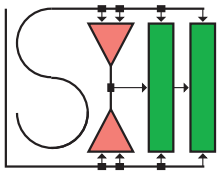
technische universität
dortmund

Diplomarbeit

Schleifenanalyse für einen WCET-optimierenden Compiler basierend auf Abstrakter Interpretation und Polylib

Daniel Cordes



Technische Universität Dortmund
Lehrstuhl Informatik XII

30. April 2008

Gutachter:

Dipl.-Inform. Paul Lokuciejewski
Prof. Dr. Peter Marwedel

Vorwort

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich bei der Entwicklung dieser Diplomarbeit in den unterschiedlichsten Funktionen unterstützt haben.

In erster Linie gebührt mein Dank vor allem meinem Betreuer Paul Lokuciejewski. Er hat mich während der Diplomarbeit immer wieder durch seinen außergewöhnlichen Einsatz motiviert und maßgeblich zur Erstellung dieser Arbeit beigetragen.

Des Weiteren möchte ich meiner Freundin Regina Fritsch danken, dass sie mich während dieser nicht ganz einfachen Zeit mit Ihrer Liebe und Rücksichtnahme unterstützt hat.

Ebenso gebührt mein Dank meinen Eltern Claudia und Ernst Walter Cordes sowie meinem Bruder Steven Cordes, die während meines gesamten Studiums mit Interesse und Unterstützung dazu beigetragen haben, dass ich mein Studium erfolgreich bestreiten konnte.

Meinem Freundeskreis möchte ich an dieser Stelle ebenfalls danken, da dieser für die notwendige Abwechslung in meiner Freizeit gesorgt hat, so dass ich mich jeden Tag mit voller Konzentration dieser Arbeit widmen konnte.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Dieses Kapitel soll die vorliegende Diplomarbeit einführen. Dabei wird die Entwicklung einer automatischen Schleifenanalyse in Abschnitt 1.1 motiviert. Im darauf folgenden Abschnitt 1.2 wird ein kurzer Überblick über Publikationen gegeben, die bereits zu diesem Thema veröffentlicht worden sind. Im Kontrast dazu werden in Abschnitt 1.3 die Ziele beschrieben, welche in dieser Arbeit erreicht werden sollen. Abschließend wird in Abschnitt 1.4 die Struktur der vorliegenden Arbeit erläutert.

1.1 Motivation

Die erste Verlagerung des Anwendungsbereichs von Informationssystemen begann bereits Anfang der 90er Jahre, in dem die großen Rechenzentralen durch die *personal computers* (PC) abgelöst wurden. Dieser Trend hat sich im späteren Verlauf dahingehend geändert, dass die zurückgehende Bedeutung der PCs in der jüngeren Vergangenheit durch die zunehmende Relevanz der *eingebetteten Systeme* abgelöst wurde. So wurde bereits im Jahr 2000 geschätzt, dass 90% aller eingesetzten Prozessoren in eingebetteten Systemen wiederzufinden sind [Mar07].

Zu diesen eingebetteten Systemen gehören laut Definition alle informationsverarbeitenden Systeme, die in ein größeres Produkt integriert sind [Mar07]. Dazu zählen Hard- und Softwaresysteme, wie sie beispielsweise in Handys, Autos oder auch Haushaltsgeräten eingesetzt werden.

Durch die Integration der eingebetteten Systeme in den Alltag entwickelten sich u.a. die zwei neuen Anwendungsgebiete *ubiquitouis computing* und *ambient intelligence*. Ubiquitous computing setzt sich in diesem Zusammenhang das Ziel, Informationen überall und für jedermann zugänglich zu machen. Ambient Intelligence soll hingegen vielmehr die Lebensqualität jedes Einzelnen verbessern, ohne dass der Benutzer merken soll, dass er mit einem Informationssystem kommuniziert. Zu dieser Kategorie der informationsverarbeitenden Systeme gehören u.a. Technologien, die beispielsweise in „intelligenten“ Häusern eingesetzt werden. Die Ziele dieser neuen Anwendungsgebiete sind mit den herkömmlichen Desktop-PCs nicht mehr zu erreichen, da diese nicht die benötigte Mobilität liefern, so dass in diesem Bereich die eingebetteten Systeme Einzug erhalten haben.

Gerade durch die Mobilität, die eine Eigenschaft von vielen eingebetteten System ist, ergeben sich neue Anforderungen an das Gesamtsystem. So werden viele eingebettete Systeme nur durch eine geringe Stromquelle, wie einem Batteriespeicher, betrieben. Des Weiteren wünschen die Benutzer, das die Geräte möglichst klein und leicht sind, so dass diese nicht zur Last fallen. Um einen geringen Energieverbrauch, sowie ein geringes Gewicht bei einer geringen Größe der verbauten Komponenten zu erreichen, müssen Hard- und Software möglichst stark optimiert werden. Im Allgemeinen gilt, dass leistungsschwache Prozessoren weniger Energie verbrauchen, als entsprechend schnellere Prozessoren. Von daher muss, bezogen auf die Anforderungen des zu entwickelnden Systems, ein Trade-off zwischen Leistung und Energieverbrauch, bezogen auf die Kompaktheit des Systems, gefunden werden.

Viele der eingebetteten Systeme müssen des Weiteren unter realen Echtzeitbedingungen arbeiten. Dabei wird eine generelle Einteilung in *weiche* und *harte* Echtzeitsysteme vorgenommen. Erstere gehören zu den Systemen, bei denen eine Überschreitung der vorgegebenen Zeitbedingungen unkritisch, aber störend sein kann. So ist es für einen Benutzer zwar unzumutbar, wenn seine Videowiedergabe wiederholte Aussetzer hat, doch kommt durch diesen Umstand niemand zu Schaden.

Zu den harten Echtzeitsystemen gehören dagegen Steuergeräte aus Autos, Flugzeugen, Kraftwerken oder auch anderen industriellen Überwachungsmechanismen. Ein zu spät einsetzendes ABS System oder ein nicht reagierender Kühlwasserprüfer kann zu lokalen oder auch globalen Katastrophen führen. Um dies zu vermeiden, müssen diese Systeme zwingend ihre vorgegebenen Zeitschranken einhalten.

Gerade für die Entwicklung und Evaluation von harten Echtzeitsystemen ist es daher erforderlich neue Verfahren zum Design von Hard- und Softwaresystemen zu entwickeln. Während bei der herkömmlichen Softwareentwicklung für Desktop-PCs für den Großteil aller Anwendungsgebiete die Optimierung bezüglich der ACET (*average-case execution time* = durchschnittliche Laufzeit) im Mittelpunkt des Interesses steht, muss für sicherheitskritische Anwendungen, wie z.B. im Automobil- und Luftfahrtbereich, eine sichere *obere* Laufzeitschranke angegeben werden.

Daher ist der Fokus bei der Entwicklung von eingebetteten Systemen auf die Berechnung und Optimierung des Zeitverhaltens, bzw. der oberen Laufzeitschranke zu setzen. Diese wird als WCET (*worst-case execution time*) bezeichnet. Da die genaue Berechnung dieser oberen Laufzeitschranke im Allgemeinen jedoch zu komplex ist, kann die WCET lediglich durch eine approximative Lösung abgeschätzt werden. In Kapitel 2 dieser Arbeit wird dabei ein Überblick über die weit verbreiteten Methoden gegeben, die zur Abschätzung der WCET eingesetzt werden.

Ein fortgeschrittenes Verfahren zur approximativen WCET-Bestimmung ist die statische WCET-Analyse. Dabei wird das Programm nicht ausgeführt, sondern seine Eigenschaften durch statische Programmanalysen bestimmt. Dieses Verfahren extrahiert die meisten Informationen automatisch aus dem Programm. Eine Ausnahme bildet die Spezifikation von Schleifenausführungshäufigkeiten, die für die Anwendung dieses Verfahrens bereits im Vorfeld berechnet werden müssen.

Diese Informationen werden aktuell häufig durch manuelle Analysen ermittelt. Die dort gefundenen

Schleifengrenzen werden in der Folge manuell im Quellcode annotiert, damit diese bei der statischen WCET-Analyse zur Verfügung stehen. Dieser Vorgang ist ebenso zeitintensiv, wie fehleranfällig.

Aus dieser Motivation heraus, soll in dieser Arbeit ein automatisiertes Verfahren entwickelt werden, das in der Lage ist, für ein gegebenes Programm selbstständig die Iterationsgrenzen für die Schleifen eines Programms zu berechnen, damit mit dieser Lösung automatisch die WCET approximiert und schließlich optimiert werden kann.

1.2 Verwandte Arbeiten

In diesem Abschnitt soll zunächst ein Überblick über die bisher veröffentlichten Forschungsarbeiten gegeben werden, die sich mit dem Thema der automatischen Schleifenanalyse beschäftigt haben.

Einen der ersten Ansätze haben Christopher Healy et al. bereits 1998 in [HSRW98] vorgestellt. Dort wurde ein Verfahren beschrieben, das auf Assembler-Ebene die Schleifeniterationsgrenzen eines gegebenen Programms bestimmen kann. Dabei werden ausschließlich die Schleifen eines Programms analysiert, ohne die restlichen Anweisungen auszuwerten. Auf diese Weise ist es mit dem vorgestellten Verfahren nur begrenzt möglich, Schleifen zu analysieren, deren Iterationshäufigkeit von Parametern abhängt. Abhilfe schafft die dort beschriebene Interaktion mit dem Benutzer, der ggf. dazu aufgefordert wird, den möglichen Wertebereich der in der Schleife involvierten Variablen manuell zu bestimmen und an die Analyse zu übergeben. Die eigentlichen Schleifeniterationsgrenzen bestimmt das Verfahren mit Hilfe einer statischen Berechnung.

Dieses Verfahren wurde 2006 in der Magisterarbeit von Martin Kirner [Kir06] so modifiziert, dass es im Gegensatz zur ursprünglichen Version die Analyse direkt auf dem C-Quellcode durchführt. Auch diese Analyse hat den Nachteil, dass der Benutzer zur Analyse von komplexeren Schleifen die möglichen Werte der abhängigen Variablen manuell in den Quellcode annotieren muss, damit die Analyse der Schleifengrenzen ein Ergebnis liefern kann.

Einen alternativen Ansatz zur automatischen Analyse von Schleifeniterationsgrenzen haben Andreas Ermedahl und Jan Gustafsson an der Mälardalen Universität Schweden in [EG97] vorgestellt. In dieser Publikation wurde eine Datenflussanalyse auf Basis der *Abstrakten Interpretation* verwendet, um die möglichen Werte der Variablen innerhalb des zu analysierenden Programms zu bestimmen, damit diese für die Berechnung der Schleifengrenzen zur Verfügung stehen. Mit Hilfe der Abstrakten Interpretation ist es möglich für jede Variable dessen mögliche Werte, abhängig vom Programmpunkt, zu bestimmen (siehe Kapitel 4). Eine solch komplexe Analyse ist nur dadurch möglich, dass die Berechnung auf einer approximierten Version des Programms durchgeführt wird. Diese Approximation wird dadurch erreicht, dass die *konkrete Domäne*, die durch die Semantik des Programms gegeben ist, durch eine *abstraktere Domäne* ersetzt wird (siehe Abschnitt 4.2.3). So wurde in dieser Arbeit die Domäne der Intervalldarstellung genutzt, wobei jeder Wert des Programms durch eine Vereinigung verschiedener Intervalle repräsentiert wird. Auf Basis der Werte, die durch die Abstrakte Interpretation berechnet werden, werden die Schleifen des Programms nach und nach ausgerollt,

um so die Iterationshäufigkeiten zu bestimmen.

Dieses Verfahren wurde in mehreren Publikationen erweitert. So hat Björn Lisper 2003 an der Mälardalen Universität Schweden in [Lis03] Polytope als Abstrakte Domänen ergänzt. Im Gegensatz zur Intervalldarstellung, wird für jede Variable ein Ungleichungssystem angegeben, das ein Polytop repräsentiert, dessen Lösung die möglichen Werte einer Variable darstellt. Dieses Verfahren hat sich als exakter, aber auch extrem viel rechenaufwändiger herausgestellt.

In weiteren Arbeiten, wie z.B. [ESG+07] und [GESL07] wurde das eigentlich vorgestellte Verfahren der Abstrakten Interpretation durch weitere Abstrakte Domänen und zusätzlichen Tools, die die Geschwindigkeit der Analyse beschleunigen sollten, erweitert. In diesen Werken wurde beispielsweise eine alternative Version der Abstraktion Interpretation mit dem Namen Abstrakte Ausführung (*abstrakt execution*) vorgestellt, welches die Präzision der Schleifengrenzen verbessern sollte. Des Weiteren wurde das Verfahren durch ein *Program Slicing* erweitert, welches das zu analysierende Programm vor der Analyse der Abstrakten Ausführung auf die wesentlichen Anweisungen reduziert, die für die Berechnung der Schleifengrenzen relevant sind. Alle Ergebnisse dieser Arbeiten wurden in das Programm SWEET (*SWEdish Execution time Tool*) der Mälardalen Universität integriert, mit dessen Hilfe so die approximative Berechnung der WCET ermöglicht wird.

Der Ansatz der Mälardalen Universität soll Ausgangspunkt dieser Arbeit sein. Dabei soll ebenfalls eine modifizierte Version der Abstrakten Interpretation entwickelt werden, um die Werte der Variablen auf Intervallebene zu approximieren. Auch hier sollen die Schleifengrenzen bereits während der Analyse der Abstrakten Interpretation berechnet werden, so dass kein weiterer Schritt nach Abschluss der Analyse erforderlich ist. Im Gegensatz zum bereits vorgestellten Verfahren der Mälardalen Universität soll die modifizierte Abstrakte Interpretation dieser Arbeit jedoch durch ein statisches Schleifenanalyseverfahren ergänzt werden, das mit Hilfe von Polytop-Berechnungen die Iterationsgrenzen von weniger komplexen Schleifen in deutlich reduzierter Laufzeit bestimmen kann. Diese statische Analyse soll, im Gegensatz zur Arbeit aus [HSRW98], ebenfalls Iterationsgrenzen von Schleifen bestimmen können, die ggf. durch Parameter bedingt werden, indem auf die laufenden Ergebnisse der Abstrakten Interpretation zurückgegriffen wird. So sollen beide Analyseverfahren miteinander kombiniert werden, wobei beide Verfahren voneinander profitieren sollen. Die Abstrakte Interpretation ermittelt die benötigten (approximierten) Werte der Variablen, die das statische Verfahren zur Berechnung der Schleifengrenzen benötigt. Die statische Schleifenanalyse versucht im Anschluss mit diesen Werten die Iterationsgrenzen in einem Bruchteil der Analysezeit der Abstrakten Interpretation zu bestimmen, um diese Ergebnisse wiederum an die Abstrakte Interpretation zurück zu geben, damit diese das weitere Programm analysieren kann. So wird die Stärke beider Verfahren miteinander kombiniert.

Der Unterschied bei der Einbindung der Polytope in dieser Arbeit und der Publikation aus [Lis03] besteht darin, dass in dem Werk aus 2003 die Polytope als Abstrakte Domäne eingesetzt wurden. Durch diese Umsetzung sollte die Präzision der Analyse verbessert werden. In dem Verfahren, das in dieser Arbeit vorgestellt werden soll, wird für die Approximation der Werte die Intervalldarstellung verwendet. Die Polytop-Berechnungen sollen vielmehr dazu genutzt werden, Schleifengren-

zen statisch auszuwerten, um die Laufzeit der Analyse stark zu reduzieren. Dazu wird die *Polyhedral Library* (kurz Polylib) eingesetzt, die mit Hilfe von Erhard Polynomen [VSB+04] die Anzahl der möglicherweise erreichten Iterationspunkte berechnet.

1.3 Ziele der Arbeit

Im Rahmen der Forschungsarbeit des Informatik Lehrstuhls 12 der Technischen Universität Dortmund ist mit dem *WCC* ein WCET-optimierender ANSI-C-Compiler für den Infineon TriCore Prozessor entstanden (siehe Kapitel 3). Der WCC ermittelt durch Anbindung des WCET-Analysetools *aiT* von AbsInt vollautomatisch die obere Laufzeitschranke eines Programms, damit diese für die weiteren Optimierungen zur Verfügung steht.

Da der WCC allerdings in seiner aktuellen Version nur eine sehr eingeschränkte Schleifenanalyse integriert hat, soll diese durch die Analyse der vorliegenden Diplomarbeit ersetzt werden. Die aktuelle Version der Schleifenanalyse durchsucht den Quellcode nach Schleifen und versucht diese auf statische Weise auszuwerten, ohne dass dabei Informationen zu den möglichen Werten der enthaltenen Variablen gesammelt werden. Daher können nur Iterationsgrenzen von Schleifen bestimmt werden, die durch konstante Werte eingeschränkt sind.

Damit die WCET-Analyse von aiT durchgeführt werden kann, muss für alle Schleifen die obere und untere Iterationsgrenze angegeben werden. Da die aktuell integrierte Schleifenanalyse jedoch nicht in der Lage ist, für einen Großteil der Programme die Iterationsgrenzen der Schleifen zu bestimmen, müssen diese zur Zeit manuell ermittelt und in den Quellcode annotiert werden. Wie bereits in der Motivation aus Abschnitt 1.1 erwähnt, ist dieser Vorgang ebenso zeitintensiv wie fehleranfällig und erlaubt keine vollautomatische WCET-Analyse von beliebigen Programmen.

Die Schleifenanalyse, die in dieser Arbeit entwickelt werden soll, analysiert das Programm direkt auf Quellcode-Ebene, bzw. auf der Zwischendarstellung ICD-C (siehe Abschnitt 3.1.1). Wie bereits im letzten Abschnitt beschrieben, gibt es Ansätze, welche die Schleifengrenzen auf einer assemblernahen Repräsentation des Quellcodes bestimmen. Bei der Durchführung der Codeselektion, die für die Umwandlung des Quellcodes in den Assemblercode verantwortlich ist, verliert der Compiler jedoch wertvolle Informationen bezüglich des Kontrollflusses, wodurch eine Schleifenanalyse erschwert wird. Daher soll in dieser Arbeit eine automatische Analyse auf der quellcodenahen Zwischendarstellung ICD-C entwickelt werden.

Um in der Analyse, die in dieser Arbeit entwickelt werden soll, auch Schleifengrenzen von Schleifen zu bestimmen, die von variablen Werten bedingt werden, soll eine statische Auswertung der möglichen Werte der Variablen durchgeführt werden, um entsprechend komplexe parameterabhängige Schleifen analysieren zu können. Für diese Programmanalyse hat sich die Abstrakte Interpretation als geeignetes Verfahren herausgestellt. Mit Hilfe einer modifizierten Version der Abstrakten Interpretation soll es möglich sein, Zustände für jeden Punkt eines Programms zu bestimmen, in dem alle möglichen Werte von Variablen beinhaltet sind. Dabei soll die Abstrakte Interpretati-

on so modifiziert werden, dass bereits während der Analyse des Programms die Schleifengrenzen bestimmt werden.

Für komplexere Anwendungen wird die Analyse der Abstrakten Interpretation sehr rechenaufwändig und benötigt somit sehr viel Zeit. Das kommt daher, dass die Abstrakte Interpretation die Schleifen eines Programms nahezu iterativ auswertet, was insbesondere bei der Analyse von häufig iterierenden Schleifen zu zeitintensiven Berechnungen führt. Daher soll die Analyse der Abstrakten Interpretation durch ein statisches Schleifenanalyseverfahren ergänzt werden, das unter einigen Vorbedingungen die Schleifengrenzen, sowie die neuen Werte der modifizierten Variablen, durch eine statische Berechnung bestimmt. Dafür soll die Bedingung der Schleife in ein Ungleichungssystem übertragen werden, welches ein passendes Polytop darstellt. Mit Hilfe von Erhard-Polynomen und der *Polyhedral Library* ist es möglich, die Anzahl der Punkte effizient zu bestimmen, die durch das Polytop eingeschlossen werden. Diese Punkte repräsentieren mögliche Iterationspunkte der Schleife, so dass von der errechneten Anzahl auf die Schleifeniterationsgrenzen geschlossen werden kann. Dieses schnelle Verfahren soll direkt in die Analyse der Abstrakten Interpretation eingebettet und für bestimmte Klassen an Schleifen anstelle der Abstrakten Interpretation verwendet werden. Falls die statische Schleifenanalyse auf Grund einer zu großen Komplexität der Schleife nicht durchgeführt werden kann, wird die Analyse wie ursprünglich vorgesehen durch die Abstrakte Interpretation vorgenommen.

Des Weiteren soll die Transformation von Bedingungen in den Raum der Polytope genutzt werden, um Überapproximationen, die während der Analyse der Abstrakten Interpretation auftreten können, effizient zu minimieren. So soll im Verlauf dieser Arbeit ein Verfahren entwickelt werden, das bei bedingten Verzweigungen die Bedingung in ein Polytop überträgt. Im Anschluss können die Werte der Variablen, die in der Bedingung enthalten sind, dadurch eingeschränkt werden, dass Ihre minimal und maximal möglichen Werte an den Eckpunkten der konvexen Hülle des Polytops liegen. Diese Punkte sind ebenfalls effizient mit Hilfe der eingesetzten Polyhedral Library berechenbar.

Die auf dieser Basis entwickelten Schleifenanalyseverfahren sollen im Folgenden als optionales Analyseverfahren in den WCC integriert werden. Damit vervollständigen diese das bestehende WCC-Framework und ermöglichen somit eine vollkommen automatische WCET-Analyse und -Optimierung von beliebigen Programmen.

Zusätzlich zur integrierten Version, soll ein Stand-Alone Tool entwickelt werden, das es ermöglicht, eine separate Schleifenanalyse ohne einen vollständigen Kompiliervorgang durchzuführen.

1.4 Aufbau der Arbeit

Zum Abschluss dieses Kapitels soll in diesem Abschnitt ein Überblick über den Aufbau dieser Diplomarbeit gegeben werden.

In *Kapitel 2* wird zunächst eine genauere Definition der WCET (*worst-case execution time*) gege-

ben. Zudem werden dort diverse Verfahren vorgestellt, mit denen die WCET approximativ bestimmt werden kann. Abgeschlossen wird das Kapitel mit einer genaueren Vorstellung des Tools aiT der Firma AbsInt, mit dessen Hilfe der WCC die WCET approximativ berechnet.

In *Kapitel 3* wird als nächstes der Aufbau des WCC detaillierter beschrieben. Dazu wird anfangs ein Überblick über den internen Aufbau gegeben, der durch die Einzelbeschreibungen der Komponenten, die für die zu entwickelnde Schleifenanalyse wichtig sind, ergänzt wird. Abgeschlossen wird das dritte Kapitel durch eine Beschreibung der geplanten Integration der Schleifenanalyse in den WCC.

Kapitel 4 beschreibt die statische Programmanalyse, die für die approximative Berechnung der Variablenwerte eingesetzt werden soll. Als spezielles Verfahren soll dabei die Abstrakte Interpretation genutzt werden, deren Aufbau nach einer allgemeinen Einführung beschrieben wird. Da die Konzepte der ursprünglichen Abstrakten Interpretation jedoch einige Nachteile aufgewiesen haben, soll in dieser Arbeit eine modifizierte Version der Abstrakten Interpretation eingesetzt werden, die im Anschluss beschrieben wird.

Für einige Berechnungen innerhalb der Abstrakten Interpretation, wie das Einschränken von Variablenwerten um Überapproximationen zu verhindern, sollen in dieser Diplomarbeit Polytop-Berechnungen eingesetzt werden. *Kapitel 5* soll diese einleitend beschreiben, um im späteren Verlauf mit der Polylib eine Bibliothek vorzustellen, die für die geplante Analyse eingesetzt werden soll.

Auf Basis der Polytop-Berechnungen aus Kapitel 5 soll eine statische Schleifenanalyse entwickelt werden, die versuchen soll, die Laufzeit der Schleifenanalyse, die auf der modifizierten Abstrakten Interpretation basiert, zu verbessern. Deren Aufbau wird zusammen mit einem Beispiel in *Kapitel 6* beschrieben.

In *Kapitel 7* soll im Folgenden zu Anfang der Ablauf der zu entwickelnden Analyse beschrieben werden. Dieser wird durch die Beschreibung des konzeptionellen Aufbaus, sowie der eingesetzten Datenstrukturen ergänzt. Diese vorher wenig spezifischen Beschreibungen werden im darauf folgenden Abschnitt durch die Vorstellung der wichtigsten Klassen ergänzt. Abgeschlossen wird das Kapitel durch eine Beschreibung der Integration der automatischen Schleifenanalyse in den WCC und einer Beschreibung des Stand-Alone Tools *irloopanalyzer*.

Kapitel 8 enthält eine ausführliche Auswertung der entwickelten Analyseverfahren. Diese Auswertung wurde in mehreren Stufen auf unterschiedlichen Benchmarks durchgeführt.

Abgeschlossen wird die vorliegende Arbeit in *Kapitel 9* mit einer Zusammenfassung. In dieser wird u.a. ein Ausblick gegeben, welcher mögliche Erweiterungen für die entwickelte Schleifenanalyse beschreibt.

Kapitel 2

WCET - Analyse

Eingebettete Systeme werden häufig unter realen Echtzeitbedingungen eingesetzt. Dabei wird zwischen *weichen* und *harten* Echtzeitsystemen unterschieden (siehe Abschnitt 1.1). Während bei den weichen Echtzeitsystemen das Überschreiten einer gegebenen Zeitschranke störend wirken kann, kann eine solche Überschreitung bei harten Echtzeitsystemen bereits zu lokalen oder auch globalen Katastrophen führen.

Daher kann bei der Validierung von harten Echtzeitsystemen nicht auf die durchschnittliche Laufzeit (*average-case execution time*) zurückgegriffen werden, so dass dort die Kenngröße der WCET (*worst-case execution time*) benötigt wird. In der Literatur wird bei der WCET zwischen der $WCET_{est}$ und der $WCET_{real}$ unterschieden. Dabei steht die $WCET_{est}$ für die approximierte WCET, die durch eine Analyse bestimmt wurde. Die $WCET_{real}$ steht hingegen für die reale, maximale Laufzeit eines Programms. Ziel einer WCET-Analyse sollte es also sein, die $WCET_{real}$ möglichst genau abzuschätzen (siehe auch Abschnitt 2.1). [Mar07].

Dieses Kapitel soll im Folgenden einen Einblick geben, wie die $WCET_{est}$ berechnet wird. Dazu wird in Abschnitt 2.1 zuerst auf die Faktoren eingegangen, die für eine möglichst genaue WCET-Abschätzung relevant sind. Im Anschluss werden mit der statischen, der dynamischen und einem hybriden Verfahren drei mögliche WCET-Analysemethoden vorgestellt, wobei der Schwerpunkt auf das statische Verfahren gelegt wird. Des Weiteren wird mit aiT ein kommerzielles Tool vorgestellt, in dem eine statische WCET-Analyse umgesetzt wurde. Abschließend wird ein Ausblick gegeben, der den Zusammenhang der beschriebenen Verfahren mit dem Rest der Arbeit herstellen soll.

2.1 Güte der WCET-Analyse

Alan M. Turing bewies bereits 1936, dass das Halteproblem nicht entscheidbar ist. Dazu zeigte er, dass nicht immer beweisbar ist, dass eine nicht-terminierende Turingmaschine nicht hält. [Weg05]

Genau genommen hat er damit gezeigt, dass für ein beliebiges Programm $\mathcal{P}$ nicht entscheidbar ist,

ob dieses terminieren wird. Bezogen auf die Berechnung der WCET besteht nun folgender Zusammenhang: Würde es gelingen, für jedes Programm $\mathcal{P}$ die $WCET_{real}$ in endlicher Zeit zu berechnen, so könnte entschieden werden, ob das Programm terminiert. Da dies die These von Alan M. Turing widerlegen würde, ist eine genaue Berechnung der $WCET_{real}$ nicht möglich, so dass es lediglich approximative Lösungen geben wird.

Anhand der Anforderungen an die $WCET_{est}$ können nun die folgenden Bedingungen an die WCET-Abschätzung gestellt werden (nach [Kir00]):

$$WCET_{est} \geq WCET_{real} \tag{2.1}$$

Die Gleichung 2.1 besagt, dass die abgeschätzte WCET mindestens genau so groß wie die reale sein muss. Diese Bedingung folgt aus den sicherheitskritischen Anwendungen. Würde die approximative WCET kleiner als die reale abgeschätzt werden, so könnte es zu oben genannten Katastrophen führen.

$$WCET_{est} - WCET_{real} \rightarrow \min \tag{2.2}$$

Die Gleichung 2.2 (nach [Kir00]) besagt, dass die Abweichung zwischen der approximativ berechneten ($WCET_{est}$) und der realen ($WCET_{real}$) WCET möglichst minimal sein sollte. Diese Bedingung hat ihren Ursprung vor allem in der Wirtschaftlichkeit. Meistens wird die zu Grunde liegende Hardware der eingebetteten Systeme möglichst minimal gewählt, um z.B. Stromverbrauch und Wirtschaftlichkeit zu optimieren. Ist die Abweichung zu stark, so wird sich die Auswahl der Hardware sehr stark von der optimalen Wahl unterscheiden.

Ziel der im Folgenden vorgestellten Verfahren sollte also stets sein, eine sichere und möglichst genaue Lösung zu liefern.

2.2 Statische WCET-Analyse

Die statische WCET-Analyse versucht anhand des Quellcodes, bzw. einer Transformation in Form eines Kontrollflussgraphen (siehe Abschnitt 2.2.1), die $WCET_{est}$ zu bestimmen. Als Ergebnis wird eine sichere obere Grenze angegeben, die bei einem korrekten Modell die Gleichung 2.1 erfüllt.

Das Problem der statischen Analyse ist, dass die Qualität des Ergebnisses (also die Größe der Abweichung aus Gleichung 2.2) von der Güte und Menge der Zusatzinformationen abhängt.

Um detaillierter auf die Berechnung einzugehen, wird vorerst der Kontrollflussgraph erläutert.

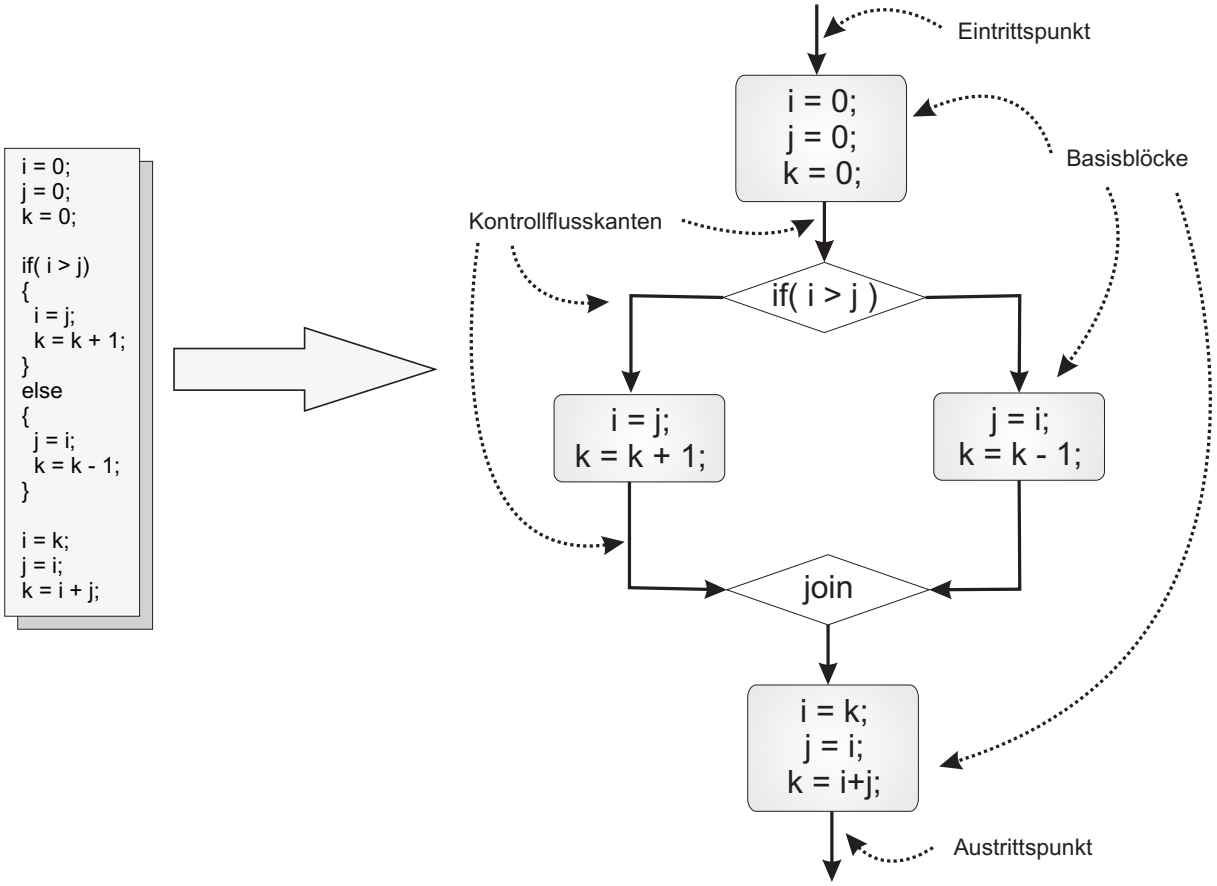


Abbildung 2.1: *Beispiel eines Kontrollflussgraphen*

2.2.1 Der Kontrollflussgraph

Der Kontrollflussgraph (Control Flow Graph = CFG) ist ein gerichteter Graph, der die Ausführungsreihenfolge eines Programms $\mathcal{P}$ repräsentiert.

Ein Beispiel für einen Kontrollflussgraphen ist in Abbildung 2.1 zu sehen.

Definition 1 (Basisblock) *Ein Basisblock BB_i ist eine maximale Sequenz von Befehlen, die immer über denselben Start- und Endbefehl als Sequenz ausgeführt werden [Kir00].*

Knoten des Kontrollflussgraphen sind die Basisblöcke. Diese bestehen, wie in Definition 1 beschrieben, aus einer maximalen Anzahl an Befehlen, die immer zusammen ausgeführt werden, falls Sie im Programmfluss erreicht werden.

Eine gerichtete Kante e_{n_1,n_2} wird genau dann in den Graphen eingefügt, wenn es einen *möglichen* Kontrollfluss im ursprünglichen Programm gibt, der direkt von Basisblock n_1 in den Basisblock n_2 übergeht.

Definition 2 (Kontrollflusspfad) *Ein Kontrollflusspfad ist eine Sequenz von Basisblöcken BB_i, die gemäß des Kontrollflusses (vorgegeben durch die gerichteten Kanten e_{n_j,n_k}) möglich ist.*

Falls es in dem Programm $\mathcal{P}$ ungebundene Schleifen gibt, also Schleifen, die nicht durch eine maximale Iterationsanzahl beschränkt sind, so gibt es unendlich viele Kontrollflusspfade.

Definition 3 ($CFP(\mathcal{P})$) *Die Menge aller möglichen Kontrollflusspfade eines Programms $\mathcal{P}$ heißt CFP(P) (Control Flow Paths) [Kir03].*

In der Menge $CFP(\mathcal{P})$ befinden sich alle möglichen Kontrollflusspfade. Dazu gehören auch die Pfade, die laut der Programmsemantik niemals in einem bestimmten Ausführungskontext erreicht werden (infeasible paths). Daher wurde die Menge $CFP_{opt}(\mathcal{P})$ eingeführt.

Definition 4 ($CFP_{opt}(\mathcal{P})$) *Die Menge $CFP_{opt}(\mathcal{P})$ ist die Untermenge von $CFP(\mathcal{P})$, die nur die ausführbaren Kontrollflusspfade enthält [Kir03].*

Die Menge $CFP_{opt}(\mathcal{P})$ ist allerdings nicht effizient berechenbar.

Definition 5 ($CFP_{WCET,opt}(\mathcal{P})$) *$CFP_{WCET,opt}(\mathcal{P})$ ist der Kontrollflusspfad, der die optimale WCET liefert [Kir03].*

Ziel der statischen WCET-Analyse ist es also, den Kontrollflusspfad $CFP_{WCET,opt}(\mathcal{P})$ zu finden. Zusammenfassend gilt:

$$CFP_{WCET,opt}(\mathcal{P}) \in CFP_{opt}(\mathcal{P}) \subseteq CFP(\mathcal{P}) \tag{2.3}$$

2.2.2 Die Berechnung

Die WCET kann nun anhand des Kontrollflussgraphen und eines Zeitmodells des Zielsystems über ein lineares Gleichungssystem bestimmt werden. Dazu wurden mehrere Ansätze entwickelt, wie z.B. die *Implicit Path Enumeration Technique*, die als Beispiel etwas näher erläutert werden soll (siehe [Wol02]).

Für die Berechnung werden folgende zusätzlichen Parameter benötigt, die bisher nicht eingeführt worden sind:

- x_i ist die Ausführungshäufigkeit des Basisblocks BB_i

- c_i sind die Kosten, die bei einer Ausführung des Basisblocks BB_i entstehen. Als Kosten wird hierbei die Ausführungszeit auf einem Prozessor ohne nebenläufige Prozesse, Kontextwechsel etc. verstanden.

Die Kosten eines Basisblocks ergeben sich dabei aus den Kosten seiner einzelnen Befehle. Diese können je nach Komplexität des Berechnungsmodells noch von dessen Vorgänger und Nachfolger abhängen. Auf diese Weise kann z.B. das Verhalten der Pipeline des Prozessors simuliert werden.

Die gesuchten Gesamtkosten des Programms $\mathcal{P}$ (bzw. die $WCET_{est}$) ergeben sich aus der Summe der Anzahl der Iterationen pro Basisblock, multipliziert mit den Kosten eines Basisblocks. Dies ist in Gleichung 2.4 abgebildet, wobei N die Menge aller Basisblöcke beschreibt:

$$C(\mathcal{P}) = \sum_{i}^{N} c_i * x_i \tag{2.4}$$

Die *Implicit Path Enumeration Technique* berücksichtigt in dieser Version keinerlei Kontexte (siehe Abschnitt 2.5.1), wodurch die Genauigkeit der ermittelten $WCET_{est}$ unter Umständen nicht optimal ist. Eine etwas komplexere statische WCET-Analyse ist in Abschnitt 2.5.2 beschrieben. Dort wird die Berechnung vorgestellt, durch die die Software aiT von AbsInt die Approximation der $WCET_{est}$ bestimmt.

Das Gleichungssystem aus 2.4 wird nun durch strukturelle Bedingungen erweitert, die den Kontrollfluss repräsentieren. So muss die Ausführungshäufigkeit eines Blocks gleich der Summe der Ausführungen seiner Vorgänger sein. Gleiches gilt für die Nachfolger. Deren Summe muss identisch mit der Ausführungshäufigkeit des vorangegangenen Basisblocks sein. Formal bedeutet dies:

$$\forall i : \sum_{BB} x_{inflow} = x_i = \sum_{BB} x_{outflow} \tag{2.5}$$

Die Vereinigung aus der Gleichung 2.4 und dem Gleichungssystem 2.5 erzeugt ein ganzzahliges lineares Optimierungsproblem, das es zu lösen gilt. Es sei an dieser Stelle darauf hingewiesen, dass auch das Lösen eines solchen ILP-Systems NP-vollständig ist. Wie in [Wol02] angegeben, terminiert das System für die meisten Fälle allerdings bereits in wenigen Minuten.

Damit die statische WCET-Analyse ein Ergebnis erzielt, muss für alle Schleifen eine obere Iterationsgrenze angegeben sein. Ansonsten gibt es Kontrollflusspfade, die unendlich lang sind und in Folge dessen eine unendlich große WCET als Ergebnis liefern. Damit daher die Berechnung der WCET im WCC in Zukunft vollautomatisch durchgeführt werden kann, soll in dieser Arbeit eine rechnergestützte Schleifenanalyse entwickelt werden.

In den folgenden Kapiteln 2.3 und 2.4 soll noch kurz auf weitere Analysemethoden eingegangen werden. Der Rest dieser Arbeit wird sich allerdings auf das statische Verfahren beziehen.

```
1  for ( i = 0; i < j; i++)
   {
     if(i > k)
     {
5      z = z + 1;
     }
     else
     {
       z = z * 9 / 10;
10   }
   }
```

Abbildung 2.2: *Beispiel für die Abhängigkeit zwischen Laufzeit und Eingabeparametern*

2.3 Dynamische WCET-Analyse (Messung)

Im Gegensatz zur statischen WCET Programmanalyse versucht die dynamische Variante ein Ergebnis durch Simulation zu erzielen. Auf Grund von möglicherweise unbekannten Eingabeparametern können sich allerdings bei unterschiedlichen Parametern unterschiedliche Laufzeiten ergeben. So hängt z.B., wie in Abbildung 2.2 zu sehen, die Laufzeit stark von den Parametern *j* und *k* ab. Die Variable *j* beeinflusst in diesem Beispiel die Schleifeniterationsanzahl und die Variable *k* entscheidet, ob die teure oder die günstige Operation der Verzweigung ausgeführt werden muss (teuer bedeutet hierbei, dass für die komplexere Operation im Else-Teil deutlich mehr CPU-Zyklen benötigt werden, als für das Inkrementieren im Then-Teil).

Um durch die Simulation ein repräsentatives Ergebnis zu erzielen, müssen möglichst viele Eingabeparameterkombinationen getestet werden. Allerdings wird pro Simulationsdurchlauf jeweils nur ein einzelner Kontrollflusspfad simuliert, wodurch auch diese Analysemethode recht zeitaufwendig sein kann. Auf Grund der Simulation einzelner Kontrollflusspfade gilt für die dynamische WCET-Analyse:

$$WCET_{est} \leq WCET_{real} \tag{2.6}$$

Dies steht in Widerspruch zur Gleichung 2.1 aus Abschnitt 2.1, wodurch keine sichere Approximation garantiert ist.

Da es sehr schwierig, bzw. extrem zeitaufwendig ist, für die statische WCET-Analyse alle nötigen Parameter, sowie ein mathematisches Modell für das Zielsystem zu finden, wird die dynamische WCET-Analyse trotz aller Nachteile auch heute noch in der Industrie eingesetzt [Wol02].

Das Hauptproblem der dynamischen WCET-Analyse ist also das Erzeugen von repräsentativen Testpattern. Neben randomisierten Verfahren werden auch häufig evolutionäre Algorithmen eingesetzt, um möglichst viele Testfälle abzudecken.

2.4 Hybride WCET-Analysen

Zusätzlich zu den bisher vorgestellten Verfahren gibt es noch hybride Ansätze. Diese versuchen die positiven Aspekte beider Analysetypen miteinander zu verbinden.

So geht das Verfahren *SYMTA* (SYMbolic Timing Analysis) den Weg, einzelne zusammenhängende Blöcke zu simulieren und diese Ergebnisse für komplexere Verzweigungen in ein Gleichungssystem einfließen zu lassen. Dafür wird der bisher vorgestellte Kontrollflussgraph um sogenannte *Single Feasible Pathes* (SFP) und *Multiple Feasible Pathes* (MFP) erweitert. Die SFP haben keine Verzweigungen und sind genau die Programmteile, deren Kosten durch Simulation ermittelt werden. Die MFP setzen sich schließlich aus mehreren SFP oder auch MFP zusammen und werden in Folge dessen auf mathematische Weise gelöst [YE95].

Die Grundidee ist, dass vor allem durch das Simulieren möglichst großer Blöcke das Verhalten von Cachespeichern, oder auch CPU-Pipelines, mit in das Ergebnis einfließen können. In einem rein mathematischem Modell ist dies nur unter hohem Aufwand möglich.

2.5 WCET-Analyse in der Praxis – aiT

Die Firma AbsInt [aiT08] hat mit aiT einen statischen WCET-Analyzer auf den Markt gebracht, der in der Industrie Anklang gefunden hat. So gehören laut Informationen der firmeneigenen Website namenhafte Unternehmen, wie z.B. Airbus Industries, Bosch, BMW, Nokia oder auch IBM, zum Klientel von AbsInt.

Auch der Informatik-Lehrstuhl 12 der Technischen Universität Dortmund hat im Rahmen seiner Forschungsarbeit den WCET-Analyzer von AbsInt in Ihren WCET-fähigen C-Compiler (*WCC*) integriert (siehe Kapitel 3).

2.5.1 Der Aufbau der Analyse

Das Tool aiT ist in mehrere Phasen eingeteilt, die sukzessiv zum Ergebnis der WCET beitragen (siehe Abbildung 2.3) [Fer04]:

- Als Eingabe erwartet aiT eine kompilierte Binärdatei des zu analysierenden Programms, sowie eine AIS-Datei, in der benutzerdefinierte Annotationen zu finden sind. In dieser kann (bzw. muss) der Benutzer diverse verpflichtende, oder auch optionale Angaben machen, die für die WCET-Analyse notwendig sind. Zu den verpflichtenden Angaben zählen vor allem obere Schleifeniterationsgrenzen und maximale Rekursionstiefen.

  Aus diesen Angaben rekonstruiert der *CFG Builder* einen annotierten Kontrollflussgraphen. Dieser wird in einer CRL2-Datei abgelegt.

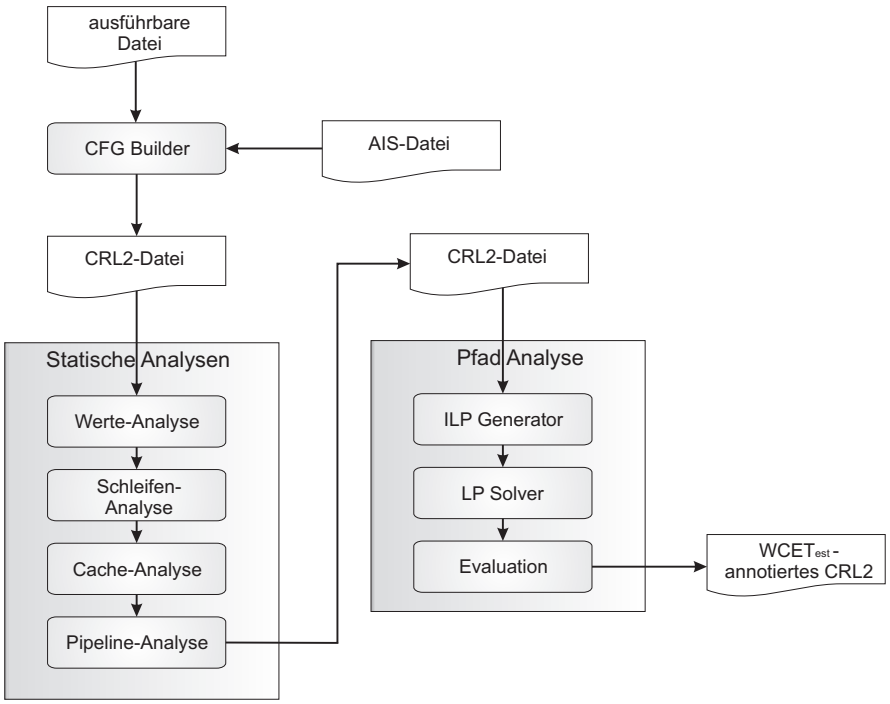


Abbildung 2.3: *Aufbau des aiT WCET-Analyzers der Firma AbsInt*

- Als nächstes folgen einige statische Analysen, die auf der Abstrakten Interpretation (siehe Kapitel 4) aufgebaut sind.

  – Die *Werte-Analyse* untersucht für alle Zugriffe die möglichen Werte von Registern, die diese innerhalb des Programms annehmen können. Dabei werden Intervalle als Abstrakte Domäne benutzt.

  – Die *Schleifen-Analyse* versucht auf Grundlage der vorher ermittelten Werte die Schleifengrenzen aller enthaltenen Schleifen zu bestimmen, falls diese vom Benutzer nicht in der AIS-Datei angegeben worden sind. Nach eigenen Angaben von AbsInt funktioniert dies jedoch nur bei sehr einfachen Schleifen, für die es konstante Grenzen gibt. Daher ist eine weiterführende Schleifenanalyse, wie sie in dieser Diplomarbeit entwickelt wurde, für eine automatisierte WCET-Analyse unabdingbar.

  – Anschließend wird mit Hilfe der *Cache-Analyse* versucht, das Speicherzugriffsverhalten zu analysieren. Dazu klassifiziert aiT die Speicherzugriffe in mehrere Kategorien (u.a. *always hit*, *always miss*), um eine exaktere $WCET_{est}$ erzeugen zu können.

  – Die letzte statische Analyse ist die *Pipeline-Analyse*. Auf Grundlage der vorher ermittelten Daten versucht aiT das Verhalten der Pipeline mit dessen komplexen Verhalten zu simulieren.

  Als Ergebnis der statischen Analyse wird eine neue CRL2-Datei zurückgegeben, für die an jedem Basisblock eine kontextabhängige Ausführungszeit annotiert ist. Kontextabhängig bedeutet in diesem Zusammenhang, dass es mehrere Zeiten für einen Basisblock, abhängig von

den aktuellen Registerwerten, oder auch der Pipelinebelegung, gibt.

Eine Funktion wird z.B. im Normalfall von mehreren Stellen und mit unterschiedlichen Werten der Parameter aufgerufen. Abhängig von diesen Parametern können Schleifen unterschiedliche Iterationsgrenzen aufweisen, wodurch eine starke Differenz in der Laufzeit derselben Funktion auftritt. Um für solche Fälle nicht eine starke Überapproximation zu erhalten, werden unterschiedliche Kontexte eingeführt um die unterschiedlichen Laufzeiten zu berücksichtigen [LFS+07].

- Den letzte Schritt der WCET-Analyse bildet die *Pfad-Analyse*. Diese bekommt die gerade beschriebene annotierte CRL2-Datei als Eingabe und baut zunächst das lineare Optimierungsproblem auf, um dieses im Anschluss zu lösen. Das Berechnungsmodell wird im folgenden Abschnitt genauer beschrieben.

2.5.2 Das Berechnungsmodell

Das Berechungsmodell ist auf Grund der Kontexte etwas komplexer, als das der *Implicit Path Enumeration Technique*, wie es in Abschnitt 2.2.2 beschrieben wurde.

Nach Abschluss der statischen Analysen wurden für alle Basisblöcke die folgenden Informationen ermittelt:

- *T(e,c)* = Die abgeschätzte WCET für die Kante e im Kontext c
- *C(e,c)* = Die Ausführungshäufigkeit der Kante e im Kontext c

Eine weitere Optimierung dieses Modells ist die Zuordnung von Zeiten, bzw. Ausführungshäufigkeiten, zu den Kanten der Basisblöcke. In der *Implicit Path Enumeration Technique* wurden diese Parameter noch direkt den Basisblöcken selbst zugeordnet. Da diese Informationen hier jedoch direkt den Kanten zugeordnet werden, können auch Laufzeiten, abhängig von den jeweiligen Vorgängern und Nachfolgern angegeben werden. Dies kann z.B. für die Simulation von Cache-Zugriffen von Bedeutung sein. Hat ein Basisblock zwei oder mehr Vorgänger, so kann sich abhängig des vorangegangenen Basisblocks ein *Cache-Hit* oder *Cache-Miss* einstellen, der die Anzahl an CPU Zyklen beeinflusst.

Der Wert der $WCET_{est}(\mathcal{P})$ ergibt sich nun aus der Maximierung der Summe der Kosten aller Iterationen der Kanten in jedem Kontext:

$$\sum_{\forall e,c} C(e,c) * T(e,c) \rightarrow \max \tag{2.7}$$

Zudem wird dieses Optimierungsproblem, wie in Kapitel 2.2.2 beschrieben, noch durch Einschränkungen des Kontrollflusses erweitert. Zusätzlich zu den Bedingungen, die sich aus dem Kontrollfluss

ergeben, kann aiT z.B. auch Ausführungsabhängigkeiten einzelner Basisblöcke in die Berechnung integrieren, falls diese vorher vom Benutzer angegeben wurden. Diese zusätzlichen Bedingungen können die Güte der abgeschätzten $WCET_{est}$ stark verbessern.

Nach Auflösung des Gleichungssystems durch den *LP Solver* ist die Berechnung der $WCET_{est}$ abgeschlossen und kann von aiT als Ergebnis der Analyse zurückgegeben werden.

2.6 Ausblick

Wie in diesem Kapitel gesehen, kann eine sichere Aussage über die maximale Laufzeit eines Programms nur durch eine statische WCET-Analyse erreicht werden. Dazu existieren bereits einige Tools, wie das vorgestellte aiT von AbsInt. Aktuell besteht jedoch das Problem, dass viele Zusatzparameter per Hand angegeben werden müssen, um die $WCET_{est}$ bestimmen zu können.

Zu diesen Zusatzparametern gehören in erster Linie Schleifeniterationsgrenzen, die aiT nur in sehr einfachen Fällen von selbst bestimmen kann. Für alle anderen Schleifen muss der Benutzer diese aktuell per Hand annotieren, was einerseits sehr zeitaufwändig, andererseits aber auch sehr fehleranfällig sein kann.

Daher ist das Ziel dieser Diplomarbeit eine deutlich komplexere Schleifenanalyse zu entwickeln, die vor dem Start von aiT die nötigen Schleifeniterationsgrenzen selbstständig bestimmt und in der AIS-Datei zur Verfügung stellt.

Kapitel 3

WCC – WCET-optimierender C-Compiler

Aktuell ist die Entwicklung eines harten Echtzeitsystems mit enormem Aufwand verbunden. So muss der Entwickler während des Entwurfs und auch später bei der Validierung prüfen, ob seine Software die vorgegebenen Zeitschranken einhält. Dafür wird die Software zunächst kompiliert und anschließend an ein WCET-Analyse-Tool übergeben, um die maximale Programmlaufzeit ($WCET_{est}$) zu approximieren. Ist die obere Zeitschranke nicht eingehalten worden, wiederholt sich dieser Zyklus nach einer manuellen Codeoptimierung. Dieser aufwändige und sehr fehleranfällige Prozess kann durch einen WCET-optimierenden Compiler signifikant vereinfacht werden.

Der Informatik Lehrstuhl 12 der Technischen Universität Dortmund hat mit dem WCC einen solchen WCET-optimierenden ANSI-C-Compiler für den Infineon TriCore Prozessor entwickelt. Im Gegensatz zu vorher veröffentlichten Werken ist, durch die Anbindung des kommerziellen Produktes *aiT* von AbsInt, ein komplexes WCET-Analyse-Tool integriert, das, wie bereits vorgestellt, von vielen namenhaften Unternehmen in der Industrie eingesetzt wird (siehe Kapitel 2.5).

Mit Hilfe des WCC ist der oben beschriebene Vorgang der manuellen, iterativen Optimierung nicht mehr nötig. Durch die Anbindung an aiT wird die approximierte $WCET_{est}$ automatisch während des Übersetzungsvorgangs ermittelt, so dass diese für automatische WCET-Optimierungen zur Verfügung steht. Der Aufbau ist dabei so modular gehalten, dass in der Zukunft auch weitere Optimierungen nach anderen Zielvorgaben (wie z.B. Reduktion des Energieverbrauchs) möglich sein sollen [FL06].

Auf Grund des modularen Aufbaus und der Anbindung des WCET-Analyzers aiT bietet der WCC eine Infrastruktur, die eine Integration der in dieser Arbeit vorgestellten Schleifenanalyse ermöglicht.

In diesem Kapitel wird zuerst der Aufbau des WCC und seiner Komponenten beschrieben. Anschließend soll im Kapitel 3.2 auf die Anforderungen und die Integration der Schleifenanalyse eingegangen werden.

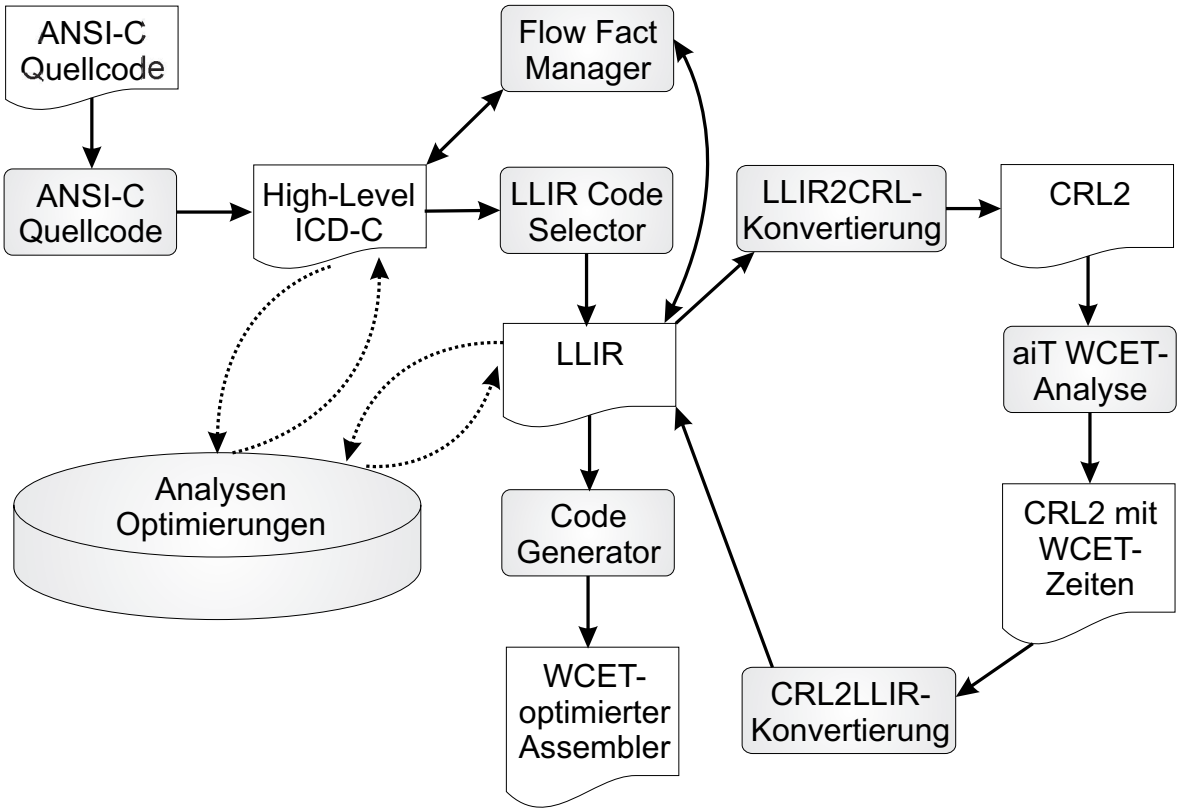


Abbildung 3.1: *Aufbau des WCC*

3.1 Aufbau des WCC

Der Aufbau des WCC ist in Abbildung 3.1 zu sehen. Er ist detailliert in [FL06] beschrieben.

Im ersten Schritt der Übersetzung wird der zu kompilierende Quellcode in die *ICD-C* Zwischendarstellung (siehe Abschnitt 3.1.1) transformiert. Diese sehr C-nahe Datenstruktur ermöglicht diverse Optimierungs- und Analyseverfahren, die vor der Umwandlung in die *ICD-LLIR* (Low-Level Intermediate Representation - siehe Abschnitt 3.1.2) vorgenommen werden können. Die ICD-LLIR ist die zweite Zwischendarstellung, die den Quellcode auf Assemblerebene repräsentiert. Die Transformation der beiden Zwischendarstellungen wird durch den *Code Selector* vorgenommen (siehe Abschnitt 3.1.3). Auch die *ICD-LLIR* bietet diverse Optimierungsverfahren, die in diesem Fall sowohl plattformunabhängig als auch Prozessor-spezifisch sind.

Der Unterschied zu Compilern, wie z.B. dem *gcc* [Fou08], ergibt sich durch die bereits erwähnte Anbindung an aiT. Diese ist ausführlich in [Lok05] beschrieben. Der *LLIR2CRL Konvertierer* transformiert die *ICD-LLIR* in die Zwischendarstellung CRL2, die aiT voraussetzt. In dieser Umformung werden der komplette Kontrollflussgraph sowie alle bis dahin gesammelten Kontrollflussinformationen (genannt *Flow Facts*), in das CRL2-Format übertragen. Der Aufruf des *exec2crl Konvertierers* von aiT, der ein Assembler-File zusammen mit benutzerdefinierten Annotationen ins CRL2-Format

transferiert, entfällt in diesem Fall, da durch den *LLIR2CRL Konvertierer* bereits eine gleichwertige CRL2-Datenstruktur erstellt worden ist.

Nachdem die Analyse von aiT beendet worden ist, werden bestimmte Informationen, die durch die Analyse gewonnen wurden, durch den *CRL2LLIR Konvertierer* zurück in die ICD-LLIR übertragen. Dazu gehören im Wesentlichen die Ausführungshäufigkeiten der Basisblöcke sowie die berechnete $WCET_{est}$. Die neu gewonnenen Informationen können im Anschluss für diverse Codeoptimierungen genutzt werden.

Um die Flow Facts bei allen Konvertierungen und Optimierungen konsistent zu halten, ist der *Flow Fact Manager* an die beiden Zwischendarstellungen *ICD-C* und *ICD-LLIR* angeschlossen. Werden z.B. Schleifen ausgerollt, so passt dieser automatisch die bisher angegebenen Schleifeniterationsgrenzen an.

Im Folgenden wird zunächst die ICD-C detailliert beschrieben. Sie bildet die Codegrundlage, auf der die in dieser Diplomarbeit entwickelte Schleifenanalyse aufgebaut ist. Anschließend werden auf Grund der Vollständigkeit die ICD-LLIR und der Code Selector in den Kapiteln 3.1.2 und 3.1.3 beschrieben, auch wenn diese für die Schleifenanalyse nicht benötigt werden. Schließlich wird der Flow Fact Manager in Abschnitt 3.1.4 etwas detaillierter beschrieben, da die Ergebnisse der Schleifenanalyse an diesen übergeben werden.

3.1.1 ICD-C

Die erste Zwischendarstellung des WCC ist die ICD-C. Es ist eine quellcodenahe ANSI-C-Datenstruktur, die vom Informatik Centrum Dortmund entwickelt worden ist. Zusätzlich zur eigentlichen Datenstruktur werden diverse Analyse- und Optimierungsverfahren angeboten [ICD07a].

3.1.1.1 Aufbau

Der Aufbau der ICD-C ist in Abbildung 3.2 zu sehen.

Das oberste Elemente der Hierarchie der ICD-C ist die *IR* (Intermediate Representation). Sie repräsentiert das vollständige zu analysierende Programm und besteht selbst aus mehreren eigenständigen *IR_CompilationUnits*. Jede Compilation Unit steht dabei für einen Teil des Programms mit eigenem Namensraum. Die Compilation Units bestehen wiederum aus mehreren Funktionen (*IR_Function*) und eigenen Symbolen, die in einer Symboltabelle (*IR_SymbolTable*) zusammengefasst werden. Die Funktionen enthalten in sich ein so genanntes *TopCompoundStatement*, ein Objekt der Klasse *IR_CompoundStmt*. Eine Compound-Anweisung ist ein zusammengesetztes Konstrukt, das mehrere Anweisungen (*IR_Stmt*) in der vorgegebenen Ausführungsreihenfolge enthält.

Auf diese Weise kann der Benutzer effektiv durch die komplette Programmhierarchie navigieren.

Die Klasse *IR_Stmt* ist als eine abstrakte Klasse implementiert, die durch die folgenden erbenden

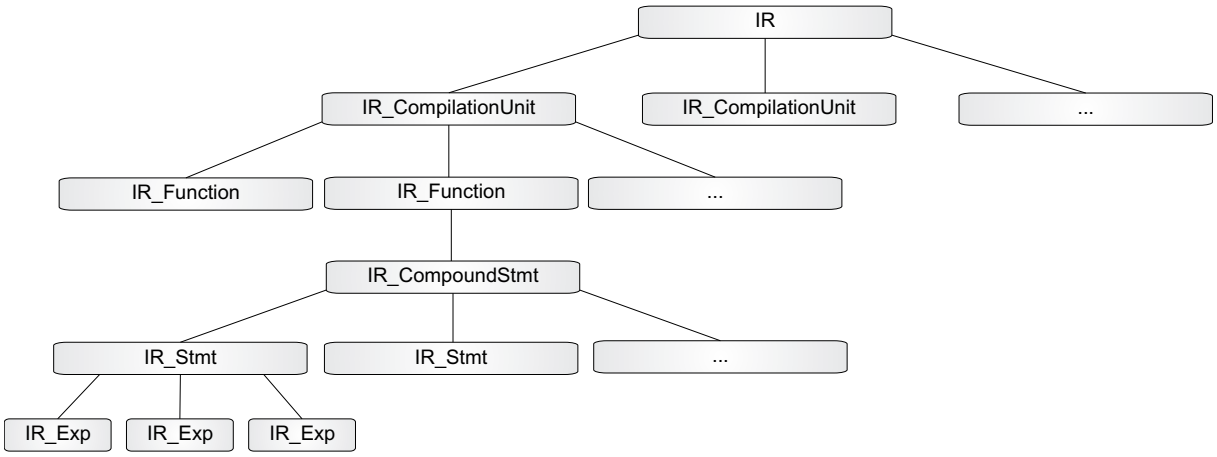


Abbildung 3.2: *Aufbau der ICD-C*

Klassen konkretisiert wird (siehe Abbildung 3.3):

- *IR_AsmStmt* - repräsentiert eine Assembler-Anweisung, die sich im Quellcode befindet.
- *IR_CompoundStmt* - repräsentiert eine zusammengesetzte Anweisung, die aus einer geordneten Liste von weiteren Anweisungen besteht. Eine weitere Konkretisierung stellt die Klasse *IR_InitClauseStmt* dar, die die Initialisierung einer For-Schleife beschreibt.
- *IR_ExpStmt* - repräsentiert einen Ausdruck, wie beispielsweise eine Zuweisung. Die einzelnen Ausdrücke (*Expressions*) werden weiter unten beschrieben.
- *IR_FunctionLoadArgStmt* - repräsentiert eine Pseudo-Anweisung, die im ursprünglichen Quellcode nicht zu finden ist. Sie wird ausschließlich für Kontrollflussanalysen benötigt und stellt die Verbindung von Funktionsargumenten und dessen Werten her.
- *IR_JumpStmt* - repräsentiert Sprünge innerhalb des ursprünglichen Quellcodes. Dazu gehören Break- (*IR_BreakStmt*) und Continue-Anweisungen (*IR_ContinueStmt*) aus Schleifen, sowie Goto- (*IR_GotoStmt*) und Return-Anweisungen (*IR_ReturnStmt*) für unbedingte Sprünge.
- *IR_LoopStmt* - repräsentiert alle Schleifentypen. Dazu gehören For- (*IR_ForStmt*), While- (*IR_WhileStmt*) und Do-While-Schleifen (*IR_DoWhileStmt*). Jede dieser Schleifen besteht dabei aus einer Bedingung und einem Schleifenrumpf. For-Schleifen bestehen zusätzlich aus einer Initialisierung und einer Continue-Anweisung, die sich, wie die Bedingung, im Schleifenkopf befindet.
- *IR_SelectionStmt* - repräsentiert alle möglichen bedingten Verzweigungen im Programm. Dazu gehören If- (*IR_IfStmt*), If/Else- (*IR_IfElseStmt*) und Switch-Anweisungen (*IR_SwitchStmt*). Alle Verzweigungen bestehen aus einer Bedingung und je nach Variante aus mehreren zusammengesetzten Anweisungen.

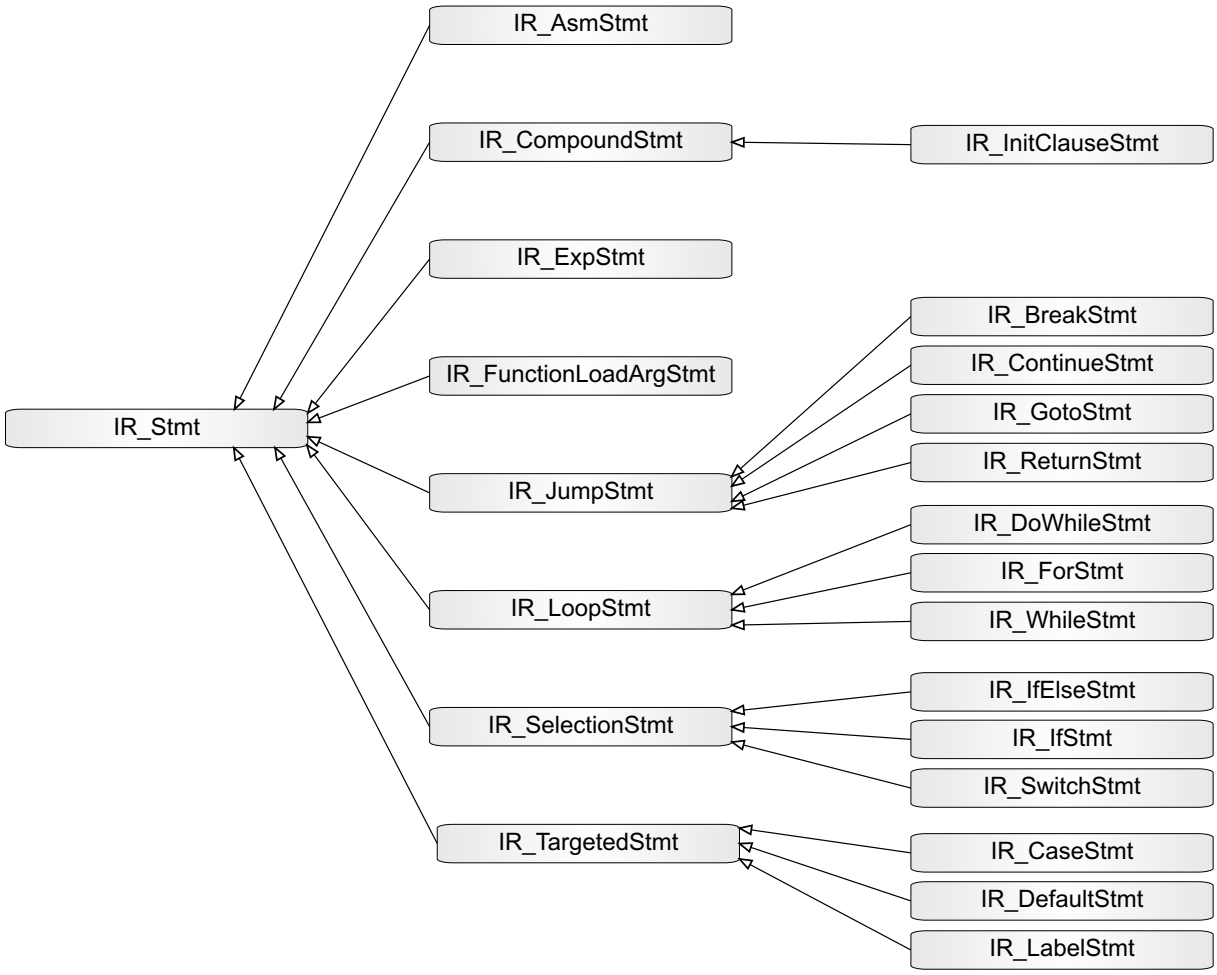


Abbildung 3.3: *Anweisungen (Statements) der ICD-C*

- *IR_TargetedStmt* - repräsentiert alle Markierungen, die für Sprungmarken von Relevanz sind. Dazu gehören Case- (*IR_CaseStmt*) und Default-Anweisungen (*IR_DefaultStmt*) für Switch-Anweisungen, sowie Labels (*IR_LabelStmt*) für unbedingte Sprünge durch Jump-Anweisungen.

Mit Hilfe dieser Anweisungen ist es möglich, alle Anweisungen des ANSI C-99 Standards darzustellen. Eine Besonderheit bilden die Ausdrucks-Anweisungen (*IR_ExpStmt*), da sich diese wiederum aus weiteren Ausdrücken zusammensetzen. Genau wie bei den Anweisungen gibt es eine abstrakte Oberklasse (*IR_Exp*), die durch die folgenden Konkretisierungen instantiiert werden kann:

- *IR_AssignExp* - repräsentiert eine Zuweisung. Diese besteht aus einem linken und einem rechten Ausdruck, wobei der rechte dem linken zugewiesen wird.
- *IR_BinaryExp* - repräsentiert alle möglichen binären Operationen, die auf zwei Ausdrücken angewendet werden können. Dazu zählt z.B. im arithmetischen Fall der „+“ oder „-“, sowie für Bedingungen der „AND“ oder „OR“ Operator. Eine vollständige Liste der möglichen Operatoren ist der entsprechenden Dokumentation aus [ICD07a] zu entnehmen.
- *IR_CallExp* - repräsentiert einen Funktionsaufruf. Sie enthält u.a. die übergebenen Argumente, sowie das Symbol der aufgerufenen Funktion.
- *IR_ComponentAccessExp* - repräsentiert einen Zugriff auf ein zusammengesetztes Objekt, wie z.B. eine Struct.
- *IR_CompoundLiteralExp* - repräsentiert ein zusammengesetztes Literal, das als linker Teil einer Zuweisung genutzt werden kann.
- *IR_CondExp* - repräsentiert einen abhängigen Ausdruck durch den ?-Operator.
- *IR_ConstExp* - repräsentiert einen konstanten Ausdruck. Dieser wird dabei je nach Typ durch einen der folgenden Untertypen konkretisiert: *IR_FloatConstExp*, *IR_IntConstExp*, *IR_SizeOfExp* oder *IR_StringConstExp*.
- *IR_IndexExp* - repräsentiert den Zugriff auf ein Element eines Feldes (Array).
- *IR_InitListExp* - repräsentiert die Initialisierung eines zusammengesetzten Typs, wie z.B. eines Array oder einer Struct.
- *IR_SymbolExp* - repräsentiert einen Ausdruck, der aus einem Symbol besteht.
- *IR_TypeExp* - repräsentiert einen Ausdruck, der ausschließlich aus einem Typ besteht.
- *IR_UnaryExp* - repräsentiert eine unäre Operation auf einem Ausdruck. Dazu gehören z.B. Casts, Dereferenzierungen von Pointern oder auch das Pre- und Postincrement. Alle weiteren Operatoren sind der entsprechenden Dokumentation aus [ICD07a] zu entnehmen.

Mit den beschrieben Klassen ist es möglich, alle relevanten Kontrollstrukturen eines ANSI-C Programms darzustellen.

3.1.1.2 Analysefunktionen

Die ICD-C besteht aus diversen Analysefunktionen, die vor allem Aussagen über den Daten- und Kontrollfluss liefern sollen. Mit diesen können z.B. Anweisungen, oder auch Ausdrücke, bequem durch einen Funktionsiterator durchlaufen werden, um neue Analysefunktionen zu ergänzen.

Bezüglich des Kontrollflusses stehen u.a. Methoden zur Verfügung, mit denen getestet werden kann, ob eine Anweisung durch eine andere dominiert wird, oder ob sie z.B. in einer anderen Anweisung in einer Vater-Kind-Relation vorkommt.

Unter den Analysefunktionen befindet sich auch eine statische Schleifenanalyse. Diese liefert aktuell nur für einfache Schleifen, deren Bedingung ausschließlich von Konstanten abhängig ist, die Iterationsgrenzen. Informationen über die möglichen Werte von Variablen werden nicht gesammelt, so dass diese Analyse nur stark eingeschränkte Ergebnisse liefert und aus diesem Grund für die meisten Analysen, insbesondere für die WCET-Analyse, nicht einsetzbar ist.

Eine für die Schleifenanalyse interessante Analysefunktion ist die *Alias-Analyse*, die auf Basis der ICD-C entwickelt worden ist (siehe [Bih05]). Diese kann separat beim Übersetzen des WCC zur ICD-C hinzugefügt werden. Die Alias-Analyse versucht durch eine statische Pointer-Analyse eine Aussage zu treffen, auf welche Speicherbereiche (bzw. Symbole) ein Pointer innerhalb einer Funktion zeigen kann. Diese Analysefunktion soll während der Schleifenanalyse genutzt werden, um Dereferenzierungen von Pointern durchführen zu können.

Des Weiteren enthält die ICD-C ein *Program Slicing* [HRB90], welches das zu analysierende Programm auf die relevanten Anweisungen reduziert. Bezogen auf die Berechnung der Schleifengrenzen, sind lediglich die Anweisungen interessant, die den Kontrollfluss, bzw. die Schleifengrenzen, direkt beeinflussen. Alle weiteren Anweisungen können mit Hilfe dieses Verfahrens eliminiert werden, so dass diese nicht zusätzlich ausgewertet werden müssen. Durch Anwendung des Program Slicings kann so die Laufzeit von komplexeren Analyseverfahren, wie die einer automatischen Schleifenanalyse, signifikant reduziert werden. Daher ist dieses Verfahren auch für diese Arbeit von großem Interesse.

3.1.1.3 Optimierungsverfahren

Wie bereits in der Einleitung dieses Kapitels erwähnt, beinhaltet die ICD-C diverse Optimierungsverfahren. Diese sind aktuell in drei Stufen eingeteilt, die über die Parameter des WCC aktiviert, bzw. deaktiviert werden können. Die Optimierungsstufen repräsentieren ausgewählte Kombinationen von Optimierungen, um die Codequalität sukzessiv zu verbessern.

Zu den bisher implementierten Verfahren gehören z.B. einfachere Optimierungen, wie das Eliminieren von nicht erreichbaren Programmteilen (Dead Code), oder eine Constant Value Propagation, die für konstante Ausdrücke das jeweilige Symbol durch den Wert ersetzt. Höhere Optimierungsstufen verändern hingegen größere Strukturen des Programms (wie Ausrollen, oder Kombinieren von

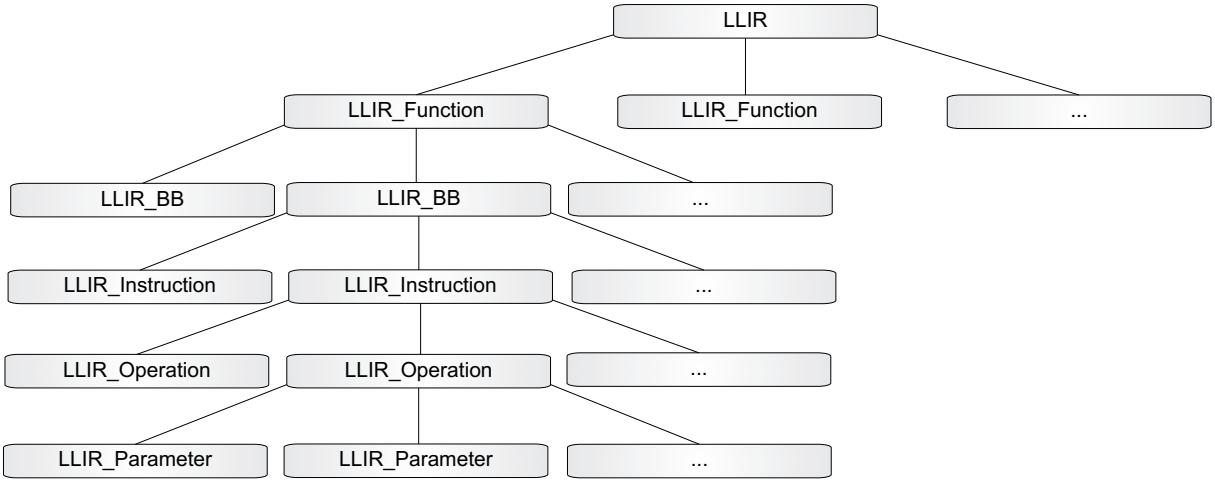


Abbildung 3.4: *Aufbau der ICD-LLIR*

Schleifen).

3.1.1.4 Benutzerdefinierte Erweiterungen

Die ICD-C bietet mit ihrem modularen Aufbau die Möglichkeit, den zu Grunde liegenden Quellcode für diverse Analysen zu erweitern, ohne dabei den eigentlichen Kern ändern zu müssen. Wird z.B. für eine Analyse eine zusätzliche Information für einen Ausdruckstyp benötigt, so kann dieser über so genannte *User Data* angehängt werden. Diese stellen generische Container dar, die an jedes ICD-C Konstrukt angeheftet werden können.

Dazu erbt jede Klasse der ICD-C Datenstruktur von der Oberklasse *IR_PersistentObject*. Diese vererbt u.a. die Methoden *getUserData(ITEM_t key)*, *setUserData(ITEM_t key, IR_PersistentObject *data)* und *removeUserData(ITEM_t key)*, über die der Benutzer die benötigten Parameter dem passenden Objekt zuordnen kann. Dabei wird jedes IR_PersistentObject über einen eindeutigen, frei wählbaren Identifikator (*key*) gekennzeichnet, um einen erneuten Zugriff zu ermöglichen. Die einzige Bedingung ist, dass auch die Parameter von der Klasse *IR_PersistentObject* erben.

3.1.2 ICD-LLIR

Die ICD-LLIR ist ebenfalls vom Informatik Centrum Dortmund entwickelt worden [ICD07b]. Sie repräsentiert den Programmcode in einer assemblernahen Datenstruktur. Innerhalb des WCC konvertiert der *Code Selector* die ICD-C in die ICD-LLIR Datenstruktur. In der High-Level-Repräsentation ICD-C ist das Programm durch seine Funktionen definiert, die wiederum aus Compound-Anweisungen bestehen (siehe Abschnitt 3.1.1.1). Nach der Übersetzung durch den *Code Selector* werden diese innerhalb der ICD-LLIR durch Basisblöcke (siehe Definition 1 auf Seite 11) repräsentiert.

Der Aufbau der ICD-LLIR-Klassen wurde wie folgt umgesetzt (vergleiche Abbildung 3.4):

- *LLIR* - ist die Klasse, die auf Ebene der ICD-C einer *IR_CompilationUnit* entspricht.
- *LLIR_Function* - repräsentiert eine Assembler-Routine, die sich aus Basisblöcken zusammensetzt.
- *LLIR_BB* - ist ein Basisblock nach Definition 1 auf Seite 11. Dieser besteht aus einer Sequenz von einzelnen Instruktionen. Der Basisblock wird durch einen eindeutigen Namen (*Label*) adressiert, bzw. identifiziert.
- *LLIR_Instruction* - ist eine Anweisung, die auch aus mehreren Operationen bestehen kann, falls es sich um einen VLIW-Prozessor (Very Long Instruction Word - Prozessor) handelt.
- *LLIR_Operation* - ist ein einzelner Maschinenbefehl mit den dazugehörigen Parametern.
- *LLIR_Parameter* - ist ein Parameter einer Operation. Dieser kann z.B. eine Integer-Konstante, ein Label, ein Operator oder ein Register sein.

Die ICD-LLIR bietet, genau wie die ICD-C, diverse Analysen und Optimierungen. Des Weiteren kann auch die ICD-LLIR durch benutzerdefinierte Daten erweitert werden, die sogenannten *Objectives*. Der Aufbau ist analog zu dem beschriebenen Mechanismus der *Persistent-Objects* aus der ICD-C (siehe Abschnitt 3.1.1.4). Da diese Methoden für die Schleifenanalyse nicht von Bedeutung sind, wird an dieser Stelle nicht weiter darauf eingegangen.

3.1.3 Code Selector

Der Code Selector transformiert, wie bereits erwähnt, eine ggf. optimierte ICD-C Datenstruktur in die ICD-LLIR. Für jede Compilation Unit wird auf Ebene der ICD-LLIR ein *LLIR*-Objekt erzeugt. Die Funktionen innerhalb der Compilation Units werden durch Objekte der Klasse *LLIR_Function* repräsentiert. Alle tieferen Ebenen der Funktionen, werden schließlich in *LLIR_BB* Basisblock-Objekte übersetzt, die sich, wie in Abschnitt 3.1.2 beschrieben, weiter in Instruktionen, Operationen und Parameter aufsplitten.

3.1.4 Flow Fact Manager

Der Flow Fact Manager verwaltet alle Flow Facts, die im Laufe der Analysen gesammelt werden [Sch07]. Dazu gehören u.a. Benutzerannotationen, die für die Berechnung der $WCET_{est}$ benötigt werden.

Ursprünglich mussten diese Benutzerannotationen in einer separaten Datei abgelegt werden. Diese Informationen bezogen sich dabei vor allem auf die Struktur der ICD-LLIR bzw. der CRL2, obwohl diese erst im Laufe der Übersetzung entstehen. Auf Grund der unterschiedlichen Optimierungsstufen und den daraus resultierenden unterschiedlichen Codeergebnissen, kann die Angabe der benötigten Kontrollflussinformationen also nur durchgeführt werden, wenn der Benutzer detaillierte

Kenntnisse über den Compiler hat, wobei die Annotation selbst dann noch sehr fehleranfällig ist. Selbst wenn der Benutzer es schafft, die Kontrollflussinformationen wie beispielsweise Schleifeniterationsgrenzen vorherzusagen und korrekt anzugeben, so kann eine leichte Änderung am Quellcode eine andere Codetransformation hervorrufen, wodurch die mühsam berechneten Informationen verworfen werden müssten.

Mit Hilfe des Flow Fact Managers ist es jedoch möglich, Benutzerannotationen bereits im Quellcode zu spezifizieren, wobei sich diese direkt auf die jeweilige Codestelle beziehen. Der Flow Fact Manager überwacht automatisch alle Optimierungsverfahren, sowie auch die Codeselektion, um die Annotationen ggf. anzupassen.

Die Ergebnisse der Schleifenanalyse, die in dieser Arbeit vorgestellt wird, sollen später automatisch an den Flow Fact Manager übergeben werden, so dass diese für den darauf folgenden Kompiliervorgang zur Verfügung stehen.

3.2 Entwicklung der Schleifenanalyse

Auf Grund des vorgestellten Funktionsumfangs ist der WCC sehr gut geeignet, die Infrastruktur für die in dieser Arbeit vorgestellte Schleifenanalyse zu bilden.

In einigen älteren Publikationen, wie z.B. in [HSRW98] oder der Schleifenanalyse von aiT (siehe Abschnitt 2.5), wurde versucht auf Grundlage einer assemblernahen Repräsentation des Codes die Schleifeniterationsgrenzen zu bestimmen. In dieser Arbeit soll ein Ansatz vorgestellt werden, der dies bereits auf einer quellcodeähnlichen Repräsentation versucht.

Auf Grund der Komplexität einer solchen Analyse kann, wie in Kapitel 2 gesehen, nur eine approximative Lösung in endlicher Zeit gefunden werden. Dies soll mit Hilfe der Abstrakten Interpretation (siehe Kapitel 4) erreicht werden.

Da eine Analyse mit Hilfe der abstrakten Interpretation unter Umständen recht zeitaufwendig sein kann, wird in Kapitel 6 eine statische Schleifenanalyse vorgestellt, die, falls möglich, die Ausführung der Programmanalyse beschleunigen soll.

Wie diese Analyse in den WCC integriert wurde, wird im nächsten Abschnitt beschrieben.

3.2.1 Integration der Schleifenanalyse

Die Integration der Schleifenanalyse in den WCC ist in Abbildung 3.5 dargestellt.

Da die Analyse direkt auf einer quellcodenahen Repräsentation durchgeführt werden soll, bietet die ICD-C ideale Voraussetzungen dafür. Mit Hilfe des Flow Fact Managers reicht es zudem aus, die Analyse ausschließlich auf dieser Abstraktionsebene durchzuführen. Die Ergebnisse werden nach

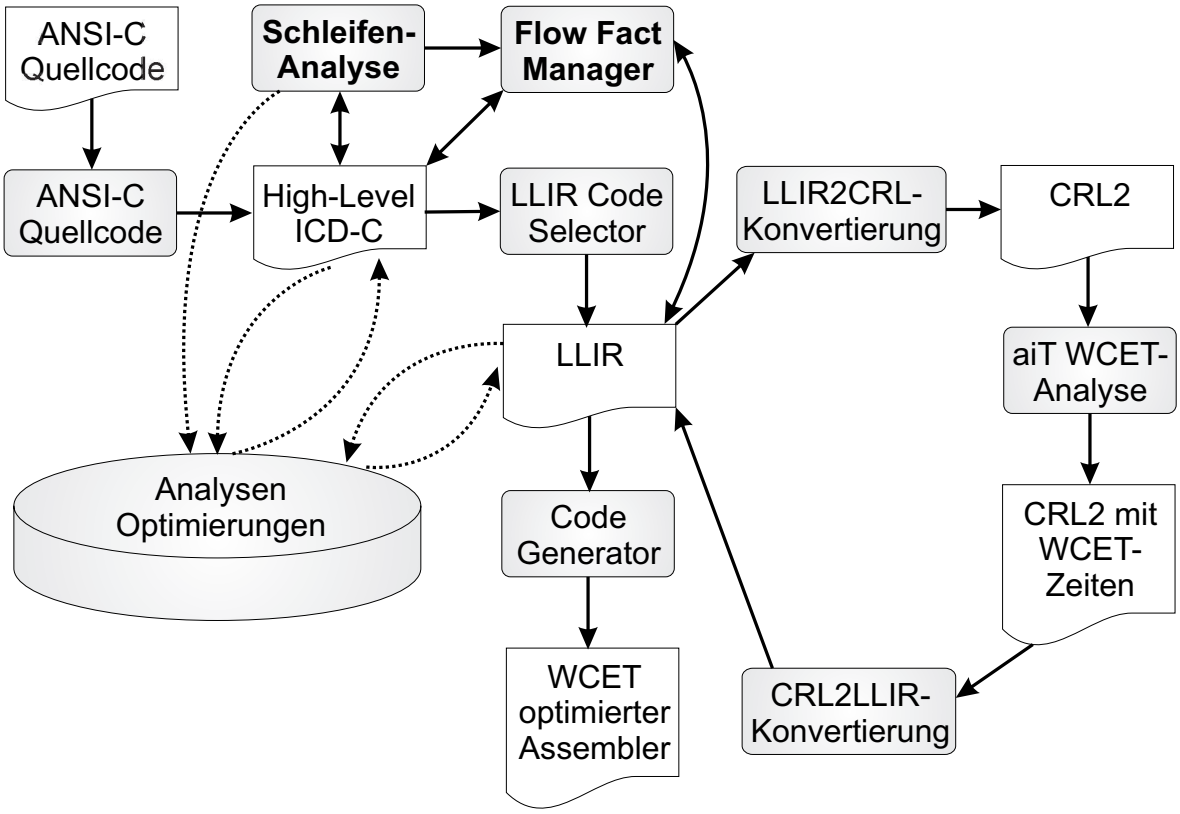


Abbildung 3.5: *Integration der Schleifenanalyse in den WCC*

Übergabe an den Flow Fact Manager automatisch bei allen Optimierungen, sowie bei der Umwandlung zwischen der ICD-C und der ICD-LLIR, angepasst. Eine weitere Analyse auf Assemblerebene ist dadurch überflüssig.

Während der Durchführung der Schleifenanalyse werden zudem mehrere Analyseverfahren der ICD-C genutzt. Um z.B. eine Unterstützung von Pointern innerhalb des zu analysierenden Programms zuzulassen, wird die bereits erwähnte Alias-Analyse durchgeführt. Des Weiteren wird z.B. auf die bereits bestehende Schleifenanalyse zurückgegriffen, um die Induktionsvariable einer Schleife zu bestimmen.

Der Kompiliervorgang sieht nach Integration der Schleifenanalyse wie folgt aus: Zuerst wird der Quellcode geparst, um daraus die ICD-C Datenstruktur aufzubauen. Im Anschluss werden diverse Analyseverfahren, unter anderem auch die Alias-Analyse, gestartet, um alle nötigen Informationen z.B. für eine Dereferenzierung zu sammeln. Der Flow Fact Manager wird ebenfalls vor der eigentlichen Schleifenanalyse instantiiert, damit die Ergebnisse später an diesen übergeben werden können.

Nach diesem Schritt sind alle nötigen Vorbereitungen getroffen, um die Schleifenanalyse durchzuführen. Wie diese im Einzelnen funktioniert, wird in den folgenden Kapiteln dieser Arbeit beschrieben. Nachdem die Analyse abgeschlossen ist, werden alle gefunden Schleifeniterationsgrenzen an

den Flow Fact Manager übergeben.

Im Anschluss werden diverse Highlevel-Optimierungen durchgeführt. Die optimierte ICD-C wird schließlich durch den Code Selector in die ICD-LLIR übersetzt, so dass mit Hilfe der Codetransformation in die CRL2 die $WCET_{est}$ durch aiT berechnet werden kann. Hierbei werden die transformierten Ergebnisse der Schleifenanalyse verwendet. Diese Informationen werden wiederum für diverse Optimierungen auf der ICD-LLIR genutzt, so dass diese Zwischendarstellung schließlich in den finalen Assemblercode übersetzt wird.

Kapitel 4

Abstrakte Interpretation

Die *Abstrakte Interpretation* bildet die theoretische Grundlage der vorliegenden Arbeit, da die in den folgenden Kapiteln beschriebene Schleifenanalyse auf einer modifizierten Version der Abstrakten Interpretation aufbaut. Das Ziel dieses Kapitels soll es daher sein, die Grundlagen der Abstrakten Interpretation einzuführen. Die Abstrakte Interpretation erlaubt eine statische Programmanalyse, mit der approximative Aussagen über ein Programm gemacht werden können. Erstmals vorgestellt wurde diese Idee von Patrick und Radhia Cousot.

Der Grundgedanke der Abstrakten Interpretation ist, dass eine möglicherweise unendlich große Menge an Programmzuständen (mögliche Werte von Variablen abhängig vom Programmpunkt) zu einer endlichen und dadurch effizient berechenbaren Menge verschmolzen werden. Durch die Verschmelzung werden unterschiedliche Programmpfade zusammengeführt, so dass die möglichen Werte der Variablen, abhängig von der *Abstrakten Domäne* (siehe Abschnitt 4.2.3), approximiert werden. Diese Abstrakte Domäne kann, wie in dieser Arbeit umgesetzt, z.B. aus einer Intervalldarstellung bestehen.

Die Abstrakte Interpretation simuliert das Programm mit Hilfe einer *abstrakten Semantik*, die sich aus den Übergangsfunktionen und der gewählten Abstrakten Domäne ergibt. Durch die abstrakte Semantik wird schließlich ein Gleichungssystem aufgebaut, das für jeden Programmpunkt, abhängig vom Kontrollfluss, einen neuen Programmzustand berechnet [Byg06].

In der Praxis wird die Abstrakte Interpretation für diverse Analyseverfahren eingesetzt. Ein Anwendungsgebiet ist beispielsweise das *Model-Checking*, das versucht, auf einem formalen Weg zu bestimmen, ob das vorliegende Programm, laut seiner Definition, korrekt arbeitet. Auf Grund der Komplexität ist auch diese Analysetechnik nur mit Hilfe einer Abstraktion möglich. Des Weiteren wurde die Abstrakte Interpretation auch bereits in anderen Schleifenanalyse-Tools, wie beispielsweise dem SWEET (SWEdish Execution Tool), als theoretische Grundlage eingesetzt [GESL07].

Dieses Kapitel soll im Folgenden das Ziel haben die modifizierte Abstrakte Interpretation dieser Arbeit vorzustellen. Dafür wird in Abschnitt 4.1 zunächst auf den Aufbau der statischen Program-

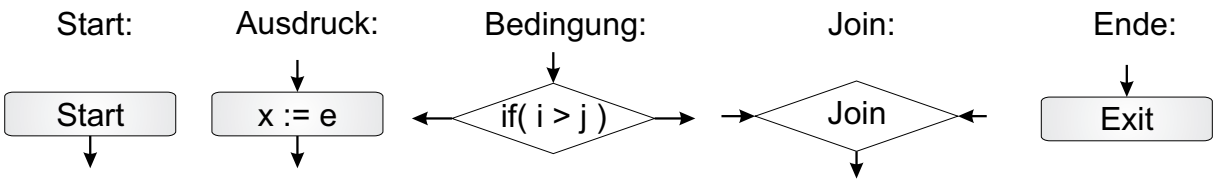


Abbildung 4.1: *Knoten eines Kontrollflussgraphen*

manalyse, zu der auch die Abstrakte Interpretation gehört, eingegangen. Im Anschluss wird in Abschnitt 4.2 detailliert die Struktur der klassischen Abstrakten Interpretation beschrieben, so wie sie ursprünglich von Patrick und Radhia Cousot vorgestellt worden ist. Auf dieser Grundlage wird schließlich in Abschnitt 4.3 beschrieben, wie mit Hilfe der modifizierten Abstrakten Interpretation dieser Arbeit die Analyse von Schleifengrenzen durchgeführt wird. Der Inhalt der Kapitel 4.1 und 4.2 ist dabei in großen Teilen an die Arbeiten aus [Byg06] und [NNH99] angelehnt, ohne dass dies im Weiteren an jeder Stelle vermerkt werden soll.

4.1 Statische Programmanalyse

Die Abstrakte Interpretation gehört zur Familie der statischen Programmanalysen. Um die Besonderheiten der Abstrakten Interpretation im Detail vorstellen zu können, wird an dieser Stelle zuerst auf die Grundlagen einer einfachen statischen Programmanalyse eingegangen, die auf der konkreten Programmsemantik basiert.

Ziel der Analyse ist es, für jede Kante im Kontrollflussgraph die Menge seiner möglichen Zustände, d.h. die möglichen Werte der Variablen, zu bestimmen.

In diesem Kapitel soll zunächst der Kontrollflussgraph beschrieben werden, auf dem die spätere Berechnung stattfinden wird. Im Anschluss werden in Abschnitt 4.1.2 einige Definitionen vorgenommen, so dass schließlich in Abschnitt 4.1.3 auf die eigentliche Berechnung der statischen Programmanalyse eingegangen werden kann.

4.1.1 Der Kontrollflussgraph

Der Aufbau der Kontrollflussgraphen wurde bereits in Abschnitt 2.2.1 definiert. Jedes Programm einer imperativen Programmiersprache kann durch einen Kontrollflussgraph beschrieben werden, der aus den 5 Knoten der Abbildung 4.1 besteht. Dabei befindet sich an jeder Kante des Kontrollflussgraphen eine Menge möglicher Zustände.

Ein Programm wird immer über einen fest definierten Eintrittspunkt betreten. Dieser wird durch den *Start-Knoten* in einem Kontrollflussgraphen repräsentiert. Der ausgehenden Kante des Start-Knotens wird ein leerer Startzustand zugeordnet. Dies bedeutet, dass an diesem Punkt des Programms noch keine Aussage über die möglichen Werte der Variablen getroffen werden kann. Ana-

log zu dem Start-Knoten existiert in jedem Programm ein *End-Knoten*, der genau dann erreicht wird, wenn das Programm terminiert.

Des Weiteren kann der Graph mehrere *Ausdrucks-Knoten* beinhalten. Diese bestehen beispielsweise aus einer Anweisung, die einer Variable einen Wert zuweist.

Um beliebige Verzweigungen, wie z.B. If/Else- oder auch Schleifenausdrücke, darstellen zu können, wird ein *Bedingungs-Knoten* benötigt. Dieser hat eine eingehende, und zwei ausgehende Kanten. Dabei steht eine der ausgehenden Kanten für den Bereich, der betreten wird, wenn die Bedingung erfüllt wird. Die andere Kante steht analog für den Fall, dass die Bedingung nicht erfüllt wird.

Abschließend gibt es *Join-Knoten*, die verzweigte Programmpfade wieder zusammenführen. Diese haben zwei eingehende und eine ausgehende Kante.

4.1.2 Definitionen

Im Folgenden sollen einige Definitionen eingeführt werden, die für das Verständnis der statischen Programmanalyse erforderlich sind [Byg06].

- $IDENT_{Prg}$ steht für die Menge aller Variablen des Programms *Prg*.
- $\mathcal{V}$ beinhaltet alle möglichen Werte, die den Variablen innerhalb des Programms zugewiesen werden können. Zusätzlich zu den Werten, die laut Variablentyp vorgegeben sind, existieren die beiden Sonderwerte $\perp$ (*bottom*) und $\top$ (*top*). Der Operator $\perp$ bedeutet in diesem Fall, dass die Variable bisher nicht initialisiert worden ist. Die Bedeutung des $\top$-Operators soll ausdrücken, dass die Variable alle möglichen Werte ihres Wertebereichs annehmen kann.
- $\sigma : IDENT \rightarrow \mathcal{V}$ ist die so genannte Umgebung (Environment). Die Abbildung σ weist einer Variable seine möglichen Werte zu.
- ENV ist die Menge aller Umgebungen, d.h. die Menge, die für jede Variable die Menge der möglichen Werte beinhaltet.
- q ist ein Programmpunkt. Jeder Kante $\langle n_1, n_2 \rangle$ des Kontrollflussgraphen ist ein solcher Programmpunkt zugewiesen.
- Q ist die Menge aller Programmpunkte.
- S_q ist ein Zustand, der zu dem Programmpunkt q gehört. Dieser besteht aus einer Relation zwischen dem Programmpunkt *q* und der Zuweisung zu einer passenden Umgebung ($\langle \sigma, q \rangle$).
- $STATES_q$ ist die Menge aller Zustände, die für den Programmpunkt *q* gültig sind.
- $STATES$ ist die Menge aller möglichen Zustände, die sich aus $ENV \times Q$ ergibt.

Ziel der statischen Programmanalyse ist also die Menge aller möglichen Umgebungen zu finden, die an jedem Programmpunkt auftreten können. Dies entspricht der Menge $STATES$.

4.1.3 Die Berechnung

Aus dem Kontrollfluss, bzw. der Semantik des Programms, kann ein *Transitionssystem* konstruiert werden, das die Grundlage der Berechnung bildet. In diesem Transitionssystem wird festgehalten, wie ein Zustand, abhängig vom Programmpunkt, in einen neuen Zustand überführt wird. Dieses Transitionssystem ist in Definition 4.1 abgebildet:

$$\tau : \mathcal{P}(STATES) \rightarrow \mathcal{P}(STATES) \tag{4.1}$$

Mit Hilfe des Transitionssystems τ wird über die Berechnung des kleinsten Fixpunktes ($lfp(\tau)$) versucht, die möglichen Zustände für alle Programmpunkte *q* zu berechnen.

Die Menge aller Umgebungen ENV bildet dabei einen vollständigen Verband, der für die Fixpunktberechnung benötigt wird. Dieser Verband wird wie folgt definiert:

$$(S, \sqcap, \sqcup, \sqsubseteq, \top, \bot) \Leftrightarrow (\mathcal{P}(ENV), \cap, \cup, \subseteq, ENV, \emptyset) \tag{4.2}$$

Der Verband ist also durch die Menge aller Umgebungen definiert. Supremum und Infimum, sowie die Teilmengenbeziehung, sind auf Grundlage der möglichen Werte der Umgebungen vorgegeben. Die Obermenge $\top$ ist durch alle möglichen Umgebungen definiert, wobei das Element $\bot$ durch die leere Menge gegeben ist. Der vollständige Verband, der direkt aus den Umgebungen ENV abgeleitet ist, nennt sich *konkrete Semantik-Domäne*.

Gestartet wird die Fixpunktberechnung mit $\tau(\bot)$. Das heißt, dass noch keiner Variable ein möglicher Wert zugewiesen worden ist. Damit die Ergebnisse, die bei der statischen Programmanalyse entstehen, korrekt sind, muss die Funktion τ monoton sein. Es gilt also:

$$\tau(STATES)^{n-1} \subseteq \tau(STATES)^{n} \tag{4.3}$$

Die Definition 4.3, bzw. die Forderung nach Monotonie für die Funktion τ, ist dadurch begründet, dass keine vorher berechneten Zustände verworfen werden dürfen. Ziel der Analyse ist es schließlich *alle* möglichen Zustände, bzw. Variablenbelegungen, zu berechnen.

Die Fixpunktberechnung iteriert nun so lange, bis durch eine erneute Anwendung des Transitionssystems keine neuen Zustände berechnet werden können:

$$\tau^{n-1} = \tau^{n} \tag{4.4}$$

Ist die Gleichung 4.4 erfüllt, so ist die Menge $STATES$ bestimmt. In diesem Fall ist also für jeden Programmpunkt die Menge seiner möglichen Zustände bestimmt.

Auf Grund der Komplexität der *konkreten Semantik-Domäne* kann jedoch nicht garantiert werden, dass die Fixpunktberechnung in endlicher Zeit terminiert. Eine solch komplexe Analyse ist nur für sehr kleine Programme möglich.

4.2 Klassische Abstrakte Interpretation

Ziel der Abstrakten Interpretation wird es daher sein, eine geeignete Approximation der *konkreten Semantik-Domäne* zur Verfügung zu stellen, so dass die vorgestellte Fixpunktberechnung aus Abschnitt 4.1.3 (möglichst zeitnah) terminiert. Dabei sollten die berechneten Ergebnisse natürlich so genau wie möglich sein. Da beide Forderungen kontrovers sind, gilt es, die Ergebnisse so exakt wie nötig zu berechnen, damit die Laufzeit so gering wie möglich bleibt.

Welche Anforderungen an die Transformation der konkreten Programmsemantik in eine abstraktere Programmsemantik, also der *Abstrakten Domäne*, einzuhalten sind, wird in Abschnitt 4.2.1 und 4.2.2 erläutert. In Abschnitt 4.2.3 werden anschließend beispielhaft zwei Abstrakte Domänen mit Ihren Abbildungsregeln vorgestellt. Damit auf den konstruierten Abstrakten Domänen gerechnet werden kann, werden in Abschnitt 4.2.4 die *Abstrakten Operatoren* vorgestellt. Mit Hilfe der bis dahin beschriebenen Methoden ist es schließlich möglich das Transitionssystem aufzubauen, dass über die Fixpunktberechnung gelöst werden soll. Der Aufbau dieses Systems ist in Abschnitt 4.2.5 erläutert. Zusammenfassend wird in Abschnitt 4.2.6 der gesamte Vorgang anhand eines Beispiels zusammengefasst. Abschließend werden die Probleme, die auf Grund der nicht garantierten Terminierung der Fixpunktberechnung beruhen, in Abschnitt 4.2.7 zusammen mit einem möglichen Lösungsansatz vorgestellt.

4.2.1 Galois-Verbindung

Der Grundgedanke der Galois-Verbindung (*Galois Connection*) ist, dass eine Berechnung auf dem vollständigen Verband *L* (wie z.B. der *konkreten Semantik-Domäne*) zu komplex ist. Daraus resultiert die Motivation, den vollständigen Verband *L* durch einen zweiten vollständigen Verband *M* zu ersetzen, auf dem die Berechnung möglicherweise weniger komplex ist. Die Ergebnisse, die im Verband *M* erzielt wurden, werden anschließend über eine weitere Transformation zurück in den ursprünglichen Verband *L* abgebildet.

Diese Transformationen werden durch die zwei Funktionen erreicht, die im folgenden eingeführt werden.

$$\alpha : L \rightarrow M \tag{4.5}$$

Die Funktion α aus Definition 4.5 wird *Abstraktionsfunktion* genannt. Sie bildet alle Elemente des ursprünglichen Verbands *L* in den neuen Verband *M* ab.

$$\gamma : M \to L \tag{4.6}$$

Die Funktion γ aus Definition 4.6 wird *Konkretisierungsfunktion* genannt. Sie bildet das Gegenstück zur Abstraktionsfunktion und bildet alle Elemente des neuen Verbands zurück in den ursprünglichen Verband ab.

Die Kombination aus Definition 4.5 und 4.6 ist in Definition 4.7 festgehalten:

$$L \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} M \tag{4.7}$$

Der ursprüngliche Verband *L* wird über die Abstraktionsfunktion α in den neuen Verband *M* abgebildet. Die Berechnung der statischen Programmanalyse wird auf diesem neuen Verband durchgeführt, bevor das Ergebnis durch die Konkretisierungsfunktion γ zurück in den ursprünglichen Verband *L* übertragen wird.

(L, α, γ, M) wird im Folgenden als Galois-Verbindung zwischen den beiden vollständigen Verbänden $(L, \sqcap, \sqcup, \sqsubseteq, \top, \bot)$ und $(M, \sqcap, \sqcup, \sqsubseteq, \top, \bot)$ bezeichnet.

Damit durch die Galois-Verbindung eine sichere Approximation vorgenommen wird, müssen die beiden Funktionen $\alpha : L \to M$ und $\gamma : M \to L$ monoton sein. Formal ist dies in Gleichung 4.8 festgehalten:

$$l \sqsubseteq \gamma(\alpha(l)) \tag{4.8}$$

Für jedes $l \subseteq L$ gilt also, dass es eine Teilmenge des Ergebnisses ist, das durch ein aufeinanderfolgendes Anwenden der Abstraktions- und Konkretisierungsfunktion berechnet wurde. Durch die Transformation der Galois-Verbindung darf die Menge l also nur größer oder gleich der ursprünglichen Menge werden. Dieser Zusammenhang ist in Abbildung 4.2 zu sehen.

4.2.1.1 Beispiel

Der ursprüngliche Verband *L* sei durch die Menge $\mathbb{Z}$ der ganzzahligen Werte gegeben. Ein möglicher Abstraktionsverband *M* könnte durch die Domäne der Intervalle gebildet werden (diese ist in Abschnitt 4.2.3.2 genauer beschrieben). Die Abstraktionsfunktion α zwischen den beiden Verbänden ist in Gleichung 4.9 dargestellt:

$$\alpha(Z) = \begin{cases} \bot & \text{falls } Z = \emptyset \\ [\inf(Z), \sup(Z)] & \text{sonst} \end{cases} \tag{4.9}$$

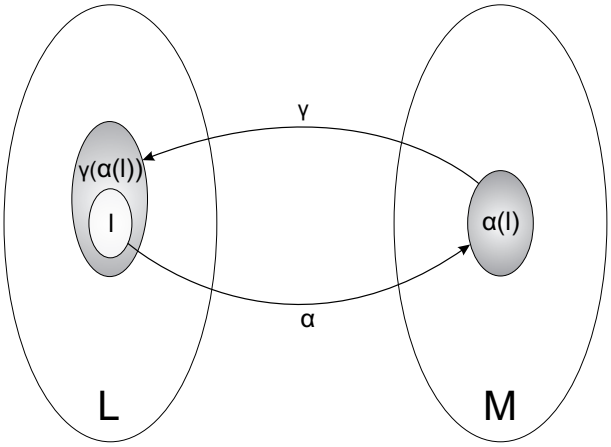


Abbildung 4.2: *Monotonie der Galois-Verbindung*

Falls die Menge $Z \subseteq \mathbb{Z}$ nicht leer ist, so wird das Intervall durch das kleinste (*Infimum*) sowie größte (*Supremum*) Element der Menge gebildet. Die passende Konkretisierungsfunktion γ ist in Gleichung 4.10 zu sehen:

$$\gamma(int) = \{z \in \mathbb{Z} \mid \inf(int) \leq z \leq \sup(int)\} \tag{4.10}$$

Die Menge setzt sich also aus allen ganzzahligen Werten zusammen, die zwischen dem Infimum und dem Supremum liegen.

Die mögliche Menge $l = \{3, 8, 9\}$ würde über die Abstraktionsfunktion α in das Intervall $m = [3, 9]$ transferiert. Wird auf dieses Intervall die Konkretisierungsfunktion γ aus Gleichung 4.10 angewendet, so entsteht die Menge $l' = \{3, 4, 5, 6, 7, 8, 9\}$. Wie durch die Monotonie gefordert, gilt also $l \subseteq l'$.

Der Beweis, dass die beiden vorgestellten Funktionen mit ihren vollständigen Verbänden eine Galois-Verbindung darstellen, kann in [NNH99] nachgelesen werden.

4.2.2 Galois-Einsetzung

Die Güte des späteren Ergebnisses hängt stark von der Wahl des abstrahierten Verbands *M* ab. Das Problem an der Galois-Verbindung ist, dass die Forderungen noch vergleichbar schwach sind. Die Funktion γ muss nicht zwingend injektiv sein, so dass es Elemente in *M* geben kann, die für die Approximation nicht relevant sind.

Um dieses Defizit auszugleichen, wurde die Galois-Einsetzung (*Galois Insertion*) eingeführt, die eine Konkretisierung der Galois-Verbindung darstellt.

Damit eine Galois-Verbindung eine Galois-Einsetzung ist, muss folgendes gelten:

1. (L, α, γ, M) ist eine Galois-Verbindung
2. α ist surjektiv: $\forall m \in M : \exists l \in L : \alpha(l) = m$.
3. γ ist injektiv: $\forall m_1, m_2 \in M : \gamma(m_1) = \gamma(m_2) \Rightarrow m_1 = m_2$.

Durch die Surjektivität der Abstraktionsfunktion α (siehe Punkt 2) ist sichergestellt, dass es für jedes Element aus *M* mindestens ein Element aus *L* gibt. Aus der Injektivität von Punkt 3 folgt, dass jeder Punkt aus *L* maximal einem Element aus *M* zugewiesen wird. Die Kombination dieser beiden Forderung stellt schließlich sicher, dass es für jeden Punkt der Menge *M* genau einen Punkt aus der Menge *L* gibt. Dadurch wird die Abstraktionsmenge *M* bei einer Galois-Einsetzung minimal gehalten, was die Güte der Approximation stark verbessert.

Das Beispiel aus Abschnitt 4.2.1.1, in dem die ganzzahlige Menge $\mathbb{Z}$ auf die Intervallrepräsentation abgebildet wurde, ist ein Beispiel für eine Galois-Einsetzung. Jedes Intervall aus der Menge *M* zeigt auf genau eine mögliche Menge aus $\mathbb{Z}$ (z.B. wird $[2, 4]$ immer auf $\{2, 3, 4\}$ aus $\mathbb{Z}$ abgebildet).

Ziel der Abstrakten Interpretation sollte es also im Folgenden sein, eine möglichst gute Galois-Einsetzung als Abstraktionsebene zu benutzen.

4.2.3 Abstrakte Domänen

Die im letzten Abschnitt vorgestellte Galois-Verbindung, bzw. Galois-Einsetzung, beschreibt die zu Grunde liegende Approximation der Abstrakten Interpretation. Dabei wird der neue Verband *M*, in dem die Berechnung durchgeführt werden soll, als Abstrakte Domäne bezeichnet.

In diesem Abschnitt sollen einige mögliche Abstrakte Domänen vorgestellt werden. Dazu gehören aus Abschnitt 4.2.3.1 die Vorzeichen-Domäne, sowie aus Abschnitt 4.2.3.2 die Intervall-Domäne. In Abschnitt 4.2.3.3 wird schließlich ein kurzer Überblick über weitere Abstrakte Domänen gegeben.

4.2.3.1 Vorzeichen-Domäne

Eine der einfachsten Abstrakten Domänen bildet die Vorzeichen-Domäne ($Sign$). Der Nutzen dieser Domäne bezüglich einer Schleifenanalyse ist sehr gering, doch auf Grund ihrer geringen Komplexität ist sie gut als erstes Beispiel geeignet. Ziel dieser Abstrakten Domäne ist es, den Wert einer Zahl auf ihr Vorzeichen zu beschränken. Dies erledigt die Funktion *sign(z)* aus Definition 4.11:

$$sign(z) = \begin{cases} - & \text{falls } z < 0 \\ o & \text{falls } z = 0 \\ + & \text{falls } z > 0 \end{cases} \tag{4.11}$$

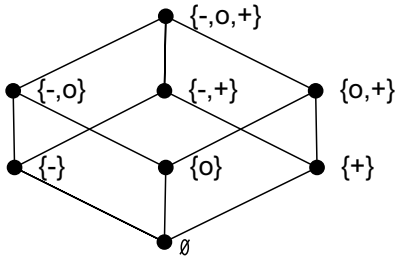


Abbildung 4.3: *Der vollständige Verband der Vorzeichen-Domäne (Sign)*

Mit Hilfe dieser Funktion ist es möglich, die Abstraktionsfunktion α für $\mathbb{Z} \rightarrow Sign$ in Gleichung 4.12 zu definieren:

$$\alpha_{sign}(Z) = \{sign(z) \mid z \in Z\} \tag{4.12}$$

Es werden also alle Zeichen in die Menge aufgenommen, die für die Zahlen der Menge Z gültig sind. Analog dazu ist die Konkretisierungsfunktion γ in Definition 4.13 dargestellt:

$$\gamma_{sign}(S) = \{z \in \mathbb{Z} \mid sign(z) \in S\} \tag{4.13}$$

In dieser Richtung werden der Zielmenge alle Zahlen aus $\mathbb{Z}$ zugewiesen, für die die Menge an Vorzeichen passend ist. Für die Gleichungen 4.12 und 4.13 gilt $Z \subseteq \mathbb{Z}$ und $S \subseteq Sign$.

Der vollständige Verband der Abstrakten Domäne $Sign$ ist in Abbildung 4.3 zu sehen.

Da es nur drei mögliche Elemente (sowie $\top$ und $\bot$) gibt, ist der Informationsverlust, der durch die Anwendung dieser Abstrakten Domäne entsteht, relativ groß. Wird z.B. die Menge $\{5, 8\} \in \mathbb{Z}$ durch α in den Verband von $Sign$ transferiert, so entsteht die Menge $\{+\}$. Eine anschließende Anwendung von γ liefert die Menge $\{1, .., \infty\}$.

4.2.3.2 Intervall-Domäne

Die Intervall-Domäne wurde bereits in Abschnitt 4.2.1.1 als Beispiel für eine Galois-Verbindung eingeführt. Die Abstraktionsfunktion α wurde dabei in Gleichung 4.9 definiert. Die Konkretisierungsfunktion γ ist in der Gleichung 4.10 wiederzufinden. Da die Intervall-Domäne die gewählte Abstrakte Domäne dieser Arbeit ist, soll diese nun im Detail vorgestellt werden.

Zu den Elementen der Intervall-Domäne gehören alle Intervalle, die sich aus der Menge $\mathbb{Z}$ bilden lassen, sowie das leere Intervall $\bot$. Dies ist in Gleichung 4.14 dargestellt:

$$\text{Interval} = \{\bot\} \cup \{[z_1, z_2] \mid z_1 \leq z_2, z_1 \in \mathbb{Z} \cup \{-\infty\}, z_2 \in \mathbb{Z} \cup \{\infty\}\} \tag{4.14}$$

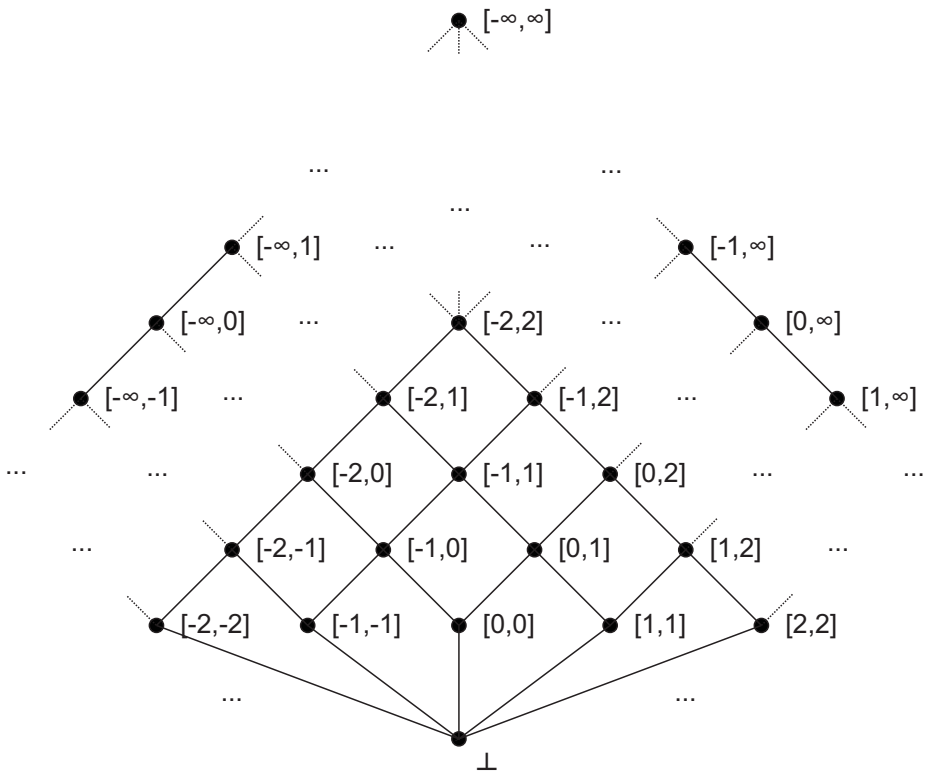


Abbildung 4.4: *Der Aufbau des Verbands der Intervall-Domäne*

Der vollständige Aufbau des Verbands der Intervall-Domäne ist in Abbildung 4.4 dargestellt. Dabei sind die Elemente einer unteren Ebene stets Teilmengen der oberen, wie es durch die Verbindungslinien angedeutet ist.

Um zu zeigen, dass die Intervall-Domäne ein vollständiger Verband ist, muss gezeigt werden, dass die Operatoren $\bot$, $\top$, $\sqsubseteq$, $\sqcap$ und $\sqcup$ für die Intervall-Domäne existieren.

Das Element $\bot$ stellt dabei, wie bereits erwähnt, das leere Intervall dar. Analog dazu bezeichnet das $\top$-Element das Element, das alle Teilintervalle beinhaltet. Dies entspricht dem Intervall $[-\infty, \infty]$.

Der $\sqsubseteq$-Operator gibt an, ob ein Intervall int_1 in einem Intervall int_2 enthalten ist. Dies gilt genau dann, wenn das Infimum von int_1 größer oder gleich dem Infimum von int_2 ist. Zusätzlich muss gelten, dass das Supremum von int_1 kleiner oder gleich dem Supremum von int_2 ist. Dies ist in Gleichung 4.15 zu sehen:

$$int_1 \sqsubseteq int_2 \text{ falls } (\inf(int_2) \leq \inf(int_1)) \wedge (\sup(int_1) \leq \sup(int_2)) \tag{4.15}$$

Der Vereinigungsoperator $\sqcup$ ist in Gleichung 4.16 definiert:

$$\bigsqcup Y = \begin{cases} \bot & \text{falls } Y \subseteq \{\bot\} \\ [\inf'\{\inf(int) \mid int \in Y\}, \sup'\{\sup(int) \mid int \in Y\}] & \text{sonst} \end{cases} \tag{4.16}$$

Falls eines der zu vereinigenden Elemente das $\bot$ Element ist, so wird das Zielintervall ebenfalls auf $\bot$ gesetzt, da keine Aussage mehr über das neue Intervall getroffen werden kann, sobald eines der zu vereinigenden Intervalle unbekannt ist. Ist dies nicht der Fall, so wird das kleinste Infimum der zu vereinigenden Mengen als neues kleinstes Element gewählt. Analog dazu bildet das größte Element der ursprünglichen Mengen die obere Grenze des neuen Intervalls.

Der Schnittoperator $\sqcap$ ist in Gleichung 4.17 definiert:

$$\bigsqcap Y = \begin{cases} \bot & \text{falls } Y \subseteq \{\bot\} \vee \exists y_1, y_2 \in Y : \sup(y_1) < \inf(y_2) \\ [\sup'\{\inf(int) \mid int \in Y\}, & \\ \inf'\{\sup(int) \mid int \in Y\}] & \text{sonst} \end{cases} \tag{4.17}$$

Falls eines der zu schneidenden Elemente $\bot$ ist, oder falls es zwei Intervalle gibt, bei dem die obere Grenze kleiner als die untere Grenze eines anderen Intervalls ist, so ist der Schnitt der Intervalle gleich der Menge $\{\bot\}$. Andernfalls wird die untere Grenze des neuen Intervalls gleich der maximalen unteren Grenze der zu schneidenden Intervalle sein. Die obere Grenze wird dadurch bestimmt, dass das kleinste Element aller oberen Grenzen gewählt wird.

Da für die Ordnung der Intervalle alle benötigten Operatoren definiert worden sind, wurde gezeigt, dass die Intervall-Domäne eine Galois-Verbindung ist. Insbesondere ist es eine Galois-Einsetzung, wie es in Kapitel 4.2.2 bereits erwähnt wurde.

4.2.3.3 Weitere Abstrakte Domänen

Eine weitere sehr verbreitete Abstrakte Domäne ist die *Kongruenz-Domäne* [Gra89]. Das Problem der Intervall-Domäne besteht darin, dass bei einem Join aus zwei Teilintervallen, die sehr weit auseinander liegen, eine starke Überapproximation auftritt. So werden durch die Operation $[1,1] \cup [1000,1000] = [1,1000]$ 998 überflüssige Objekte zum neuen Intervall hinzugefügt.

Dieser Umstand soll durch die Kongruenz-Domäne umgangen werden. Dort werden zu jeder Menge zwei Zahlen a und b festgelegt, die die Untermenge aus $a\mathbb{Z} + b \subseteq \mathbb{Z}$ bilden. Es werden also grundsätzlich Vielfache einer möglichen Zahl gespeichert. So würde die Menge $\{3,9\} \subseteq \mathbb{Z}$ durch $3 * \mathbb{Z} + 3$ dargestellt.

Problematisch ist diese Darstellung vor allem dann, wenn die Elemente sehr dicht zusammen liegen. So wird die Menge $\{1,2\} \subseteq \mathbb{Z}$ durch $1 * \mathbb{Z} + 1$ dargestellt, wodurch schließlich alle Elemente aus $\mathbb{Z}$

in der neuen Menge enthalten sind.

Des Weiteren gibt es diverse Ansätze, die die Grenzen der Intervalle nicht nur zweidimensional darstellen sollen. Dazu gehört u.a. die *Polyhedra-Domäne* [CH78]. Diese speichert für jede Variable ein lineares Gleichungssystem, dessen Lösung die möglichen Werte der Variable repräsentiert. Der Nachteil dieser relativ genauen Domäne ist, dass sie extrem rechenaufwändig ist.

Die Komposition von Galois-Verbindungen ist ebenfalls eine Möglichkeit, neue Galois-Verbindungen, bzw. Abstrakte Domänen, zu erzeugen. Daher können teilweise auch interessante Domänen entstehen, indem verschiedene Ansätze zu *hybriden Domänen* kombiniert werden.

4.2.4 Abstrakte Operatoren

Bisher wurde lediglich beschrieben, wie sich eine Abstrakte Domäne bilden lässt. Dabei wurde noch nicht darauf eingegangen, wie Berechnungen auf diesen Domänen durchgeführt werden können. Ein normales Programm modifiziert die Werte seiner Variablen mit Hilfe von ein- oder mehrstelligen Operatoren. Abhängig von der Abstrakten Domäne sind diese jeweils separat zu definieren. Wie dies funktioniert, soll beispielhaft an der Abstrakten Domäne der Intervalle gezeigt werden.

4.2.4.1 Anforderungen

Vorerst soll allerdings definiert werden, welche Eigenschaft ein Abstrakter Operator haben muss, damit die Abstrakte Interpretation ein korrektes Ergebnis liefert. Die Funktion $f : \mathcal{V}^n \rightarrow \mathcal{V}$ stellt einen n-dimensionalen Operator einer Programmiersprache dar. Anhand dessen kann die folgende Definition gemacht werden:

$$f_{\mathcal{P}}(M) = \{f(z^n) \mid z^n \in M\} \tag{4.18}$$

Die resultierende Menge $f_{\mathcal{P}}(M)$ beinhaltet also alle Elemente, die nach Anwendung der Funktion *f* auf die Menge *M* entstanden sind. Analog zur Funktion *f* kann eine Funktion $\hat{f}$ definiert werden, die die gleiche Operation auf der Abstrakten Domäne durchführt. Auf Grund der Forderung nach Monotonie für die Abstrakte Interpretation, muss auch für einen Abstrakten Operator gelten:

$$f_{\mathcal{P}}(M) \sqsubseteq \gamma \circ \hat{f} \circ \alpha(M) \tag{4.19}$$

Nachdem eine Menge *M* durch die Abstraktionsfunktion α in die Abstrakte Domäne transferiert wurde, wird der entsprechende Operator $\hat{f}$ auf die neue Menge angewendet. Anschließend wird diese neue Menge wieder durch die Konkretisierungsfunktion γ in den ursprünglichen Verband abgebildet. Die daraus resultierende Menge muss eine Teilmenge der Menge sein, die durch einfache Anwendung aus der Funktion $f_{\mathcal{P}}(M)$ entstanden wäre.

4.2.4.2 Beispiel

Abstrakte Operatoren sind also die Operatoren, die für den Verband der Abstrakten Domäne definiert werden. Dies soll in diesem Abschnitt anhand der Intervall-Domäne beispielhaft gezeigt werden. Im Folgenden sind die vier wichtigsten arithmetischen Operatoren für die Intervall-Domäne aufgeführt:

$$A\hat{+}B = [\inf(A) + \inf(B), \sup(A) + \sup(B)]$$

$$A\hat{-}B = [\inf(A) - \sup(B), \sup(A) - \inf(B)]$$

$$A\hat{*}B = [\inf(A) * \inf(B), \sup(A) * \sup(B)]$$

$$A\hat{/}B = [\inf(A)/\sup(B), \sup(A)/\inf(B)]$$

Analog zu den arithmetischen Operatoren können auch boolsche Operatoren definiert werden:

$$A\hat{\leq}B = \begin{cases} \bot & \text{falls } A \sqcup B \sqsubseteq \bot \\ TRUE & \text{falls } \sup'\{\sup(int) \mid int \in A\} \leq \inf'\{\inf(int) \mid int \in B\} \\ FALSE & \text{falls } \inf'\{\inf(int) \mid int \in A\} > \sup'\{\sup(int) \mid int \in B\} \\ \top & \text{sonst} \end{cases}$$

Der $\bot$-Operator bedeutet in diesem Fall, dass der Wert nicht definiert werden kann, da das $\bot$-Element in den Mengen A oder B enthalten ist. Falls sich die Intervalle überschneiden, so kann ebenfalls keine Aussage über den Wahrheitswert gemacht werden. Dies wird durch den $\top$-Operator dargestellt. Für alle anderen Fälle kann jedoch ausgewertet werden, ob die Aussage immer wahr oder immer falsch ist. Die anderen boolschen Operatoren können nach dieser Vorlage gebildet werden.

Mit Hilfe der Abstrakten Operatoren ist es nun möglich, in dem abstrahierten Verband *M* Berechnungen durchzuführen. Wie das Transitionssystem anhand des Kontrollflussgraphen aufgebaut wird, wird im nächsten Abschnitt gezeigt.

4.2.5 Aufbau des Transitionssystems

Wie bereits erwähnt, wird das Transitionssystem der Abstrakten Interpretation auf Grundlage des Kontrollflussgraphen erstellt. Dabei gilt für jeden Programmpunkt *q*, dass ihm eine Menge an möglichen Zuständen $STATES_q$ zugeordnet ist. Für die Menge aller Programmpunkte *Q* existiert dabei eine Funktion *pre*, die für jeden Programmpunkt die Menge seiner direkten Vorgänger (*predecessor*) ermittelt. Dies ist in Definition 4.20 festgelegt:

$$\text{pre: } Q \to \mathcal{P}(Q) \tag{4.20}$$

Analog zu der Menge der Vorgänger existiert die Menge der Nachfolger *suc* (*successors*), wie in Definition 4.21 zu sehen:

$$\text{suc: } Q \to \mathcal{P}(Q) \tag{4.21}$$

Je nach Typ des Programmpunkts liefern die Abbildungen unterschiedlich große Mengen. Für den Bedingungs-Knoten ist $|pre| = 1$ und $|suc| = 2$. Die Join-Knoten, die den Kontrollfluss wieder zusammenführen, haben jeweils zwei Vorgänger und einen Nachfolger. Für alle anderen Knoten kann die Anzahl an Vorgängern und Nachfolgern aus Abbildung 4.1, anhand der ein- und ausgehenden Kanten, entnommen werden.

Im Folgenden soll für die 5 möglichen Knoten eines Kontrollflussgraphen (siehe Abschnitt 4.1.1) beschrieben werden, welche Regeln durch diese zum Transitionssystem hinzugefügt werden. Dabei gilt, dass $\hat{S}$ eine Menge an abstrakten Zuständen darstellt. Die Menge $\hat{S}'$ ist die Menge, die nach Ausführung des Transitionssystems τ auf $\hat{S}$ entsteht. Analog dazu existieren die Mengen $\hat{S}_q$ und $\hat{S}'_q$, die die Menge an möglichen Zuständen für den Programmpunkt *q* darstellen.

1. *Start-Knoten*

   Der Start-Knoten ist der Eintrittspunkt eines jeden Programms. Ziel der Abstrakten Interpretation ist es, bei der Berechnung jede mögliche Eingabekombination zu berücksichtigen. Daher gilt, dass für jede Variable vorerst angenommen wird, dass sie jeden möglichen Wert, abhängig vom Variablentyp, annehmen kann. Dies kann durch den $\top$-Operator ausgedrückt werden. Daher gilt für den Start-Knoten *q* die Gleichung 4.22:

   $$\hat{S}'_q = \top \tag{4.22}$$

2. *Ausdrucks-Knoten*

   Der Ausdrucks-Knoten steht für die Anwendung eines Operators auf eine Variable. Dazu gehört z.B. der Zuweisungsoperator. Wie in Abschnitt 4.2.4 gesehen, ist es möglich, sowohl die Werte als auch die Operatoren $\oplus$ durch ihre abstrakten Versionen $\hat{\oplus}$ zu ersetzen. Eine solche Operation wird also vollständig im abstrahierten Verband durchgeführt.

   Für einen Ausdrucks-Knoten *q*, der eine Zuweisung der Form $x = E$ repräsentiert, gilt die Gleichung 4.23:

   $$\hat{S}'_q = \hat{S}_{pre_q}[x \mapsto \hat{S}_{pre_q}(E)] \tag{4.23}$$

   Der neue Zustand des Programmpunkts *q* wird also dadurch gebildet, dass im Zustand des

Vorgängers ($\hat{S}_{pre_q}$) der Wert von *x* durch den Wert von *E* ersetzt wird, der im Zustand des Vorgängers gültig war.

3. *Bedingungs-Knoten*

   Ähnlich wie bei den Ausdrucks-Knoten, wird die Berechnung der Bedingungs-Knoten im abstrahierten Verband durchgeführt. Der vollständige boolsche-Ausdruck wird mit allen Operatoren und Werten in den abstrahierten Verband übertragen, um die Bedingung auszuwerten.

   Im Gegensatz zu den bisher vorgestellten Knoten besitzt der Bedingungs-Knoten zwei Nachfolger. Einer besteht aus den Werten, die sich ergeben, wenn die Bedingung erfüllt wird ($\hat{S'}_{q_{true}}$). Der andere beinhaltet alle Werte, für die die Bedingung nicht erfüllt ist ($\hat{S'}_{q_{false}}$). In Gleichung 4.24 wird dargestellt, wie sich die Werte für den Zustand $\hat{S'}_{q_{true}}$ ergeben:

$$\hat{S'}_{q_{true}} = \bigsqcup \left\{ s | s \sqsubseteq \hat{S}_{pre_q} \wedge \widehat{cond}(s) \sqsubseteq True \right\} \tag{4.24}$$

   Die Funktion $\widehat{cond}(s)$ transferiert dabei die Bedingung in den abstrahierten Verband. Der neue Zustand setzt sich aus allen Umgebungen des Vorgängers zusammen, für die die Bedingung erfüllt ist.

   Analog dazu existiert eine Gleichung für den Programmpunkt $\hat{S'}_{q_{false}}$:

$$\hat{S'}_{q_{false}} = \bigsqcup \left\{ s | s \sqsubseteq \hat{S}_{pre_q} \wedge \widehat{cond}(s) \sqsubseteq False \right\} \tag{4.25}$$

4. *Join-Knoten*

   Der Join-Knoten ist das Gegenstück zum Bedingungs-Knoten. Er führt vorher verzweigte Programmflusspfade wieder zusammen. Die Werte innerhalb des Zustands werden durch einen Join zusammengefasst. Dies ist in Gleichung 4.26 zu sehen:

$$\hat{S'}_q = \bigsqcup \hat{S}_{pre(q)} \tag{4.26}$$

Aus den hier vorgestellten Transitionsregeln kann aus einem Kontrollflussgraphen das Transitionssystem τ erstellt werden. Wie in Abschnitt 4.1.3 beschrieben, wird im Anschluss der Versuch unternommen den kleinsten Fixpunkt des Transitionssystems τ zu bestimmen.

4.2.6 Beispiel

Nachdem nun die Abstrakte Interpretation soweit beschrieben ist, dass alle wichtigen Bestandteile eingeführt worden sind, kann eine vollständige Berechnung durchgeführt werden. Dies soll beispielhaft an dem Programm geschehen, dessen Kontrollflussgraph in Abbildung 4.5 dargestellt ist.

Um das Beispiel zu berechnen, wird zuerst das Transitionssystem τ anhand des Kontrollflussgraphen gebildet. Anschließend wird versucht, den kleinsten Fixpunkt des Transitionssystem, auf Grundlage der Intervall-Domäne, zu finden.

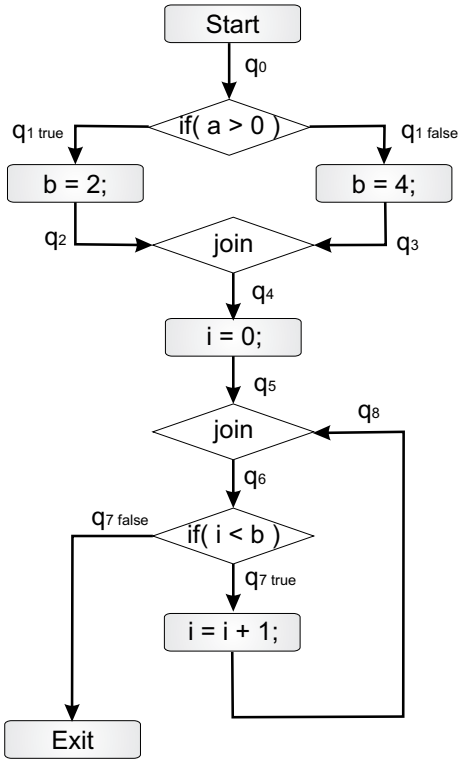


Abbildung 4.5: *Kontrollflussgraph des Beispiels zur Abstrakten Interpretation*

Anhand des Regelwerks, das im letzten Abschnitt beschrieben worden ist, wird das folgende Transitionssystem gebildet:

$$\hat{S}_{q0} = [b \to \top, i \to \top]$$
$$\hat{S}_{q1_{true}} = \bigsqcup \left\{ s | s \sqsubseteq \hat{S}_{q0} \wedge \hat{S}_{q0}(a) \hat{>} \alpha(0) \sqsubseteq True \right\}$$
$$\hat{S}_{q1_{false}} = \bigsqcup \left\{ s | s \sqsubseteq \hat{S}_{q0} \wedge \hat{S}_{q0}(a) \hat{>} \alpha(0) \sqsubseteq False \right\}$$
$$\hat{S}_{q2} = \hat{S}_{q1_{true}}[b \to \alpha(2)] = \hat{S}_{q1_{true}}[b \to [2,2]]$$
$$\hat{S}_{q3} = \hat{S}_{q1_{false}}[b \to \alpha(4)] = \hat{S}_{q1_{false}}[b \to [4,4]]$$
$$\hat{S}_{q4} = \hat{S}_{q2} \sqcup \hat{S}_{q3}$$
$$\hat{S}_{q5} = \hat{S}_{q4}[i \to \alpha(0)] = \hat{S}_{q4}[i \to [0,0]]$$
$$\hat{S}_{q6} = \hat{S}_{q5} \sqcup \hat{S}_{q8}$$
$$\hat{S}_{q7_{true}} = \bigsqcup \left\{ s | s \sqsubseteq \hat{S}_{q6} \wedge \hat{S}_{q6}(i) \hat{<} \hat{S}_{q6}(b) \sqsubseteq True \right\}$$
$$\hat{S}_{q7_{false}} = \bigsqcup \left\{ s | s \sqsubseteq \hat{S}_{q6} \wedge \hat{S}_{q6}(i) \hat{<} \hat{S}_{q6}(b) \sqsubseteq False \right\}$$
$$\hat{S}_{q8} = \hat{S}_{q7_{true}}[i \to \hat{S}_{q7_{true}}(i) \hat{+} \alpha(1)] = \hat{S}_{q7_{true}}[i \to \hat{S}_{q7_{true}}(i) \hat{+} [1,1]]$$

Gestartet wird die Berechnung dadurch, dass jeder Zustand mit $\perp$ initialisiert wird. Eine Regel wird genau dann angewendet, wenn sich einer seiner abhängigen Zustände geändert hat. Nach der Initialisierung ist grundsätzlich die Regel des Start-Knotens anwendbar, da diese als einzige keine

abhängigen Zustände beinhaltet. In der Folge der Berechnung werden nun alle Zustände aktualisiert, die vom Zustand des Start-Knotens abhängig sind. Diese Aktualisierung wird nun so lange wiederholt, bis sich kein Zustand mehr ändert, wodurch der kleinste Fixpunkt gefunden ist.

Die Berechnung für das aktuelle Beispiel ist in Tabelle 4.1 zu sehen, wobei einige Zustände ausgeblendet sind, um die Berechnung übersichtlicher zu gestalten:

Zustand	q_0		q_4		q_6		$q_{7_{true}}$		$q_{7_{false}}$	
Variable	b	i	b	i	b	i	b	i	b	i
Init	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
Iter 0	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
Iter 1	$\top$	$\top$	$[2,4]$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
Iter 2	$\top$	$\top$	$[2,4]$	$\top$	$[2,4]$	$[0,0]$	$\bot$	$\bot$	$\bot$	$\bot$
Iter 3	$\top$	$\top$	$[2,4]$	$\top$	$[2,4]$	$[0,0]$	$[2,4]$	$[0,0]$	$\bot$	$\bot$
Iter 4	$\top$	$\top$	$[2,4]$	$\top$	$[2,4]$	$[0,1]$	$[2,4]$	$[0,0]$	$\bot$	$\bot$
Iter 5	$\top$	$\top$	$[2,4]$	$\top$	$[2,4]$	$[0,1]$	$[2,4]$	$[0,1]$	$\bot$	$\bot$
Iter 6	$\top$	$\top$	$[2,4]$	$\top$	$[2,4]$	$[0,2]$	$[2,4]$	$[0,1]$	$\bot$	$\bot$
Iter 7	$\top$	$\top$	$[2,4]$	$\top$	$[2,4]$	$[0,2]$	$[2,4]$	$[0,2]$	$[2,2]$	$[2,2]$
Iter 8	$\top$	$\top$	$[2,4]$	$\top$	$[2,4]$	$[0,3]$	$[2,4]$	$[0,2]$	$[2,2]$	$[2,2]$
Iter 9	$\top$	$\top$	$[2,4]$	$\top$	$[2,4]$	$[0,3]$	$[2,4]$	$[0,3]$	$[2,3]$	$[2,3]$
Iter 10	$\top$	$\top$	$[2,4]$	$\top$	$[2,4]$	$[0,4]$	$[2,4]$	$[0,3]$	$[2,3]$	$[2,3]$
Iter 11	$\top$	$\top$	$[2,4]$	$\top$	$[2,4]$	$[0,4]$	$[2,4]$	$[0,3]$	$[2,4]$	$[2,4]$
Iter 12	$\top$	$\top$	$[2,4]$	$\top$	$[2,4]$	$[0,4]$	$[2,4]$	$[0,3]$	$[2,4]$	$[2,4]$

Tabelle 4.1: *Fixpunktberechnung des Beispiels zur Abstrakten Interpretation*

Nach der 11. Iteration stabilisiert sich das Transitionssystem, da alle Zustände in der 11. und 12. Iteration identisch sind. Der Fixpunkt ist also berechnet und die Analyse der Abstrakten Interpretation ist beendet. In der letzten Zeile der Tabelle stehen nun alle Werte, die die Variablen in den entsprechenden Zuständen annehmen können.

4.2.7 Widening- und Narrowing-Operator

Das Problem der Abstrakten Interpretation ist, dass nicht garantiert werden kann, dass die Analyse terminiert. Bereits bei einer einfachen Modifikation des oberen Beispiels (siehe Abbildung 4.6) wird die Abstrakte Interpretation nicht mehr terminieren. Durch die Erweiterung des Basisblocks $i = 0$ durch eine weitere Initialisierung $c = 0$, sowie durch die Erweiterung des Basisblocks $i = i+1$ durch $c = c+1$, wird erreicht, dass die Abstrakte Interpretation nicht mehr terminiert. Die Bedingung $i < b$ beinhaltet keine Einschränkung für die Variable c, so dass diese bei jeder Iteration weiter anwächst und die Fixpunktberechnung dadurch nicht terminiert.

Ein weiteres Problem der Abstrakten Interpretation ist, dass eine Analyse, abhängig vom Programm, sehr zeitaufwendig sein kann. Wie in dem oberen Beispiel gezeigt, wird jeder Schritt der Schleife iterativ simuliert. Das heißt, dass die Analyse bei einer Schleife, die eine hohe Iterationsanzahl hat, auch sehr lange dauern kann.

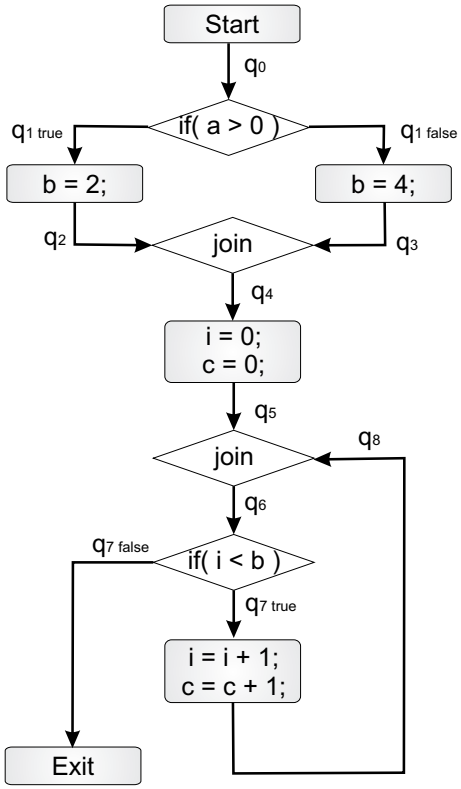


Abbildung 4.6: *Kontrollflussgraph des modifizierten Beispiels zur Abstrakten Interpretation*

Beide Probleme können durch die Einführung eines *Widening-Operators* gelöst werden. Der Grundgedanke des Widening-Operators ist, dass eine unendliche Sequenz an Zuständen ($(\hat{S}^i)^{i\in\mathbb{N}}$) eines Programmpunkts q durch eine endliche, sicher abgeschätzte Sequenz $(\hat{S}^m_\nabla)^{m\in\mathbb{N}}$) ersetzt wird. Dies wird durch den Widening-Operator $\nabla : L \times L \to L$ erreicht, der auf dem vollständigen Verband L definiert wird.

Die neue Sequenz wird durch die Funktion f^n_∇ wie folgt berechnet:

$$f^n_\nabla = \begin{cases} \bot & \text{falls } n = 0 \\ f^{n-1}_\nabla & \text{falls } n > 0 \wedge f(f^{n-1}_\nabla) \sqsubseteq f^{n-1}_\nabla \\ f^{n-1}_\nabla \nabla f(f^{n-1}_\nabla) & \text{sonst} \end{cases} \tag{4.27}$$

Falls die Iterationsanzahl n der Sequenz gleich 0 ist, startet die Berechnung, wie auch bei der normalen Abstrakten Interpretation ohne Widening-Operator, mit dem Wert $\bot$ für alle Variablen in allen Zuständen. Für den Fall, dass bereits Iterationen stattgefunden haben und die erneute Anwendung der Funktion f eine Teilmenge der ursprünglichen Menge liefert, so ist ein Fixpunkt erreicht und die alte Menge bleibt bestehen. Die Approximation tritt im dritten Fall ein, bei dem der eigentliche Widening-Operator auf dem letzten und dem neu berechneten Wert angewendet wird.

Der Widening-Operator kann je nach Anforderung (z.B. Optimierung der Laufzeit, oder Optimierung

des Ergebnisses) unterschiedlich genau definiert werden. Ein möglicher Operator für die Intervalldomäne wurde in [NNH99] beschrieben. Dort wird der Operator ∇ für die beiden Intervalle int_1 und int_2 wie folgt definiert:

$$int_1 \nabla int_2 = \begin{cases} \bot & \text{falls } int_1 = int_2 = \bot \\ [LB_k(\inf(int_1), \inf(int_2)), & \\ UB_k(\sup(int_1), \sup(int_2))] & \text{sonst} \end{cases} \tag{4.28}$$

Haben beide Intervalle den Wert $\bot$ zugewiesen bekommen, so ist auch der neue Wert der approximativen Verschmelzung gleich $\bot$. Andernfalls wird ein neues Intervall gebildet, das die Funktionen LB_k und UB_k mit den beiden Infima, bzw. Suprema, aufruft. Die beiden Funktionen sind in den Definitionen 4.29 und 4.30 abgebildet:

$$LB_k(z_1, z_3) = \begin{cases} z_1 & \text{falls } z_1 \leq z_3 \\ k & \text{falls } z_3 < z_1 \wedge k = \max\{k \in K \mid k \leq z_3\} \\ -\infty & \text{falls } z_3 < z_1 \wedge \forall k \in K : z_3 < k \end{cases} \tag{4.29}$$

$$UB_k(z_2, z_4) = \begin{cases} z_2 & \text{falls } z_4 \leq z_2 \\ k & \text{falls } z_2 < z_4 \wedge k = \min\{k \in K \mid z_4 \leq k\} \\ \infty & \text{falls } z_2 < z_4 \wedge \forall k \in K : k < z_4 \end{cases} \tag{4.30}$$

Die Menge K ist eine endliche Menge an ganzzahligen Werten, über die die Genauigkeit der Approximation durch den Widening-Operator gesteuert werden kann. Die Gleichung LB_k, die die neue untere Grenze des Intervalls bestimmt, ist wie folgt zu verstehen: Ist der kleinste Wert des alten Intervalls kleiner als der des neuen, so wird der alte Wert übernommen. Ist jedoch der neu berechnete Wert kleiner als der ursprüngliche, so wird das größte Element der Menge K gesucht, das kleiner oder gleich dem kleinsten Element des neuen Intervalls ist. Wird ein solcher Wert $k \in K$ gefunden, so bildet dieser die neue, untere Grenze. Existiert dieser Wert nicht, so wird der neue kleinste Wert des Intervalls auf $-\infty$ gesetzt. Die Funktion UB_k ist analog dazu definiert.

Dieses Vorgehen soll anhand des oben genannten modifizierten Beispiels aus Abbildung 4.6 verdeutlicht werden, bei dem die Abstrakte Interpretation ohne Anwendung eines geeigneten Widening-Operators nicht terminieren würde. Die Variable c des Programmpunkts q_6 durchläuft während der Suche nach dem kleinsten Fixpunkt folgende Sequenz:

$$[0,0], [0,1], [0,2], [0,3], [0,4], [0,5], [0,6], [0,7], [0,8], [0,9], ..., [0,\infty]$$

Durch Anwendung des beschriebenen Widening-Operators aus Definition 4.28 mit der Menge $K = \{0, 1, 2, 3, 4, 5, 10, 100\}$, sieht die neue Sequenz wie folgt aus:

$$[0,0], [0,1], [0,2], [0,3], [0,4], [0,5], [0,10], [0,100], [0,\infty]$$

Auf diese Weise stabilisiert sich der Wert des Programmpunkts q_6 innerhalb von wenigen Iterationen. Dies kann entsprechend dadurch verkürzt werden, dass die Menge K weniger Elemente enthält, wodurch die Präzision des Ergebnisses im Allgemeinen durch diese Maßnahme jedoch deutlich schlechter wird.

Für das genannte Beispiel wurde über den Widening Operator schließlich nach einigen Iterationen für die Variable c im Programmpunkt q_6 das Intervall $[0, \infty]$ bestimmt. Dieses Ergebnis ist in der Praxis jedoch unbrauchbar. Daher ist analog zum Widening-Operator ein Narrowing-Operators definiert, der im Anschluss versucht die möglichen Werte der überschätzten Variablen nachträglich wieder einzuschränken. Eine detaillierte Beschreibung kann z.B. in [NNH99] gefunden werden.

4.3 Die modifizierte Abstrakte Interpretation dieser Arbeit

Die Abstrakte Interpretation eignet sich sehr gut, um die möglichen Werte der Variablen zu bestimmen, um diese für die Bestimmung der Schleifengrenzen nutzen zu können. Die klassische Variante, die bisher vorgestellt worden ist, hat jedoch einige Nachteile, so dass in dieser Arbeit eine modifizierte Variante umgesetzt worden ist.

Ein großer Nachteil der klassischen Abstrakten Interpretation ist im letzten Abschnitt beschrieben worden. Um eine Terminierung eines Programms fordern zu können, muss grundsätzlich eine weitere Approximation durch einen Widening-Operator durchgeführt werden. Selbst wenn dieser so exakt wie möglich definiert ist, kann für die Variable c des letzten Beispiels nur das Intervall $[0, \infty]$ ermittelt werden. Diese Angabe ist für eine Schleifenanalyse nicht brauchbar, falls es eine Abbruchbedingung gibt, die von der Variable c abhängig ist.

Des Weiteren wurde in dieser Arbeit, zusätzlich zur Abstrakten Interpretation, eine statische Schleifenanalyse entwickelt. Diese soll, falls es möglich ist, die Schleife während der Fixpunktberechnung analysieren, um das iterative Verhalten der Abstrakten Interpretation zu umgehen. Falls die statische Analyse erfolgreich ist und alle Werte, sowie die Schleifengrenzen, bestimmt werden konnten, so soll dieser Bereich nicht noch zusätzlich durch die Abstrakte Interpretation ausgewertet werden. Da die klassische Variante der Abstrakten Interpretation allerdings vor der Fixpunktberechnung ein statisches Transitionssystem aufbauen muss, kann dieses nicht während der laufenden Fixpunktberechnung modifiziert werden.

Daher soll im folgenden Abschnitt beschrieben werden, wie die Analyse dieser Arbeit genau aufgebaut ist.

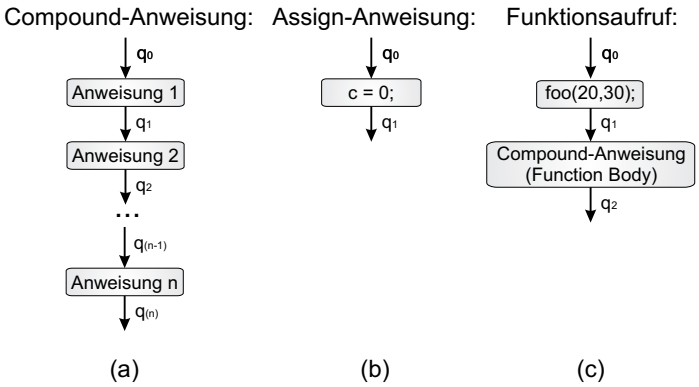


Abbildung 4.7: *Modifizierte Abstrakte Interpretation - Allgemeine Anweisungen*

4.3.1 Aufbau der modifizierten Abstrakten Interpretation

Die modifizierte Abstrakte Interpretation ist sehr nah am Kontrollflussgraphen orientiert. Der Hauptunterschied zur klassischen Abstrakten Interpretation besteht darin, dass am Anfang der Analyse kein Transitionssystem aufgebaut wird. Die Analyse startet, wie auch in der klassischen Variante, bei dem Einsprungpunkt des zu analysierenden Programms. Jedem Befehlstyp des Programms, bzw. Knotentyp des Kontrollflussgraphen, ist eine Regel zugeordnet, die in diesem Fall angewendet wird. Nach diesem Regelkatalog wird nun iterativ versucht, das Programm vollständig zu analysieren. Die hier umgesetzte Variante der Abstrakten Interpretation kann auch als eine Art der Simulation angesehen werden, die auf Grundlage des Regelwerks der Abstrakten Interpretation die möglichen Werte berechnet.

Der Aufbau der modifizierten Abstrakten Interpretation ist dabei an den Aufbau der Anweisungen der ICD-C Datenstruktur (siehe Abschnitt 3.1.1) angelehnt. Für jede Anweisung, die während der Analyse gefunden wird, wird der zugehörige Anweisungstyp ermittelt, um die passende Regel anzuwenden. Jede dieser Anweisungen, und deshalb auch jede Regel, kann wiederum aus diversen anderen Anweisungen zusammengesetzt werden, so dass eine hierarchisch aufgebaute Analyse durchgeführt wird. Wie die wichtigsten Anweisungen umgesetzt wurden, wird in der folgenden Aufzählung beschrieben:

- *Compound-Anweisung* (siehe Abbildung 4.7 (a))

  Eine Compound-Anweisung ist eine Anweisung, die aus einer Sequenz von Anweisungen besteht, die in der Hierarchie untergeordnet sind. Die Anweisungen werden nacheinander abgearbeitet, wobei jede Bearbeitung den zurückgegebenen Zustand des vorherigen Befehls als Eingabe erhält. Der erste Befehl bekommt den Zustand $\hat{S}_{q_0}$ als Eingabe, mit der die Analyse der Compound-Anweisung gestartet wurde. Die Rückgabe der Bearbeitung des letzten Befehls (q_n) ist der Zustand, der als Ergebnis der Bearbeitung der Compound-Anweisung zurückgegeben wird.

- *Assign-Anweisung* (siehe Abbildung 4.7 (b))

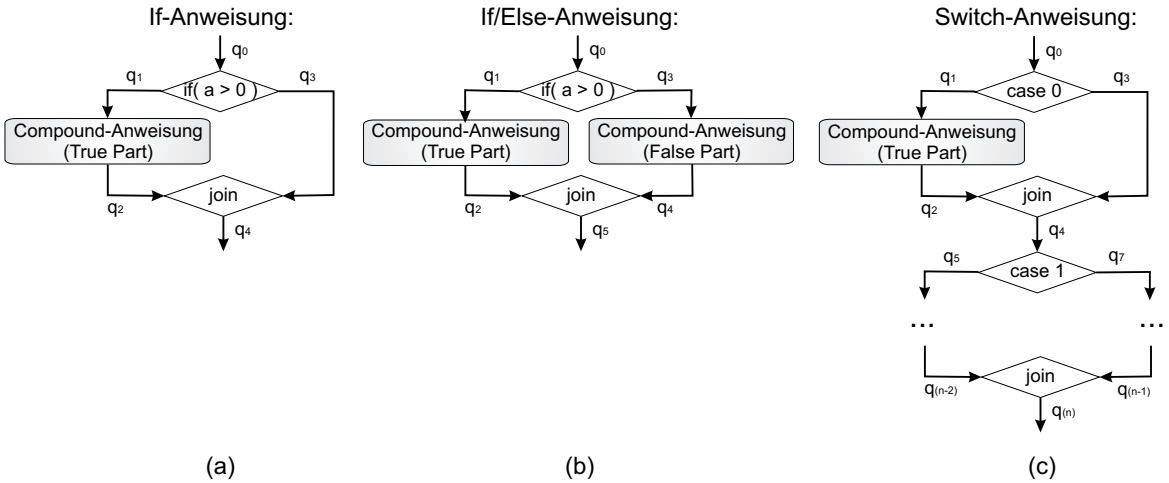


Abbildung 4.8: *Modifizierte Abstrakte Interpretation - Bedingte Verzweigungen*

Die Assign-Anweisung steht für alle Variationen, mit denen eine Zuweisung möglich ist. Zu diesen gehören u.a. die Operatoren $=$, $+=$ oder auch $<<=$. Wenn eine Assign-Anweisung während der Analyse gefunden wird, so werden, wie in der Abstrakten Interpretation vorgeschrieben, alle Werte der Variablen des Zustands $\hat{S}_{q_0}$ übernommen, wobei die Zuweisung den Wert der Variablen auf der linken Seite modifiziert. Diese Modifikation wird auf der Ebene der Abstrakten Domäne der Intervalle durchgeführt. Der neue Zustand, der durch diese Operation entsteht, heißt $\hat{S}_{q_1}$ und wird als Ergebnis dieses Befehls zurückgegeben.

- *Funktionsaufruf* (siehe Abbildung 4.7 (c))

  Der Startzustand $\hat{S}_{q_0}$ eines Funktionsaufrufs wird um die Zuweisung der Parameter zu den übergebenen Argumenten modifiziert. Der daraus resultierende Zustand $\hat{S}_{q_1}$ wird als Eingabe der Auswertung der Compound-Anweisung des Funktionsrumpfs genommen. Sobald die Auswertung des Funktionsrumpfes beendet ist, wird der ermittelte Zustand $\hat{S}_{q_2}$ als Ergebnis des Funktionsaufrufs zurückgegeben.

- *If-Anweisung* (siehe Abbildung 4.8 (a))

  Eine If-Anweisung steht für eine bedingte Verzweigung. Die Bedingung, die zur Verzweigung gehört, wird auf Ebene der Abstrakten Domäne ausgewertet. Zum Zustand $\hat{S}_{q_1}$ werden alle Werte von $\hat{S}_{q_0}$ hinzugefügt, die die Bedingung erfüllen. Werte von Variablen, die in der Bedingung nicht vorkommen, werden uneingeschränkt übernommen. Der Zustand $\hat{S}_{q_3}$ besteht aus den Werten, die die Bedingung nicht erfüllen. Die Compound-Anweisung, die den Then-Teil der Verzweigung bildet, wird mit dem Zustand $\hat{S}_{q_1}$ als Eingabe ausgewertet. Der Zustand, der nach der Berechnung des Then-Parts zurückgegeben wird ($\hat{S}_{q_2}$), wird schließlich mit dem Zustand $\hat{S}_{q_3}$ durch einen Join verschmolzen. Der Join führt, wie bei der Abstrakten Interpretation definiert, alle möglichen Werte aus beiden Zuständen zusammen.

  Falls für die Bedingung der Verzweigung festgestellt wird, dass die Bedingung für alle Werte erfüllt ist, so wird der errechnete Zustand $\hat{S}_{q_2}$ nicht mehr mit dem Zustand $\hat{S}_{q_3}$ zusammengeführt, sondern direkt als Ergebnis zurückgegeben. Analog wird die Analyse der Compound-

Anweisung im Then-Teil nicht durchgeführt, wenn die Bedingung durch keinen Wert erfüllt wird. Diese Maßnahme beschleunigt die Laufzeit der Analyse merkbar, ohne dabei die Approximationsgüte zu verschlechtern.

- *If/Else-Anweisung* (siehe Abbildung 4.8 (b))

  Die If/Else-Anweisung ist ähnlich wie die If-Anweisung aufgebaut. Abhängig von der Bedingung werden zwei neue Zustände $\hat{S}_{q_1}$ und $\hat{S}_{q_3}$ berechnet, die die Bedingung erfüllen, bzw. nicht erfüllen. Die Analyse des Then-Parts wird mit dem Startzustand $\hat{S}_{q_1}$ aufgerufen. Das Ergebnis wird in dem Zustand $\hat{S}_{q_2}$ festgehalten, damit es mit dem Zustand $\hat{S}_{q_4}$, der nach Ausführung des Else-Parts entsteht, über einen Join verschmolzen werden kann. Der Zustand, der aus der Verschmelzung resultiert, bildet das Ergebnis des If/Else-Statemens.

  Wird während der Auswertung der Bedingung festgestellt, dass die Bedingung durch alle Werte erfüllt oder nicht erfüllt wird, so wird die zugehörige Compound-Anweisung des Then- oder Else-Blocks, analog zu den If-Anweisungen, nicht ausgewertet. Der anschließende Join der beiden Programmpfade kann dann ebenfalls entfallen.

- *Switch-Anweisung* (siehe Abbildung 4.8 (c))

  Die Simulation der Switch-Anweisungen ist ähnlich aufgebaut, wie die der If-Anweisungen. Für jeden Case-Block wird der eingehende Zustand, abhängig von der Bedingung, in einen Zustand der die Bedingung erfüllt und einen anderen, der die Bedingung nicht erfüllt, aufgeteilt. Der Zustand, der den Fall repräsentiert, dass die Bedingung erfüllt ist, wird als Eingabe für die Berechnung der Compound-Anweisungen verwendet, die innerhalb des Case-Blocks liegt. Das Ergebnis dieser Berechnung wird wieder mit dem Zustand verschmolzen, für den die Bedingung nicht erfüllt wurde. Das Ergebnis der Verschmelzung bildet wiederum die Eingabe für den nächsten Case-Block. Falls von keiner Case-Bedingung festgestellt werden konnte, dass sie immer wahr ist, so wird nach allen Case-Blöcken schließlich noch der Default-Block analog zu den Case-Blöcken ausgewertet. Der daraus resultierende Zustand ist das Ergebnis der Auswertung einer Switch-Anweisung.

  Auch für die Switch-Anweisungen gilt, dass alle Compound-Anweisungen, deren Bedingungen durch keinen Wert erfüllt werden, nicht analysiert werden.

- *Loop-Anweisung* (siehe Abbildung 4.9)

  Die Loop-Anweisungen bilden einen Spezialfall, da diese durch die Schleifenanalyse in besonderer Weise ausgewertet werden müssen. In dieser Arbeit wurden zwei unterschiedliche Analysemethoden umgesetzt, die mit Hilfe der Abstrakten Interpretation die Schleife analysieren sollen. Dabei wird einerseits der Zustand ermittelt, der nach der Schleife gültig ist, und andererseits wird während der Analyse direkt ermittelt, wie häufig die Schleife für den eingehenden Zustand iteriert. Während versucht wird, einen Zustand zu finden, für den die Schleifenabbruchbedingung nicht mehr erfüllt ist, wird bei jeder Iteration eine Zählvariable inkrementiert. Nachdem die Bedingung schließlich nicht mehr erfüllt ist, steht der Wert der Schleifeniterationsgrenze in dieser Zählvariable.

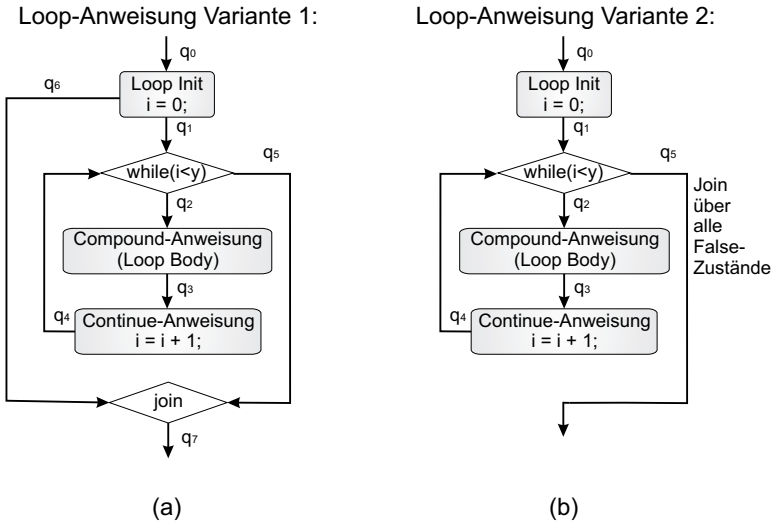


Abbildung 4.9: *Modifizierte Abstrakte Interpretation - Schleifen*

Im entwickelten Programm kann die gewünschte Analysemethode durch einen Parameter ausgewählt werden.

Variante 1 Die erste Variante der Schleifenanalyse (siehe Abbildung 4.9 (a)) führt als erstes für den eingehenden Zustand $\hat{S}_{q_0}$ die Initialisierung der Schleife durch, falls es sich um eine For-Schleife handelt. Alle anderen Schleifentypen besitzen keine Schleifeninitialisierung im Schleifenkopf, so dass in diesem Fall der Zustand $\hat{S}_{q_1}$ durch eine Kopie des Zustands $\hat{S}_{q_0}$ gebildet wird. Auf Basis des ermittelten Zustands $\hat{S}_{q_1}$ werden durch Auswertung der Abbruchbedingung, auf Ebene der Abstrakten Domäne, die Zustände $\hat{S}_{q_2}$ (Werte, die die Bedingung erfüllen) und $\hat{S}_{q_5}$ (Werte, die die Bedingung nicht erfüllen) gebildet. Solange nicht für alle Werte des Zustands sichergestellt ist, dass sie die Bedingung nicht erfüllen, wird der Schleifenrumpf iterativ ausgeführt. Dabei wird durch Analyse des Schleifenrumpfs der Zustand $\hat{S}_{q_3}$ gebildet, der, abhängig vom Schleifentyp, noch durch eine Continue-Anweisung modifiziert wird. Der daraus resultierende Zustand $\hat{S}_{q_4}$ wird schließlich wieder ausgewertet, wodurch sich nach erneuter Auswertung der Bedingung neue Zustände für die Programmpunkte q_2 und q_5 ergeben.

Der Zustand $\hat{S}_{q_5}$, der sich nach der Schleifenanalyse ergibt, wird nun noch mit dem Zustand $\hat{S}_{q_1}$ über einen Join verbunden, um alle möglichen Zustände zu vereinigen, die auch während der Schleife gültig waren. Dieser Join muss durchgeführt werden, da es auch bereits während der Iteration Zustände gegeben haben kann, für die nicht bestimmt werden konnte, ob die Bedingung erfüllt wurde, oder nicht. Da diese Zustände ebenfalls zum Abbruch der Schleife führen würden, müssen diese in den resultierenden Zustand $\hat{S}_{q_7}$ aufgenommen werden.

Variante 2 Durch die Verschmelzung des Zustands $\hat{S}_{q_5}$ mit dem Zustand, der vor der Schleife gültig war, gehen bei der ersten Variante der Schleifenanalyse sehr viele Informationen verloren, auch wenn die Approximation korrekt ist. Für einen Schleifeniterator, der Beispielsweise von 0 bis 10 läuft, wird durch die erste Analyse angegeben, dass sein Wert

durch das Intervall $[0, 10]$ approximiert wird, obwohl vielleicht nur der Wert $[10, 10]$ gültig wäre. Vor allem bei verschachtelten Schleifen hat sich gezeigt, dass durch diesen Informationsverlust keine guten Ergebnisse erzielt werden können, wenn die Bedingung der inneren Schleife beispielsweise durch den Iterator der äußeren bedingt wird.

Aus diesem Grund wurde in dieser Arbeit eine zweite Analysemethode entwickelt, die in Abbildung 4.9 (b) zu sehen ist. Der Aufbau ist identisch mit dem der ersten Variante, mit dem Unterschied, dass nach Verlassen der Analyse der Join mit dem Zustand $\hat{S}_{q_1}$ entfällt. Um nun aber ein korrektes Ergebnis zu erzielen, wird während der Analyse automatisch ein Join aller Zustände erstellt, die die Bedingung möglicherweise nicht erfüllen. Durch diese Art der Verschmelzung ist garantiert, dass eine sichere und zudem auch deutlich genauere Analyse durchgeführt wird.

Zusätzlich zu den hier vorgestellten Befehlstypen, gibt es noch diverse andere, wie z.B. die unbedingten Sprünge, die durch ein *Break*, oder ein *Return* entstehen können. Des Weiteren wurden auch Analysetechniken entwickelt, die für die Auswertung der Ausdrücke (*Expressions*) zuständig sind. Dazu gehören z.B. die binären Ausdrücke, die zwei einzelne Ausdrücke über den $\hat{+}$ Operator verbinden. Diese sollen hier allerdings nicht weiter erläutert werden.

Auf die implementierungsnahe Umsetzung der modifizierten Abstrakten Interpretation wird später in Kapitel 7 eingegangen.

Kapitel 5

Softwarebasierte Auswertung von Bedingungen durch Polytop-Berechnungen

Ein *Polyeder* ist ein mehrdimensionaler Körper, der durch gerade Flächen begrenzt wird. Dazu gehören z.B. Würfel oder auch pyramidenartig geformte Körper.

Mathematisch betrachtet ist ein Polyeder eine Teilmenge des $\mathbb{R}^n$, die sich als Durchschnitt von endlich vielen Halbräumen beschreiben lässt [Fai06]. Ein Polyeder kann sich daher durch das lineare Ungleichungssystem aus Definition 5.1 darstellen lassen:

$$P := \{x \mid Ax \leq b\} \tag{5.1}$$

Laut Definition ist jeder affine Unterraum des $\mathbb{R}^n$, sowie auch der gesamte $\mathbb{R}^n$ und die leere Menge $\emptyset$, ein Polyeder. Ein beschränktes Polyeder wird als *Polytop* bezeichnet. Beschränkt bedeutet in diesem Fall, dass für alle Eckpunkte des Polyeders, und dadurch auch für alle Punkte auf der konvexen Hülle, gilt, dass dessen Abstand zum Ursprung durch eine reelle Zahl *R* begrenzt ist. Formal ausgedrückt muss für ein Polyeder *P* die Definition 5.2 erfüllt sein, damit es ein Polytop ist:

$$\forall x \in P : \|x\| \leq R \in \mathbb{R} \tag{5.2}$$

Für diese Arbeit sind die Polytope deshalb interessant, da jede affine Bedingung in einen Polytop-Körper transformiert werden kann. Auf diesen Polytopen können schließlich diverse Operationen ausgeführt werden, die wichtige Ergebnisse für die verschiedensten Optimierungsverfahren liefern können.

In der Forschung der Informatik wird die Polytop-Berechnung hauptsächlich in der Optimierung von

Parallelisierung der Programmausführung und der Analyse von verschachtelten Schleifen benutzt [Tea02]. Dazu gehört z.B. die Arbeit [FM03], in der ein Verfahren beschrieben wird, mit dessen Hilfe durch Polytop-Berechnungen komplexe, verschachtelte Schleifen automatisch optimiert werden.

Bezogen auf diese Arbeit sollen über Polytop-Berechnungen die folgenden Ziele erreicht werden:

- *Einschränkung des Wertebereichs von Variablen* (siehe Abschnitt 5.2)

  In Kapitel 4 wurde für die Bedingungs-Knoten definiert, dass diese jeweils einen nachfolgenden Zustand haben, der alle Werte enthält, für die die Bedingung erfüllt wird, und einen Zustand, der sich aus allen Werten zusammensetzt, die die Bedingung nicht erfüllen. Mit Hilfe der Polytop-Berechnungen sollen diese beiden Zustände berechnet werden. Würden die Wertebereiche der Variablen ohne eine solche Einschränkung übernommen, so würde eine ungewollte Überapproximation entstehen.

- *Berechnung der Iterationshäufigkeit von Schleifen für die statische Analyse* (siehe Abschnitt 5.3)

  Mit Hilfe der sogenannten *Erhard-Polynome* [VSB+04] ist es möglich, die Anzahl an ganzzahligen Punkte in dem Raum eines Polytops zu bestimmen. Dieser Wert stellt eine obere Grenze für die Iterationshäufigkeit von Schleifen dar und soll für die Berechnung der statischen Schleifenanalyse aus Kapitel 6 genutzt werden.

In diesem Kapitel soll beschrieben werden, wie diese Ziele erreicht werden. Dazu wird in Abschnitt 5.1 zunächst erklärt, wie Bedingungen einer Programmiersprache in das genannte lineare Ungleichungssystem eines Polytops transformiert werden können. In den Abschnitten 5.2 und 5.3 wird anschließend erklärt, wie die genannten Ziele mit Hilfe des errechneten Polytops erreicht werden. Schließlich wird in Abschnitt 5.4 auf die *Polyhedral Library* (kurz Polylib) eingegangen, mit dessen Hilfe die Polytop-Berechnungen dieser Arbeit durchgeführt werden.

5.1 Transformation von Bedingungen in Polytope

Um die gewünschten Ziele aus dem vorherigen Abschnitt zu erreichen, muss vorerst ein Polytop berechnet werden, das die vollständige Bedingung auf Code-Ebene darstellt. Dieses Polytop wird durch ein normiertes Ungleichungssystem nach Definition 5.1 repräsentiert, das wie folgt gebildet wird [FM03]:

1. Die Bedingung der auszuwertenden Programmverzweigung wird in eine äquivalente Form $if(C_1 \oplus C_2 \oplus ... \oplus C_n)$ gebracht.

   Die C_i stehen dabei für beliebige boolsche Ausdrücke. Die Menge der Operatoren $\oplus$ besteht aus den Elementen $\{\wedge, \vee\}$. Zusätzlich kann jeder Ausdruck noch durch den Not-Operator ($\overline{x}$)

negiert werden. Mit Hilfe der Aussagenlogik und dem Satz von De Morgan ist es möglich, jeden boolschen Ausdruck in die oben genannte, äquivalente Form zu bringen [MK06].

2. Jeder atomare boolsche Ausdruck C_x wird in die Form $C_x = \sum_{l=1}^{N}(c_l * i_l) + c \geq 0$ gebracht.

   Die normierte Form des Ungleichungssystems setzt voraus, dass bei jedem boolschen Ausdruck alle Werte auf der linken Seite einer $\geq$-Relation stehen. c_l sowie c sind dabei konstante Werte, wobei i_l für Variablen steht, die innerhalb des boolschen Ausdrucks vorkommen. Jeder affine boolsche Ausdruck, der durch die Operatoren $\{<, \leq, >, \geq, =\}$ gegeben ist, kann durch eine Transformation in die oben genannte Form überführt werden [MK06].

Jeder atomare boolsche Ausdruck stellt nach dieser Transformation durch sich und die Grenzen der beteiligten Variablen ein lineares Ungleichungssystem, wie in Definition 5.1 gegeben, dar, welches ein vollständiges Polytop beschreibt. Abhängig von den Verknüpfungsoperatoren wird das Polytop, das die gesamte Bedingung repräsentiert, aus einzelnen Teilpolytopen durch eine passende Verknüpfung erzeugt. Abhängig vom Operator wird eine der folgenden Verknüpfungen ausgeführt:

- $\wedge$: Es wird durch den $\cap$-Operator der Schnitt der Polytope gebildet.
- $\vee$: Es wird durch den $\cup$-Operator die Vereinigung der Polytope gebildet.
- $\neg$: Es wird das Komplement des Polytops gebildet.

Alle anderen logischen Operatoren werden durch eine Kombination aus diesen drei Operatoren innerhalb der Normierung ersetzt.

5.1.1 Beispiel

Anhand eines Beispiels soll die Ausnutzung der Polytop-Berechnungen für eine Programmverzweigung demonstriert werden.

Es sei die folgende bedingte Programmverzweigung gegeben:

$$\text{if } (\overline{(2 * i + 2 \leq j \text{ NAND } j > 15)})$$

Zuvor wurde über die modifizierte Abstrakte Interpretation folgender Zustand bestimmt, der vor der Verzweigung gültig ist:

$$\hat{S}_q = \{i \rightarrow [5..15], j \rightarrow [10..20]\}$$

Wird der erste Teil der Bedingung $2 * i + 2 \leq j$ auf Ebene der Abstrakten Domäne ausgewertet, so ergibt sich für den Zustand $\hat{S}_{q_{true}}$ die Bedingung $[12..32] \hat{\leq} [10..20]$. Da einige Werte laut Definition

des Abstrakten Operators $\hat{\leq}$ kleiner oder gleich sind, andere aber nicht, wird als Ergebnis der Bedingung $\top$ zurückgegeben (siehe Kapitel 4.2.4). Gleiches gilt für den rechten Teil der Bedingung, für den die Bedingung $j > 15$ auszuwerten ist. Anhand des Zustands ergibt sich folgende Bedingung: $[10..20] \hat{>} [15..15]$. Auch diese Bedingung kann nicht mit $true$ oder $false$ beantwortet werden, so dass $\top$ als Ergebnis zurückgegeben wird. Der Schnitt beider Werte ergibt als Rückgabe der gesamten Auswertung ebenfalls den $\top$-Operator. Daher müssen bei der Analyse des Programms beide Programmpfade analysiert werden.

Da die Bedingung auf Ebene der Abstrakten Domäne also nicht die Lösung $true$ oder $false$ geliefert hat, soll nun versucht werden über die Bildung von Polytopen die Zustände $\hat{S}_{q_{true}}$ und $\hat{S}_{q_{false}}$ zu bestimmen, bei denen der gültige Wertebereich der Variablen *i* und *j* entsprechend eingeschränkt wird.

Als erstes muss die Bedingung bezüglich der ersten Forderung normiert werden. Da der NAND-Operator nicht in der Menge $\oplus$ der gültigen Operatoren enthalten ist, muss dieser durch Transformation eliminiert werden. In der Aussagenlogik gilt, dass jeder NAND-Operator durch folgende Gleichung ersetzt werden kann.

$$x \text{ NAND } y = \overline{x \wedge y}$$

Für das oben genannte Beispiel, kann die Bedingung also in die folgende Form überführt werden:

$$\text{if } (\overline{(\overline{2 * i + 2 \leq j} \vee \overline{j > 15})})$$

Laut dem Satz von de Morgan gilt:

$$\overline{(a \wedge b)} = \overline{a} \vee \overline{b} \text{ und } \overline{(a \vee b)} = \overline{a} \wedge \overline{b}$$

Wird dieser auf die obige Formel angewendet, so entsteht der folgende Ausdruck:

$$\text{if } (2 * i + 2 \leq j \wedge j > 15)$$

Die Bedingung erfüllt nun die Form $if(C_1 \oplus C_2 \oplus ... \oplus C_n)$, so dass die Normierung und somit der erste Schritt der Umformung abgeschlossen ist. Als zweites müssen beide Bedingungen in die Form $C_x = \sum_{l=1}^{N}(c_l * i_l) + c \geq 0$ gebracht werden. Das Ergebnis sieht wie folgt aus:

$$\text{if } (-2 * i + j - 2 \geq 0 \wedge j - 16 \geq 0)$$

Anhand der transformierten Teilbedingungen können nun die beiden Ungleichungssysteme aufgestellt werden, die die Teilpolytope bilden:

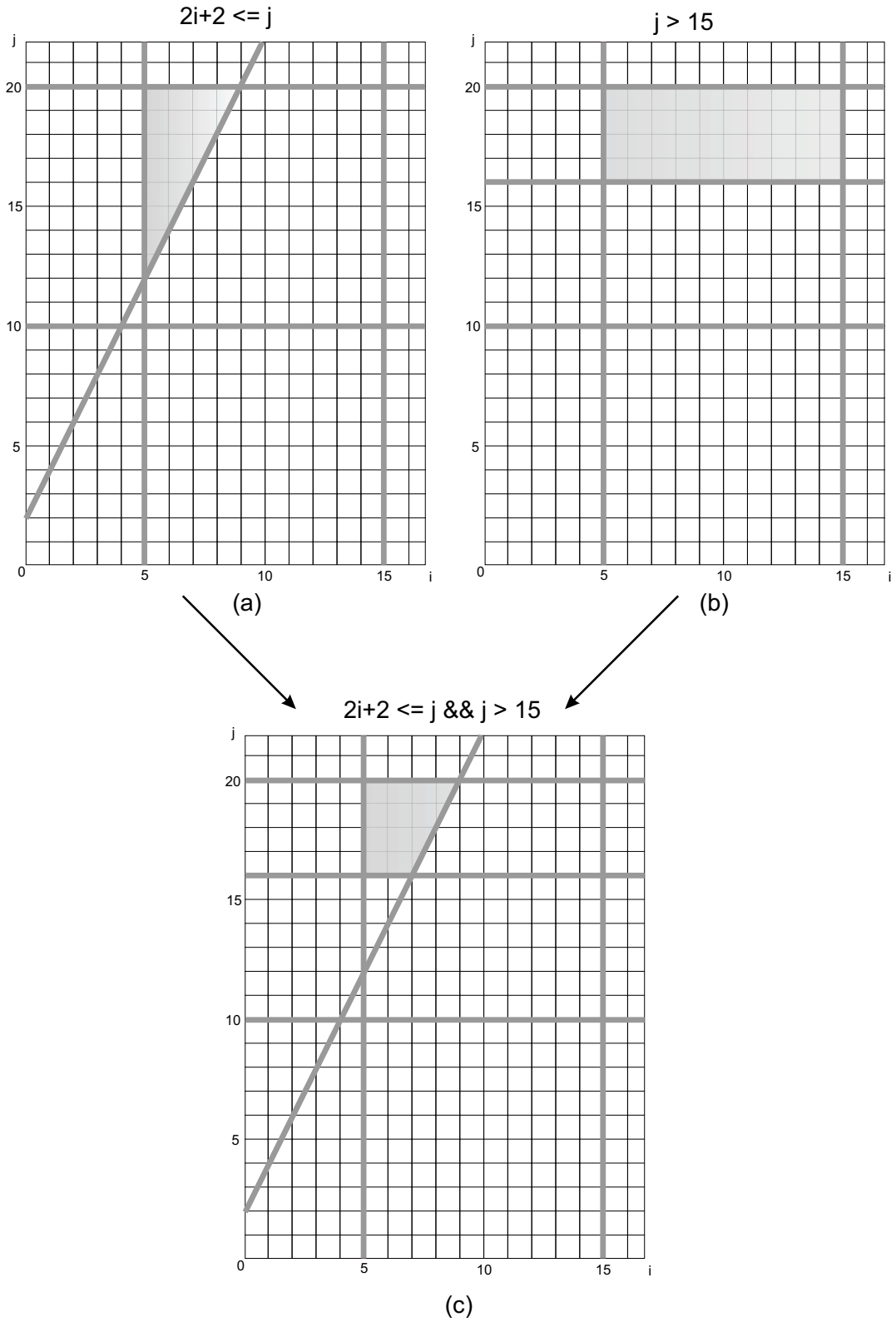


Abbildung 5.1: *Beispiel zur Transformation einer Bedingung zu einem Polytop*

$$P_1 = \left\{ x \in \mathbb{Z}^2 \middle| \begin{pmatrix} -2 & 1 \\ 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} x \geq \begin{pmatrix} 2 \\ 5 \\ -15 \\ 10 \\ -20 \end{pmatrix} \right\}$$

$$P_2 = \left\{ x \in \mathbb{Z}^2 \middle| \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} x \geq \begin{pmatrix} 16 \\ 5 \\ -15 \\ 10 \\ -20 \end{pmatrix} \right\}$$

Die erste Zeile der Ungleichungssysteme wird grundsätzlich durch die Bedingung selbst gebildet. Alle weiteren Zeilen, repräsentieren den Wertebereich, den die Variablen, abhängig vom Zustand, annehmen können. In diesem Beispiel gilt, dass die Variable *i* im Zustand $\hat{S}_q$ alle Werte zwischen 5 und 15 annehmen kann. Dies ist jeweils in den Zeilen 2 und 3 der Ungleichungssysteme festgehalten. Analog werden die Zeilen 4 und 5 gebildet, indem der Wertebereich der Variable *j* auf 10 bis 20 eingeschränkt wird. Die beiden Teilpolytope sind in Abbildung 5.1 (a) und (b) visualisiert.

Da die beiden Teilbedingungen durch den $\vee$-Operator verknüpft sind, werden diese auf Ebene der Polytope durch den Schnitt-Operator $\cap$ zu dem gesuchten Zielpolytop verbunden. Dieses kann durch das folgende Ungleichungssystem beschrieben werden:

$$P_{gesamt} = \left\{ x \in \mathbb{Z}^2 \middle| \begin{pmatrix} -2 & 1 \\ 0 & 1 \\ 0 & -1 \\ 1 & 0 \end{pmatrix} x \geq \begin{pmatrix} 2 \\ 16 \\ -20 \\ 5 \end{pmatrix} \right\}$$

Das errechnete Polytop P_{gesamt} ist das Polytop, welches die möglichen Werte des Zustands $\hat{S}_{q_{true}}$ beschreibt. Es ist in Abbildung 5.1 (c) zu sehen. Das Polytop des Zustands $\hat{S}_{q_{false}}$ wird analog dadurch gebildet, dass die Bedingung negiert wird.

Wie anhand des berechneten Polytops nun die möglichen Werte der Variablen bestimmt werden können, wird im nächsten Abschnitt beschrieben.

5.2 Einschränkung des Wertebereichs von Variablen

Sobald das Polytop definiert ist, das die gesamte Bedingung darstellt, kann anhand dessen der Wertebereich des neuen Zustands approximiert werden. Im Unterraum, der durch das Polytop eingeschlossen wird, befinden sich alle Wertekombinationen der beteiligten Variablen, die die Bedingung erfüllen. Um eine sichere Abschätzung der möglichen Werte zu erhalten, müssen die globalen

Extrema des Polytops bestimmt werden. Da ein Polytop laut seiner Definition nur durch gerade Flächen beschränkt werden darf, liegen dessen Extrema auf den Eckpunkten und sind daher effizient berechenbar.

Bezogen auf das Beispiel aus Abschnitt 5.1.1, besitzt das gesuchte Polytop, das die gesamte Bedingung repräsentiert, die folgenden Eckpunkte:

$$\{(5,16),(7,16),(9,20),(5,20)\}$$

Die Punkte können aus Abbildung 5.1 (c) abgelesen, bzw. über mathematische Berechnungen ermittelt werden.

Der gesuchte Zustand $\hat{S}_{q_{true}}$ kann nun dadurch gebildet werden, dass für jede Variable ein Join über die vorkommenden Eckpunktkoordinaten vorgenommen wird. Dabei bildet die erste Koordinate des obigen Tupels jeweils einen möglichen Wert der Variable *i*. Der zweite Wert stellt eine mögliche Koordinate der Variable *j* dar. Der Zustand $\hat{S}_{q_{true}}$, des oberen Beispiels, beinhaltet also folgende Werte:

$$\hat{S}_{q_{true}} = \{i \rightarrow [5..9], j \rightarrow [16..20]\}$$

5.3 Berechnung der Iterationshäufigkeit von Schleifen

Das zweite Ziel dieser Arbeit, bezogen auf die Polytop-Berechnungen, ist eine statische Bestimmung der oberen Iterationsgrenze für Schleifen. Jeder ganzzahlige Punkt, der in dem Polytop enthalten ist, steht für einen Punkt, der möglicherweise während der Abarbeitung einer Schleife erreicht werden kann. Mit Hilfe von *Erhard-Polynomen* ist es möglich, anhand eines gegebenen Polytops, die Anzahl der ganzzahligen Punkte zu berechnen, die durch das Polytop eingeschlossen werden. Dafür wird das Polytop *P* der Dimension *n* in ein parametrisierbares Polynom *E* der gleichen Dimension transformiert. Dieses Polynom hat die Form $E_P(k) = a_n * k^n + a_{n-1} * k^{n-1} + ... + a_0$, wobei für den Parameter $k \in \mathbb{Z}$ gelten muss, da die Menge der ganzzahligen Punkte gesucht wird. Durch den Satz von Eugène Ehrhart wird garantiert, dass für jedes Polytop ein solches Polynom existiert.

Welche Möglichkeiten es gibt, die Erhard-Polynome und mit dessen Hilfe die Anzahl der ganzzahligen eingeschlossenen Punkte des Polytops computerbasiert zu berechnen, ist in [VSB+04] beschrieben.

Würde die Anzahl der ermittelten Punkte allerdings direkt als Schleifeniterationsgrenze übernommen, so würde daraus eine starke Überapproximation resultieren. Alle Punkte innerhalb des Polytops werden nur dann erreicht, wenn der Schleifeniterator bei jeder Iteration nur um eins erhöht wird. Ist die Modifikation des Iterators größer, so werden einige Punkte innerhalb des Polytops nicht erreicht. In diesem Fall muss die Anzahl der Punkte anschließend durch den Wert dividiert werden,

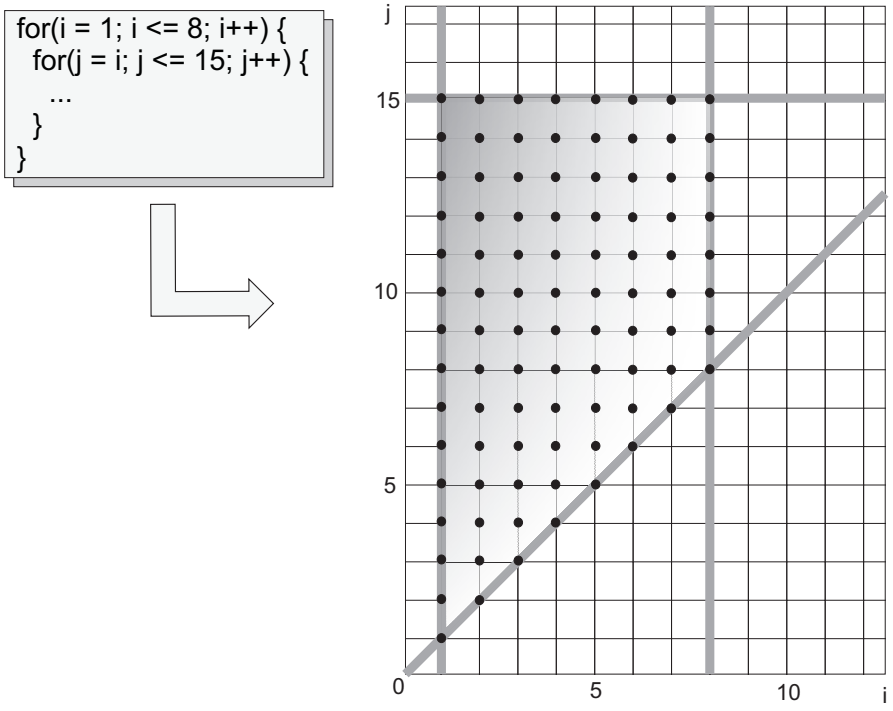


Abbildung 5.2: *Beispiel zur Transformation einer Schleife zu einem Polytop*

um den der Iterator erhöht würde. Des Weiteren kann die Iterationshäufigkeit der Schleife auch durch Break- und Continue-Anweisungen beeinflusst werden. Dessen Bedingungen müssen in die Polytop-Berechnungen einfließen, damit das Ergebnis nicht verfälscht wird.

Ein Beispiel für die Berechnung der Iterationshäufigkeit von Schleifen ist in Abbildung 5.2 zu sehen. Dort soll mit Hilfe von Erhard-Polynomen die Anzahl der Iterationen der inneren Schleife berechnet werden. Dazu wird, wie in Abschnitt 5.1 beschrieben, das Ungleichungssystem aufgestellt, das die beiden Schleifen beschreibt. Diese werden anschließend über den Schnitt-Operator $\cap$ zu einem einzelnen Polytop zusammengefasst. Auf Grundlage dieses berechneten Polytops werden schließlich die Erhard-Polynome gebildet, mit denen es möglich ist die Anzahl der ganzzahligen Punkte innerhalb des Polytops zu bestimmen. Für das obige Beispiel sind dies 92 Punkte, die auch direkt der Iterationshäufigkeit entsprechen, da beide Iteratoren jeweils nur um den Wert eins pro Iteration erhöht werden.

Um die Anzahl der Iterationen der äußeren Schleife zu bestimmen, reicht es aus, das zugehörige Polytop ohne den Schnitt des Polytops der inneren Schleife zu analysieren. In diesem Fall würde sich ein eindimensionales Polytop ergeben.

5.4 Polyhedral Library (Polylib)

Die Polytop-Berechnungen, die in diesem Kapitel beschrieben worden sind, sollen in dieser Arbeit mit Hilfe der *Polyhedral Library* (kurz: Polylib) durchgeführt werden. Die Polylib ist eine frei erhältliche

ANSI-C-Bibliothek. Als Unterstützte Betriebssysteme werden Unix, Linux und Windows angegeben. Entwickelt wurde die Bibliothek von der Brigham Young University, dem ICPS in Straßburg, dem LIP in Lyon und der Irisa in Rennes [Tea02].

Zu den Operationen, die mit Hilfe der Polylib durchgeführt werden können, gehören u.a. die folgenden:

- Bildung des Schnitts von zwei Polytopen
- Bildung der Vereinigung von zwei Polytopen
- Bildung der Differenz von zwei Polytopen
- Bildung des Komplements eines Polytops
- Überprüfung, ob zwei Polytope eine Teilmengenbeziehung haben
- Überprüfung, ob zwei Polytope identisch sind
- Überprüfung, ob ein Polytop leer ist oder das komplette Universum abbildet
- Berechnung der eingeschlossenen ganzzahligen Punkte des Polytops

Für diese Arbeit sind hauptsächlich die ersten vier, sowie der letzte Operator interessant.

Um die Berechnung, sowie auch die Erzeugung, von Polytopen effizient zu gestalten, bietet die Polylib u.a. Methoden zur Erzeugung und Manipulation von Matrizen.

Um die Polylib generell nutzen zu können, wird die Bibliothek direkt in das auszuführende Rahmenprogramm eingebunden. Als erstes werden Matrizen, die als Eingabe dienen sollen, z.B. anhand der Argumente des Programms über die Funktion *Matrix_Read()* der Polylib, erstellt. Mit der Funktion *Constraints2Polyhedron(poly, maxDim)* kann anschließend das Ungleichungssystem eines Polytops mit maximaler Dimension *maxDim* und der Matrix *poly* gebildet werden. Der Parameter *maxDim* gibt die maximale Anzahl an Zeilen des Ungleichungssystems an. Sobald auf diesem Weg alle Polytope erstellt worden sind, können über die unterschiedlichen manipulierenden Funktionen, wie beispielsweise *DomainIntersection(poly1,poly2, maxDim)*, neue Polytope erstellt oder die existierenden verändert werden.

Die Bibliothek bietet allerdings auch noch weitere Möglichkeiten, mit dessen Hilfe die Eingabe-Polytope erstellt werden können. So bietet die Polylib neben der Matrixdarstellung auch die Möglichkeit, Polytope über eine Menge an Eckpunkten, Strahlen und Linien zu definieren. Intern werden diese Darstellungen geeignet transformiert, so dass der Benutzer die Art der Eingabe frei wählen kann.

Da sich die Eingabe in dieser Arbeit auf die Polytop-Erstellung anhand von Matrizen beschränken wird, soll diese Vorgehensweise genauer vorgestellt werden. Als Eingabe erwartet die Funktion

Matrix_Read() eine Matrix, die sich Zeilenweise aus den Bedingungen zusammensetzt. So wird die Bedingung $\vec{a} * \vec{x} \geq b$ durch die einzeilige Matrix $\begin{pmatrix} \vec{a} & -\vec{b} & 0 \end{pmatrix}$ dargestellt. Die Polylib bietet eine Unterstützung für Gleichungen und Ungleichungen. Um zu unterscheiden, ob eine Bedingung eine Gleichung oder eine Ungleichung darstellt, wird in der ersten Spalte der Eingabe eine 0 (für den Fall einer Gleichung) oder eine 1 (für den Fall einer Ungleichung) angegeben.

Die Bedingung $-2 * i + j - 2 \geq 0$ aus Abschnitt 5.1.1 würde also durch folgende Matrixzeile dargestellt: $\begin{pmatrix} 1 & -2 & 1 & -2 \end{pmatrix}$. Auch für die Polylib gilt, dass die Bedingungen auf die Form $C_x = \sum_{l=1}^{N}(c_l * i_l) + c \geq 0$ transformiert werden müssen, damit diese als Bedingung in die Matrix aufgenommen werden können.

Die auf diese Weise gebildeten Matrizen werden im Normalfall durch eine Eingabedatei an das zu verarbeitende Programm übergeben. Die beiden Matrizen aus dem obigen Beispiel

$$P_1 = \left\{ x \in \mathbb{Z}^2 \middle| \begin{pmatrix} -2 & 1 \\ 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} x \geq \begin{pmatrix} 2 \\ 5 \\ -15 \\ 10 \\ -20 \end{pmatrix} \right\}$$

$$P_2 = \left\{ x \in \mathbb{Z}^2 \middle| \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} x \geq \begin{pmatrix} 16 \\ 5 \\ -15 \\ 10 \\ -20 \end{pmatrix} \right\}$$

werden durch die Eingabedatei aus Abbildung 5.3 repräsentiert:

```
1   5 4
    1 -2  1  -2
    1  1  0  -5
    1 -1  0  15
5   1  0  1 -10
    1  0 -1  20

    5 4
    1  0  1  -16
10  1  1  0  -5
    1 -1  0  15
    1  0  1  -10
    1  0 -1  20
```

Abbildung 5.3: *Beispieleingabedatei zur Berechnung von Polytopen mit der Polylib*

Vor jeder Matrix der Eingabedatei befinden sich zwei Zahlen, die die Anzahl der Zeilen und die Anzahl der Spalten des folgenden Polytops angeben. Diese Informationen werden benötigt, damit der Inhalt der Eingabedatei effizient verarbeitet werden kann.

Diese beiden Polytope sollten im obigen Beispiel durch den Schnitt-Operator ∩ ein neues Polytop bilden, das die gesamte Bedingung der Verzweigung repräsentiert. Diese Operation wird in der Polylib durch die Methode *DomainIntersection(poly1,poly2, maxDim)* durchgeführt.

Ein Programm, das die oben beschriebene Berechnung durchführt, ist in Abbildung 5.4 zu sehen:

```
1  int main () {
     Matrix *a1 , *a2;
     Polyhedron *D1 , *D2 , *D3;

5    // Bildung der Matrizen anhand der Eingabe :
     a1 = Matrix_Read ();
     a2 = Matrix_Read ();

     // Erstellen der Polytope auf Grundlage der Matrizen :
10   D1 = Constraints2Polyhedron (a1 , 5);
     D2 = Constraints2Polyhedron (a2 , 5);

     // Berechnung des Zielpolytops durch Bildung des Schnitts :
     D3 = DomainIntersection (D1 ,D2 ,5);
15
     // Ausgabe des Zielpolytops :
     Polyhedron_Print (stdout , P_VALUE_FMT ,D3 );
   }
```

Abbildung 5.4: *Beispielquelltext zur Berechnung von Polytopen mit der Polylib*

Die Ausgabe, die in Zeile 17 des Programms erzeugt wird, liefert für das obige Beispiel das Ergebnis aus Abbildung 5.5:

```
1  POLYHEDRON Dimension :2
                    Constraints :4 Equations :0 Rays :4 Lines :0
   Constraints 4 4
   Inequality :    [ -2   1  -2  ]
5  Inequality :    [  0   1 -16 ]
   Inequality :    [  0  -1  20 ]
   Inequality : [  1   0  -5  ]
   Rays 4 4
   Vertex : [ 5  16 ]/1
10 Vertex : [ 7  16 ]/1
   Vertex : [ 9  20 ]/1
   Vertex : [ 5  20 ]/1
```

Abbildung 5.5: *Ergebnisausgabe der Polylib bezüglich des Beispiels*

Die gezeigte Ausgabe beschreibt das berechnete Ziel-Polytop. Dabei werden in den ersten beiden Zeilen allgemeine Informationen über das Polytop zusammengefasst. Die Zeilen, die mit *Inequality* anfangen, bilden das Ungleichungssystem, durch welches das Polytop beschrieben werden kann. Darunter befindet sich eine Auflistung der Eckpunkte, die für die Einschränkung des Wertebereichs aus Abschnitt 5.2 gesucht werden.

Mit dem in diesem Abschnitt vorgestellten Verfahren können beliebig komplexe Polytop-Berechnungen durchgeführt werden.

5.4.1 Integration der Polylib in die Schleifenanalyse

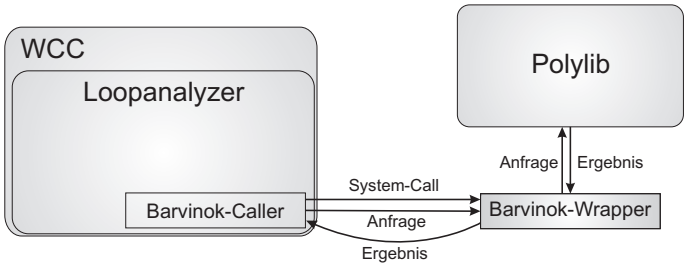


Abbildung 5.6: *Ursprüngliche Kommunikation zwischen dem WCC und der Polylib*

Die Polylib wurde bereits in den WCC integriert, um Bedingungen der ICD-C *IR*, die immer wahr oder falsch sind, zu eliminieren und dadurch den Kontrollfluss zu vereinfachen.

Auf Grund der unterschiedlichen Lizenzgrundlagen des WCC's (kommerzielle Lizenz) und der Polylib (GNU General Public License [GPL08]) ist es nicht möglich gewesen, die Polylib direkt auf dem oben beschriebenen Weg in den Quellcode des WCC's zu integrieren. Stattdessen wurde eine Wrapper-Klasse entwickelt (*Barvinok-Wrapper*), die von den Analysetechniken des WCC's über einen Betriebssystemaufruf (*System-Call*) gestartet werden kann. Die Argumente werden über diverse Eingabedateien an den Wrapper übergeben, der in dem beschriebenen Weg die passenden Objekte der Polylib erzeugt und die angegebenen Berechnungen durchführt. Das Ergebnis der Berechnung wird vom Wrapper in eine vorher definierte Datei geschrieben, so dass diese auf Seite des WCC's wieder eingelesen und weiterverarbeitet werden kann. Dieser Vorgang ist in Abbildung 5.6 zu sehen.

Die ursprüngliche Version dieses Wrappers war für die in dieser Arbeit entwickelte Analyse jedoch nur bedingt einsatzfähig. Der System-Call, über den der Barvinok-Wrapper bei jeder Auswertung neu gestartet wird, erzeugt einen Overhead, der sich in einer langen Programmlaufzeit niederschlägt (siehe Tabelle 5.1 auf Seite 70). Dazu kommt der Overhead, der dadurch entsteht, dass vor jedem Aufruf des Wrappers die Eingabematrizen in Dateien geschrieben werden müssen. Damit dies auf Ebene des C-Programms funktioniert, muss bei jeder Datei auf einen Interrupt des Betriebssystems gewartet werden, wodurch die Laufzeit enorm ansteigt.

In der ursprünglichen Analyse der erwähnten Arbeit ist das Problem der Laufzeit allerdings nicht ins Gewicht gefallen. Dort wurde lediglich für jede Bedingung einer Schleife des zu optimierenden Programms einmalig eine Auswertung durchgeführt. Die in dieser Arbeit entwickelte Schleifenanalyse ruft hingegen für jede Programmverzweigung, die nicht $true$ oder $false$ bei der Auswertung liefert, die Polytop-Berechnung auf. Dies gilt insbesondere auch dann, wenn die Bedingung auf Grund einer Schleife mehrfach mit einem neuen Zustand erreicht wird.

Daher wurde in dieser Arbeit über mehrere Iterationen versucht, die Kommunikation mit der Wrapper-Klasse zu optimieren. In einem ersten Lösungsversuch wurde eine Kommunikation über sogenannte Pipes entwickelt, um in erster Linie den Overhead zu umgehen, der durch die Erzeugung der Da-

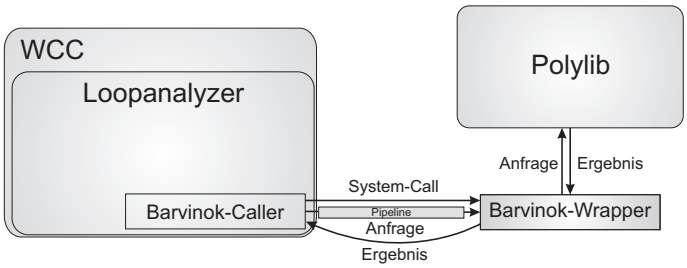


Abbildung 5.7: *Kommunikation zwischen dem Loopanalyzer und der Polylib via Pipes*

teien für die Eingabematrizen entsteht (siehe Abbildung 5.7). Der Ansatz der Pipes baut vor der eigentlichen Kommunikation eine Art Kanal auf, durch den beliebige Dateien ausgetauscht werden können, ohne dass diese vorher auf die Festplatte ausgelagert werden müssen. Dieser Ansatz hat die Laufzeit einiger Analysen in etwa halbiert (siehe Tabelle 5.1). Auf Grund der Tatsache, dass die *Pipeline-Kommunikation* jedoch nur unidirektional ist, muss für das Übermitteln der Ergebnisse in Rückrichtung vom Wrapper zur Analyse des WCC's erneut in eine Datei geschrieben werden. Auch das Problem, dass der Wrapper jedes Mal neu über einen System-Call gestartet werden muss, konnte durch diesen Ansatz nicht behoben werden.

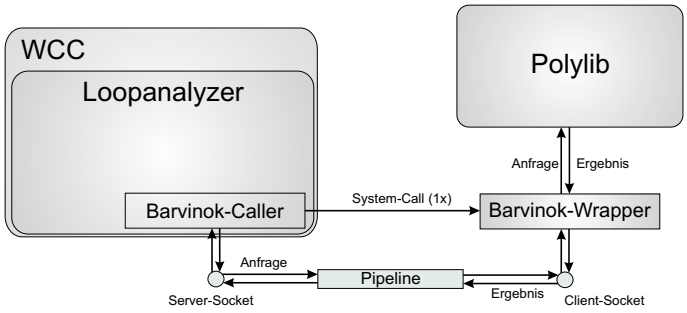


Abbildung 5.8: *Kommunikation zwischen dem Loopanalyzer und der Polylib via Sockets*

Dieser Umstand konnte schließlich durch die Einführung einer socketbasierten Kommunikation umgangen werden (siehe Abbildung 5.8). Der Datenaustausch, der bei einer *Socket-Kommunikation* stattfinden kann, ist bidirektional, so dass auch die erzeugten Ergebnisse vom Wrapper ohne I/O-Aufruf (erzeugt durch das Erstellen der Dateien) an die Analyse des WCC's übertragen werden können. Dabei wird zwischen betriebssysteminternen und einer rechnerübergreifenden Kommunikation, z.B. über das lokale Netzwerk oder das Internet, unterschieden. In dem vorliegenden Fall der Schleifenanalyse wurde eine lokale Kommunikation über das Protokoll *AF_Unix* umgesetzt.

Die beiden Partner einer socketbasierten Kommunikation werden in die sogenannten Rollen *Server* und *Client* eingeteilt. Als Server wird der Teilnehmer bezeichnet, der die Kommunikation instantiiert. Dies ist in diesem Fall die implementierte Schleifenanalyse. Auf der anderen Seite wartet der Client auf einen Verbindungsaufbau des Servers, der in diesem Fall im modifizierten Barvinok-Wrapper implementiert ist.

Um eine Verbindung zu instantiieren, wird auf beiden Seiten ein Socket angelegt. Bei dem verwen-

deten Protokoll *AF_Unix* wird dabei ein entsprechender Port angegeben, an dem der Client lauscht, bzw. über den der Server eine Verbindung aufbauen wird. Der Server versucht nun über den erstellten Socket eine Verbindung zum Client herzustellen. Dafür wird mit der Funktion *connect()* auf Serverseite, bzw. *accept()* auf Clientseite ein *Handshake* durchgeführt, mit dem die Kommunikation final instantiiert wird. Von diesem Zeitpunkt an ist, ähnlich wie bei der Pipelineverbindung, ein Kanal erstellt, durch den beide Seiten über die Funktionen *send()* und *recv()* Daten beliebig übertragen können.

Beim Start der Schleifenanalyse wird nun direkt durch einen System-Call der Client des Barvinok-Wrappers gestartet. Der Client wartet nun auf eine Anfrage, die durch die erste Analyse einer Bedingung durchgeführt wird. Durch den vorher erstellten Kanal wird die Anfrage, inklusive aller beteiligten Matrizen, als Argument übergeben. Der Wrapper erzeugt auf der anderen Seite des Kanals die benötigten Objekte der Polylib und führt die beschriebenen Polytop-Berechnungen, abhängig von der Anfrage, durch. Nachdem die Ergebnisse vorliegen, werden diese durch den Socket-Kanal als Argumente zurückgesendet, so dass diese in der Folge durch die Analyse ausgewertet und benutzt werden können. Der Client verweilt nun so lange in einem wartenden Zustand, bis die nächste Anfrage gestellt wird.

Wenn die Analyse aller Schleifen abgeschlossen ist, so wird der Client von der Schleifenanalyse beendet und der Socket geschlossen.

Welche Auswirkung die beiden Kommunikationstypen auf die Schleifenanalyse haben, wird anhand von zwei Testprogrammen (*Benchmarks*) aus der *WCETBENCH* des WCC's in der Tabelle 5.1 dargestellt:

Benchmark	Laufzeit vorher	Laufzeit Pipes	Laufzeit Sockets
janne_complex	03:20 Min	01:35 Min	2,8 Sek
bsort100	40:35 Min	25:27 Min	32,7 Sek

Tabelle 5.1: *Laufzeitreduktion durch Anpassung der Kommunikationstechnik zwischen dem Loopanalyzer und der Polylib*

Die angegebenen Zeiten wurden durch das Unix-Programm *time*, bei der verwendeten Rechnerkonfiguration eines AMD Sempron(tm) 3000+ Processors (128 KB Cache) mit 2GB RAM, ermittelt. Von den drei zurückgegebenen Zeiten wurde hier die *User*-Angabe gewählt, die angibt, wie lange sich das Programm im User-Level des Betriebssystems befunden hat. Als Schleifenanalyseverfahren wurde in diesem Fall die erste Variante der modifizierten Abstrakten Interpretation benutzt (siehe Abschnitt 4.3), da durch den Join nach jeder Schleifeniterationen die Intervalle deutlich größer werden, wodurch mehr Polytop-Berechnungen durchgeführt werden müssen (siehe Abschnitt 4.3).

Wie in Tabelle 5.1 zu sehen ist, wurde die Laufzeit der Analyse durch die Anwendung von Pipes, bzw. Sockets erheblich reduziert. So wird durch die Kommunikation via Pipes ca. 50% der ursprünglichen Laufzeit eingespart. Noch größer ist der Laufzeitgewinn, wenn die ursprüngliche Verbindung mit der Socket-Verbindung verglichen wird. Für beide Benchmarks gilt, dass gerade mal

1,4% der ursprünglichen Zeit benötigt wird. Daher wurde die Socket-Variante umgesetzt, um die Kommunikationsgrundlage zwischen der Schleifenanalyse und dem Wrapper der Polylib zu bilden.

Kapitel 6

Statische Schleifenanalyse durch Integration der Polylib

Die statische Schleifenanalyse wurde entwickelt, um die Laufzeit der Schleifenanalyse, die auf Grundlage der modifizierten Abstrakten Interpretation aus Kapitel 4.3 entwickelt worden ist, signifikant zu beschleunigen. Ein großer Nachteil der Abstrakten Interpretation ist, dass die Werte innerhalb der Schleife nahezu iterativ ermittelt werden. Durch diese simulationsartige Auswertung der Schleife wird viel Rechenzeit benötigt. Dieser Nachteil soll durch das statische Vorgehen, der in diesem Kapitel vorgestellten Schleifenanalyse aufgehoben werden. Dabei geht es weniger darum, ein Analyseverfahren zu entwickeln, das alle möglicherweise vorkommenden Schleifen analysieren kann, als vielmehr ein Verfahren in die Gesamtanalyse zu integrieren, das für Schleifen mit einem einfachen Aufbau ein schnelles Analyseergebnis erzielen kann.

Die statische Schleifenanalyse soll innerhalb der Analyse der modifizierten Abstrakten Interpretation bei jeder Schleife aufgerufen werden. Bevor die Analyse startet, soll zunächst überprüft werden, ob die gefundene Schleife durch die statische Analyse ausgewertet werden kann. Ist dies nicht der Fall, so soll ein Analyseergebnis über die modifizierte Abstrakte Interpretation, nach dem vorgestellten Regelwerk, ermittelt werden. Können allerdings mit der statischen Analyse ein gültiger Folgezustand, sowie die gesuchten Schleifeniterationsgrenzen, ermittelt werden, so soll der gefundene Zustand für die restliche Analyse der modifizierten Abstrakten Interpretation genutzt werden. Die Schleife muss in diesem Fall nicht mehr mit der modifizierten Abstrakten Interpretation ausgewertet werden.

Um die Anzahl der Schleifeniterationen zu bestimmen, soll zunächst die Modifikation bestimmt werden, um die der Schleifeniterator bei jedem Durchlauf der zu analysierenden Schleife modifiziert wird. Anschließend soll mit Hilfe der Polylib berechnet werden, wie viele ganzzahlige Punkte sich in dem Polytop der Basisblöcke innerhalb der Schleife befinden. Anhand dieser Informationen soll in der Folge bestimmt werden, wie häufig jeder Basisblock aufgerufen wird, wodurch im Anschluss das Intervall der möglichen Werte aller beteiligten Variablen bestimmt werden kann.

Da im weiteren Verlauf der Analyse davon ausgegangen werden muss, dass eine der im Schleifenrumpf vorkommenden Variablen die Schleifeniterationsgrenze einer späteren Schleife bedingt, reicht es nicht aus, ausschließlich die Schleifeniterationsgrenzen zu bestimmen. Daher wurde ein Verfahren entwickelt, das, neben den Schleifeniterationsgrenzen, auf statische Weise ebenfalls die möglichen Werte bestimmt, die die Variablen im Laufe der Schleife annehmen können.

Ziel dieses Kapitels soll es sein, die statische Schleifenanalyse konzeptionell zu beschreiben, ohne dabei auf Implementierungsdetails einzugehen (siehe Kapitel 7). Dazu wird zunächst in Abschnitt 6.1 definiert, welche Vorbedingungen erfüllt sein müssen, damit eine Schleife durch die statische Analyse ausgewertet werden kann. Im darauf folgenden Abschnitt 6.2 wird der genaue Ablauf der Analyse beschrieben. In Abschnitt 6.3 wird erklärt, wie komplexere, verschachtelte Schleifen mit der vorgestellten Analysemethode ausgewertet werden können. Ein passendes Beispiel, das alle bis dahin beschriebenen Abläufe erläutern soll, wird in Abschnitt 6.4 vorgestellt. Bezogen auf das vorgestellte Beispiel, wird schließlich in Abschnitt 6.5 eine Zusammenfassung gegeben, in der u.a. auf die Güte der Approximation des vorgestellten Analyseverfahrens hingewiesen wird.

6.1 Vorbedingungen für die statische Schleifenanalyse

Damit eine Schleife mit Hilfe der statischen Schleifenanalyse analysiert werden kann, müssen einige Bedingungen erfüllt sein. Diese sind in der folgenden Aufzählung zusammengefasst:

1. Alle Bedingungen innerhalb der Schleife, einschließlich der Abbruchbedingung, sind nur von einer Variable, nämlich der Induktionsvariable, abhängig. Würde eine Bedingung von mehreren Variablen abhängen, so würde sich auf Polytop-Ebene ein mehrdimensionaler Körper ergeben, der eine verschachtelte Schleife darstellt und so zu viele ganzzahlige Punkte beinhaltet.

2. Alle Bedingungen müssen durch einen affinen boolschen Ausdruck gegeben sein, damit diese durch die Polylib-Analyse ausgewertet werden können.

3. Es muss eine eindeutige Induktionsvariable bestimmt werden können. Diese Induktionsvariable darf lediglich in der Continue-Anweisung oder in der obersten Ebene des Schleifenrumpfs modifiziert werden, damit die Iterationsanzahl der Schleife bestimmt werden kann.

4. Innerhalb der Schleife gibt es keine Sprung-Anweisungen wie Break, Continue oder Goto. Dies ist lediglich eine Einschränkung, die den aktuellen Stand der Implementierung betrifft. Sowohl Break- als auch Continue-Anweisungen können in die vorgestellte, statische Analyse integriert werden. Goto-Anweisungen können vernachlässigt werden, da diese in dieser Arbeit auch innerhalb der Abstrakten Interpretation nicht unterstützt werden.

5. Alle Zuweisungen müssen in die Form $=$, $+=$ oder $-=$ transformiert werden können, um zu garantieren, dass die Variablen in jeder Iteration um einen konstanten Faktor erhöht werden oder einmalig einen konstanten Wert zugewiesen bekommen. Außerdem darf der Wert auf

der rechten Seite der Zuweisung nicht durch z.B. ein Postincrement o.ä. modifiziert werden. Dies ist wichtig, da anhand der bestimmten ganzzahligen Punktemenge und der Modifikation einer Variable dessen mögliche Werte ermittelt werden.

6. Die Zuweisungsabhängigkeit muss topologisch sortierbar sein. D.h. dass eine Reihenfolge ermittelbar sein muss, in der die Werte der Variablen bestimmt werden können. Dies wird in einem späteren Beispiel (siehe Abschnitt 6.4) genauer erläutert.

7. Jede Variable, die über eine Zuweisung der Form $=$ modifiziert wird, darf innerhalb der aktuellen Schleife kein zweites mal modifiziert werden. Würde eine solche Variable zuerst über eine Zuweisung modifiziert, um anschließend um einen konstanten Faktor erhöht zu werden, so kann nicht genau bestimmt werden, welchen Wert diese Variable nach Abschluss der Schleife annehmen kann.

8. Es dürfen keine Pointer oder Structs in der zu analysierenden Schleife vorkommen. Auch diese Bedingung resultiert aus der aktuellen Implementierung und kann durch eine Erweiterung des Codes aufgehoben werden.

Diese Bedingungen werden dadurch erleichtert, dass vor der Analyse ein *Program Slicing* (siehe Abschnitt 3.1.1.2) gestartet werden kann, das alle Anweisungen des Programms entfernt, die nicht für den Kontrollfluss, bzw. eine Schleifenbedingung, interessant sind. Dadurch ergeben sich häufig Schleifen, die entweder einen leeren, oder einen recht einfachen Aufbau, haben. Dies hat sich insbesondere in der Evaluation aus Kapitel 8 bestätigt.

6.2 Ablauf der statischen Schleifenanalyse

Sind alle Vorbedingungen aus Abschnitt 6.1 erfüllt, so kann die eigentliche Analyse durchgeführt werden. Diese ist in insgesamt 7 Schritte eingeteilt, die in der folgenden Auflistung zu sehen sind:

1. *Klassifizieren der Variablen*

   In diesem Schritt werden alle Variablen, die innerhalb der Schleife modifiziert werden, herausgesucht und in einen Abhängigkeitsgraphen übertragen. Dies ist notwendig, damit der zweite Schritt der Berechnung durchgeführt werden kann. Zusätzlich wird für jede Variable ein Flag gesetzt, ob diese innerhalb der Schleife um einen konstanten Wert erhöht wird oder ob ein fester Wert zugewiesen wird.

2. *Berechnungsreihenfolge der Variablen bestimmen*

   Der Abhängigkeitsgraph der Variablen, der im letzten Schritt aufgebaut wurde, wird in diesem Schritt mit Hilfe des *Top-Sort Algorithmus* in eine topologisch geordnete Berechnungsreihenfolge gebracht. Wird eine Variable *b* beispielsweise um eine Variable *a* innerhalb der Schleife erhöht, so muss zunächst das Intervall der möglichen Werte von *a* bestimmt werden, um eine Aussage über die möglichen Werte von *b* zu erzielen.

3. *Bestimmung der Modifikation der Induktionsvariable*

   Als nächstes wird die gesamte Schleife nach Modifikationen der Induktionsvariable durchsucht. Alle Wertveränderungen der Anweisungen werden dabei akkumuliert, um so den Modifikationswert der Induktionsvariable zu bestimmen.

4. *Berechnung der Anzahl der ganzzahligen Punkte aller Bedingungen mit Hilfe der Polylib*

   Der gesamte Schleifenrumpf wird nach bedingten Verzweigungen durchsucht. Diese werden schließlich, wie in Kapitel 5 beschrieben, in Ungleichungssysteme transformiert und an die Polylib übergeben. Als Ergebnis liefert die Polylib in diesem Fall die Menge an ganzzahligen Punkten, die innerhalb des jeweiligen Basisblocks liegen.

5. *Berechnung der Iterationshäufigkeit jedes Basisblocks*

   Anhand der berechneten Anzahl der ganzzahligen Punkte innerhalb jedes Basisblocks und der Modifikation der Induktionsvariable pro Schleifendurchlauf, kann nun berechnet werden, wie häufig jeder Basisblock in der Schleife durchlaufen wird. Dieser Wert ergibt sich, indem die Anzahl der Punkte durch den Wert dividiert wird, durch den die Induktionsvariable pro Durchlauf modifiziert wird.

6. *Berechnung der möglichen Werte der Induktionsvariable innerhalb der Schleife*

   Nachdem im vorherigen Schritt berechnet worden ist, wie häufig die enthaltenen Basisblöcke in der Schleife aufgerufen werden, kann der Wert bestimmt werden, den die Induktionsvariable innerhalb der Schleife annehmen kann. Dieser wird durch das Intervall beschrieben, dessen untere Grenze durch den Wert gebildet wird, der vor der Schleife gültig war. Als obere Grenze wird der Wert gewählt, der sich aus der Summe der Modifikationswerte, multipliziert mit den Iterationshäufigkeiten des zugehörigen Basisblocks, ergibt.

7. *Bestimmung der Werte anderer Variablen*

   Genau wie bei der Induktionsvariable wird für alle weiteren Variablen, abhängig von der ermittelten Berechnungsreihenfolge, zunächst der Wert bestimmt, um den die Variable insgesamt erhöht wird. Dies ist auch hier die Summe aus den Modifikationen, multipliziert mit den Iterationshäufigkeiten der zugehörigen Basisblöcke. Als möglicher Wert wird für jede Variable das Intervall bestimmt, das als untere Grenze den Wert enthält, der vor der Schleife gültig war. Als obere Grenze wird der alte Wert, der vor der Schleife gültig war, um die gesamte Modifikation der Variable innerhalb der Schleife erhöht. Ist der Modifikator negativ, werden obere und untere Grenze des Intervalls vertauscht.

Nachdem diese Schritte ausgeführt wurden, ist die statische Schleifenanalyse abgeschlossen. Der errechnete Zustand enthält alle möglichen Werte, die die Variablen innerhalb der Schleife annehmen können. Als Schleifeniterationsgrenze wird die Iterationshäufigkeit der obersten Compound-Anweisung (innerhalb der Hierarchie) des Schleifenrumpfs gewählt.

6.3 Analyse verschachtelter Schleifen

Enthält die Schleife weitere Schleifen auf tieferer Ebene, so wird zunächst die äußerste Schleife ausgewertet, ohne dabei die innere Schleife zu betrachten. Damit die Analyse so ein korrektes Ergebnis liefert, muss geprüft werden, ob die äußere Schleife nicht durch eine seiner inneren Schleifen bedingt wird. Nachdem alle möglichen Werte der Variablen ermittelt wurden, wird anschließend die Analyse der inneren Schleife gestartet. Diese erhält den errechneten Zustand, sowie einen globalen Multiplikator, der angibt, wie häufig diese Schleife aufgerufen wird. Die Analyse der inneren Schleife funktioniert nach dem gleichen Prinzip, wie die der äußeren Schleife. Der errechnete End-Zustand wird zurück an die Analyse der äußeren Schleife gegeben, die diesen wiederum als Ergebnis der gesamten Schleifenanalyse zurück gibt.

Leider muss die gesamte Analyse der äußeren Schleife abgebrochen werden, sobald eine der inneren Schleifen nicht analysiert werden kann. In diesem Fall wird der gesamte Schleifenkomplex mit Hilfe der modifizierten Abstrakten Interpretation analysiert.

6.4 Beispiel

In diesem Abschnitt soll ein Beispiel mit einer verschachtelten Schleife vorgestellt werden, um die vorher angeführten Berechnungsregeln zu verdeutlichen. Der Quellcode des Beispiels ist in Abbildung 6.1 zu finden:

```
1   int main ( void )
    {
     int i = 0;
     int a = 0;
5    int b = 0;
     int c = 0;
     int d = 0;

     for ( i = 0; i < 20; i++ ) {
10     b = a + 3;
       d++;

       for( c = 0; c < 15; c++ ) {
         b++;
15       d++;
       }

       a += 5;
       i = i + 2;
20   }

     return i;
    }
```

Abbildung 6.1: *Beispiel zur statischen Schleifenanalyse*

Während die Analyse mit Hilfe der Abstrakten Interpretation das Programm auswertet, wird in Zeile

9 die erste Schleife gefunden. Der Zustand, der vor Betreten der Schleife gültig ist, ist wie folgt gegeben:

$$\hat{S} = \{i \to [0,0], a \to [0,0], b \to [0,0], c \to [0,0], d \to [0,0]\}$$

Die Analyse wird nun für die äußere Schleife gestartet, wobei die innere Schleife aus Zeile 13 vorerst ignoriert wird. Im ersten Schritt werden die Variablen *i, b, d* und *a* innerhalb des Schleifenrumpfs gefunden. Durch den Top-Sort Algorithmus wird die Berechnungsreihenfolge *i, a, d, b* bestimmt. Die Variable *a* muss in der Berechnungsreihenfolge vor der Variable *b* berechnet werden, da *b* durch *a* in der Zuweisung aus Zeile 10 bedingt wird. Zusätzlich wird bei der Erstellung des Abhängigkeitsgraphen festgehalten, dass die Variablen *i, a* und *d* innerhalb der äußeren Schleife um einen konstanten Faktor modifiziert werden. Der Variable *b* wird dagegen ein Wert zugewiesen.

Im nächsten Schritt wird festgestellt, dass die Induktionsvariable *i* an zwei Stellen modifiziert wird. Die erste Modifikation befindet sich in der Continue-Anweisung des Schleifenkopfs. Dort wird der Wert durch das Inkrement um den Wert eins erhöht. In Zeile 19 wird der Wert der Induktionsvariable ein weiteres Mal um den Wert zwei erhöht. Insgesamt kann also festgehalten werden, dass die Induktionsvariable pro Schleifendurchlauf um den Wert drei erhöht wird.

Nachdem dieser Wert ermittelt wurde, wird nun für alle Bedingungen die Anzahl der Punkte innerhalb des daraus gebildeten Polytops errechnet. In diesem eingeschränkten Beispiel gibt es außer der Abbruchbedingung im Schleifenkopf keine weiteren Bedingungen für die äußere Schleife. Mit Hilfe der Polylib kann nun berechnet werden, dass sich in dem gebildeten Polytop 20 ganzzahlige Punkte befinden, die möglicherweise erreicht werden könnten. Da die Induktionsvariable allerdings pro Iteration um drei erhöht wird, wird nur jeder dritte Punkt innerhalb des Polytops erreicht. Die 20 möglichen Iterationspunkte werden nun durch den Wert 3 dividiert, wodurch der Wert 6,6 ermittelt wird. Da eine Iterationsanzahl nur durch einen ganzen Wert angegeben werden kann, wird dieser Wert auf 7 aufgerundet.

Für die Induktionsvariable kann nun berechnet werden, dass ihre möglichen Werte innerhalb der Schleife durch das Intervall $[0, 21]$ gegeben sind, da die Werterhöhung um 3 insgesamt 7 Mal stattfindet.

Als nächstes werden die möglichen Werte der weiteren Variablen bestimmt, die innerhalb der Schleife modifiziert werden. Laut Berechnungsreihenfolge wird dies zunächst für die Variable *a* gemacht. Deren Wert wird pro Iteration um 5 erhöht. Für die errechneten 7 Iterationen ergibt sich also das Intervall $[0, 35]$. Für die Variable *d* wird der Wert pro Iteration um den Wert eins erhöht. Analog zu den anderen Variablen werden die möglichen Werte der Variable *d* durch das Intervall $[0, 7]$ beschrieben. Für die Variable *b* kann nun ein approximierter Wert berechnet werden, da die möglichen Werte für *a* bereits bestimmt sind. Diese Variable kann innerhalb der Schleife die Werte $[0, 35]$ annehmen, woraus sich für *b* die sichere Approximation aus den Werten $[0, 38]$ ergibt. Da es sich hierbei um eine Zuweisung handelt, wird dieser Wert nicht mit der Anzahl der Iterationen multipliziert.

Damit ist die Analyse der äußeren Schleife abgeschlossen, wobei der folgende Eingabezustand für die innere Schleife errechnet wurde:

$$\hat{S}' = \{i \to [0, 21], a \to [0, 35], b \to [0, 38], c \to [0, 0], d \to [0, 7]\}$$

Die Analyse der inneren Schleife wird nun mit dem Zustand $\hat{S}'$, sowie dem globalen Multiplikator 7 aufgerufen, der angibt, dass die innere Schleife insgesamt sieben Mal aufgerufen wird. Analog zur Analyse der äußeren Schleife wird bestimmt, dass die Induktionsvariable *c* pro Iteration um den Wert eins erhöht wird. Durch die Verwendung der Polylib wird berechnet, dass in dem Schleifenrumpf insgesamt 15 mögliche Iterationspunkte liegen. Da die Induktionsvariable jeweils nur um eins erhöht wird, iteriert die innere Schleife also 15 Mal. Der Wert von *c* wird in Folge dessen auf das Intervall $[0, 15]$ approximiert.

Die Variable *b* hat im Zustand $\hat{S}'$ den Wert $[0, 38]$. In der inneren Schleife wird dieser pro Iteration um eins erhöht. So ergibt sich bei 15 Iterationen das Intervall von $[0, 15]$, das auf den Wert der äußeren Schleife addiert wird. So ergibt sich für die Variable *b* das Intervall $[0, 53]$, das die möglichen Werte innerhalb beider Schleifen beschreibt. Da der Wert in der äußeren Schleife durch eine Zuweisung auf einen festen Wert gesetzt worden ist, wird der errechnete Wert nicht durch den globalen Multiplikator 7 modifiziert.

Anders verhält es sich mit der Variable *d*. Sie wird ebenfalls pro Iteration um eins erhöht, wodurch der alte Wert ebenfalls um 15 pro Schleifenaufruf erhöht wird. Da der Wert von *d* jedoch in der äußeren Schleife nicht durch eine Zuweisung zurückgesetzt wird, wird die Werterhöhung um 15 insgesamt 7 Mal ausgeführt. Durch alle Aufrufe der inneren Schleife wird der Wert von *d* so um insgesamt 105 erhöht. Nachdem dieser Wert auf das Intervall der äußeren Schleife addiert wurde, ergibt sich insgesamt das Intervall $[0, 112]$ für die möglichen Werte der Variable *d*.

Der errechnete Zustand der inneren Schleife ist nun durch die folgende Menge gegeben:

$$\hat{S}'' = \{i \to [0, 21], a \to [0, 35], b \to [0, 53], c \to [0, 15], d \to [0, 112]\}$$

Der Zustand $\hat{S}''$ ist der finale Zustand, der nach der Analyse beider Schleifen für die weitere Berechnung der Abstrakten Interpretation zurückgegeben wird. Als Schleifeniterationsgrenzen wurde für die Schleife aus Zeile 9 das Intervall $[7, 7]$ und für die Schleife aus Zeile 13 das Intervall $[15, 15]$ ermittelt.

6.5 Approximationsgüte der statischen Analyse

Die Approximationsgüte der hier vorgestellten statischen Schleifenanalyse ist, verglichen mit der Analyse der modifizierten Abstrakten Interpretation, genau dann schlechter, wenn es Zuweisungsabhängigkeiten von Variablen gibt. Welche Auswirkungen diese Abhängigkeiten auf die Güte der

Approximation haben, wird im Verlauf dieses Abschnitts genauer erläutert. Außerdem wird das Intervall der Werte der Variablen recht groß aufgespannt, da es alle möglichen Werte der Variablen innerhalb der Schleife repräsentieren muss.

Der Zustand $\hat{S'''}$, der alternativ über die modifizierte Abstrakte Interpretation berechnet wurde, lautet:

$$\hat{S'''} = \{i \rightarrow [21, 21], a \rightarrow [35, 35], b \rightarrow [48, 48], c \rightarrow [15, 15], d \rightarrow [112, 112]\}$$

Alle Werte sind in diesem Fall exakt ermittelt worden, so dass ein Vergleich der Güte vorgenommen werden kann.

Werden die Zustände $\hat{S''}$ und $\hat{S'''}$ miteinander verglichen, so fällt auf, dass $\hat{S''}$ eine sichere Approximation für $\hat{S'''}$ darstellt. Die unteren Grenzen aus $\hat{S''}$ sind alle auf 0 gesetzt, obwohl diese laut $\hat{S'''}$ gleich der oberen Grenze sein könnten (mit Ausnahme der Variable b, da diese überapproximiert ist). Dieser Unterschied kommt dadurch zu Stande, dass bei der statischen Analyse ein Intervall ermittelt wird, das alle möglichen Zustände der Variablen innerhalb der Schleife darstellen muss. Für die Bestimmung von Schleifeniterationsgrenzen, im angegebenen Beispiel, hat diese Überapproximation jedoch keine direkten Auswirkungen.

Die zweite Überapproximation, die in diesem Beispiel durch die Auswertung der statischen Analyse entstanden ist, ist durch die Abhängigkeiten von Variablen entstanden. Die obere Grenze der Variable *b* entspricht in der Approximation $\hat{S''}$ dem Wert 53, wobei bei der exakteren Analyse der modifizierten Abstrakten Interpretation der Wert 48 als obere Grenze ermittelt worden ist. Dieser Unterschied entsteht dadurch, dass die statische Analyse für alle Variablen nacheinander bestimmt, welche möglichen Werte sie annehmen können. In dem Beispiel wird der Wert der Variable *b* vor der Variable *a* modifiziert, so dass der letzte gültige Wert der Variable *a* nicht in den Wert von *b* einfließen kann. Da die hier vorgestellte statische Analyse diese Abhängigkeit jedoch nicht berücksichtigen kann, ist die genannte Differenz der oberen Grenze entstanden.

Für viele einfache Schleifen ohne Zuweisungsabhängigkeiten ist es dennoch möglich, auch mit der statischen Analyse ein ebenso exaktes Ergebnis, wie das der modifizierten Abstrakten Interpretation, zu ermitteln.

Solch einfache Schleifen sind in realen Programmen natürlich eher selten zu finden. Daher wird die statische Schleifenanalyse in dieser Arbeit durch ein Program Slicing unterstützt, das das komplette Programm auf die Anweisungen reduziert, die den Kontrollfluss bezüglich der Schleifenanalyse beeinflussen. Auf diese Weise können sehr komplexe Schleifen häufig auf eine Form verkleinert werden, die mit der vorgestellten Analyse effizient berechenbar ist.

Auf Grund der beschriebenen Überapproximation, die eintreten kann, kann die statische Schleifenanalyse im entwickelten Programm durch einen Parameter aktiviert, bzw. deaktiviert werden. Über diesen kann so die Güte der gesamten Approximation gesteuert werden. Zusätzlich kann für die statische Analyse ausgewählt werden, ob eine Abhängigkeitsauflösung durchgeführt werden

soll. Falls diese deaktiviert ist, bricht die Analyse bereits in dem Fall ab, in dem eine Zuweisungsabhängigkeit zwischen mehreren Variablen vorliegt. So kann der Benutzer selbst entscheiden, ob Überapproximationen durch die statische Analyse gewünscht sind, oder nur einfache Schleifen mit diesem Verfahren analysiert werden sollen.

Die Überapproximation, die durch die statische Analyse entstehen kann, kann jedoch in manchen Fällen sinnvoll sein. Möchte der Benutzer beispielsweise eine Idee durch eine Ad-hoc Analyse belegen, so kann es ohne weiteres sein, dass die Überapproximation gegebenenfalls nicht relevant ist. Auch bei einfachen Optimierungen kann eine grobe Angabe der Schleifengrenzen meist schon ausreichend sein. Dafür ist die Analyse, wenn sie denn durchgeführt werden kann, in einem Bruchteil der Zeit, verglichen mit der Analyse der abstrakten Interpretation, durchgeführt.

Wie sich jedoch in der Evaluation aus Kapitel 8 gezeigt hat, können in Kombination mit einem geeigneten Program Slicing viele Schleifen ohne eine solche Überapproximation analysiert werden. Dabei wird zudem eine signifikante Reduzierung der Gesamtlaufzeit durch Integration der statischen Schleifenanalyse erreicht.

Kapitel 7

Aufbau des Loopanalyzers

Anhand der vorgestellten Konzepte der Abstrakten Interpretation (Kapitel 4), der Polytop-Berechnung (Kapitel 5) und der statischen Schleifenanalyse (Kapitel 6), wurde eine softwarebasierte Schleifenanalyse in der Programmiersprache C++ entwickelt, die in diesem Kapitel vorgestellt werden soll. Wie bereits in Kapitel 3 beschrieben wurde, wird die Schleifenanalyse dabei auf der plattformunabhängigen Zwischendarstellung *ICD-C* aufgebaut. Auf diese Weise kann die Schleifenanalyse für beliebige Zielsysteme und Programmiersprachen erweitert werden.

In Abschnitt 7.1 wird zunächst anhand eines Beispiels gezeigt, wie das entwickelte Programm eine Analyse auf Basis der modifizierten Abstrakten Interpretation durchführt. Im folgenden Abschnitt 7.2 wird der konzeptionelle Aufbau der entwickelten Schleifenanalyse beschrieben, ohne dabei auf die Implementierungsdetails der einzelnen Klassen einzugehen. Im Anschluss werden in Abschnitt 7.3 die Datenstrukturen vorgestellt, in denen die Werte der Variablen gespeichert werden, die innerhalb der Analyse ermittelt wurden. In Abschnitt 7.4 werden die wichtigsten Klassen der Schleifenanalyse im Detail vorgestellt. Die erstellte automatische Schleifenanalyse wurde einerseits direkt in den Kompiliervorgang des WCC's integriert, andererseits aber auch in einem Stand-Alone Tool umgesetzt. Wie die Integration und das Stand-Alone Tool aufgebaut sind, wird in Abschnitt 7.5 erläutert.

7.1 Ablauf der Analyse

In diesem Abschnitt soll beispielhaft beschrieben werden, wie der interne Ablauf einer Programmanalyse aufgebaut ist. Die wichtigsten Methoden, die nötig sind, um das gegebene Programm mit Hilfe der modifizierten Abstrakten Interpretation zu analysieren, sind in der Klasse *IR_AIAnalyzer* hinterlegt (eine genauere Beschreibung der Schnittstelle dieser Klasse befindet sich in Abschnitt 7.4.1). Wie die konzeptionelle Analyse der unterschiedlichen Anweisungstypen, auf Basis der modifizierten Abstrakten Interpretation durchgeführt wird, wurde bereits in Abschnitt 4.3 beschrieben.

Ziel der modifizierten Abstrakten Interpretation ist es, für jeden Programmpunkt, bzw. für jede Anweisung, den Zustand zu berechnen, der nach Ausführung der Anweisung gültig ist. Als Eingabe dient

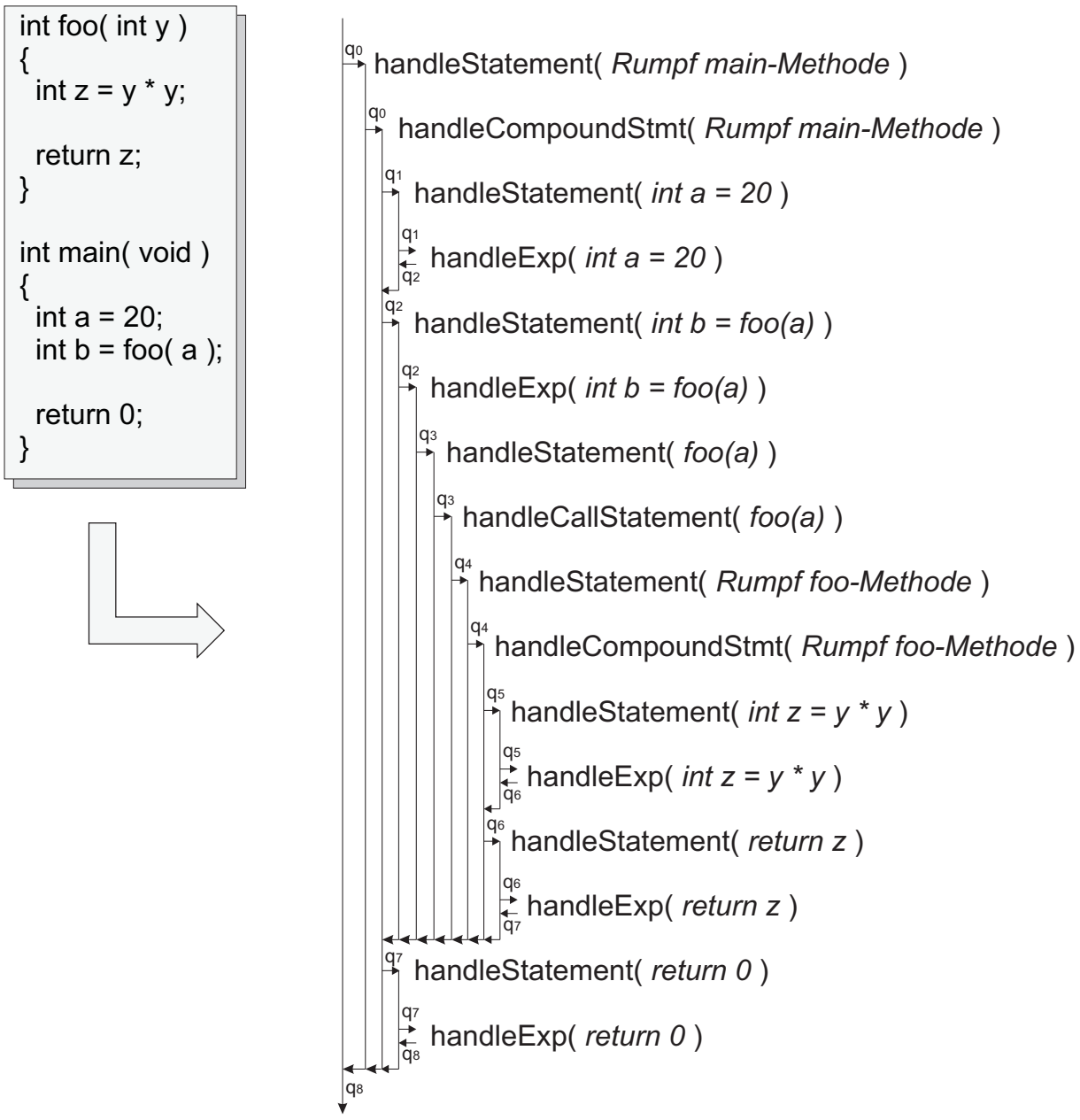


Abbildung 7.1: *Beispielprogramm zur Demonstration des Ablaufs der Analyse*

dabei der Zustand, der vor der Analyse der Anweisung gültig war. Diese Berechnung wird durch die Methode *handleStatement* der Klasse *IR_AIAnalyzer* durchgeführt, die als Argument u.a. die zu analysierende Anweisung und den Zustand erhält, der vor der Auswertung der Anweisung gültig war. Als Ergebnis liefert die Methode den Zustand, der nach der Analyse der Anweisung gültig ist.

Dabei wird, je nach Anweisungstyp, eine passende Analysemethode durch die Methode *handleStatement* aufgerufen, die z.B. die Auswertung einer If-Anweisung oder auch einer zusammengesetzten Compound-Anweisung durchführt. Die Methoden rufen im Verlauf der Analyse, abhängig vom Kontrollfluss, rekursiv die Methode *handleStatement* auf, um untergeordnete Anweisungen auszuwerten.

In Abbildung 7.1 sind ein Beispielprogramm mit dem zugehörigen Graph, der die Funktionsaufrufe der Analyse repräsentiert, dargestellt.

Gestartet wird die Analyse mit dem Rumpf der main-Methode als zu analysierende Anweisung. Der Zustand q_0 ist in diesem Fall leer, so dass für alle Variablen der Wert $\top$ gesetzt ist. Für den genannten Funktionsrumpf wird nun zunächst die Methode *handleStatement* aufgerufen. Diese ruft in der Folge die passende Methode auf, die für die Analyse des Anweisungstyps zuständig ist. Im Falle des Funktionsrumpfes ist es die Methode *handleCompoundStmt*, welche die Auswertung von zusammengesetzten Ausdrücken durchführt. Als Eingabe wird zusätzlich der Zustand q_0 übermittelt. Wie in Abschnitt 4.3 beschrieben, werden zusammengesetzte Anweisungen so ausgewertet, dass nacheinander alle eingebetteten Anweisungen analysiert werden. Dabei bekommt die erste Anweisung den Eingabezustand übergeben, für den die Analyse des zusammengesetzten Ausdrucks gestartet wurde. Dies ist in diesem Fall der Zustand q_0. Alle weiteren Anweisungen erhalten jeweils den Zustand als Eingabe, der durch Analyse der vorangegangenen Anweisung berechnet wurde. Als Ergebnis der Analyse des zusammengesetzten Zustands wird schließlich der Zustand zurückgegeben, der sich nach Auswertung der letzten Anweisung ergeben hat.

In dem Beispiel aus Abbildung 7.1 ruft die Methode *handleCompoundStmt* für den Rumpf der main-Methode die *handleStatement* Funktion für die drei Anweisungen *int a = 20*, *int b = foo(a)* und *return 0* auf. Die erste Anweisung kann durch einen Aufruf der Methode *handleExp* ausgewertet werden, wobei der resultierende Zustand der Auswertung als Eingabezustand für die Analyse der zweiten Anweisung übergeben wird. Die Anweisung *int b = foo(a)* wird ebenfalls über die Methode *handleExp* ausgewertet, wobei durch den rechten Teil des Ausdrucks ein Funktionsaufruf durchgeführt wird. Für die Anweisung *foo(a)* ruft die *handleStatement* Methode die spezialisierte Methode *handleCallStatement* auf, die u.a. die Funktionsparameter verarbeitet und die Analyse des Funktionsrumpfs startet. Dieser wird analog zur Main-Funktion ausgewertet, so dass nach der Analyse der Funktion der resultierende Zustand an die Analyse des Rumpfes der main-Methode zurückgegeben wird. Zum Schluss wird noch die return-Anweisung von der passenden Methode ausgewertet, bevor der resultierende Endzustand als Ergebnis der Analyse zurückgegeben werden kann.

In diesem Beispiel wurde bewusst auf die Integration einer Schleife verzichtet, da die einzelnen Iterationen eines Schleifenrumpfs den Graph, der die Funktionsaufrufe der Analyse repräsentiert,

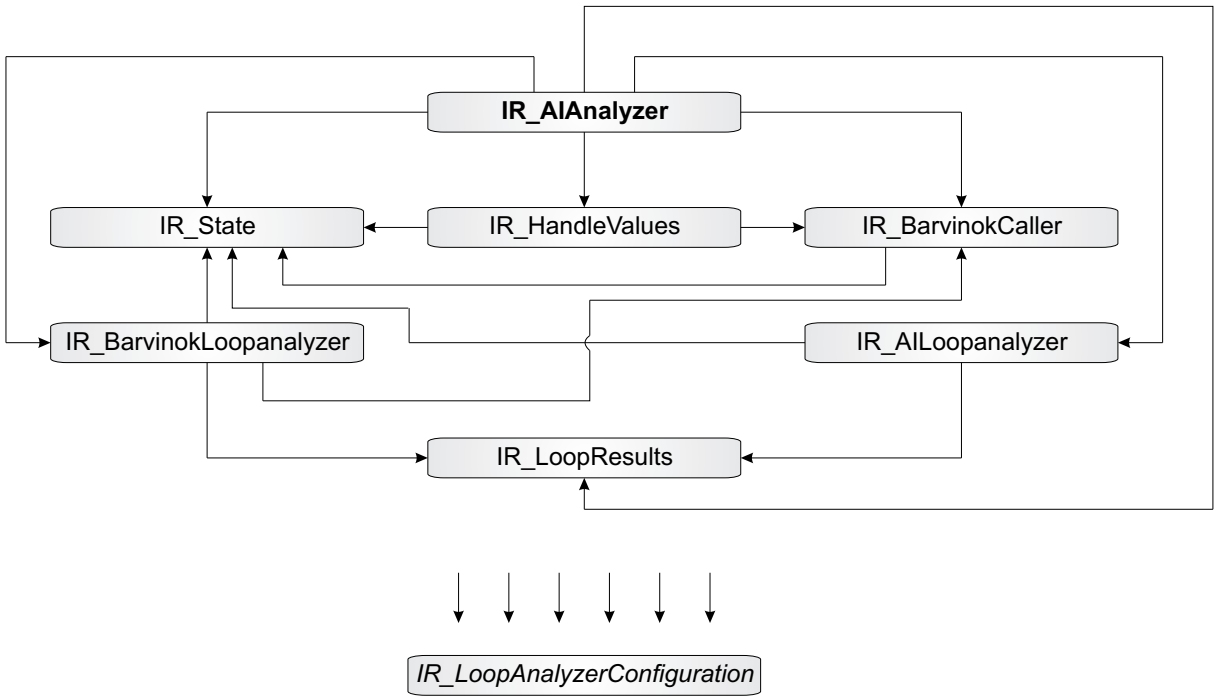


Abbildung 7.2: *Konzeptioneller Aufbau der Schleifenanalyse*

zu komplex anwachsen lassen würde. Der konzeptionelle Aufbau der Analyse von Schleifen wurde jedoch ausführlich in Abschnitt 4.3 beschrieben.

7.2 Konzeptioneller Aufbau

In diesem Abschnitt sollen die einzelnen Klassen, bzw. deren Kommunikation, erläutert werden, ohne dabei auf Details der Implementierung einzugehen. Alle hier vorgestellten Klassen werden später in Abschnitt 7.4 detaillierter erläutert. Der Aufbau der entwickelten automatischen Schleifenanalyse ist in Abbildung 7.2 zu sehen.

Die Controller-Klasse, welche die Analyse steuert, heißt *IR_AlAnalyzer*. Sie ist zudem die Schnittstelle, durch die die Analyse von außen gesteuert werden kann. Zusätzlich zu der Controller-Eigenschaft sind in dieser Klasse die Methoden implementiert, die benötigt werden, um die Anweisungen der ICD-C auf Basis der modifizierten Abstrakten Interpretation (siehe Abschnitt 4.3) auszuwerten. Alle globalen Einstellungen, die für mehrere Klassen relevant sind, sind in der Klasse *IR_LoopAnalyzerConfiguration* hinterlegt. Die Einstellungen werden in statischen Klassenvariablen gespeichert, so dass alle weiteren Klassen der Analyse ohne eine Instantiierung darauf zugreifen können.

Die Klasse *IR_State* repräsentiert die Zustände, die durch die Analyse ermittelt werden. In den Objekten dieser Klasse werden alle Werte gespeichert, die eine Variable, abhängig vom Programmpunkt, annehmen kann. Die *IR_State*-Objekte, werden fast von jeder anderen Klasse der Analyse benötigt, da die Berechnungen der modifizierten Abstrakten Interpretation kontextabhängig, also

abhängig vom aktuellen Programmzustand, sind.

Erreicht die Analyse eine Anweisung, die aus einem Ausdruck besteht, so wird eine Instanz der Klasse *IR_HandleValues* erzeugt, die den möglichen Wert eines Ausdrucks, abhängig vom übergebenen Zustand, auf Ebene der gewählten Abstrakten Domäne auswertet.

Kann eine Bedingung einer Programmverzweigung auf Ebene der Abstrakten Domäne nicht ausgewertet werden, bzw. wenn die Auswertung den Wert $\top$ liefert, so wird ein passendes Polytop erzeugt, mit dessen Hilfe der Wertebereich der Variablen eingeschränkt wird (siehe Kapitel 5). Die Methoden, die nötig sind um die Ungleichungssysteme der Polytope zu erstellen und mit Hilfe der Polylib auszuwerten, sind in der Klasse *IR_BarvinokCaller* umgesetzt.

Wird in der Analyse, die durch das Objekt der Klasse *IR_AIAnalyzer* durchgeführt wird, eine Schleife gefunden, so wird diese, je nach Konfiguration der Schleifenanalyse, zuerst mit einer statischen Analyse, so wie sie in Kapitel 6 beschrieben wurde, ausgewertet. Diese statische Analyse ist in der Klasse *IR_BarvinokLoopAnalyzer* umgesetzt. Schlägt die statische Analyse fehl, oder soll sie auf Grund der eingestellten Konfiguration übersprungen werden, so wird die Schleife auf Grundlage der modifizierten Abstrakten Interpretation ausgewertet. Dazu wurden, wie in Abschnitt 4.3 beschrieben, zwei alternative Analysemethoden entwickelt, die über die Konfiguration ausgewählt werden können. Beide Varianten sind in der Klasse *IR_AILoopAnalyzer* implementiert.

Die Ergebnisse der Schleifenauswertungen werden in einer Instanz der Klasse *IR_LoopResults* festgehalten. Diese Containerklasse sammelt und kategorisiert alle Ergebnisse, damit diese am Ende der Analyse in das *IR*-Objekt der ICD-C eingetragen und an den Flow Fact Manager (siehe Abschnitt 3.1.4) übergeben werden können.

7.3 Datenstrukturen zur Speicherung der Werte der Abstrakten Interpretation

Jeder Zustand, der durch ein Objekt der Klasse *IR_State* instantiiert ist, enthält für jede Variable dessen mögliche Werte, abhängig vom gegebenen Programmpunkt. Die Datenstruktur, in der diese Werte gespeichert werden, ist in Abbildung 7.3 zu sehen.

Alle Objekte, die in der Abbildung mit einem * markiert sind, sind abstrakte Klassen. Alle weiteren Klassen sind entsprechend implementierte Konkretisierungen. Das allgemeinste Element der Datenstruktur ist durch die abstrakte Klasse *IR_ApproxValue* gegeben. Jeder Wert, der einer Variable in einem Zustand *IR_State* zugewiesen werden kann, ist ein Objekt, das von dieser Klasse erbt. Als entsprechende Konkretisierungen stehen die Klassen *IR_ArrayApproxValue*, *IR_StructApproxValue* und *IR_ScalarApproxValue* zur Verfügung. Die beiden erstgenannten repräsentieren zusammengesetzte Objekte wie Arrays und Structs, so wie sie im C99-Standard definiert sind. Für die Objekte beider Klassen gilt, dass sie wiederum aus weiteren Objekten der Klasse *IR_ApproxValue* zusammengesetzt sind. Alle skalaren Objekte, wie beispielsweise Integer oder Floating-Point Variablen,

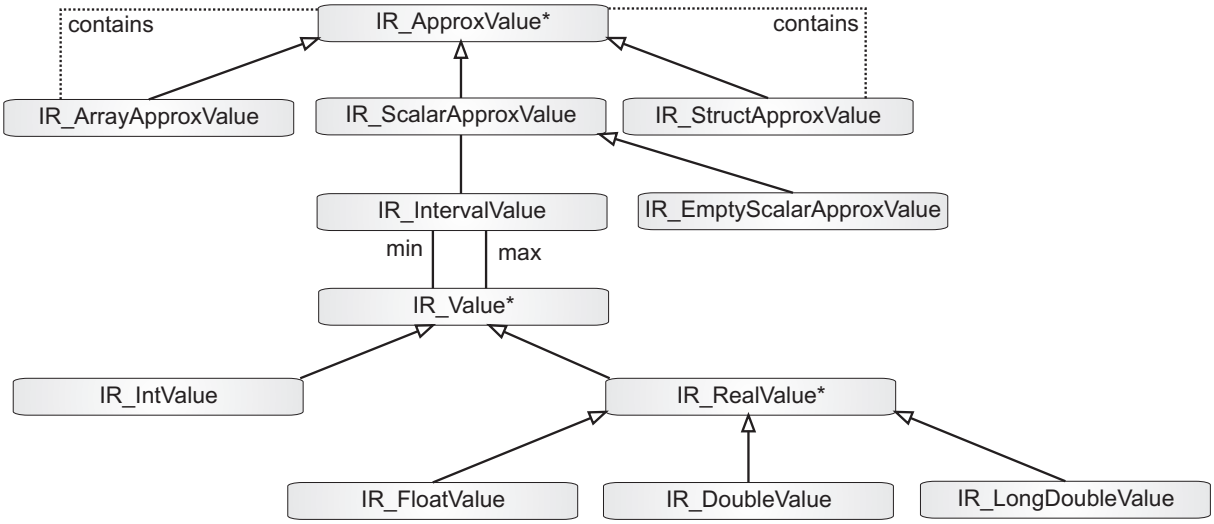


Abbildung 7.3: *Datenstruktur der Schleifenanalyse*

werden durch Objekte der Klasse *IR_ScalarApproxValue* repräsentiert.

Da die Analyse auf der plattformunabhängigen Zwischendarstellung *ICD-C* aufgebaut ist (siehe Abschnitt 3.1.1), kann die Schleifenanalyse für einen gegebenen Anlass entsprechend erweitert werden. Falls erforderlich, können weitere zusammengesetzte Typen dadurch repräsentiert werden, dass diese als weitere Spezialisierung der Klasse *IR_ApproxValue* implementiert werden.

Die angesprochene Klasse *IR_ScalarApproxValue* repräsentiert alle skalaren Werte und hat die Operatoren, die laut C-99 Standard auf Variablen definiert sind, überschrieben, so dass diese unabhängig vom Typ der Variable angewendet werden können. Auf Programmebene können so alle Operationen auf den Objekten durchgeführt werden, ohne das vorher geprüft werden muss, von welchem Typ die skalaren Variablen sind oder welche abstrakte Domäne implementiert ist. In dieser Arbeit wurde die Intervalldarstellung als Abstrakte Domäne umgesetzt, so dass jedes Objekt der Klasse *IR_ScalarApproxValue* ein Objekt der Klasse *IR_IntervalValue* enthält, in dem das Intervall des jeweiligen Wertes gespeichert ist.

Weitere Abstrakte Domänen könnten dadurch integriert werden, dass die Klasse *IR_ScalarApproxValue* durch eine abstrakte Klasse ersetzt wird. Die aktuelle Klasse *IR_ScalarApproxValue* würde von dieser neuen abstrakten Klasse erben und z.B. durch den Namen *IR_ScalarIntervalApproxValue* ersetzt werden. Alle weiteren Abstrakten Domänen würden schließlich von der dann abstrakten Klasse *IR_ScalarApproxValue* erben, so dass auch dort die jeweils wichtigen Abstrakten Operatoren implementiert werden könnten. Da in dieser Arbeit jedoch nur die Intervall-Domäne umgesetzt worden ist, wurde diese direkt in der konkretisierten Klasse *IR_ScalarApproxValue* umgesetzt.

Ein Spezialfall der skalaren Werte stellt die Klasse *IR_EmptyScalarApproxValue* dar. Diese Klasse enthält kein Intervall, da in ihr keine Werte gespeichert werden sollen. Diese Klasse kann für alle Datentypen benutzt werden, deren Wert für die Analyse nicht von Bedeutung ist. So wird beispielsweise für die Dereferenzierung der Pointer die Alias-Analyse der ICD-C benutzt. Die Werte, die ei-

nem Pointer zugewiesen werden, brauchen innerhalb der Schleifenanalyse nicht berechnet werden, da diese Informationen bereits durch die Alias-Analyse ermittelt wurden. Um innerhalb der Schleifenanalyse nicht an jeder Stelle überprüfen zu müssen, ob eine Operation ausgeführt werden kann, wurde diese leere Containerklasse entwickelt, in der ebenfalls jeder Operator überschrieben wurde. Der Unterschied besteht bei dieser Klasse allerdings darin, dass die Ausführung dieser Operatoren keine Änderung des Programmzustands hervorrufen.

Wenn Operationen auf Objekten der eigentlichen Klasse *IR_ScalarApproxValue* durchgeführt werden, so werden diese Operationen an das integrierte Intervall des Objekts weitergegeben, da auch dort alle Operatoren überladen sind. Auf dieser Ebene sind schließlich die Abstrakten Operatoren der Abstrakten Domäne implementiert, so wie sie für die Intervalldarstellung definiert sind. Jedes Intervall der Klasse *IR_IntervalValue* enthält wiederum zwei Objekte der Klasse *IR_Value*, die die obere und untere Grenze des Intervalls bilden. Die Klasse *IR_Value* ist ebenfalls eine abstrakte Klasse, die durch die Implementierung der Klassen *IR_IntValue* und *IR_RealValue* konkretisiert wird. Auch in der *IR_Value*-Klasse sind alle möglichen Operatoren durch abstrakte Methoden überschrieben, so dass auch die Intervall-Klasse, ohne Kenntnis der Implementierung, Operationen auf seinen Grenz-Objekten durchführen kann.

Die Klasse *IR_IntValue* repräsentiert alle ganzzahligen Werte, unabhängig davon, ob diese vorzeichenlos oder vorzeichenbehaftet sind. Etwas komplizierter ist die Umsetzung der Fließkommazahlen. Dort wird durch die Klasse *IR_RealValue* ebenfalls eine abstrakte Oberklasse eingeführt, die von den jeweils definierten Typen konkretisiert wird, da die unterschiedlichen Fließkommazahlen nicht mit Hilfe von Transformationen in einer Klasse repräsentiert werden können. Nach Definition des C99-Standards gibt es die vordefinierten Fließkommatypen Float, Double und Long Double, die in den Klassen *IR_FloatValue*, *IR_DoubleValue* und *IR_LongDoubleValue* umgesetzt worden sind.

Mit Hilfe dieses modularen Aufbaus ist es möglich, beliebige weitere Datentypen in die Schleifenanalyse zu integrieren.

7.4 Relevante Klassen

In diesem Abschnitt sollen die Klassen, die in Abschnitt 7.2 eingeführt wurden, im Detail vorgestellt werden. Zu jeder Klasse ist dabei ein Klassendiagramm angegeben, in dem aus Komplexitätsgründen ausschließlich die Public-Methoden angegeben sind, die die Schnittstellen der jeweiligen Klassen bilden. Eine weiterführende Erläuterung der Klassen würde den Rahmen dieser Arbeit überschreiten. Kursiv geschriebene Methoden in den Klassendiagrammen sind statisch, so dass diese ohne vorherige Instantiierung der Klasse benutzt werden können.

7.4.1 IR_AIAnalyzer

Das Klassendiagramm der Klasse *IR_AIAnalyzer* ist in Abbildung 7.4 zu sehen.

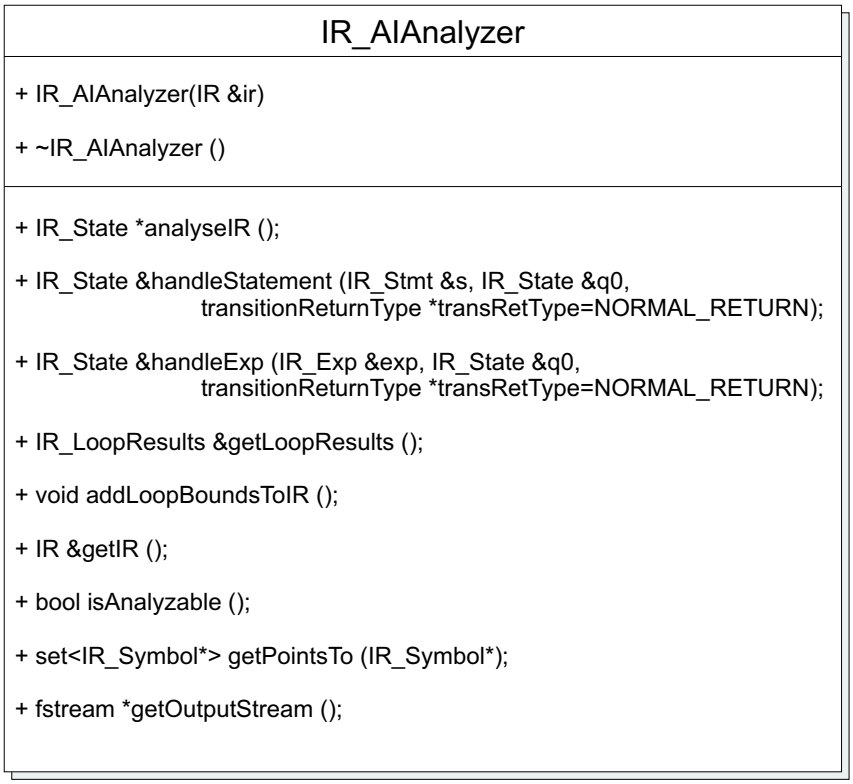


Abbildung 7.4: *Klasse IR_AIAnalyzer*

Die Klasse *IR_AIAnalyzer* ist der Controller, der die komplette Analyse steuert. Als Argument wird dem Konstruktor die vorher eingelesene *IR*, die das vollständig zu analysierende C-Programm auf Basis der *ICD-C Zwischendarstellung* repräsentiert, übergeben.

Über die Methode *analyseIR()* wird die Analyse des Programms gestartet, welches dem Konstruktor als *IR*-Objekt übergeben worden ist. In dieser Funktion, wird zunächst mit Hilfe der Methode *isAnalyzable()* geprüft, ob das übergebene Programm analysiert werden kann. Ist dies der Fall, so wird die Analyse durchgeführt, wobei der Zustand, der nach der Terminierung des Programms am End-Knoten gültig ist, als Ergebnis zurückgegeben wird. Konnte die Analyse nicht durchgeführt werden, so wird statt eines Objekts der Klasse *IR_State* ein Null-Pointer zurückgegeben, so dass außerhalb der Analyse geprüft werden kann, ob die Analyse erfolgreich war. Allgemein gilt, dass alle zurückgegebenen Zustände im Freispeicher erstellt werden. Daher sind diese über den *delete*-Operator wieder freizugeben, sobald diese nicht mehr benötigt werden.

Die Methode *handleStatement(IR_Stmt &s, IR_State &q0, transitionReturnType *transRetType=NORMALRETURN)* wertet jede Anweisung der *ICD-C* auf Basis der Abstrakten Interpretation aus. Als Eingabe erwartet die Methode eine Referenz der zu analysierende Anweisung (*s*), den Zustand, der vor dieser Anweisung gültig ist (*q0*) und einen Pointer vom Typ *transitionReturnType*. Der Typ *transitionReturnType* ist ein Aufzählungstyp, durch den während der Analyse geprüft werden kann, ob die Auswertung einer Anweisung, z.B. durch eine Break-Anweisung, unterbrochen worden ist. Dies ist vor allem für die Analyse von Compound-Anweisungen wichtig, damit bekannt ist, ob die

noch folgenden Anweisungen ebenfalls analysiert werden müssen. Die Methode *handleStatement* prüft den Typ der übergebenen Anweisung durch eine Folge von dynamischen Casts, um anhand des Typs die entsprechende Methode aufzurufen, mit der die Anweisung ausgewertet werden kann. Dazu gehören z.B. die privaten Methoden *handleIf(IR_Stmt &s, IR_State &q0, transitionReturnType *transRetType=NORMAL_RETURN)* oder *handleLoopStmt(IR_Stmt &s, IR_State &q0, transitionReturnType *transRetType=NORMAL_RETURN)*, die in dem Klassendiagramm nicht dargestellt sind. Als Ergebnis liefert die Methode *handleStatement*, wie auch jede Spezialisierung dieser Methode, den Zustand, der nach Ausführung der Anweisung gültig ist, zurück.

Wird während der Analyse eine Anweisung gefunden, die einen Ausdruck enthält (*IR_ExpStmt*), so wird in der analysierenden Methode die Methode *handleExp(IR_Exp &exp, IR_State &q0, transitionReturnType *transRetType=NORMAL_RETURN)* aufgerufen. Diese ist analog zu der Methode *handleStatement* aufgebaut, so dass sie über dynamische Casts die entsprechende konkretisierte Funktion zur Berechnung des Folgezustands aufruft.

Nachdem die Analyse des *IR*-Objekts abgeschlossen ist, kann durch die Funktion *getLoopResults()* das Objekt der Klasse *IR_LoopResults* zurückgegeben werden, in dem sich alle Ergebnisse der analysierten Schleifen befinden. Des Weiteren können nach Terminierung der Analyse die Schleifeniterationsgrenzen mit der Methode *addLoopBoundsToIR()* an die *IR* übertragen werden. Die gefundenen Grenzen werden durch diese Methoden ebenfalls an den *Flow Fact Manager* übergeben, damit auch andere Analyseverfahren des WCC's darauf zugreifen können.

Zusätzlich hat die Klasse *IR_AIAnalyzer* einige weitere Methoden, über die der Benutzer auf diverse Objekte zugreifen kann.

Die Ausgaben der Analyse werden in einen Ausgabestream geschrieben, der in der Konfigurationsklasse *IR_LoopAnalyzerConfiguration* definiert wird. Mit Hilfe der Methode *getOutputStream()* kann für eine Ausgabe auf diesen Stream zugegriffen werden. Damit auch andere Klassen Ausgaben erzeugen können, wurde diese Methode öffentlich gemacht.

Die letzte erwähnenswerte Methode, ist die Methode *getPointsTo(IR_Symbol *)*. Diese wird immer dann aufgerufen, wenn ein Pointer zu dereferenzieren ist. Über das Argument wird das Symbol des zu dereferenzierenden Pointers übergeben, so dass dieser mit Hilfe der bereits erwähnten Alias-Analyse der ICD-C ausgewertet werden kann (siehe Abschnitt 3.1.1.2). Als Ergebnis liefert die Methode eine Menge an möglichen Zielen, auf die der übergebene Pointer zeigen kann.

7.4.2 IR_HandleValues

Das Klassendiagramm der Klasse *IR_HandleValues* ist in Abbildung 7.5 zu sehen.

Die Klasse *IR_HandleValues* wird genau dann instantiiert, wenn eine der anderen Klassen die möglichen Werte eines Ausdrucks, abhängig vom Zustand des Programmpunkts, ermitteln muss. Wie auch fast jeder anderen Klasse, wird dem Konstruktor eine Referenz auf das Objekt der Con-

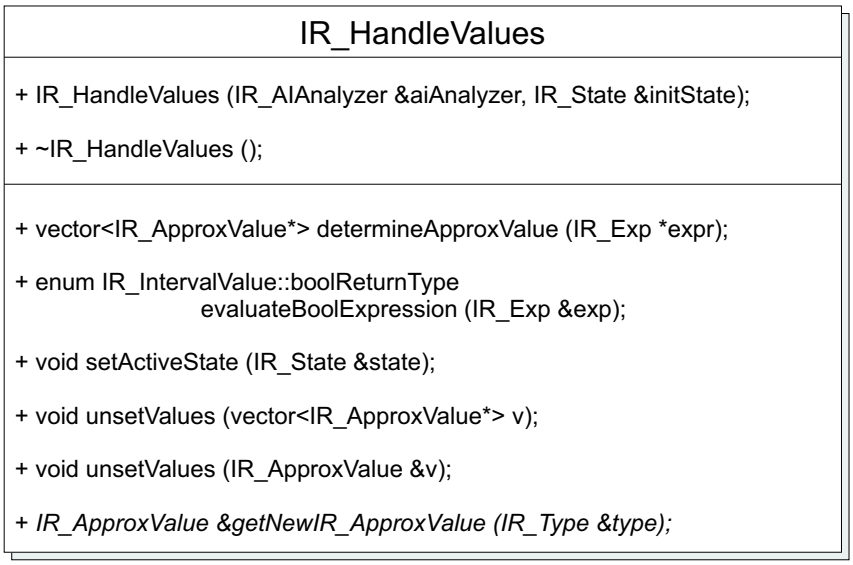


Abbildung 7.5: *Klasse IR_HandleValues*

trollerklasse *IR_AIAnalyzer* übergeben. Über diese Referenz kann z.B. auf das Objekt der zu analysierenden *IR* zugegriffen werden. Zusätzlich wird dem Konstruktor der initiale Zustand übergeben, der bei der Instantiierung der Klasse gültig ist.

Falls sich der Zustand außerhalb der Klasse *IR_HandleValues* ändert, so kann dieser über die Methode *setActiveState(IR_State &state)* aktualisiert werden.

Um mit Hilfe dieser Klasse einen Ausdruck auszuwerten, wird auf dem Objekt die Methode *determineApproxValue(IR_Exp *expr)* aufgerufen. Diese bekommt als Argument den auszuwertenden Ausdruck übergeben. Dieser kann sich auch aus komplexeren Ausdrücken zusammensetzen, so dass, ähnlich wie bei der Methode *handleStatement* der Klasse *IR_AIAnalyzer*, über dynamische Casts der Typ des Ausdrucks ermittelt wird. Anschließend wird die passende private Methode aufgerufen, die den entsprechenden Ausdruckstyp auf Ebene der Abstrakten Domäne auswertet. Als Ergebnis liefert die Methode einen Vektor an möglichen *IR_ApproxValue* Objekten. Wurde z.B. über einen Index-Ausdruck auf ein verschachteltes Array zugegriffen, so kann die Rückgabe aus mehreren möglichen Werten bestehen, so dass die Rückgabe eines einzelnen Objekts der Klasse *IR_ApproxValue* nicht ausreichen würde.

Die zurückgegebenen Werte der Methode *determineApproxValue* setzten sich teilweise aus temporären Objekten, teilweise aber auch aus Werten, die in einem Objekt der Klasse *IR_State* gespeichert sind, zusammen. Um alle überflüssigen Werte aus dem Freispeicher zu entfernen, sollte nach Abschluss jeder Bearbeitung eine der möglichen *unsetValues* Methoden aufgerufen werden, die je nach Argumenttyp überladen ist. Diese löscht automatisch nur die temporären Objekte, ohne dabei die Objekte freizugeben, die noch in der weiteren Analyse benötigt werden. Auf den *IR_ApproxValue* Objekten, die so ermittelt werden, können zustandsverändernde Operationen ausgeführt werden.

Für die Auswertung von boolschen Ausdrücken auf Ebene der Abstrakten Domäne, bietet die Klas-

se *IR_HandleValues* die Methode *evaluateBoolExpression(IR_Exp &exp)*, die den auszuwertenden Ausdruck als Argument übergeben bekommt. Als Rückgabe wird ein Objekt des Aufzählungstyps *IR_IntervalValue::boolReturnType* erstellt, das nach Vorgabe der Abstrakten Interpretation einen der Werte *ALWAYS_TRUE* ($true$), *ALWAYS_FALSE* ($false$) oder *UNKNOWN* ($\top$) annehmen kann.

7.4.3 IR_AILoopAnalyzer

Das Klassendiagramm der Klasse *IR_AILoopAnalyzer* ist in Abbildung 7.6 zu sehen.

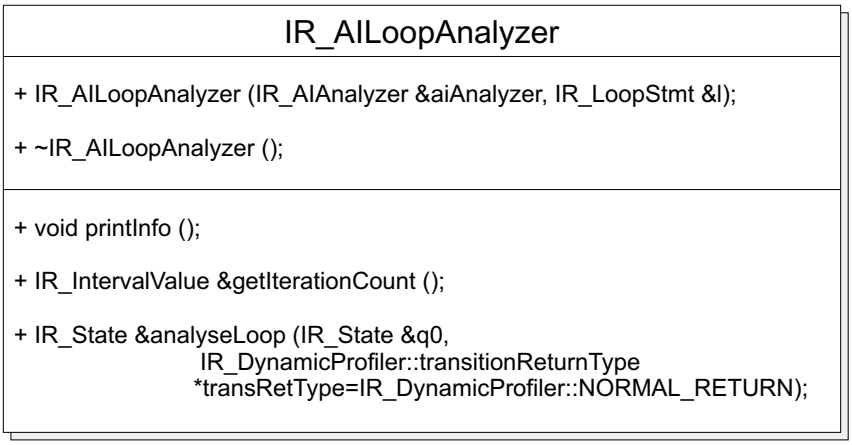


Abbildung 7.6: *Klasse IR_AILoopAnalyzer*

Die Klasse *IR_AILoopAnalyzer* stellt die Methoden bereit, die nötig sind, um eine Schleife auf Grundlage der vorgestellten modifizierten Abstrakten Interpretation zu analysieren. Wird in dem Controller *IR_AIAnalyzer* während der Analyse einer *handleStatement*-Methode eine Schleife gefunden, so wird ein Objekt der Klasse *IR_AILoopAnalyzer* erstellt. Abhängig von der Konfiguration der Schleifenanalyse kann es auch sein, dass stattdessen ein Objekt der Klasse *IR_BarvinokLoopAnalyzer* erstellt wird, um vorerst zu versuchen, die Schleife statisch zu analysieren.

Soll jedoch eine Analyse auf Basis der modifizierten Abstrakten Interpretation durchgeführt werden, so erwartet der Konstruktor als erstes Argument eine Referenz der Controllerklasse *IR_AIAnalyzer*. Als zweites folgt eine Referenz auf die auszuwertende Schleife (*l*).

Die Schleife wird nun durch einen Aufruf der Methode *analyseLoop(IR_State &q0, IR_AIAnalyzer::transitionReturnType *transRetType=IR_AIAnalyzer::NORMAL_RETURN)* analysiert. Als Eingabe wird dabei, wie für die Analyse jeder anderen Anweisung auch, eine Referenz auf den gültigen Start-Zustand, sowie ein Objekt des Aufzählungstyps transitionReturnType der Klasse *IR_AIAnalyzer* übergeben. Abhängig von der gewählten Konfiguration wird die Schleife nun mit einem der beiden möglichen Verfahren der modifizierten Abstrakten Interpretation ausgewertet. Nachdem die Analyse der Schleife abgeschlossen ist, wird der ermittelte Zustand, der nach Abarbeitung der Schleife gültig ist, zurückgegeben. Nun kann durch die Methode *getIterationCount()* auf die ermittelten Schleifengrenzen zugegriffen werden. Auch hier wird das erstellte Intervall im Freispeicher abgelegt, so dass

dieses nach der Benutzung wieder gelöscht werden sollte.

Die Methode *printInfo()* schreibt diverse Informationen über die zu analysierende Schleife in die Standardausgabe.

7.4.4 IR_BarvinokLoopAnalyzer

Das Klassendiagramm der Klasse *IR_BarvinokLoopAnalyzer* ist in Abbildung 7.7 zu sehen.

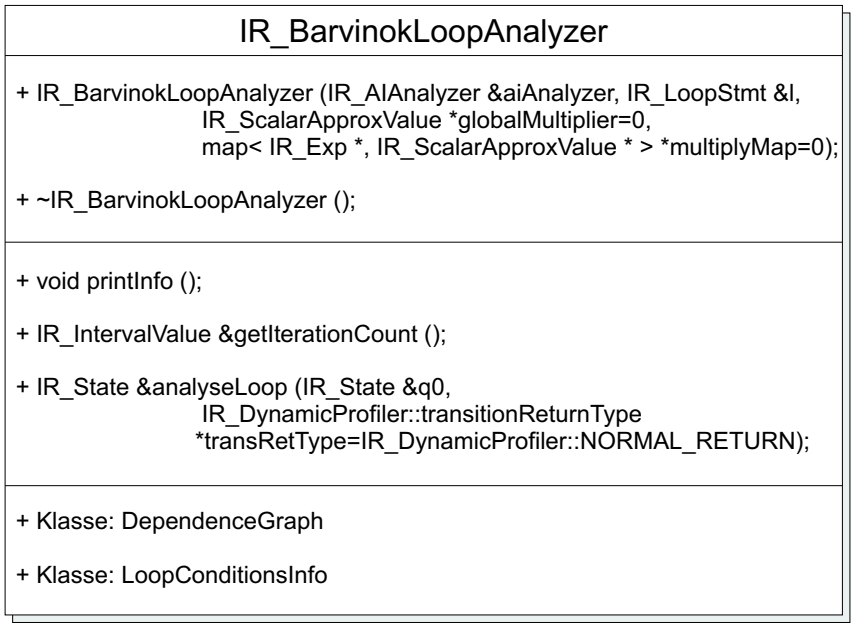


Abbildung 7.7: *Klasse IR_BarvinokLoopAnalyzer*

In dieser Klasse ist das statische Schleifenanalyseverfahren implementiert, das in Kapitel 6 vorgestellt worden ist. Falls dieses Verfahren nicht durch die Konfiguration deaktiviert wurde, wird ein Objekt der Klasse erzeugt, sobald die Methode *handleStatement* der Klasse *IR_AIAnalyzer* bei der Analyse eine Loop-Anweisung findet.

Auf Grund der Komplexität dieser Klasse, sind zusätzlich die beiden lokalen Klassen *DependenceGraph* und *LoopConditionsInfo* in dieser Klasse implementiert. Erstere ist dafür zuständig, den Abhängigkeitsgraphen aufzubauen, um aus diesem eine topologische Sortierung der Berechnungsreihenfolge der Variablen zu bilden. Die Zweite ist dafür zuständig, die zu analysierende Schleife nach Verzweigungen zu durchsuchen und mit Hilfe der Polylib die Anzahl der Punkte innerhalb der Schleife zu bestimmen.

Der Konstruktor der Klasse *IR_BarvinokLoopAnalyzer* ist ähnlich aufgebaut, wie der Konstruktor der Klasse *IR_AILoopAnalyzer*. Als erstes Argument bekommt er eine Referenz auf die Controllerklasse *IR_AIAnalyzer* übergeben. Des Weiteren wird eine Referenz auf die zu analysierende Schleife (*l*) übergeben. Zusätzlich zu den bereits genannten Argumenten erhält der Konstruktor der statischen

Schleifenanalyse einen globalen Multiplikator (*globalMultiplier*), der angibt, wie häufig diese Schleife, durch Einbettung in andere Schleifen, aufgerufen wird. Da jedoch nicht jede Variablenmodifikation gleich behandelt werden kann, wird durch die *multiplyMap* eine Zuordnung zwischen Variable und Multiplikator übergeben. Wird die statische Schleifenanalyse für die äußerste zu analysierende Schleife aufgerufen, so können die letzten beiden Parameter ausgelassen werden. Diese werden in diesem Fall durch Null-Pointer repräsentiert und automatisch im Konstruktor passend initialisiert.

Um die Schleife nun mit der statischen Schleifenanalyse auszuwerten, wird die Methode *analyseLoop(IR_State &q0, IR_AIAnalyzer::transitionReturnType *transRetType=IR_AIAnalyzer::NORMAL_RETURN)* aufgerufen. Sie bekommt eine Referenz auf den Start-Zustand (*q0*), sowie einen Pointer des *transitionReturnType's* übergeben. Nachdem diese Methode aufgerufen wurde, wird die komplette Analyse durchgeführt, so wie sie in Kapitel 6 beschrieben wurde.

Wie auch bei der Klasse *IR_AILoopAnalyzer* können die Schleifeniterationsgrenzen nach der Analyse durch die Methode *getIterationCount()* zurückgegeben werden. Auch die *printInfo()* Methode, welche die Informationen zur analysierten Schleife ausgibt, ist in dieser Klasse vorhanden.

7.4.5 IR_BarvinokCaller

Das Klassendiagramm der Klasse *IR_BarvinokCaller* ist in Abbildung 7.8 zu sehen.

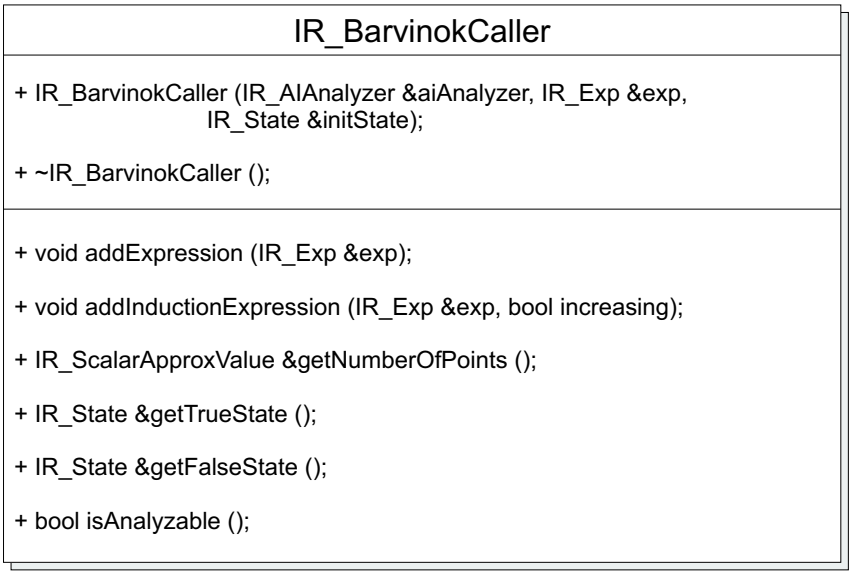


Abbildung 7.8: *Klasse IR_BarvinokCaller*

Die Klasse *IR_BarvinokCaller* baut die Ungleichungssysteme der Bedingungen auf, welche der Polylib als Eingabe für die Polytop-Berechnungen dienen. Außerdem ist die vollständige Socket-Kommunikation zwischen der Schleifenanalyse und dem WCC in dieser Klasse implementiert, die in Abschnitt 5.4.1 beschrieben worden ist.

Der Konstruktur der Klasse *IR_BarvinokCaller* erwartet eine Referenz auf das Objekt der Controller-

Klasse *IR_AIAnalyzer*. Zusätzlich werden der auszuwertende Ausdruck (*exp*) und der initiale Zustand (*initState*) übergeben, der für die Auswertung der Bedingung gültig ist.

Zusätzlich zu der Bedingung, die bereits im Konstruktor übergeben worden ist, können weitere einschränkende Ausdrücke mit der Methode (*addExpression(IR_Exp &exp)*) übergeben werden. Soll mit dem erstellten Objekt des Barvinok-Callers die Anzahl der Punkte eines Polytops ermittelt werden, so muss die Induktionsvariable festgelegt werden. Dies wird über die Methode *addInductionExpression(IR_Exp &exp, bool increasing)* erreicht, wobei für die Referenz der Induktionsvariable (*exp*) zusätzlich noch angegeben werden muss, ob die Induktionsvariable innerhalb der Schleife wächst oder erniedrigt wird. Diese Angabe wird über den boolschen Parameter (*increasing*) angegeben.

Nachdem die zu analysierende Bedingung über die vorgestellten Methoden definiert worden ist, kann über die Methode *getNumberOfPoints()* die Berechnung der ganzzahligen Punkte innerhalb des Polytops gestartet werden. Sollen hingegen die möglichen Werte der Variablen, abhängig von einer Verzweigung, eingeschränkt werden, so können nacheinander die Methoden *getTrueState()* und *getFalseState()* aufgerufen werden. Diese liefern als Rückgabe die jeweils gültigen Folgezustände $\hat{S}_{q_{true}}$ und $\hat{S}_{q_{false}}$.

Über die Methode *isAnalyzable()* kann im Vorfeld geprüft werden, ob die angegebene Bedingung mit Hilfe der Polylib ausgewertet werden kann.

7.4.6 IR_State

Das Klassendiagramm der Klasse *IR_State* ist in Abbildung 7.9 zu sehen.

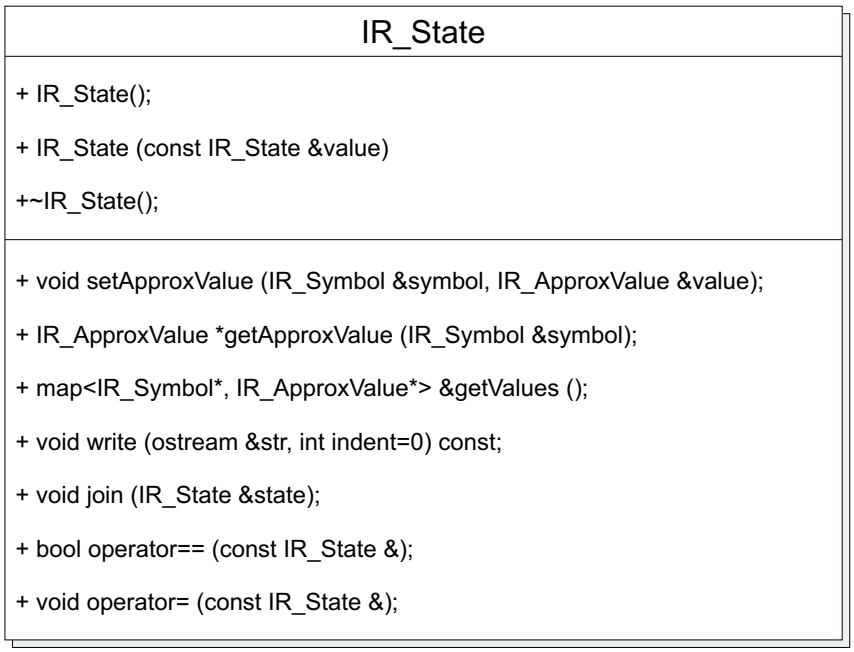


Abbildung 7.9: *Klasse IR_State*

Durch die Objekte der Klasse *IR_State* werden die Zustände beschrieben, die den Programmpunkten des zu analysierenden Programms zugeordnet werden. In diesen Objekten ist eine Zuordnung vorhanden, die den Variablen ihre möglichen Werte zuweist.

Um ein solches Objekt zu erstellen, kann der Konstruktor *IR_State()* verwendet werden, der ohne Argumente aufgerufen wird. In diesem Fall wird ein leerer Zustand erstellt, der noch für keine Variable eine Wertzuweisung hinterlegt hat. Alternativ kann ein Zustand durch den Copy-Konstruktur *IR_State (const IR_State &value)* als Kopie eines anderen Zustands erzeugt werden.

Um dem Zustand Elemente hinzuzufügen, kann die Methode *setApproxValue(IR_Symbol &symbol, IR_ApproxValue &value)* aufgerufen werden, die sowohl das Symbol der zu speichernden Variable, als auch den approximierten Wert als Argumente erwartet. Die durch diese Methode hinterlegten Zuordnungen werden intern in einer Map gespeichert. Ist dem Symbol aktuell noch kein Wert zugeordnet, so wird es mit dem neuen Wert zur Map hinzugefügt. Befindet sich bereits ein Wert innerhalb der Map, so wird der alte Wert durch den neuen ersetzt. Jedes Symbol wird also nur einmal pro Zustand verwaltet.

Soll ein approximierter Wert anhand eines Symbols ermittelt werden, so kann die Methode *getApproxValue(IR_Symbol &symbol)* aufgerufen werden. Die Methode durchsucht die interne Map nach dem übergebenen Symbol und gibt, falls vorhanden, den hinterlegten, approximierten Wert zurück.

Falls für eine Analyse die vollständige Zuordnung benötigt wird, so kann die gesamte Map über die Methode *getValues()* ausgelesen werden. Die Methode *write(ostream &str, int indent=0)* ermöglicht es dem Benutzer den aktuellen Zustand an den übergebenen Stream (*str*) zu senden.

Da für die Abstrakte Interpretation Zustände auch zusammengeführt werden müssen, ist die Methode *join(IR_State &state)* implementiert worden, welche die Werte des aktuellen Objekts mit den Werten des übergebenen Zustands vereinigt.

7.4.7 IR_LoopResults

Das Klassendiagramm der Klasse *IR_LoopResults* ist in Abbildung 7.10 zu sehen.

In dieser Klasse werden alle ermittelten Schleifeniterationsgrenzen gesammelt, die während der Analyse durch Objekte der Klassen IR_AILoopAnalyzer und IR_BarvinokLoopAnalyzer ermittelt worden sind.

Um ein Objekt dieser Klasse zu erstellen, steht der leere Konstruktor *IR_LoopResults()* zur Verfügung. Dieser initialisiert die interne Map, in der die ermittelten Informationen abhängig von den Loop-Anweisungen gespeichert werden. Um die Informationen, die zu den Analysen gespeichert werden können so variabel wie möglich zu halten, wird jeder Loop-Anweisung ein Struct zugewiesen, auch wenn dieses aktuell nur aus den Schleifeniterationsgrenzen besteht. Dieses Struct-Element kann so später um weitere Elemente erweitert werden, die mit der Analyse zusammenhängen.

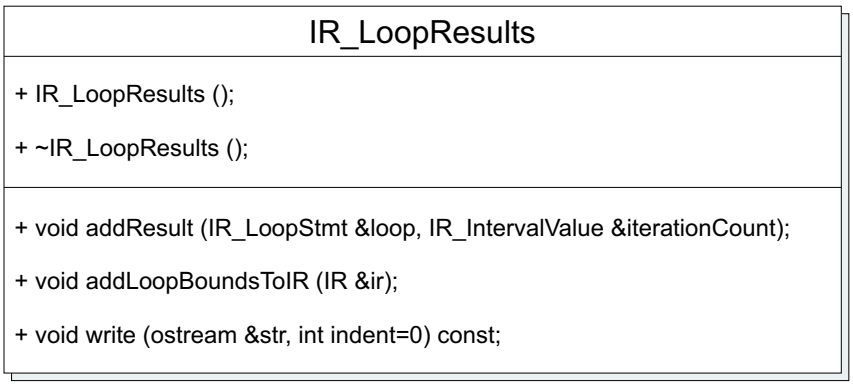


Abbildung 7.10: *Klasse IR_LoopResults*

Nach einer erfolgreich durchgeführten Schleifenanalyse kann mit der Methode *addResult(IR_LoopStmt &loop, IR_IntervalValue &iterationCount)* die berechnete Schleifeniterationsgrenze (*iterationCount*) der Loop-Anweisung (*loop*) zugewiesen werden.

Nach Abschluss der gesamten Analyse können die Schleifengrenzen an den Flow Fact Manager übergeben werden. Dies wird über einen Aufruf der Methode *addLoopBoundsToIR(IR &ir)* erreicht. Zusätzlich werden die ermittelten Schleifeniterationsgrenzen auch in das *IR* Objekt der *ICD-C* aufgenommen. Diese Methode wird von der Methode *addLoopBoundsToIR* der Klasse *IR_AIAnalyzer* aufgerufen.

Um dem Benutzer die Analyseergebnisse auf der Konsole ausgeben zu können, wurde zusätzlich die Methode *write(ostream &str, int indent=0)* für die Klasse *IR_LoopResults* überschrieben.

7.5 Integration der Schleifenanalyse

Wie bereits in der Einleitung dieses Kapitels erwähnt wurde, wurden die beschriebenen Klassen in den Kompiliervorgang des WCC's integriert. Standardmäßig ist die Schleifenanalyse jedoch deaktiviert, da für die Abstrakte Interpretation nicht garantiert werden kann, dass diese terminiert. Die Einbettung in den Kompiliervorgang des WCC's ist in Abschnitt 7.5.1 genauer beschrieben.

Da die automatische Schleifenanalyse, abhängig vom zu analysierenden Programm, ggf. eine recht lange Laufzeit verursachen kann, wurde zusätzlich ein Stand-Alone Tool entwickelt, das eine eigenständige Schleifenanalyse durchführen kann. Mit Hilfe dieses Tools ist es möglich ein Programm einmalig zu analysieren, um die Schleifeniterationsgrenzen zu bestimmen. Das entsprechende Programm ist im Ordner *tools* der Loopanalyzer-Bibliothek zu finden und heißt *irloopanalyzer*.

Die Schleifeniterationsgrenzen, die über das Tool ermittelt wurden, können im Anschluss im Quellcode annotiert werden. So stehen diese für alle folgenden Kompiliervorgänge zur Verfügung, ohne das die Schleifenanalyse jedes Mal erneut durchgeführt werden muss. Wie das Stand-Alone Tool *irloopanalyzer* im Detail aufgebaut ist, wird in Abschnitt 7.5.2 beschrieben.

7.5.1 Einbettung in den Kompiliervorgang des WCC's

Die Schleifenanalyse wurde, wie in Abschnitt 3.2.1 beschrieben, in den WCC integriert. Nachdem die Quellcodedateien eingelesen sind und daraus die ICD-C Zwischendarstellung entstanden ist, wird zunächst der Flow Fact Manager instantiiert. Dieser verarbeitet die Flow Facts, die ein Benutzer per Hand annotiert hat.

Nachdem dieser Vorgang abgeschlossen ist, sind bereits alle Vorbedingungen erfüllt, so dass die automatische Schleifenanalyse gestartet werden kann, falls diese beim Aufruf des WCC durch den Parameter *--enable-loopanalyzer* aktiviert wurde. Dazu wird ein Objekt der Klasse *IR_AIAnalyzer* erstellt, dem eine Referenz des *IR*-Objekts der erstellten ICD-C übergeben wird. Nachdem die automatische Analyse der Schleifeniterationsgrenzen abgeschlossen ist, werden die ermittelten Schleifengrenzen über einen Aufruf der Methode *addLoopBoundsToIR* in die *IR* aufgenommen und zusätzlich an den Flow Fact Manager übergeben. Wurden bei der Analyse abweichende Schleifengrenzen zu den annotierten gefunden, so wird eine entsprechende Warnmeldung auf dem Bildschirm ausgegeben. Da davon ausgegangen werden muss, dass der Benutzer die Grenzen ggf. exakter bestimmt hat, werden diese nicht durch die ermittelten Grenzen der Schleifenanalyse ersetzt.

Im Anschluss kann der normale Kompiliervorgang des WCC's fortgesetzt werden.

7.5.1.1 Konfiguration der integrierten Schleifenanalyse

Im Gegensatz zum Stand-Alone Tool, wurden bis auf den Parameter *--enable-loopanalyzer* keine weiteren Konfigurationsmöglichkeiten der Schleifenanalyse in den WCC integriert, da die automatische Schleifenanalyse nur einen kleinen Teil des WCC's ausmacht. Würden alle verfügbaren Einstellungsmöglichkeiten der Schleifenanalyse in die Konfiguration des WCC's übernommen, so würde dieser nur unnötig verkompliziert. Daher wurde eine Standardeinstellung vorgenommen, die laut Evaluation (siehe Kapitel 8) für die meisten Programme gute Ergebnisse liefert.

Als Schleifenanalyseverfahren wurde dabei die zweite Variante der modifizierten Abstrakten Interpretation gewählt, da dieses die besten Ergebnisse geliefert hat. Das statische Schleifenanalyseverfahren wurde vorerst deaktiviert, da bei einem WCET-optimierenden Kompiliervorgang mögliche Überapproximationen vermieden werden sollten.

7.5.2 Stand-Alone Analysetool – irloopanalyzer

Die Funktionsweise des *irloopanalyzer*-Tools ist an die integrierte Variante des WCC's angelehnt. Hier muss die *IR* der ICD-C allerdings selbstständig vom Analysetool erstellt werden, da dies nicht bereits durch den Compiler geschehen ist. Auf Basis dieser IR wird im Anschluss ebenfalls der Flow Fact Manager instantiiert, bevor die Schleifenanalyse gestartet wird.

Nachdem die Analyse beendet ist, wird eine kurze Zusammenfassung auf dem Bildschirm ausgegeben. Dazu gehört u.a. der ermittelte Zustand, der am End-Knoten des Programmflussgraphen gültig ist. Außerdem werden an dieser Stelle ggf. Warnungen ausgegeben, falls es Differenzen zwischen den ermittelten und den vorher annotierten Schleifengrenzen geben sollte. Abgeschlossen wird die Ausgabe jeweils durch eine Zusammenfassung der gefundenen Flow Facts.

7.5.2.1 Konfiguration des irloopanalyzer-Tools

Die Parameter des *irloopanalyzer*-Tools sind leider auf unäre Argumentbezeichnungen begrenzt. Die Parameter des WCC's werden durch die C-Bibliothek *AnyOption* erstellt. Diese konnte jedoch für das Stand-Alone Tool nicht genutzt werden, da die Schleifenanalyse sonst nicht unabhängig vom WCC einsatzfähig wäre. Über welche Parameter das Verhalten des *irloopanalyzer*-Tools beeinflusst werden kann, kann über den Aufruf des Programms ohne Parameter herausgefunden werden. Diese sind im Folgenden kurz zusammengefasst:

- *-m [1..2]*

  Wird dieser Parameter angegeben, so kann über diesen das gewünschte Schleifenanalyseverfahren der modifizierten Abstrakten Interpretation gewählt werden. Genau wie bei der integrierten Variante der Schleifenanalyse ist standardmäßig die zweite Schleifenanalysevariante vorausgewählt.

- *-s*

  Aktiviert die statische Schleifenanalyse, welche die Grenzen mit Hilfe der Polytop-Berechnungen ermitteln soll.

- *-t*

  Ist dieser Parameter gesetzt, so wird die topologische Sortierung der Zuweisungsabhängigkeiten innerhalb der statischen Analyse nicht ausgeführt. Standardmäßig wird diese durchgeführt, um mehr Schleifen analysieren zu können. Dieser Parameter hat keine Auswirkung auf die Analyse, wenn der Parameter *-s* nicht gesetzt ist.

- *-S*

  Dieser Parameter aktiviert das Program Slicing, das vor der Analyse durchgeführt wird. Hierdurch wird die Geschwindigkeit der Analyse beschleunigt, da alle Anweisungen, welche für die Schleifenanalyse nicht relevant sind, entfernt werden. Das Slicing sollte auf jeden Fall aktiviert werden, wenn die statische Schleifenanalyse eingesetzt wird, um möglichst viele Schleifen analysieren zu können.

- *-v [0..2]*

  Mit diesem Parameter kann gesteuert werden, wie viele Ausgaben während der Analyse erstellt werden sollen. Je höher der Wert gesetzt wird, desto mehr Ausgaben werden erzeugt.

Wird der Parameter nicht gesetzt, so wird die Menge der Ausgaben auf 1 gesetzt, was einer mittelmäßig detaillierten Ausgabe entspricht.

Die Schleifenanalyse muss in dem Fall des Stand-Alone Tools nicht separat aktiviert werden, so dass ein alternativer Parameter zu *--enable-loopanalyzer* nicht benötigt wird.

7.5.2.2 Beispielausgabe

```
1  Analysing function - call for function : complex (a, b)

   Analysis terminated with the following state :
   { { int main ( void ), [1, 1] }, { int a, [1, 1] },
5    { int b, [1, 1] }, { int answer , [1, 1] } }

   ********** List of Flow Facts **********
   Loopbound min: 0 max: 9 for : 0 x825d030 ; l. 81: while ( b < a ) {
   Loopbound min: 9 max: 9 for : 0 x825cdb8 ; l. 72: while ( a < 30 ) {
```

Abbildung 7.11: *Beispielausgabe des irloopanalyzer-Tools*

Zum Abschluss dieses Kapitels soll beispielhaft die Ausgabe abgebildet werden, die durch die Analyse des *irloopanalyzer* Tools bei dem Benchmark *janne_complex* auf Ausgabe-Stufe 1 erzeugt wird. Auf dieser Ausgabestufe werden, wie in Abbildung 7.11 zu sehen, alle Funktionsaufrufe ausgegeben. So kann der Fortschritt während der Analyse beobachtet werden. Nachdem die Analyse beendet wurde, wird, wie in Zeile 3 zu sehen, der finale Zustand ausgegeben, der nach Beendigung des Programms gültig ist. Abgeschlossen wird die Ausgabe durch die Auflistung der Flow Facts, die durch das Programm berechnet wurden.

Kapitel 8

Evaluation

Die in diesem Kapitel vorgestellten Ergebnisse basieren auf den Analyseergebnissen, die durch das Stand-Alone Tool *irloopanalyzer* ermittelt wurden, welches bereits im letzten Kapitel vorgestellt wurde (siehe Abschnitt 7.5.2). Um aussagekräftige Ergebnisse zu erzielen, wurde die entwickelte Schleifenanalyse mit allen Testprogrammen der Benchmarksammlung $WCETBENCH$ des WCC's ausgeführt. In dieser Sammlung befinden sich hauptsächlich Testprogramme aus der Testsuite der Mälardalen Universität Schweden [Sch08] und der *Mediabench* [LPMS97]. Diese so zusammengestellte Benchmarksammlung enthält Testprogramme aus den unterschiedlichsten Anwendungsbereichen, so dass eine repräsentative Aussage über die Güte der entwickelten Analyseverfahren getroffen werden kann.

Das ursprüngliche Ziel dieser Arbeit war es, eine automatische Schleifenanalyse für ANSI-C Programme zu entwickeln, die ausschließlich aus Integralen-, sowie zusammengesetzten Datentypen bestehen. Die Ergebnisse dieser Benchmarks sind in Abschnitt 8.1 aufgeführt. Da das ursprünglich definierte Ziel dieser Arbeit bereits relativ früh erreicht wurde, wurde dieses im weiteren Verlauf um optionale Ziele erweitert. So wurde im Folgenden eine Unterstützung für Gleitkommazahlen integriert. Die Ergebnisse, die sich nach Analyse der zugehörigen Benchmarks ergeben haben, sind in Abschnitt 8.2 beschrieben. Nachdem schließlich auch für die Benchmarks mit Gleitkommazahlen ausreichend gute Ergebnisse erzielt wurden, wurde die Analyse um eine Pointer-Unterstützung erweitert. Dies wurde durch die Integration der Alias-Analyse (siehe Abschnitt 3.1.1.2), welche Bestandteil der ICD-C ist, erreicht. Die Ergebnisse der Benchmarks, die sich sowohl aus Integralen-, Gleitkomma- und auch Pointer-Datentypen zusammensetzen, sind in Abschnitt 8.3 zu sehen. Abgeschlossen wird das Kapitel mit einer Vorstellung der Gesamtergebnisse, in der die Ergebnisse noch einmal zusammengefasst werden. Die vorgestellte Aufteilung der Abschnitte wurde insbesondere wie vorgestellt gewählt, um den Entwicklungsverlauf dieser Diplomarbeit wiederzugeben.

Die Tabellen in den folgenden Abschnitten haben alle einen identischen Aufbau. In der ersten Spalte befindet sich die Bezeichnung des zu analysierenden Benchmarks. Danach folgen jeweils 3 Spalten, welche die Analyseergebnisse der unterschiedlichen Verfahren präsentieren. Die ersten zwei Spalten stellen dabei die Ergebnisse der beiden Analyseverfahren der modifizierten Abstrakten In-

terpretation (siehe Abschnitt 4.3) dar und die dritte Spalte die der statischen Schleifenanalyse (siehe Kapitel 6). Pro Verfahren wurde dabei die Gesamtlaufzeit der Analyse und die Anzahl der exakt analysierten Schleifen angegeben. Exakt analysiert heißt in diesem Fall, dass keine Überapproximation der oberen oder unteren Schleifengrenze entstanden ist. Im Fall des statischen Schleifenanalyseverfahrens, entspricht dies jedoch nur der Anzahl an Schleifengrenzen, welche bereits ohne Überapproximation durch das statische Verfahren berechnet werden konnte. Falls eine Schleife nicht durch die statische Methode ausgewertet werden konnte (Vorbedingungen für die Auswertungen siehe Abschnitt 6.1), so wird diese mit der zweiten Variante der Abstrakten Interpretation ausgewertet. Die Anzahl an Schleifen, die durch beide Verfahren zusammen ohne Überapproximationen ausgewertet werden konnten, sind in Klammern dahinter angegeben. Die angegebenen Laufzeiten ergaben sich bei Ausführung der Analyse auf einem PC mit einem AMD Sempron(tm) 3000+ Prozessor (128 KB Cache) mit 2GB RAM.

Die beiden Analyseverfahren der modifizieren Abstrakten Interpretation wurden auf den unveränderten Versionen der Benchmarks durchgeführt. Für die statische Schleifenanalyse wurde im Vorfeld ein Program Slicing (siehe Abschnitt 3.1.1.2) durchgeführt, damit die Schleifenrümpfe des zu analysierenden Programms soweit vereinfacht werden, dass diese durch die statische Schleifenanalyse ausgewertet werden können. Im Verlauf der Evaluation wurde dabei festgestellt, dass das Program Slicing, welches von der ICD-C zur Verfügung gestellt wird, nicht genügend Anweisungen entfernt, welche die Analyse der Schleifengrenzen nicht beeinflussen. Die aktuelle Version des Program Slicings arbeitet auf einem *Program Dependence Graph* und ist intraprozedural, so dass es alle Funktionen einzeln betrachtet und dabei Aufrufkontexte ignoriert. Daher kann u.a. für den Rückgabewert einer Funktion nicht entschieden werden, ob dieser im späteren Programmverlauf benötigt wird. Die Evaluation der vorliegenden Benchmarks hat gezeigt, dass die Berechnung des Rückgabeparameters stark von den vorherigen Anweisungen abhängt, was dazu führt, dass das intraprozedurale Slicing nur einen kleinen Teil der nicht relevanten Anweisungen entfernt. Eine Abhilfe schafft eine erweiterte Form des Program Slicings, welche auf einem sogenannten *System Dependence Graph* basiert, der interprozedural arbeitet (siehe [HRB90]). Diese Analyse betrachtet die Übergabe von Parametern als auch von Rückgabewerten. Zusätzlich werden Aufrufkontexte unterschieden, wodurch eine viel aggressivere Vereinfachung des Programms ermöglicht wird. Eine Implementierung des interprozeduralen Program Slicing erfordert jedoch sehr komplexe Datenstrukturen und Mechanismen. Zusätzlich muss dafür die bestehende Datenflußanalyse der ICD-C um eine Alias-Analyse erweitert werden, die kontextsensitiv ist. Die Realisierung dieser Code-Optimierung würde jedoch weit über die Ziele der Diplomarbeit gehen, so dass für eine Evaluation der statischen Schleifenanalyse das interprozedurale Program Slicing manuell durchgeführt wurde, um die Machbarkeit (*proof of concepts*) der statischen Schleifenanalyse aufzuzeigen. Da das manuelle Program Slicing für einige Benchmarks zu komplex war, wurde ein *-Symbol in die Zelle der Tabelle eingesetzt um zu zeigen, dass aktuell über dieses Programm keine Aussage bezüglich der statischen Schleifenanalyse getroffen werden konnte. Analog dazu bedeutet das „-“-Symbol, dass das Programm nicht mit Hilfe der Abstrakten Interpretation oder der statischen Schleifenanalyse ausgewertet werden konnte.

8.1 Integrale Benchmarks

In Tabelle 8.1 sind die Ergebnisse zu sehen, welche sich bei der Analyse der Benchmarks ergeben haben, die ausschließlich aus Integralen-, sowie zusammengesetzten Datentypen bestehen.

Testbench	AI Analyse 1		AI Analyse 2		statische Analyse	
	Zeit	Schleifen	Zeit	Schleifen	Zeit	Schleifen
binarysearch	$<$ 1 Sek	1/1	$<$ 1 Sek	1/1	-	0/1 (1/1)
bsort	42 Sek	2/3	23 Sek	3/3	-	1/3 (3/3)
countnegative	22 Sek	4/4	22 Sek	4/4	$<$ 1 Sek	4/4 (4/4)
cover	$<$ 1 Sek	3/3	$<$ 1 Sek	3/3	$<$ 1 Sek	3/3 (3/3)
expint	$<$ 1 Sek	3/3	$<$ 1 Sek	3/3	$<$ 1 Sek	3/3 (3/3)
fibcall	$<$ 1 Sek	1/1	$<$ 1 Sek	1/1	$<$ 1 Sek	1/1 (1/1)
insertsort	$<$ 1 Sek	2/2	$<$ 1 Sek	2/2	-	0/2 (2/2)
janne_complex	3 Sek	0/2	$<$ 1 Sek	2/2	-	0/2 (2/2)
lcdnum	$<$ 1 Sek	1/1	$<$ 1 Sek	1/1	$<$ 1 Sek	1/1 (1/1)
ludcmp	8 Sek	2/11	8 Sek	11/11	9 Sek	2/11 (11/11)
matmult	8:16 Min	5/5	8:16 Min	5/5	$<$ 1 Sek	5/5 (5/5)
petrinet	$<$ 1 Sek	1/1	$<$ 1 Sek	1/1	$<$ 1 Sek	1/1 (1/1)
recursion	$<$ 1 Sek	0/0	$<$ 1 Sek	0/0	$<$ 1 Sek	0/0 (0/0)
searchmultiarray	17 Sek	4/4	17 Sek	4/4	-	0/4 (4/4)
selection_sort	3:12 Min	2/2	3:12 Min	2/2	$<$ 1 Sek	2/2 (2/2)

Tabelle 8.1: *Ergebnisse der Analyse von Integralen-Benchmarks der WCETBENCH*

Aus dieser Tabelle kann abgelesen werden, dass die automatische Schleifenanalyse mit beiden Varianten der Schleifenanalyse der modifizierten Abstrakten Interpretation für alle Benchmarks ein Ergebnis liefert. Ein Großteil (9 von 15) der Benchmarks kann in weniger als einer Sekunde ausgewertet werden.

Die zweite Variante der Schleifenanalyse der modifizierten Abstrakten Interpretation wertet alle Schleifengrenzen ohne Überapproximation aus. Für das erste Verfahren gibt es ebenfalls nur drei Benchmarks, für die eine Überapproximation entsteht. Dazu gehören die Benchmarks *bsort*, *janne_complex* und *ludcmp*. Die Überapproximationen dieser Benchmarks ergeben sich dadurch, dass dieses Verfahren bei der Auswertung der Schleifeniterationen stets alle möglichen Werte zusammenfügt, die in der Schleife gültig sein könnten. Dieser Nachteil des Verfahrens wurde bereits während der Entwicklungsphase festgestellt, so dass die zweite Variante der Schleifenanalyse entwickelt wurde.

Auffällig ist, dass die beiden Benchmarks *matmult* und *selection_sort* als einzige Testprogramme Laufzeiten im Minuten-Bereich haben. Dies ergibt sich dadurch, dass dort auf sehr großen, teilweise verschachtelten Arrays, komplexe mathematische Operationen ausgeführt werden. In diesem Fall kann die statische Schleifenanalyse Abhilfe schaffen. Für beide Benchmarks wird die Analysezeit durch Hinzunahme der statischen Analyse auf weniger als eine Sekunde reduziert. So ergibt sich, bezogen auf den Benchmark *matmult*, eine Beschleunigung um einen Faktor, der größer als 500 ist. Solche komplexen Operationen auf Arrays werden häufig in den Benchmarks durchgeführt, ohne

das die Arrays selbst die Schleifengrenzen beeinflussen. Daher lässt sich anhand von *matmult* die Vermutung aufstellen, dass sich durch die statische Schleifenanalyse bei komplexeren Benchmarks die Analysezeit drastisch reduzieren lässt.

Insgesamt kann für das statische Schleifenanalyseverfahren festgehalten werden, das 10 der 15 Benchmarks durch Hinzunahme des statischen Verfahrens beschleunigt werden konnten. Dies entspricht 2/3 aller Benchmarks. Zudem sei angemerkt, dass das statische Analyseverfahren gegen die zuvor geäußerte Befürchtung aus Kapitel 6, alle Schleifengrenzen exakt bestimmt hat, wenn eine Bestimmung möglich war.

Insgesamt werden durch Kombination der statischen Schleifenanalyse mit der zweiten Variante der modifizierten Abstrakten Interpretation 12 der 15 Benchmarks unter 1 Sekunde ohne Überapproximation ausgewertet.

8.2 Benchmarks mit Gleitkommazahlen

In Tabelle 8.2 sind die Ergebnisse zu sehen, welche sich bei der Analyse der Benchmarks ergeben die Gleitkomma-Datentypen, jedoch keine Pointer, beinhalten.

Testbench	**AI Analyse 1**		**AI Analyse 2**		**statische Analyse**	
	Zeit	Schleifen	Zeit	Schleifen	Zeit	Schleifen
fft1	$<$ 1 Sek	11/11	$<$ 1 Sek	11/11	*	*
minver	-	-	$<$ 1 Sek	17/17	*	*
qsort-exam	-	-	$<$ 1 Sek	6/6	*	*
qurt	$<$ 1 Sek	1/1	$<$ 1 Sek	1/1	$<$ 1 Sek	1/1 (1/1)
select	-	-	$<$ 1 Sek	4/4	*	*
sqrt	$<$ 1 Sek	2/2	$<$ 1 Sek	2/2	$<$ 1 Sek	1/2 (2/2)
statemate	6 Sek	1/1	6 Sek	1/1	$<$ 1 Sek	1/1 (1/1)

Tabelle 8.2: *Ergebnisse der Analyse von Gleitkomma-Benchmarks der WCETBENCH ohne Pointer*

Insgesamt existieren aktuell 7 Benchmarks in der *WCETBENCH*, die zusätzlich Gleitkomma-Datentypen, aber keine Pointer, enthalten. Die beiden Benchmarks *qsort-exam* und *select* mussten allerdings leicht modifiziert werden, da diese innerhalb einer Schleifenabbruchbedingung auf nicht initialisierten Speicherbereich zugegriffen haben. Dieses undefinierte Verhalten (*undefinied behavior*) führte dazu, dass die Schleifenanalyse nicht terminierte. Daher wurden die betroffenen Arrays um eine Position erweitert, die mit 0 gefüllt wurde.

Die erste Variante der Schleifenanalyse, auf Basis der modifizierten Abstrakten Interpretation, kann in diesem Fall nur noch die Hälfte der Benchmarks analysieren. Bei den anderen Benchmarks terminiert dieses Verfahren nicht. Der Grund dafür liegt in der Vereinigung des Start-Zustands mit dem End-Zustand einer jeden Schleifenauswertung. Iteriert eine Variable beispielsweise innerhalb einer Schleife über die Werte 0 bis 10, so wird für den Zustand nach der Schleife bestimmt, dass die

Variable die Werte $[0, 10]$ annehmen kann. Wenn diese Variable wiederum die Abbruchbedingung einer äußeren Schleife bedingt, so kann es sein, dass beispielsweise eine $\leq$ Bedingung niemals für alle möglichen Werte erfüllt wird.

Dieser Umstand wurde bereits früh erkannt, so dass die zweite Variante der Schleifenanalyse der modifizierten Abstrakten Interpretation entwickelt wurde. Mit diesem Verfahren können hingegen auch bei den Benchmarks, die aus Gleitkomma-Datentypen bestehen, 100% der gegebeben Testprogramme ohne Überapproximation der Schleifengrenzen analysiert werden (bezogen auf die korrigierten Versionen von *qsort-exam* und *select*). Auch hier dauert die Analyse der Testprogramme, mit einer Ausnahme, weniger als 1 Sekunde.

Das statische Schleifenanalyseverfahren konnte in diesem Abschnitt für 3 der 7 Benchmarks getestet werden. Durch die Hinzunahme dieses Verfahrens terminieren alle Benchmarks in unter einer Sekunde, was mit der modifizierten Abstraktion Interpretation für den Benchmark *statemate* vorher nicht galt. Die anderen 4 Testprogramme waren zu komplex, um ein manuelles Slicing durchzuführen. Bezogen auf den Benchmark *sqrt* kann in der Tabelle gesehen werden, dass nur eine der zwei Schleifengrenzen über die statische Analyse ermittelt werden konnte. Daher wurde die zweite Schleife mit der zweiten Variante der modifizierten Abstrakten Interpretation analysiert. An diesem Testprogramm kann daher gut das Zusammenspiel der beiden Verfahren beobachtet werden.

8.3 Benchmarks mit Pointern

In Tabelle 8.3 sind die Ergebnisse zu sehen, welche sich bei der Analyse der Benchmarks ergeben haben, die neben den bisher unterstützten Datentypen zusätzlich Pointer beinhalten.

Auch bei den Benchmarks, in denen Pointer die Schleifengrenzen direkt oder indirekt beeinflussen, ergeben sich Unterschiede zwischen den beiden Verfahren der modifizierten Abstrakten Interpretation. So terminieren die drei Benchmarks *adpcm_decoder*, *adpcm_encoder* und *lms* bei Anwendung des ersten Verfahrens der modifizierten Abstrakten Interpretation nicht, obwohl das Testprogramm mit der zweiten Variante des Analyseverfahrens ausgewertet werden kann.

Unter den Benchmarks, in denen Pointer enthalten sind, befinden sich, im Gegensatz zu den bisher vorgestellten Testprogrammen, Programme, die nicht analysiert werden konnten. Bei den meisten Benchmarks, die nicht lauffähig sind, besteht das Problem, dass nicht genügend Informationen von der Alias-Analyse bereitgestellt werden. Die Alias-Analyse versucht vor dem Start der Schleifenanalyse eine statische Auswertung vorzunehmen. Dabei wird für jedes Symbol gespeichert, auf welche anderen Symbole des Programms es zeigen könnte. Leider wird in der Analyse nicht zwischen einzelnen Kontexten unterschieden, so dass beispielsweise eine funktionsinterne Auswertung nicht möglich ist. Das bedeutet, dass die Analyse bei einer Dereferenzierung grundsätzlich alle Symbole, auf die ein Pointer im Laufe des Programms zeigen könnte, liefert. Durch diesen Informationsverlust ist es nicht möglich Schleifen zu analysieren, die direkt oder indirekt durch einen Pointer bedingt oder modifiziert werden.

Testbench	AI Analyse 1		AI Analyse 2		statische Analyse	
	Zeit	Schleifen	Zeit	Schleifen	Zeit	Schleifen
adpcm_decoder	-	-	1:23 Min	14/14	*	*
adpcm_encoder	-	-	1:37 Min	15/15	*	*
compressdata	$<$ 1 Sek	4/4	$<$ 1 Sek	4/4	$<$ 1 Sek	3/4 (4/4)
crc	34 Sek	3/3	34 Sek	3/3	$<$ 1 Sek	2/3 (3/3)
edn	53 Sek	12/12	53 Sek	12/12	$<$ 1 Sek	12/12 (12/12)
epic	-	-	-	-	-	-
fdct	$<$ 1 Sek	2/2	$<$ 1 Sek	2/2	$<$ 1 Sek	2/2 (2/2)
fir	11 Sek	2/2	11 Sek	2/2	*	*
g721_encode	-	-	-	-	-	-
g723_encode	-	-	-	-	-	-
gsm_encode	-	-	-	-	-	-
gsm_decode	-	-	-	-	-	-
hamming_window	14 Sek	3/3	14 Sek	3/3	$<$ 1 Sek	2/3 (3/3)
jfdctint	$<$ 1 Sek	3/3	$<$ 1 Sek	3/3	$<$ 1 Sek	3/3 (3/3)
lms	-	-	1:29 Min	10/10	*	*
md5	-	-	-	-	-	-
mpeg2	-	-	-	-	-	-
ndes	43 Sek	12/12	43 Sek	12/12	4 Sek	4/12 (12/12)
st	7:08 Min	5/5	7:08 Min	5/5	*	*

Tabelle 8.3: *Ergebnisse der Analyse von Pointer-Benchmarks der WCETBENCH*

Ein weiteres Problem, welches bei der Benutzung der Alias-Analyse aufgetreten ist, ist dadurch gegeben, dass bei einer Dereferenzierung von zusammengesetzten Typen nicht unterschieden werden kann, ob das Ergebnis sich auf eine interne Position oder das gesamte Objekt bezieht. Existiert beispielsweise ein Pointer, der auf ein internes Element einer Struct zeigt, so wird lediglich das betroffene Symbol des Struct-Objekts zurückgegeben. Diese Angabe ist in diesem Fall für die Analyse nicht ausreichend, so dass die Schleifenanalyse an dieser Stelle häufig abgebrochen werden muss.

Von diesen Problemen sind die Benchmarks *g721_encode*, *g723_encode*, *gsm_encode* und *md5* betroffen, so dass für diese keine Auswertung möglich war. Des Weiteren terminiert die Alias-Analyse für die beiden Benchmarks *gsm_decode* und *mpeg2* nicht, so dass auch dort die Schleifenanalyse mit der aktuellen Version der Alias-Analyse nicht durchgeführt werden kann. Letztlich sind die hohen Iterationszahlen der Schleifen des Benchmarks *epic* zu groß, um diese in einer akzeptablen Laufzeit mit Hilfe der Abstrakten Interpretation zu bestimmen. Es kann jedoch für diesen Benchmark davon ausgegangen werden, dass dieser in Zukunft durch eine Kombination des zu entwickelnden interprozeduralen Program Slicings mit der statischen Schleifenanalyse ein Ergebnis in einer akzeptablen Laufzeit liefern wird.

Trotzdem wurde durch die Integration der Alias-Analyse der ICD-C gezeigt, dass die Schleifenanalyse mit Hilfe einer hinreichend genauen Pointer-Analyse in der Lage ist, auch Benchmarks zu analysieren, in denen Pointer enthalten sind. Die zweite Variante der modifizierten Abstrakten Interpretation terminiert immerhin in 12 von 19 Fällen, so dass über 63% der Benchmarks ohne Überapproximation der Schleifengrenzen analysiert werden konnten. Durch eine Erweiterung der Alias-Analyse, die wir in Zukunft durchführen wollen, versprechen wir uns eine erfolgreiche Schlei-

fenanalyse aller Benchmarks, die im Moment auf Grund der genannten Probleme scheitert.

Bezogen auf die statische Schleifenanalyse ist es gelungen, für insgesamt 7 der 12 lauffähigen Benchmarks ein manuelles Program Slicing durchzuführen. Für die restlichen Programme war auch an dieser Stelle ein manuelles Program Slicing zu komplex. Auch bei den Testprogrammen, in denen Pointer integriert waren, wurde die Laufzeit der Gesamtanalyse durch Verwendung des statischen Verfahrens deutlich reduziert, falls dieses anwendbar war. So wurde beispielsweise die Analysezeit des Benchmarks *edn* von 53 Sekunden auf unter eine Sekunde reduziert. Ähnliches gilt für die Benchmarks *ndes*, *crc* und *hamming_window*, deren Laufzeiten von 43, 34 und 14 Sekunden ebenfalls auf unter eine Sekunde reduziert wurden. Für die weiteren Benchmarks, bei denen die statische Schleifenanalyse einsatzfähig war, ist die Laufzeit ebenfalls reduziert worden, doch war diese bereits bei der Anwendung der modifizierten Abstrakten Interpretation vergleichsweise gering.

Bei den Benchmarks *compressdata*, *crc*, *hamming_window* und *ndes* wurden dabei nicht alle Schleifengrenzen durch das statische Verfahren analysiert. Auch dort wurden die Schleifen, die nicht über die statische Variante auswertbar waren, mit Hilfe der zweiten Variante der modifizierten Abstrakten Interpretation analysiert. Dies zeigt, dass das statische Analyseverfahren ebenfalls die Laufzeit reduzieren kann, wenn die Programme komplexer werden und einige Schleifen nicht mit diesem Verfahren analysiert werden können. Außerdem wurde für keinen der Benchmarks eine Überapproximation durch die Verwendung beider Verfahren hervorgerufen (siehe Angaben in Klammern der Tabelle). Daher lässt sich vermuten, dass auch die anderen Programme, für die das manuelle Program Slicing zur Zeit zu komplex war, ebenfalls eine deutliche Beschleunigung der Analysezeit erfahren werden, wenn das aktuelle intraprozedurale Program Slicing der ICD-C durch das geplante interprozedurale Program Slicing ersetzt wird, so dass das statische Analyseverfahren auch dort eingesetzt werden kann.

8.4 Gesamtergebnis

In diesem Kapitel wurden die Ergebnisse von über 40 analysierten Testprogrammen aus den unterschiedlichsten Anwendungsbereichen angegeben. In den Abschnitten 8.1 und 8.2 wurde dabei gezeigt, dass die entwickelte Schleifenanalyse, bezogen auf das zweite Schleifenanalyseverfahren der Abstrakten Interpretation, in der Lage ist 100% der vorgestellten Testprogramme zu analysieren, in denen Integrale-, Gleitkomma- und zusammengesetzte Datentypen enthalten sind. Lediglich die Analyse von Benchmarks, in denen Pointer enthalten sind, hat auf Grund der zu ungenauen Alias-Analyse gezeigt, dass es Fälle geben kann, in denen die modifizierte Abstrakte Interpretation nicht terminiert, bzw. dass das Bereitstellen von unpräzisen Alias-Analyseergebnissen eine exakte Schleifenanalyse unmöglich macht. Dennoch konnten insgesamt 34 der 41 Testprogramme korrekt analysiert werden (dies entspricht 83% aller Programme). Damit wurden mehr als doppelt so viele Benchmarks analysiert, als für das ursprüngliche Ziel (Analyse der 15 integralen Benchmarks) dieser Arbeit vorgesehen waren.

Wie sich im Laufe der Evaluation gezeigt hat, gibt es keinen Benchmark in der *WCETBENCH*, für den das erste Schleifenanalyseverfahren ein genaueres Ergebnis geliefert hat als das zweite Verfahren. Die Nachteile des ersten Schleifenanalyseverfahrens der modifizierten Abstrakten Interpretation, haben sich bereits zu Anfang der Implementierungsphase gezeigt. Um jedoch den Entwicklungsprozess der vorgestellten Arbeit wiedergeben zu können, wurde das Verfahren bis zum Schluss in der Schleifenanalyse als optionales Verfahren gelassen. Bezogen auf das zweite Schleifenanalyseverfahren der modifizierten Abstrakten Interpretation hat sich herausgestellt, dass bei den gegebenen Benchmarks immer ein exaktes Ergebnis ausgegeben wurde, wenn die Analyse terminierte.

Die Auswertung der statischen Schleifenanalyse in Kombination mit einem Program Slicing war ebenfalls erfolgreich, auch wenn die Testprogramme aktuell nur über ein manuelles Program Slicing vorverarbeitet werden konnten. Das Program Slicing, welches aktuell in der ICD-C integriert ist, konnte leider nicht genügend überflüssige Anweisungen entfernen, so dass die statische Schleifenanalyse mit diesem Verfahren keine Ergebnisse liefern konnte. Dennoch konnte über das manuelle Program Slicing gezeigt werden, dass die Laufzeit der Schleifenanalyse durch Hinzunahme des statischen Verfahrens die Laufzeit signifikant reduzieren konnte. So wurde beispielsweise in Abschnitt 8.1 für den Benchmark *matmult* gezeigt, dass die Laufzeit der modifizierten Varianten der Abstrakten Interpretation von 8 Minuten auf unter 1 Sekunde verkürzt werden konnte. Auch für das statische Verfahren gilt, dass für alle Schleifen, die durch das statische Verfahren analysiert wurden, eine Iterationsgrenze ohne Überapproximation angegeben werden konnte.

Abschließend sei noch angemerkt, dass bereits während der Implementierungsphase dieser Arbeit für einige Benchmarks festgestellt wurde, dass die manuell annotierten Schleifengrenzen fehlerhaft waren. Von den insgesamt 41 Benchmarks gab es in 5 Testprogrammen Abweichungen der unteren oder oberen Schleifengrenze. Diese Abweichungen waren zwar meist nur minimal, doch kann auch bereits eine kleine Abweichung einer Schleifengrenze die Berechnung der $WCET_{est}$ maßgeblich beeinflussen. Damit dient die entwickelte Analyse nicht nur der automatischen Bestimmung von Schleifengrenzen neuer Programme, sondern kann auch zur Verifikation von manuell ermittelten (und damit fehleranfälligen) Annotationen genutzt werden.

Kapitel 9

Zusammenfassung / Ausblick

Das Ziel dieser Diplomarbeit war es, eine automatische Schleifenanalyse für ANSI-C Programme zu entwickeln. In diesem Kapitel soll schließlich zusammenfassend beschrieben werden, wie dieses Ziel erreicht worden ist.

Um das Ziel dieser Arbeit einleitend zu motivieren, wurden in den Kapiteln 1 und 2 zunächst unterschiedliche WCET-Analyseverfahren vorgestellt, mit denen es möglich ist, die Optimierung von echtzeitgestützten eingebetteten Systemen durchzuführen. In Kapitel 3 wurde im nächsten Schritt der WCC (WCET-optimierender C-Compiler) vorgestellt, mit dessen Hilfe eine automatische WCET-Optimierung während der Kompilierung eines Programms möglich ist. Diese automatische WCET-Optimierung ist jedoch nur dann möglich, wenn die Iterationsgrenzen aller Schleifen gebunden sind. Dies war der Ausgangspunkt der vorliegenden Arbeit, so dass ein automatisiertes Verfahren in den WCC integriert werden sollte, mit dessen Hilfe die Schleifengrenzen des zu analysierenden Programms automatisch bestimmt werden.

In Kapitel 4 wurde als nächstes die Abstrakte Interpretation beschrieben, welche die konzeptionelle Grundlage dieser Arbeit beschreibt. Dazu wurde zunächst die Theorie der Abstrakten Interpretation ausführlich beschrieben, um im Anschluss die modifizierte Abstrakte Interpretation vorzustellen, welche in dieser Arbeit Verwendung gefunden hat. Die ursprüngliche Version der Abstrakten Interpretation musste u.a. deshalb modifiziert werden, da eine Integration der statischen Schleifenanalyse in der ursprünglichen Form nicht möglich gewesen wäre.

Der zweite theoretische Schwerpunkt dieser Arbeit basierte auf den Polytop-Berechnungen, die in Kapitel 5 beschrieben wurden. Mit Hilfe der dort vorgestellten Polylib ist es möglich, die Verzweigungen, die bei der Analyse der Abstrakten Interpretation entstehen, genauer auszuwerten. Des Weiteren wurde mit Hilfe der Polylib ein statisches Schleifenanalyseverfahren entwickelt, dessen Aufbau in Kapitel 6 beschrieben worden ist. Mit dieser optionalen, statischen Schleifenanalyse ist es möglich, die unter Umständen zeitaufwendige Analyse der Abstrakten Interpretation signifikant zu beschleunigen.

Der Ablauf der implementierten Analyse, sowie die Schnittstellen der entwickelten C++ Bibliothek sind in Kapitel 7 wiederzufinden. Dieses Kapitel wurde durch die Beschreibung der Integration der Schleifenanalyse in einem Stand-Alone Tool, sowie dem WCC beendet.

Abschließend wurden die entwickelten Analyseverfahren in Kapitel 8 einer umfangreichen Evaluation unterzogen. Es konnte gezeigt werden, dass diese Verfahren in der Lage sind, für viele Programme, ohne Einwirkung des Benutzers, die Schleifengrenzen automatisch zu bestimmen.

Durch die umgesetzte Integration der entwickelten Schleifenanalyse in den WCC ist der Benutzer in der Lage eine vollständig automatisierte WCET-optimierende Kompilierung seines Quellcodes vorzunehmen. Zusätzlich kann er mit dem Stand-Alone Tool die Schleifengrenzen für Programme, die wiederholt optimiert werden sollen, einmalig bestimmen. Als Ausgabe der Analyse erhält der Benutzer alle gefundenen Schleifengrenzen, so dass er diese auf diesem Weg manuell in den Code annotieren kann, ohne diese jedoch per Hand zu bestimmen. So ist eine wiederholte Schleifenanalyse erst dann notwendig, wenn der Quellcode geändert wurde. Dies ist vor allem dann sinnvoll, wenn das Programm sehr komplex ist und die Schleifenanalyse dadurch viel Rechenzeit benötigt.

9.1 Mögliche Erweiterungen

Wie sich im Laufe der Evaluation gezeigt hat, liefert die automatische Schleifenanalyse bereits für einen Großteil der Testprogramme sehr gute Ergebnisse, ohne dabei Überapproximationen für die Schleifengrenzen zu berechnen.

Die Angaben, die von der aktuell integrierten Alias-Analyse für Pointer-Dereferenzierungen geliefert werden, reichen jedoch teilweise nicht aus, um Programme exakt zu analysieren, in denen Schleifengrenzen durch Pointer modifiziert werden. Daher wäre es sinnvoll diese Alias-Analyse durch ein Verfahren zu erweitern, das kontextabhängige Informationen bezüglich der Pointer-Dereferenzierungen liefern kann. Außerdem sollte die Analyse um eine Feldsensitivität erweitert werden, mit dessen Hilfe exakt ermittelt werden kann, welches Feld, bzw. welche Variable, eines Arrays oder einer Struct bei einer Anweisung referenziert wird.

Des Weiteren hat sich herausgestellt, dass die aktuelle Version des intraprozeduralen Program Slicings der ICD-C zu unpräzise Ergebnisse liefert, wodurch nicht genügend Anweisungen eliminiert werden können, um eine statische Schleifenanalyse zu ermöglichen. Aus diesem Grund wurde die Auswertung des statischen Schleifenanalyseverfahrens in Kapitel 8 auf manuell reduzierten Quellcodedateien durchgeführt. Damit das statische Analyseverfahren auch ohne den Eingriff eines Benutzers sinnvoll aktiviert werden kann, müsste das intraprozedurale Program Slicing durch ein interprozedurales Program Slicing ersetzt werden. Dieses Verfahren basiert auf einem *System Dependence Graph*, wodurch auch funktionsübergreifende Aufrufbeziehungen betrachtet werden können. Auf diese Weise ist es u.a. möglich sowohl die Argumente als auch die Rückgabewerte einer Funktion dahingehend zu untersuchen, ob diese für die gewünschte Analyse entfernt werden können. Diese Erweiterung des aktuellen Program Slicings ist allerdings sehr zeitaufwändig, da hierzu eben-

falls eine exaktere Alias-Analyse für die notwendige Erweiterung der Datenflussanalyse der ICD-C benötigt wird.

Ein weiterer Aspekt, der in der Zukunft optimiert werden könnte, wurde bereits in Abschnitt 6.1 erwähnt. Dort wurden die Vorbedingungen beschrieben, die für eine statische Schleifenanalyse gelten müssen. In dieser Aufzählung befinden sich bereits einige Punkte, in denen Einschränkungen getroffen wurden, die den aktuellen Stand der Implementierung betreffen. So könnte beispielsweise eine Unterstützung für Structs in die statische Schleifenanalyse integriert werden.

Außerdem kann auch die Analyse der Abstrakten Interpretation durch weitere Abstrakte Domänen ergänzt werden. Die Intervalldarstellung hat sich in dieser Arbeit als sehr effektiv für die Bestimmung von Schleifengrenzen bei kleineren Benchmarks herausgestellt. Viele komplexere Benchmarks wie *mpeg2* konnten im Moment auf Grund der Probleme der Alias-Analyse nicht ausgewertet werden. Daher bleibt zu untersuchen, ob die aktuelle Intervalldarstellung auch für komplexere Benchmarks präzise Ergebnisse liefert. Andernfalls kann die Präzision durch mächtigere Abstrakte Domänen, wie sie in Abschnitt 4.2.3.3 kurz erwähnt wurden, ersetzt werden. Diese Abstrakten Domänen liefern zwar bei komplexen Benchmarks präzisere Ergebnisse, sind aber zugleich wegen ihrer Mächtigkeit komplexer. Daher kann ihre Integration in die aktuelle Schleifenanalyse zu erhöhten Laufzeiten führen. Aus diesem Grund sollte in Zukunft die gewünschte Präzision gegen den ggf. entstehenden Overhead abgewogen worden und dem Benutzer die Freiheit geboten werden, das gewünschte Verfahren durch einen Parameter zu steuern.

Zusammenfassend kann jedoch gesagt werden, dass das entwickelte Verfahren auch in dem aktuellen Stand der Implementierung bereits für viele zu analysierende Programme Ergebnisse liefert, die für die $WCET_{est}$-Berechnung eingesetzt werden können.

Literaturverzeichnis

[aiT08] aiT Worst-Case Execution Time Analyzers. http://www.absint.com/ait/, 2008.

[Bih05] Holger Bihr. Entwicklung einer plattformunabhängigen, kontext-sensitiven Aliasanalyse für das ICD-C Compiler-Framework. Diplomarbeit, Technische Universität Dortmund, Fakultät für Informatik, Lehrstuhl 12, 2005.

[Byg06] Stefan Bygde. Abstract Interpretation and Abstract Domains. Master's thesis, Mälardalen University, Department of Computer Science and Electronics, June 2006.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[EG97] Andreas Ermedahl and Jan Gustafsson. Deriving Annotations for Tight Calculation of Execution Time. In *European Conference on Parallel Processing*, pages 1298–1307, 1997.

[ESG+07] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis. In *Seventh International Workshop on Worst-Case Execution Time Analysis*, Pisa, Italy, July 2007.

[Fai06] Prof. Dr. Ulrich Faigle. Einführung in die Mathematik des Operations Research - Kapitel 2: Polyeder und Polytope, 2006. http://www.zaik.uni-koeln.de/AFS/teachings/ss06/OR/OR2.pdf.

[Fer04] Christian Ferdinand. Worst-Case Execution Time Prediction by Static Program Analysis. In *18th International Parallel and Distributed Processing Symposium*, Los Alamitos, CA, USA, 2004.

[FL06] Heiko Falk and Paul Lokuciejewski. Design of a WCET-Aware C Compiler. In *6th Intl. Workshop on WCET Analysis, Dresden*, July 2006.

[FM03] Heiko Falk and Peter Marwedel. Control Flow driven Splitting of Loop Nests at the Source Code Level. In *DATE*, March 2003.

[Fou08] GNU Project Free Software Foundation. GCC, the GNU Compiler Collection. http://gcc.gnu.org/, 2008.

[GESL07] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis using Abstract Execution. In *Real-Time in Sweden (RTiS) 2007*, Västerås, Sweden, August 2007.

[GPL08] GNU General Public License - Free Software Foundation. http://www.fsf.org/licensing/licenses/gpl.html, 2008.

[Gra89] Phillipe Granger. Static Analysis of Arithmetical Congruences. In *International Journal of Computer Mathematics*, 1989.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *ACM Trans. Program. Lang. Syst.*, volume 12, pages 26–60, New York, NY, USA, 1990. ACM.

[HSRW98] Christopher Healy, Mikael Sjodin, Viresh Rustagi, and David B. Whalley. Bounding Loop Iterations for Timing Analysis. In *IEEE Real Time Technology and Applications Symposium*, pages 12–21, 1998.

[ICD07a] *ICD-C Compiler framework Developer Manual - Confidential*. Informatik Centrum Dortmund, Dortmund, Germany, July 2007.

[ICD07b] *ICD Low Level Intermediate Representation backend infrastructure (LLIR) Developer Manual - Confidential*. Informatik Centrum Dortmund, Dortmund, Germany, July 2007.

[Kir00] Raimund Kirner. Integration of Static Runtime Analysis and Program Compilation. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, 2000.

[Kir03] Raimund Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. PhD thesis, Technische Universität Wien, May 2003.

[Kir06] Martin Kirner. Automatic Loop Bound Analysis of Programs written in C. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, 2006.

[LFS+07] Paul Lokuciejewski, Heiko Falk, Martin Schwarzer, Peter Marwedel, and Henrik Theiling. Influence of procedure cloning on WCET prediction. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 137–142, New York, NY, USA, 2007. ACM.

[Lis03] Björn Lisper. Fully Automatic, Parametric Worst-Case Execution Time Analysis. In Jan Gustafsson, editor, *Proc. Third International Workshop on Worst-Case Execution Time Analysis*, pages 77–80, Porto, July 2003.

[Lok05] Paul Lokuciejewski. Design and Realization of Concepts for WCET Compiler Optimization. Master's thesis, University of Dortmund, Department of Computer Science 12, 2005.

[LPMS97] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[Mar07] Peter Marwedel. *Eingebettete Systeme*. Springer Verlag Berlin, 2007.

[MK06] Stefan Kühling Martin Keuzer. *Logik für Informatiker*. Pearson, 2006.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[Pol08] Polylib - A library of polyhedral functions. http://www.irisa.fr/polylib/, 2008.

[Sch07] Daniel Schulte. Modellierung und Transformation von Flow Facts in einem WCET-optimierenden Compiler. Diplomarbeit, Technische Universität Dortmund, Fakultät für Informatik, Lehrstuhl 12, 2007.

[Sch08] Mälardalen Universität Schweden. WCET project / Benchmarks. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html, 2008.

[Tea02] The Polylib Team. Polylib User's Manual, September 2002. http://www.irisa.fr/polylib/document.pdf.tar.gz.

[VSB+04] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Analytical computation of Ehrhart polynomials: enabling more compiler analyses and optimizations. In *Proceedings of International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 248–258, September 2004.

[Weg05] Ingo Wegener. *Theoretische Informatik, eine algorithmenorientierte Einführung 3. überarb. Aufl.* Teubner, 2005.

[Wol02] Fabian Wolf. *Behavioural Intervals in Embedded Software*. Kluwer Academic Publishers, US, 2002.

[YE95] Wei Ye and Rolf Ernst. Worst Case Timing Estimation based on Symbolic Execution. In *COBRA report, Institute of Computer Engineering, Technical University Braunschweig*, October 1995.

Stichwortverzeichnis