# WCET-Analyse für ein Java-Echtzeitbetriebssystem

## Diplomarbeit im Fach Informatik

vorgelegt von

***Michael Danel***

geb. am 29.03.80 in Tichau

Angefertigt am

Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: ***Prof. Dr. Wolfgang Schröder-Preikschat***
***Dipl. Inf. Christian Wawersich***

Beginn der Arbeit: 1. März 2005
Abgabe der Arbeit: 1. September 2005

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 30. August 2005 _____

**Kurzzusammenfassung**

Genaues Wissen über die maximale Ausführungszeit (*WCET*) eines Programms ist essenziell für die Entwicklung und Verifizierung echtzeitfähiger Software. Aus diesem Grund benötigen Entwickler statische Analyseprogramme, mit deren Hilfe sie das Zeitverhalten ihre Anwendungen untersuchen können.
Die Programmiersprache Java sieht zunächst keinerlei Hilfestellungen vor, die Ausführungszeiten eines Java-Programms abzuschätzen. Das in dieser Arbeit entwickelte Programm ist in der Lage, nur anhand der Java-Bytecodes das Zeitverhalten einer Applikation zu analysieren, ohne dem Programmierer besondere Restriktionen bezüglich dem Einsatz bestimmter Programmiertechniken aufzuerlegen. In die Berechnung der maximalen Ausführungszeit fließt der gesamte Kontrollfluss der Anwendung, inklusive all seiner Schleifenkonstrukte, virtueller Methodenaufrufe und Rekursionen mit ein.
Zu Beginn der Arbeit werden die Grundlagen der statischen Programmanalyse vorgestellt. Anschließend erfolgt eine detailierte Beschreibung der Anforderungen, Problemstellungen und Lösungsvarianten der statischen WCET-Berechnung. Abgerundet wird die Arbeit durch eine kurze Evaluierung und Zusammenfassung der Ergebnisse.

**Abstract**

Exact knowledge of a program's worst case execution time (WCET) is essential for the development and verification of real-time software. On this account developers depend on static analysis programs that are able to examine the time response of applications.
The programming language Java per se provides no support to estimate the execution time of a Java application. The program developed in this thesis is able to analyze the time response of a Java application considering its bytecode only and without imposing on the developer specific restrictions concerning the use of certain programming techniques. The calculation of the worst case execution time incorporates the entire control flow of the application, including all its loop constructs, virtual method calls and recursions.
This thesis starts out with an introduction to the basics of static program analysis. Afterwards, all requirements, the problem domain, and a solution to the static WCET-calculation are presented in detail. The thesis is concluded with an evaluation and a resumé of the results.

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

Ziel dieser Arbeit ist der Entwurf und die Implementierung eines Worst-Case-Execution-Time (*WCET*) Analyseprogramms für ein Java-Echtzeitbetriebssystem.

Java hält aufgrund seiner großen Popularität Einzug in immer mehr Gebiete der Informatik, die bisher ausschließlich der Programmiersprache C und hardwarenaher Assemblerprogrammierung vorbehalten waren. Galt Java noch vor einigen Jahren als zu Ressourcen raubend für den Einsatz auf Kleinstgeräten, so findet man heute immer mehr Java-Anwendungen nicht nur für Applikationsserver oder Desktop-PCs, sondern gerade auch für Mikrokontroller, wie die wachsende Zahl neuer Java-fähiger Handys belegt. Inzwischen geht die Entwicklung so weit, dass man mit der *Java Card* Spezifikation (vgl. [SM]) von Sun Java Applikationen für SmartCards entwickeln kann.

Aus dieser Entwicklung heraus ist abzusehen, dass Java künftig auch im Echtzeitbetrieb eine wichtige Rolle spielen wird, zumal es bereits eine Spezifikation für Echtzeit-Java gibt (vgl. [Bol00]). Dass gerade auf diesem Gebiet ein Bedarf an einer robusten Programmiersprache besteht, ist offensichtlich. Auf keinem anderen Gebiet der Softwareentwicklung sind fehlerfreie Anwendungen so wichtig, wie in der Gerätesteuerung. Man denke nur an die Steuerung von Kraftwerken, Turbinen oder eines Airbags. Java hilft aufgrund seines Typkonzepts, seiner Sicherheitsmechanismen und seines Speicherkonzepts von vornherein viele Programmierfehler zu vermeiden. Zudem sind im Java-Umfeld Konzepte der Testbarkeit und des Refactoring (um nur zwei zu nennen) weiter fortgeschritten als bei anderen Programmiersprachen, so dass Java seine Berechtigung im Echtzeitumfeld durchaus besitzt.

Der Echtzeitbetrieb ist durch *rechtzeitige (vorhersagbare) Programmabläufe* (vgl. [SP03, Erscheinungsformen, S.13]) geprägt. Ein Gerät gibt Zeitschranken vor, in denen eine Steuerungsapplikation auf Ereignisse des Gerätes

reagieren muss, um einen fehlerfreien Betrieb des gesamten Echtzeitsystem gewährleisten zu können. Missachtet die Applikation diese Schranken, so ist ein katastrophaler Schaden nicht ausgeschlossen. Daher ist es von essentieller Wichtigkeit die Ausführungsdauer eines jeden echtzeitfähigen Programms zu kennen, um Aussagen darüber treffen zu können, ob das System jederzeit in der Lage ist, auf Ereignisse *rechtzeitig* reagieren zu können. Soll ein Java-Programm auf Echtzeitfähigkeit geprüft werden, so ist es notwendig zunächst dessen Ausführungsdauer zu bestimmen. Erst wenn diese bekannt ist, ist es möglich zu überprüfen, ob das Programm alle hinsichtlich der Rechtzeitigkeit gestellten Anforderungen erfüllen wird.

Kapitel 2 klärt die grundlegensten Begriffe dieser Arbeit. Anschließend werden in Kapitel 3 ausgewählte Algorithmen und Techniken der Kontrollfluss- und Datenflussanalyse vorgestellt.

Basierend auf diesen Grundlagen beschreibt Kapitel 4 die Anforderungen, Problemstellungen und Lösungsvarianten der eigentlichen Problematik dieser Arbeit: der *WCET-Analyse*. Systematisch wird der Leser in einzelnen Abschnitten auf die Bestimmung der WCET vorbereitet: Die Schleifenanalyse erläutert zunächst, wie man den maximalen Durchlauf von Schleife errechnen kann; die Methodenanalyse beschäftigt sich mit Methodenaufrufen und welchen Einfluss sie auf die WCET haben; bevor am Ende dieses Kapitels mathematische Berechnungsmethoden der WCET präsentiert werden, wird der Einfluss der Hardware auf die WCET näher beleuchtet.

Abgerundet wird die gesamte Arbeit durch ein Evaluierungskapitel, in dem beschrieben wird, wie man das Analyseprogramm benutzt, welche Resultate es liefert und wie diese Resultate validiert werden könnten.

# Kapitel 2

# Grundlegende Begriffe

Zum besseren Verständnis der in den folgenden Kapiteln dargestellten Analyse werden zunächst die grundlegensten Begriffe der Arbeit erläutert. Es folgt ein kurzer Abriss über den Begriff der maximalen Ausführungszeit und dessen Bestimmung (s. 2.1), das Betriebssystems JX (s. 2.2) und den der Analyse zu Grunde liegendem Java-Bytecode (s. 2.3).

## 2.1 Maximale Ausführungszeit – WCET

*„Unter der maximalen Ausführungszeit (**W**orst-**C**ase-**E**xecution-**T**ime,* WCET[1]*) versteht man die längste zu beobachtende Ausführungszeit eines Programms, wenn es in seiner Ausführungsumgebung läuft“* (vgl. [Eng02, S.4]). Im Gegensatz dazu beschreibt die **B**est-**C**ase-**E**xecution-**T**ime, *BCET*, die kürzeste zu beobachtende Ausführungszeit des Programms. Im Allgemeinen ist es sehr schwer, wenn nicht sogar unmöglich, diese Zeiten exakt zu bestimmen, da Beide von Eingabedaten des Programms zur Laufzeit abhängen.

Gewöhnlich ist es ausreichend eine Abschätzung dieser Zeiten zu ermitteln, jedoch müssen diese Abschätzungen sicher sein. Gerade im Hinblick auf die WCET darf die Abschätzung nicht optimistisch ausfallen, sondern muss größer oder gleich dem exakten Wert sein. Andernfalls würde man ein System konfigurieren, das auf falschen Annahmen beruht und somit ausfallen könnte. Beispielsweise würde man annehmen, dass das System in kürzerer Zeit auf ein Ereignis reagieren kann, als dies tatsächlich der Fall ist. Folglich würde man davon ausgehen, dass man häufiger auf Ereignisse reagieren kann, als de facto der Fall ist, was, wie bereits in Kapitel 1 erwähnt, fatale Auswirkungen nach sich ziehen würde.

[1] Im Folgenden nur noch als WCET bezeichnet.

WCET-Abschätzungen müssen demnach konservativ ermittelt werden, gleichzeitig aber möglichst nahe am exakten Wert liegen. Die triviale Aussage „Das Programm brauche im schlimmsten Fall unendlich lang“ mag zwar korrekt sein, ist aber völlig nutzlos.

### 2.1.1 Bestimmung

Um die WCET eines Programms bestimmen zu können, unterscheidet man zwei verschiedene Ansätze (vgl. [Eng02]):

- **Experimentelle Messung**
  Im Fall der experimentellen Messung startet man das Programm mit möglichst *schlechten* oder für die Anwendung besonders *typischen* Eingabedaten und misst die Laufzeit. Schließlich wird zu der auf diese Weise bestimmten Zeit noch ein kleines Delta dazuaddiert und man hofft, dass die daraus resultierende Zeit schlechter als die tatsächliche WCET ist. Obwohl wiederholte Messungen zur statistischen Sicherheit beitragen, erhält man auf diese Weise natürlich keinerlei Garantie, dass die WCET tatsächlich eingeschlossen ist – man erhält eben *nur* eine Wahrscheinlichkeit!

- **Statische Analyse**
  Um sicherzugehen, dass die ermittelte Zeit schlechter als die exakte WCET ist, müssen mathematische Analysetechniken herangezogen werden. Im Gegensatz zu herkömmlichen Messmethoden wird bei der statischen Analyse das Programm nicht ausgeführt und gemessen, sondern man versucht anhand des Programmcodes (Quelldateien oder Binärcode) Eigenschaften des Programms zu bestimmen, welche die Ausführungszeit des gesamten Programms beeinflussen. Das zu analysierende Programm wird in ein mathematisches Modell umgewandelt und mit Zuhilfenahme der ermittelten Eigenschaften wird die WCET *errechnet*. Von der statischen Analyse muss berücksichtigt werden, dass die WCET nicht nur vom Programmcode auf Softwareebene, also dem Kontrollfluss, sondern auch von der Hardware, auf dem das Programm ausgeführt wird, abhängt. Das heißt man muss zusätzlich ein Modell der Hardware erstellen, das Auskunft darüber gibt, wie lange eine Elementaroperation auf der Hardware dauert oder wie viel Zeit vergeht bis eine Gruppe von Elementaroperation abgearbeitet wurde, da die Ausführungsdauer einer Gruppe von Operationen nicht unbedingt der Summe der Zeiten jeder Elementaroperation entspricht. Soll dies berücksichtigt werden, muss man ein möglichst genaues Modell der

Hardware erstellen, das Auswirkungen von Pipelines und Caches der Zielarchitektur auf die Elementaroperationen berücksichtigt. Die Analysen werden dadruch komplexer, nähern sich allerdings dem exakten Wert immer mehr an.

Einzig eine auf einer statischen Analyse basierte Bestimmung der WCET eines Programms kann Sicherheit darüber geben, dass die so ermittelte Zeit mindestens so schlecht ist, wie die exakte WCET. Aus diesem Grund beschränken sich die folgenden Kapiteln allein auf diese Art der Bestimmung.

## 2.2 JX

Das im Rahmen dieser Arbeit entwickelte Programm zur Bestimmung der WCET, analysiert Java-Bytecode, der im Kontext jeder beliebigen Laufzeitumgebung für Java-Applikationen ausgeführt werden kann. Dennoch war die Hauptmotivation ein Werkzeug zu kreieren, das Entwicklern bei der Programmierung echtzeitfähiger Anwendungen für das Betriebssystem JX unterstützt (vgl. [MG02]).

JX ist ein Betriebssystem, das vollständig in Java geschrieben ist[2]. Es stellt neben gängigen Betriebssystemdiensten eine Laufzeitumgebung für Java-Applikationen bereit. Bemerkenswert ist sein Speicherschutzkonzept, das anstatt auf einer Memory-Management-Unit (MMU) ausschließlich auf dem Typkonzept von Java und einem erweiterten Bytecode-Verifier basiert. So können sowohl Komponenten des Betriebssystem, als auch gewöhnliche Anwendungen in *einem* Adressraum ausgeführt werden und der Speicherschutz bleibt weiterhin bewahrt.

Je nach Betriebsart kann JX unterschiedlich konfiguriert werden. Für alle Konfigurationen ist lediglich ein Mikrokern als zentrales Element vorgeschrieben. Alle übrigen Komponenten sind frei wählbar und können für den speziellen Anwendungsfall zusammengestellt werden.

Für eine detaillierte Beschreibung des gesamten Systems sei auf [Gol02] verwiesen. Dort findet der Leser einen Überblick über alle wichtigen Konzepte und Komponenten von JX.

### 2.2.1 Echtzeitbetrieb

*„Eine Berechnung geschieht in Echtzeit, wenn ihr Ergebnis innerhalb eines gewissen Zeitraumes [...] garantiert vorliegt, das heißt bevor eine bestimm-*

[2]Einzig der in Assembler und C geschriebener Mikrokern weicht hiervon ab. Er bietet aber ausschließlich Funktionalität, wie das Starten des Systems während der Bootphase, die nicht mit Hilfe der Programmiersprache Java implementiert werden konnte.

*te Zeitschranke erreicht ist“* [wik, Echtzeit]. Diese Definition von Echtzeit unterstreicht besonders die Bedeutung von Zeit als physikalische Größe, die mit dieser Betriebsart verbunden ist. Die Korrektheit des Systems ist somit neben dem richtigen Berechnungsergebnis auch mit dem Zeitpunkt, zu dem das Ergebnis vorliegt, eng verflochten. Liegt das Resultat einer Berechnung *nicht rechtzeitig* vor, so ist es gleichermaßen wertlos wie ein falsches Ergebnis. Über die Länge der Zeitspanne, zu der das Ergebnis vorliegen muss, kann keine generelle Aussage getroffen werden – sie ist systemabhängig und variiert mit der konkreten Anwendung.

Wie bereits angedeutet, kann das Betriebssystem JX für den Einsatz im Echtzeitbetrieb konfiguriert werden. Die Basiskonfiguration des Systems besteht dabei aus dem Mikrokern, einer echtzeitfähigen automatischen Speicherverwaltung (vgl. [Gab05]) und einem *offline Scheduler* (vgl. [SP03, Ablaufplanung, S.11]). Daneben werden Arbeitseinheiten (s. 2.2.2) vom Anwender festgelegt, durch welche die Regelungs- und Steuerprozesse des Gesamtsystems durchgeführt werden.

Das offline Scheduling kennt die Zeiten aller im System eingesetzter Arbeitseinheiten und erstellt anhand dieser einen Fahrplan für den Scheduler, der zur Laufzeit des System zu festen Zeitpunkten zwischen den Einheiten umschaltet. Die für das offline Scheduling benötigten Zeiten liefert das in dieser Arbeit entwickelte Analyseprogramm.

Natürlich können auch je nach Anwendungsszenario andere Konfigurationen erzeugt werden, die beispielsweise gar keinen (im Falle nur einer einzigen Arbeitseinheit) oder einen ausgefeilteren Scheduler als den offline Scheduler verwenden. Für viele Anwendungen ist die Basiskonfiguration aber durchaus ausreichend.

### 2.2.2 Task

*Task* ist die am häufigsten benutzte Bezeichnung für Arbeitseinheiten in Echtzeitbetriebssystemen. Tasks sind fest mit einem Programm verbunden, das die Regelungs- und Steueraufgaben beschreibt. Jeder Task durchläuft zur „Lebzeit“ drei Phasen: Aktivierungs-, Arbeits-, und Abschaltphase.

1. **Aktivierungsphase – *startup()***
   In der Aktivierungsphase initialisiert der Task alle für seine Aufgabe benötigen Ressourcen. Steuert der Task beispielsweise ein Gerät, so kann er in dieser Phase alle Geräteparameter entsprechend seiner Aufgabe einstellen.

2. **Arbeitsphase – *run()***
   Die Arbeitsphase besteht im Grunde aus einer Schleife, wobei der Task

in jedem Schleifendurchlauf auf äußere Ereignisse reagiert und so einen Prozess steuert.

3. **Abschaltphase – *shutdown()***
   Die Abschaltphase wird erreicht, wenn der Task seine Aufgabe erledigt hat, beispielsweise wenn das Gerät, das er steuert, abgeschaltet wird. Er hat hier die nötige Zeit alle belegten Ressourcen freizugeben, um das System in einem konsistenten Zustand zu hinterlassen.

Für die Implementierung dieser drei Phasen ist von JX eine feste Schnittstelle festgelegt. Sie kann Listing 2.1 entnommen werden.

```
public interface Task {
    // Aktivierungsphase
    public void startup();
    // Arbeitsphase
    public void run();
    // Abschaltphase
    public void shutdown();
}
```

Listing 2.1: Taskschnittstelle

Die Zuordnung der Methoden zu den drei Phasen ist den Kommentaren der Schnittstellendefinition zu entnehmen.

## 2.3 Java-Bytecode

Die in den folgenden Kapiteln beschriebene Bestimmung der WCET konzentriert sich ausschließlich auf die Analyse des Java-Bytecodes. Der Java-Bytecode ist eine Zwischensprache, die nicht direkt auf einer realen Maschine ausgeführt werden kann. Er wird gewöhnlich von einem Java-Compiler erzeugt und schließlich von einer virtuellen Maschine, der Java-Virtual-Machine (*JVM*), interpretiert oder von einem *just in time* Compiler (JIT) zur Laufzeit in Maschinencode übersetzt und ausgeführt. Im Fall von JX existiert ein *ahead of time* Compiler (AOT), der den Java-Bytecode bereits vor seiner Ausführung in Maschinencode übersetzt (vgl. [Waw01]). Der auf diese Weise gewonnene Code kann schließlich von einer realen Maschine ausgeführt werden. Der Java-Bytecode ist ein Code für eine Kellermaschine mit Halde (vgl. [Phi04]). Der Befehlssatz enthält zum Einen Instruktionen um Konstanten, Felder oder lokale Variablen auf den Operandenstack zu laden, zum Anderen Instruktionen, die diese Operanden vom Stack entnehmen, auf ihnen arbeiten

und das Resultat zurück auf den Stack legen. Ebenfalls werden Methodenparameter und Rückgabewerte mit Hilfe des Operandenstacks zwischen Methoden ausgetauscht. Im Folgenden soll lediglich ein grober Überblick über den Befehlssatz und das Format des Java-Bytecodes gegeben werden. Eine detaillierte Beschreibung der Spezifikation findet der Leser in [TL99].

### 2.3.1 Classfile

Jedes Java-Programm besteht aus einer Reihe von Klassen, die alle einzeln übersetzt und in sogenannten *Java-Classfiles* abgespeichert werden. Der typische Aufbau eines Classfiles kann Abbildung 2.1 (vgl. [MD]) entnommen werden.

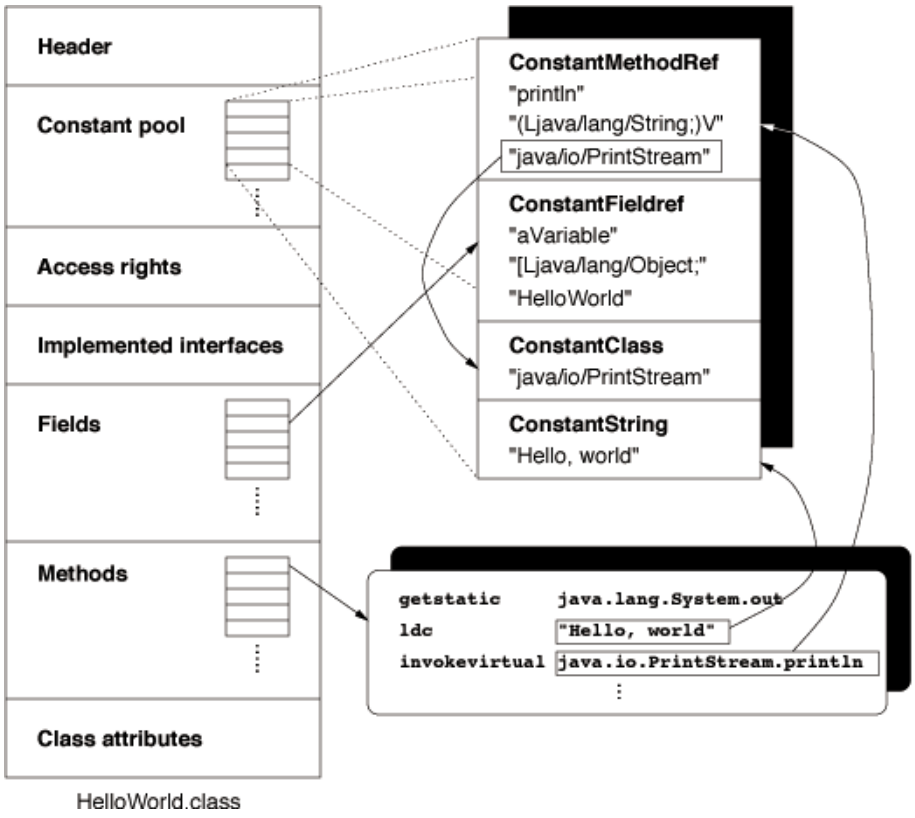



Abbildung 2.1: Aufbau eines typischen Java-Classfiles

Jedes Classfile enthält einen Kopfeintrag gefolgt vom Konstantenpool, den man sich grob als das Textsegment einer ausführbaren Datei vorstellen kann. Anschließend werden die Zugriffsrechte der Klasse und eine Liste aller, von der Klasse implementierten, Schnittstellen kodiert. Alle in der Klasse

definierten Felder und Methoden werden in zwei separaten Listen gespeichert. In Abbildung 2.1 entsprechen diese Listen den Kästchen *Fields* und *Methods*. Neben Verweisen auf den Konstantenpool, der die Namen der Felder und Methoden enthält, werden auch die Zugriffsrechte auf die einzelnen Komponenten der Listeneinträge gesichert. Das letzte Element jedes Eintrags ist ein Array von Attributen. Das wichtigste Attribut jedes Methodeneintrags ist wohl das Codeattribut, das die Bytecodesequenz der Methode beinhaltet. Komplettiert wird das Classfile durch eine Liste von Klassenattributen, die beispielsweise den Namen der Quelldatei speichern.

Da alle symbolischen Referenzen, die zur Auflösung von Klassen, Methoden und Feldern benötigt werden, als Zeichenketten abgespeichert sind, belegt der Konstantenpool im Schnitt mit ca. 60% den größten Anteil der Classfiles. Zum Vergleich verbraucht der Bytecode mit ca. 12% relativ wenig Speicherplatz.

### 2.3.2 Befehlssatz

Jede Bytecodeinstruktion ist, wie der Name bereits andeutet, genau ein Byte lang, so dass maximal 256 verschiedene Instruktionen kodiert werden könnten. Der Java-Bytecode Befehlsatz umfasst zur Zeit 212 Instruktionen – die Restlichen 44 sind für künftige Erweiterungen reserviert. Grob kann der Befehlssatz in folgende Kategorien gruppiert werden:

- **Stackinstruktionen**
  Der Java-Bytecode enthält Instruktionen zur Manipulation des Operandenstacks. So können Operanden entfernt, verschoben oder dupliziert werden.

- **Arithmetische Instruktionen**
  Arithmetische oder logische Instruktionen entnehmen meist einen oder zwei Operanden vom Stack, führen die gewünschte Operation aus und legen das Ergebnis auf den Stack zurück.

- **Kontrollflussinstruktionen**
  Der Java-Bytecode umfasst eine Palette an Instruktionen, die den Kontrollfluss eines Programms beeinflussen können. So gibt es zum Einen Instruktionen für bedingte und unbedingte Sprünge, zum Anderen Instruktionen um Ausnahmen (Exceptions) auszulösen und so den Kontrollfluss des Programms umzulenken.

- **Lade- und Speicherinstruktionen**
  Lade- und Speicherinstruktionen können Konstanten, Felder und lo-

kale Variablen auf den Stack laden bzw. an die entsprechenden Speicherstellen speichern. Daneben existieren spezielle Instruktionen für die Manipulation von Arrays.

- **Methodenaufrufsinstruktionen**
  Der Befehlssatz enthält gesonderte Instruktionen für den Aufruf statischer und virtueller Methoden, sowie für den Aufruf von Schnittstellenmethoden und Klassenkonstruktoren.

- **Objektverwaltungsinstruktionen**
  Zur Erzeugung von Objekten und Arrays sieht der Java-Bytecode getrennte Instruktionen vor. Ebenso existieren Instruktionen um die Länge eines Arrays oder den Typ eines Objekts abzufragen.

- **Typüberprüfungsinstruktionen**
  Da alle Instruktionen streng typisiert sind, existieren neben Typumwandlungs- auch Typüberprüfungsinstruktionen, mit denen zur Laufzeit der Typ jedes Datenobjekts abgefragt werden kann.

Die in den kommenden Kapiteln verwendeten Bytecodeinstruktionsnamen sind meist selbsterklärend. Für eine detaillierte Auflistung aller Instruktionen sei an dieser Stelle nochmals auf [TL99] verwiesen.

# Kapitel 3

# Ausgewählte Algorithmen und Techniken der Programmanalyse

Mit statischen Programmanalyseverfahren verfolgt man das Ziel, Eigenschaften über ein Programm zu erfahren, ohne das Programm vorher auszuführen. Betrachtet man dieses Vorhaben aus Sicht der Theoretischen Informatik etwas genauer, so scheint dieses Vorhaben aufgrund des *Satzes von Rice* (vgl. [Sch01]) utopisch. Er besagt, dass jede nichttriviale Eigenschaft eines beliebigen Programms unentscheidbar ist. Eine Folgerung daraus ist, dass Entscheidungsverfahren, hier also Analyseverfahren, die versuchen Eigenschaften über ein Programm zu berechnen, nicht gleichzeitig vollständig und korrekt sein können.

Aus diesem Grund unterliegen alle Analyseverfahren einem Kompromiss. So darf eine Analyse, die das Ziel verfolgt, eine bestimmte Eigenschaft eines Programms zu erkennen, diese Eigenschaft bei einem Programm *übersehen*, obwohl sie vorhanden ist, niemals aber einem Programm eine Eigenschaft zusprechen, die es nicht besitzt.

Das folgende Kapitel beschäftigt sich mit, für diese Arbeit relevanten, Algorithmen und Techniken der Programmanalyse. Zunächst wird das Konzept des Kontrollflusses (s. 3.1) und der Kontrollflussgraphen (s. 3.1.2) näher beleuchtet und einige grundlegende graphentheoretische Algorithmen vorgestellt. Der zweite Abschnitt beschäftigt sich mit der Datenflussanalyse – im Speziellen mit der Suche nach verfügbaren Definitionen (s. 3.3.1).

## 3.1 Kontrollfluss

Der Kontrollfluss bezeichnet die Reihenfolge, in der die Anweisungen eines Computerprogramms abgearbeitet werden. Solange in einem Programm keine Kontrollstrukturen, wie if-then-else-Konstrukte, switch-Anweisungen, Schleifen etc., eingesetzt werden, bleibt der Kontrollfluss geradlinig. Das heißt man kann im Voraus sagen, welche Anweisung als nächstes ausgeführt wird. Erst der Einsatz von Kontrollstrukturen führt zu Verzweigungen im Ablaufplan eines Programms.

Betrachten wir als kleines Beispiel den Quellcode aus Listing 3.1, welcher eine Methode zur Berechnung der Fakultätsfunktion beschreibt. Die Methode bekommt als Eingabeparameter eine Ganzzahl, berechnet nach $n$-Schleifendurchläufen deren Fakultät und liefert schließlich das Resultat.

```
public int factorial(int n) {
    int result = 1;

    while (n > 1) {
        result = n * result;
        n--;
    }

    return result;
}
```

Listing 3.1: Codebeispiel: Fakultätsfunktion

Eines der wichtigsten Ziele der Kontrollflussanalyse ist das Auffinden von Kontrollstrukturen. Gerade im Hinblick auf die Berechnung der WCET ist das Auffinden von Schleifen von großer Bedeutung. Hat man den Quellcode des zu analysierenden Programms zur Hand, so erkennt man auf den ersten Blick alle Schleifenkonstrukte. Als nicht mehr ganz so einfach stellt sich dieses Unterfangen heraus, wenn man das gleiche Programm anhand seines Java-Bytecodes (s. Listing 3.2) untersuchen möchte, da dieser nur reine Sprunganweisungen und keinerlei syntaktische Visualisierungshilfen kennt.

```
 0: iconst_1
 1: istore_2
 2: iload_1
 3: iconst_1
 4: if_icmple -> 17
 7: iload_1
 8: iload_2
```

```
 9: imul
10: istore_2
11: iinc 1 -1
14: goto -> 2
17: iload_2
18: ireturn
```

Listing 3.2: Bytecode zu Listing 3.1

Um aus dieser Darstellung dennoch alle nötigen Kontrollstrukturen erkennen zu können, wird der Bytecode zunächst in einen Kontrollflussgraphen transformiert. Hierfür ist zuvor eine Unterteilung des Bytecodestroms in *Basisblöcke* nötig.

### 3.1.1 Basisblockerzeugung

Ein Basisblock, auch Grundblock genannt, *„ist eine Folge fortlaufender Anweisungen, in die der Kontrollfluss am Anfang eintritt und die er am Ende verlässt, ohne dass er dazwischen anhält oder - außer am Ende - verzweigt“* (vgl. [AVA88, S.645]).

Aus dieser Definition lässt sich leicht folgender Algorithmus ableiten, der einen Bytecodestrom in seine Grundblöcke zerlegt:

1. **Blockanfänge markieren:**
   (a) Der erste Befehl ist ein Blockanfang.
   (b) Jeder Befehl, der Ziel eines bedingten oder unbedingten Sprungs ist, ist ein Blockanfang.
   (c) Jeder Befehl, der direkt auf einen bedingten oder unbedingten Sprung folgt, ist ein Blockanfang.
2. **Basisblöcke ablesen:** Zu jedem Blockanfang besteht der zugehörige Basisblock aus dem Blockanfang und allen darauf folgenden Befehlen bis zum nächsten Blockanfang oder bis zum Ende des Bytecodestroms.

Als Besonderheit von Regel 1b gilt für den Java-Bytecode, dass auch jeder *Exceptionhandler* Ziel eines Sprungs ist. Exceptionhandler sind speziell markierte Codeabschnitte, die nur dann von der Laufzeitumgebung aufgerufen werden, wenn in einer Methode eine Ausnahme (*engl. Exception*) auftritt. Eine mögliche Ausnahme könnte ein Fehler beim Öffnen einer Datei sein.

### 3.1.2 Kontrollflussgraph

Wurde ein Bytecodestrom erfolgreich in seine Basisblöcke zerlegt, so kann durch Hinzufügen von Kontrollflussinformationen die Menge der Basisblöcke zu einem gerichteten Graphen, erweitert werden. Diesen Graphen nennt man *Kontrollflussgraph* oder kurz *Flussgraph*. Die Knoten des Flussgraphen entsprechen den Grundblöcken. Für die Kanten gilt folgende Regel: *„Es gibt eine gerichtete Kante vom Block B1 zum Block B2, falls B2 in einer Ausführungssequenz direkt nach B1 folgen kann“* (vgl. [AVA88, S.650]).

Wendet man zunächst den Algorithmus zur Basisblockerzeugung aus Kapitel 3.1.1 an und generiert daraufhin mit der obigen Regel die Kanten für für das Listing 3.2, so entsteht der Kontrollflussgraph aus Abbildung 3.1.

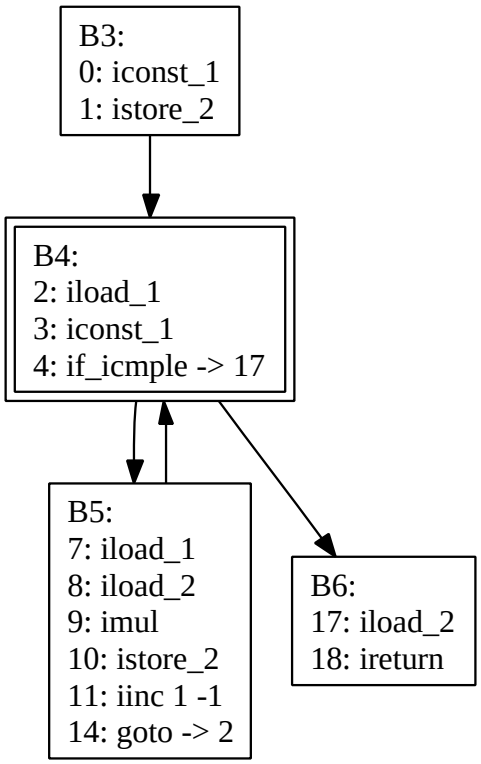



Abbildung 3.1: Kontrollflussgraph entsprechend dem Listing 3.2

Der *Startknoten* ist hierbei mit B3, der *Endknoten*, der die **return**-Anweisung enthält, mit B6 beschriftet. Die Knoten B4 und B5 repräsentieren einen Zyklus, der mit Hilfe des in Kapitel 3.2.3 beschriebenen Algorithmus gefunden wurde.

## 3.2 Kontrollflussanalyse

Aufgabe der Kontrollflussanalyse ist das Auffinden der hierarchischen Ablaufsteuerung innerhalb jeder Prozedur und das Sammeln globaler Informationen über die Manipulationen von Daten für die anschließende Datenflussanalyse. Die folgenden Kapitel beschränken sich auf nur einen einzigen Aspekt der Kontrollflussanalyse, nämlich den der *Schleifensuche*.

Für die WCET-Analyse ist die Kenntnis darüber, welche Blöcke eine Schleife bilden, entscheidend für die spätere Berechnung des maximalen Schleifendurchlaufs, da dieser einer der wichtigsten Faktoren für die gesamte WCET ist.

Vorab folgt aber noch die Behandlung zweier wichtiger Algorithmen: *Postorder-Traversierung* und *Dominatorenbestimmung*.

### 3.2.1 Postorder-Traversierung

In vielen Verfahren müssen Graphen nicht wie üblich von der Wurzel zu den Blättern (*Preorder-Traversierung*), sondern in umgekehrter Reihenfolge, von den Blättern zur Wurzel, traversiert werden. Diese Art der Traversierung wird als *Postorder-Traversierung* oder manchmal auch als *Durchlauf in Nebenreihenfolge* bezeichnet. Ein rekursiv formulierter Algorithmus (vgl. [Sed02, S.71]) für gerichtete Graphen könnte wie folgt aussehen:

1. Markiere den aktuellen Knoten als bearbeitet.
2. Traversiere alle nicht markierten Teilbäume (in beliebiger Reihenfolge).
3. Bearbeite den aktuellen Knoten.

Oftmals ist es auch sinnvoll den einzelnen Knoten aufsteigende Nummern zuzuweisen, um im Nachhinein einfacher auf bestimmte Knoten zugreifen zu können. So entsprechen die *Post-Nummern* Zahlen von 1 bis n, wobei der in Postorderreihenfolge zuerst besuchte Knoten die Nummer 1, der zuletzt besuchte Knoten, die Wurzel des Graphen, die Nummer n bekommt.

### 3.2.2 Dominanz

Der Begriff *Dominanz* bezeichnet im Zusammenhang mit gerichteten Graphen eine binäre Relation auf der Menge $G$ der Knoten des Graphen. Man sagt: „Ein Knoten $d \in G$ dominiert einen Knoten $k \in G$, geschrieben $d\ dom\ k$, falls jeder Pfad von der Wurzel des Graphen zu Knoten $k$ den Knoten $d$ enthält.“ Des Weiteren bezeichnet man $d$ in diesem Fall als *Dominator* von $k$.

Eigenschaften dieser Relation sind:

1. reflexiv: $x\ dom\ x$
2. transitiv: $x\ dom\ y \wedge y\ dom\ z \Rightarrow x\ dom\ z$
3. antisymmetrisch: $x\ dom\ y \wedge y\ dom\ x \Rightarrow x = y$

Um die Dominatoren eines beliebigen gerichteten Graphen $G$ zu berechnen, bedient man sich folgender Idee:
Man wählt die Menge der Dominatoren jedes Knoten $n$, geschrieben $D(n)$, von $G$ zu groß und verkleinert sie sukzessive mit Hilfe der Menge der Dominatoren vom Vorgänger $p \rightarrow n$, $D(p)$, solange, bis sich schließlich auf allen Mengen eine Stabilität einstellt, d.h. bis keine Veränderungen in den $D(i)$ $(\forall i \in G)$ mehr auftreten.

Listing 3.3 beschreibt den Algorithmus.

```
D(s) := {s}; // s ist der Startknoten
for n ∈ G \ {s} do
    D(n) := G;

while Veraenderungen in einem D(n) auftreten do
    for n ∈ G \ {s} do
        D(n) := {n} ∪ ⋂_{p → n} D(p)
```

Listing 3.3: Algorithmus zur Dominatorenbestimmung

Man beachte, dass sich das Verfahren sehr effizient implementieren lässt, wenn die Mengen der Dominatoren als Bitvektoren repräsentiert werden. So lassen sich die Mengenoperationen aus der letzten Zeile des Listings mit Hilfe der logischen *und* und *oder* Operationen realisieren.

### Direkte Dominanz

Die *direkte Dominanz* (engl. *immediate dominance*) beschreibt ebenfalls eine binäre Relation auf der Menge $G$ der Knoten eines gerichteten Graphen. Man sagt: „Ein Knoten $i \in G$ dominiert einen Knoten $k \in G$ *direkt*, geschrieben $i\ idom\ k$, falls $i\ dom\ k$, $i \neq k$ und es keinen weiteren Knoten $c \in G$ gibt, so dass $c \neq i$ und $c \neq k$, für den gilt: $i\ dom\ c$ und $c\ dom\ k$.“

Man spricht demnach dann vom *direkten Dominator* eines beliebigen Knotens $x$, geschrieben $Idom(x)$, falls man den Dominator meint, der „am nächsten“ zu $x$ ist. Dieser Dominator ist augenscheinlich *eindeutig*.

Sind alle Dominatoren errechnet, so lassen sich mit dem Algorithmus aus Listing 3.4 ([Muc97, FIG. 7.15, S.185] die direkten Dominatoren bestimmen.

Will man den direkten Dominator eines Knotens finden, so betrachtet man zunächst die Liste aller seiner Dominatoren. Ist einer dieser Dominatoren gleichzeitig auch Dominator eines anderen Knotens (Dominators) dieser Liste, so kann er laut Definition nicht der direkte Dominator des ursprünglichen Knoten sein. Anhand dieser Idee verkleinert man die Dominatorenlisten aller Knoten, bis sie schließlich nur noch ein Element enthalten: den direkten Dominator des jeweiligen Knotens. Der Algorithmus aus Listing 3.4 arbeitet am effizientesten, wenn die äußerste Schleife in Postorder traversiert wird.

```
for n ∈ G do
    Idom(n) := D(n) \ {s};

    for n ∈ G \ {s} do // in postorder
        for x ∈ Idom(n) do
            for y ∈ Idom(n) \ {x} do
                if y ∈ Idom(x)
                    Idom(n) := Idom(n) \ {x}
```

Listing 3.4: Algorithmus zur Bestimmung der direkten Dominatoren

### Dominatorbaum

Hat man alle direkten Dominatoren berechnet, lässt sich daraus ganz einfach der *Dominatorbaum* ablesen: Knoten $k$ ist Kind seines direkten Dominators $Idom(k)$. So erhalten wir für den Kontrollflussgraph aus Abbildung 3.1 den Dominatorbaum in Abbildung 3.2.

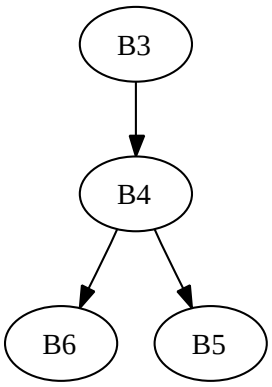



Abbildung 3.2: Dominatorbaum zum Flussgraph aus Abbildung 3.1

### 3.2.3 Schleifensuche

Wie bereits in Kapitel 3.1 festgestellt wurde, lassen sich Kontrollstrukturen, wie beispielsweise Schleifen, nicht so einfach anhand reiner Bytecodesequenzen erkennen, wenn man nicht zusätzlich den Quellcode zur Hand hat. Ist der Bytecode aber erst in Form eines Flussgraphen repräsentiert, so lassen sich mit Hilfe der Dominatorrelation auf diesem Flussgraph relativ einfach Schleifen finden.

Schleifen in einem Kontrollflussgraph besitzen zwei prägende Eigenschaften:

1. Jede Schleife besitzt genau einen Eingangspunkt, der sogenannte *Kopf* der Schleife. Dieser Kopf hat die Eigenschaft, dass er alle Knoten innerhalb der Schleife dominiert.

2. Es gibt mindestens einen Weg zum Iterieren der Schleife, d.h. es existiert mindestens eine Kante aus dem Inneren der Schleife zurück zum Kopf.

Anhand dieser beiden Eigenschaften ist es möglich, einen Algorithmus zum Erkennen beliebiger Schleifenkonstrukte zu formulieren.

#### Rückwärtskante

Eine Kante $n \to d$ in einem Kontrollflussgraph heißt Rückwärtskante, genau dann wenn: $d$ *dom* $n$, also der *„Der Schwanz den Kopf dominiert“*.

#### Natürliche Schleife

Zu einer Rückwärtskante $n \to d$ lässt sich eine *natürliche Schleife* definieren als der Subgraph, bestehend aus der Menge aller Knoten, die den Knoten $d$ und alle weiteren Knoten $k$ enthalten, von denen aus $n$ erreicht werden kann, ohne dass $d$ dabei durchlaufen werden muss.

Hinter der Konstruktion einer natürliche Schleife anhand der Rückwärtskante $n \to d$ steckt folgende Idee: Um alle Knoten zu erhalten, von denen aus $n$ (der Schwanz der Rückwärtskante) erreicht werden kann, muss man zunächst alle seine Vorgänger und schließlich, rekursiv von ihnen aus, alle deren Vorgänger betrachten. Dies tut man so lange, bis man an den Kopf der Rückwärtskante, gleichzeitig auch der Kopf der Schleife, gestossen ist. Da der Kopf das Ende der Betrachtung markiert, kann man sicher gehen, dass dessen Vorgänger nicht weiter betrachtet werden und man daher nur die Knoten erhält, die $n$ erreichen, ohne $d$ zu durchlaufen. Listing 3.5 beschreibt ausführlich diese Idee.

```
NatLoop natLoop(Edge backEdge) {
    WorkList WL := ∅
    NatLoop L := backEdge.head;
    insert(WL, L, backEdge.tail);

    while (WL ≠ ∅) {
        m := WL.pop();
        forall (p ∈ m.predecessor) do {
            insert(WL, L, p);
        }
    }

    return L;
}

void insert(WL, L, m) {
    if (m ∉ L) {
        L := L ∪ {m};
        WL := WL ∪ {m};
    }
}
```

Listing 3.5: Algorithmus zur Bestimmung natürlicher Schleifen

### Unnatürliche Schleife

Im Gegensatz zu natürlichen Schleifen bezeichnet man mit *unnatürlichen Schleifen* alle Zyklen in Flussgraphen, die über keinen eindeutigen Kopf und somit über keine Rückwärtskante verfügen. Nun stellt sich die Frage, ob man mit dem Konzept der natürlichen Schleifen auch wirklich alle Schleifen erkennt, die man in einem Java-Programm formulieren kann. Antwort hierauf findet man in [AVA88, S.743]. Solange eine Sprache keine *goto*-Anweisungen erlaubt, werden alle Schleifenkonstrukte ausschließlich durch natürliche Schleifen dargestellt. Und da das Schlüsselwort *goto* nicht im Sprachumfang der Programmiersprache Java enthalten ist, lässt Java nur die Konstruktion natürlicher Schleifen zu. Demnach genügt es vollkommen, ausschließlich den Algorithmus aus Listing 3.5 zu betrachten, will man alle möglichen Schleifen im Java-Bytecode erkennen.

## 3.3 Datenflussanalyse

Die Datenflussanalyse hat zur Aufgabe den *Fluss* von Daten durch ein Programm zu verfolgen, genauer soll hierbei Wissen (Informationen) über Variablenwerte, Definitionen (s.u.), Ausdrücke, Datenobjekte usw. ausgerechnet werden. Dieses Wissen dient meist dem Zweck der Codeoptimierung oder Codevalidierung. Grundlage der Datenflussanalyse bildet gewöhnlich der Kontrollflussgraph, der die Mengen von Zuständen beschreibt, die ein Programm zur Laufzeit annehmen kann. Diese disjunkten Zustandsmengen und alle Operationen, die eine Menge in eine andere überführen, werden für die Berechnung genutzt.

Datenflussinformationen können durch Aufstellen und Lösen von Gleichungssystemen bestimmt werden, die Informationen an verschiedenen Punkten des Programms miteinander verbinden. Die beiden folgenden Gleichungen bilden oft die Basis für alle weiteren Berechnungen:

$$in[B] = \bigcup_{P \in predecessor(B)} out[P] \tag{3.1}$$

$$out[B] = gen[B] \cup (in[B] - kill[B]) \tag{3.2}$$

Seien $B$ und $P$ Basisblöcke eines beliebigen Flussgraphen, so lassen sich die Mengen *gen[B]*, *kill[B]*, *in[B]* und *out[B]* auf folgende Weise interpretieren:

- **gen[B]** beschreibt die Menge der Informationen, die im Basisblock $B$ erzeugt werden.
- **kill[B]** ist das Pendant zu gen[B]. Hier wird die Menge der Informationen beschrieben, die durch die Anweisungen im Basisblock $B$ zerstört werden. Eine Anweisung zerstört eine Information, wenn durch sie eine bereits bekannte Informationen überschrieben wird.
- **in[B]** entspricht der Menge an Informationen, die den Basisblock $B$ erreichen, während
- **out[B]** die Informationen beschreibt, die den Basisblock $B$ verlassen.

Die Mengen gen[B] und kill[B] lassen sich unabhängig von allen anderen Basisblöcken und somit vorab bestimmen. Die beiden Gleichungen 3.1 und 3.2 ermöglichen die Berechnung der beiden übrigen Mengen. Sie lassen sich auf folgende Weise interpretieren: in[B] vereinigt alle Informationen, welche die Vorgänger vom Basisblock $B$ verlassen; out[B] enthält die Informationen,

die den Basisblock $B$ verlassen, die demnach entweder in ihm erzeugt wurden (gen[B]), oder, falls sie aufgrund des Kontrollflusses $B$ erreichten (in[B]), nicht durch eine Anweisung in $B$ zerstört wurden (kill[B]).

Wird ein Programm durch $n$ Basisblöcke beschrieben, so erhält man $2n$ Gleichungen. Listing 3.6 beschreibt einen Algorithmus, wie man diese $2n$ Gleichungen lösen kann.

```
void computeInOut(FlowGraph, gen[], kill[]) {
    foreach B ∈ FlowGraph do {
        out[B] := gen[B];
    }
    change := true;

    while (change) do {
        change := false;
        foreach B ∈ FlowGraph do {
            in[B] := ⋃_{P ∈ predecessor(B)} out[P]
            oldout := out[B];
            out[B] := gen[B] ∪ (in[B] − kill[B]);
            if (out[B] ≠ oldout) {
                change := true;
            }
        }
    }
}
```

Listing 3.6: Algorithmus zur Bestimmung der Mengen *in[B]* und *out[B]*

Eine genaue Analyse dieses Algorithmus würde den Rahmen dieser Arbeit überspannen. Dennoch einige Anmerkungen:

Die Mengen out[] werden zunächst unter der Annahme initialisiert, dass bisher keinerlei Informationen einen Basisblock von außen erreicht haben, d.h. einen Basisblock $B$ verlassen ausschließlich die Informationen, welche in ihm erzeugt wurden (out[B] := gen[B]). In jedem Schleifendurchlauf werden schließlich die Mengen in[] und out[] angenähert, indem der Algorithmus Informationen solange an andere Basisblöcke weitergibt, bis sie von diesen zerstört werden. Nachdem eine Information von einem Vorgängerblock an seinen Nachfolger erfolgreich weitergegeben wurde, wird sie diesem nie wieder entzogen. Werden irgendwann keine neuen Informationen weitergeleitet, d.h. alle out[B]-Mengen bleiben nach einem Schleifendurchlauf unverändert, ist die Berechnung beendet. Um die Effizienz des Algorithmus zu steigern,

werden die Mengen gen[], kill[], in[] und out[] wie auch schon bei der Dominatorenberechnung (vgl. Listing 3.3) als Bitvektoren repräsentiert.

### 3.3.1 Verfügbare Definitionen

Eine *Definition* einer Variablen `x` ist eine Anweisung, welche `x` einen Wert zuweist oder zuweisen kann. Die häufigste Art von Definitionen sind Zuweisungen der Form `x = 10` oder `x = foo()`.

Vor diesem Hintergrund versteht man unter den *verfügbaren Definitionen* einer Variablen `x` an einer beliebigen Stelle $p$ im Programmcode, alle Definitionen der Variablen `x`, die den Punkt $p$ erreichen. Die Anzahl, der an diesem Punkt verfügbaren Definitionen, hängt davon ab, wie viele unterschiedliche Kontrollpfade diesen Punkt erreichen und wie oft die Variable auf diesen Pfaden neu definiert wird. Interessante Stellen $p$ sind meist die Stellen, an denen eine Variable benutzt, d.h. gelesen, wird. Im Compilerbau benötigt man diese Informationen, um herauszufinden, ob man beispielsweise eine Variable durch eine Konstante ersetzen kann. Im Rahmen dieser Arbeit werden – wie man später noch sehen wird – diese Informationen benötigt, um die Parameter einer Schleife zu initialisieren, anhand derer die maximale Iteration der Schleife errechnet wird.

#### Berechnung

Sollen alle verfügbaren Definitionen der Variablen `x` an Position $p$ ermittelt werden, kann man folgendermaßen vorgehen:

Zunächst wird der Basisblock $B$ gesucht, der Position $p$ enthält. Dieser Block wird bis Position $p$ durchlaufen und gleichzeitig nach Definitionen von `x` durchsucht. Hat man tatsächlich eine oder mehrere Definitionen in diesem Block gefunden, so merkt man sich die Letzte, denn diese ist gleichzeitig die einzige Definition, die an Position $p$ verfügbar ist (alle anderen werden von ihr zerstört).

Hat man keine Definition von `x` in $B$ gefunden, so betrachtet man die Menge in[B]. Sie enthält alle Definitionen, die aufgrund des Kontrollflusses $B$ erreichen können. Nachdem, wie wir wissen, $B$ bis zum Punkt $p$ keine Definitionen von `x` zerstört, sind alle in in[B] enthaltenen Definitionen von `x` am Punkt $p$ verfügbar und man hat somit die Menge der verfügbaren Definition von `x` gefunden.

Die Mengen in[] können mit dem Algorithmus aus Listing 3.6 berechnet werden. Nun stellt sich die Frage, wie man anhand der Java-Bytecodes die Mengen gen[] und kill[] ermittelt. Zuweisungen an lokale Variablen werden im Java-Bytecode nur durch die Instruktionen `ASTORE`, `DSTORE`, `FSTORE`,

`ISTORE`, `LSTORE` und `IINC` vollzogen. Daneben existieren die Instruktionen `PUTSTATIC` und `PUTFIELD`, die Werte einer Klassen- oder Membervariablen zuweisen. Zuweisungen an die einzelnen Elemente von Arrays wurden nicht näher beleuchtet, so dass die Instruktionen {`A`, `B`, `C`, `D`, `F`, `L`, `I`, `S`}`ASTORE` außer Acht gelassen werden konnten.

Um gen[] zu ermitteln, läuft man einmal komplett über alle Basisblöcke $B$ eines Flussgraphen und sucht gezielt nach den oben genannten Bytecodeinstruktionen. Jede dieser Instruktionen in $B$ erzeugt einen Eintrag in gen[B]. Daneben muss beachtet werden, dass jeder Methodenparameter einen Eintrag im Startknoten (genauer gen[entry]) erzeugt. Im zweiten Durchlauf berechnet man die Menge kill[]. Man geht hierbei nach folgender Regel vor: Beschreibt $D_x$ die Menge aller Definitionen von `x`, so zerstören die Definitionen $d_1$ bis $d_n$ in $B$ alle übrigen Definitionen von `x`. Daher entspricht kill[B] $= D_x - \{d_1, ..., d_n\}$.

### Anmerkungen

Bleibt nun noch zu klären, ob das errechnete Wissen über die verfügbaren Informationen auch wirklich exakt genug ist. Betrachtet man beispielsweise nun den durch eine if-Anweisung erzeugten Subgraph aus Abbildung 3.3 und die mit Hilfe Listing 3.6 berechneten Mengen.

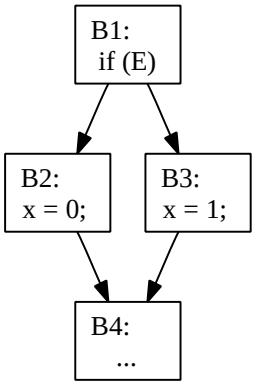



Abbildung 3.3: **if** $(E)$ **then** B1 **else** B2

Man beachte, dass an Basisblock B4 sowohl die Informationen aus Basisblock B2, als auch B3 propagiert werden, da der Ausdruck $E$ der if-Anweisung aus B1 uninterpretiert bleibt und so zur Analysezeit beiden Pfade, sowohl durch B2, als auch B3, zugelassen werden, obwohl zur Laufzeit nur einer von Beiden durchlaufen wird.

Die Menge von Definitionen, die einen Punkt erreichen können, ist somit nur eine Abschätzung der tatsächlichen Menge. Damit aufgrund des errech-

neten Wissens keine Missinterpretationen des Programms entstehen, muss diese Abschätzung *sicher* sein.

Algorithmus 3.6 approximiert demnach die zu errechnenden Mengen grösser, als sie in Wirklichkeit sind: Besteht B2 lediglich aus `x = 0;`, B3 aus `x = 1;` und $E$ wäre zur Laufzeit immer **true**, so erreichen, laut dem oben beschriebenen Algorithmus, beide Definitionen den Anfang von B4, obwohl tatsächlich lediglich die Definition `x = 0;` gültig sein dürfte.

Allerdings stellt dies kein Problem dar. Die verfügbaren Definitionen werden später dazu genutzt, alle möglichen Werte für die Eingabeparameter einer Schleife zu bestimmen, um aus ihnen den maximalen Schleifendurchlauf zu errechnen. *Zu viele* verfügbare Definitionen führen an dieser Stelle höchstens dazu, dass die maximale Anzahl an Schleifendurchläufen im schlimmsten Falle als zu hoch bestimmt wird und die daraus resultierende WCET dieser Schleife sich als schlechter herausstellt, als sie tatsächlich ist. Wie aber bereits in Kapitel 2.1 diskutiert, ist eine solche Abschätzung im Sinne der WCET *sicher* und somit ausreichend für alle folgenden Betrachtungen.

# Kapitel 4

# WCET-Analyse

Wie bereits in Kapitel 2.1 erläutert, interessiert man sich bei der WCET eines echtzeitfähigen Tasks im Wesentlichen für die Dauer des längsten bzw. teuersten Ausführungspfads seiner Arbeitsphase (vgl. 2.2.2). Wie ebenfalls diskutiert, möchte man keine reinen Messmethoden einsetzen, weil diese nur eine begrenzte Sicherheit darüber liefern können, ob tatsächlich der längste Ausführungspfad des Programms gemessen wurde. Vor diesem Hintergrund möchte man Verfahren einsetzen, die den Programmcode des Tasks analysieren und anhand der Analyseergebnisse die Zeit des längsten Ausführungspfads (WCET) *berechnen*.

Je komplexer und umfangreicher das zu analysierende Programm ist, desto deutlicher wird, dass diese Berechnungen nicht ohne Unterstützung seitens eines Werkzeugs durchgeführt werden können. Um sich der Problematik, vor die der automatisierte Berechnungsvorgang gestellt wird, bewusst zu werden, muss man sich zunächst klarmachen, von welchen Faktoren die WCET eines Programms eigentlich abhängt.

Das erste Problem, auf das man stößt, ist, dass man bestimmen muss, wie lange eine beliebige Java-Anweisung auf einem konkreten Rechner benötigt. Es genügt jedoch nicht, ausschließlich den Quellcode zu analysieren, da man sich nicht sicher sein kann, in welche einzelnen Bytecodeinstruktionen eine Ausweisung übersetzt wird. Bei der Bytecodebetrachtung stellt man allerdings wiederum fest, dass die Ausführungsumgebung die Dauer gleicher Bytecodes oder gleicher Bytecodesequenzen an unterschiedlichen Stellen im Programm beeinflussen kann. Sowohl Optimierungen der Laufzeitumgebung als auch begünstigende Cacheeffekte können zur Folge haben, dass einzelne Instruktionen je nach Kontext unterschiedlich schnell ausgeführt werden. Eine Optimierung könnte beispielweise darin bestehen, dass eine Variable als *Laufzeitkonstante* (vgl. [Phi04, JIT technologies]) identifiziert wird und somit alle lesenden Zugriffe auf „diese Variable“ beschleunigt werden; diese

Optimierung wirkt sich allerdings nicht auf die gesamte Lebenszeit, sondern erst im Laufe der Zeit, positiv auf das Laufzeitverhalten der Anwendung aus. Eine weitere Optimierung könnte darin bestehen, dass eine Methode direkt an der Aufrufstelle expandiert (*„geinlinet“*) wird und daher der Methodenaufruf schneller von Statten geht, als ein ähnlicher Methodenaufruf in einem anderen Kontext. Natürlich beschreiben diese beiden Beispiele Fälle, in denen sich die Laufzeit des Programms und somit auch seine WCET verbessert und man weiterhin auf der sicheren Seite ist, wenn man eine Analyse durchführt, die all diese Optimierungen vernachlässigen. Je genauer man aber die WCET eines Programms berechnen möchte, desto genauer muss man sich mit dieser Thematik auseinandersetzen.

Der zweite wichtige Faktor von dem die WCET abhängt, ist die Anzahl der wiederholten Ausführungen einer Bytecodesequenz. Oder in anderen Worten ausgedrückt: Man muss herausfinden, wie häufig jede Schleife im schlechtesten Fall durchlaufen wird. Da man aber aufgrund der erst genannten Problematik den Bytecode und nicht den Quellcode analysiert, steht man zunächst vor dem Problem, alle Schleifen erkennen zu müssen. Da wie bereits in Kapitel 3.1 diskutiert, bei der Umwandlung von Quell- nach Bytecode alle syntaktischen Strukturen der Programmiersprache Java verloren gehen, muss der Bytecode zunächst in eine adäquate Repräsentation umgewandelt werden, welche die Schleifensuche erst ermöglicht. Hat man alle Schleifen gefunden, müssen im nächsten Schritt die maximalen Schleifendurchläufe bestimmt werden. Doch auch dies stellt sich aus mehreren Gründen problematisch dar:

- Zunächst muss man herausfinden, von welchen Parametern die Funktion abhängt, welche die Anzahl der Schleifendurchläufe bestimmt.
- Des Weiteren ist diese Funktion nicht unbedingt linear, so dass allein die Betrachtung der Grenzen im Allgemeinen *nicht* ausreicht.
- Auch ist nicht ausgeschlossen, dass das Inkrement der Schleife in jedem Schleifendurchlauf konstant bleibt. Auch dahinter könnte sich eine komplexe Funktion verbergen.
- Und schließlich stellt man oft genug fest, dass der Programmcode zu wenig Informationen liefert, um den maximalen Schleifendurchlauf zu errechnen. Oft ist der Wertebereich der Eingabeparameter der Schleife nicht ermittelbar, so dass die gesamte Schleifenanalyse ohne zusätzliches Wissen nicht berechenbar ist.

Im Allgemeinen ist es sogar unmöglich die WCET nur anhand des Programmcodes zu bestimmen. Auch wenn alle Schleifen begrenzt sind, also kei-

ne Endlosschleifen vorkommen, ist es nicht immer möglich eine obere Grenze für die Anzahl der Schleifendurchläufe zu bestimmen. Daher muss es eine Möglichkeit geben, dass ein Programmierer seinen Code annotieren kann, um die benötigten Informationen bereitzustellen. Das Analyseprogramm muss hierfür geeignete Fehlermeldungen generieren, um den Programmierer darauf hinzuweisen, *welche* Informationen *wo* fehlen. Diese Informationen müssen schließlich eingelesen und bei der Analyse berücksichtigt werden.

In einem ähnlichen Kontext sind rekursive Methodenaufrufe zu sehen. Auch wenn man leicht herausfinden kann, welche Methoden sich rekursiv aufrufen, bzw. über eine Aufrufkette rekursiv aufgerufen werden, so kennt man nicht gleichzeitig deren maximale Rekursionstiefe. Die Rekursionstiefe ist allerdings entscheidend für die Laufzeit einer Methode, so dass sie entweder ermittelt, was im Allgemeinen unmöglich ist, oder annotiert werden muss.

Zu guter Letzt steht man vor dem Problem virtueller Methodenaufrufe. Deklariert der Programmierer eine Methode nicht ausdrücklich als `static` oder `final`, kann nur im seltensten Fall zur Übersetzungszeit der Methodenaufruf aufgelöst werden. Man steht demnach bei der WCET-Analyse vor dem Problem, dass man keine genaue Aussage darüber treffen kann, welche Methode zur Laufzeit tatsächlich ausgeführt wird. Daher müssen alle in Frage kommenden Methoden betrachtet und diejenige ausgewählt werden, welche den größten Beitrag zur WCET leistet.

Erst wenn man alle diese Probleme bewältigt hat, kann man die WCET eines Programms bestimmen. Zusammengefasst bilden folgende Punkte den Anforderungsrahmen für ein WCET-Analyseprogramm:

- Bestimmung der Ausführungszeit einzelner Instruktionen
- Erkennung aller Schleifen und Bestimmung deren maximaler Iterationen
- Bereitstellung einer flexiblen Schnittstelle für Annotationen
- Ermittlung der Aufrufhierarchie und schließlich
- Berechnung der maximalen Ausführungszeit des Programms

## 4.1 Architektur

Abbildung 4.1 zeigt einen groben Überblick über die wichtigsten Komponenten der WCET-Analysearchitektur. Der Entwickler einer echtzeitfähigen Java-Anwendung übersetzt seinen Quellcode mit einem Java-Compiler seiner

Wahl und erhält daraus den Bytecode seiner Anwendung. Der Annotationsmechanismus ist so gewählt, dass er wie der Quellcode ebenfalls in das Kompilat mit einfließt. Diese Annotationen sind Informationen über Variablen und Methoden des Programms, die das Analyseprogramm nicht ermitteln kann und somit vom Entwickler angegeben werden müssen. Solche Informationen sind beispielsweise Wertebereiche für Eingabeparameter des Programms. Obgleich sie nirgends im Programmcode stehen, sind sie dem Entwickler wohlbekannt und entscheidend für das Laufzeitverhalten der gesamten Anwendung. Obwohl diese Annotationen in der Skizze extra hervorgehoben sind, werden sie vollständig in den Quellcode der Anwendung integriert.

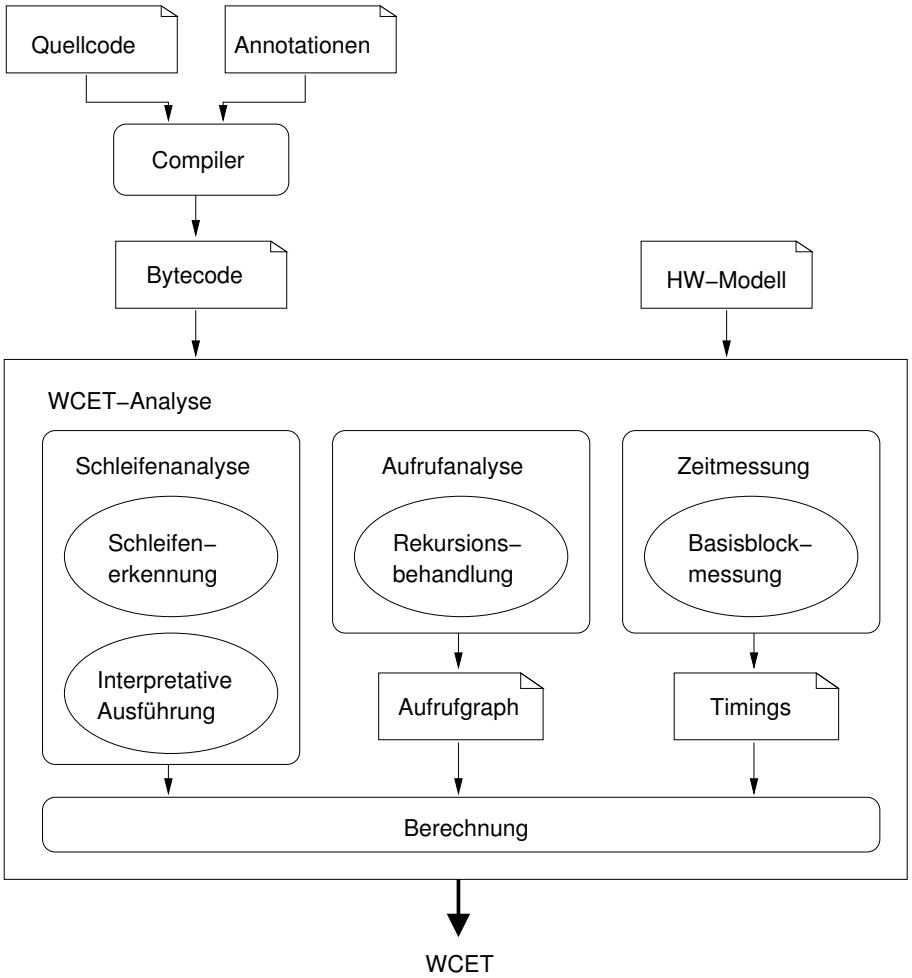



Abbildung 4.1: Überblick über die Architektur

Da neben dem Programmcode auch die Hardwarearchitektur das Laufzeitverhalten der Anwendung beeinflusst, fließt neben dem Bytecode auch

das Hardwaremodell der Zielarchitektur in die WCET-Analyse ein.

Sobald der Bytecode eingelesen wurde, beginnt die eigentlich Analyse des Programms. Diese ist in drei Hauptkomponenten unterteilt; jede dieser Komponenten greift während der Analyse auf die vom Programmierer annotierten Informationen zurück, sofern sich die benötigten Informationen nicht aus dem Programmcode extrahieren lassen.

1. **Schleifenanalyse**
   Die *Schleifenanalyse* bildet einen wichtigen Faktor bei der Bestimmung der WCET. Sie setzt sich zusammen aus:

   (a) Schleifenerkennung und

   (b) interpretativer Ausführung.

   Im Zuge der *Schleifenerkennung* werden Kontrollflussgraphen des Programms auf Zyklen analysiert und so alle Schleifen erkannt. Anschließend wird der maximale Durchlauf jeder Schleife mithilfe interpretativer Ausführung berechnet.

2. **Aufrufanalyse**
   Die *Aufrufanalyse* ermittelt den Aufrufgraph der Anwendung und analysiert anhand dieses Graphen Rekursionen und virtuelle Methodenaufrufe.

3. **Zeitmessung**
   Während der *Zeitmessung* werden die Zeiten jedes einzelnen Basisblocks der Anwendung ermittelt. Sie bilden die quantitative Grundlage für die anschließende Berechnung.

Nachdem die Analyse des Programms abgeschlossen ist, werden alle ermittelten Informationen zusammengetragen und die WCET des gesamten Programms errechnet. Hierbei wird das Problem der WCET-Bestimmung auf ein *Integer-Linear-Programming*-Problem abgebildet und mit Standardwerkzeugen gelöst.

Im Folgenden werden die einzelnen Arbeitsphasen der Analyse genauer betrachtet und an geeigneten Stellen alternative Lösungsansätze diskutiert.

## 4.2 Kontrollflussbestimmung

Nachdem der Kontrollfluss eines Programms neben der Hardware, auf der das Programm ausgeführt wird, entscheidend für das Laufzeitverhalten dieses

Programms ist, muss im Vorfeld jeder weiteren Analyse das Hauptaugenmerk auf dessen Bestimmung gerichtet sein. Zur Repräsentation des Kontrollflusses werden, wie in Kapitel 3.1.2 bereits vorgestellt, gerichtete Graphen, die sogenannten Kontrollflussgraphen, eingesetzt. Alle für die Konstruktion dieser Graphen benötigten Informationen müssen den Classfiles (vgl. Kapitel 2.3.1) des zu analysierenden Programms entnommen werden. Hierfür wurde in dieser Arbeit die im *Apache Jakarta Project* (vgl. [jak]) entwickelte Bibliothek *BCEL* verwendet.

### 4.2.1 BCEL

Die ***B**yte-**C**ode-**E**ngineering-**L**ibrary (BCEL)* bietet alle nötige Funktionalität für den Zugriff auf Java-Classfiles und deren Manipulation. Dabei liest die Bibliothek zunächst eine im Binärformat codierte Klasse ein und repräsentiert alle in ihr definierten Methoden und Felder in Form eines `JavaClass`-Objekts (vgl. A.1). Dieses wiederum ermöglicht über `Method`-Objekte den Zugriff auf alle in der Klasse definierten Methoden. Für die Kontrollflussbestimmung einer bestimmten Methode interessiert man sich in erster Linie für das `Code`-Attribut dieser Methode – es gewährt Zugriff auf dessen Bytecodestrom. Jedr einzelne Bytecode dieses Stroms lässt sich in Form von `Instruction`-Objekten darstellen, die den Opcode und die Parameter der Bytecodeinstruktion speichern. Wichtige Parameter einzelner Bytecodeinstruktionen bilden im Kontext der Kontrollflussberechnung beispielsweise die Sprungziele bedingter und unbedingter Sprunginstruktionen.

### 4.2.2 Kontrollflussgraph

Der mit Hilfe der BCEL-Bibliothek ausgelesene `Instruction`-Objektstrom lässt sich einfach mit dem in Kapitel 3.1.1 beschriebenen Algorithmus zur Basisblockerzeugung in einzelne Basisblöcke unterteilen. Die auf diese Weise berechneten Basisblöcke werden schließlich mit Hilfe der in Kapitel 3.1.2 vorgestellten Kantenregel zu einem gerichteten Graphen, dem Kontrollflussgraphen der zu analysiereden Methode, verbunden. Dieser Graph bildet nun die Grundlage aller nachfolgende Untersuchungen.

## 4.3 Schleifen

Nachdem der Kontrollflussgraph einer Methode aufgestellt worden ist, richtet sich das Augenmerk nun auf mögliche Schleifen innerhalb dieses Graphen.

Für die WCET-Analyse spielt die maximale Anzahl möglicher Schleifeniterationen eine sehr wichtige Rolle. Daher gilt es zunächst, alle möglichen Schleifen zu erkennen und auf ihren maximalen Durchlauf zu analysieren.

Betrachten wir zunächst folgendes Listing und die zugehörigen Flussgraphen aus Abbildung 4.2. Sie zeigen die Grundformen der beiden einfachsten Schleifentypen: **do-while** und **while**[1].

```
i = 0;                  i = 0;
do {                    while (i < 10) {
    doSomething();          doSomthing();
    i++;                    i++;
} while (i < 10);       }
```

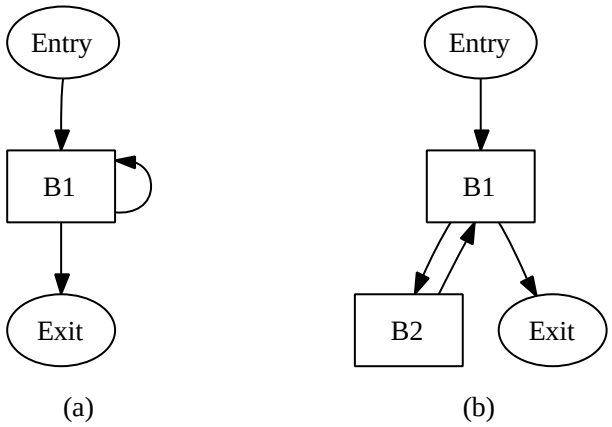



Abbildung 4.2: Grundformen von *do-while*- und *while*-Schleifen

Während die do-while-Schleife (Abbildung 4.2 (a)) bereits durch einen einzigen Knoten mit entsprechender Rückwärtskante repräsentiert werden kann, benötigt die while-Schleife genau zwei Knoten (Abbildung 4.2 (b)). Der Grund hierfür liegt darin, dass in der do-while-Schleife sowohl die Abbruchbedingung als auch das Schleifeninnere innerhalb eines Basisblock kodiert werden kann, während die while-Schleife einen zusätzlichen Block für die Abbruchbedingung benötigt. Nur im Falle einer nicht erfüllten Abbruchbedingung wird in der do-while-Schleife zum Schleifenanfang gesprungen; ist die Abbruchbedingung erfüllt, erfolgt kein Sprung, sondern die erste Anweisung *nach* der Schleife wird ausgeführt. In der while-Schleife hat sowohl

[1]for-Schleifen werden durch den Compiler gewöhnlich auf while-Schleifen abgebildet und müssen daher nicht gesondert betrachtet werden.

eine erfüllte Abbruchbedingung einen Sprung zum Schleifenausgang, als auch das Erreichen der letzten Anweisung des Schleifeninneren einen Sprung zum Schleifenkopf zur Folge.

Um die Anzahl der maximalen Iterationen einer Schleife zu ermitteln, muss man zunächst herausfinden, was die Abbruchbedingung ist und von welchen Parametern sie abhängt. Wenn man dies herausgefunden hat, betrachtet man das Schleifeninnere und versucht zu ermitteln, wie diese Schleifenparameter in jedem Schleifendurchlauf modifiziert werden. Dies bildet schließlich die Grundlage für die Berechnung des maximalen Durchlaufs.

Der einfache Aufbau der gerade beschriebenen Schleifen, erweckt vielleicht den Eindruck, dass diese Berechnung einfach durchzuführen ist. In der Regel sind Schleifen jedoch bei Weitem nicht so einfach gebaut. Gewöhnlich existieren Verzweigungen innerhalb des Schleifeninneren, so dass nur die aller einfachste do-while-Schleife mit nur einem einzigen Basisblock auskommt. Auch die Abbruchbedingung besteht gewöhnlich aus mehreren Bedingungen – jede führt zu einem zusätzlichen Knoten im Kontrollflussgraphen und zu zusätzlichen Schleifenparametern. Zu guter Letzt führen `continue`- und `break`-Anweisungen im Schleifeninneren dazu, dass komplexe Schleifengebilde über mehrere Rückwärtskanten verfügen und der Schleifenausgang nicht nur über eine Kante von einen Knoten der Abbruchbedingung, sondern auch über eine Kante aus dem Schleifeninneren erreicht werden kann. Dies führt gleich zu mehreren Problemen; eines sei an dieser Stelle allerdings besonders hervorgehoben: Da einzig der Kontrollflussgraph für alle folgenden Analysen zur Verfügung steht, besteht eine große Herausforderung darin, den Typ der Schleife zu bestimmen. Einfach ausgedrückt muss man zunächst herausfinden, ob die Blöcke der Schleifenbedingung „oben“ oder „unten“ zu finden sind, um sich anschließend Gedanken machen zu können, was die Schleifenparameter sind.

Im Folgenden wird der Ablauf der Schleifenanalyse genau beschrieben, welche Probleme bei der Analyse entstehen und wie sie gelöst werden können.

### 4.3.1 Modifizierte Schleifenerkennung

Wie bereits in Kapitel 3.2.3 beschrieben, lassen sich Schleifen über ihre Rückwärtskanten aufspüren. Hierfür wird als erstes ein dem Flussgraph entsprechender Dominatorbaum (vgl. Kapitel 3.2.2) bestimmt, der die Dominanzrelation auf dem Flussgraphen repräsentiert. Die Dominanzrelation ermöglicht auf einfache Weise, alle *Rückwärtskanten* des Flussgraphen abzulesen. Die so ermittelten Rückwärtskanten bilden wiederrum den Einstiegspunkt für die Konstruktion *natürlicher Schleifen* (vgl. hierzu nochmals den in Kapitel 3.2.3 beschriebenen Algorithmus).

Diese Vorgehensweise ist bestens geeignet, alle Knoten einer Schleifen mit nur einer Rückwärtskante zu finden. Jedoch müssen an dieser Stelle Auswirkungen einer `continue`-Anweisung auf den Kontrollflussgraphen beachtet werden. Eine `continue`-Anweisung innerhalb einer Schleife hat im Falle einer while-Schleife die sofortige Auswertung der Schleifenabbruchbedingung, im Falle einer do-while-Schleife den Rücksprung zum Schleifenanfang zur Folge. Das bedeutet, dass jede `continue`-Anweisung zu einer zusätzliche Rückwärtskante im Flussgraphen führt, ohne dass dabei eine neue Schleife entsteht. Der Schleifensuchalgorithmus aus Kapitel 3.2.3 *erkennt* daher zwei Schleifen, kann allerdings nicht feststellen, dass beide zu einem gemeinsamen Konstrukt auf Quellcodeebene gehören.

Um dieses Problem zu beheben, müssen alle erkannten Schleifen noch einmal betrachtet werden. Führen ihre Rückwärtskanten zu demselben Knoten, so müssen beide natürlichen Schleifen *verschmolzen* werden, d.h. ihre Knotenmengen werden vereinigt und alle weiteren Untersuchungen betrachten nur noch die zusammengeführte Knotenmenge.

### 4.3.2 Typbestimmung

Dieser Abschnitt beschäftigt sich mit der Typbestimmung endlicher Schleifen. Man möchte allein anhand der Topologie des Kontrollflussgraphen folgende Eigenschaften herausfinden:

- **Schleifentyp**: Liegt eine while- oder do-while-Schleife vor?
- **Knotenklassifikation**:
  - Welche Knoten gehören zur Abbruchbedingung: **Condition-Knoten**,
  - welche hingegen zum Schleifeninneren: **Body-Knoten**?
  - Gibt es unter den Body-Knoten bestimmte Knoten, hier als **Break-Knoten** bezeichnet, die ebenfalls zum Verlassen der Schleife führen?

#### Schleifentyp

Um den Typ einer Schleife zu bestimmen, betrachtet man den Knoten mit Post-Nummer 1, den *tiefsten* Knoten der Schleife. Verfügt dieser Knoten nur über eine Kante, so ist diese Kante zwangsläufig eine Rückwärtskante der Schleife und es handelt sich um eine while-Schleife. Der Knoten mit Post-Nummer 1 einer endlichen do-while Schleife besitzt hingegen zwei Kanten:

Die eine führt zum Schleifenanfang, die andere zu einem Knoten außerhalb der Schleife. Dennoch kann man in einem solchen Fall nicht mit 100%-iger Sicherheit sagen, dass es sich um eine do-while-Schleife handelt. Betrachten wir folgendes Listing:

```
while (loopCondition) {
    <do_something>

    if (breakCondition) {
        [do_something_before_break]
        break;
    }
}
```

Je nachdem, ob das Listing `do_something_before_break`-Anweisungen enthält oder nicht, entsteht einer der beiden Graphen aus Abbildung 4.3.

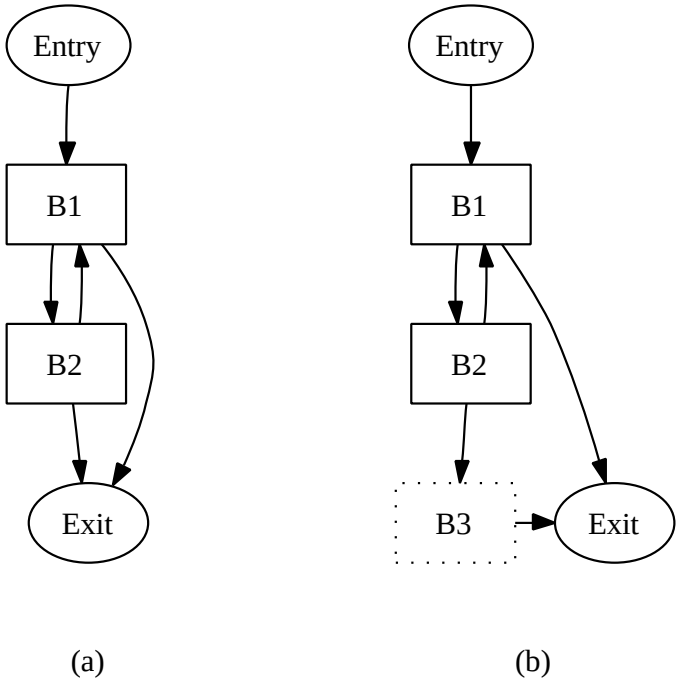



Abbildung 4.3: Links eine Schleife ohne, rechts mit `do_something_before_break`-Anweisung

Falls, wie in Abbildung 4.3 (b), ein Knoten B3 existiert, der eine Kante zum Ausgangsknoten der Schleife besitzt, kann man sicher gehen, dass

B2 einem Break-Knoten entspricht und es sich bei dieser Schleife um eine while-Schleife handelt. Andernfalls ist der Schleifentyp nicht immer korrekt bestimmbar, da auch das anstehende Listing zu dem gleichen Graphen aus Abbildung 4.3 (a) führen kann.

```
do {
    if (breakCondition) {
        break;
    }
    <do_something>
} while (loopCondition);
```

Der in dieser Arbeit verwendete Compiler generierte allerdings in beiden Fällen Code, der einen zusätzlichen Basisblock, wie den aus Abbildung 4.3 (b), zur Folge hatte. Aus diesem Grund konnten alle Schleifen korrekt erkannt werden.

### Knotenklassifikation

Nachdem der Schleifentyp bestimmt worden ist, stellt sich die Frage, welche Knoten zur Abbruchbedingung (*Condition-Knoten*) und welche zum Schleifeninneren (*Body-Knoten*) gehören. Eine Schleifenabbruchbedingung besteht gewöhnlich aus mehreren booleschen Ausdrücken, die mit logischen Operatoren miteinander verbunden sind. Jeder dieser Ausdrücke wird von Compiler übersetzt und es entsteht ein Flussgraph, in dem jeder Ausdruck einem eigenen Knoten entspricht. Da die Schleifenparameter genau in diesen Knoten abgefragt werden und so die maximale Anzahl der Iterationen einer Schleife beeinflussen, muss man zunächst herausfinden, welche Knoten zur Abbruchbedingung gehören.

Betrachten wir zunächst Abbildung 4.4, die den typischen Aufbau eines Graphen einer while-Schleife zeigt. Die Schleife wird immer im Knoten B1 betreten und es werden so viele boolesche Ausdrücke ausgewertet wie nötig, um entscheiden zu können, ob das Schleifeninnere mit B3 betreten oder die Schleife verlassen wird. Dies sei durch die gepunktete Kante zwischen B1 und B2 angedeutet. Die beiden zu B1 führenden Kanten verdeutlichen, dass der erste Knoten des Schleifeninneren über verschiedene Kanten der Condition-Knoten erreicht werden kann. Es ist auch nicht zwingend notwendig, dass einzig der letzte Condition-Knoten, hier B2, eine Kante zum Schleifenausgang enthält – auch dort könnte es mehrere Kanten geben.

Die Idee für die Klassifikation der Knoten einer while-Schleife beruht auf folgender Beobachtung: Definiert man den ersten Knoten der Schleife als *Condition-Kopf* und den ersten Knoten des Schleifeninneren als *Body-Kopf*,

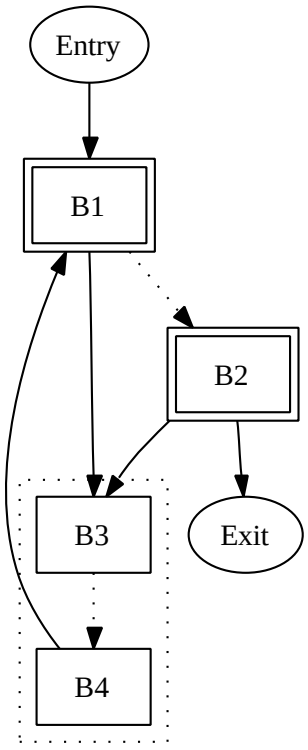



Abbildung 4.4: While-Schleife

so dominieren *ausschließlich* diese beiden Knoten alle Knoten der Schleife unterhalb von sich selbst. Der Body-Kopf ist somit der erste Knoten *nach* dem Condition-Knoten, der alle nachfolgenden Knoten der Schleife dominiert.

Vor diesem Hintergrund klassifiziert man die Knoten einer while-Schleife nach folgendem Muster: Man traversiert die Schleife in Preorder und sucht den ersten Knoten, der nach dem Schleifenkopf alle weiteren Knoten dominiert. Alle Knoten, die eine größere Postnummer haben als er selbst, gehören somit zu den Condition-Knoten, als Knoten mit einer kleineren Nummer zu den Body-Knoten.

Die Klassifikation der Knoten einer do-while-Schleife verfolgt eine ähnliche Idee. Betrachten wir aber erst Abbildung 4.5, die den typischen Aufbau eines Graphen einer do-while-Schleife zeigt. Die Anordnung der Condition-Knoten und Body-Knoten ist im Vergleich zur while-Schleife eine andere. Die Schleife wird sofort mit dem ersten Body-Knoten B1 betreten. Erst nachdem ein Pfad über das Schleifeninnere den Knoten B3 erreicht, wird der erste boolesche Ausdruck der Abbruchbedingung berechnet. Wie schon in Abbildung 4.4 soll die gepunktete Kante zwischen B3 und B4 die Möglichkeit mehrere Condition-Knoten andeuten. Es können sowohl mehrere Kanten von den

Condition-Knoten zum Ausgang der Schleife führen, als auch wieder zurück zum Body-Kopf – auch wenn hier nur eine Kante von B4 nach B1 skizziert wurde.

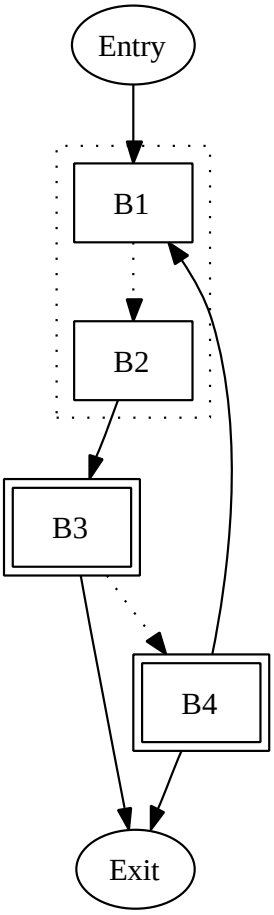



Abbildung 4.5: Do-While-Schleife

Um die Knoten einer do-while-Schleife zu klassifizieren, verfolgt man das Ziel den Condition-Kopf zu finden. Von diesem Knoten weiß man, dass *nur* er alle darunter liegenden Condition-Knoten dominiert. Man traversiert daher die Knoten der Schleife in Postorder und sucht den letzten Knoten des Schleifeninneren. Von ihm weiß man wiederum, dass er der erste Knoten ist, der alle Knoten unterhalb von sich selbst *nicht* dominiert. Dies hingegen bedeutet, dass der Knoten mit der um eins kleineren Postordernummer als der soeben gefundene der Condition-Kopf ist. Den Body-Kopf kennt man ohnehin, er entspricht dem ersten Knoten der Schleife.

Zu guter Letzt interessiert man sich noch für die Break-Knoten der Schleife. Sie sind innerhalb der Body-Knoten zu finden und entsprechen Knoten, die

eine Kante zu einem Knoten außerhalb der Schleife besitzen. Auf diese Weise lassen sie sich auch finden: Man betrachtet alle Nachfolger der Body-Knoten und markiert all die als Break-Knoten, die einen Nachfolger außerhalb der Schleife besitzen. Sie sind gerade im Hinblick auf unendliche Schleifen mit einer `break`-Anweisung als Schleifenabbruchkriterium von Interesse.

### 4.3.3 Auswertbarkeit

Nachdem der Typ der Schleife bestimmt ist und alle Knoten klassifiziert wurden, beginnt die Analyse der Schleifenabbruchbedingung. Zunächst muss geklärt werden, ob die Anweisungen innerhalb der Abbruchbedingung zum Analysezeitpunkt *auswertbar* sind. Um den Begriff der Auswertbarkeit zum Analysezeitpunkt zu veranschaulichen, betrachte man Listing 4.1.

```
public int foo() { ... }

public void bar() {
    [...]

    while (i < 10 && n > foo()) {
        [do something]
    }
}
```

Listing 4.1: Codebeispiel: Auswertbarkeit

Die Abbruchbedingung der while-Schleife besteht aus zwei booleschen Ausdrücken: Der Erste besteht lediglich aus primitiven Objekten, der Variablen `i` und der Konstanten 10; der Zweite hingegen neben der Variablen `n` auch aus dem Methodenaufruf `foo()`. Sobald man herausfindet, welche Werte die Variable `i` annehmen kann, ist der erste Ausdruck bereits zum Analysezeitpunkt *auswertbar*. Um den zweiten Ausdruck auswerten zu können, müsste man nicht nur den Anfangswert der Variablen `n` kennen, sondern zudem herausfinden, welche Rückgabewerte die Methode `foo()` liefert. Diese Methode kann jedoch beliebig komplex sein, so dass ihr Rückgabewert womöglich von Parametern abhängt, die erst zur Laufzeit bekannt werden. Obwohl es mit Sicherheit Fälle gibt, in denen der Rückgabewertebereich einer beliebigen Methode bereits zum Analysezeitpunkt berechenbar ist, bricht die Analyse einer solchen Schleife mit einer entsprechenden Fehlermeldung ab.

Dennoch muss zunächst geklärt werden, ob eine zu analysierende Schleife auswertbar ist. Hierfür betrachtet man den Bytecode der Condition- und

Break-Knoten der Schleife und versucht pro Basisblock die einzelnen booleschen Ausdrücke zu rekonstruieren. Tabelle 4.1 gibt einen Überblick über alle möglichen Komponenten eines booleschen Ausdrucks.

| **Komponente** | **Bedeutung** |
|---|---|
| *ArithmeticOperand* | Arithmetischer Ausdruck |
| *ArrayLength* | Java-Längenoperator für Arrays |
| *ArrayOperand* | Arrayzugriff |
| *ConstantOperand* | Konstante |
| *Conversion* | Cast-Operator |
| *Invoke* | Methodenaufruf |
| *VariableOperand* | Variable |

Tabelle 4.1: Komponenten boolescher Ausdrücke

Die booleschen Ausdrücke werden mit Hilfe einer *top-down* Simulation des Bytecodes errechnet und zu `Condition`-Objekten zusammengeführt. Eine detaillierte Beschreibung des Simulationsvorgangs folgt in Kapitel 4.3.6. Wichtig an dieser Stelle ist noch die Analyse der Auswertbarkeit der `Condition`-Objekten.

`Condition`-Objekten werden während der Simulation immer dann erzeugt, wenn die Simulationseinheit auf eine *bedingte Sprunganweisung* wie IFLT, IFGE, IFGT, IFLE etc. im Bytecodestrom stößt. Die meisten dieser Anweisungen erwarten zwei Parameter auf dem Operandenstack (einige Anweisungen wie z.B. IFNULL erwarten jedoch auch nur einen Parameter), vergleichen diese und entscheiden anhand des Vergleichs ob gesprungen wird. Während der Simulation werden primitive Anweisungen zu komplexeren zusammengefasst und auf den Operandenstack gelegt. Beispielsweise wird die Anweisungsfolge:

```
iconst_2
iload_1
iadd
```

zu einem `ArithmeticOperand`-Objekt zusammengefasst und auf den Stack gelegt. Auf diese Weise können beliebig komplexe Parameter für boolesche Ausdrücke entstehen. Tabelle 4.1 zeigt alle möglichen Parameter.

Jedes dieser Objekte implementiert das Interface `Computable`, das u.a. eine Methode `isComputable(...)` anbietet. Um nun entscheiden zu können, ob die Parameter eines booleschen Ausdrucks zum Analysezeitpunkt auswertbar sind, ruft man rekursiv auf den einzelnen Komponenten eines `Condition`-

Objektes diese Methode auf. Nur wenn dieser rekursive Aufruf zu *true* evaluiert, ist der Gesamtausdruck auswertbar. `ConstantOperand`s evaluieren beispielsweise immer zu *true*, während nicht annotierte (vgl. Kapitel 4.4) `Invoke`-Objekte immer *false* zurückliefern. ArithmeticOperand-Objekte liefern hingegen nur dann *true*, wenn jeder arithmetischer Teilausdruck auswertbar ist, d.h. beispielsweise keine nicht annotierten Methodenaufrufe enthält.

#### Fehlermeldungen

Damit eine Analyse nicht grundlos abbricht und dem Anwender Hinweise gegeben werden, warum eine Schleife nicht auswertbar ist, generiert das Werkzeug passende Fehlermeldungen. Sofern der Java-Quellcode mit Debuginformationen compiliert wurde, enthält jede Fehlermeldung neben einer genauen Fehlerbeschreibung auch den genauen Fehlerort, d.h. Klassenname, Methodenname, Dateiname und Zeilennummer der nicht zu analysierenden Anweisung.

### 4.3.4 Schleifenparameter

Nachdem entschieden wurde, dass alle booleschen Ausdrücke der Abbruchbedingung berechenbar sind, werden alle *Schleifenparameter* bestimmt. Das sind all die Parameter, von denen die Anzahl der Schleifeniterationen abhängt. Die Schleifenparameter bestehen allerdings nicht nur aus den Variablen der Schleifenabbruchbedingung, wie man irrtümlich annehmen könnte – sie sind nur ein Teil von ihnen. Diese Variablen tragen zwar zu der Entscheidung bei, wann eine Schleife abgebrochen wird, jedoch sagen sie nichts darüber aus, auf welche Weise sie in jedem Durchlauf des Schleifeninneren verändert werden. Dies bedeutet, dass zu den Schleifenparametern auch alle Variablen gehören, welche an der Manipulation der Variablen der Schleifenabbruchbedingung beteiligt sind. So zählt neben der Variablen `i` auch die Variable `z` zu der Schleifenparametermenge des folgenden Listing:

```
while (i < 10) {
    [...]
    i += z;
}
```

Für die Bestimmung aller Schleifenparameter geht man daher folgendermaßen vor:

1. Bestimme die Variablen der Schleifenabbruchbedingung und füge sie zur Menge *SP* der Schleifenparameter.

2. Bestimme alle möglichen Ausführungspfade durch das Schleifeninnere.

3. Untersuche jeden Ausführungspfad auf Manipulationen bisher gefundener Schleifenparameter und füge zur Menge *SP* alle Variablen hinzu, die bisher nicht Teil von *SP* sind und sich an der Manipulation eines Elementes aus *SP* beteiligen.

### Ausführungspfad

Unter einem möglichen *Ausführungspfad* einer Schleife versteht man eine Folge von Knoten, die den Schleifenkopf mit einem *Endknoten* über gültige Kanten des Kontrollflussgraphen miteinander verbindet. Die Endknoten einer Schleife sind all die Knoten, welche eine Rückwärtskante der Schleife besitzen. Abbildung 4.6 (a) zeigt eine Schleife mit zwei möglichen Pfaden: B1 → B2 → B3 → B5 und B1 → B2 → B4 → B5. Die Kanten des ersten Pfades sind dabei hervorgehoben.

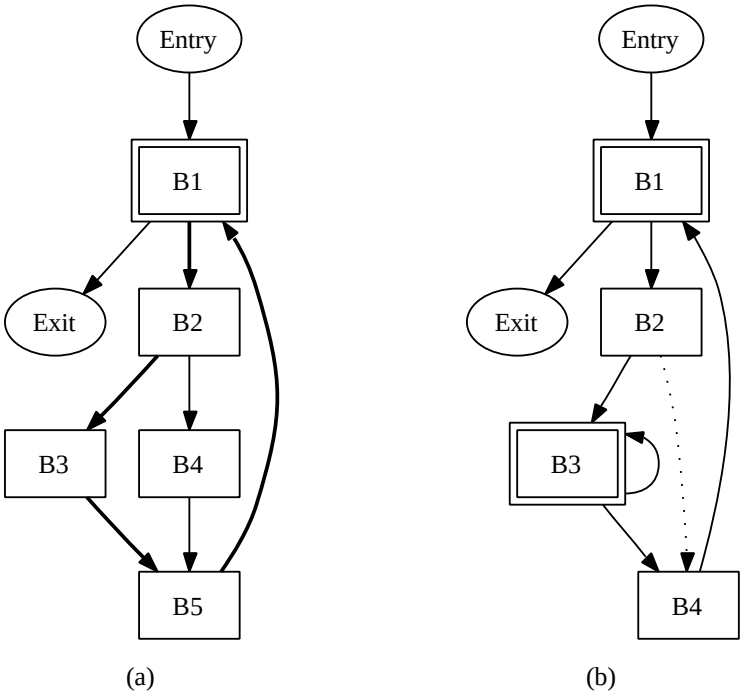



Abbildung 4.6: Pfade verschiedener Schleifen

Um alle Pfade einer Schleife zu finden, traversiert man die Knoten der Schleife in Preorder beginnend ab dem Startknoten. Der Startknoten bildet

dabei den ersten Knoten des ersten noch unvollständigen Pfades. Man betrachtet nun die Nachfolger des momentanen Schlussknotens eines Pfades und vervielfältigt den aktuellen Pfad, falls der Schlussknoten mehr als einen Nachfolger besitzt. Jeder Nachfolgerknoten wird an das Ende des soeben vervielfältigten Pfades gehängt, es sei denn, es handelt sich um den Startknoten der Schleife – in diesem Fall hat man einen vollständigen Ausführungspfad gefunden. Eine gesonderte Behandlung bedarf es auch bei verschachtelten Schleifen. Führt ein Pfad auf den Startknoten einer inneren Schleife, so werden die Kanten der inneren Schleife ausgelassen und einer gesonderten Betrachtung unterzogen (s.u.). Der Ausführungspfad wird hingegen so gelegt, dass die innere Schleife ausgelassen wird und der erste Knoten nach dem Verlassen der inneren Schleife zum Ausführungspfad gehängt wird. Demnach entspricht der einzige Ausführungspfad der Schleife aus Abbildung 4.6 (b): B1 $\rightarrow$ B2 $\rightarrow$ B4, was durch die gepunktete Kante zwischen B2 und B4 angedeutet ist. Die innere Schleife aus B3 wird dabei umgangen.

### Manipulation

Ein Schleifenparameter wird im Schleifeninneren immer dann *manipuliert*, wenn ihm ein neuer Wert zugewiesen wird. Um alle Zuweisungen an Schleifenparameter zu finden, wird die Ausführung jedes Ausführungspfads simuliert (vgl. Kapitel 4.3.6) und dabei `Assignment`-Objekte erzeugt.

`Assignment`-Objekte entstehen immer dann, wenn der Simulator auf `PUTFIELD`-, `PUTSTATIC`-, {`A`, `F`, `D`, `L`, `I`}`STORE`- oder `IINC`-Instruktionen stößt. Ein `Assignment`-Objekt speichert stets Links- und Rechtswert einer Zuweisung. Mögliche Rechtswerte sind alle Operanden aus Tabelle 4.1. Liegt dem Rechtswert einer Zuweisung beispielsweise eine komplexe arithmetische Operation zugrunde, so wird im `Assignment`-Objekte neben dem Zuweisungsziel, ein `ArithmeticOperand` abgespeichert.

### Direkte und indirekte Schleifenparameter

Um *alle* Parameter zu finden, welche die maximale Iterationsanzahl einer Schleife beeinflussen, wird ein *Multipassverfahren* eingesetzt. Nach Betrachtung der Schleifenabbruchknoten kennt man alle Parameter, die direkt an der Iterationsanzahl beteiligt sind – im folgenden *direkte Parameter* genannt. Noch fehlen jedoch die Parameter, welche im Schleifeninneren an der Manipulation der direkten Parameter und somit ebenfalls am Erreichen der Abbruchbedingung beteiligt sind – im folgenden als *indirekte Parameter* bezeichnet. Listing 4.2 beschreibt einen Algorithmus, zur Berechnung aller Schleifenparameter. Der Algorithmus bekommt als Eingabe die Menge aller

```
Set findLoopParameter(Set directParameter, Set execPaths) {
    Set loopParameter := directParameter;
    forall (path ∈ execPaths) {
        change := true;
        while (change) do {
            change := false;
            forall (assignment ∈ path) {
                if (assignment.leftValue ∈ loopParameter) {
                    variables := assignment.
                        rightValueVariables();
                    forall (parameter ∈ variables) {
                        if (parameter ∉ loopParameter) {
                            loopParameter := parameter ∪
                                loopParameter;
                            change := true;
                        }
                    }
                }
            }
        }
    }
    return loopParameter;
}
```

Listing 4.2: Algorithmus zur Berechnung der Schleifenparameter

direkten Parameter (`directParameter`) und die Menge aller Ausführungspfade (`execPaths`), untersucht daraufhin jeden Ausführungspfad der Schleife einzeln und betrachtet auf diesem Pfad jedes gefundene `Assignment`-Objekt (`assignment`). Wird darin ein bereits erkannter Schleifenparameter manipuliert, so untersucht man die Variablen des Rechtswertes der Zuweisung (`variables`) auf ihre Zugehörigkeit zu den aktuellen Schleifenparametern (`loopParameter`). Findet man unter ihnen Variablen, die bislang nicht in der Menge der Schleifenparameter enthalten sind, so werden sie nun in diese Menge aufgenommen.

Wichtig ist noch zu erwähnen, dass für das Aufspüren aller Parameter *ein einziger* Durchlauf nicht ausreicht, weshalb die Hauptschleife nicht nur einmal den Code in lineare Reihenfolge traversiert, sondern solange ausgeführt wird, bis sich keine Veränderungen mehr einstellen. Betrachten wir hierzu

folgende Schleife:

```
while(i < 100) {
    z += 5;
    j = z + 1;
    i += j;
}
```

Würde man sich auf nur einen einzigen Durchlauf verlassen, so würde man anhand der letzten Zuweisung des Listings neben der Variablen `i` die Variable `j` in die Menge der Schleifenparameter aufnehmen. Da aber Variable `j` von Variable `z` beeinflusst wird, muss `z` ebenfalls zur Menge der indirekten Schleifenparameter gezählt werden. Dies könnte man aber in nur einem Durchlauf nicht ohne vorherigem Aufbau komplexer Abhängigkeitsketten der einzelnen Variablen erkennen, so dass an dieser Stelle eine einfache Implementierung gewählt wurde.

#### Manipulation innerhalb verschachtelter Schleifen

Wie bereits erwähnt, werden die Knoten verschachtelter Schleifen bei der Generierung der Ausführungspfade ausgelassen, so dass sie auch bei der Manipulationsbetrachtung der Schleifenparameter vorerst außer Acht gelassen werden. Wird ein Schleifenparameter einer äußeren Schleifen innerhalb einer inneren Schleife manipuliert, so trägt diese Manipulation entscheidend zu Gesamtiteration der äußeren Schleife bei. Da man allerdings bei einer statischen Analyse im Allgemeinen keine *genaue* Aussage darüber treffen kann, wie häufig die innere Schleife durchlaufen wird und daher auch nicht exakt genug beschreiben kann, wie sich die Manipulationen im Inneren auf die Gesamtiteration der äußeren Schleife auswirken, wird in einem solchen Fall die Analyse der Schleife mit entsprechender Fehlermeldung abgebrochen. Das Analyseprogramm erwartet in derartigen Fällen Unterstützung vom Anwender in Form geeigneter Annotationen (vgl. Kapitel 4.4).

Dies bedeutet allerdings nicht, dass das Analyseprogramm verschachtelte Schleifen vollkommen außer Acht lässt! Sofern die Schleifenparameter der äußeren Schleifen von den Anweisungen der inneren Schleife *nicht* manipuliert werden, können beide Schleifen problemlos untersucht werden.

### 4.3.5 Wertebereich der Schleifenparameter

Der nächste Schritt bei der Bestimmung der maximalen Schleifeniteration beschäftigt sich mit der Frage nach den Wertebereichen der Schleifenparameter – genauer mit Berechnung der Anfangswerte der Schleifenparameter.

Hierfür betrachtet man zunächst alle verfügbaren Definitionen (vgl. 3.3.1) jedes einzelnen Schleifenparameters am Schleifenkopf und analysiert folgende Gesichtspunkte:

1. *Ist die verfügbare Definition zum Analysezeitpunkt auswertbar im Sinne von Kapitel 4.3.3?*
   Wird ein Schleifenparameter beispielsweise über den Rückgabewert eines Methodenaufrufs initialisiert, so kann der Rückgabewert der Methode im Allgemeinen nicht bestimmt werden, so dass die Analyse mit einer entsprechenden Fehlermeldung abbrechen muss.

2. *Enthält die Definition weitere Parameter, deren Anfangswerte bislang noch nicht ermittelt wurden?*
   Dies lässt sich am besten an einem Beispiel erläutern: Erreicht die Definition `x = a + 5;` den Kopf einer Schleife und ist `x` ein Schleifenparameter, so muss man zunächst alle gültigen Werte von `a` ermitteln, um alle möglichen Anfangswerte von `x` zu bestimmen. Dies geschieht über rekursive Berechnungen der verfügbaren Definitionen aller Variablen, die Teil der verfügbaren Definitionen der entsprechenden Schleifenparameter sind.

3. *Ist eine verfügbare Definition Teil einer Schleife?*
   Wird ein Schleifenparameter innerhalb einer vorherigen Schleife initialisiert, so kann man nicht definitiv sagen, mit welchem Wert dieser Parameter diese Schleife verlassen wird. Eine dahingehende Analyse hätte den Rahmen dieser Arbeit überspannt, so dass in solchen Fällen die Analyse mit entsprechender Fehlermeldung abbricht. Es wäre aber an dieser Stelle denkbar, zunächst die vorhergeschaltete Schleife zu analysieren und, mit dem Wissen über die Manipulation der in dieser Schleife veränderten Variablen, die Anfangswerte für alle nachfolgenden Schleifen zu errechnen.

Wurden die soeben gestellten Fragen geeignet beantwortet, lassen sich die Anfangswerte durch entsprechende Kombination der Variablen errechnen. Betrachten wir hierfür folgendes Beispiel:

```
par0 = if (E1) ? 0 : 1;
par1 = if (E2) ? 2 : 3;

i = par0 + par1;
while (i < 10) {
    [...]
}
```

Der Anfangswert des Schleifenparameters `i` hängt von den beiden Variablen `par0` und `par1` ab. Der Wertebereich von `par0` ist {1, 2} und von `par1` {2, 3}. Durch entsprechende Kombination der Wertebereiche beider Variablen, ergibt die Berechnung der möglichen Anfangswerte von `i`: {2, 3, 4}.

Je komplexer und umfangreicher der arithmetische Ausdruck der verfügbaren Definition eines Schleifenparameters ist, desto größer wird in der Regel auch der Wertebereich der Anfangswerte dieses Parameters.

Zum Abschluss dieser Betrachtung sei noch angemerkt, dass auch Annotationen (vgl. Kapitel 4.4) in die Berechnung der Anfangswerte einfließen. Der Wertebereich von Methodenparametern kann beispielsweise annotiert werden, sollte ein Methodenparameter den maximalen Durchlauf der Schleife beeinflussen.

### 4.3.6 Simulation mithilfe interpretativer Ausführung

Zum besseren Überblick sei kurz der aktuelle Stand der Schleifenanalyse rekapituliert: Wir kennen den *Typ* der Schleife und von welchen Parametern die *Schleifenabbruchbedingung* abhängt. Des Weiteren kennen wir alle möglichen *Ausführungspfade* innerhalb des Schleifenrumpfes und alle Anweisungen, die die Schleifenparameter *manipulieren*. Zu guter Letzt kennen wir alle möglichen *Anfangswerte* der Schleifenparameter. Was nun immer noch fehlt, ist das Wissen über die maximale Anzahl der möglichen Schleifendurchläufe!

Ein möglicher Ansatz wäre die Rekonstruktion der Schleifenabbruchbedingung. Wenn es gelingen würde, einzig anhand der Beschaffenheit des Kontrollflussgraphen, den kompletten booleschen Ausdruck mit allen logischen Verknüpfungen der Schleifenabbruchbedingung zu rekonstruieren, so könnte man, durch Zuhilfenahme des Wissens über die Anfangswerte und die Manipulation der Schleifenparameter, den maximalen Durchlauf berechnen. Dies ist allerdings *nicht* möglich, da das Vorhaben an der Rekonstruktion eines allgemeinen booleschen Ausdrucks scheitert. Auf die Frage, ob man boolesche Ausdrücke, beispielsweise mit Hilfe struktureller Analysetechniken, rekonstruieren kann, schreibt [MC04]: *„[...] complex boolean expressions are reducible regions that cannot be mapped to any known predefined pattern [...]“*. Wie wir bereits aus vorhergehenden Betrachtungen wissen, lassen sie sich zwar erkennen, jedoch nicht wieder in ihre Originalform rekonstruieren.

Der zweite Ansatz schlägt eine etwas andere Richtung ein. Hier versucht man mithilfe *interpretativer Ausführung* eine Schleife zu simulieren und so die maximale Iterationsanzahl zu bestimmen. Dabei werden die Anweisungen innerhalb der Schleife bereits zur Analysezeit durch eine stark vereinfachte virtuelle Maschine interpretiert und auf diese Weise Informationen über die Schleifeniterationen gesammelt. Man beachte, dass hierbei nur diejeni-

gen Anweisungen interpretiert werden müssen, welche die Anzahl der Schleifendurchläufe beeinflussen. Konkret genügt die Interpretation sämtlicher Instruktionen der Schleifenabbruchbedingungsknoten und alle Anweisung des Schleifenrumpfes, welche die Variablen der Abbruchbedingung manipulieren.

Die Interpretation des Schleifencodes beginnt stets mit dem Schleifenkopf, d.h. je nach Schleifentyp mit der Ausführung des ersten Schleifenabbruchknotens im Falle von while-Schleifen oder mit der Ausführung des ersten Rumpfknotens im Falle von do-while-Schleifen. Im Folgenden soll die Simulation einer while-Schleife beschrieben werden. Man kann sich leicht davon überzeugen, dass sich die Simulation einer do-while-Schleife nur geringfügig davon unterscheidet und daher hier nicht näher beleuchtet werden muss.

Die Simulation wurde mithilfe des *Besucher-Entwurfsmusters* [EG97, S.96] realisiert. Jede Bytecodeinstruktion ist dabei in einem `Instruction`-Objekt gekapselt und bietet eine Methode `visit(Visitor v)`. Der Simulator, eine abgeleitete Klasse von `Visitor`, implementiert seinerseits für jede der 212 Bytecodeinstruktionen eine geeignete Methode, so dass je nach Instruktionsobjekt am Simulator eine passende Methoden aufgerufen werden kann.

Der Simulator benötigt für die Interpretation zwei wichtige Datenstrukturen:

1. **Stackframe**
   Der Stackframe bietet zum Einen Speicherplatz für alle lokalen Variablen, zum Anderen besitzt er einen Operandenstack, auf dem die Operanden der einzelnen Bytecodebefehle abgelegt werden können.

2. **Ausführungsstack**
   Die wichtigste Aufgabe des Ausführungsstacks ist die Zwischenspeicherung von bedingten Sprunganweisungen. Jedes mal, wenn der Simulator auf eine bedingte Sprunganweisung stößt, erzeugt er ein `Condition`-Objekt und legt es auf den Ausführungsstack. Sobald ein neues Objekt auf dem Ausführungsstack abgelegt worden ist, wird ein registriertes Kontrollobjekt benachrichtigt, das anhand dieser Anweisung *und* den aktuellen Variablenzuständen entscheidet, welche Anweisung als Nächste ausgeführt werden muss.

Wie bereits erwähnt, beginnt die Simulation einer while-Schleife mit der Auswertung der Anweisungen des ersten Abbruchbedingungsknotens. Die einzelnen Bytecodeinstruktionen werden dabei vom Simulator zu immer komplexeren Ausdrücken aus Tabelle 4.1 zusammengefasst und auf dem Operandenstack abgelegt. Die letzte Anweisung eines Bedingungsknoten ist stets ein bedingter Sprung. Entsprechend dieser Sprunganweisung wird ein `Condition`-Objekt erzeugt und auf dem Ausführungsstack abgelegt. Die Auswertung

des bedingten Sprunges erfolgt rekursiv: Sowohl das `Condition`-Objekt, als auch seine Operanden implementieren die Schnittstelle `Computable`. Sie dient nicht nur, wie in Kapitel 4.3.3 beschrieben, zur Auswertbarkeitsüberprüfung, sondern besitzt zudem die Methode `compute(...)`, welche die Auswertung eines Ausdrucks ermöglicht. Je nach Resultat des ersten `Condition`-Objekts und der Beschaffenheit des Kontrollflussgraphen, muss die Simulation der Schleife mit

- der Auswertung des nächsten Bedingungsknotens,
- der Auswertung der ersten Anweisung eines möglichen Ausführungspfads oder
- dem Abbruch der Ausführung

fortgesetzt werden. Führt der Pfad zum nächsten Bedingungsknoten, wird dessen Bytecodesequenz interpretiert und am Ende wiederum ein `Condition`-Objekt ausgewertet. Führt der Pfad hingegen in den Schleifenrumpf, wird zunächst ein Zähler inkrementiert und anschließend der erste mögliche Ausführungspfad der Schleife simuliert, d.h. es werden alle Anweisungen ausgewertet, welche die Variablen der Schleifenabbruchbedingung manipulieren. Nachdem alle Manipulationsanweisungen des ersten Pfades ausgewertet worden sind, startet die Simulation erneut mit der Auswertung des Schleifenkopfes. Solange die Abbruchbedingung nicht erreicht worden ist, wird ein Pfad des Schleifenrumpfes ausgewertet und der Iterationszähler erhöht. Sobald ein Pfad die Schleife verlässt, wird der aktuelle Zählerstand zwischengespeichert und die Simulation mit der Betrachtung eines weiteren Ausführungspfads der Schleife fortgesetzt. Zwei Dinge sind noch anzumerken:

1. Ausführungspfade können neben Manipulationsanweisungen auch bedingte Sprünge enthalten, da eine Schleife auch über eine `break`-Anweisung verlassen werden kann. Ist dies der Fall, wird der Iterationszähler ebenfalls zwischengespeichert und die Simulation mit einem weiteren Pfad fortgesetzt.

2. Wie schon in Kapitel 4.3.5 beschrieben, können die Parameter einer Schleife zu Beginn mit unterschiedlichen Werten initialisiert sein. Da sich diese Werte unterschiedlich auf die Iterationsanzahl der Schleife auswirken können, muss jeder mögliche Ausführungspfad mit allen zulässigen Anfangswerten der Schleifenparameter analysiert werden.

Sobald sämtliche Kombinationen aus den möglichen Anfangswerten der Schleifenparameter und allen möglichen Ausführungspfaden ausgewertet worden sind, ist eine Aussage über die maximale Iterationsanzahl möglich – sie entspricht dem bislang höchsten zwischengespeichertem Wert des Iterationszählers.

**Anmerkungen**

Diese Art der Iterationsbestimmung funktioniert sehr gut, solange eine Schleife nicht wie im folgenden Beispiel aufgebaut ist:

```
while (E1) {
    if (E2) {
        [modify loop parameter to approach
        the break condition]
    } else {
        [modify loop parameter to depart
        the break condition]
    }
}
```

Ein Pfad der Schleife modifiziert die Schleifenparameter so, dass sie sich der Abbruchbedingung nähern, ein andere modifiziert sich nicht oder nur so, dass sie sich von der Abbruchbedingung entfernen. Sobald der zweite Pfad der Schleife analysiert wird, findet die Analyse in ihren Augen eine Endlosschleife und bricht die Analyse mit entsprechender Fehlermeldung ab, obwohl die Schleife nach oben durchaus beschränkt sein kann.

Wie bereits mehrmals angedeutet, lassen sich zum Analysezeitpunkt nicht alle für die Analyse benötigten Informationen aus dem Programmcode gewinnen. An dieser Stelle muss der Anwender die benötigten Informationen anderweitig bereit stellen. Das anschließende Kapitel beschäftigt sich mit der Problematik, wie der Anwender diese Informationen im Konkreten bereitstellen muss, damit sie in die Analyse miteinfließen können.

## 4.4 Annotationen

Das Analyseprogramm bietet eine einfache und compilerunabhängige Möglichkeit, wie ein Anwender sein Programm auf Quellcodeebene mit Informationen annotieren kann. Annotationen müssen stets dann bereitgestellt werden, wenn die zur Berechnung der WCET benötigten Informationen nicht aus dem Programmcode zur Analysezeit extrahiert werden können. Hängt beispielweise die obere Grenze der Schleifeniterationen von einem ganzzahligen

Laufzeitparameter ab, so kann dieser Parameter zur Analysezeit nur sehr unpräzise nach oben abgeschätzt werden. Die einzige Aussage, die man zur Analysezeit über den möglichen Wertebereich dieses Parameters treffen kann, ist aufgrund der physikalischen Beschränktheit dieses Parameters, dass er sich irgendwo zwischen $-2^{31}$ und $2^{31}-1$ aufhalten muss – diese Aussage ist aber aufgrund der in Kapitel 2.1 geführten Diskussion völlig wertlos.

Da Entwickler echtzeitfähiger Anwendungen meist sehr gut über das Laufzeitverhalten ihrer Programme informiert sind, ist es für sie meist nicht allzuschwer gerade solche Wertebereiche einzuschränken. Um solche Informationen einem Analysewerkzeug bereitstellen zu können, gibt es verschiedene Ansätze.

Ein möglicher Ansatz wäre, dass der Entwickler seine Annotationen in einer separaten Datei bereitstellt und dem Analysewerkzeug diese Datei neben dem Programmcode vorlegt. Dieser Ansatz ist allerdings sehr fehlerbehaftet, da die Entwickler bei jeder Änderung an ihren Quelldateien überprüfen müssten, ob die bereitgestellten Annotationen immer noch gültige Positionen im Quellcode referenzieren. Aus diesem Grund ist ein Annotationsmechanismus vorzuziehen, bei dem der Entwickler seine Informationen direkt im Quellcode verankern kann. Auch hier gibt es verschiedene Varianten:

- Die Annotationen könnten als Java-Kommentare angegeben werden. Dies hat jedoch den Nachteil, dass Kommentare im Bytecode nicht mehr sichtbar sind. Das heißt der Entwickler müsste dem Analysewerkzeug nun nicht mehr allein den Bytecode, sondern auch den Quellcode vorlegen.

- Man erweitert den Java-Sprachumfang durch zusätzliche Schlüsselwörter, mit deren Hilfe der Programmcode annotiert werden kann. Dies bedeutet aber gleichzeitig, dass man auf einen Spezialcompiler angewiesen ist, der diese Schlüsselwörter kennt und in der Lage ist, diese Annotationen für das Analysewerkzeug in wiederauffindbare Bytecodesequenzen zu übersetzen. Dieser Ansatz ist aufgrund der Verankerung mit einem Compiler nicht praktikabel.

- [GB00] stellen einen Lösungsansatz vor, mit dem man Annotationen direkt im Quellcode anbringen kann, die compilerunabhängig sind und dennoch von allen Compilern in wiederauffindbare Bytecodesequenzen übersetzt werden. Die Lösung besteht darin, eine Annotationsklasse mit *leeren* statischen Methodenaufrufen bereitzustellen. Möchte ein Entwickler seinen Code annotieren, so ruft er eine entsprechende Methode an dieser Annotationsklasse auf. Der Compiler erzeugt daraufhin eine

Bytecodesequenz für diesen Methodenaufruf, in der zunächst die Parameter für diese Methode – beispielsweise Annotationen bezüglich des Wertebereichs einer Variablen – auf den Operandenstack gelegt werden und anschließende ein `INVOKESTATIC`-Befehl den Aufruf auslöst. Man beachte, dass diese Bytecodesequenzen *vor* der eigentlichen Ausführung des Programms aus den Classfiles wieder entfernt werden können, da sie keine funktionalen Aspekte des Programmcodes enthalten. Sie belasten demnach keinesfalls das Laufzeitverhalten der Anwendung!

Das in dieser Arbeit entworfene Werkzeug, stellt eine Annotationsklasse bereit, die der von [GB00] vorgestellten `WCETAn`-Klasse ähnelt. Die Klasse heißt `WCETAnnotation` und all ihre Einträge können Abbildung 4.7 entnommen werden.

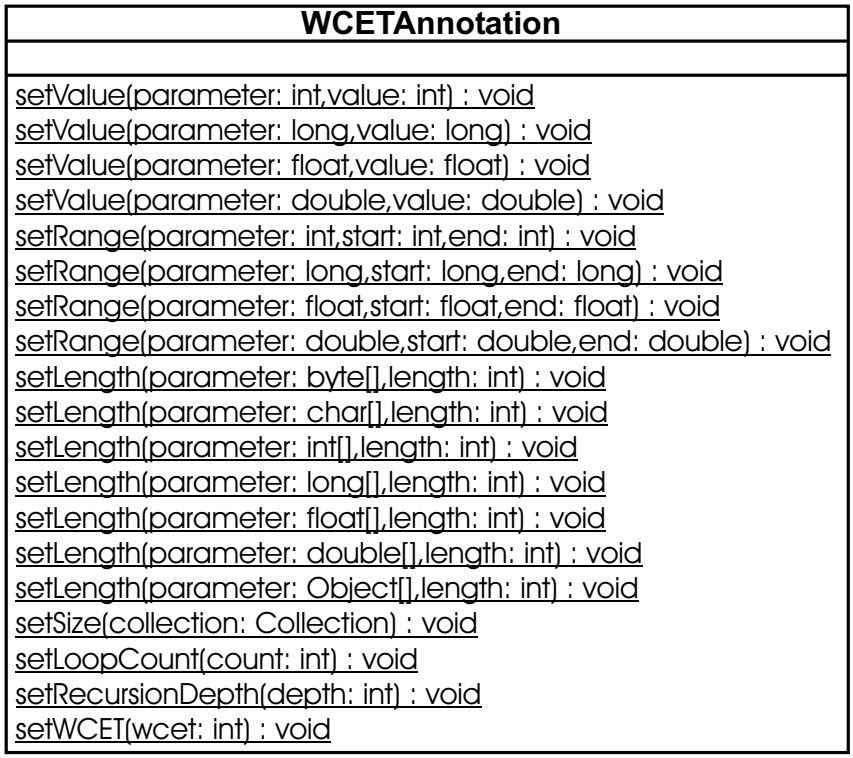



Abbildung 4.7: UML-Diagramm der Klasse `WCETAnnotation`

Die Klasse bietet statische Methoden um Variablen, Arrays, Collections, Schleifen, Rekursionstiefen oder direkt die WCET einer Methode zu annotieren. Die meisten Annotationsmethoden sind so deklariert, dass der Programmierer die zu annotierende Variable und den entsprechenden Maximalwert bzw. Wertebereich übergeben muss. Tabelle 4.2 kommentiert die Bedeutung

der einzelnen Methode. Beispiele für die konkrete Verwendung dieser Klasse befinden sich in Kapitel 5.1.

| Methode | Bedeutung |
|---|---|
| *setValue* | Annotiert den Maximalwert einer Variablen. |
| *setRange* | Annotiert den Wertebereich einer Variablen |
| *setLength* | Annotiert die Maximallänge eines Arrays. |
| *setSize* | Annotiert die Maximalgrösse einer Collection. |
| *setLoopCount* | Annotiert die maximale Schleifeniterationsanzahl. |
| *setRecursionDepth* | Annotiert die maximale Rekursionstiefe. |
| *setWCET* | Annotiert die WCET einer Methode. |

Tabelle 4.2: Statische Methodenaufrufe von `WCETAnnotation`

### 4.4.1 AnnotationScanner und Orakel

Aufgefunden werden die statischen Methodenaufrufe an `WCETAnnotation` wie schon beim Simulator mithilfe eines Besuchers – dem `AnnotationScanner`. Sobald er auf einen statischen Methodenaufruf stößt, wird überprüft, ob es sich um eine der Annotationsmethoden handelt. Falls ja, werden die Aufrufparameter dieser Methode vom Operandenstack abgetragen und intern in einem `Annotation`-Objekt gekapselt. Die Suche nach den Annotationen eignet sich bereits vor der eigentlichen WCET-Analyse, so dass die annotierten Informationen in alle anschließenden Analysen direkt einfließen können.

Jeder zu analysierenden Methode ist ein *Orakel* zugewiesen, das die annotierten Informationen verwaltet. Da jedes Orakel bereits vor der eigentlichen Analyse vom `AnnotationScanner` über alle Annotationen informiert worden ist, kann es an beliebigen Stellen der Analyse befragt werden. Erst wenn das Orakel auch keine Antwort kennt, wird ein Analyseschritt mit entsprechender Fehlermeldung abgebrochen.

### 4.4.2 Fehlerhafte Annotationen

Da eine annotierte Information kaum validiert werden kann, wird nur überprüft, ob der annotierte Wert eine Konstante ist. Ansonsten werden Annotation zunächst als korrekt angenommen und nicht weiter hinterfragt. Dies wirft jedoch auch Risiken auf, falls ein Entwickler falsche Informationen annotiert, oder die Annotationen nach einer Programmüberarbeitung nicht aktualisiert. In Kapitel 5.3.2 wird diese Problematik genauer diskutiert, so dass an dieser Stelle nicht weiter darauf eingegangen werden soll.

## 4.5 Methodenaufrufe

Der folgende Abschnitt abstrahiert von der intraprozeduralen Analyse anhand von Kontrollflussgraphen und widmet sich einer weiteren Herausforderung der WCET-Bestimmung, nämlich der Betrachtung von interprozeduralen *Methodenaufrufen*. Methodenaufrufe werden im Java-Bytecode durch vier verschiedene *Invokeinstruktionen* ausgelöst:

1. `INVOKESTATIC`
   wird verwendet um einen statischen Methodenaufruf auszulösen.
2. `INVOKESPECIAL`
   findet seinen Einsatz in genau drei Fällen:
   (a) um die Instantiierungsmethode `<init>` aufzurufen,
   (b) um eine private Methode eines Objektes aufzurufen oder
   (c) um eine Methode einer Superklasse aufzurufen.
3. `INVOKEINTERFACE`
   ruft eine Methode an einem Java-Interface auf.
4. `INVOKEVIRTUAL`
   wird schließlich in allen restlichen Fällen eingesetzt, sprich um einen virtuelle Methodenaufruf an einem beliebigen Objekt durchzuführen.

Man beachte, dass in den beiden letzten Fällen zur Übersetzungszeit nicht klar definiert ist, welche konkrete Methode zur Laufzeit ausgeführt wird! Man kann in den meisten Fällen keine Aussage darüber treffen, welches konkrete Objekt sich hinter einer Schnittstelle verbirgt oder ob sich hinter einer Basisklassenvariablen nicht doch ein abgeleitetes Objekt verbirgt. Und genau dieses fehlende Wissen erschwert die Bestimmung der WCET. Man betrachte folgendes Beispiel:

```
void foo(boolean b) {
    X x = (b) ? new A() : new B();
    String s = x.toString();
    [...]
}
```

Klasse $A$ und Klasse $B$ sind abgeleitet von der abstrakten Klasse $X$ und je nach Wert des Methodenparameters `b`, wird der Variablen `x` eine Instanz von $A$ oder $B$ zugewiesen. Da man zur Übersetzungszeit meist keinerlei Wissen über `b` hat, lässt sich vorab nicht entscheiden, ob sich hinter der Variablen `x`

ein Objekt der Klasse $A$ oder $B$ verbergen wird und ob somit die Methode `A.toString()` oder `B.toString()` ausgeführt werden wird.

Soll die WCET genau dieses Listings bestimmt werden, muss man sich dennoch Gedanken machen, welche der beiden Methode man in die Analyse miteinfließen lassen möchte. Man löst dieses Problem, indem man beide Methoden analysiert und die Ausführungsdauer der Langsameren zur Gesamtausführungszeit addiert.

Zuvor muss aber noch eine wichtige Frage geklärt werden: Welche Methoden können sich überhaupt hinter einem virtuellen Methodenaufruf verbergen? Hierfür unterscheidet man zunächst zwei Begriffe:

- **call site**
  Unter der *call site* eines Methodenaufrufs versteht man im Wesentlichen seine Signatur. Die einzelnen Methoden, die sich hinter einer call site verbergen können, nennt man

- **call instance**
  Die *call instances* eines Methodenaufrufs sind demnach die konkreten Bytecodesequenzen, die sich hinter einem virtuellen Methodenaufruf verbergen können.

Bevor das Zusammenspiel zwischen *call site* und *call instance* genauer untersucht werden kann, müssen zunächst die Abhängigkeiten auf Klassenebene geklärt werden. Hierzu behilft man sich der Klassenhierarchieanalyse.

### 4.5.1 Klassenhierarchieanalyse

Die *Klassenhierarchieanalyse* klärt die Ableitungsverhältnisse aller Klassen untereinander. Jede Java-Klasse hat genau eine Basisklasse und kann beliebig viele Schnittstellen implementieren. Der Name der Basisklasse und die Namen aller implementierter Schnittstellen stehen im Konstantenpool des Classfiles einer Klasse (vgl. Abbildung 2.1). Um daher die Klassenhierarchie, welche die Ableitungsverhältnisse aller Klassen darstellt, zu konstruieren, genügt ein einmaliger Durchlauf über alle Klassen. Dabei betrachtet man jede Klasse einzeln und annotiert an ihrer Basisklasse den Namen ihrer abgeleiteten Klassen – analog an jeder Schnittstelle die Namen aller abgeleiteten Schnittstellen und die Namen aller Klassen, die diese Schnittstelle implementieren. So ist man im Stande einen Baum zu erzeugen, dessen Knoten die Klassen und Schnittstellen darstellen und dessen Kanten die Ableitungsverhältnisse symbolisieren. Abbildung 4.8 zeigt eine Klassenhierarchie für das oben angedeutete Beispiel.

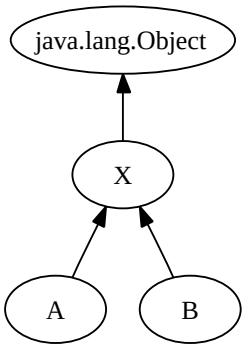



Abbildung 4.8: Klassenhierarchie

Die wichtigste Aufgabe der Klassenhierarchie ist die Klärung der Frage nach den Laufzeittypen einer Programmvariablen. Dabei bedarf es bei der Analyse einer als Klasse deklarierten Variablen einer anderen Betrachtung als bei der einer als Schnittstelle deklarierten (vgl. [VS00]):

- Die Menge der Laufzeittypen einer als Klasse $C$ deklarierten Variablen, umfasst die Klasse $C$ selbst und alle abgeleiteten Klassen von $C$.
- Die Menge der Laufzeittypen einer als Schnittstelle $I$ deklarierten Variablen, umfasst hingegen:
  1. Die Menge aller Klassen, die $I$ oder eine abgeleitete Schnittstelle von $I$ implementieren, welche von nun als *implements(I)* bezeichnet werden soll und
  2. die Menge alle abgeleiteten Klassen von *implements(I)*.

Man kann sich leicht davon überzeugen, dass sich die Frage nach allen möglichen Laufzeittypen einer beliebigen Variablen leicht mithilfe einer Traversierung über den Klassenhierarchiebaum implementieren lässt. Die Klassenhierarchieanalyse ist Voraussetzung für die Konstruktion einer weiteren, sehr wichtigen Datenstruktur: dem *Aufrufgraph*.

### 4.5.2 Aufrufgraph

Der *Aufrufgraph* beschreibt die Abhängigkeiten zwischen Methoden, die sich aufgrund der Aufrufanweisungen im Bytecodestrom einer Anwendung ergeben. Er zeigt zum Einen die sich ergebenden Aufrufhierarchien, zum Anderen visualisiert er sehr deutlich das Zusammenspiel zwischen *call site* und *call instance*. Abbildung 4.9 zeigt den Aufrufgraph der Methode `foo(...)` aus dem obigen Listing.

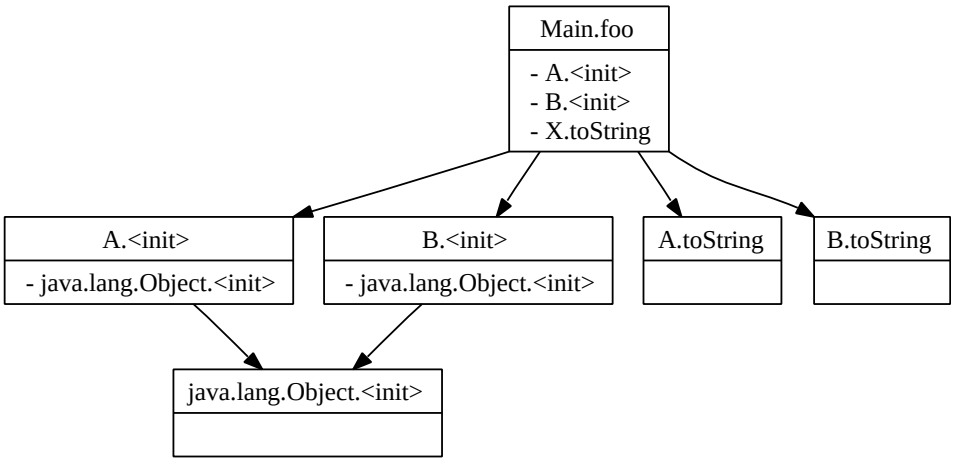



Abbildung 4.9: Aufrufgraph

Man erkennt, dass die Methode `foo(...)` potentiell drei Methodenaufrufe durchführt: zwei Konstruktoraufrufe an den Klassen $A$ und $B$ und einen virtuellen Methodenaufruf an Klasse $X$ – `X.toString()`. Des Weiteren ziehen die beiden Konstruktoraufrufe die Initialisierung ihrer Basisklasse nach sich. Anhand der vier ausgehenden Kanten der Wurzelknotens erkennt man, dass sich hinter der *call site* `X.toString()` zwei verschiedene *call instances* verbergen: `A.toString()` und `B.toString()`.

Konstruiert werden Aufrufgraphen mithilfe einer Klassenhierarchieanalyse. Man beginnt die Konstruktion des Aufrufgraphen eines Programms an einer *Ankermethode* – beispielsweise der *main*-Methode – für die man den Wurzelknoten des Aufrufgraphen erzeugt. Den Bytecode dieser Methode durchsucht man nach Invokeinstruktionen und erkennt auf diese Weise alle *call sites* dieses Knotens. Da sich hinter *statischen* oder *speziellen* Methodenaufrufen immer nur eine *call instance* verbirgt, erzeugt man für diesen Methodenaufruf genau einen Knoten und verbindet ihn mit dem Wurzelknoten. Man beachte, dass sich die *call instance* in diesem Fall nicht unbedingt an der Klasse $C$ befinden muss, welche die *call site* referenziert – die *call instance* kann sich auch an einer der Basisklassen befinden! Man findet sie, indem man die Klassenhierarchie ab der Klasse $C$ in postorder traversiert und nach einer der *call site* adäquaten Methodensignatur sucht. Die erste Methode, die man auf diesem Wege findet, ist die gesuchte *call instance*.

Trifft man auf einen *virtuellen* Methodenaufruf, muss man zudem beachten, dass sich hinter der *call site* potentiell auch die *call instances* aller abgeleiteten Klassen verbergen können. Daher geht man zunächst wie bei statischen Methodenaufrufen vor, nur dass man anschließend auch die

Laufzeittypen von $C$ betrachtet. Diese erhält man, wie in Kapitel 4.5.1 beschrieben, durch eine Analyse der Klassenhierarchie. Schließlich durchsucht man diese Typen nach der Signatur der *call site*, erzeugt unter Umständen entsprechende Aufrufknoten und verbindet sie mit dem Wurzelknoten.

*Schnittstellenaufrufe* werden ähnlich den virtuellen Methodenaufrufen behandelt. Die Klassenhierarchie liefert alle Klassen, die eine Schnittstelle implementieren. Diese werden nach einer entsprechenden Methodensignatur durchsucht. Für jede gefundene Signatur erzeugt man einen Aufrufknoten und verbindet ihn mit dem Wurzelknoten.

Um einen vollständigen Aufrufgraphen zu erzeugen, speichert man jeden erzeugten Knoten zusätzlich in einer Arbeitsliste und führt das soeben beschriebene Verfahren an jedem Knoten der Arbeitsliste durch. Besitzt der Aufrufgraph hingegen bereits den Knoten einer entsprechenden *call instance*, so wird nur eine entsprechende Kante im Aufrufgraph eingetragen, der Knoten aber nicht nochmals in die Arbeitsliste eingefügt. Die Konstruktion endet mit der Bearbeitung des letzten Knotens der Arbeitsliste.

### Größe eines Aufrufgraphen

Man kann sich vorstellen, dass der Graph umso größer wird, je mehr Methodenaufrufe ein Programm besitzt, aber auch je tiefer und breiter die Klassenhierarchie des zugrunde liegenden Programms ist. Da sich hinter einem Aufruf, der in seiner Signatur die Wurzel der Klassenhierarchie trägt, potentiell die Methodenimplementierungen *aller* Klassen verbergen können, welche diese Methode überschreiben, wächst der Aufrufgraph sehr schnell in die Breite. Liegt der Aufrufanalyse eine Klassenhierarchie mit 100 verschiedenen Klassen zugrunde, die alle die Methode `toString()` überschreiben, so entstehen im Aufrufgraph durch den Aufruf `Object.toString()` 100 Kanten zu allen *call instances* der Klassenhierarchie! Falls im Programm nicht wirklich alle 100 Klassen instantiiert worden sind, ist ein solcher Aufrufgraph viel breiter und größer als wirklich nötig. Folglich werden bei der anschließenden WCET-Berechnung Methoden von nicht instantiiert Klassen untersucht, so dass die berechnete WCET dieses Programms u.U. schlechter wird als sie tatsächlich ist, da man sich im Zweifel immer für die teuerste *call instance* entscheiden muss.

Im folgenden Abschnitt wird ein Verfahren beschrieben, welches zum Ziel hat, die Aufrufgraphen zu verschlanken, und so dabei hilft, die zu errechnende WCET exakter zu ermitteln.

### 4.5.3 Rapid-Type-Analyse

Die *Rapid-Type-Analyse* ist ein sehr einfaches Verfahren, um die Typen möglicher *call instances* einzugrenzen. Die Idee, die dem Verfahren zu Grunde liegt, beruht auf der Beobachtung, dass ein Aufruf ausschließlich an einem Objekt durchgeführt werden kann, das zum Zeitpunkt des Aufrufs bereits existiert. Das bedeutet beispielsweise für einen virtuellen Methodenaufruf an der Klasse $C$, dass man neben den Laufzeittypen von $C$, *runtimeTypes(C)*, auch alle bislang vom Programmcode instantiierten Typen, *instantiatedTypes(C)*, betrachten sollte. Die *Rapid-Types* der Klasse $C$, *rapidTypes(C)*, ergeben sich somit durch Gleichung 4.1.

$$rapidTypes(C) = runtimeTypes(C) \cap instantiatedTypes(C) \tag{4.1}$$

#### Implementierung

[Bac85] schlägt eine mögliche Implementierung der Rapid-Type-Analyse vor. Man startet die Analyse an einer Ankermethode des Programms, markiert diese Methode als künftigen Knoten des Aufrufgraphen und untersucht schließlich alle in ihr erzeugten Objekte und fügt deren Typen in die Menge der *lebendigen* Klassen $C_L$ hinzu. Danach untersucht man alle *call sites* dieser Methode. Direkte Methodenaufrufe, wie beispielsweise statische Methodenaufrufe, werden ebenfalls als künftige Aufrufknoten markiert, virtuelle Methodenaufrufe bedürfen einer gesonderten Betrachtung. Man ermittelt mithilfe der Klassenhierarchie alle in Frage kommenden *call instances* und vergleicht deren Objekttypen mit den bislang instantiierten Typen aus der Menge $C_L$. Die *call instances*, welche einen bereits lebendigen Objekttyp haben, werden ebenfalls als künftige Aufrufknoten markiert. Die Übrigen speichert man zunächst in einer gesonderten Menge $Q_V$. Alle bis dahin markierten, aber noch nicht weiter untersuchten Methoden, werden anschließend nach gleichem Muster rekursiv untersucht.

Jedes Mal, wenn ein neuer Typ in die Menge $C_L$ der lebendigen Klassen aufgenommen wird, wird die Menge $Q_V$, der virtuellen, noch nicht aufgenommenen Methodenaufrufe analysiert. Finden sich darin Methodenaufrufe, deren Objekttyp sich nun unter den lebendigen Klassen befindet, wird diese Methode aus der Menge $Q_V$ entfernt, als Aufrufknoten markiert und ebenfalls, wie soeben beschrieben, rekursiv analysiert.

Diese Form der Implementierung ist sehr konservativ und daher *pessimistisch*, da man *call instances* in den Aufrufgraph nachträglich aufnimmt, auch wenn ihr Objekttyp erst tiefer im Aufrufgraph instantiiert wird. Diese Form der Implementierung ist aber notwendig, wenn man Schleifen innerhalb des

Aufrufgraphen, die sich beispielweise durch rekursive Methodenaufrufe ergeben, zulässt. Dadurch können Objektinstanzen im Aufrufgraph *„von unten nach oben“* wandern.

*Optimistische* Implementierungen gehen diesem nachträglichen Eintragen nicht nach und erzeugen daher noch kleinere Aufrufgraphen. Sie könnten aber u.U. zu einer falschen, da zu geringen, WCET führen und sind daher auf diesem Gebiet nicht geeignet.

#### Anmerkungen

Wie man beispielsweise den Benchmarkergebnissen von [VS00] entnehmen kann, liefert die Rapid-Type-Analyse zwar deutlich kleinere Aufrufgraphen als die in Kapitel 4.5.2 beschriebenen (bis zu 50% weniger Knoten), aber bei weitem keine perfekten Ergebnisse. Ein Grund hierfür ist, dass der lokale Programmfluss völlig außer Acht gelassen wird. Dies wird an folgendem Beispiel ganz deutlich:

```
A x = new B();
x = new A();
result = x.foo();
```

Man erkennt sofort, dass sich hinter dem Aufruf `x.foo()` stets ein Objekt vom Typ $A$ und niemals vom Typ $B$ befindet. Da aber beide Klassen instantiiert werden, wird auch die Rapid-Type-Analyse einen Aufrufgraph mit zwei *call instances* erzeugen. Dennoch konnten mit diesem sehr einfachen Verfahren durchaus erkennbare Verbesserungen zum Zwecke einer exakteren WCET-Bestimmung erzielt werden.

### 4.5.4 Rekursion

Rekursive Methodenaufrufe lassen sich anhand von Zyklen innerhalb des Aufrufgraphen erkennen. Wie man sich leicht denken kann, spielt zum Zwecke der WCET-Bestimmung die maximale Rekursionstiefe eine entscheidende Rolle. Da diese nur mir sehr viel Aufwand und zudem nur in den seltensten Fällen anhand einer statischen Analyse errechnet werden kann, muss sie dem Analyseprogramm über Annotationen bekannt gemacht werden. Hierfür bietet die `WCETAnnotation`-Klasse eine spezielle Methode: `setRecursionDepth(int depth)` (vgl. Abbildung 4.7).

Innerhalb eines Zyklus im Aufrufgraphen spielen genau zwei Knoten eine wichtige Rolle:

1. Das *Rekursionsziel*
   bezeichnet den Knoten im Aufrufgraph, welcher den Zielknoten der Rekursionskante darstellt.

2. Die *Rekursionsquelle*
   bezeichnet hingegen den Knoten im Aufrufgraph, welche dem Ausgangsknoten der Rekursionskante entspricht.

Rekursionsziel und Rekursionsquelle können ein und derselbe Knoten sein, wichtig für das Analyseprogramm ist jedoch, dass die maximale Rekursiontiefe einer bestimmten Rekursion am Rekursionsziel annotiert wird! Der Grund hierfür wird schnell an den beiden Zyklen im Aufrufgraph von Abbildung 4.10 deutlich.

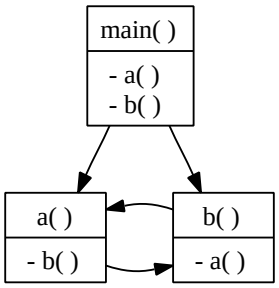



Abbildung 4.10: Aufrufgraph mit Zyklen

Methode `b()` löst im Kontext von Methode `a()` eine Rekursion aus, ist aber auch gleichzeitig Ziel einer Rekursionskante. Demnach ist sowohl `a()` als auch `b()` gleichzeitig Quelle und Ziel zweier verschiedener Rekursionen. Da beide Rekursionen unabhängig voneinander unterschiedliche Rekursiontiefen haben können, ist wichtig zu wissen, an *welcher Stelle* eine Annotation *wie* wirkt!

## 4.6 Abarbeitungszeit

Die Suche nach dem längsten Ausführungspfad eines Programms ist gleichzusetzen mit der Suche nach dem *teuersten* Pfad durch den Kontrollflussgraphen, der das Programm beschreibt, unter Berücksichtigung von Schleifeniterationen und Methodenaufrufen. Die Bestimmung maximaler Schleifeniterationen und die Behandlung von Methodenaufrufen wurde bereits in den vorangegangen Kapiteln ausführlich behandelt. Bislang wurde allerdings nicht angesprochen, was einen Pfad *teuer* macht bzw. wie man die Abarbeitungszeit eines beliebigen Pfades definiert.

Die Abarbeitungszeit eines Pfades durch ein Java-Programm definiert man als die Zeit, die eine JVM benötigt, um alle Bytecodeinstruktionen entlang dieses Pfades durchzuführen. Weil man bei der Suche nach dem teuersten Pfad von den einzelnen Instruktionen abstrahieren möchte, um direkt die Basisblöcke, die Knoten des Graphen, betrachten zu können, erscheint es geeigneter die Abarbeitungszeit als die Zeit zu definieren, die verstreicht, um die Basisblöcke entlang eines Pfades zu interpretieren. Setzt man Basisblock gleich Instruktionsbündel, so ist diese Definition durchaus korrekt. Der teuerste Ausführungspfad eines Programms ist demnach der Pfad, der die längste Zeit zur Ausführung benötigt.

Die Frage, die sich dabei als erstes stellt, ist: *Wie bestimmt man die Zeit eines Basisblocks?* Hierfür gibt es im wesentlichen zwei Ansätze:

1. Der triviale Ansatz geht davon aus, dass man die Zeiten für *jede* Bytecodeinstruktion kennt. Man addiert anschließend die Zeiten der einzelnen Instruktionen innerhalb eines Basisblocks und erhält so die Abarbeitungszeit für diesen Knoten. Doch dieses Vorhaben ist komplizierter als man denkt. Zum Einen gibt es eine Reihe von *parametrisierbaren* Instruktionen, die in Abhängigkeit aktueller Daten unterschiedliches Laufzeitverhalten zeigen. Ein typischer Vertreter hierfür ist der `LOOKUPSWITCH`, bei dem nicht klar ist, wie lang es dauert den korrekten Zweig zu finden, oder die Instruktionen `CHECKCAST` und `INSTANCEOF`, die in Abhängigkeit der Klassenhierarchie unterschiedlich lang dauern können. Zum Anderen kann fast jede Bytecodeinstruktion schneller oder langsamer ausgeführt werden, je nachdem in welchem Kontext sie steht. Die Caches und Pipelines der verschiedenen Prozessorarchitekturen tragen entscheidend zu der tatsächlichen Ausführungszeit eines Befehls an einer bestimmten Stelle im Instruktionsstrom bei. Soll dieser Ansatz dennoch bei der Bestimmung der Ausführungszeiten verfolgt werden, so misst man die Zeiten aller Bytecodeinstruktionen und setzt für alle parametrisierbaren Instruktionen möglichst pessimistische Annahmen. Man erhält auf diese Weise allerdings nur bedingt gute Messergebnisse. Ein jedoch nicht zu unterschätzender Vorteil liegt in der Berechnung der gesamten WCET. Da man die Zeiten der einzelnen Bytecodeinstruktionen für eine Reihe unterschiedlicher Architekturen bereits im Voraus bestimmen kann, lässt sich die WCET für die verschiedenen Systeme sehr schnell ermitteln, da nur das Hardwaremodell mit den Zeittabellen angepasst werden muss.

2. Der zweite Ansatz möchte genau dieser Problematik begegnen und schlägt daher einen etwas anderen Weg ein. Statt die Zeiten für jede

einzelne Instruktion auf einem bestimmten System zu bestimmen, versucht man gleich die Gesamtzeit eines Basisblocks zu messen. Hierfür zerstückelt man beim Übersetzungsvorgang den Bytecodestrom in seine Basisblöcke und misst die Dauer jedes einzelnen Basisblocks. Auch hierbei stößt man sehr schnell auf erste Schwierigkeiten. Beispielsweise müssen Methodenaufrufe auf *leere* Methodenrümpfe abgebildet werden, da man die Zeit der aufgerufenen Methode nicht von vorherein in der aktuellen Basisblockzeit haben möchte. Zudem muss man darauf achten, dass die zur Laufzeit erzeugten Objektreferenzen auf beliebige gültigen Objekte verweisen, so dass es zu keinen Ausnahmesituationen kommt. Doch der Vorteil dieses Verfahrens liegt auf der Hand: Da man nun jede Instruktion im Kontext ihres tatsächlichen Bytecodestroms betrachtet, fließen Cache- und Pipelineeffekte direkt in die Messergebnisse mitein, so dass exaktere Zeiten zu erwarten sind. Ein wesentlicher Nachteil jedoch ist, dass man diese Messung für jedes zu analysierende Programm und für jede Architektur stets wiederholen muss. Dieser Nachteil sollte aber im Hinblick auf eine genauere WCET-Bestimmung als eher gering eingestuft werden.

Der Vollständigkeithalber sollte noch erwähnt werden, dass [IB00a] einen Ansatz vorgestellt haben, wie man das erste Verfahren um die fehlenden Cache- und Pipelineeffekte erweitern kann. Sie besitzen für ihr Vorhaben zwei Tabellen – eine, welche die Zeiten der einzelnen Instruktionen enthält, eine Zweite für den sogenannten *Gain-Faktor* $\lambda$. $\lambda$ listet die Verbesserungen auf, welche sich dadurch ergeben, dass bestimmte Instruktionen in einer genauen Reihenfolge im Bytecodestrom auftreten. Da sich nicht alle Kombinationen bestimmen lassen, aber auch nicht alle Kombinationen Vorteile bringen, und sie außerdem festgestellt haben, dass bereits die Betrachtung von Instruktionspaaren für ein deutlich verbessertes Gesamtresultat genügt, ist diese Tabelle nicht sonderlich lang. Für jedes Paar, das man im Bytecodestrom findet, zieht man den entsprechenden Gain-Faktor ab und erhält so eine exaktere Ausführungszeit der gesamten Bytecodesequenz. In dieser Arbeit wurde der zweite Ansatz verfolgt, da er aufgrund der Messung des aktuellen Codes direkt auf der Zielarchitektur noch exaktere Werte verspricht.

Weitere Gründe, die für die zweite Variante sprechen, werden erst bei der Betrachtung der einzusetzende JVM sichtbar. Obwohl der Einsatz eines JITs sich positiv auf das Laufzeitverhalten eines Programms auswirken kann, ist er für den Einsatz im Echtzeitumfeld nicht geeignet. Das größte Problem ergibt sich in der Nichtdeterminiertheit der Dauer einer Bytecodesequenz. Man kann den Overhead, der sich aufgrund der zusätzlichen Rechenzeit für die Sammlung von Laufzeitinformationen über das aktuelle Programm er-

gibt, nur sehr schwer abzuschätzen. Dieses Sammeln ist aber Voraussetzung für die Codeverbesserungen eines JITs und darf daher nicht einfach außer Acht gelassen werden. Zudem kann nicht vorherbestimmt werden, wann eine Methode zur Laufzeit übersetzt wird und wie lange dieser Übersetzungsvorgang dauern wird. Man weiß nicht genau, welche Optimierungen während der Codeerzeugung greifen werden und könnte daher nur unter größtem Aufwand eine äußerst schlechte WCET des Programms angeben.

Ein Interpreter bietet zwar ein deterministisches Laufzeitmodell, ist aber für viele Einsatzgebiete nicht performant genug, so dass es ebenfalls keinen Einsatz im Echtzeitbetrieb findet.

Einzig ein AOT stellt ein vorhersagbares Laufzeitverhalten bei gleichzeitig performanter Codeausführung bereit. Er wirft jedoch gleichzeitig einige Probleme hinsichtlich der Zeitmessung auf Java-Bytecodeinstruktionsebene auf. Ein AOT kann beispielsweise nach einer genauen Datenflussanalyse virtuelle Methodenaufrufe auflösen und auf diese Weise Code erzeugen, der sich die Indirektion über eine Methodentabelle spart. Er kann des Weiteren einen kompletten Methodenaufruf weglassen und den Code der Methode direkt an der Aufrufstellen expandieren (*„inlinen“*). Auch dies führt zu einer Anweisungsfolge, die ein verbessertes Laufzeitverhalten zur Folge hat. Er könnte aber auch Berechnungen vorziehen oder umgestalten, so dass sich u.U. die Ausführungsdauer eines Basisblocks verschlechtert, wenn sich diese Veränderungen gleichzeitig positiv auf die Laufzeit des gesamten Programms auswirken. Man sieht, dass der AOT einen enormen Einfluss auf das Laufzeitverhalten eines Programms und somit auch auf das Laufzeitverhalten einzelner Bytecodeinstruktionen hat. Diesem Einfluss wird man nur gerecht, wenn man nicht versucht die Zeiten einzelner Bytecodeinstruktionen, sondern direkt von den Basisblöcken zu bestimmen.

Da sich ein AOT an die Kontrollanweisungen im Bytecode halten muss, um den Kontrollfluss eines Programms nicht zu verändern, gleichen die Basisblöcke auf Maschinencodeebene den Basisblöcken auf Bytecodeebene[2]. Man übersetzt daher zunächst den Bytecodestrom in ein Maschinenprogramm, zerhackt es anschließend in seine Basisblöcke und misst deren Zeiten. Die gemessenen Zeiten werden schließlich den entsprechenden Basisblöcken auf Bytecodeebene zugeordnet, so dass man nun alle Voraussetzungen für die Suche nach dem teuersten Ausführungspfad erfüllt hat.

[2]Von dieser Regel ausgeschlossen sind natürlich AOTs, die aggressive Codeoptimierungen wie beispielsweise *dead code elimination* durchführen. Sie sollen hier allerdings vernachlässigt werden.

## 4.7 Berechnung der maximalen Abarbeitungszeit

Im folgenden Abschnitt soll zunächst ein einfaches Verfahren zur Berechnung der maximalen Abarbeitungszeit der Anweisungsfolgen eines Kontrollflussgraphen vorgestellt werden. Demgegenüber wird eine praktikablere Berechnungsmethode beschrieben, welche die Struktur des zu analysierenden Graphen in ein Gleichungssystem transformiert, dessen Lösung der maximalen Abarbeitungszeit entspricht.

Beide Verfahren verfolgen spezielle graphbasierte Ansätze mit einer Reihe von Anforderungen an die zu analysierenden Graphen. Da Kontrollflussgraphen jedoch nicht allen diesen Anforderungen gerecht werden, müssen sie zuerst in sogenannte *T-Graphen* (Timing-Analysis-Graphen) (vgl. [Pus93, S.33]) umgewandelt werden.

### 4.7.1 T-Graph

Ein *T-Graph* ist ein zusammenhängender, gerichteter Graph, der die folgenden Eigenschaften besitzt:

1. Er besitzt genau eine Quelle $q$ – ein Knoten, von dem nur Kanten wegführen – und

2. eine Senke $s$ – ein Knoten, zu dem nur Kanten hinführen.

3. Jede Kante ist Teil mindestens eines gerichteten Pfades von der Quelle zur Senke.

4. Es existiert eine Abbildung $\tau$, die jeder Kante $e$ eine nichtnegative Zahl $t = \tau(e)$ zuordnet.

Vor dem Hintergrund der in Kapitel 4.6 geführten Diskussion über die Bestimmung der Abarbeitungzeit von Basisblöcken, lässt sich ein Kontrollflussgraph leicht in einen T-Graphen umwandeln. Dabei müssen folgende Punkte beachtet werden:

- Sowohl die Basisblöcke als auch die Kanten des Kontrollflussgraphen werden bijektiv auf einen zweiten Graphen, den T-Graphen, abgebildet.

- Da jeder Kontrollflussgraph genau einen Einstiegsknoten, seine Wurzel, besitzt, wird dieser zur Quelle des neuen T-Graphen.

- Die Senke des T-Graphen ist ein neuer Knoten, welcher zum Nachfolgerknoten aller ursprünglichen Basisblöcke mit einer **return**-Anweisung wird.

- Die Abbildung $\tau(e)$ weist jeder Kante $e$, die der Kante $Bx \rightarrow By$ des ursprünglichen Kontrollflussgraphen entspricht, die Abarbeitungszeit des Basisblocks $Bx$ (vgl. Kapitel 4.6) zu.

### 4.7.2 Einfaches Verfahren

Da Kontrollflussgraph und T-Graph im Wesentlichen eine Bijektion darstellen, ist die Abarbeitungszeit eines Pfades im Kontrollflussgraph mit den Kosten des entsprechenden Pfades im T-Graph identisch. Betrachtet man daher den Pfad $P = (e_1, ..., e_m)$, mit $e_1 = q$ und $e_m = s$, eines T-Graphen, so lassen sich seine Kosten mit Gleichung 4.2 bestimmen.

$$costs(P) = \sum_{i=1}^{m} \tau(e_i) \tag{4.2}$$

Somit lässt sich die WCET eines Programms auf einfache Weise dadurch berechnen, dass man zunächst die Menge $\pi$ aller möglichen Pfade durch den das Programm repräsentierenden T-Graphen bestimmt, die Kosten dieser Pfade ermittelt und daraus den Maximalwert wählt (vgl. Gleichung 4.3).

$$WCET = \max_{p \in \pi}(costs(p)) \tag{4.3}$$

### 4.7.3 Ganzzahlige lineare Programmierung

Es ist ein mühsames Unterfangen, zuerst *alle* möglichen Ausführungspfade aufzuzählen, anschließend ihre Kosten zu berechnen, um daraufhin den teuersten von ihnen zu ermitteln. Aus diesem Grund soll nun ein zweites Verfahren vorgestellt werden, das seinen Hauptnutzen aus einer einfachen und kompakten Problembeschreibung zieht. Anstatt alle zulässigen Pfade auszuzählen, werden die gemeinsamen Eigenschaften aller Pfade in ein mathematisches Gleichungssystem konvertiert, das anschließend mithilfe eines effizienten Verfahrens gelöst wird, um auf diese Weise die Kosten des teuersten Pfades zu bestimmen.

Die Idee für das Berechnungsverfahren basiert auf dem graphentheoretischem Konstrukt der *Zirkulation* auf einem Graphen. Mit ihrer Hilfe werden eine *lineare Zielfunktion* und eine endliche Anzahl von *Restriktionen* aufgestellt, welche als lineare Gleichungen und Ungleichungen vorliegen.

Die Maximierung der Zielfunktion unter Berücksichtigung der Restriktionen wird auch als *lineare Optimierung* oder *ganzzahlige lineare Programmierung* (engl. *Integer-Linear-Programming*, kurz *ILP*) bezeichnet. Die Lösung dieses ILP-Problems entspricht gleichzeitig der Berechnung der maximalen Abarbeitungszeit (vgl. [Pus93]).

### Definitionen

Um die weiteren Erläuterungen zu vereinfachen, vorab einige Definitionen:

- Bezeichne $E$ die Menge aller Kanten und $V$ die Menge aller Knoten eines Graphen, so heißt *„eine Abbildung $f : E \to \Re$ [...] Zirkulation auf einem Graphen G, wenn sie dem Erhaltungssatz (Flusserhaltungssatz) genügt ($e^+$ steht dabei für den Anfang, $e^-$ für das Ende einer gerichteten Kante)“* (vgl. [Pus93, S.44]):

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

  Vereinfacht ausgedückt bedeutet dies, dass die Summe aller Flüsse, die einen Knoten über seine Kanten erreicht, der Summe aller Flüsse entspricht, die diesen Knoten wieder verläßt.

- Die *Kapazitätsfunktionen* $b : E \to \Re$ und $c : E \to \Re$, mit $b(e) \leq c(e)$, begrenzen den Fluss einer Kante $e$. *„Eine Zirkulation $f$ heißt dann zulässig, wenn für alle Kanten $e$ gilt“* (vgl. [Pus93, S.44]):

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

- Bezeichnen $\mu(e)$ die Kosten einer Kante, so definiert man die *Kosten* $\gamma(f)$ einer Zirkulation $f$ durch:

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

### Erweiterter T-Graph

Da der Erhaltungssatz für alle Knoten eines Graphen gelten muss, kann man die Zirkulation nur auf *geschlossenen* Graphen berechnen. Um daher die Zirkulation auf einem T-Graphen berechnen zu können, erweitert man ihn um eine *Rückkehrkante* $e_r = s \to q$ mit der Abarbeitungszeit $\tau(e_r) = 0$ und

legt die Kapazitätsfunktionen *b(e)* und *c(e)* folgendermaßen fest:

$$b(e) = \begin{cases} 1, & \text{wenn } e = e_r \\ 0, & \text{sonst} \end{cases} \tag{4.7}$$

$$c(e) = \begin{cases} 1, & \text{wenn } e = e_r \\ r(e), & \text{sonst} \end{cases} \tag{4.8}$$

Dabei bezeichnet $r(e)$ eine nichtnegative Funktion, die jeder Kante ihre maximale Anzahl an Durchläufen auf einem beliebigen Pfad zuordnet. Für die meisten Kanten ist $r(e) = 1$, da sie höchstens einmal durchlaufen werden. Im Falle von „Schleifenkanten“ ist $r(e) \geq 1$ und bezeichnet so die maximale Anzahl an Iterationen (s.u.).

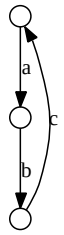
Zudem weist man den Kosten der Kanten $\mu(e)$ die Abarbeitungszeiten $\tau(e)$ des T-Graphen zu, so dass gilt:

$$\forall e \in E : \mu(e) = \tau(e) \tag{4.9}$$

und man erhält auf diese Weise den *erweiterten T-Graph*.

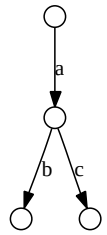
**Folgerungen**

- *Sequenzen*



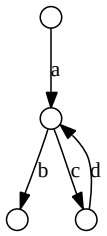

Die Abbildung links zeigt einen Graphen mit einer Kantensequenz $a$ und $b$ sowie einer Rückkehrkante $c$. Damit der Fluss an Kante $c$ den Wert 1 (Voraussetzung des erweiterten T-Graphen s.o.) erhält (Flusserhaltung: $f(a) = f(b) = f(c)$), müssen beide Kanten $a$ und $b$ ebenfalls den Wert 1 annehmen.

- *Verzweigungen*




Angenommen ein Fluss von 1 läuft über Kante $a$ in den zentralen Knoten des linken Graphen, so muss ebenfalls ein Fluss von 1 diesen Knoten *entweder* über Kante $b$ *oder* über Kante $c$ verlassen ($f(a) = f(b) + f(c)$). Eine Zirkulation diskreter Werte beschreibt einen bestimmten Pfad an einer Verzweigung, indem sie den Fluss nur über eine Kante erhält – die andere Kante nimmt zwangsläufig den Wert 0 ihrer unteren Kapatitätsfunktion $b(e)$ an.

- *Schleifen*




Angenommen, die durch die beiden Kanten $c$ und $d$ beschriebene Schleife des linken Graphen würde maximal 10mal durchlaufen, so kann man für diesen Subgraphen eine gültige Zirkulation angeben, indem man die Kapazitätsfunktionen für die Kanten $c$ und $d$ auf $c(c) = c(d) = 10$ setzt. Zusammen mit $c(a) = c(b) = 1$ und $b(a) = b(b) = b(c) = b(d) = 0$ kann auf diese Weise nicht nur ein einziger Pfad, sondern eine Menge gültiger Pfade beschrieben werden: {(f(a) = 1, f(b) = 1, f(c) = 0, f(d) = 0), ..., (f(a) = 1, f(b) = 1, f(c) = 10, f(d) = 10)}.

Besonders der letzte Punkt macht deutlich, dass man mithilfe von Gleichungen eine Zirkulation auf einem T-Graphen angeben kann, die in der Lage ist, eine Menge von Pfaden zu beschreiben. Es ist in der Tat so, dass man allein aufgrund der Struktur des Graphen ein Gleichungssystem erhält, dessen ganzzahlige Lösungen *alle* möglichen Pfade des Graphen beschreiben.

[Pus93] hat jedoch gezeigt, dass Lösungen eines solchen Gleichungssystems auch Lösungen beinhalten, die innerhalb der Zirkulation *nicht stark zusammenhängende Zirkulationssubgraphen* enthalten können (graphisch veranschaulicht sind das Lösungen, die mehrere Pfade innerhalb eines Graphen besitzen, die keine verbindenden Kanten haben). Da in der WCET-Analyse Programmpfade betrachtet werden, muss man dafür Sorge tragen, dass nur Lösungen zugelassen werden, die ausschließlich stark zusammenhängende Zirkulationsgraphen repräsentieren! [Pus93] zeigt, wie man dies mit sogenannten *Flussrestriktionen* garantieren kann.

### Flussrestriktionen

[Pus93, S.42] definiert *Flussrestriktionen* als *„[...] Gleichungen oder Ungleichungen der Form*

$$\sum_{e_i \in E} a_i f(e_i) \circ \sum_{e_i \in E} a_i' f(e_i) + k, \tag{4.10}$$

*mit* $a_i, a_i' \in Z_0$, $k \in N_0$ *und* $\circ \in \{<, \leq, =\}$, *die die Flüsse von Kanten eines erweiterten Flussgraphen in Beziehung setzen.“*

Da *Zirkulationssubgraphen* nur an Zyklen (Schleifen des Kontrollflusses) innerhalb des erweiterten T-Graphen entstehen können, verhindert man ihr Auftreten einfach, indem man die Kanten der Zyklen in eine zusätzliche Beziehung zu den Kanten des restlichen Graphen setzt. Konkret setzt man den

in den Schleifenkopf von außen eintreffenden Fluss mit dem Fluss der ersten Kante der Schleife in Beziehung.

Um das obige Beispiel aus den Folgerungen zu Schleifen noch einmal aufzugreifen, setzt man die Kanten $a$ und $c$ durch die Flussrestriktion $f(c) \leq 10f(a)$ in Beziehung. Um dies zu verifizieren, betrachte man folgendes Gleichungssystem, das sich aus der Zirkulation des obigen Beispiels ergibt:

$$f(a) + f(d) = f(b) + f(c) \tag{4.11}$$
$$f(c) = f(d) \tag{4.12}$$

Sei $f(a) = 1$, so ergibt sich aus der der Flussrestriktion $f(c) \leq 10f(a)$ ein maximaler Fluss von 10 an $f(c)$. Das führt zu $f(d) = 10$ und dies wiederum zu $f(b) = 1$.

### Berechnung

In den vorangegangenen Abschnitten wurde beschrieben, was man unter einer Zirkulation auf einem Graphen versteht und wie man mithilfe einer Zirkulation einen Pfad im erweiterten T-Graph beschreiben kann, der einem gültigen Pfad im Kontrollflussgraph entspricht. Die Kosten einer Zirkulation wurden ebenfalls in Gleichung 4.6 definiert. Da im erweiterten T-Graph die Kosten der Kanten den Abarbeitungszeiten im T-Graph gleichgesetzt sind und eine Zirkulation im erweiterten T-Graph genau einem Pfad im T-Graph entspricht (dieser wiederum genau einem Pfad im Kontrollflussgraph), ist die Abarbeitungszeit eines Pfades im Kontrollflussgraph mit den Kosten seiner Zirkulation im erweiterten T-Graph identisch. Die maximiale Abarbeitungszeit ergibt sich daher aus Gleichung 4.13.

$$WCET = \max_{f} \sum_{e} \mu(e) f(e) \tag{4.13}$$

Glücklicherweise müssen nun aber nicht erst alle zulässigen Lösungen der Zirkulation aufgezählt werden, um aus ihnen die teuerste auszuwählen. Alle zulässigen Lösungen lassen sich wie beschrieben allgemein mithilfe eines Gleichungssystems und entsprechenden Flussrestriktionen beschreiben. Um daher die WCET eines Programms zu erlangen, erzeugt man zunächst den erweiterten T-Graph, der das Programm beschreibt. Stellt daraufhin eine Reihe von Gleichungen und Ungleichungen auf, welche eine Zirkulation unter Restriktionen auf diesem Graphen beschreibt und maximiert schließlich die WCET-Zielfunktion aus Gleichung 4.13 unter Berücksichtigung der aufgestellten Restriktionen. Hierfür verwendet man ein lineares Optimierungsverfahren und erlangt auf diese Weise die WCET eines Programms.

### LP-Solve

Für das Lösen der Optimierungsaufgaben wurde in dieser Arbeit die Bibliothek *LP-Solve* verwendet. LP-Solve implementiert für das Lösen linearer Optimierungsaufgaben das *revidierte Simplexverfahren*, welches mit das bekannteste und effizienteste Verfahren zur Lösung von ILP-Problemen beschreibt. Für seine Funktionsweise sei auf [IB00b, S.878] verwiesen.

Man kann der LP-Solve-Bibliothek eine lineare Optimierungsaufgabe mit Gleichungen und Restriktionen in Form einer Matrix übergeben und diese von ihr lösen lassen. Neben der eigentlichen Funktionalität enthält LP-Solve etliche Wrapper-Methoden, welche die Benutzung der Bibliothek aus verschiedenen Programmiersprachen erlaubt. Für eine genaue Beschreibung der API sei auf [lps] verwiesen.

## 4.8 WCET-Bestimmung

In den vorangegangenen Abschnitten wurde systematisch erläutert, wie man einzelne Methoden bezüglich ihrer maximalen Ausführungszeit analysiert. Die Frage, die sich als Letzte stellt ist, wie man anhand dieser Zeiten die gesamte WCET eines Programms mit all seinen Unterfunktionsaufrufen, z.T. auch rekursiver Art, bestimmt. Das in dieser Arbeit entwickelte Analyseprogramm geht folgendermaßen vor:

Als erstes wird, in Abhängigkeit einer Ankermethode (bspw. der main-Funktion), der Aufrufgraph des zu analysierenden Programms erstellt. Anschließend wird dieser Aufrufgraph drei vorgezogenen Analysen unterzogen:

1. Enthält der Aufrufgraph *native*[3] Methodenaufrufe und sind diese Aufrufe annotiert?

2. Enthält der Aufrufgraph Rekursionen (Zyklen) und sind diese Rekursionen hinreichend annotiert?

3. Ist die maximale Anzahl der Iterationen jeder Schleife berechenbar?

Nur wenn alle drei Analysen erfolgreich abgeschlossen werden konnten, startet die eigentliche Berechnung der WCET – andernfalls endet die Analyse mit entsprechenden Fehlermeldungen.

Die Idee, die hinter der eigentlichen Bestimmung der WCET steckt, ist relativ simpel:

[3] Unter *nativen Methoden* versteht man Codeabschnitte, die direkt Maschinencode enthalten.

Man startet die Analyse an der Ankermethode und betrachtet zunächst deren Basisblöcke und die darin potentiell enthaltenen Methodenaufrufe. Die Abarbeitungszeit jedes Basisblocks besteht einerseits aus der Zeit, welche für die Abarbeitung der darin kodierten Bytecodesequenz benötigt wird, andererseits aus der Zeit, welche in den aus diesem Block referenzierten Methoden verbracht wird.

Wie aus Kapitel 4.5.2 bekannt ist, können sich hinter einer Methodensignatur verschiedene Implementierungen verbergen. Um daher die schlechtmöglichste Abarbeitungszeit für einen Basisblock zu ermitteln, werden die Ausführungszeiten *aller* potentiell gültigen Methoden errechnet und die schlechteste ausgewählt, damit diese schließlich zur Gesamtzeit des Basisblocks addiert werden kann. Da diese Methodenaufrufe natürlich selbst auch wieder Basisblöcke enthalten, in denen weitere Methoden referenziert werden können, läuft diese Berechnung rekursiv ab. Selbstverständlich werden die Zeiten der Annotationsmethoden herausgefiltert, so dass sie nicht in die Gesamtzeit einfließen.

Sobald die Zeiten aller Basisblöcke bestimmt worden sind, wird, mit dem in Kapitel 4.7.3 beschriebenen Verfahren, die längste Ausführungszeit errechnet.

Einer gesonderten Betrachtung bedürfen rekursive Methodenaufrufe. Würde die Analyse blind einem rekursiven Methodenaufruf folgen, würde sie zwangsläufig in eine endlose Berechnung münden. Aus diesem Grund wird bei der Analyse ein Aufrufstapel (engl. *callstack*) verwaltet. Jedes Mal, bevor eine Methode weiteren Analysen unterzogen wird, wird sie auf dem Aufrufstapel abgelegt. Wenn im Laufe der Analyse nochmals ein Methodenaufruf an eine momentan auf dem Aufrufstapel liegende Methode untersucht werden soll, wird die Abarbeitungszeit für diesen Methodenaufruf mit 0 Zeiteinheiten gewertet und ein Vermerk an der auf dem aktuellen Aufrufstapel liegenden Methode eingetragen. Bevor die mathematische Berechnung der maximalen Abarbeitungszeit einer Methode durchgeführt wird, wird ihr Eintrag vom Aufrufstapel entfernt und auf einen möglichen Vermerk untersucht, ob ein rekursiver Aufruf an diese Methode vorlag. Ist dies der Fall, wird die dann errechnete maximale Abarbeitungszeit der Methode zusätzlich mit der annotierten Rekursionstiefe[4] multipliziert und ergibt auf diese Weise die Gesamtzeit des Aufrufs.

[4]Wie aus Kapitel 4.5.4 bekannt, muss die maximale Rekursionstiefe einer Methode annotiert werden, da momentan dafür keine eigenständige Analyse vorgesehen ist.

# Kapitel 5

# Evaluierung und Ausblick

In den vorangegangenen Kapiteln wurden die einzelnen Phasen zur Bestimmung der WCET eines Java-Programms vorgestellt und auf diese Weise *ein* mögliches Verfahren herausgearbeitet. Dieses Kapitel beschäftigt sich nicht mehr so sehr mit den Techniken zur Bestimmung der WCET, sondern viel mehr mit der Benutzung des in dieser Arbeit entwickelten Programms, mit der Evaluierung eines Fallbeispiels und mit möglichen Ansatzpunkten zu einer umgreifenderen Analyse.

## 5.1 Benutzung

Das entwickelte Programm liegt als jar-Archiv vor und wird mit folgender Kommandozeile gestartet:

```
java -jar wcet.jar [opt] <method> <signature> <archives> <time>

options:
   - f      Construct non-verbose flowgraphs
   - F      Construct verbose flowgraphs
   - c      Construct callgraph
   - s      Print callgraph stats
```

Der Benutzer muss neben dem Namen einer Methode (`<method>`) und deren genauer Signatur (`<signature>`) zusätzlich eine durch Doppelpunkte getrennte Liste von jar-Archiven (`<archives>`), die sämtliche Klassen des zu analysierenden Programms enthalten, und eine Datei (`<time>`) mit allen gemessenen Basisblockzeiten angeben. Die zu spezifizierende Methode entspricht dabei der Ankermethode der anschließenden Analyse.

Daneben können dem Programm einige Optionen übergeben werden, welche die einzelnen Schritte der WCET-Analyse visualisieren: `-f` / `-F` erzeugt

für jede analysierte Methode einen Flussgraphen im Postscriptformat, `-c` generiert den gesamte Aufrufgraphen (ebenfalls als Postscript-Datei), während `-s` einige Statistiken bezüglich den Optimierungen am Aufrufgraphen ausgibt, die durch die Rapid-Type-Analyse (vgl. Kapitel 4.5.3) erzielt werden konnten. Alle Graphen werden im Standardverzeichnis für temporäre Dateien des aktuellen Betriebssystems abgelegt.

Man beachte, dass durch Angabe unterschiedlicher (`<time>`)-Dateien die WCET eines Programms problemlos für verschiedene Architekturen berechnet werden kann.

Falls die Analyse erfolgreich beendet werden konnte, werden neben der WCET des gesamten Programms auch die maximalen Ausführungszeiten aller potentiell aufzurufenden Methoden ausgegeben. Der Benutzer erhält auf diese Weise einen sehr guten Überblick über das Zeitverhalten seiner Methoden und kann diese Zeiten zudem bestens für künftige Optimierungen an seinem Programm nutzen.

Sollten während der Analyse Fehler auftreten, da beispielsweise dem Analyseprogramm eine Annotation fehlt, bricht die Analyse mit einer Fehlermeldung ab, die neben der Fehlerstelle auch eine Fehlerbeschreibung enthält. Da alle Fehlermeldungen anhand folgendem Format generiert werden, lassen sich die Fehlerstellen leicht lokalisieren:

```
ERROR: Could not analyse code
    at <className>.<methodName>(<fileName>:<lineNumber>)
    <description>
```

Angenommen die maximale Iterationsanzahl der Schleife aus dem nachfolgendem Listing konnte nicht ermittelt werden, da der Schleifenparameter `MAX` ein nicht annotierter Methodenparameter ist, bricht die Analyse dieser Schleife mit der Fehlerbeschreibung: `MAX is not an annotated method parameter` ab. Dieses Problem kann leicht dadruch gelöst werden, dass man den Wertebereich von `MAX` einschränkt und folgende Annotation bereitstellt:

```
WCETAnnotation.setRange(MAX, 10, 20);

for (int i = 0; i < MAX; i++) {
    [...]
}
```

Enthält das zu analysierende Programm eine rekursiv formulierte Fakultätsfunktion, von der man genau weiß, dass sie stets mit Werten von 1 bis maximal 42 aufgerufen wird und daher die maximale Rekursionstiefe den Wert 42 nicht überschreitet, so wird dies dem Analyseprogramm durch nachfolgende Annotation bekanntgegeben:

```
public int factorial(int n) {
    WCETAnnotation.setRecursionDepth(42);

    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Auf diese Weise verhindert man den Abbruch der Analyse mit der Fehlerbeschreibung: `No recursion depth annotation found`.

## 5.2 Fallbeispiel

Der folgende Abschnitt widmet sich der Betrachtung eines kleinen Fallbeispiels, um die korrekte Funktionsweise des Analyseprogramms mit einigen Messwerten zu untermauern. Zu diesem Zwecke soll nochmals die iterative Implementierung der Fakultätsfunktion aus Listing 3.1 herangezogen werden. An dieser einfachen Funktion soll die Abweichung zwischen real gemessenen Werten und dem Resultat der statischen Analyse untersucht werden. Aus diesem Grund wird die Laufzeit der Fakultätsfunktion mit linear ansteigenden Eingabedaten gemessen, wobei für die Berechnung der WCET ein fiktiver Maximaleingabeparameter festgelegt wurde.

Alle anschließenden Messungen wurden auf einem Intel Pentium III Prozessor mit 500 Mhz ausgeführt. Um einen besseren Überblick über den auszuführenden Maschinencode zu gewinnen, zeigt Listing 5.1 eine Gegenüberstellung des aus Listing 3.1 erzeugten Java-Bytecodes und des wiederrum daraus resultierenden Maschinencodes sowie deren Aufteilung in einzelne Basisblöcke.

```
# B0
                        sub     $0x4,%esp
  0: iconst_1
  1: istore_2           movl    $0x1,0xfffffffc(%ebp)
  2: goto 12            xor     %eax,%eax
                        jmp     0x4054c1fd

# B5
  5: iload_1            mov     0xc(%ebp),%edx
  6: iload_2            mov     0xfffffffc(%ebp),%ebx
```

```
  7: imul            imul    %ebx,%edx
  8: istore_2        mov     %edx,0xfffffffc(%ebp)
  9: iinc 1 -1       mov     0xc(%ebp),%esi
                     dec     %esi
                     mov     %esi,0xc(%ebp)

# B12
 12: iload_1         mov     0xc(%ebp),%edx
 13: iconst_1
 14: if_icmpgt 5     cmp     $0x1,%edx
                     xor     %eax,%eax
                     test    %eax,%eax
                     jne     0x4054c252

# B17
 17: iload_2         mov     0xfffffffc(%ebp),%eax
 18: ireturn         leave
                     ret
```

Listing 5.1: Bytecode und Assemblercode zu Listing 3.1

Man beachte, dass bei der Basisblockzeitmessung Sprungbefehle nicht zu 100% exakt simuliert werden können, da während der Messung nur „Dummywerte“ zur Verfügung stehen. Grund hierfür ist die Tatsache, dass die Pentium-CPU vor einem Sprungbefehl Annahmen darüber macht, ob nach dem Vergleich *gesprungen wird oder nicht* und auf diese Weise bereits nachfolgende Befehle ausführt. War die Sprungvorhersage richtig, verbessert sich das Zeitverhalten, war sie falsch wird die Ausführungszeit länger. Da der Pentium die Sprungvorhersage zur Laufzeit dynamisch anpasst, kann kaum garantiert werden, dass stets der „langsamere Pfad“ gewählt wird – es sei denn, man schaltet die dynamische Sprungvorhersage des Pentium ab und ordnet die Befehle so, dass das Resultat der Vergleichsoperation vor dem Sprung stets zum „falschen Ziel“ führt. Auf diese Weise würde man stets die für die WCET-Bestimmung *sicherere* Messung erhalten.

Da im Experiment wiederholte Messungen stets zu exakt identischen Messwerten führten, zeigt Tabelle 5.1 keine Messreihen, sondern nur den Absolutwert der Messung. Für die Messung wurde der `rdtsc`-Befehl des Pentium Prozessors verwendet, der den *Time-Stamp-Counter* nach jedem CPU-Takt um eins inkrementiert. Mithilfe der Taktzahl der CPU wurden schließlich die Messergebnisse in Zeitwerte konvertiert.

Um die WCET der Fakultätsfunktion schließlich bestimmen zu können, wurde der Eingabeparamter `n` mit einem Maximalwert von 20 annotiert: `WCETAnnotation.setValue(i, 20)`. Mithilfe der Annotation und den Mess-

| **Basisblock** | **Messwert** |
|---|---|
| *B0* | 6 ns |
| *B5* | 14 ns |
| *B12* | 6 ns |
| *B17* | 1 ns |

Tabelle 5.1: Ergebnisse der Basisblockmessung für Listing 5.1

werten der Basisblöcke, errechnete das Analyseprogramm die WCET zu 393 ns ($\hat{=} t(B0) + 19t(B5) + 20t(B12) + t(B17)$).

Eine endgültige Bestätigung dieses Ergebnisses liefert der Graph aus Abbildung 5.1. Hier wurde die Zeit der gesamten Fakultätsberechnung für unterschiedliche Eingabewerte gemessen. Die Messanordnung war dabei identisch wie zur Basisblockmessung. Man sieht deutlich, dass alle Messergebnisse unterhalb der errechneten WCET lagen. Man betrachte besonders die gemessene Zeit von $factorial(20)$, die bei 384 ns liegt. Dabei ergibt sich eine Abweichung von nur 2,3% zur berechneten WCET, so dass an dieser Stelle besonders die Genauigkeit des Analyseverfahrens betont werden soll.

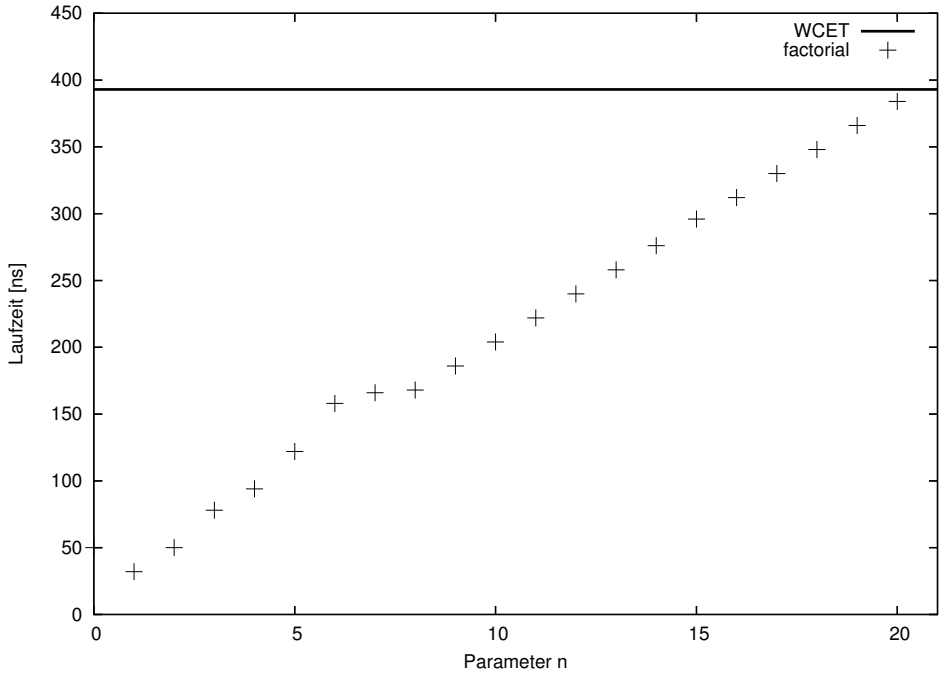



Abbildung 5.1: Gemessene Dauer der Fakultätsfunktion aus Listing 3.1

## 5.3 Ausblick

Obwohl das in dieser Arbeit entwickelte Programm in der Lage ist, beliebigen Java-Bytecode (notfalls mit Rückgriff auf Annotationen) auf sein Zeitverhalten zu analysieren, stellt sich dennoch die Frage, inwieweit man das Konzept erweitern könnte, um exaktere Zeiten bestimmen zu können oder den Analysevorgang weiter zu automatisieren, damit man vor allem mit weniger Zusatzinformationen in Form von Annotation auskommen kann. Zusätzlich sollte geklärt werden, ob nicht eine Möglichkeit besteht, die berechneten Zeiten zu validieren. Dies würde zu mehr Sicherheit bezüglich der Korrektheit der Ergebnisse beitragen.

### 5.3.1 Erweiterungen

Ein momentan unbefriedigender Zustand ist die Tatsache, dass keinerlei Methoden innerhalb der Abbruchbedingung einer Schleife (vgl. Kapitel 4.3.3) auswertbar sind. Einzig die Methode `size()` einer Collection bildet hier eine Ausnahme, da sie ausgewertet werden kann, sobald die Collection annotiert ist. Doch dies allein ist für einen allgemeinen Einsatz nicht ausreichend.

Aus diesem Grund sollte ein weiter greifenderer Annotationsmechanismus eingeführt werden, damit beliebige Methoden annotiert werden können. Eine elegante Lösungsvariante könnten dabei die im *JDK 5* eingeführten *Annotations* (vgl. [JG05]) bieten. Wie Listing 5.2 zeigt, wäre es mit ihrer Hilfe möglich, ein spezielles „Annotationsinterface“ zu definieren, durch das der Rückgabewertebereich einer Methode annotiert und so während der Analyse ausgewertet werden könnte.

```
public @interface ReturnValueCoDomain {
    int beginning();
    int end();
}

public class A {

    @ReturnValueCoDomain(beginning = -10, end = 10)
    public int foo() { ... }
}
```

Listing 5.2: Annotationsmechanismen auf Methodenebene

Wünschenswert wäre natürlich ein Mechanismus, der es erlaubt, *ohne* Annotationen auszukommen. Man müsste daher bereits zum Analysezeit-

punkt wenigstens in der Lage sein, Methoden soweit auszuwerten, dass man ihren Rückgabewertebereich möglichst genau abschätzen kann. Mit einem derartigen Verfahren könnte man gleichzeitig auch die maximale Rekursionstiefe jeder Methode ausrechnen, ohne dass ein Programmierer sie annotieren müsste. Aus der Theoretischen Informatik weiß man jedoch, dass ein solches Unterfangen nicht allgemein umsetzbar ist. Inwiefern es sich daher lohnt, Algorithmen zu entwerfen, die einige wenige Spezialfälle abdecken könnten, ist fraglich – im Einzelfall aber vielleicht dennoch sinnvoll und somit überlegenswert.

Ein bislang nur am Rande angesprochenes Thema ist die Behandlung von Ausnahmen (*Exceptions*) während der Zeitanalyse. Die meisten Ausnahmen können nur in speziell gekennzeichneten Codeabschnitten geworfen werden und müssen anschließend von sogenannten *Exceptionhandlern* aufgefangen und behandelt werden. Wenn man sich daher strikt an die Definition der WCET halten möchte, müsste man zur maximalen Ausführungszeit eines Codeabschnittes, in dem eine Exception auftreten kann, zusätzlich auch die Abarbeitungszeit der Instruktionen im Handler addieren.

Zusätzlich zu den soeben angesprochenen Ausnahmen kennt die Programmiersprache Java sogenannte *Runtime-Exceptions*. Das sind Ausnahmen, für die keine Handler angegeben werden müssen, allerdings angegeben werden können! Sie können nach fast jeder Bytecodeinstruktion auftreten, beispielsweise nach einer Division durch Null, einem Pufferüberlauf oder dem Überschreiten von Arraygrenzen.

Ausnahmen ermöglichen zwar eine elegante Behandlung von Fehlersituationen, machen es allerdings sehr schwer, ein Modell zu beschreiben, das sie adäquat während der WCET-Analyse berücksichtigt. Würde man den zusätzlichen Aufwand, der aufgrund von Ausnahmen entsteht, während der WCET-Bestimmung penibel dazuaddieren, würde man am Ende keine aussagekräftige Zeit über das Laufzeitverhalten des analysierten Programms mehr erhalten. Diese Zeit ist schlicht zu weit entfernt von der WCET, die normalerweise während eines fehlerfreien Betriebs entsteht. Aus diesem Grund behandelt das Analyseprogramm grundsätzlich *keine* Zeiten von Exceptionhandlern.

Solange Programmiere Ausnahmen nur dann einsetzen, um schwerwiegende Fehler eines Programms anzuzeigen, die notgedrungen zum Abbruch führen, ist dieses Vorgehen durchaus praktikabel. Sobald Ausnahmen allerdings für die gesamte Fehlerkommunikation eingestzt werden sollen, ist dieses Verfahren zu strikt und hat zu optimistische Ergebnisse zur Folge. Man müsste sich daher Gedanken über einen zusätzlichen Annotationsmechanismus machen, mit dessen Hilfe Exceptionhandler markiert werden können, deren Ausführungszeit während der Analyse berücksichtigt werden soll. Die-

se Zeiten würden in die anschließende Berechnung einfließen und auf diese Weise zu einer höheren Verlässlichkeit der Ergebnisse beitragen.

Die Realisierung einer *Points-To-Analyse* (vgl. [Ste96]) für die WCET-Bestimmung könnte sich gleich in zweierlei Beziehungen positiv auswirken. Sie versucht die Beziehung zwischen Daten und Referenzen zu verfolgen und erlaubt daher sehr exakte Aussagen darüber, welche Objekte sich zur Laufzeit hinter einer Referenz verbergen. Wenn es dadurch möglich ist, beispielsweise eine Aussage darüber treffen zu können, wie gross ein Array zu einem bestimmten Zeitpunkt im Programmfluss ist, müsste die Arraygrösse nicht länger annotiert werden und der `length`-Operator von Arrays könnte direkt ausgewertet werden. Analog gilt dies natürlich auch für die Größe von Collections, wenn man zum Analysezeitpunkt protokolliert, wie viele Elemente maximal in eine Collection eingehängt werden.

Ein exaktes Wissen über die tatsächlichen Objekte hinter Referenzen hilft zudem bei der Auflösung von *call sites* zu *call instances* (vgl 4.5.2). Je mehr Objekttypen man ausschließen kann, die sich potentiell hinter einer Objektreferenz befinden können, desto schmaler wird der Aufrufgraph und desto präziser die zu errechnende WCET. In [AR01] heißt es beispielsweise, dass deren Points-To-Implementierung Aufrufgraphen erzeugt, die im Schnitt 14% weniger Knoten enthalten als die durch eine Rapid-Type-Analyse optimierten Graphen. Es ist damit zu rechnen, dass sich eine derartige Verkleinerung spürbar positiv auf die Ergebnisse der WCET-Bestimmung auswirken würde.

### 5.3.2 Validierung

Die einfachste Variante, mehr Sicherheit über die Korrektheit der errechnete WCET eines Programms zu erlangen, ist die Messung. Man startet das analysierte Programm mit beliebigen Eingabewerten, misst die für die Berechnung benötigte Zeit und vergleicht sie anschließend mit der berechneten WCET. Liegen die gemessenen Zeiten unter der WCET, scheint die Berechnung korrekt zu sein. Um mögliche Fehler in der Berechnung schneller lokalisieren zu können, bieten sich auch *Timer* für die einzelnen Methoden des Programms an, die vor einem Methodenaufruf mit 0 initialisiert und gestartet, nach Rückkehr der Methode gestoppt und schließlich ausgewertet werden. Die Auswertung könnte darin bestehen, den gemessenen Wert mit dem zuvor für diese Methode berechneten zu vergleichen und gegebenenfalls eine Fehlermeldung zu generieren. Aus der gleichen Argumentation wie in Kapitel 2.1, ist die Messvariante jedoch nicht sonderlich befriedigend. Daher sollte man sich einer etwas anderen Betrachtung zuwenden.

Wenn man davon ausgeht, dass das Analyseprogramm korrekte Werte liefert, sofern die Grundlagen für die Berechnung, wie beispielweise der Werte-

bereich von Variablen oder die maximale Schleifeniterationsanzahl, zutreffen, reicht im Grunde die Validierung der Grundlagen aus, um die Korrektheit der Berechnung zu garantieren. Geht man des Weiteren davon aus, dass das Analyseprogramm den Wertebereich von Schleifenparametern oder die maximale Iterationsanzahl korrekt ermittelt, sofern sie sich ausschließlich anhand des Programmcodes berechnen lassen, so kann man die Validierung auf die Überprüfung der Annotationen beschränken. Unter den soeben beschriebenen Voraussetzungen liefert das Anaylseprogramm daher nur dann inkorrekte Werte für die WCET, wenn fehlerhafte Annotationen vorliegen. Aus diesem Grund sollte man sich auf die Validierung dieses Aspektes konzentrieren.

Realisieren ließe sich eine derartige Validierung mithilfe aspektorientierter Programmierung. Soll beispielsweise der Wertebereich eines Schleifenparameters überprüft werden, könnte man in das Programm Validierungscode einweben, der vor dem Zugriff auf einen Schleifenparameter dessen aktuellen Wert überprüft und Alarm schlägt, sobald dieser Parameter den zulässigen bzw. annotierten Wertebereicht verlässt. Die Überprüfung der maximalen Schleifeniterationsanzahl ließe sich auf ähnliche Weise realisieren, indem man für jede annotierte Schleife einen zusätzlichen Zähler mitführt, der in jedem Schleifendurchlauf inkrementiert und nach Verlassen der Schleife überprüft wird. Ein überhöhter Wert könnte anschließend zum Auslösen einer Fehlermeldung genutzt werden. Analog ließen sich auf diese Weise alle weiteren Annotationen, wie Arraygröße, Collectiongröße oder Rekursionstiefe bestätigen.

# Kapitel 6

# Resümee

Ziel dieser Arbeit war es, ein Programm zur Berechnung der *Worst-Case-Execution-Time* (WCET) von Java-Programmen zu entwerfen und zu implementieren. Es sollte in der Lage sein, unabhängig vom Compiler und der zugrunde liegenden Laufzeitumgebung, diese Zeit möglichst exakt zu bestimmen.

Nachdem zu Beginn (Kapitel 2) die grundlegenden Begriffe dieser Arbeit diskutiert worden sind, beschäftigte sich Kapitel 3 mit ausgewählten Konzepten der Programmanalyse. Dabei stand besonders die *Kontrollflussanalyse* mit ihren Graphen und Algorithmen, sowie die *Datenflussanalyse*, im Speziellen die Problematik der *verfügbaren Definitionen*, im Vordergrund.

Kapitel 4 widmete sich schließlich der eigentlichen Berechnung der WCET. Dabei galt zunächst alle Aufmerksamkeit der Analyse des Kontrollflusses eines Java-Programms. Einen sehr hohen Stellenwert nahm in diesem Zusammenhang das Aufspüren von Schleifen, deren Typbestimmung und die Berechnung ihrer Schranken ein. Dabei wurde detailliert beschrieben wie man alle für die Berechnung benötigten *Schleifenparameter* findet, ihren Wertebereich bestimmt und anschließend mithilfe *interpretativer Ausführung* die maximale Anzahl der Schleifeniterationen bestimmt.

Gleichzeitig jedoch wurden auch die Grenzen der automatisierten Schleifenanalyse aufgezeigt, worauf hin ein *Annotationsmechanismus* vorgestellt wurde, der es einem Programmierer erlaubt sein Programm mit Informationen zu versehen, die für eine erfolgreiche WCET-Bestimmung unumgänglich sind.

Ein weiterer Kernpunkt dieser Arbeit war außerdem eine möglichst genaue Untersuchung *virtueller Methodenaufrufe*, da sie neben der maximalen Schleifenanzahl eine entscheidende Rolle bei der Ausführungszeit eines Programms einnehmen. Im Zuge dessen wurden Konzepte der *Klassenhierarchieanalyse* vorgestellt und *Aufrufgraphen* untersucht, wobei für Letztere in

Form der *Rapid-Type-Analyse* ein Optimierungsverfahren präsentiert wurde.

Um die eigentliche WCET-Berechnung möglichst elegant zu gestalten, wurde auf das graphentheoretische Konstrukt der *Zirkulation* auf einem Graphen zurückgegriffen. Mit dessen Hilfe lässt sich ein Un-/Gleichungssystem erzeugen, das durch *ganzzahlige lineare Programmierung* gelöst werden kann und dessen Lösung der WCET des zu analysierenden Programms entspricht. Hierfür wurden sowohl das theoretische Hintergrundwissen, als auch die praktische Umsetzung vorgestellt.

Wie man Kapitel 5.3 entnehmen kann, wurden nicht alle Möglichkeiten der statischen Programmanalyse ausgereizt. Obwohl man mit zusätzlichen Analyseschritten den Vorgang der WCET-Berechnung weiter automatisieren könnte bzw. zu exakteren und verlässlicheren Ergebnissen gelangen würde, werden bereits jetzt viele Probleme der Zeitanalyse durch das in dieser Arbeit entwickelte Analyseprogramm abgedeckt. Es nimmt dem Programmierer echtzeitfähiger Anwendungen weitestgehend die mühsame Aufgabe ab, das Zeitverhalten seiner Anwendungen von Hand zu bestimmen, ohne ihn bei seiner eigentlichen Problemlösung einzuschränken. Es legt keine Restriktionen bezüglich der Programmiersprache Java auf, so dass sämtliche Schleifenkonstrukte, das Überschreiben von Methoden und sogar Rekursionen zugelassen sind, um nur einige beliebte Techniken zu nennen. Dies ist für ein Analyseprogramm dieser Art nicht selbstverständlich, wie man beispielsweise in [Kir02] sieht. Dort wird die Programmiersprache WCETC vorgestellt, die der Programmiersprache C unter anderem Einschränkungen bezüglich zulässiger Schleifenkonstrukte auferlegt und zudem die Verwendung direkter und indirekter Rekursionen untersagt. Erst durch diese Maßnahmen konnte dort die Grundlage für eine anschließende Berechnung der WCET geschaffen werden. Zudem erzeugt das Programm umfassende visuelle und textuelle Analyseergebnisse, die dem Programmierer echtzeitfähiger Anwendungen bei der Interpretation des Zeitverhaltens seiner Applikationen weiterhelfen können.

Zusammenfassend kann man daher sagen, dass es tatsächlich gelungen ist, ein solides Konzept für die Berechnung der WCET von Java-Programmen zu erarbeiten. Das Analyseprogramm ist in der Lage, beliebigen Java-Bytecode zu untersuchen, macht den Programmierer durch geeignete Fehlermeldungen auf nicht analysierbare Codeabschnitte aufmerksam und realisiert erste Optimierungen hinsichtlich der eigentlichen Berechnung.

# Anhang A

# BCEL - JavaClass

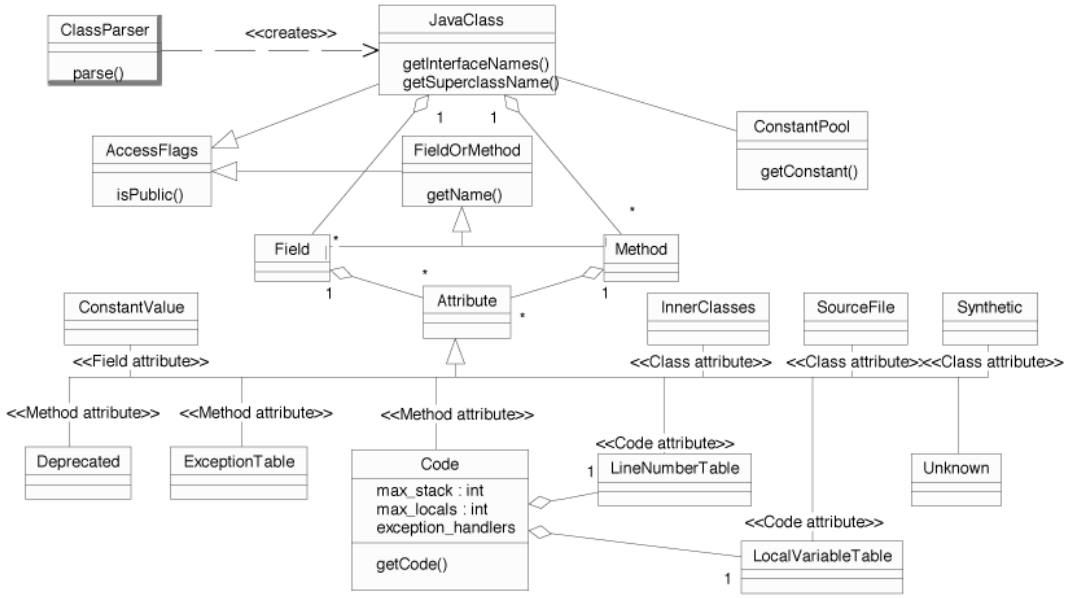



Abbildung A.1: JavaClass

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Listings

# Literaturverzeichnis


[AR01] Atanas Rountev, Ana Milanova und Barbara G. Ryder: *Points-to Analysis for Java Using Annotated Constraints.* 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, 2001.

[AVA88] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: *Compilerbau.* Addison-Wesley, 1988.

[Bac85] Bacon, David Francis: *Fast and Effective Optimization of Statically Typed Object-Oriented Languages.* Dissertation, University of California, Berkeley, 1985.

[Bol00] Bollella, Greg: *The Real-Time Specification for Java.* Addison-Wesley, 2000.

[EG97] Erich Gamma, Richard Helm und Ralph E. Johnson: *Design Patterns.* Addison-Wesley Professional, 1997.

[Eng02] Engblom, Jakob: *Processor Pipelines and Static Worst-Case Execution Time Analysis.* Dissertation, Uppsala, 2002.

[Gab05] Gabriel, Stefan: *Evaluierung und Implementierung einer echtzeitfähigen Speicherbereinigung für das Betriebssystem JX.* Studienarbeit, Friedrich Alexander Universität, 2005.

[GB00] Guillem Bernat, Alan Burns und Andy Wellings: *Portable Worst-Case Execution Time Analysis Using Java Byte Code.* 12th Euromicro Conference on Real-Time Systems, 2000.

[Gol02] Golm, Michael: *The Structure of a Type-Safe Operating System.* Dissertation, Friedrich Alexander Universität, 2002.

[IB00a] Iain Bate, Guillem Bernat, Greg Murphy und Peter Puschner: *Low-Level Analysis of a Portable Java Byte Code*

*WCET Analysis Framework*. 7th International Conference on Real-Time Computing Systems and Applications, 2000.

[IB00b] Ilja Bronstein, Konstantin Semendjajew, Gerhard Musiol und Heiner Mühlig: *Taschenbuch der Mathematik*. Harri Deutsch Verlag, 2000.

[jak] *The Apache Jakarta Project*. http://jakarta.apache.org/, Stand: 9.8.2005.

[JG05] James Gosling, Bill Joy, Guy Steele und Gilad Bracha: *The Java Language Specification, Third Edition*. Addison-Wesley Professional, 2005.

[Kir02] Kirner, Raimund: *The Programming Language wcetC*. Technische Universität Wien, Institut für Technische Informatik, 2002.

[lps] *lpsolve reference guide*. http://www.geocities.com/lpsolve/, Stand: 1.7.2005.

[MC04] Matteo Corti, Thomas Gross: *Approximation of the Worst-Case Execution Time Using Structural Analysis*. EMSOFT'04, 2004.

[MD] Markus Dahm, Jason van Zyl, Enver Haase: *Byte Code Engineering Library*. http://jakarta.apache.org/bcel/index.html, Stand: 9.8.2005.

[MG02] Michael Golm, Meik Felser, Christian Wawersich und Jürgen Kleinöder: *The JX Operating System*. 2002 USENIX Annual Technical Conference, 2002.

[Muc97] Muchnick, Steven: *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, 1997.

[Phi04] Philippsen, Michael: *Übersetzerbau*. Friedrich Alexander Universität Lehrstuhl für Informatik 2, 2004.

[Pus93] Puschner, Peter: *Zeitanalyse von Echtzeitprogrammen*. Dissertation, Technische Universität Wien, 1993.

[Sch01] Schöning, Peter: *Theoretische Informatik – kurzgefasst*. Spektrum Akademischer Verlag, 2001.

[Sed02] Sedgewick, Robert: *Algorithmen in C++.* Addison-Wesley Longman, 2002.

[SM] Sun Microsystems, Inc.: *Java Card Technology Overview.* http://java.sun.com/products/javacard/overview.html, Stand: 1.7.2005.

[SP03] Schröder-Preikschat, Wolfgang: *Betriebssysteme.* Friedrich Alexander Universität Lehrstuhl für Informatik 4, 2003.

[Ste96] Steensgaard, Bjarne: *Points-to Analysis in Almost Linear Time.* Conference Record of the Symposium on Principles of Programming Languages, 1996.

[TL99] Tim Lindholm, Frank Yellin: *The Java Virtual Machine Specification.* Addison-Wesley Professional, 1999.

[VS00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefe et al.: *Practical Virtual Method Call Resolution for Java.* 15th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, 2000.

[Waw01] Wawersich, Christian: *Design und Implementierung eines Profilers und optimierenden Compilers fuer das Betriebssystem JX.* Diplomarbeit, Friedrich Alexander Universität, 2001.

[wik] *Wikipedia. Die Freie Enzyklopädie.* http://de.wikipedia.org/, Stand: 1.7.2005.

# Index