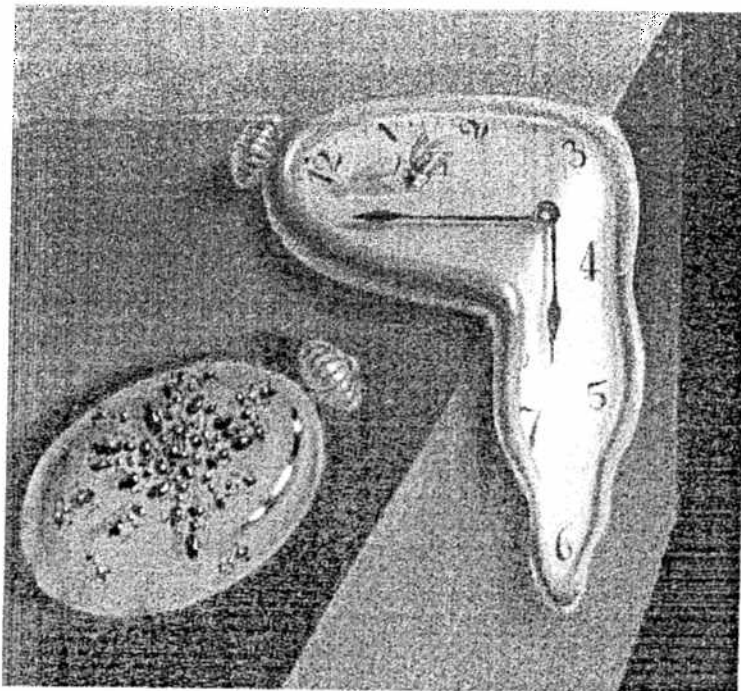


Projektbericht Nr. 183/1-42
November 1993

FWF-Projektantrag
Worst Case Performance Objekt-Orientierter
Echtzeit-Programme (WOOP)
J. Blieberger



Ausschnitt aus: Salvador Dalí, "Die Beständigkeit der Erinnerung"

Projektantrag WOOP:

Worst Case Performance
Objekt-
Orientierter Echtzeit-
Programme

Johann Blieberger
Institut für Automation (183/1),
TU Wien,
Treitlstraße 3/4,
A-1040 Wien

1. November 1993



Zusammenfassung

Die Realisierung von Echtzeitsystemen kann im wesentlichen als Schedulingproblem betrachtet werden. Dabei ist zu garantieren, daß alle zeitlichen Vorgaben eingehalten werden. Zu diesem Zweck wird der dem Echtzeitsystem zugrundeliegende Code in Code-Blöcke unterteilt. Um überhaupt ein Scheduling durchführen zu können, muß man eine obere Schranke für die Laufzeit dieser Code-Blöcke kennen. Man spricht in diesem Zusammenhang von *Worst Case Performance* von Code-Stücken.

Der ständig wachsende Umfang der zu realisierenden Echtzeitsysteme und die solchen Systemen innewohnende Sicherheitsproblematik machen erforderlich, daß sowohl beim Entwurf als auch bei der Implementierung von Echtzeitsystemen entsprechend verantwortungsbewußt vorgegangen wird. In dieser Hinsicht bietet sich ein objektorientiertes Vorgehen an, da damit die Probleme komplexer Systeme sehr gut behandelt werden können.

Daraus resultiert, daß auf Objektebene eine vernünftige Unterstützung für Worst Case Performance-Abschätzungen möglich sein muß. Das Projekt WOOP soll genau dafür entsprechende Voraussetzungen und Konzepte erarbeiten. Dabei soll im speziellen auf eine mögliche Wiederverwendbarkeit von bereits implementierten Objekten und Worst Case Performance-Abschätzungen Rücksicht genommen werden und es soll im Gegensatz zu bereits existierenden Echtzeitsprachen möglich sein, datenabhängige Schleifen und Rekursionen bei der Implementierung von Echtzeitsystemen zuzulassen.

Darüberhinaus soll im Projekt WOOP eine Programmiersprache und ein Prototyp-Compiler für diese Sprache entwickelt werden, die die entwickelten Konzepte geeignet realisieren.

Inhaltsverzeichnis

1	Problemstellung und Stand der Forschung	3
1.1	Einleitung	3
1.2	Der objektorientierte Ansatz zum Entwurf von Computersystemen	3
1.2.1	Wesentliche Konzepte der Objektorientierung	3
1.2.2	Ziele der Objektorientierung	6
1.2.3	Erweiterungen des klassischen Objektmodells	7
1.3	Echtzeitsysteme - Definition und Problematik	8
1.4	Objektorientierte Echtzeitsysteme	9
1.4.1	Aktive und passive Echtzeitobjekte	9
1.4.2	Scheduling von aktiven Objekten	10
1.4.3	Bedeutung von passiven Echtzeitobjekten	11
1.5	Worst case performance: Stand der Forschung	12
2	Projektziele	18
2.1	Allgemeine Ziele des Projektes WOOP	18
2.2	Integration von WCP in das Objektmodell	18
2.3	Für WCP geeignete Datenstrukturen	20
2.3.1	Sortieralgorithmen	21
2.3.2	Suchbäume	22
2.4	Die aWCP-Abschätzung	24
2.5	Eine Hierarchie von WCP-Abschätzungen	25
2.6	Voraussetzungen an eine Programmiersprache	25
2.6.1	Generische passive Echtzeitobjekte	26
2.6.2	Sprachkonstrukte für den Spezifikationsteil von passiven Echtzeitobjekten	26
2.6.3	Sprachkonstrukte für den Implementationsteil von passiven Echtzeitobjekten	27
2.6.4	Objektorientierte Elemente und Relationen und ihr Bezug zu WCP-Abschätzungen	28
2.7	Detaillierte Ziele des Projektes WOOP	29
2.7.1	Erarbeitung von Konzepten	30
2.7.2	Praktische Umsetzung der Konzepte	30
2.8	Projektdurchführung	31
2.9	Mögliche Implikationen und Folgeaktivitäten	32

3	Angaben zur Forschungsstätte und zu beantragten Fördermitteln	39
3.1	Ort der Forschung	39
3.2	Personalressourcen	39
3.3	Vorhandenes Personal	40
3.4	Beantragtes Personal	40
3.5	Vorhandene Sachausstattung	41
3.6	Beantragtes Material	41
3.7	Reisekosten	41
3.8	Publikationskosten	42
4	Allfällige zusätzliche Angaben	43
5	Tabellarische Übersichten	44

1 Problemstellung und Stand der Forschung

1.1 Einleitung

Das Projekt WOOP ruht auf zwei grundlegenden Fundamenten:

- *Objektorientierung* und
- *Echtzeitigkeit*.

Im folgenden soll nun kurz auf objektorientierte Prinzipien eingegangen werden, bevor Echtzeitsysteme definiert werden. Danach werden Konsequenzen dargelegt, die daraus resultieren, wenn man versucht diese beiden Begriffe zusammenzubringen.

1.2 Der objektorientierte Ansatz zum Entwurf von Computersystemen

Das Konzept der Objektorientierung, wie wir es heute verstehen, hat sich innerhalb der letzten beiden Jahrzehnte herausgebildet und (einigermaßen) gefestigt. Als Ahnen der Objektorientierung gelten die Sprache Simula und vor allem Smalltalk (vgl. [GR89]). Es gibt bereits einige empfehlenswerte Lehrbücher, die objektorientierten Entwurf (OOD, object-oriented design) oder objektorientiertes Programmieren (OOP) vermitteln, z.B. [Boo91] oder [RBP⁺91]. Einen hervorragenden Übersichtsartikel stellt [Weg91] dar.

1.2.1 Wesentliche Konzepte der Objektorientierung

Beim objektorientierten Entwurf werden alle im Problemraum auftretenden Einheiten als Objekte betrachtet. Ein Objekt kann von außen nur beeinflusst werden, indem *Methoden* des Objektes verwendet werden. Eine Methode kann man sich als Unterprogramm vorstellen, das den internen Zustand des Objektes liest und/oder modifiziert.

Das *klassische Objektmodell*, besteht aus folgenden Elementen:

- Abstraktion
- Datenkapselung (engl. encapsulation)

- Modularität
- Hierarchie
- Typkonzept

Weiters gibt es die folgenden Relationen zwischen Objekten:

- Verwendungsrelation
- Beinhaltungsrelation

und folgende, die zwischen Klassen wirken:

- Vererbungsrelation (ein- oder mehrfache Vererbung)
- Verwendungsrelation
- Instanziierungsrelation
- Metaklassenrelation

Exakte Definitionen dieser Elemente und Relationen können in [Boo91] nachgelesen werden, wir geben hier nur kurze Definitionen und Erklärungen obiger Elemente und Relationen.

Abstraktion: Eine Abstraktion stellt die wesentlichen Eigenschaften eines Objektes dar, die es von allen anderen Arten von Objekten unterscheiden.

Datenkapselung: Unter Datenkapselung versteht man den Vorgang, alle Einzelheiten eines Objektes, die nicht zu den wesentlichen Eigenschaften zählen, zu verbergen (*information hiding*).

Modularität: Modularität ist die Eigenschaft eines Systems, das in eine Menge von zusammenhängenden, lose gekoppelten Module unterteilt ist.

Hierarchie: Unter Hierarchie versteht man eine Ordnung zwischen Abstraktionen.

Typkonzept: Ein Typkonzept erzwingt, daß Objekte unterschiedlichen Typs nicht oder nur in sehr eingeschränkter Weise vertauscht werden können.

Verwendungsrelation zwischen Objekten: Ein Objekt *A* verwendet ein Objekt *B*, wenn *A* Methoden von *B* aufruft.

Beinhaltungsrelation zwischen Objekten: Ein Objekt *A* beinhaltet ein Objekt *B*, wenn *B* ein Teil der internen Darstellung von *A* ist.

Vererbungsrelation zwischen Klassen: Eine Klasse *B* erbt von seiner Vaterklasse *A* dessen interne Darstellung und dessen Verhalten. Die interne Darstellung von *B* kann aber beim Vererbungsprozeß durch Hinzufügen von zusätzlichen Teilen (Instanzvariablen) vergrößert werden. Genauso kann *B* mehr Methoden haben als *A* und darf die Implementierung einer Methode von *B* anders sein, als die der entsprechenden Methode von *A*.

Man spricht von *einfacher Vererbung*, wenn eine Klasse genau eine Vaterklasse haben darf, und von *mehrfacher Vererbung*, wenn eine Klasse mehr als nur eine Vaterklasse besitzen darf. Ein Beispiel für einfache Vererbung stellt Ada9X ([Ada93]) dar, Beispiele für mehrfache Vererbung finden sich in CLOS ([BDG⁺88]) und in C++ ([ES90]).

Verwendungsrelation zwischen Klassen: Eine Klasse *A* verwendet eine Klasse *B*, wenn entweder in der Spezifikation oder in der Implementation von *A* die Klasse *B* gebraucht wird.

Instanziierungsrelation zwischen Klassen: Eine *generische Klasse* ist eine Klasse, deren Spezifikation und Implementation sich auf andere Klassen, die als *generische Parameter* fungieren, beziehen. Eine generische Klasse eignet sich nicht direkt dazu, daß man Objekte dieser Klasse deklarieren kann, vielmehr muß durch den Vorgang der *Instanziierung* zuerst eine „herkömmliche“ Klasse gewonnen werden. Bei dieser Instanziierung müssen alle generischen Parameter mit aktuellen Werten versehen werden.

Metaklassenrelation zwischen Klassen: Metaklassen treten dann auf, wenn man beginnt Klassen selbst als Objekte einer übergeordneten Klasse zu betrachten. Ein Beispiel für die Unterstützung dieses Konzeptes stellt die Sprache CLOS ([BDG⁺88]) dar.

Da die Metaklassenrelation im Projekt WOOP keine Rolle spielt, werden wir sie von nun an nicht mehr betrachten.

1.2.2 Ziele der Objektorientierung

Die Konzepte der Objektorientierung wurden unter anderem eingeführt, um den Problemen entgegenzuwirken, die beim Entwurf und bei der Realisierung sehr großer Software-Systeme auftreten. Dies betrifft zu allererst die Probleme der Modularisierung und der Wartung. Die einzelnen Elemente des klassischen Objektmodells stehen zu diesen Problemen in folgender Beziehung:

- Die Elemente Abstraktion, Datenkapselung, Modularität und Typkonzept führen gemeinsam mit der Methode des objektorientierten Entwurfes einerseits zur Möglichkeit, bestehende Software-Module wiederzuverwenden. Andererseits ergeben sie eine dem Problembereich hervorragend angepaßte Unterteilung des Gesamtsystems in Untersysteme, die auch noch bei der Wartung und Anpassung der Software an geänderte Umgebungsbedingungen beste Flexibilität bietet. Durch diese Vorteile der Objektorientierung können die Entwicklungskosten von Software beträchtlich gemindert werden, sofern die Software tatsächlich objektorientiert entworfen wird und ein geeignetes Repository für wiederverwendbare Software-Module angelegt und genutzt werden kann.
- Die durch Vererbung aufgebaute Hierarchie von Klassen und Objekten spiegelt ebenfalls den Problemraum bestens wider. Sie kann durch die Wiederverwendung der Implementierung von Methoden gute Dienste im Sinne von Code-Sharing leisten, was zu kompakterem Programm-Code führt.
- Von den verbleibenden Elementen des Objektmodells soll nur auf die Instanziierungsrelation genauer eingegangen werden, da die Vorteile der anderen Relationen ohnehin auf der Hand liegen. Die Instanziierungsrelation spiegelt die Beziehung zwischen generischen Klassen/Objekten, die mit parametrierbaren Schablonen vergleichbar sind, und tatsächlichen Ausprägungen dieser Klassen/Objekte wider. Als Beispiel können die *generischen Einheiten* der Programmiersprache Ada (vgl. [Ada83]) und die *Templates* von C++ (vgl. [ES90]) dienen. Der Hauptzweck generischer Klassen/Objekte liegt darin, daß sie dafür geeignet sind, als wiederverwendbare Software-Module zu fungieren, die im Unterschied

zu den oben genannten „normalen“ Klassen/Objekten auch noch den großen Vorteil aufweisen, parametrierbar zu sein. So ist es etwa denkbar, den abstrakten Datentyp *Stack* zu realisieren, ohne den Typ der Elemente, die auf den *Stack* gepusht und vom *Stack* gepopt werden sollen, all zu sehr einzuschränken, da im wesentlichen nur die Zuweisung für diese Elemente als gegeben vorausgesetzt werden muß.

Läßt man die Argumente des Software-Entwurfes, die für die Verwendung von OOD sprechen, beiseite, so bleibt als wesentlicher Vorteil objektorientierten Vorgehens die *Wiederverwendbarkeit von Klassen und Objekten*, der gegebenenfalls durch *parametrierbare generische Klassen und Objekte* um ein Vielfaches gesteigert werden kann (vgl. [Con87, Hol92]).

1.2.3 Erweiterungen des klassischen Objektmodells

Das im letzten Unterabschnitt eingeführte Objektmodell hat sich jedoch für größere praktische Anwendungen als zu einschränkend erwiesen. So können etwa *Multitasking-Anwendungen*¹ nicht in diesem Objektmodell abgebildet werden. Andere Problembereiche, die ebenfalls zu Änderungen im klassischen Objektmodell Anlaß gegeben haben, sind etwa *verteilte Systeme*, sowie die Problematik, daß es sinnvoll ist, daß die Lebenszeit mancher Objekte länger ist als die des Programmes (*persistente Objekte*) und nicht zuletzt *Echtzeitsysteme*.

Wie schon an anderer Stelle dargelegt (vgl. [Bli92]) ist es bei solchen Erweiterungen des klassischen Objektmodells günstig, wenn *orthogonal* vorgegangen wird, d.h., daß sich die durchgeführten Erweiterungen gegenseitig nicht oder kaum beeinflussen sollen.

Als Resultat solcher Erweiterungen besteht das Objektmodell aus *passiven* und *aktiven* Objekten, wobei letztere den in nicht-objektorientierten Sprachen üblicheren Begriff *Prozeß* ersetzt haben², aus *lokalen* und *verteilten* Objekten, aus *persistenten* und *flüchtigen* Objekten und aus *Echtzeit-* und *Nicht-Echtzeit*objekten.

¹Das sind Programme, die aus mehreren gleichzeitig ablaufenden Prozessen bestehen.

²Dabei handelt es sich nur um eine Änderung der Namensgebung; alle anderen Probleme bei der Realisierung von aktiven Objekten (Prozessen) wie z.B., ob mit ihnen asynchron oder synchron kommuniziert werden soll, sind natürlich vom Designer einer passenden objektorientierten Programmiersprache für Multitasking-Anwendungen zu lösen.

Die schon oben erwähnte Orthogonalität zwischen solchen Erweiterungen führt dazu, etwa auch *aktive, persistente Objekte* als durchaus denkbare und darüberhinaus auch praktische Programmelemente zu akzeptieren. Dies impliziert auch natürlich andere Kombinationen von Erweiterungen. Mit einer speziellen, nämlich mit *passiven Echtzeitobjekten*, werden wir uns noch genau auseinandersetzen, da sie den Hauptbestandteil des Projektes WOOP bilden.

1.3 Echtzeitsysteme - Definition und Problematik

Echtzeitsysteme können laut DIN 44 300 wie folgt durch den Begriff „Realzeitverarbeitung“ definiert werden (vgl. [DIN85]):

Realzeitverarbeitung: Eine Verarbeitungsart, bei der Programme zur Verarbeitung anfallender Daten ständig ablaufbereit sind, derart daß die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorbestimmten Zeitpunkten anfallen.

Aus heutiger Sicht ist die Bezeichnung „Programme“ in obiger Definition etwas zu ungenau, man könnte sie durch „Prozesse, die möglicherweise in einem Netzwerk heterogener Rechner verteilt sind“ ersetzen. Das entspricht mehr dem heute üblichen Aufbau von Echtzeitsystemen. Der Aussage über die zeitgerechte Verfügbarkeit ist jedoch nichts hinzuzufügen.

Wenn man davon ausgeht, daß man genügend Information über die „äußeren Einflüsse“ besitzt, z.B., daß man die maximale Anzahl des Auftretens aller externen Ereignisse pro Zeiteinheit kennt, so besteht die Realisierung eines Echtzeitsystems im wesentlichen aus einem *Scheduling-Problem*. Das heißt, es muß sichergestellt werden, daß alle benötigten Ressourcen (externe Geräte, Speicher, Rechenzeit, ...) in allen denkbaren Situationen so zur Verfügung stehen, daß auf ein Ereignis zeitgerecht reagiert werden kann.

Ein erfolgversprechender Ansatz zur Lösung dieses Scheduling-Problems besteht darin, den (Source-)Code des Echtzeitsystems zu analysieren, ob er den oben genannten Anforderungen entspricht. Diese Analyse wird gewöhnlich automatisch von einem sogenannten *Schedulability Analyzer* ausgeführt (vgl. etwa [HS91]). Die Ergebnisse des Schedulability Analyzer's sollen dabei gewährleisten, daß

- ein System, das nicht geschedult werden kann, *immer* als solches erkannt wird und
- ein System, das geschedult werden kann, *möglichst sicher* als solches erkannt wird.

Die „Kunst“ besteht nun darin, die Anzahl der Systeme, die geschedult werden können, aber vom Schedulability Analyzer nicht als solche erkannt werden, möglichst gering zu halten. Das ist insofern schwierig, als Scheduling-Probleme im allgemeinen *NP-vollständig* sind. Man ist daher, wenn man eine annehmbare Performance erzielen will, auf *Heuristiken* angewiesen.

Alle gängigen Schedulability Analyzer arbeiten nach dem Prinzip, den gesamten (Source-)Code in Code-Stücke (auch Blöcke genannt) zu zerschlagen und diese Blöcke einem Scheduling zuzuführen (vgl. [HS91, SM92, LY86, PK89]). Ein wesentliches Problem besteht dabei darin, die *Performance der Blöcke nach oben abzuschätzen*, da überhaupt nur dann ein Scheduling durchgeführt werden kann.

Ziel des Projektes WOOP ist es, Beiträge zur Abschätzung der oben genannten Performance zu leisten. Dabei wird im speziellen, darauf geachtet werden, daß ein objektorientiertes Modell zugrundeliegt und daß diverse Einschränkungen, die von allen bisher üblichen Schedulability Analyzern gemacht werden, aufgehoben werden. Näheres dazu wird im Kapitel 1.5 erläutert.

1.4 Objektorientierte Echtzeitsysteme

In diesem Abschnitt sollen die Probleme erörtert werden, die bei der Integration von objektorientierten Konzepten in Echtzeitsysteme auftreten.

1.4.1 Aktive und passive Echtzeitobjekte

Gemäß der schon in Kapitel 1.2 eingeführten Nomenklatur bestehen objektorientierte Echtzeitsysteme einerseits aus

- passiven und
- aktiven Objekten (Prozessen)

und andererseits aus

- normalen (nicht echtzeitfähigen) und
- Echtzeitobjekten.

Im Prinzip können diese unterschiedlichen Objekte beliebig untereinander kombiniert werden (z.B. aktive Echtzeitobjekte).

Nicht echtzeitfähige Objekte eines Echtzeitsystems dürfen aber nur in jenen Teilen des Echtzeitsystems Verwendung finden, die die Echtzeitfähigkeit des Systems nicht mitbestimmen. Das können etwa Programmteile sein, die weniger wichtige Arbeit „im Hintergrund“ erledigen wie z.B. niedrig prior laufende Statistiken zur Evaluierung der Systemauslastung. Da wir nur an „wirklichen“ Echtzeitsystemen interessiert sind, werden wir im folgenden solche Objekte nicht mehr erwähnen.

Es bleiben daher nur mehr zwei Arten von Objekten genauer zu untersuchen, nämlich

1. aktive Echtzeitobjekte und
2. passive Echtzeitobjekte.

Diese werden in den beiden folgenden Kapiteln beleuchtet.

1.4.2 Scheduling von aktiven Objekten

Aktive Objekte (Prozesse), die in Echtzeitsystemen Verwendung finden, unterliegen stärkeren Restriktionen als aktive Objekte in gewöhnlichen (nicht echtzeitfähigen) Systemen. Wie schon in Kapitel 1.3 dargelegt, ist die wesentliche Problematik der Echtzeitigkeit die des zeitgerechten Scheduling aller Ressourcen.

Der diesbezüglich vielversprechenste Ansatz ist der, den Source-Code eines Echtzeitsystems zu analysieren, ob ein passendes Scheduling gefunden werden kann (vgl. etwa [ITM90, HS91, SM92]).

Dabei wird, wie schon in Kapitel 1.3 erwähnt, der Code der einzelnen aktiven Objekte in Blöcke unterteilt. Diese Blöcke werden entsprechend ihrer zeitlichen Eigenschaften und der Ressourcen, die sie benötigen, daraufhin untersucht, ob sie einem zeitgerechten Scheduling unterworfen werden können. Dabei kommen unterschiedliche Scheduling-Strategien zum Einsatz (vgl. etwa [XP93] für einen hervorragenden Übersichtsartikel und weiters etwa [LL73, SM92, LY86] zur engeren Thematik).

Ein wesentlicher Punkt ist dabei die Abschätzung der Laufzeit dieser Code-Blöcke. Genauer definieren wir:

Unter *Worst Case Performance* (*WCP*) eines Code-Blocks $\mathcal{B}$ verstehen wir das Maximum aller Laufzeiten von $\mathcal{B}$, wobei alle erreichbaren Programmezustände berücksichtigt werden.

In der wissenschaftlichen Literatur hat sich dafür allerdings noch kein eindeutiger Begriff herausgebildet, man findet etwa auch *maximum execution time*, *worst case execution* oder andere Bezeichnungen. Eine Übersicht über diesen Forschungsschwerpunkt findet sich in Kapitel 1.5.

1.4.3 Bedeutung von passiven Echtzeitobjekten

Dadurch, daß die Größe von Echtzeit- und Automatisierungssystemen ständig im Wachsen begriffen ist, wird es nun allerdings auch notwendig, Gedanken des Software-Entwurfes großer und sehr großer Systeme (*programming-in-the-large*, *programming-in-the-colossal*) in den Entwurf und die Implementierung von Echtzeitsystemen einfließen zu lassen.

Das betrifft vor allem eine vernünftige *Modularisierung*. Diesbezüglich stellen objektorientierte Systeme den vielversprechendsten Ansatz dar.

Im Zusammenhang mit *WCP* bedeutet das, daß Teile der zu untersuchenden Code-Blöcke ausgegliedert und getrennt analysiert werden sollen und können.

Hier treten nun passive Echtzeitobjekte in den Vordergrund. Das sind nämlich Objekte, deren Echtzeitverhalten, genauer das Echtzeitverhalten ihrer Methoden, spezifiziert wird, sodaß der Benutzer eines solchen Objektes nicht über die inneren Details der Implementierung Bescheid wissen muß. Er kann sich voll und ganz darauf verlassen, daß die spezifizierten Zeitschranken von der Implementierung eingehalten werden. Darüberhinaus ist bei entsprechender Vorgangsweise eine Wiederverwendbarkeit passiver Echtzeitobjekte durchaus möglich und wünschenswert, ja sogar viel eher zu erwarten als die Wiederverwendbarkeit aktiver Echtzeitobjekte, die sehr genau an den Problemraum der Applikation angepaßt werden müssen.

Beispiele für solche wiederverwendbaren passiven Echtzeitobjekte sind: Listen, Bäume, Mengen und Stacks. Natürlich kommen nur solche Datenstrukturen in Frage, die ein echtzeitfähiges Verhalten aufweisen³.

³Was wir unter echtzeitfähigem Verhalten von Datenstrukturen verstehen, werden wir

Als primäres Anwendungsgebiet für passive Echtzeitsysteme ergeben sich Ressourcen, die nicht von mehreren aktiven Objekten gleichzeitig verwendet werden. Das heißt, daß sie sich vor allem für die Verwaltung von Daten innerhalb eines aktiven Echtzeitobjektes eignen. Dabei sorgt dann das aktive Echtzeitobjekt für ein passendes Scheduling und dafür, daß auf die eingeschlossenen passiven Echtzeitobjekte nicht parallel zugegriffen wird, und das passive Echtzeitobjekt garantiert das zeitgerechte Verhalten dann, wenn das umschließende aktive Echtzeitobjekt tatsächlich einen Prozessor zugewiesen bekommen hat.

Abschließend soll noch festgestellt werden, daß das Konzept von passiven Echtzeitobjekten nicht zum Standard im Bereich der Programmiersprachen zählt. Als Beispiel sei RTC++ ([ITM90]) erwähnt. Diese Sprache ist eine Echtzeiterweiterung von C++, d.h., RTC++ ist objektorientiert und kennt Echtzeitobjekte. Diese Echtzeitobjekte sind allerdings nur als aktive Objekte definiert, die bestimmte Eigenschaften aufweisen, passive Echtzeitobjekte im oben definierten Sinne kennt RTC++ nicht.

1.5 Worst case performance: Stand der Forschung

In diesem Kapitel wird der Stand der Forschung im Bereich WCP zusammengefaßt. Prinzipiell kann man zwei Arten unterscheiden, wie versucht wird, die Echtzeitproblematik in den Griff zu bekommen:

1. Der erste Ansatz besteht darin, mittels Erweiterungen von Logikkalkülen, die den Begriff „Zeit“ beinhalten, Echtzeitsysteme so spezifizieren zu können, daß mit Hilfe des Kalküls eine Verifikation des implementierten Systems in Bezug auf die Spezifikation möglich ist. Beispiele dafür findet man etwa in [Jon89, ST88, ST89, Wir90, Sch92, DGU91, JM86, Ost92, AK86, Haa81].

Dabei kann man unterscheiden zwischen Systemen, deren Implementation im nachhinein gegen die Spezifikation verifiziert werden, und Systemen, deren Spezifikation durch schrittweise Verfeinerung in eine Implementation verwandelt werden, wobei die durchführbaren Schritte von einem Kalkül festgelegt sind.

noch eingehend erläutern.

2. Der zweite Ansatz besteht darin, Echtzeitsysteme im wesentlichen genauso wie andere Systeme zu entwerfen und zu implementieren, aber den (Source-)Code einer genauen Analyse auf seine Echtzeiteignung hin zu unterwerfen, z.B. [HS91, ITM90, PK89].

Das Projekt WOOP liefert Unterstützung für den zweiten Ansatz, der deswegen vielversprechender ist als der erste, da damit (zumindest gegenwärtig) größere Systeme entworfen und implementiert werden können. Außerdem benötigt man für die Implementierung eines Echtzeitsystemes nach dem zweiten Ansatz keine spezielle Schulung in mathematischer Logik.

Im folgenden erörtern wir, inwieweit sowohl in der Industrie eingesetzte, als auch in der Forschung entwickelte Echtzeitsprachen auf die Thematik WCP-Abschätzung eingehen⁴, stellen wichtige Forschungsergebnisse, die mehr oder weniger programmiersprachenunabhängig sind, für den oben genannten zweiten Ansatz dar und diskutieren ihre Vor- und Nachteile.

1. Die Echtzeitsprache PEARL (vgl. [WW85, DIN82, DIN89]) verfügt zwar über eine große Anzahl von Sprachelementen, die bei der Implementierung von Echtzeitsystemen erforderlich sind⁵, sie ist aber eigentlich nicht echtzeitfähig. Um das zu erreichen, muß die Sprache erst geeignet modifiziert werden (vgl. [HS91]). Darüberhinaus ist PEARL nicht objektorientiert und bietet keine Möglichkeit, die Performance von Code-Blöcken zu spezifizieren.
2. Die Sprache occam-2 (vgl. [Bur88]) basiert auf dem Konzept *CSP* (*Communicating Sequential Processes*) (vgl. [Hoa85]) und ist speziell für Transputer entwickelt worden. Sie verfügt über ein Konstrukt, das Parallelität auf sehr niedriger Ebene einführt (die **par**-Anweisung). Allerdings gibt es keine nennenswerte Unterstützung für das auch für Transputer-Netzwerke erforderliche Scheduling oder für WCP-Abschätzungen.
3. Die Sprache Ada wurde 1983 genormt (vgl. [Ada83]) und ist zur Zeit einem Revisionprozeß unterworfen (vgl. [Ada93]). Ada9X, das Ergebnis dieses Prozesses, wird im Gegensatz zu Ada83 eine objektorientierte

⁴Eine allgemeine Gegenüberstellung einiger Echtzeitsprachen, ohne auf die WCP-Thematik einzugehen, findet man in [BW89].

⁵Das bezieht sich vor allem auf das notwendige Scheduling der einzelnen Prozesse.

Sprache sein, die einfache Vererbung unterstützen wird. Ada besitzt ein Sprachmittel, um Prozesse innerhalb der Sprache realisieren zu können (tasks), und Prozeßsynchronisations- und -kommunikationsmittel auf Sprachebene (das *Rendezvous*-konzept). Andererseits gibt es auf Sprachebene keine Unterstützung zur WCP-Abschätzung⁶.

4. Die objektorientierte Echtzeitsprache RTC++ (vgl. [ITM90]) ist eine Erweiterung von C++ (vgl. [Str86]). Es wurden einige Anweisungen zu C++ hinzugefügt, die die Programmierung von Echtzeitsystemen erleichtern, z.B. eine *within*-, eine *at*- und eine *before*-Anweisung. Allerdings kennt RTC++, wie bereits weiter oben erwähnt, keine passiven Echtzeitobjekte. Darüberhinaus gibt es keine Hilfsmittel zur Gewinnung von WCP-Abschätzungen, obwohl ein Schedulability Analyzer ähnlich dem in [HS91] Erwähnung findet.
5. Die Problematik von WCP wird auch in [GR91] erwähnt. Real-time Concurrent C benutzt ein Werkzeug, das auf [MACT89] beruht. Der Code darf nur Schleifen mit benutzerdefinierten Obergrenzen haben. Rekursion ist nicht erlaubt.
6. In [HS91] werden die Programmiersprache Real-Time Euclid (vgl. dazu auch [KS86]) und ein dazugehöriger *Schedulability Analyzer* beschrieben. Die Abschätzung des Laufzeitverhaltens von Real-Time Euclid-Programmen wird dadurch erleichtert, daß bestimmte Sprachkonstrukte eingeschränkt oder verboten werden, z.B. werden *konstante Schleifengrenzen* verlangt und sind *Rekursionen* und *dynamische Datenstrukturen* verboten. Diese Ideen fußen im wesentlichen auf [Ehr83].

Dieser Ansatz wird in [SM92] fortgesetzt. Hier werden deadline-erhaltende Transformationen, die in polynomieller Zeit ablaufen, eingeführt, die eine Untersuchung betreffend eines möglichen Scheduling unter Berücksichtigung anderer Ressourcen erlauben. Allerdings werden die Sprachkonstrukte noch mehr eingeschränkt als in [HS91].

Da aber die Autoren hauptsächlich an Scheduling-Problemen interessiert sind, kann ihre Arbeit nicht direkt als Quelle für die Problematik

⁶Ein einfacher Ansatz, WCP-Abschätzungen für Ada zu bewerkstelligen, findet sich in [For93]. Näheres auf Seite 17, Punkt 11.

von WCP herangezogen werden. Allerdings kann aus den oben erwähnten Arbeiten geschlossen werden, daß ein Fortschritt im Bereich von WCP auch zu einem Fortschritt in ihrem Forschungsbereich führen wird.

7. In [PK89] werden Sprachkonstrukte eingeführt, die es dem Programmierer gestatten, Wissen über das tatsächliche Verhalten seines Algorithmus in sein Programm zu integrieren, das nicht mittels der standardmäßig vorhandenen Sprachkonstrukte ausgedrückt werden kann.

Diese Konstrukte sind

- *Scopes*,
- *Markers* und
- *Loop-Sequences*.

Die Information über das Verhalten des Programmes, die mit diesen Konstrukten angegeben wird, wird von einem Analysewerkzeug benutzt, um schärfere Schranken für WCP zu erhalten als das ohne sie möglich gewesen wäre.

8. *Partielle Evaluierung* wird in [NP93] benutzt, um die Laufzeit eines Programmes zur Compile-Zeit abzuschätzen. Dazu werden *Compile-Zeit-Variablen* verwendet, d.h., Variablen deren Wert zur Compile-Zeit definitiv bekannt ist. Indem man diese Werte benutzt, kann man bestimmte Programmteile vereinfachen und damit auch die Laufzeit des Programmes verbessern.

Dieser Ansatz kommt ohne Restriktionen im Bereich von Schleifen, Rekursionen oder dynamischer Datenstrukturen aus, solange Werte verwendet werden, die zur Compile-Zeit bekannt sind. Er kann sogar einige einfache Probleme bei parallel ablaufenden Prozessen und bei der Synchronisation paralleler Prozesse lösen.

Andererseits gibt es Probleme, die dieser Ansatz nicht adäquat lösen kann:

- *Generische Objekte* oder *generische Klassen* müssen für jeden Satz aktueller generischer Parameter unabhängig voneinander partiell

evaluiert werden, d.h., man kann keine allgemeine Formel für generische Objekte angeben.

- Es gibt eine gute Chance für den Programmierer, einen besseren Algorithmus zu finden, wenn er weiß, daß die Parameter zur Laufzeit bekannt sind.

Zum Beispiel kann man beim Sortieren eines Feldes, dessen Größe bekannt ist, eine Off-line-Methode finden (vgl. z.B. [Knu73b]), die besseres Laufzeitverhalten erbringen wird als die, die bei partieller Evaluierung verwendet wird. Ein anderes Beispiel betrifft die Potenzierung: Falls n bekannt ist, bevor man x^n berechnet, kann man mit einer minimalen Anzahl von Multiplikationen auskommen (vgl. z.B. [Knu81]).

Es erscheint daher sehr fragwürdig, ob alle diese Optimierungen in einem automatischen partiellen Evaluierungswerkzeug integriert werden können.

9. In [Sha89] wird die Idee eingeführt, WCP-Abschätzungen von Programmen auf Hochsprachenniveau durchzuführen. Sogenannte *Schemas* werden eingeführt, um die beste und schlechteste Ausführungszeit von Anweisungen einer Hochsprache abzuschätzen. Eine Erweiterung der Hoare-Logik (vgl. [Hoa69]) wird verwendet, um die Rechtzeitigkeit der Echtzeitprogramme zu beweisen. Dieser Ansatz kann bestimmte Echtzeitsprachkonstrukte wie Delays und Time-outs korrekt behandeln. Überraschenderweise muß der Programmierer konstante Schleifengrenzen angeben, obwohl Hoare-Logik verwendet wird.
10. Aufbauend auf [Sha89] werden in [Par93] Abschätzungen für die beste und schlechteste Ausführungszeit gewonnen, indem eine statische und dynamische Analyse der Programmpfade durchgeführt wird. Dabei werden Programmpfade mittels einfacher Ausdrücke (Regular Expression) festgelegt. Da die Bearbeitung dieser Information manchmal exponentiell viel Zeit benötigt, wird eine *Interface Definition Language* eingeführt, die eine effiziente Analyse gestattet, allerdings nicht die Ausdruckskraft von Regular Expressions besitzt.

Die angegebenen Beispiele (vgl. [Par93]) zeigen, daß scharfe Abschätzungen mit diesem Ansatz erreicht werden können. Allerdings muß der

Programmierer Obergrenzen für Schleifen angeben und die benutzte Sprache (eine Untermenge von C) ist nicht objektorientiert.

11. In [For93] wird für eine Untermenge der Sprache Ada ein Werkzeug beschrieben, das WCP-Abschätzungen durchführen kann. Die Realisierung dieser Aufgabe wird durch Einführung eines Pragmas namens `loopcount` erleichtert, mit dem für datenabhängige Schleifen eine obere Schranke für die Anzahl der Durchläufe der Schleife angegeben werden muß.

Die oben erwähnten Einschränkungen der Sprache Ada beziehen sich darauf, daß `gotos`, Rekursionen und das Anlegen dynamischer Datenstrukturen verboten sind.⁷ Wichtig ist auch, daß die Verwendung von generischen Einheiten nur bedingt möglich ist.

Das Analysewerkzeug untersucht den von einem Compiler erzeugten Objekt-Code und berechnet seine WCP aufgrund vorgegebener Laufzeiten von Maschinenbefehlen.

⁷Es gibt noch weitere Einschränkungen, die aber für unsere Thematik nicht von großer Bedeutung sind.

2 Projektziele

2.1 Allgemeine Ziele des Projektes WOOP

Ziel des Projektes WOOP ist es, wissenschaftliche Beiträge zur Abschätzung des Laufzeitverhaltens von Source-Code zu liefern. Dabei soll besonders berücksichtigt werden, daß ein objektorientiertes System zugrundeliegt, d.h., es soll im speziellen untersucht werden, welche Voraussetzungen auf programmiersprachlicher Basis dafür notwendig sind, um *passive Echtzeitobjekte* auf vernünftige Weise unterstützen zu können. Andererseits wird großes Augenmerk darauf gerichtet, daß WCP-Abschätzungen für *generische Objekte* ermöglicht werden, was eine wesentliche Neuerung innerhalb dieser Thematik darstellt. Außerdem sollen so wesentliche Sprachelemente wie unbegrenzte Schleifen⁸ und Rekursionen neben der Echtzeiteigenschaft von Programm-Code erlaubt sein. In diesem Fall sind die entsprechenden Voraussetzungen zu finden.

Abschließend soll eine Programmiersprache definiert und ein Prototyp-Compiler geschaffen werden, der alle oben erwähnten Möglichkeiten realisiert.

Auf die Konzepte Verteilung und Persistenz von Objekten wird nicht näher eingegangen, da erwartet werden kann, daß durch geeignete objektorientierte Unterstützung auf untergeordneten Systemebenen (objektorientiertes, echtzeitfähiges Netzwerk und objektorientiertes, echtzeitfähiges Filemanagement) die Möglichkeiten geschaffen werden, die Ergebnisse des Projektes WOOP direkt anzuwenden.

In den folgenden Unterkapiteln wird detailliert, wie die oben beschriebenen Ziele angestrebt werden und wie die zu behandelnden passiven Echtzeitobjekte beschaffen zu sein haben bzw. welche Algorithmen für die Implementierung passiver Echtzeitobjekte in Frage kommen.

2.2 Integration von WCP in das Objektmodell

In diesem Unterkapitel soll geklärt werden, welche Datenstrukturen und Algorithmen für die Realisierung passiver Echtzeitobjekte geeignet sind. Davor klären wir einige Begriffe aus dem Bereich der Analyse von Algorithmen und Datenstrukturen.

⁸Genauer: Schleifen ohne konstante Ober- und Untergrenzen.

Wichtige Begriffe bei der Analyse von Algorithmen und Datenstrukturen

Die Forschungsrichtung *Analyse von Algorithmen und Datenstrukturen* beschäftigt sich primär damit, das Verhalten von Algorithmen und Datenstrukturen *theoretisch*, d.h. *mathematisch* zu untersuchen. Zu diesem Thema gibt es umfangreiche Literatur, z.B. [Knu73a, Knu81, Knu73b, GKP89, Meh84, VF90, Gon84, AHU74, AHU83, Kem84].

Ziel ist es meist, das Laufzeitverhalten oder den Speicherbedarf zu analysieren. Dabei sind das „beste“, das „mittlere“ und das „schlechteste“ Verhalten von Interesse⁹.

Bei den Abschätzungen für das beste und schlechteste Verhalten versucht man, jene Eingaben zu konstruieren die zum entsprechenden Verhalten führen. Bei Abschätzungen des Mittelwertes (oder höherer Momente) ist es hingegen notwendig Annahmen über das „statistische“ Eintreffen der Daten zu treffen.

Wenn z.B. nur n unterschiedliche Daten eintreffen können, so kann man annehmen, daß alle $n!$ möglichen Eingaben gleich wahrscheinlich sind. Als anderes Beispiel sei angeführt, daß man oft annimmt, alle möglichen Ausprägungen von Datenstrukturen (etwa alle binären Bäume mit n internen Knoten) seien gleich wahrscheinlich.

Diese Annahmen führen oft zu guten Ergebnissen. Als Beispiel sei die mittlere Höhe eines binären Baumes angeführt. Ein *binärer Baum*¹⁰ ist ein Baum, dessen interne Knoten zwei Nachfolger haben. Die Höhe eines solchen Baumes ist definiert als der maximale Abstand, den ein Blatt von der Wurzel des Baumes hat. Man erhält für die mittlere Höhe eines binären Baumes mit n internen Knoten unter der Annahme, daß alle Bäume gleich wahrscheinlich sind (vgl. [FO82])

$$\sim 2\sqrt{\pi n},$$

außerdem wird in [FO82] gezeigt, daß bei geeigneter Normalisierung die Verteilung der Höhe gegen die Theta-Verteilung konvergiert.

Eine andere Möglichkeit besteht darin, eine möglichst allgemeine Verteilung über die zu untersuchenden Daten anzunehmen, um die Ergebnisse der studierten Parameter möglichst nicht einzuengen. In [BS92b, SB92, BS92a,

⁹Für Echtzeitsysteme ist natürlich hauptsächlich das schlechteste Verhalten wichtig.

¹⁰Ein binärer Baum ist nicht zu verwechseln mit einem *binären Suchbaum*, der unterschiedlich konstruiert wird.

DS92] etwa konnte unter sehr geringen Einschränkungen über die Verteilung des zeitlichen Auftretens und der Länge von Tasks die Grenzverteilung eines Parameters bestimmt werden, der die Zeit beschreibt, die verstreicht, bis in einem Echtzeitsystem die erste Verletzung einer Deadline passiert.

Abschließend kann man festhalten, daß die Analyse von Mittelwerten immer in Bezug auf die vorausgesetzte Verteilung der Eingabedaten zu sehen ist. Wenn diese Voraussetzungen in einem System nicht erfüllt sind, kann man klarerweise die Ergebnisse der theoretischen Untersuchung nicht auf dieses System anwenden. Diese Aussage gilt aber, wie gesagt, nur für Mittelwertanalysen, auf für Echtzeitsysteme relevante Analysen des „schlechtesten“ Verhaltens kann man ohne diese Bedenken zurückgreifen.

Im folgenden werden wir studieren, welche Algorithmen und Datenstrukturen überhaupt für die Verwendung in Echtzeitsystemen in Frage kommen.

2.3 Für WCP geeignete Datenstrukturen

Aus der Definition von Echtzeitsystemen folgt, daß Algorithmen, von denen keine definitive Laufzeitobergrenze angegeben werden kann, nicht für die Implementation passiver Echtzeitobjekte in Frage kommen. Wir definieren daher:

Eignung von Algorithmen und Datenstrukturen für die Verwendung in Echtzeitsystemen: Ein Algorithmus bzw. eine Datenstruktur eignet sich genau dann für die Realisierung eines passiven Echtzeitobjektes, wenn es eine Charakterisierung des Algorithmus bzw. der Datenstruktur in Form einer Menge von generischen Parametern $\mathcal{P}$ gibt, sodaß die Laufzeit jeder Methode M_i des Echtzeitobjektes nach oben mittels einer Formel $\text{gWCP}_i(\mathcal{P})$ abgeschätzt werden kann. Dabei steht **gWCP** für *generische Worst Case Performance*.

Sollten mehrere Möglichkeiten zur Realisierung von Echtzeitobjekten existieren, d.h., sollte es mehrere Algorithmen bzw. Datenstrukturen, die der obigen Definition entsprechen, geben, so kann man mit Hilfe der **gWCP**-Formeln ein passender Algorithmus bzw. eine passende Datenstruktur entsprechend der zu realisierenden Applikation ausgewählt werden.

Im folgenden werden beispielhaft einige Algorithmen und Datenstrukturen bezüglich ihrer Eignung für die Realisierung von passiven Echtzeitsystemen diskutiert.

2.3.1 Sortieralgorithmen

Sortieralgorithmen treten in irgendeiner Art und Weise in beinahe jeder nicht-trivialen Applikation auf. Daher waren sie schon lange Brennpunkt wissenschaftlicher Forschung. Es gibt viele unterschiedliche Sortieralgorithmen, deren Verhalten unterschiedlich genau untersucht worden ist.

Beschränkt man sich auf Sortieralgorithmen, die durch Vergleich je zweier Schlüssel von Datensätzen arbeiten, so kann man eine theoretische obere Schranke für die Anzahl von Vergleichen angeben, die notwendig sind, um eine Menge von n Datensätzen zu sortieren. Diese Schranke ist

$$\lceil \log_2 n! \rceil,$$

wobei $\lceil x \rceil$ für die kleinste ganze Zahl größer oder gleich x steht.

Der Parameter n charakterisiert dabei die Sortieralgorithmen genau in der von uns geforderten Art und Weise und er ist auch als generischer Parameter für ein passives Echtzeitobjekt geeignet.

Man kann also gängige Sortieralgorithmen im Hinblick auf ihre Laufzeit überprüfen, indem man ihre **gWCP** mit obiger Schranke vergleicht. Dabei kann folgende Formel hilfreich sein (vgl. z.B. [GKP89]):

$$\log_2 n! = n \log_2 n - \frac{n}{\log 2} + \frac{\log_2 n}{2} + \frac{1}{2} \log_2 2\pi + O(1/n)$$

Wir können also nicht erwarten, ein Sortierverfahren zu finden, das mit weniger als ungefähr $n \log_2 n$ Vergleichen auskommt. Die Frage ist, ob wir einen Algorithmus finden können, dessen **gWCP** an diese untere Schranke herankommt. Ein Vergleich einiger bekannter Verfahren (vgl. [Knu73b]) zeigt folgendes Bild:

Sortierverfahren	Erwartungswert der Anzahl von Vergleichen	obere Schranke
Sortieren durch Einfügen	$O(n^2)$	$O(n^2)$
Quicksort	$O(n \log n)$	$O(n^2)$
Heapsort	$O(n \log n)$	$O(n \log n)$

In dieser Tabelle zeigt einzig und allein Heapsort, daß es möglich ist, ein Sortiervorgehen zu finden, dessen **gWCP** bis auf einen konstanten Faktor mit der theoretischen Untergrenze übereinstimmt. Eine exaktere Analyse von Heapsort, vor allem ein Vergleich des Erwartungswertes mit dem von Quicksort war bis vor kurzem ausstehend. Erst in [SS93] wurde eine exakte Analyse vorgestellt. Diese ergab, daß das Verhalten im Mittel asymptotisch mit dem des schlechtesten Falles übereinstimmt, beide Werte sind $\sim 2n \log_2 n$. Im Vergleich dazu liegt Quicksort im Mittel bei $\sim 2n \log n \approx 1.39n \log_2 n$. Mittelwerte sind allerdings für Echtzeitsysteme von geringerer Bedeutung, hier zählt einzig und allein das Verhalten im schlechtesten Fall. In diesem Fall heißt das, daß für großes n Heapsort besser sein wird als die anderen, oben angeführten Sortiervorgehen. Für kleines n können allerdings die anderen, oben angegebenen Vorgehen durchaus besseres **gWCP**-Verhalten haben (vgl. [Knu73b]).

2.3.2 Suchbäume

Bei Sortialgorithmen geht man davon aus, daß alle zu behandelnden Datensätze bereits vor dem Sortieren bereit stehen. Bei Suchbäumen werden die sukzessiv anfallenden Datensätze so abgespeichert, daß ein nachträgliches Manipulieren der Daten (z.B. Löschen oder das Suchen des größten Elements) und das Einfügen selbst in vertretbarer Zeit erfolgen.

Für alle interessanten Operationen ist dabei die Höhe des Baumes von entscheidender Bedeutung. Betrachten wir in diesem Zusammenhang die einfachste Datenstruktur, den *binären Suchbaum*, der aus r Datensätzen besteht. In diesem Fall ist die obere Schranke für die Höhe des Baumes durch r gegeben (vgl. [Knu73b, Meh84]). Der Erwartungswert für die Höhe liegt allerdings bei $\sim c \log r$, wobei $c = 4.311070 \dots$ die Wurzel von $(2e/c)^c = e$ bezeichnet, die ≥ 2 ist (vgl. [Dev86]).

Im Gegensatz zum binären Suchbaum, dessen Höhe nur linear in r beschränkt ist, zeichnet sich ein *2-3-Baum* dadurch aus, daß seine Höhe logarithmisch beschränkt ist. Ein 2-3-Baum (vgl. [Meh84, Knu73b]) ist definiert dadurch, daß

1. sich alle Datensätze in den Blättern des Baumes aufhalten,
2. alle Blätter denselben Abstand zur Wurzel des Baumes (dieselbe Höhe) haben und

3. die internen Knoten entweder zwei oder drei Nachfolger haben.

Damit ergibt sich, daß die Höhe des Baumes nach oben beschränkt ist durch $\log_2 r$, falls sich r Datensätze im Baum befinden. Außerdem ergibt sich für geeignete (und geschickte) Implementierungen aller interessanten Operationen wie *Einfügen*, *Löschen* und *Suchen* eine WCP-Abschätzung der Form

$$C \log_2 r.$$

Der Vollständigkeit halber sei noch angeführt, daß die Untersuchung der mittleren Höhe eines 2-3-Baumes ein noch ungelöstes Problem darstellt, lediglich die Anzahl von unterschiedlichen 2-3-Bäumen, wenn r Datensätze eingefügt werden, konnte ermittelt werden. Diese Anzahl erfüllt (vgl. [Odl82])

$$\sim \frac{\varphi^r}{r} W(\log r),$$

wobei $\varphi = (1 + \sqrt{5})/2$ der goldene Schnitt ist und $W(x)$ eine stetige und periodische Funktion bezeichnet. Numerische Berechnungen lassen für die mittlere Höhe

$$\sim \log r V(\log r)$$

erwarten, wobei $V(x)$ für eine periodische Funktion steht. Das ist jedoch reine Spekulation und durch keine theoretische Berechnung gestützt.

Im Unterschied zu den Sortieralgorithmen, bei denen der Parameter n ausreichte, um sowohl das Verhalten in der Zeit und den Speicherbedarf zu charakterisieren, benötigen wir bei den Suchbäumen zusätzlich zum Parameter r , der aktuellen Anzahl von im Baum befindlichen Datensätzen, noch einen Parameter, um für Echtzeitanwendungen Abschätzungen des zeitlichen Verhaltens und des Speicherbedarfs durchführen zu können.

Wir definieren daher den Parameter n als obere Schranke für die Anzahl von Datensätzen. Folglich gilt $r \leq n$. Zum Beispiel erhalten wir dann als Schranke für die Höhe eines 2-3-Baumes

$$\log_2 n$$

und als gWCP-Abschätzung für die oben genannten Operationen

$$C \log_2 n.$$

Klarerweise bleiben aber trotzdem die obigen Abschätzungen, bei denen die aktuelle Anzahl von Datensätzen r verwendet wurde, aufrecht.

2.4 Die aWCP-Abschätzung

Die Betrachtungen am Ende des vorigen Kapitels führen uns dazu, eine neue Art von WCP-Abschätzung zu definieren, nämlich die *aktuelle* WCP, abgekürzt aWCP. Eine Formel einer aWCP-Abschätzung kann sich nun auch auf den internen Zustand eines passiven Echtzeitobjektes (z.B. die Anzahl der momentan vorhandenen Datensätze) beziehen. Wir definieren:

Eignung eines Algorithmus oder einer Datenstruktur für eine aktuelle WCP-Abschätzung: Ein Algorithmus bzw. eine Datenstruktur eignet sich genau dann für eine aktuelle WCP-Abschätzung, wenn es eine Charakterisierung des Algorithmus bzw. der Datenstruktur in Form einer Menge von generischen Parametern $\mathcal{P}$ und eine Menge $\mathcal{A}$ von sogenannten *Zustandsparemetern* gibt, die den aktuellen internen Zustand des Echtzeitobjektes beschreibt, sodaß die Laufzeit jeder Methode M_i des Echtzeitobjektes nach oben mittels einer Formel $\text{gWCP}_i(\mathcal{P})$ und mittels einer Formel $\text{aWCP}_i(\mathcal{P}, \mathcal{A})$ abgeschätzt werden kann.

Eine aWCP-Abschätzung kann natürlich nur dann genutzt werden, wenn man den entsprechenden internen Zustand des Objektes auch „außen“ weiß. Als Beispiel betrachte man den Unterschied zwischen gWCP und aWCP im Falle, daß ein Benutzer eines 2-3-Baumes n aufeinander folgende Datensätze einfügen möchte. Man erhält für gWCP

$$Cn \log_2 n,$$

für aWCP allerdings

$$C \sum_{r=1}^n \log_2 r = C \left(n \log_2 n - \frac{n}{\log 2} + \frac{\log_2 n}{2} + \frac{1}{2} \log_2 2\pi + O(1/n) \right)$$

für große Werte von n .

Offensichtlich ist die durch die aWCP erhaltene Abschätzung schärfer als die gWCP-Abschätzung, wenn n groß genug ist.

Abschließend soll noch bemerkt werden, daß aWCP-Abschätzungen eigentlich dem Prinzip des *information hidings* widersprechen. Dies ist jedoch in den Augen des Autors schon deswegen tolerierbar, weil Laufzeitabschätzungen von Methoden eines Objektes ohnehin diesem Prinzip entgegen laufen, da sie Implementierungsdetails „nach außen“ verraten.

Zum Beispiel kann man aus der Tatsache, daß die Laufzeit gewisser Operationen, die auf einem Baum definiert sind, nach oben mit $C \log n$ abgeschätzt werden können, schließen, daß sicher kein binärer Suchbaum zur Implementierung verwendet worden sein kann.

Da dies aus jeder Art von Laufzeitabschätzung folgt, kann man eigentlich davon ausgehen, daß Echtzeitsysteme *per definitionem* nicht dem *information hiding* entsprechen können.

2.5 Eine Hierarchie von WCP-Abschätzungen

Zusammenfassend haben wir also eine Hierarchie von drei WCP-Abschätzungen erhalten:

- | | |
|------|--|
| WCP | die Abschätzung der Laufzeit durch einen <i>konstanten Wert</i> |
| gWCP | die Abschätzung der Laufzeit durch eine <i>Formel</i> , die <i>generische Parameter</i> des Echtzeitobjektes beinhaltet |
| aWCP | die Abschätzung der Laufzeit durch eine <i>Formel</i> , die <i>generische Parameter</i> des Echtzeitobjektes und <i>Zustandsparameter</i> beinhaltet |

Im allgemeinen gilt

$$\text{tatsächliche Laufzeit} \leq \text{aWCP}(\dots) \leq \text{gWCP}(\dots) \leq \text{WCP},$$

wobei (...) bedeutet, daß die entsprechenden Parameter der Formeln mit aktuellen Werten versehen werden müssen.

Obige Ungleichung kann folgendermaßen interpretiert werden: Je mehr Information über den internen Zustand eines Echtzeitobjektes preis gegeben wird, desto bessere Laufzeitabschätzungen sind möglich.

2.6 Voraussetzungen an eine Programmiersprache

Eine objektorientierte Programmiersprache, die passive Echtzeitobjekte in der oben diskutierten Form unterstützen soll, muß natürlich gewisse Eigenschaften aufweisen. In diesem Kapitel wird abrißhaft dargestellt, welche Sprachkonstrukte dafür notwendig sind; eine genauere Untersuchung wird innerhalb des Projektes WOOP unternommen werden.

2.6.1 Generische passive Echtzeitobjekte

Wesentlich für ein generisches passives Echtzeitobjekt sind natürlich die *generischen Parameter*. Auf Grund unserer bisherigen Abhandlung ergibt sich die Notwendigkeit für zwei unterschiedliche Parameterarten:

1. Parameter zur Spezifikation der gWCP und
2. Parameter zur Spezifikation der aWCP.

Allgemein gesprochen sind solche generischen Parameter Bilder von *Funktionen des internen Zustandes des Objektes*.

2.6.2 Sprachkonstrukte für den Spezifikationsteil von passiven Echtzeitobjekten

Für jede Methode eines passiven Echtzeitobjektes muß die Möglichkeit bestehen, WCP-, gWCP- und/oder aWCP-Abschätzungen in Form von Funktionen der generischen Parameter zu spezifizieren. Die Ausdrücke für diese Funktionen beziehen ihre Teile aus drei Quellen:

1. den generischen Parametern und den Zustandsparametern,
2. aus Gegebenheiten der Implementation der Methode, die algorithmusabhängig aber maschinenunabhängig sind, und
3. aus maschinenabhängigen Teilen der Implementierung, d.h., im wesentlichen Ausführungszeiten von Maschinenbefehlen.

Die ersten beiden Teile stellen kein großes Problem dar, da sie maschinenunabhängig sind. Der maschinenabhängige Teil der Ausdrücke allerdings bedeutet, daß grundsätzlich die Möglichkeit bestehen muß, für jeden Prozessortyp getrennte WCP-Abschätzungen angeben zu können. Dies ergibt sich auch daraus, daß Echtzeitsysteme meist als verteiltes System implementiert werden, das auf heterogenen Prozessoren basiert.

aWCP-Abschätzungen stellen insofern ein Problem dar, als sie implizieren, daß mit jeder Methode spezifiziert werden muß, wie durch ihre Ausführung die Zustandsparameter verändert werden. So muß etwa die Einfügeoperation für einen 2-3-Baum folgende Spezifikation adjungiert haben:

$$r \mapsto r + 1, \text{ falls } r < n \text{ und } r \mapsto n \text{ sonst.}$$

Diese Information kann dann beispielsweise von einem Schedulability Analyzer genutzt werden, um statische Laufzeitabschätzungen des verwendeten Programmes zu verbessern.

2.6.3 Sprachkonstrukte für den Implementationsteil von passiven Echtzeitobjekten

Generell ist zu sagen, daß die Unterteilung in Spezifikations- und Implementationsteil von Objekten die notwendige Anzahl von Übersetzungen herabsetzt, wenn Änderungen nur im Implementationsteil eines Objektes notwendig sind. Um diese Entkopplung bei WCP-Abschätzungen zu ermöglichen, sollte ein passender Compiler daher entsprechend der folgenden Tabelle reagieren, wenn die für den Implementationsteil gewonnene Abschätzung nicht mit der des Spezifikationsteils übereinstimmen.

Bedingung	Reaktion
$WCP(\text{Spez.}) = WCP(\text{Imp.})$	keine
$WCP(\text{Spez.}) > WCP(\text{Imp.})$	passende Warnung
$WCP(\text{Spec.}) < WCP(\text{Imp.})$	Fehlermeldung

Dabei steht $WCP(.)$ für entsprechende Paare von WCP-, gWCP- und aWCP-Abschätzungen und bei einer *Fehlermeldung* entsteht definitiv kein ablauffähiges Programm.

Die für die Implementation kritischen Sprachkonstrukte, die von den meisten Echtzeitsprachen der Einfachheit halber verboten sind, sind *datenabhängige Schleifen* und *Rekursionen*.

Datenabhängige Schleifen, d.h. Schleifen, deren Grenzen nicht zur Übersetzungszeit feststehen, können mit dem in WOOP zu realisierenden Konzept der gWCP- und aWCP-Abschätzungen insofern behandelt werden, als Schleifen, deren Grenzen nur von generischen Parametern oder von Zustandsparametern abhängen, erlaubt werden können.

Inwieweit Rekursionen unter Berücksichtigung von aWCP-Abschätzungen erlaubt werden können, wird Bestandteil des Projektes WOOP sein.

Im übrigen können Rekursionen prinzipiell immer in äquivalentem Code, der ohne Rekursionen auskommt, umgewandelt werden. Das bedeutet, daß

Rekursionen nur dann erlaubt werden können, wenn die Anzahl der rekursiven Aufrufe in irgendeiner Art und Weise nach oben abgeschätzt werden kann. Das ist zum Beispiel bei einfachen linearen Rekursion, wie etwa $n! = n * (n - 1)!$, leicht durchführbar, bei nicht-linearen Rekursionen allerdings scheint eine „gute“, d.h. nicht zu hoch gegriffene Abschätzung der Anzahl der rekursiven Aufrufe nicht so leicht durchführbar. Als Beispiel für den letzteren Fall sei

$$Q(n) = Q(n - Q(n - 1)) + Q(n - Q(n - 2)), Q(1) = Q(2) = 1$$

angeführt (vgl. [Hof79]).

Abschließend sei bemerkt, daß bei den notwendigen Abschätzungen der Anzahl von rekursiven Aufrufen nicht nur die generischen Parameter und die Zustandsparameter eingehen werden, sondern auch aktuelle Parameter der rekursiven Prozedur.

2.6.4 Objektorientierte Elemente und Relationen und ihr Bezug zu WCP-Abschätzungen

Den Elementen Abstraktion, Datenkapselung und Modularität entsprechen WCP-Abschätzungen in folgender Weise:

- Die zu den Methoden eines Objektes adjungierten WCP-Abschätzungen ermöglichen es dem Benutzer eines passiven Echtzeitobjektes, bezüglich des Laufzeitverhaltens von der Implementierung des Objektes zu abstrahieren.
- Die für das Laufzeitverhalten unwesentlichen Details der Implementierung, d.h. die Frage, wie das spezifizierte Laufzeitverhalten realisiert wird, wird verborgen. Allerdings kann dem *information hiding*, wie bereits oben erläutert, nicht ganz Rechnung getragen werden.
- Durch die Spezifikation von WCP-Abschätzungen wird die Kopplung zwischen einzelnen Modulen verringert und die Modularität des Gesamtsystems wird durch die Trennung in einen Spezifikationsteil und einen Implementationsteil erleichtert.

Den Themenbereich „Hierarchie“ werden wir bei der Vererbungsrelation behandeln und vom Typkonzept des Objektmodells sind WCP-Abschätzungen unabhängig zu sehen.

Verwendungsrelation und WCP-Abschätzungen

Der Compiler einer für WCP-Abschätzungen geeigneten Sprache muß gewährleisten, daß ein passives Echtzeitobjekt nur Methoden eines aktiven oder passiven Echtzeitobjektes verwendet. Ein Programm, das ein passives Echtzeitobjekt beinhaltet, das Methoden eines nicht echtzeitfähigen Objektes verwendet, ist ungültig, da dann keine Aussagen über die Echtzeitfähigkeit des Programmes gemacht werden können.

Beinhaltungsrelation und WCP-Abschätzungen

Ein passives Echtzeitobjekt darf nur Echtzeitobjekte beinhalten.

Vererbungsrelation und WCP-Abschätzungen

Bei der Vererbung ist darauf zu achten, daß Methoden, deren Implementierung im Zuge der Vererbung geändert wird, der geerbten Spezifikation der WCP-Abschätzung entsprechen. Natürlich müssen alle neu hinzugekommenen Methoden ebenfalls eine Spezifikation einer WCP-Abschätzung besitzen.

Verwendungsrelation und WCP-Abschätzungen

Eine passive Echtzeitklasse darf nur andere Echtzeitklassen verwenden.

Instanziierungsrelation und WCP-Abschätzungen

Als Parameter eines generischen passiven Echtzeitobjektes kommen nur Echtzeitobjekte in Frage.

2.7 Detaillierte Ziele des Projektes WOOP

In diesem Abschnitt werden die Ziele des Projektes WOOP genauer ausgeführt. Dabei erfolgt gleichzeitig eine Einteilung des Projektes in einzelne Arbeitspakete, deren Durchführung in Kapitel 2.8 dargelegt wird.

Eine grobe Unterscheidung der Ziele des Projektes kann insofern durchgeführt werden, als zwischen Aufgaben, deren Ziel es ist, Konzepte zu entwerfen, und Aufgaben, deren Ziel es ist, diese Konzepte in die Praxis umzusetzen, unterschieden werden kann.

2.7.1 Erarbeitung von Konzepten

Als Ziel bei der Erarbeitung der Konzepte kann gelten, daß für jeden der folgenden Punkte eine wissenschaftliche Publikation erarbeitet wird:

1. Das hier vorgestellte Konzept der Hierarchie von WCP-Abschätzungen ist auszuarbeiten.
2. Die Sprachkonstrukte für den Spezifikationsteil von passiven Echtzeitobjekten sind zu präzisieren.

Das gilt vor allem für die Syntax der WCP-Formeln und der Formeln, mit denen die Änderung der Zustandsparameter während der Ausführung einer Methode beschrieben werden soll.

3. Die Sprachkonstrukte für den Implementationsteil von passiven Echtzeitobjekten sind zu präzisieren.

Das betrifft vor allem die Frage nach datenabhängigen Schleifen und Rekursionen.

2.7.2 Praktische Umsetzung der Konzepte

Bei der praktischen Umsetzung der erarbeiteten Konzepte gilt als generelles Ziel der Entwurf einer Sprache und einer Prototyp-Implementation eines entsprechenden Compilers. Die dabei gewonnenen Erkenntnisse sollen in Form einer wissenschaftlichen Publikation veröffentlicht werden.

Eine Einteilung der gerade angesprochenen Entwurfs- und Implementationstätigkeit sieht folgendermaßen aus:

1. Entwurf von Syntax und Semantik einer passenden Sprache, die WCP-Abschätzungen ermöglicht
2. Implementation eines Compilers für die unter Punkt 1 entworfene Sprache
3. Entwurf und Implementation eines Werkzeugs zur Abschätzung der Laufzeiten der Anweisungen der Sprache (ohne Schleifen und Rekursionen)

2.8 Projektdurchführung

In diesem Kapitel werden die im vorigen Kapitel aufgestellten Arbeitspakete in eine zeitliche Reihenfolge gebracht. In diesem und in den folgenden Kapiteln werden wir uns auf die konzeptuellen Arbeitspakete in der Form K1 bis K3 und auf die Arbeitspakete, die der praktischen Realisierung dienen, in der Form P1 bis P3 beziehen.

Die notwendigen Konzepte sollen im Laufe des ersten Jahres des Projektes erarbeitet werden. Dabei ist es durchaus möglich, an allen Arbeitspaketen (K1-K3) gleichzeitig zu arbeiten.

Nach Abschluß dieser Arbeitspakete sollen die Arbeitspakete P1 bis P3 in Angriff genommen werden. Als Beginn des Arbeitspaketes P1 ist das letzte Quartal des ersten Jahres, als Beginn der Arbeitspakete P2 und P3 ist der Beginn des zweiten Jahres geplant. Für jedes der praktischen Arbeitspakete ist ein Diplomand vorgesehen. Als Dauer für jede der Diplomarbeiten wird ein Jahr veranschlagt.

Für die Implementierung des Compilers wird entweder die Sprache C mit dem GNU C Compiler (*gcc*) und *bison* (der *GNU yacc*) verwendet werden oder Ada9X mit dem GNU Ada Translator (*gnat*) und *ayacc* (ein *yacc*-ähnliches Werkzeug, das Ada-Code erzeugt). Prinzipiell wird Ada9X bevorzugt, allerdings hängt die endgültige Entscheidung von der Qualität des Ada9X-Compilers (*gnat*) ab, dessen Fertigstellungstermin Dezember 1993 ist.

Gründe, warum GNU-Compiler (*gcc* oder *gnat*) und GNU-Werkzeuge verwendet werden sollen, sind, daß

- beide Compiler auf sehr vielen unterschiedlichen Zielrechnern ablaufen,
- für GNU-Produkte der Source-Code zur Verfügung steht,
- daher für das in Punkt P3 erwähnte Werkzeug der Source-Code von *gcc* (und *gas*, dem GNU Assembler) geeignet modifiziert werden kann und nicht zuletzt daß
- sie unentgeltlich zur Verfügung stehen und daher für das Projekt WOOP keinerlei Software-Kosten anfallen.

2.9 Mögliche Implikationen und Folgeaktivitäten

In diesem Kapitel soll aufgezeigt werden, welche Forschungsrichtungen aufbauend auf die im Projekt WOOP erhaltenen Ergebnisse möglich sind.

1. Die Ergebnisse des Projektes WOOP können zum Teil sicher auch für die Abschätzung von Code-Blöcken aktiver Echtzeitobjekte verwendet werden. In diesem Fall wäre also Unterstützung für einen Scheduler gewährleistet.
2. Die Ergebnisse des Projektes WOOP gemeinsam mit den in Punkt 1 angesprochenen könnten in einen *Schedulability Analyzer* integriert werden.
3. Die im Projekt WOOP entwickelte Sprache kann als Basis für die Entwicklung eines Echtzeitbetriebssystems dienen, das die Konzepte aktiver und passiver Echtzeitobjekte unterstützt.
4. Durch die im Projekt verwendeten GNU-Werkzeuge ist eine gute Verbreitung der entwickelten Sprache schon wegen der breiten Hardware-Plattform gesichert. Darüberhinaus ergibt sich durch die GNU-Werkzeuge eine leicht Verwendbarkeit in heterogenen Netzwerken, wie sie für Echtzeitsysteme typisch sind.

Literatur

- [Ada83] ANSI/MIL-STD 1815 A. *Reference manual for the Ada programming language*, 1983.
- [Ada93] Ada 9X Mapping/Revision Team. *Ada 9X Reference Manual (Draft), Version 3.0*, 1993.
- [AHU74] A. V. Aho, J. E. Hopcroft und J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.
- [AHU83] A. V. Aho, J. E. Hopcroft und J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.
- [AK86] B. Auernheimer und R. A. Kemmerer. RT-ASLAN: A specification language for real-time systems. *IEEE Trans. Software Engineering*, 12(9):879–889, 1986.
- [BDG⁺88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales und D. A. Moon. Common Lisp Object System specification X3J13 Document 88-002R. *SIGPLAN Notices*, 23, September 1988.
- [Bli92] J. Blieberger. Real-time concepts and the attributed object model. (to appear), 1992.
- [Boo91] G. Booch. *Object-oriented design with applications*. Benjamin/Cummings, Redwood City, CA, 1991.
- [BS92a] J. Blieberger und U. Schmid. FCFS scheduling in a hard real-time environment under rush-hour conditions. *BIT*, 32(3):370–383, 1992.
- [BS92b] J. Blieberger und U. Schmid. Preemptive LCFS scheduling in hard real-time applications. *Performance Evaluation*, 15(3):203–215, 1992.
- [Bur88] A. Burns. *Programming in occam 2*. Addison-Wesley, Reading, MA, 1988.

- [BW89] A. Burns und A. Wellings. *Real-time systems and their programming languages*. Addison-Wesley, Reading, MA, 1989.
- [Con87] R. Conn. *The Ada Software Repository and the Defense Data Network (A Resource Handbook)*. Zoetrope, New York, 1987.
- [Dev86] L. Devroye. A note on the height of binary search trees. *Journal of ACM*, 33:489–498, 1986.
- [DGU91] J. P. Diaz-Gonzalez und J. E. Urban. Language aspects of ENVISAGER: An object-oriented environment for the specification of real-time systems. *Comput. Lang.*, 16(1):19–37, 1991.
- [DIN82] DIN 66 253, Teil 2, Beuth Verlag, Berlin. *Programmiersprache PEARL, Full PEARL*, 1982.
- [DIN85] DIN 44 300, Teil 9, Beuth Verlag, Berlin. *Informationsverarbeitung, Begriffe, Verarbeitungsabläufe*, 1985.
- [DIN89] DIN 66 253, Teil 3, Beuth Verlag, Berlin. *Programmiersprache PEARL, Mehrrechner-PEARL*, 1989.
- [DS92] M. Drmota und U. Schmid. Exponential limiting distributions in queueing systems with deadlines. *SIAM J. on Appl. Math.* (to appear), 1992.
- [Ehr83] W. Ehrenberger. Softwarezuverlässigkeit und Programmiersprache. *Regelungstechnische Praxis*, 25(1):24–29, 1983.
- [ES90] M. A. Ellis und B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.
- [FO82] P. Flajolet und A. M. Odlyzko. The average height of binary trees and other simple trees. *J. Comput. System Sci.*, 25:171–213, 1982.
- [For93] C. Forsyth. Hard real-time operating system kernel study: Using the worst-case execution analyser. Technical report, York Software Engineering Limited, 1993.

- [GKP89] R. L. Graham, D. E. Knuth und O. Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading, MA, 1989.
- [Gon84] G. H. Gonnet. *Handbook of Algorithms and Data Structures*. Addison-Wesley, London, 1984.
- [GR89] A. Goldberg und D. Robson. *Smalltalk-80*. Addison-Wesley, Reading, MA, 1989.
- [GR91] N. Gehani und K. Ramamritham. Real-time Concurrent C: A language for programming dynamic real-time systems. *The Journal of Real-Time Systems*, 3:377–405, 1991.
- [Haa81] V. H. Haase. Real-time behavior of programs. *IEEE Trans. Software Engineering*, 7(5):494–501, 1981.
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of ACM*, 12:576–580, 1969.
- [Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ, 1985.
- [Hof79] D. R. Hofstadter. *Gödel, Escher, Bach - ein endloses geflochtenes Band*. Klett Verlag, Stuttgart, 1979.
- [Hol92] J. E. Hollingsworth. *Software Component Design-for-Reuse: A Language-Independent Discipline Applied to Ada*. PhD thesis, Ohio State University, 1992.
- [HS91] W. A. Halang und A. D. Stoyenko. *Constructing predictable real time systems*. Kluwer Academic Publishers, Boston, 1991.
- [ITM90] Y. Ishikawa, H. Tokuda und C. W. Mercer. Object-oriented real-time language design: Constructs for timing constraints. In *ECOOP/OOPSLA '90 Proceedings*, pages 289–298, October 1990.
- [JM86] F. Jahanian und A. K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Trans. Software Engineering*, 12(9):890–904, 1986.

- [Jon89] H. Jonkers. An introduction to COLD-K. In M. Wirsing und J. Bergstra, editors, *Algebraic Methods: Theory, Tools, and Applications*, volume 394 of *Lecture Notes in Computer Science*, pages 139–205. Springer, Berlin, 1989.
- [Kem84] R. Kemp. *Fundamentals of the Average Case Analysis of Particular Algorithms*. Teubner Verlag, Stuttgart, 1984.
- [Knu73a] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., second edition, 1973.
- [Knu73b] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., 1973.
- [Knu81] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., second edition, 1981.
- [KS86] E. Kligerman und A. D. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, 12(9):941–949, 1986.
- [LL73] C. Liu und J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.
- [LY86] D. W. Leinbaugh und M.-R. Yamini. Guaranteed response times in a distributed hard-real-time environment. *IEEE Trans. Software Engineering*, 12(12):1139–1144, 1986.
- [MACT89] A. K. Mok, P. Amerasinghe, M. Chen und K. Tantisirivat. Evaluating tight execution time bounds of programs by annotations. In *Proc. IEEE Workshop on Real-Time Operating Systems and Software*, pages 74–80, 1989.
- [Meh84] K. Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, Berlin, 1984.

- [NP93] V. Nirkhe und W. Pugh. A partial evaluator for the Maruti hard real-time system. *The Journal of Real-Time Systems*, 5:13–30, 1993.
- [Odl82] A. M. Odlyzko. Periodic oscillations of coefficients of power series that satisfy functional equations. *Adv. Math.*, 44:180–205, 1982.
- [Ost92] J. S. Ostroff. A verifier for real-time properties. *Journal of Real-Time Systems*, 4:5–35, 1992.
- [Par93] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *The Journal of Real-Time Systems*, 5:31–62, 1993.
- [PK89] P. Puschner und C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1:159–176, 1989.
- [RBP⁺91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy und W. Lorenson. *Object-Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.
- [SB92] U. Schmid und J. Blieberger. Some investigations on FCFS scheduling in hard real-time applications. *Journal of Computer and System Sciences*, 45(3):493–512, 1992.
- [Sch92] D. J. Scholefield. *A Refinement Calculus for Real-Time Systems*. PhD thesis, University of York, 1992.
- [Sha89] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, 1989.
- [SM92] A. D. Stoyenko und T. J. Marlowe. Polynomial-time transformations and schedulability analysis of parallel real-time programs with restricted resource contention. *The Journal of Real-Time Systems*, 4:307–329, 1992.
- [SS93] R. Schaffer und R. Sedgewick. The analysis of heapsort. *Journal of Algorithms*, 15:76–100, 1993.

- [ST88] D. Sannella und A. Tarlecki. Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Informatica*, 25:233–281, 1988.
- [ST89] D. Sannella und A. Tarlecki. Toward formal development of ML programs: foundations and methodology. In J. Diaz und F. Orejas, editors, *Proc. TAPSOFT 89, Vol. 2*, volume 352 of *Lecture Notes in Computer Science*, pages 375–389, Berlin, 1989. Springer.
- [Str86] B. Stroustrup. *The C++ programming language*. Addison-Wesley, Reading, MA, 1986.
- [VF90] J. S. Vitter und P. Flajolet. Average-case analysis of algorithms and data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 431–524. North-Holland, 1990.
- [Weg91] P. Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1:7–87, 1991.
- [Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 675–788. North-Holland, 1990.
- [WW85] W. Werum und H. Windauer. *Introduction to PEARL: Process and Experiment Automation Realtime Language*. Program Applied Informatics. Vieweg, Braunschweig, 1985.
- [XP93] J. Xu und D. L. Parnas. On satisfying timing constraints in hard real-time systems. *IEEE Trans. Software Engineering*, 19(1):70–84, 1993.

3 Angaben zur Forschungsstätte und zu beantragten Förderungsmitteln

3.1 Ort der Forschung

Das beantragte Projekt WOOP soll an der TU Wien, am *Institut für Automation*, Abteilung für *Automatisierungssysteme* (Leiter: *O.Prof. Dr. G.-H. Schildt*) durchgeführt werden. Die Abteilung (1 Professor, 5 Univ. Assistenten, 1 Techniker, 1 Systemadministrator, 1 Sekretärin) wurde 1988 gegründet und befindet sich in der *Treitlstraße 3/4, 1040 Wien*.

Das Projekt liegt in einer der primären Forschungsrichtungen der Abteilung. Da der Projektleiter als Univ. Ass. an dem genannten Institut tätig ist, können die umfangreichen Ressourcen der Forschungsstätte intensiv genutzt werden. Dies betrifft unter anderem:

Workstations und PCs: Es können mehrere der an der Forschungsstätte vorhandenen HP/Apollo-Workstations und PCs für das Projekt verwendet werden.

Personal: Hier ist vor allem die Hilfestellung von anderen Institutsmitgliedern (Univ. Ass., Techniker, Systemadministrator, Sekretärin) zu erwarten. Auch die Delegation von einfacheren Implementierungstätigkeiten an Studenten im Rahmen von Praktika sei hier erwähnt.

Räumlichkeiten: Für alle Mitarbeiter des Projektes stehen geeignete Räumlichkeiten zur Verfügung.

Sonstiges: Ebenfalls zu erwähnen ist die Nutzung der Instituts- und TU-Bibliothek, sowie bestehende Kontakte zu anderen Forschungsstellen, z.B. zu anderen TU-Instituten oder außeruniversitären Einrichtungen wie dem Forschungszentrum Seibersdorf.

3.2 Personalressourcen

Dieser Abschnitt enthält eine Aufstellung der Mitarbeiter an dem oben beschriebenen Forschungsprojekt WOOP. Die einzelnen Aufgabengebiete wurden in den Kapiteln 2.7 (Seite 29) und 2.8 (Seite 31) ausführlich behandelt.

3.3 Vorhandenes Personal

Als direkter Mitarbeiter am Projekt WOOP ist ein Univ. Ass. zu nennen:

Univ. Ass. Dr. Johann Blieberger (Antragsteller, Projektleiter): Die primäre Aufgabe von Dr. Blieberger ist die Organisation und Koordination des Gesamtprojektes. Das beinhaltet unter anderem die Erstellung einer Grobspezifikation der im Rahmen des Projektes zu erarbeitenden Konzepte (K1 - K3) und die Integration und Abstimmung der einzelnen Projektteile. Darüberhinaus ist die Betreuung der am Projekt mitarbeitenden Diplomanden (P1 - P3) zu erwähnen.

Das Projekt WOOP liegt in einer primären Forschungsrichtung des Antragstellers. Deswegen ist vor allem eine befriedigende Beantwortung der wissenschaftlichen Fragestellungen für die Forschungstätigkeit von großer Bedeutung.

3.4 Beantragtes Personal

Aus Förderungsmitteln wären insgesamt ein Vertragsassistent und drei Diplomanden zu beschäftigen.

Da die zu erwartenden Diplomarbeiten im Vergleich zu anderen Diplomarbeiten sowohl vom Schwierigkeitsgrad als auch vom Zeitaufwand außergewöhnlichen Einsatz erfordern, erscheint in allen drei Fällen eine *Forschungsbeihilfe bis zur Graduation* in maximaler Höhe (dzt. öS 5.000,-/Monat) angemessen.

Dies erscheint auch deswegen motivationsfördernd, weil die ungewöhnlich lange Dauer der Diplomarbeiten einen frühen Eintritt in das hervorragend bezahlte Berufsleben verhindert.

VA: ¹¹ Die Aufgaben von VA finden sich einerseits in einer unterstützenden Mitarbeit an der Erarbeitung der Konzepte (K1 - K3) und andererseits bei der Definition der Syntax und der Semantik der zu entwerfenden Sprache (P1) und beim Entwurf des in Punkt 3 beschriebenen Werkzeuges.

¹¹Der Name des vorläufig mit VA bezeichneten Vertragsassistenten wird erst mit dem Start des Projektes festgelegt werden.

N.N.¹² Die Aufgabe dieses Diplomanden ist die Erarbeitung der Syntax und der Semantik der zu realisierenden Sprache (P1). Dabei ist intensive Unterstützung durch Dr. Blieberger und VA gewährleistet.

N.N.¹³ Die Aufgabe dieses Diplomanden ist die Implementierung eines Compilers für die unter dem vorigen Punkt entworfene Sprache (P2). Wieder ist dabei Unterstützung durch Dr. Blieberger und VA gewährleistet.

N.N.¹⁴ Die Aufgabe dieses Diplomanden ist Entwurf und Implementierung des in Punkt 3 beschriebenen Werkzeuges. Wieder ist dabei Unterstützung durch Dr. Blieberger und VA gewährleistet.

3.5 Vorhandene Sachausstattung

An der Forschungsstätte sind mehrere HP/Apollo-Workstations und mehrere PCs vorhanden, die für das Projekt genutzt werden können. Allerdings soll auf den PCs, die nur im Wintersemester verwendet werden können, da sie im Sommersemester zur Gänze für die Laborübung „Prozeßautomatisierung“ benötigt werden, das Betriebssystem Linux installiert werden. Das in der oben genannten Laborübung eingesetzte DOS kann nicht verwendet werden, da der im Projekt WOOP eingesetzte Compiler gnat nicht unter DOS läuft.

3.6 Beantragtes Material

Diese Kosten können aus der ordentlichen Dotation der Forschungsstätte gedeckt werden.

3.7 Reisekosten

Da bezüglich Reisekosten die ordentliche Dotation nicht verwendbar ist, sollten aus Förderungsmitteln zwei Reisen von je einer Person zur Präsentation

¹²Da diese Diplomarbeit erst am Ende des ersten Jahres des Projektes beginnen wird, kann hier noch niemand namentlich erwähnt werden.

¹³Da diese Diplomarbeit erst im zweiten Jahr des Projektes beginnen wird, kann hier noch niemand namentlich erwähnt werden.

¹⁴Da diese Diplomarbeit erst im zweiten Jahr des Projektes beginnen wird, kann hier noch niemand namentlich erwähnt werden.

der Projektergebnisse, vor allem in die USA, finanziert werden. Dafür sollten etwa öS 30.000,- vorgesehen werden.

Die Tagungsgebühren können aber aus der ordentlichen Dotation der Forschungsstelle beglichen werden.

3.8 Publikationskosten

Diese Kosten können aus der ordentlichen Dotation der Forschungsstätte gedeckt werden.

4 Allfällige zusätzliche Angaben

Es wurde bei keiner anderen Stelle um Subventionen nachgesucht.

5 Tabellarische Übersichten

Übersicht über die zeitliche Entwicklung der Kosten

Kostenarten	1. Jahr	2. Jahr	Summe
Personalkosten	275.000,-	425.000,-	700.000,-
Gerätekosten	0,-	0,-	0,-
Materialkosten	0,-	0,-	0,-
Reisekosten	0,-	30.000,-	30.000,-
Sonstige Kosten	0,-	0,-	0,-
Summe	275.000,-	455.000,-	730.000,-

Zeitliche Entwicklung der Personalkosten

Personal	1. Jahr	2. Jahr	Summe
VA	260.000,-	260.000,-	520.000,-
N.N.	15.000,-	45.000,-	60.000,-
N.N.	0,-	60.000,-	60.000,-
N.N.	0,-	60.000,-	60.000,-
Summe	275.000,-	425.000,-	700.000,-