bubble_sort (var a: array[0..N] of integer);

var i, j: integer;
    tmp:  integer;

begin

    i := 1;
    scope S
        repeat max N-1 times
            j := N-1;
            if i <= j
            then
                repeat max N-1 times
                    in S max N*(N-1) div 2 times
                    if a[j-1] > a[j]
                    then
                        tmp := a[j-1];
                        a[j-1] := a[j];
                        a[j] := tmp;
                    endif;
                    j := j - 1;
                until j < i;
            endif
            i := i + 1;
        until i = N;
    endscope S;
end bubble_sort;
```

Abbildung 6.11: Modula/R Listing einer Bubble Sort Implementierung

6.2.4 Rückführung von Zeitinformation in die Ausgangsrepräsentation

Bei der Generierung des Programmierungsproblems für Programmstücke werden den einzelnen Konstrukten Teilgraphen des T-Graphen bzw. die Teile des entsprechenden Programmierungsproblems zugeordnet (Abschnitt 6.2.2). Diese Beziehung zwischen Ausgangsdarstellung und Programmierungsproblem kann genutzt werden, um für die gefundene Lösung der Programmierungsaufgabe den Beitrag jedes einzelnen Konstrukts zur maximalen Abarbeitungszeit zu bestimmen. Einerseits kann man aus der Belegung der Variablen unmittelbar ablesen, wie oft ein Konstrukt bzw. dessen Komponenten bei einer Worst Case Abarbeitung ausgeführt werden. Die Zeit, die das untersuchte Programmstück im Worst Case mit der Abarbeitung eines bestimmten Konstrukts zubringt, erhält man, indem man die Lösung der Programmierungsaufgabe in den Teil der Zielfunktion, der aus der Beschreibung genau dieses Konstrukts

```
procedure bubble_sort
    68
    scope S
        loop
            maxcount 6
            body
                4
                if
                    condition  4
                    oh_true    8
                    oh_false  10
                    then
                        loop
                            maxcount 6
                            body
                                MarkerM1
                                if
                                    condition 56
                                    oh_true    8
                                    oh_false  10
                                    then      40
                                endif
                            condition  8
                            oh_back   10
                            oh_exit    8
                        endloop
                endif
            condition 12
            oh_back   10
            oh_exit    8
        endloop
        MarkerM1 <= 21
    endscope S
    68
end bubble_sort
```

Abbildung 6.12: Beschreibung der Struktur und des Zeitverhaltens der Bubble Sort Implementierung

hervorging, einsetzt und die so erhaltenen Summanden addiert.

Die Informationsrückführung wird an Hand des Beispiels aus Abbildung 6.11 bis 6.13 veranschaulicht. Aus dem am tiefsten geschachtelten If-Statement geht die Teilsumme $40f_3 + 56f_4 + 8f_5 + 10f_6$ der Zielfunktion hervor (siehe auch Anhang A). Die Variable f_3 steht für die Anzahl der Abarbeitungen der Anweisungen im then-Zweig, f_4, f_5 und f_6 für die Anzahl der Exekutionen von Bedingung, Verzweigung zum then-Zweig und Verzweigung zum Ende des Konstrukts. Die Variablenbelegungen der Lösung, $f_3 = 21$, $f_4 = 21$, $f_5 = 21$ und $f_6 = 0$ zeigen, wie oft die einzelnen Teile auf den der Lösung entsprechenden Pfaden ausgeführt werden. Die Auswertung der Teilsumme

$(68f_1 + 4f_2 + 40f_3 + 56f_4 + 8f_5 + 10f_6 + 8f_7 + 10f_8 + 8f_9 + 4f_{12} + 8f_{13} + 10f_{14} +$
$12f_{15} + 10f_{16} + 8f_{17} + 68f_{21},$
$\{f_1 = f_{18},$
$f_4 = f_5 + f_6,$
$f_5 = f_3,$
$f_{11} \leq 6f_{10},$
$f_{10} + f_8 = f_{11},$
$f_{11} = f_4,$
$f_3 + f_6 = f_7,$
$f_7 = f_8 + f_9,$
$f_{12} = f_{13} + f_{14},$
$f_{13} = f_{10},$
$f_2 = f_{12},$
$f_{19} \leq 6f_{18},$
$f_{18} + f_{16} = f_{19},$
$f_{19} = f_2,$
$f_9 + f_{14} = f_{15},$
$f_{15} = f_{16} + f_{17},$
$f_{11} \leq 21f_{20},$
$f_{20} = f_{18},$
$f_{17} = f_{21},$
$f_{22} = f_1,$
$f_{21} = f_{22},$
$f_{22} = 1\},$
$\emptyset, \emptyset, \emptyset, \emptyset)$

$$Z = 2920.$$

$f_1 = 1, \quad f_2 = 6, \quad f_3 = 21, \quad f_4 = 21, \quad f_5 = 21, \quad f_6 = 0, \quad f_7 = 21,$
$f_8 = 17, \quad f_9 = 4, \quad f_{10} = 4, \quad f_{11} = 21, \quad f_{12} = 6, \quad f_{13} = 4, \quad f_{14} = 2,$
$f_{15} = 6, \quad f_{16} = 5, \quad f_{17} = 1, \quad f_{18} = 1, \quad f_{19} = 6, \quad f_{20} = 1, \quad f_{21} = 1,$
$f_{22} = 1.$

Abbildung 6.13: Ganzzahliges Programmierungsproblem für die Bubble Sort Prozedur und dessen optimale Lösung

ergibt, daß bei der Worst Case Abarbeitung 2184 von den 2920 Zeiteinheiten mit der Verarbeitung des If-Statements zugebracht werden.

6.3 Berücksichtigung der Laufzeitüberprüfungen von Restriktionen

Bisher ging es darum, die Struktur von Programmen und Wissen über ihr Verhalten so zu beschreiben, daß knappe Schranken für die maximale Abarbeitungszeit berechnet werden können. Restriktionen wurden dazu verwendet, Informationen über tatsächlich auftretende Abarbeitungspfade in die Zeitanalyse einzubringen. Dies geschah bis jetzt rein deskriptiv, d.h. die Restriktionen selbst beeinflußten die Abarbeitungszeit der untersuchten Codestücke nicht. Sie wirkten sich ausschließlich auf das Ergebnis der Zeitanalyse aus.

Führt man Restriktionen ein, um das dynamische Verhalten eines Codestückes zu beschreiben, so ist in der Praxis sicherzustellen, daß diese Restriktionen auch korrekt sind, d.h. keine falschen Sachverhalte beschreiben. Wir nehmen an, daß Restriktionen vom Designer bzw. vom Programmierer aus dem Wissen über die Applikation bzw. über das Verhalten eines Algorithmus' abgeleitet werden und daß es im allgemeinen Fall nicht möglich ist, diese Restriktionen automatisch aus dem Code zu generieren. Um nun sicher zu gehen, daß falsche Annahmen oder etwaige Fehler in Restriktionen nicht zu einem Fehlverhalten des Systems führen, werden alle Schranken von Schleifen und alle anderen Restriktionen zur Laufzeit auf ihre Einhaltung überprüft. Diese Überprüfungen erfordern zur Laufzeit das Setzen und Testen von Variablen. Diese Operationen benötigen selbst Zeit und müssen daher bei der Ermittlung der maximalen Abarbeitungszeit berücksichtigt werden:

- Für jede *Schleifenbeschränkung* wird eine interne Variable eingeführt. Die Variable wird bei jedem Eintritt in die Schleife mit der Anzahl der erlaubten Iterationen initialisiert und bei jeder Iteration auf die Einhaltung der Iterationsanzahl überprüft und dekrementiert. Sowohl Initialisierung als auch Überprüfung und Dekrement sind bei der Berechnung der maximalen Abarbeitungszeit einzurechnen.

- Für jede *Restriktion* der Form $\sum a_{ji} f_i \circ c_j,\ \circ \in \{<, \leq\}$ wird ebenfalls eine Variable reserviert. Diese Variable wird bei jedem Eintreten in den Scope, in dem die Restriktion gilt, mit dem Wert c_j initialisiert. Überall dort, wo sich Marker befinden, die in der Restriktion referenziert werden, wird Code zur Überprüfung der Restriktion und zur Änderung des Wertes der Variablen generiert. Die Variable wird bei jedem Marker um den Wert des Koeffizienten, mit dem der Marker in der Restriktion auftritt (a_{ji}), verringert.

Ein qualitativer Unterschied zwischen Schleifenbeschränkungen und Restriktionen besteht insofern, als für jede Schleife die Angabe der maximalen Anzahl von Iterationen Voraussetzung für die Berechenbarkeit einer Schranke für die Abarbeitungszeit ist. Die Angabe von weiteren Restriktionen ist hingegen für die Berechenbarkeit einer MAXT_C nicht unbedingt notwendig. Sie stellen lediglich einen Mechanismus dar, der es erlaubt, durch die Ausnutzung von zusätzlichem Wissen über das dynamische Verhalten eines Codestückes knappere Schranken für dessen Ausführungszeit zu berechnen. Dies ist für die folgenden Überlegungen von Bedeutung.

6.3.1 Knappere Schranke versus höhere Abarbeitungszeit

Restriktionen wurden eingeführt, um bei der Berechnung der maximalen Abarbeitungszeit bekannte Information über das dynamische Verhalten eines Programmstückes nutzen zu können. Sie werden während der Ausführung des Programmstückes laufend überprüft, um auf etwaige Verletzungen der angegebenen Bedingungen reagieren zu können. Man kann also folgendes feststellen:

- Auf der einen Seite tragen Restriktionen dazu bei, knappere Schranken für die maximalen Abarbeitungszeiten von Codestücken zu berechnen, d.h. das Resultat der MAXT-Analyse wird verbessert.
- Auf der anderen Seite konsumiert die Überprüfung der Restriktionen während der Programmausführung selbst Zeit. Diese zusätzlich anfallenden Zeiten führen wiederum zu einer Erhöhung der maximalen Abarbeitungszeit, was dem angestrebten Ziel einer geringen Abarbeitungszeit zuwider läuft.

Bei der Verwendung von Restriktionen zur Verbesserung des Resultats der MAXT-Analyse ist dieses Gegenspiel von Nutzen der Restriktionen und den Kosten, die durch die Überprüfung zur Laufzeit entstehen, zu berücksichtigen. Das Ziel wird es sein, Restriktionen nur soweit einzusetzen, als ihr Beitrag zur Verringerung der berechneten Zeitschranke größer ist als der Aufwand zu ihrer Überprüfung. Konkret kann dies durch die folgende Maßnahmen erreicht werden:

- Nutzung der Kontrollsemantik der vorhandenen Sprachkonstrukte: Schließen Teile eines Codestückes einander bei der Abarbeitung aus, so sollte dies soweit möglich durch die Kontrollstruktur des Codes explizit gemacht und nicht durch Restriktionen beschrieben werden (Beispiel: `if-then-else` statt `if-then-if-then`). Auf diese Art erübrigen sich die Überprüfungen zur Laufzeit.
- Restriktionen, deren Beitrag zur Verringerung der berechneten Schranke für die maximale Abarbeitungszeit gering ist, sind zu vermeiden.

- Restriktionen, die einander überdecken, d.h. den gleichen Sachverhalt ausdrücken oder einander implizieren, sind ebenfalls zu vermeiden.

Zum Abschluß dieses Kapitels soll noch ein Algorithmus angegeben werden, mit dem eine knappe und niedrige Schranke für die maximale Abarbeitungszeit eines Codestückes unter Berücksichtigung der Laufzeitüberprüfungen von Restriktionen berechnet werden kann. Der Algorithmus operiert auf dem zu untersuchenden Programmstück sowie einer Menge von Restriktionen, die zusätzlich angegeben werden. Folgende Schritte werden ausgeführt:

1. Berechne die maximale Abarbeitungszeit für das Programm, wobei nur Schleifengrenzen (Loop Bounds) und keine Restriktionen berücksichtigt werden.
2. Ermittle die Restriktion aus der Menge der Restriktionen, die die größte Verringerung der Schranke der Abarbeitungszeit verspricht.
3. Füge die Restriktion (sowie den Code für deren Überprüfung) zum Programmstück hinzu und berechne die neue Schranke für die Abarbeitungszeit.
4. Ergab Schritt 3 eine Verbesserung gegenüber des zuvor berechneten Wertes verfahre wie folgt:

   a. Gibt es noch weitere, nicht berücksichtigte Restriktionen, so entferne die zuletzt verwendete Restriktion aus dieser Liste und setze bei Punkt 2 fort.
   b. Wurden bereits alle Restriktionen berücksichtigt, so terminiere mit der zuletzt gefundenen Lösung.

   Brachte Schritt 3 keine Verbesserung so verwirf die letzte Restriktion und retourniere den zuvor ermittelten Wert als Lösung.

Die Schwierigkeit des Algorithmus liegt im Punkt 2, bei der Auswahl der Restriktion, die als nächste hinzugefügt wird. Durch die Betrachtung der bisher gültigen optimalen Lösung kann zwar der Beitrag (Abarbeitungszeit sowie Anzahl der Ausführungen) der einzelnen Programmteile zur entsprechenden maximalen Abarbeitungszeit bestimmt werden, es ist aber nicht sichergestellt, daß durch eine Beschränkung, die sich auf die Menge der Teilpfade mit dem größten Beitrag zur Abarbeitungszeit bezieht, auch die größte Reduktion der ermittelten Schranke erzielen läßt (siehe auch Abschnitt 5.3). Erst ausführliche Analysen der Auswirkungen möglicher Restriktionen auf die berechnete Zeitschranke machen eine sinnvolle Selektion der nächsten Restriktion möglich. Auf die algorithmischen Details dieser Lösungsfindung soll allerdings im Rahmen dieser Arbeit nicht näher eingegangen werden.

Eine quantitative Diskussion darüber, wie sich der Einsatz von Restriktionen auf die berechneten Schranken für die Abarbeitungszeit auswirkt, ist in Kapitel 7 zu finden.

6.4 Zusammenfassung

In diesem Teil der Arbeit wurde der Anwendbarkeit der zuvor entwickelten Theorie der Berechnung von Schranken der Abarbeitungszeit für die praktische Durchführung von MAXT-Analysen gezeigt. Codestücke, die im Assemblercode bzw. in Form einer höher strukturierten Beschreibung vorliegen, können mit den hier erklärten Mechanismen in erweiterte T-Graphen mit Restriktionen bzw. ganzzahlige Programmierungsprobleme transformiert werden und dann auf ihr Worst Case Zeitverhalten hin untersucht werden.

Um sicherzustellen, daß die von Schleifengrenzen und Restriktionen beschriebenen Bedingungen nicht verletzt werden, werden diese zur Laufzeit überprüft. Diese Überprüfungen benötigen selbst Zeit und müssen daher bei der Berechnung der Schranke für die maximale Abarbeitungszeit berücksichtigt werden. Ein Algorithmus zur MAXT-Analyse, der auch diese Laufzeitkosten einbezieht, wurde vorgestellt.

Kapitel 7

Experimente

In diesem Kapitel soll der beschriebene Ansatz zur MAXT-Berechnung anhand von Beispielen untersucht werden. Dies geschieht mit zwei Schwerpunkten. Einerseits soll festgestellt werden, wie sich Informationen über den Programmfluß auf die Qualität der berechneten Schranke für die maximalen Abarbeitungszeiten von Prozeduren auswirken bzw. wie knappe Schranken in der Praxis für deren Abarbeitungszeiten mit den vorgestellten Möglichkeiten berechnet werden können. Auf der anderen Seite wird untersucht, inwieweit die Überprüfungen von Restriktionen zur Laufzeit die maximale Abarbeitungszeit eines Unterprogrammes erhöhen. Die Erhöhung der MAXT wird zur gleichzeitig erreichten Verbesserung der MAXT_C in Bezug gestellt.

7.1 Vorgehen bei den Experimenten

Die Experimente wurden wie folgt durchgeführt: Zunächst wurden die zu untersuchenden Prozeduren in einer höheren Programmiersprache, C, implementiert und mit einem Compiler für den MC-68000 Prozessor übersetzt. Anschließend wurden die Programme in unterschiedlichem Maß mit Restriktionen zur Beschreibung von möglichen Programmpfaden versehen und Schranken für die maximale Abarbeitungszeit einerseits wie beschrieben analytisch, andererseits die MAXT selbst — soweit möglich — in Simulationsläufen mit Testdaten ermittelt. Diese Verfahren wurden sowohl mit, als auch ohne Berücksichtigung der Overheads für die Überprüfung von Schleifengrenzen und Restriktionen angewandt, um zu den Daten für beide Teile der Auswertung zu gelangen.

Die Untersuchung der Prozeduren in Hinblick auf die Restriktionen zur Beschreibung der möglichen Abarbeitungspfade und die Ermittlung der zugehörigen Zeitschranken erfolgten in zwei streng getrennten Phasen. Als erstes wurde die Semantik des Codes untersucht. Mit Restriktionen für die unmittelbar einsichtigen Auswirkungen der Semantik auf die Abarbeitungspfade beginnend, wurden die Scopes und Ungleichungen

für immer komplizierter werdende Szenarien festgehalten. In dieser Phase wurden noch keine Zeitberechnungen angestellt. Erst in einem zweiten Schritt wurden die Abarbeitungszeiten zu den gewählten Szenarien ermittelt. Dieses zweistufige Vorgehen wurde gewählt, um einen Eindruck davon zu bekommen, wie schwer bzw. leicht es ist, genaue Aussagen über das dynamische Verhalten eines Programmstückes zu machen.

Berechnung von Abarbeitungszeiten

Für die Berechnung der Schranken für die maximalen Abarbeitungszeiten wurde der Code der betrachteten Algorithmen im Source Code untersucht und für jedes Szenario mit den entsprechenden Scopes und Restriktionen versehen. Der Code wurde mit dem Compiler zu einem Assemblerprogramm übersetzt. Aus dem Assemblerprogramm und den aus dem Source Code übernommenen Restriktionen wurde dann ein erweiterter T-Graph aufgebaut und nach der Vorgangsweise aus Kapitel 6 das entsprechende ganzzahlige Optimierungsproblem formuliert. Die Lösung der Optimierungsaufgabe lieferte die Schranke für die MAXT.

Simulation zur Ermittlung von Abarbeitungszeiten

Als Gegenstück zu den analytisch ermittelten Schranken für die Abarbeitungszeiten der Programmstücke wurden Ausführungszeiten von Abarbeitungen mit unterschiedlichen Eingabedaten ermittelt. Um eine Konzentration auf die gestellten Zielsetzungen zu erlauben und Auswirkungen von Hardwareeigenheiten, die die Analyse der Ergebnisse erschweren, auszuschalten, wurden die Ausführungszeiten des Codes nicht durch Messungen auf einer Zielmaschine sondern durch Simulationen bestimmt. Interrupts, Verzögerungen der Programmausführung durch DMA bzw. Memory Refresh, usf. wurden auf diese Art unterbunden. Weiters wurde zur Vereinfachung angenommen, daß die Ausführungszeit jeder Instruktion unveränderlich ist. Für Instruktionen mit variabler Abarbeitungszeit (Multiplikation, Shift bzw. Rotation um eine nicht bekannte Anzahl von Bits, etc.) wurde bei jeder Ausführung die maximale Abarbeitungszeit angenommen. Das entspricht dem Wert, der auch bei der Berechnung der MAXT_C für diese Instruktionen herangezogen wird.

Eine wesentliche Rolle für das Resultat der Simulationen spielt die Wahl der Eingabedaten, denn sie bestimmen die Pfade der einzelnen Abarbeitungen. Im Rahmen der für diese Arbeit durchgeführten Experimente war es uns daher wichtig, die Daten so zu wählen, daß möglichst auch Abarbeitungen maximaler Laufzeit generiert wurden.

7.2 Beispiele und Daten

Für die Experimente wurden vier verschiedene Beispielprogramme gewählt und auf ihr Zeitverhalten hin untersucht. Die Programme unterscheiden sich durch ihre unterschiedliche statische Struktur und durch das Ausmaß, in dem Eigenheiten von Abarbeitungspfaden durch Restriktionen charakterisiert werden können. Die folgenden Prozeduren wurden für die Experimente gewählt:

- Matrixmultiplikation,
- Sortieren mittels Bubble Sort Algorithmus,
- Sortieren durch Mischen und
- die Prozedur *Calculate_Center*, die die Mittelpunktkoordinaten eines auf einem Videobild dargestellten Objektes berechnet.

Matrixmultiplikation

Die getestete Prozedur berechnet das Produkt zweier Matrizen mit jeweils fünf Zeilen und fünf Spalten. Sie besteht aus drei geschachtelten *for*-Schleifen, die jeweils mit konstanter Anzahl von Iterationen abgearbeitet werden und einigen Anweisungen zur Durchführung von Berechnungen (siehe Abbildung 7.1). Der Code enthält neben den Schleifen keine Verzweigungen und auch die Anzahl der Iterationen bei der Abarbeitung der Schleifen ist unabhängig von den Eingabedaten. Die Restriktionen beschränken sich daher auf die Beschreibung der maximalen Anzahl von Iterationen für die Schleifen, die jeweils gleich fünf ist (Szenario *a* in den Auswertungen).

Bubble Sort

Der Kontrollfluß der verwendeten Bubble Sort Prozedur (siehe Abbildung 7.2) wird durch die beiden Schleifen und eine Verzweigung bestimmt. Im Gegensatz zum vorigen Beispiel ist die Anzahl der Iterationen der inneren Schleife nicht mehr konstant. Sie ändert sich mit jeder Iteration der äußeren Schleife. Die Auswirkung dieser dynamischen Schleifengrenze auf die Anzahl der Abarbeitungen des Rumpfes der inneren Schleife läßt sich nicht mehr allein durch das Angeben einer maximalen Iterationsanzahl beschreiben. Es wurden daher weitere Restriktionen notwendig (siehe unten, Szenarien *b* und *c*).

Bei den Experimenten wurden die MAXTs für drei verschiedene Szenarien, die sich durch die angegebenen Restriktion unterscheiden, untersucht (siehe Tabelle 7.1). Sortiert wurde ein Feld mit sieben Elementen. Die Beschränkung auf ein kleines Feld erlaubte die Generierung aller Permutationen der zu sortierenden Elemente und somit

```
void    mmult(short a[N][N], short b[N][N], short c[N][N])
{
    short i, j, k, tmp;

    for (i=0; i<N; i++)                    /* bound 5 */
    {
        for (k=0; k<N; k++)                /* bound 5 */
        {
            tmp = 0;
            for (j=0; j<N; j++)            /* bound 5 */
            {
                tmp = tmp + a[i][j] * b[j][k];
            }
            c[i][k] = tmp;
        }
    }
}
```

Abbildung 7.1: Matrixmultiplikation

die Untersuchung *aller* möglichen Abarbeitungspfade. So konnte die MAXT gesichert experimentell ermittelt werden.

Szenario	*Schleifen*	*Restriktionen*
a	$B1 \leq 6,\ B2 \leq 6$	—
b	$B1 \leq 6,\ B2 \leq 6$	in $S1$: $M1 \leq 21$
c	$B1 \leq 6,\ B2 \leq 6$	in $S1$: $M2 \leq 21$

Tabelle 7.1: Untersuchte Szenarien für Bubble Sort

Sortieren durch Mischen

Die hier im Pseudocode vorgestellte iterative Implementierung von Sortieren durch Mischen hat unter den untersuchten Prozeduren die komplizierteste Kontrollstruktur (Abbildung 7.3).

Ein Feld von N Elementen wird in $ld\ N$ Schritten durch das wiederholte Mischen von Teilfolgenpaaren sortiert. Bei der Verarbeitung von Listenpaaren werden zunächst die Längen und Anfangsindizes der Listen bestimmt. Anschließend erfolgt das Mischen dieser Teillisten.

Für unsere Experimente wurde das Verhalten der Prozedur beim Mischen von sieben Elementen analysiert. Dabei wurde die Information über ihr dynamisches Verhalten nach und nach verfeinert, bzw. das Maß an Information, das der MAXT-Analyse zur

```
void bubble (int *a)
{
        int i, j, tmp;

        i = 1;
        /* begin scope S1 */
        do                      /* bound B1 */
        {
            j = N - 1;
            if (i <= j)
            {
                do              /* bound B2 */
                {
                    /* marker M1 */
                    if (a[j-1] > a[j])
                    {
                        /* marker M2 */
                        tmp = a[j-1];
                        a[j-1] = a[j];
                        a[j] = tmp;
                    }
                    j--;
                } while (j >= i);
            }
            i++;
        } while (i < N);
        /* end scope S1 */
}
```

Abbildung 7.2: Prozedur Bubble Sort

Verfügung gestellt werden sollte, verändert. Dies führt zu den in Tabelle 7.2 aufgelisteten Restriktionen für die Programmierungsaufgaben. Bei den Szenarien a bis e ist jedes Szenario eine Verfeinerung seines Vorgängers, d.h. es enthält mehr oder restriktivere Information. Beispiel f ist eine "Optimierung" von e im Hinblick auf die Kosten für die Laufzeitüberprüfung von Restriktionen: Die erste Ungleichung wurde weggelassen, sie wird ohnehin durch die letzten drei Ungleichungen subsumiert (siehe Pseudocode in Abbildung 7.3). Szenario g entstand aus f durch neuerliches Hinzufügen von Information, h aus g durch das Weglassen der Restriktionen für die Marker $M2$, $M3$ und $M4$, die in der in g neu hinzugefügten Restriktion gemeinsam beschrieben werden.

Wie beim Bubble Sort wurden bei der Simulation wiederum alle Permutationen der untersuchten Testdaten generiert. Auf diese Art konnte die MAXT experimentell ermittelt werden.

```
merge_sort(list1, list2)
{
        /* begin scope S1 */
        loop (ld N times)                                          /* B1 */
        {
            loop (for all list pairs)                              /* B2 */
            {
                /* M1 */
                determine the length of both lists, prepare for loops
                    -> both complete /* M5 */
                    -> first complete, second incomplete /* M6 */
                    -> second empty /* M7 */

                while (elements in list1 and list2)                /* B3 */
                    /* M2 */
                    if (first element of list1 <= first el. of list2)
                        copy first element of list1 to resulting list
                    else
                        copy first element of list2 to resulting list
                while (elements in list1)                          /* B4 */
                    /* M3 */
                    copy first element of list1 to resulting list
                while (elements in list2)                          /* B5 */
                    /* M4 */
                    copy first element of list2 to resulting list
            }
        }
        /* end scope S1 */
}
```

Abbildung 7.3: Sortieren durch Mischen

Prozedur "Calculate_Center"

Die Prozedur *Calculate_Center* berechnet den Mittelpunkt eines in Schwarz/Weiß-Pixelgraphik dargestellten Objektes. Sie wurde in [Pus89] als Beispiel für die Verwendung von Markern vorgestellt und ist nochmals in Abbildung 7.4 skizziert. Die Prozedur zeichnet sich dadurch aus, daß der Marker $M1$, der die Anzahl der maximal gesetzten Punkte angibt, eine sehr starke Einschränkung der Anzahl der Aufrufe von *Calculate_Weight*, von 128000 auf 3480, beschreibt.

Für die Berechnung der Schranken für die Abarbeitungszeit wurden zwei Fälle, einer ohne Berücksichtigung von Marker $M1$ (a) und einer mit Berücksichtigung des Markers (b), betrachtet (siehe Tabelle 7.3). Würde man auch prozedurübergreifende Restriktionen betrachten, könnte eine weitere Restriktion, die die maximale Summe über die Anzahl von Nachbarn gesetzter Punkte beschreibt (Marker $M2$ im Listing), in die Untersuchungen aufgenommen werden. Diesen Fall wollen wir allerdings in dieser

Szenario	*Schleifen*	*Restriktionen*
a	$B1 \leq 3$, $B2 \leq 4$, $B3 \leq 6$, $B4 \leq 4$, $B5 \leq 3$	—
b	$B1 \leq 3$, $B2 \leq 4$, $B3 \leq 6$, $B4 \leq 4$, $B5 \leq 3$	in $S1$: $M1 \leq 7$
c	$B1 \leq 3$, $B2 \leq 4$, $B3 \leq 6$, $B4 \leq 4$, $B5 \leq 3$	in $S1$: $M1 \leq 7$, $M2 \leq 21$, $M3 \leq 21$, $M4 \leq 21$
d	$B1 \leq 3$, $B2 \leq 4$, $B3 \leq 6$, $B4 \leq 4$, $B5 \leq 3$	in $S1$: $M1 \leq 7$, $M2 \leq 21$, $M3 \leq 21$, $M4 \leq 21$, $M5 \leq 4$, $M6 \leq 2$, $M7 \leq 1$
e	$B1 \leq 3$, $B2 \leq 4$, $B3 \leq 6$, $B4 \leq 4$, $B5 \leq 3$	in $S1$: $M1 \leq 7$, $M2 \leq 14$, $M3 \leq 12$, $M4 \leq 9$, $M5 \leq 4$, $M6 \leq 2$, $M7 \leq 1$
f	$B1 \leq 3$, $B2 \leq 4$, $B3 \leq 6$, $B4 \leq 4$, $B5 \leq 3$	in $S1$: $M2 \leq 14$, $M3 \leq 12$, $M4 \leq 9$, $M5 \leq 4$, $M6 \leq 2$, $M7 \leq 1$
g	$B1 \leq 3$, $B2 \leq 4$, $B3 \leq 6$, $B4 \leq 4$, $B5 \leq 3$	in $S1$: $M2 \leq 14$, $M3 \leq 12$, $M4 \leq 9$, $M5 \leq 4$, $M6 \leq 2$, $M7 \leq 1$, $M2 + M3 + M4 \leq 21$
h	$B1 \leq 3$, $B2 \leq 4$, $B3 \leq 6$, $B4 \leq 4$, $B5 \leq 3$	in $S1$: $M5 \leq 4$, $M6 \leq 2$, $M7 \leq 1$, $M2 + M3 + M4 \leq 21$

Tabelle 7.2: Untersuchte Szenarien für Sortieren durch Mischen

Arbeit nur kurz andiskutieren.

Szenario	*Schleifen*	*Restriktionen*
a	$B1 \leq 200,\ B2 \leq 640$	—
b	$B1 \leq 200,\ B2 \leq 640$	in $S1$: $M1 \leq 3480$

Tabelle 7.3: Untersuchte Szenarien für *Calculate_Center*

Als Referenzwert für die tatsächlichen Abarbeitungszeiten wurde nicht die MAXT herangezogen, weil ein entsprechendes Testbeispiel schwer zu erzeugen und das Auftreten dieses Falles sehr unwahrscheinlich ist[1]. Stattdessen wurde eine Szene mit einem runden Objekt und voller Anzahl von Störungspunkten, die jeweils isoliert, ohne gesetzte Nachbarn sind, generiert und simuliert. Die Differenz zwischen MAXT_C und experimentell ermitteltem Wert wird zwar dadurch vergrößert, das Resultat läßt sich aber dennoch gut für unsere Zwecke verwenden.

[1] Die MAXT würde sich durch ein Szenario ergeben, in dem die Punkte des Objektes und die maximale Anzahl von Störungspunkten einen Kreis mit 3480 gesetzten Pixel bilden. Damit wäre a) die maximale Anzahl von Pixel gesetzt und b) die Summe der Nachbarn aller Punkte wäre ebenfalls maximal.

```
int Calculate_Weight(image, x, y)
{
        for (i :=  y-1 ... y+1)
            for (j := x-1 ... x+1)
                if (pixel image[i,j] is set)
                    /* (M2) */
                    increment count

        return f(count)
}

Calculate_Center(image, x_center, y_center)
{
        count := 0;
        /* begin scope S1 */
        for (y := 1 ... number of lines) /* B1 */
        {
            for (x := 1 ... number of columns) /* B2 */
            {
                if (pixel image[y,x] is set)
                {
                    /* M1 */
                    wt := Calculate_Weight(image, x, y);
                    (x_sum, y_sum) := (x_sum + x * wt, y_sum + y *wt);
                    count := count + weight;
                }
            }
        }
        /* end scope S1 */

        if (count > 0)
            (x_center, y_center) := (x_sum / count, y_sum / count);
        else
            (x_center, y_center) := (0, 0);
}
```

Abbildung 7.4: Prozedur *Calculate_Center*

7.3 Ergebnisse der MAXT$_C$-Berechnungen

Im ersten Teil der Experimente ging es darum, die Verwendbarkeit des beschrieben Ansatzes für die Berechnung knapper MAXT-Schranken zu evaluieren. Die folgenden Fragen sollten dabei beantwortet werden:

- Wie knapp kommt man in der Praxis mit der Berechnung einer MAXT$_C$ an die MAXT eines Programmstückes heran?

- Wieviel Zusatzinformation über die Semantik bzw. den Progammfluß ist für die Berechnung einer knappen MAXT-Schranke erforderlich?
- Wie schwer ist es, Information über die Zusammenhänge innerhalb des Codes herauszufiltern bzw. zu beschreiben?

In dieser Testreihe wurden die MAXT_Cs bzw. MAXTs für alle oben beschriebenen Szenarien ermittelt. Es wurde jedoch angenommen, daß die Überprüfung der Einhaltung von Iterationsgrenzen und Restriktionen *keine* Zeit benötigt. Die Tabellen 7.4 bis 7.7 und Abbildungen 7.5 bis 7.8 zeigen die Ergebnisse, wobei in den ersten Zeilen der Tabellen als Referenzwert jeweils die MAXTs, bzw. bei der Prozedur *Calculate_Center* die Ausführungszeit des beschriebenen Referenzszenarios, angegeben sind. In den folgenden Zeilen der Tabellen enthalten die Spalten der Reihe nach den Namen des Szenarios, die für dieses Szenario berechnete MAXT_C und die Differenz zwischen MAXT_C und dem Referenzwert. In den zugehörigen Abbildungen sind die MAXT_Cs der Szenarien als Balken, der Referenzwert durch eine punktierte horizontale Linie dargestellt.

Szenario	MAXT_C	*Differenz*
Ref.	27 666	—
a	27 666	0

Tabelle 7.4: MAXT_C der Matrixmultiplikation in CPU-Zyklen

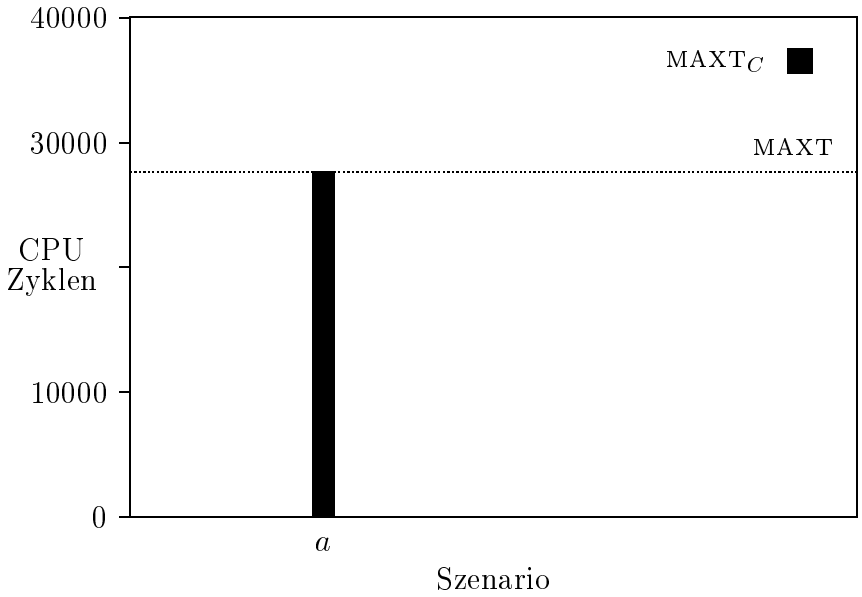


Abbildung 7.5: MAXT_C und MAXT der Matrixmultiplikation

Durch die Prozedur *mmult* gibt es genau einen Pfad, der unabhängig von den Daten bei allen Exekutionen durchlaufen wird. In unserer Simulation liefert dieser Pfad für alle Paare von Matrizen dieselbe Ausführungszeit, die notwendiger Weise die MAXT ist[2]. Alle Schleifen von *mmult* iterieren bei jeder Ausführung gleich oft. Das Wissen über die statische Struktur des Programms und die Iterationsgrenzen der Schleifen ist daher für die Berechnung einer MAXT_C, die gleich der MAXT ist, ausreichend (siehe Tabelle 7.4 und Abbildung 7.5).

Szenario	MAXT_C	*Differenz*
Ref.	2 912	—
a	4 742	1 830
b	2 920	8
c	4 172	1 260

Tabelle 7.5: MAXT_Cs von Bubble Sort in CPU-Zyklen

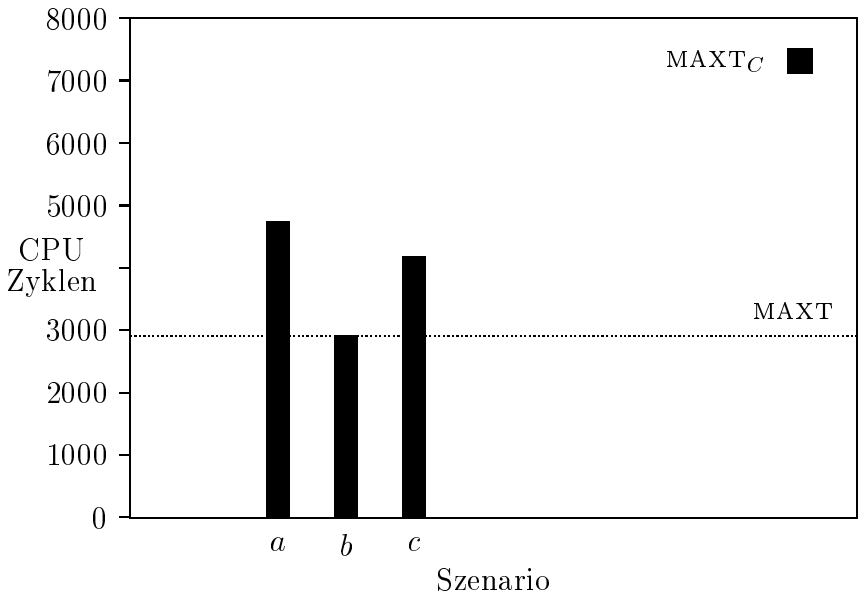


Abbildung 7.6: MAXT_Cs und MAXT von Bubble Sort

Bubble Sort enthält ebenfalls geschachtelte Schleifen. Die Anzahl der Iterationen der inneren Schleife ist jedoch nicht konstant, sondern nimmt jeden Wert zwischen eins und sechs genau ein Mal an. Für die innere Schleife ergeben sich daher in Summe $\sum_{i=1}^{6} i = 21$

[2]Der Multiplikation von Matrizenelementen wird im Assemblercode von *mmult* mit der Instruktion *mult* realisiert. Bei der tatsächlichen Exekution von *mmult* auf einem Computersystem wären daher variable Ausführungszeiten, kleiner gleich der MAXT, zu erwarten.

Iterationen, die bei jeder Abarbeitung ausgeführt werden. Im Worst Case liefert die Bedingung des *if*-Statements bei jeder Auswertung den Wahrheitswert "true". Obwohl dieses Wissen in Szenario b enthalten ist, liefert die Lösung der Optimierungsaufgabe eine MAXT_C, die um acht Zyklen über der tatsächlichen MAXT liegt. Die Untersuchung der Ursache für diese im ersten Moment überraschende Differenz durch Betrachtung der Variablenbelegungen der Lösung erklärt sofort das Resultat: Die Lösung konsumiert die 21 Iterationen der inneren Schleife in vier Iterationen der äußeren Schleife, nicht wie erwartet in sechs. In die Zeit der restlichen beiden Iterationen der äußeren Schleife geht der *then*-Zweig des äußeren *if* gar nicht ein.

Überlegt man sich die Semantik des Codes genauer, so sieht man, daß die Bedingung $(i \leq j)$ des äußeren *if*s in jeder Iteration der äußeren Schleife "true" liefert. Die entsprechende Modifikation der Optimierungsaufgabe, die dies berücksichtigt, liefert dann auch die 1912 Zyklen für die MAXT_C (Zur Verbesserung der Lesbarkeit der Prozedur ist es sinnvoll, die Bedingung überhaupt aus dem Code zu entfernen).

Szenario	MAXT_C	*Differenz*
Ref.	12 794	—
a	55 250	42 456
b	32 710	19 916
c	22 252	9 458
d	22 156	9 362
e	16 010	3 216
f	16 010	3 216
g	12 818	24
h	14 176	1 382

Tabelle 7.6: MAXT_Cs von Sortieren durch Mischen in CPU-Zyklen

Während die Einführung des Markers in Szenario b eine Verbesserung der MAXT_C um gute 38%, vom 1.6-fachen der MAXT praktisch auf die MAXT, bringt, liefert Szenario c, in dem nur die Anzahl von Vertauschungen, nicht aber der Iterationen der inneren Schleife beschränkt wird, ein relativ pessimistisches Ergebnis. Die Schranke liegt gut 1.4 Mal über der maximalen Abarbeitungszeit (Tabelle 7.5 und Abbildung 7.6).

Die Resultate der Szenarien von *Sortieren durch Mischen* zeigen, daß sehr wenig Zusatzinformation ausreichen kann, um eine substantielle Verbesserung der berechneten Zeitschranke zu erreichen. Diese Information ist außerdem sehr leicht aus der Semantik des Codes abzuleiten (siehe Tabelle 7.6 und Abbildung 7.7, Szenarien b und c im Vergleich zu a).

Ist man mit den ersten, relativ schnell erreichten Verbesserungen nicht zufrieden und setzt sich das Ziel, eine sehr knappe oder die bestmögliche MAXT_C zu berechnen, so kann man zwei Punkte beobachten: Die Suche nach weiteren Zusammenhängen im Code wird

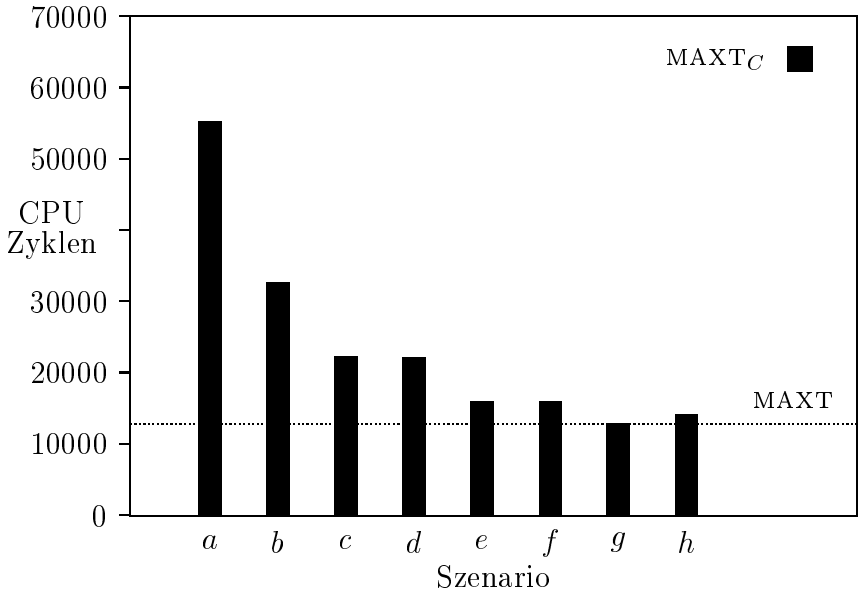


Abbildung 7.7: MAXT$_C$s und MAXT von Sortieren durch Mischen

Szenario	MAXT$_C$	*Differenz*
Ref.	39 522 442	—
a	468 771 858	429 095 920
b	39 675 938	153 496

Tabelle 7.7: MAXT$_C$s von *Calculate_Center* in CPU-Zyklen

aufwendiger und der Qualitätsgewinn für die Schranken nimmt ab. Während die erste Schranke noch 4.3 Mal über der MAXT liegt, liegt der für Szenario *b* errechnete Wert bei 59% (2.6 Mal MAXT), das Resultat von *c* bei 40% (1.7 Mal MAXT) des ursprünglichen Wertes. Mit den weiteren Informationen der Szenarien *d* bis *h* wird zwar die MAXT bis auf 0.2% erreicht, die Verbesserungen der Schranken sind aber kleiner als in den ersten beiden Schritten.

Da uns die Ursache für die Differenz von 24 Zyklen zwischen bester MAXT$_C$ und MAXT interessierte, wurden nach den Experimenten der Code und die Worst Case Abarbeitungspfade noch einmal untersucht. Die Annahme, daß die Differenz dadurch entsteht, daß die verknüpfte Bedingung der ersten *while*-Schleife nicht bei jeder Iteration vollständig getestet wird, erwies sich für den Worst Case als falsch. Im Worst Case wird die Bedingung jedes Mal vollständig getestet. Es fiel jedoch auf, daß die Exekution des *then*-Zweiges des in dieser Schleife enthaltenen *if*-Statements um 8 Zyklen länger braucht als die des *else*-Zweiges. Bei der Ausführung von *merge_sort* mit den Testdaten, die die MAXT liefern, wird drei Mal der *else*-Zweig durchlaufen. Im Gegen-

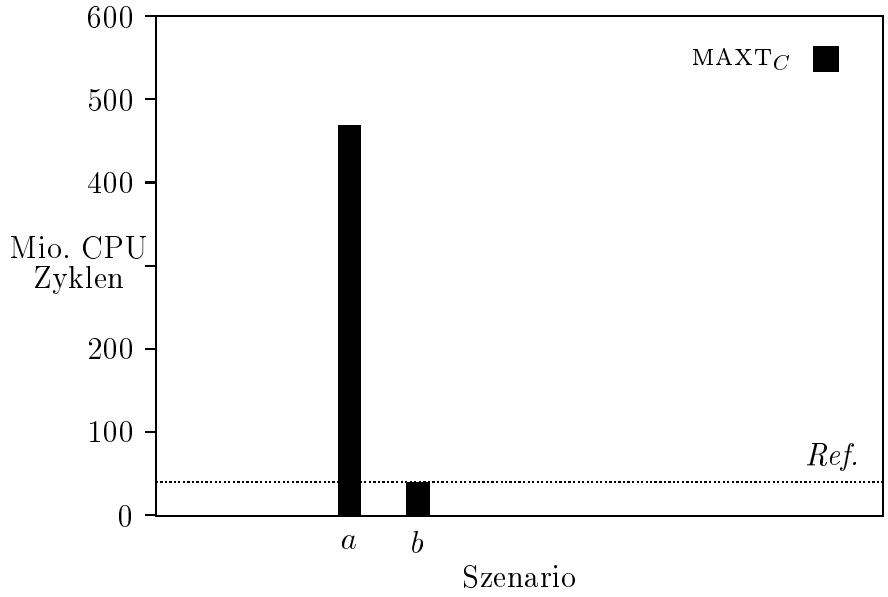


Abbildung 7.8: MAXT$_C$s und Referenzausführungszeit von *Calculate_Center*

satz dazu, wird bei der Berechnung der MAXT$_C$ für jede Iteration die Ausführung des längeren *then*-Zweiges angenommen. Dies erklärt die Differenz.

Im Rahmen der durchgeführten Experimente ist bei der Prozedur *Calculate_Center* die erzielbare Verringerung der MAXT$_C$ am größten. Während die MAXT$_C$ für Szenario a ungefähr 11.8 Mal über der MAXT liegt, liegen die Werte beim Szenario b nur um einige Promille auseinander. Wir können die Differenz aus den vorhandenen Daten für Szenario b zwar nicht genau bestimmen, die vorhandenen Referenzdaten erlauben aber diese Abschätzung (siehe Tabelle 7.7 und Abbildung 7.8).

7.4 Laufzeitüberprüfungen von Restriktionen

In Kapitel 6 wurde erläutert, daß es sinnvoll ist, die Restriktionen, die die möglichen Abarbeitungspfade durch eine Prozedur beschreiben, zur Laufzeit zu überprüfen. Diese Überprüfungen benötigen zur Laufzeit selbst Zeit und erhöhen damit die Abarbeitungszeiten des Codes. Die Art und das Ausmaß, wie die Laufzeitkontrolle der Restriktionen die MAXT und MAXT$_C$ beeinflußt, soll hier präsentiert werden.

Die Tabellen 7.8 bis 7.10 und Abbildungen 7.9 bis 7.11 zeigen jeweils die aus den Simulationen gewonnenen und analytisch ermittelten MAXT-Werte für die Beispiele unter Berücksichtigung der Laufzeitüberprüfung von Restriktionen in CPU-Zyklen, sowie die Differenz dieser beiden Werte für jedes Szenario. Die mit MAXT bezeichnete Spalte enthält die durch Simulation gewonnene maximale Abarbeitungszeit, die mit MAXT$_C$ bezeichnete die Schranke für das angegebene Szenario, die folgende Spalte die Differenz

dieser beiden Werte. Die letzten Spalten der Tabellen enthalten die Zeiten (in Zyklen), die im jeweiligen Worst Case für die Überprüfung der Restriktionen aufgewendet werden müssen. Wie schon im vorigen Abschnitt wurden für die Prozedur *Calculate_Center* nicht die MAXTs sondern die Ausführungszeit des oben beschriebenen Szenarios, mit XT bezeichnet, ermittelt (Tabelle 7.11 und Abbildung 7.12). Die Referenzwerte in den ersten Zeilen der Tabellen sind in den Tabellen 7.8 bis 7.10 die maximalen Abarbeitungszeiten, bzw. in Tabelle 7.11 die Ausführungszeit des gegebenen Szenarios *ohne* Berücksichtigung von Laufzeitüberprüfungen.

Szenario	MAXT	MAXT_C	*Differenz*	*Überprfg.*
Ref.	27 666	—	—	—
a	34 052	34 052	0	6 386

Tabelle 7.8: MAXT und MAXT_C der Matrixmultiplikation unter Berücksichtigung der Laufzeitüberprüfung von Restriktionen in CPU-Zyklen

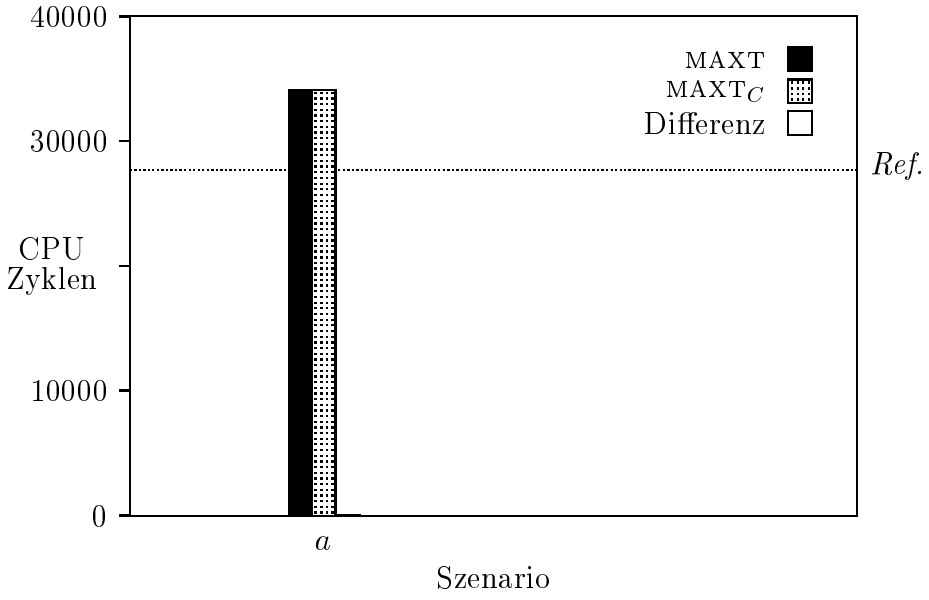


Abbildung 7.9: MAXT und MAXT_C der Matrixmultiplikation mit Überprüfung der Restriktionen

In den Abbildungen 7.9 bis 7.12 werden die Werte für MAXT, MAXT_C (jeweils mit Laufzeitüberprüfung der Restriktionen und Schleifengrenzen) und die Differenz dieser Werte für alle Szenarien in Form von Balken angegeben. Der Referenzwert, der keine Zeiten für Überprüfungen inkludiert, wird als gepunktete horizontale Linie dargestellt.

In den meisten Fällen ist zu beobachten, daß sich durch das Hinzufügen von Restriktionen zwar die MAXT erhöht, daß sich aber auf der anderen Seite die berechnete

Schranke, die MAXT$_C$, reduziert. Diese Aussage gilt allerdings nur bedingt; nämlich genau dann, wenn die durch eine Restriktion in die MAXT-Berechnung eingebrachte Information die berechnete Schranke stärker reduziert, als die Laufzeitkosten, die die Überprüfung dieser Restriktion verursacht, die MAXT$_C$ wiederum erhöhen.

Szenario	MAXT	MAXT$_C$	*Differenz*	*Überprfg.*
Ref.	2 912	—	—	—
a	4 050	6 450	2 400	1 138
b	4 906	4 914	8	1 994
c	4 906	6 736	1 830	1 994

Tabelle 7.9: MAXTs und MAXT$_C$s von Bubble Sort unter Berücksichtigung der Laufzeitüberprüfung von Restriktionen in CPU-Zyklen

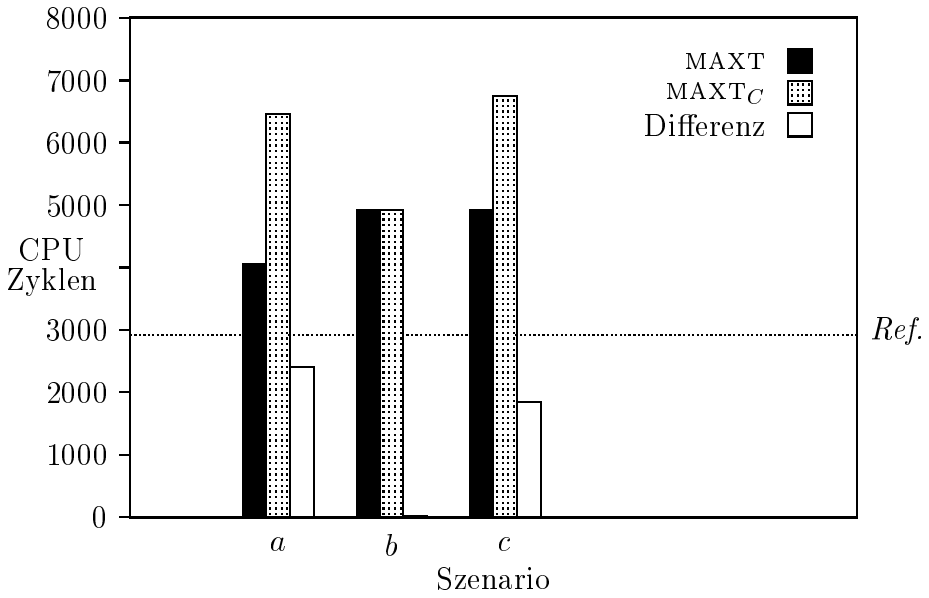


Abbildung 7.10: MAXTs und MAXT$_C$s von Bubble Sort mit Überprüfung der Restriktionen

Beispiele für Restriktionen, die mehr Aufwand für ihre Überprüfung kosten, als sie zur Verringerung der MAXT$_C$ beitragen, findet man beim Sortieren durch Mischen. Vergleicht man die Szenarien *c* und *d* ohne Berücksichtigung der Überprüfungskosten (Tabelle 7.6), so ist die MAXT$_C$ von Szenario *d* geringer als die von *c*. Berücksichtigt man jedoch den Aufwand für die Überprüfung der zusätzlichen Ungleichung von Beispiel *d*, so liegt die MAXT$_C$ von Szenario *d* über der von *c* (Tabelle 7.10). Änliches gilt für die Szenarien *e* und *f*. Ohne Berücksichtigung des Überprüfungsaufwands sind die Zeitschranken gleich. Zieht man die Zeit für die Überprüfung in Betracht, so wirkt sich

die redundante Information aus Beispiel e in einer höheren Abarbeitungszeit aus (siehe Tabelle 7.10).

Szenario	MAXT	MAXT_C	*Differenz*	*Überprfg.*
Ref.	12 794	—	—	—
a	15 322	63 452	48 130	2 528
b	15 618	37 928	22 310	2 824
c	16 506	28 694	12 188	3 712
d	16 834	28 926	12 092	4 040
e	16 834	21 142	4 308	4 040
f	16 538	20 846	4 308	3 744
g	17 394	17 418	24	4 600
h	16 506	17 888	1 382	3 712

Tabelle 7.10: MAXTs und MAXT_Cs für das Sortieren durch Mischen unter Berücksichtigung der Laufzeitüberprüfung von Restriktionen in CPU-Zyklen

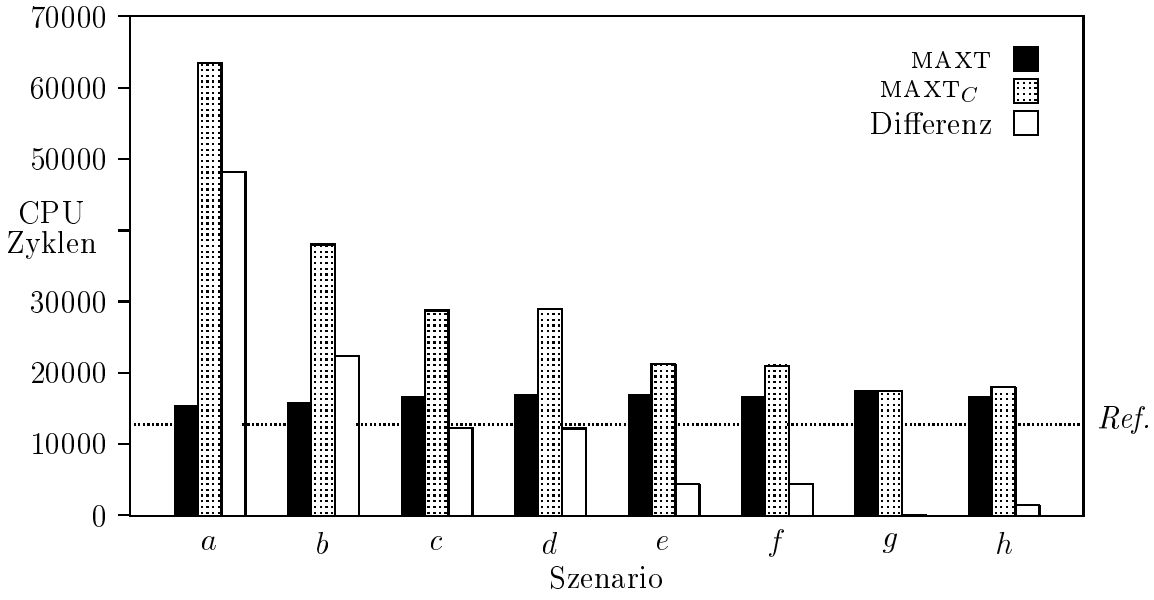


Abbildung 7.11: MAXT und MAXT_C für das Sortieren durch Mischen mit Überprüfung der Restriktionen

Die Überprüfung von Restriktionen macht in den untersuchten Prozeduren zwischen 15% (*Calculate_Center*) und 40% (Bubble Sort) der Gesamtabarbeitungszeit aus. Der relative Zeitaufwand für Überprüfungen hängt dabei wesentlich von der Anzahl und Komplexität der Berechnungen innerhalb der Schleifen des untersuchten Programmstückes ab. Sind die Berechnungen selbst zeitaufwendig, fallen die Überprüfun-

gen weniger ins Gewicht. Werden, wie beim Bubble Sort, im Applikationscode nur einfache Datenmanipulationen durchgeführt, wirken sich die Restriktionen stärker aus.

Szenario	XT	MAXT_C	*Differenz*	*Überprfg.*
Ref.	—	39 675 938	—	—
a	46 517 520	551 475 096	504 957 576	6 841 582
b	46 656 736	46 810 232	153 496	6 980 798

Tabelle 7.11: MAXTs und MAXT_Cs von *Calculate_Center* unter Berücksichtigung der Laufzeitüberprüfung von Restriktionen in CPU-Zyklen

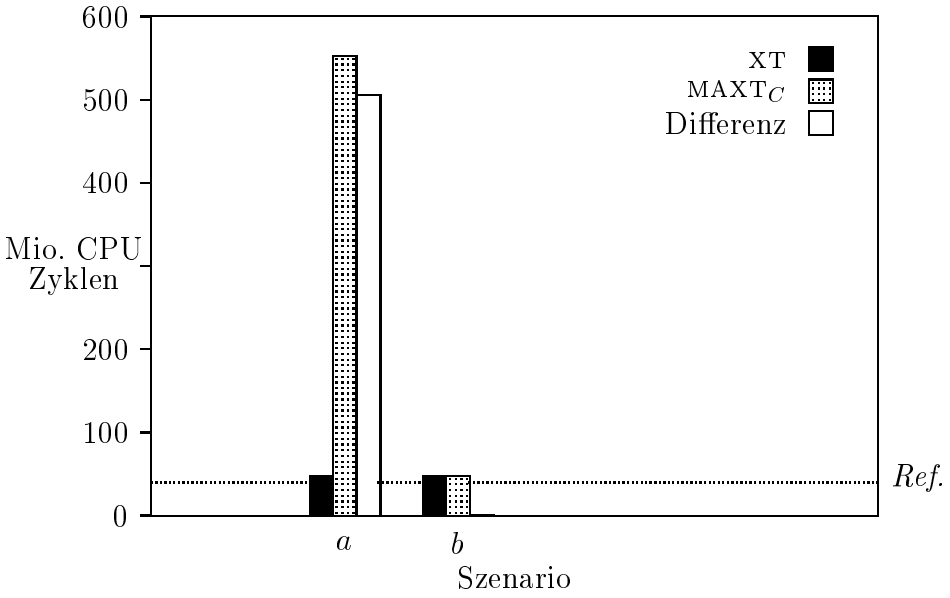


Abbildung 7.12: MAXT und MAXT_C von *Calculate_Center* mit Überprüfung der Restriktionen

7.5 Zusammenfassung

Die Resultate der Experimente geben Auskunft über die Qualität, die bei der Berechnung der MAXT_C für praktische Beispiele erreicht werden kann. Spiegeln die Struktur des analysierten Codes bzw. die angegebenen maximalen Iterationsanzahlen von Schleifen das Verhalten von tatsächlichen Abarbeitungspfaden wider, so können ohne viel Zusatzinformation knappe Zeitschranken berechnet werden. Sind die Abarbeitungspfade eines Codestücks stark datenabhängig und werden Schleifen mit stark variierender

(nicht maximaler) Iterationsanzahl exekutiert, so kann dies zu anfänglich pessimistischen MAXT-Schätzungen führen. In diesen Fällen kann das durch Restriktionen ausgedrückte Wissen über das dynamische Verhalten zu wesentlichen Reduktionen der MAXT_C, mit dem Resultat, daß die berechnete Schranke sehr knapp an der MAXT liegt, führen.

Im zweiten Teil der Experimente wurde untersucht, wie groß der Laufzeitaufwand ist, wenn man die Einhaltung von Iterationsgrenzen, die für Schleifen angegeben werden, und Restriktionen zur Pfadbeschreibung zur Laufzeit überprüft. In Abhängigkeit davon, wie komplex die vom Programm durchzuführenden Berechnungen im Vergleich zur Anzahl von Restriktionen und Schleifen waren, betrug der Aufwand für die Überprüfungen zwischen 15 und 40%. Dieser Aufwand ist zwar im Vergleich zu den durch zusätzliche Information erreichbaren Verbesserungen bei der Berechnung von Zeitschranken gering, kann auf der anderen Seite aber doch nicht als vernachlässigbar angesehen werden.

Kapitel 8

Zusammenfassung und Ausblick

Zusammenfassung

In dieser Arbeit wurde eine Methode zur Berechnung der maximalen Abarbeitungszeit von Programmen vorgestellt. Diese Methode verwendet Verfahren der *ganzzahligen linearen Optimierung*, um sehr knappe Schranken für die Abarbeitungszeiten zu ermitteln. Dabei werden Struktur, Kontrollfluß und Zeitinformation eines gegebenen Programms in einer linearen Zielfunktion und einer Menge von Nebenbedingungen beschrieben. Durch die Maximierung der Zielfunktion unter Beachtung der Restriktionen erhält man eine obere Schranke für die Abarbeitungszeit.

Ausgangspunkt für die Generierung der linearen Optimierungsprobleme bilden sogenannte erweiterte T-Graphen. Erweiterte T-Graphen sind Graphen zur Beschreibung der Struktur von Codestücken, deren Kanten mit den Ausführungszeiten der entsprechenden Instruktionsfolgen versehen sind. Außerdem sind sie mit Restriktionen annotiert, die angeben, wie oft bestimmte Kanten oder Mengen von Kanten maximal abgearbeitet werden können. Die Berechnung der maximalen Abarbeitungszeit für einen erweiterten T-Graphen wird durch die Suche nach einem Pfad maximaler Ausführungszeit realisiert. Diese Aufgabe wiederum wird auf das Problem der Ermittlung einer Zirkulation maximaler Kosten unter den gegebenen Restriktionen in einem Flußnetzwerk zurückgeführt, das mit den Methoden der linearen Optimierung gelöst wird. Die konstanten Terme der Zielfunktion bilden die Abarbeitungszeiten der Kanten, die Variablen repräsentieren die (zunächst unbekannte) Anzahl der Abarbeitungen der den Kanten entsprechenden Anweisungen. Restriktionen beschreiben die Struktur von Programmen und vom Benutzer angegebene Informationen über das Programmverhalten in Form von Gleichungen und Ungleichungen, die mögliche Flüsse im Flußnetzwerk abschranken.

Besonderes Augenmerk wurde bei der Entwicklung der Zeitanalysemethode darauf gelegt, daß die berechneten Zeitschranken von hoher Qualität sind: Restriktionen, die vom Programmierer angegeben werden, erlauben es, Wissen über unmögliche Ab-

arbeitungspfade bzw. Zusammenhänge der Anzahl der Abarbeitungen verschiedener Programmteile soweit zu berücksichtigen, daß für den erweiterten T-Graphen jedes Programms die *tatsächliche* maximale Abarbeitungszeit, nicht nur eine Schranke für diesen Wert, berechnet werden kann. Zusätzlich zur maximalen Abarbeitungszeit liefert das Verfahren Information darüber, wie oft jeder Programmteil zur gefundenen Lösung beiträgt. Damit kann der Anteil jedes einzelnen Konstruktes zur maximalen Abarbeitungszeit bestimmt werden. Dies ist besonders dann von Interesse, wenn für eine Optimierung des Codes zeitintensive Programmabschnitte identifizieren werden sollen.

Die Praxistauglichkeit der Zeitanalysemethode wurde in Experimenten getestet. Dabei wurden einige Tasks implementiert, anschließend wurden ihre maximalen Abarbeitungszeiten analytisch und durch Simulation ermittelt. Es konnte gezeigt werden, daß durch die Berücksichtigung der Benutzerinformation über Eigenschaften von Programmpfaden in einigen Fällen erheblich bessere Schranken berechnet werden können. In einem Beispiel konnte die berechnete Schranke um einen Faktor 12 verbessert werden. Auf der anderen Seite konnte festgestellt werden, daß schon die Angabe einiger, schnell erkennbarer Restriktionen die Ermittlung wesentlich knapperer Zeitschranken erlaubt. Die Suche nach einer vollständigen Beschreibung von möglichen bzw. unmöglichen Abarbeitungspfaden ist aber mitunter aufwendig. In der Praxis kann es in solchen Fällen erwägenswert sein, zugunsten eines geringeren Aufwands auf eine vollkommene Analyse des dynamischen Verhaltens des Programms verzichten.

In einer zweiten Serie von Experimenten wurde untersucht, wie groß der zeitliche Aufwand ist, wenn die vom Benutzer angegebenen Restriktionen zur Beschreibung des Programmverhaltens zur Laufzeit überprüft werden. Dieser Overhead hängt sehr von der Komplexität des Codes und der Menge an Zusatzinformationen ab. Er bewegte sich in den betrachteten Beispielen zwischen 15% und 40%. Im Vergleich zu den durch die zusätzliche Information erreichbaren Verbesserungen bei der Berechnung von Zeitschranken ist dieser Aufwand zwar gering, er ist aber doch so groß, daß er nicht als vernachlässigbar bezeichnet werden kann.

Ausblick

Ausgehend von der vorliegenden Arbeit sind Weiterentwicklungen in verschiedene Richtungen denkbar. An dieser Stelle sollen nur einige Ideen für zukünftige Arbeiten angeführt werden.

- Im Rahmen des MARS-Projekts wurde die Echtzeitprogrammiersprache MODULA/R [Vrc92] entwickelt. Diese Sprache enthält einerseits nur abschrankbare Konstrukte, auf der anderen Seite aber auch die Marker, Scopes und Loop Sequences, die in [Pus89] vorgestellt wurden. Es ist geplant, die in dieser Arbeit vorgestellten Restriktionen zur Beschreibung des Programmverhaltens in

den Sprachumfang von MODULA/R aufzunehmen.

- Eine weitere Zielsetzung wäre, den Zeitaufwand für die Laufzeitüberprüfung von Schleifengrenzen und Restriktionen zu verringern. Dies könnte dadurch erreicht werden, daß die Korrektheit von Schleifengrenzen bzw. Restriktionen soweit möglich schon vor der Laufzeit verifiziert wird. So wäre zum Beispiel eine Laufzeitüberprüfung der Anzahl der Iterationen von *for*-Schleifen nicht nötig, wenn die Iterationsgrenzen bereits zur Übersetzungszeit bekannt sind. Weitere Informationen über das Programmverhalten könnten eventuell durch Datenflußanalyse gewonnen werden.

- Die in dieser Arbeit verwendeten Restriktionen zur Beschreibung des dynamischen Programmverhaltens sind zwar in ihrer Aussagekraft sehr mächtig, aber nicht leicht zu verwenden. Aussagen wie "Die Abarbeitung von Abschnitt A schließt die Abarbeitung von Abschnitt B aus" müssen in lineare Gleichungen oder Ungleichungen übersetzt werden, die für den Menschen schwer verständlich sind. Eine eigene, leicht verständliche Beschreibungssprache, wie sie z.B. in [Par93] vorgestellt wird, könnte den Umgang mit Information über das Verhalten von Programmen für den menschlichen Benutzer erleichtern.

- Eine weiterer interessanter Forschungsschwerpunkt wäre es, die Resultate der Zeitanalyse einem optimierenden Compiler zuzuführen. Dieser Compiler könnte die detaillierte Zeitinformation dazu verwenden, Programme so zu optimieren, daß ihre maximale Abarbeitungszeit minimiert wird.

Literaturverzeichnis

[Age93] R. Ager. Hierarchische Speicher in Echtzeitsystemen. Master's Thesis, Institut für Technische Informatik, Technische Universität Wien, Vienna, Austria, Sept. 1993.

[Ame85] P. Amerasinghe. *A Universal Hardware Simulator.* Undergraduate Honors Thesis, Dept. of Computer Sciences, University of Texas, Austin, TX, USA, Dec. 1985.

[Ame88] P. Amerasinghe. An Extended C Compiler for Timing Analysis of Real-Time Software. Documentation, Department of Computer Science, University of Texas, Austin, Texas, 1988.

[Bur72] R.E. Burkard. *Methoden der ganzzahligen Optimierung.* Springer-Verlag, 1972.

[Che87] M. Chen. *A Timing Analysis Language – (TAL).* Programmer's Manual, Dept. of Computer Sciences, University of Texas, Austin, TX, USA, 1987.

[Cog91] B. Cogswell and Z. Segall. MACS: A Predictable Architecture for Real Time Systems. In *Proc. 12th Real-Time Systems Symposium*, pages 296–305, Dec. 1991.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[Fuc93] E. Fuchs. Berücksichtigung eines Instruktionscaches bei der Task-Zeitanalyse für Echtzeitsysteme. Master's Thesis, Institut für Technische Informatik, Technische Universität Wien, Vienna, Austria, Sept. 1993.

[Har91] M. G. Harmon. *Predicting Execution Time on Contemporary Computer Architectures.* PhD Thesis, Department of Computer Science, Florida State University, Tallahassee, FL, USA, May 1991.

[Har92] M. G. Harmon, T. P. Baker, and D. B. Whalley. A Retargetable Technique for Predicting Execution Time. In *Proc. 13th Real-Time Systems Symposium*, pages 68–77, Phoenix, AZ, USA, Dec. 1992.

[Hil88] F. S. Hillier and G. J. Lieberman. *Operations Research*. Internationale Standardlehrbücher der Wirtschafts- und Sozialwissenschaften. R. Oldenbourg; Wien, München, 1988.

[Jun90] D. Jungnickel. *Graphen, Netzwerke und Algorithmen*. BI-Wissenschaftsverlag, Mannheim, Wien, Zürich, 2^{nd} edition, 1990.

[Kir89] D. B. Kirk. SMART (Strategic Memory Allocation for Real-Time) Cache Design. In *Proc. 10th Real-Time Systems Symposium*, pages 229–237, Santa Monica, CA, USA, Dec. 1989.

[Kir90] D. B. Kirk and J. K. Strosnider. SMART (Strategic Memory Allocation for Real-Time) Cache Design Using the MIPS R3000. In *Proc. 11th Real-Time Systems Symposium*, pages 322–330, Lake Buena Vista, Florida, USA, Dec. 1990.

[Kli86] E. Kligerman and A. Stoyenko. Real-Time Euclid: A Language for Reliable Real-Time Systems. *IEEE Transactions on Software Engineering*, SE-12(9):941–949, Sep. 1986.

[Kop91] H. Kopetz, R. Zainlinger, G. Fohler, H. Kantz, P. Puschner, and W. Schütz. The Design of Real-Time Systems: From Specification to Implementation and Verification. *IEE Software Engineering Journal*, 6(3):72–82, May 1991.

[Kop93] H. Kopetz, G. Fohler, G. Grünsteidl, H. Kantz, G. Pospischil, P. Puschner, J. Reisinger, R. Schlatterbeck, W. Schütz, A. Vrchoticky, and R. Zainlinger. Real-Time System Development: The Programming Model of MARS. In *Proc. of the International Symposium on Autonomous Decentralized Systems*, pages 290–299, Kawasaki, Japan, Mar/Apr 1993.

[Kor71] A.A. Korbut and J.J. Finkelstein. *Diskrete Optimierung*. Akademie-Verlag Berlin, 1971.

[Lap92] J.-C. Laprie, Editor. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault Tolerance*. Springer-Verlag, 1992.

[Lei80] D. W. Leinbaugh. Guaranteed Response Times in a Hard-Real-Time Environment. *IEEE Transactions on Software Engineering*, SE-6(1), Jan. 1980.

[Lei82] D. W. Leinbaugh and M.-R. Yamini. Guaranteed Response Times in a Distributed Hard-Real-Time Environment. In *Proc. 3rd Real-Time Systems Symposium*, pages 157–169, Dec. 1982.

[Lei86] D. W. Leinbaugh and M.-R. Yamini. Guaranteed Response Times in a Distributed Hard-Real-Time Environment. *IEEE Transactions on Software Engineering*, SE-12(12), Dec. 1986.

[Lin91] L. Lindh and F. Stanischewski. FASTCHART – A Fast Time Desterministic CPU and Hardware Based Real-Time Kernel. In *Proc. Euromicro Workshop on Real-Time Systems*, pages 36–40, Paris-Orsay, France, 1991.

[Liu73] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[McH90] J. A. McHugh. *Algorithmic Graph Theory.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Mok84] A. K. Mok. The Design of Real-Time Programming Systems Based on Process Models. In *Proc. 5th Real-Time Systems Symposium*, pages 5–17, Dec. 1984.

[Mok89] A. K. Mok, P. Amerasinghe, M. Chen, and K. Tantisirivat. Evaluating Tight Execution Time Bounds of Programs by Annotations. In *Proc. 6th IEEE Workshop on Real-Time Operating Systems and Software*, pages 74–80, Pittsburgh, PA, USA, May 1989.

[Nem88] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons Ltd., 1988.

[Nem89] G.L. Nemhauser, A.H.G. Rinnooy Kan, and M.J.Todd. *Optimization*, volume 1 of *Handbooks in Operations Research and Management Science.* North Holland, 1989.

[Nie91a] D. Niehaus. Program Representation and Translation for Predictable Real-Time Systems. In *Proc. 12th Real-Time Systems Symposium*, pages 53–63, Dec. 1991.

[Nie91b] D. Niehaus, E. Nahum, and J. A. Stankovic. Predictable Real-Time Caching in the Spring System. In *Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 80–87, Atlanta, GA, USA, May 1991.

[Par89] C. Y. Park and A. C. Shaw. A Source-Level Tool for Predicting Deterministic Execution Times of Programs. Technical Report 89-09-12, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, USA, Sep. 1989.

[Par90] C. Y. Park and A. C. Shaw. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. In *Proc. 11th Real-Time Systems Symposium*, pages 72–81, Lake Buena Vista, FL, USA, Dec. 1990.

[Par92] C. Y. Park. *Predicting Deterministic Execution Time of Real-Time Programs.* PhD Thesis, Department of Computer Science and Engineering, University of Washington, Seattle, WA, USA, Aug. 1992.

[Par93] C. Y. Park. Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. *Real-Time Systems*, 5(1):31–62, 1993.

[Pos92] G. Pospischil, P. Puschner, A. Vrchoticky, and R. Zainlinger. Developing Real-Time Tasks with Predictable Timing. *IEEE Software*, 9(5):35–44, Sep. 1992.

[Pus89] P. Puschner and Ch. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1(2):159–176, Sep. 1989.

[Pus90] P. Puschner and R. Zainlinger. Developing Software with Predictable Timing Behavior. In *Proc. 7th IEEE Workshop on Real-Time Operating Systems and Software*, pages 70–76, Charlottesville, VA, May 1990.

[Pus91] P. Puschner and A. Vrchoticky. An Assessment of Task Execution Time Analysis. In *Proc. 10th IFAC Workshop on Distributed Computer Control Systems*, pages 41–45, Semmering, Austria, Sept. 1991. published by Pergamon Press, Oxford, New York, 1992.

[Pus93] P. Puschner and A. Schedl. A Tool for the Computation of Worst Case Task Execution Times. In *Proc. Euromicro Workshop on Real-Time Systems*, pages 224–229, Oulu, Finland, June 1993.

[Reh87] P. H. Rehm. A C Language Cross Development Environment for Real-Time Programming. Master's Thesis, University of California at Irvine, Irvine, CA, USA, Jul. 1987.

[Rei93] J. Reisinger. *Konzeption und Analyse eines zeitgesteuerten Betriebssystems für Echtzeitanwendungen.* PhD Thesis, Technisch-Naturwissenschaftliche Fakultät, Technische Universität Wien, Wien, Österreich, Juli 1993.

[Sch78] T. M. Schaefges. Estimating Execution Times of Path Pascal Programs. Master's Thesis, University of Illinois, IL, USA, 1978.

[Sch92] W. Schütz. *The Testability of Distributed Real-Time Systems.* PhD Thesis, Technisch-Naturwissenschaftliche Fakultät, Technische Universität Wien, Vienna, Austria, Feb. 1992.

[Sch93] A. Schedl. Zeitanalyse von Echtzeitprogrammen mittels linearer Optimierung. Master's Thesis, Institut für Technische Informatik, Technische Universität Wien, Vienna, Austria, Sept. 1993.

[Sha79] M. Shaw. A Formal System for Specifying and Verifying Program Performance. Technical Report CMU-CS-79-129, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, USA, June 1979.

[Sha89] A. C. Shaw. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, SE-15(7):875–889, July 1989.

[Sta87] J. A. Stankovic and K. Ramamritham. The Design of the Spring Kernel. In *Proc. 8th Real-Time Systems Symposium*, pages 146–157, Dec. 1987.

[Sta88] J. A. Stankovic. Misconceptions about Real-Time Computing: A Serious Problem for Next-Generation Systems. *IEEE Computer*, 21(10):10–19, Oct. 1988.

[Sta90] J. A. Stankovic and K. Ramamritham. Editorial: What is Predictability for Real-Time Systems? *Real-Time Systems*, 2(4):247–254, 1990.

[Sta91] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Operating Systems. *IEEE Software*, pages 62–72, May 1991.

[Sta92] W. Stallings. *Operating Systems*. Maxwell Macmillan Publishing Company, 1992.

[Sto87] A. Stoyenko. *A Real-Time Language With A Schedulability Analyzer*. PhD Thesis, Computer Systems Research Institute, University of Toronto, Dec. 1987. Technical Report CSRI-206.

[Vrc89] A. Vrchoticky. Programmiersprachen für verteilte Echtzeitsysteme mit vorhersehbarem Zeitverhalten. Master's Thesis, Institut für Technische Informatik, Technische Universität Wien, Vienna, Austria, Sept. 1989.

[Vrc91] A. Vrchoticky and P. Puschner. On the Feasibility of Response Time Predictions — An Experimental Evaluation. Research Report 2/91, Institut für Technische Informatik, Technische Universität Wien, Vienna, Austria, Jan. 1991.

[Vrc92] A. Vrchoticky. Modula/R Language Definition. Research Report 2/92, Institut für Technische Informatik, Technische Universität Wien, Vienna, Austria, March 1992.

[Vrc93] A. Vrchoticky. Integrating Compilation and Timing Analysis. Research Report 1/93, Institut für Technische Informatik, Technische Universität Wien, Vienna, Austria, Jan. 1993.

[Wei81] Y. Wei. *Real-Time Programming with Fault Tolerance*. PhD Thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA, June 1981.

[WR92] Inc. Wolfram Research. *Mathematica*. Wolfram Research, Inc., Champaign, Illinois, Version 2.1 edition, 1992.

[Zed88] H. Zedan. On the Analysis of OCCAM Real-Time Distributed Computations. *Microprocessing and Microprogramming*, 24:491–500, 1988.

[Zha93] N. Zhang, A. Burns, and M. Nicholson. Pipelined Processors and Worst Case Execution Times. *Real-Time Systems*, 5(4):319–343, Oct. 1993.

Anhang A

Transformation eines Programmstückes in ein Programmierungsproblem

Die folgenden Seiten zeigen die einzelnen Schritte der Konstruktion eines Programmierungsproblems aus einer Zwischendarstellung, die die Struktur und das Basiszeitverhalten eines Codestückes beschreibt, mit Hilfe der in Abschnitt 6.2.2 eingeführten Produktionsregeln.

Den Ausgangspunkt für die Transformation bildet die Bubble Sort Prozedur aus Abbildung 6.11 bzw. die Zwischendarstellung aus Abbildung 6.12. Am Ende steht ein vollständiges ganzzahliges Programmierungsproblem, das ein Zeitanalyseproblem auf einem erweiterten T-Graphen mit Restriktionen beschreibt. Der entsprechende erweiterte T-Graph wird in Abbildung A.1 auf Seite 125 gezeigt.

```
procedure bubble_sort
    68
    scope S
        loop
            maxcount 6
            body
                4
                if
                    condition 4
                    oh_true 8
                    oh_false 10
                    then
                        loop
                            maxcount 6
                            body
                                MarkerM1
                                if
                                    condition 56
                                    oh_true 8
                                    oh_false 10
                                    then 40
                                endif
                            condition 8
                            oh_back 10
                            oh_exit 8
                        endloop
                endif
            condition 12
            oh_back 10
            oh_exit 8
        endloop
        MarkerM1 <= 21
    endscope S
    68
end bubble_sort
```

```
procedure bubble_sort
    (68f1, ∅, {e1}, {e1}, ∅, ∅)
    scope S
        loop
            maxcount 6
            body
                4
                if
                    condition 4
                    oh_true 8
                    oh_false 10
                    then
                        loop
                            maxcount 6
                            body
                                MarkerM1
                                if
                                    condition 56
                                    oh_true 8
                                    oh_false 10
                                    then 40
                                endif
                            condition 8
                            oh_back 10
                            oh_exit 8
                        endloop
                endif
            condition 12
            oh_back 10
            oh_exit 8
        endloop
        MarkerM1 <= 21
    endscope S
    68
end bubble_sort
```

```
procedure bubble_sort
    (68f1, ∅, {e1}, {e1}, ∅, ∅)
    scope S
        loop
            maxcount 6
            body
                (4f2, ∅, {e2}, {e2}, ∅, ∅)
                if
                    condition 4
                    oh_true 8
                    oh_false 10
                    then
                        loop
                            maxcount 6
                            body
                                MarkerM1
                                if
                                    condition 56
                                    oh_true 8
                                    oh_false 10
                                    then 40
                                endif
                            condition 8
                            oh_back 10
                            oh_exit 8
                        endloop
                endif
            condition 12
            oh_back 10
            oh_exit 8
        endloop
        MarkerM1 <= 21
    endscope S
    68
end bubble_sort
```

```
procedure bubble_sort
    (68f_1, ∅, {e_1}, {e_1}, ∅, ∅)
    scope S
        loop
            maxcount 6
            body
                (4f_2, ∅, {e_2}, {e_2}, ∅, ∅)
                if
                    condition 4
                    oh_true 8
                    oh_false 10
                    then
                        loop
                            maxcount 6
                            body
                                MarkerM1
                                if
                                    condition 56
                                    oh_true 8
                                    oh_false 10
                                    then
                                    (40f_3, ∅, {e_3}, {e_3}, ∅, ∅)
                                endif
                            condition 8
                            oh_back 10
                            oh_exit 8
                        endloop
                endif
            condition 12
            oh_back 10
            oh_exit 8
        endloop
        MarkerM1 <= 21
    endscope S
    68
end bubble_sort
```

```
procedure bubble_sort
    (68f_1, ∅, {e_1}, {e_1}, ∅, ∅)
    scope S
        loop
            maxcount 6
            body
                (4f_2, ∅, {e_2}, {e_2}, ∅, ∅)
                if
                    condition 4
                    oh_true 8
                    oh_false 10
                    then
                        loop
                            maxcount 6
                            body
                                MarkerM1
                                (40f_3 + 56f_4 + 8f_5 + 10f_6,
                                {f_4 = f_5 + f_6,
                                f_5 = f_3},
                                {e_4}, {e_3, e_6}, ∅, ∅)
                            condition 8
                            oh_back 10
                            oh_exit 8
                        endloop
                endif
            condition 12
            oh_back 10
            oh_exit 8
        endloop
        MarkerM1 <= 21
    endscope S
    68
end bubble_sort
```

```
procedure bubble_sort
    (68f_1, ∅, {e_1}, {e_1}, ∅, ∅)
    scope S
        loop
            maxcount 6
            body
                (4f_2, ∅, {e_2}, {e_2}, ∅, ∅)
                if
                    condition 4
                    oh_true 8
                    oh_false 10
                    then
                        (40f_3 + 56f_4 + 8f_5 + 10f_6 + 8f_7 + 10f_8 + 8f_9,
                        {f_4 = f_5 + f_6,
                        f_5 = f_3,
                        f_11 ≤ 6f_10,
                        f_10 + f_8 = f_11,
                        f_11 = f_4,
                        f_3 + f_6 = f_7,
                        f_7 = f_8 + f_9},
                        {e_10}, {e_9}, ∅, {(MarkerM1, f_11)})
                endif
            condition 12
            oh_back 10
            oh_exit 8
        endloop
        MarkerM1 <= 21
    endscope S
    68
end bubble_sort
```

```
procedure bubble_sort
    (68f_1, ∅, {e_1}, {e_1}, ∅, ∅)
    scope S
        loop
            maxcount 6
            body
                (4f_2, ∅, {e_2}, {e_2}, ∅, ∅)
                (40f_3 + 56f_4 + 8f_5 + 10f_6 + 8f_7 + 10f_8 + 8f_9 + 4f_12 + 8f_13 + 10f_14,
                {f_4 = f_5 + f_6,
                f_5 = f_3,
                f_11 ≤ 6f_10,
                f_10 + f_8 = f_11,
                f_11 = f_4,
                f_3 + f_6 = f_7,
                f_7 = f_8 + f_9,
                f_12 = f_13 + f_14,
                f_13 = f_10},
                {e_12}, {e_9, e_14}, ∅, {(MarkerM1, f_11)})
            condition 12
            oh_back 10
            oh_exit 8
        endloop
        MarkerM1 <= 21
    endscope S
    68
end bubble_sort
```

```
procedure bubble_sort
    (68f_1, ∅, {e_1}, {e_1}, ∅, ∅)
    scope S
        loop
            maxcount 6
            body
                (4f_2 + 40f_3 + 56f_4 + 8f_5 + 10f_6 + 8f_7 + 10f_8 + 8f_9 + 4f_12 + 8f_13 + 10f_14,
                {f_4 = f_5 + f_6,
                f_5 = f_3,
                f_11 ≤ 6f_10,
                f_10 + f_8 = f_11,
                f_11 = f_4,
                f_3 + f_6 = f_7,
                f_7 = f_8 + f_9,
                f_12 = f_13 + f_14,
                f_13 = f_10,
                f_2 = f_12},
                {e_2}, {e_9, e_14}, ∅, {(MarkerM1, f_11)})
            condition 12
            oh_back 10
            oh_exit 8
        endloop
        MarkerM1 <= 21
    endscope S
    68
end bubble_sort

procedure bubble_sort
    (68f_1, ∅, {e_1}, {e_1}, ∅, ∅)
    scope S
        (4f_2 + 40f_3 + 56f_4 + 8f_5 + 10f_6 + 8f_7 + 10f_8 + 8f_9 + 4f_12 + 8f_13 + 10f_14 +
        12f_15 + 10f_16 + 8f_17,
        {f_4 = f_5 + f_6,
        f_5 = f_3,
        f_11 ≤ 6f_10,
        f_10 + f_8 = f_11,
        f_11 = f_4,
        f_3 + f_6 = f_7,
        f_7 = f_8 + f_9,
        f_12 = f_13 + f_14,
        f_13 = f_10,
        f_2 = f_12,
        f_19 ≤ 6f_18,
        f_18 + f_16 = f_19,
        f_19 = f_2,
        f_9 + f_14 = f_15,
        f_15 = f_16 + f_17},
        {e_18}, {e_17}, ∅, {(MarkerM1, f_11)})
        MarkerM1 <= 21
    endscope S
    68
end bubble_sort
```

```
procedure bubble_sort
```
$(68f_1, \emptyset, \{e_1\}, \{e_1\}, \emptyset, \emptyset)$
$(4f_2 + 40f_3 + 56f_4 + 8f_5 + 10f_6 + 8f_7 + 10f_8 + 8f_9 + 4f_{12} + 8f_{13} + 10f_{14} +$
$12f_{15} + 10f_{16} + 8f_{17},$
$\{f_4 = f_5 + f_6,$
$f_5 = f_3,$
$f_{11} \leq 6f_{10},$
$f_{10} + f_8 = f_{11},$
$f_{11} = f_4,$
$f_3 + f_6 = f_7,$
$f_7 = f_8 + f_9,$
$f_{12} = f_{13} + f_{14},$
$f_{13} = f_{10},$
$f_2 = f_{12},$
$f_{19} \leq 6f_{18},$
$f_{18} + f_{16} = f_{19},$
$f_{19} = f_2,$
$f_9 + f_{14} = f_{15},$
$f_{15} = f_{16} + f_{17},$
$f_{11} \leq 21f_{20},$
$f_{20} = f_{18}\},$
$\{e_{20}\}, \{e_{17}\}, \emptyset, \{(MarkerM1, f_{11})\})$
68
```
end bubble_sort
```

```
procedure bubble_sort
```
$(68f_1, \emptyset, \{e_1\}, \{e_1\}, \emptyset, \emptyset)$
$(4f_2 + 40f_3 + 56f_4 + 8f_5 + 10f_6 + 8f_7 + 10f_8 + 8f_9 + 4f_{12} + 8f_{13} + 10f_{14} +$
$12f_{15} + 10f_{16} + 8f_{17},$
$\{f_4 = f_5 + f_6,$
$f_5 = f_3,$
$f_{11} \leq 6f_{10},$
$f_{10} + f_8 = f_{11},$
$f_{11} = f_4,$
$f_3 + f_6 = f_7,$
$f_7 = f_8 + f_9,$
$f_{12} = f_{13} + f_{14},$
$f_{13} = f_{10},$
$f_2 = f_{12},$
$f_{19} \leq 6f_{18},$
$f_{18} + f_{16} = f_{19},$
$f_{19} = f_2,$
$f_9 + f_{14} = f_{15},$
$f_{15} = f_{16} + f_{17},$
$f_{11} \leq 21f_{20},$
$f_{20} = f_{18}\},$
$\{e_{20}\}, \{e_{17}\}, \emptyset, \{(MarkerM1, f_{11})\})$
$(68f_{21}, \emptyset, \{e_{21}\}, \{e_{21}\}, \emptyset, \emptyset)$
```
end bubble_sort
```

```
procedure bubble_sort
```
$(68f_1, \emptyset, \{e_1\}, \{e_1\}, \emptyset, \emptyset)$
$(4f_2 + 40f_3 + 56f_4 + 8f_5 + 10f_6 + 8f_7 + 10f_8 + 8f_9 + 4f_{12} + 8f_{13} + 10f_{14} +$
$12f_{15} + 10f_{16} + 8f_{17} + 68f_{21},$
$\{f_4 = f_5 + f_6,$
$f_5 = f_3,$
$f_{11} \leq 6f_{10},$
$f_{10} + f_8 = f_{11},$
$f_{11} = f_4,$
$f_3 + f_6 = f_7,$
$f_7 = f_8 + f_9,$
$f_{12} = f_{13} + f_{14},$
$f_{13} = f_{10},$
$f_2 = f_{12},$
$f_{19} \leq 6f_{18},$
$f_{18} + f_{16} = f_{19},$
$f_{19} = f_2,$
$f_9 + f_{14} = f_{15},$
$f_{15} = f_{16} + f_{17},$
$f_{11} \leq 21f_{20},$
$f_{20} = f_{18}\},$
$f_{17} = f_{21}\},$
$\{e_{20}\}, \{e_{21}\}, \emptyset, \{(MarkerM1, f_{11})\})$
```
end bubble_sort
```

```
procedure bubble_sort
```
$(68f_1 + 4f_2 + 40f_3 + 56f_4 + 8f_5 + 10f_6 + 8f_7 + 10f_8 + 8f_9 + 4f_{12} + 8f_{13} + 10f_{14} +$
$12f_{15} + 10f_{16} + 8f_{17} + 68f_{21},$
$\{f_1 = f_{18},$
$f_4 = f_5 + f_6,$
$f_5 = f_3,$
$f_{11} \leq 6f_{10},$
$f_{10} + f_8 = f_{11},$
$f_{11} = f_4,$
$f_3 + f_6 = f_7,$
$f_7 = f_8 + f_9,$
$f_{12} = f_{13} + f_{14},$
$f_{13} = f_{10},$
$f_2 = f_{12},$
$f_{19} \leq 6f_{18},$
$f_{18} + f_{16} = f_{19},$
$f_{19} = f_2,$
$f_9 + f_{14} = f_{15},$
$f_{15} = f_{16} + f_{17},$
$f_{11} \leq 21f_{20},$
$f_{20} = f_{18}\},$
$f_{17} = f_{21}\},$
$\{e_1\}, \{e_{21}\}, \emptyset, \{(MarkerM1, f_{11})\})$
```
end bubble_sort
```

$(68f_1 + 4f_2 + 40f_3 + 56f_4 + 8f_5 + 10f_6 + 8f_7 + 10f_8 + 8f_9 + 4f_{12} + 8f_{13} + 10f_{14} +$
$12f_{15} + 10f_{16} + 8f_{17} + 68f_{21},$
$\{f_1 = f_{18},$
$f_4 = f_5 + f_6,$
$f_5 = f_3,$
$f_{11} \leq 6f_{10},$
$f_{10} + f_8 = f_{11},$
$f_{11} = f_4,$
$f_3 + f_6 = f_7,$
$f_7 = f_8 + f_9,$
$f_{12} = f_{13} + f_{14},$
$f_{13} = f_{10},$
$f_2 = f_{12},$
$f_{19} \leq 6f_{18},$
$f_{18} + f_{16} = f_{19},$
$f_{19} = f_2,$
$f_9 + f_{14} = f_{15},$
$f_{15} = f_{16} + f_{17},$
$f_{11} \leq 21f_{20},$
$f_{20} = f_{18},$
$f_{17} = f_{21},$
$f_{22} = f_1,$
$f_{21} = f_{22},$
$f_{22} = 1\},$
$\emptyset, \emptyset, \emptyset, \emptyset)$

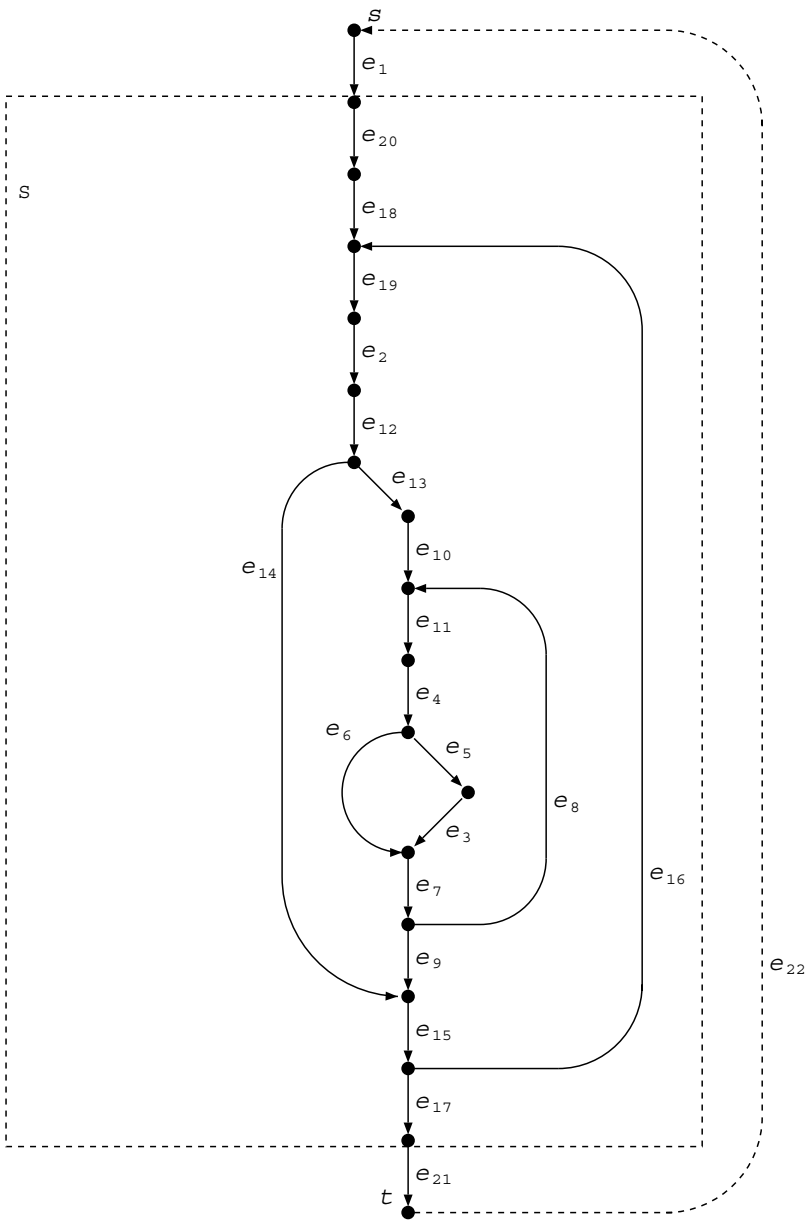


Abbildung A.1: Erweiterter T-Graph für die Bubble Sort Prozedur

Anhang B

Sprache zur Beschreibung von Codestruktur und Zeitverhalten

Dieser Anhang enthält die Grammatik für die Sprache zur Beschreibung der Struktur und der Zeitinformation von Programmen bzw. Programmstücken, wie sie in Kapitel 6 verwendet wurde. Auf den rechten Seiten der angegebenen Produktionsregeln stehen kursive Zeichenketten (z.B. *Stmts*) mit Ausnahme von t_i, t_c, t_t, t_f, t_b und t_e für Nonterminalsymbole. Alle anderen Zeichenketten sind Terminalsymbole. Symbole im Teletype-Font (z.B. `else` oder `+`) stehen für Zeichenketten, die in genau dieser Form lexikalische Einheiten der Sprache bilden. Zeichenketten in Großbuchstaben stehen für Namen (IDENT) bzw. nicht negative ganze Zahlen (NUMBER). Die Terminalsymbole t_i, t_c, t_t, t_f, t_b und t_e repräsentieren ebenfalls nicht negative ganze Zahlen. Sie werden dann verwendet, wenn der Zahlenwert für eine Anzahl von Zeiteinheiten steht.

Procedure → `procedure` IDENT
Stmts
Restr
`end` IDENT

Stmts → *Stmt*
| *Stmt Stmts*

Stmt → *Simple*
| *If*
| *Loop*
| *Exit*
| *Scope*

Simple → t_i

If → `if`
`condition` t_c
`oh_true` t_t
`oh_false` t_f
`then` *Marker Stmts*
Else
`endif`

Else → ϵ
| `else` *Marker Stmts*

Marker → ϵ
| IDENT

Loop → `loop`
`maxcount` NUMBER
`body` *Marker Stmts*
`condition` t_c
`oh_back` t_b
`oh_exit` t_e
`endloop`

Exit → `exit` IDENT

Scope → `scope` IDENT
Stmts
Restr
`endscope` IDENT

Restr → ϵ
| *Restr Rstr*

Rstr → *Expr RelOp Expr* ;

Expr → *Term*
| *Expr AddOp Term*

Term → IDENT
| *MulTerm*

MulTerm → NUMBER
| NUMBER IDENT

RelOp → `<`
| `<=`
| `=`
| `>=`
| `>`

AddOp → `+`
| `-`