Mälardalen University Press Licentiate Theses
No. 115

STATIC WCET ANALYSIS BASED ON ABSTRACT INTERPRETATION AND COUNTING OF ELEMENTS

Stefan Bygde

2010

MÄLARDALEN UNIVERSITY
SWEDEN

School of Innovation, Design and Engineering

ISSN 1651-9256
ISBN 978-91-86135-55-3
Printed by Mälardalen University, Västerås, Sweden

CUGS

National Graduate School of Computer Science

Abstract

In a real-time system, it is crucial to ensure that all tasks of the system hold their deadlines. A missed deadline in a real-time system means that the system has not been able to function correctly. If the system is safety critical, this can lead to disaster. To ensure that all tasks keep their deadlines, the Worst-Case Execution Time (WCET) of these tasks has to be known. This can be done by measuring the execution times of a task, however, this is inflexible, time consuming and in general not safe (i.e., the worst-case might not be found). Unless the task is measured with all possible input combinations and configurations, which is in most cases out of the question, there is no way to guarantee that the longest measured time actually corresponds to the real worst case.

Static analysis analyses a safe model of the hardware together with the source or object code of a program to derive an estimate of the WCET. This estimate is guaranteed to be equal to or greater than the real WCET. This is done by making calculations which in all steps make sure that the time is exactly or conservatively estimated. In many cases, however, the execution time of a task or a program is highly dependent on the given input. Thus, the estimated worst case may correspond to some input or configuration which is rarely (or never) used in practice. For such systems, where execution time is highly input dependent, a more accurate timing analysis which take input into consideration is desired.

In this thesis we present a framework based on abstract interpretation and counting of possible semantic states of a program. This is a general method of WCET analysis, which is language independent and platform independent. The two main applications of this framework are a loop bound analysis and a parametric analysis. The loop bound analysis can be used to quickly find upper bounds for loops in a program while the parametric framework provides an input-dependent estimation of the WCET. The input-dependent estimation can give much more accurate estimates if the input is known at run-time.

Acknowledgements

First, I would like to express my deepest thanks to my supervisors. Thank you Björn Lisper, Andreas Ermedahl and Jan Gustafsson, without you this thesis wouldn't exist. In addition, I would like to thank Björn Lisper, Hans Hansson and Christer Norström for deciding to employ me as a Ph.D. student at Mälardalen University!

I also would like to thank the people from my research group (working on Worst-Case Execution Time) for support, ideas, useful discussions, comments and reviews: Christer Sandberg, Jan Gustafsson, Björn Lisper, Andreas Ermedahl, Andreas Gustavsson, Marcelo Santos and Linus Källberg.

Outside the thesis work I have also been involved in teaching, course development and assisting at the division of Computer Science and Network. I would like to thank the people I have had the pleasure to working quite a lot with: Christer Sandberg, Gunilla Eken, Gordana Dodig-Crnkovic, Jan Gustafsson and Rikard Lindell. In addition, many thanks goes to Anne and Zebo at CUGS, Åsa, Monica, Else-Maj, Maria, Gunnar and Harriet at IDT for making things a lot easier.

While working on this thesis I have met a lot of people and I would like to thank the people I have spent most time with during coffee breaks, conferences and parties: Aneta, Séve, Hüseyin, Luis, Cristina, Tibi, Aida (Blondie), Adnan, Marcelo, Juraj (the soup), Ana, Luka, Josip, Leo, Dag, Batu, Kathrin (Mermy), Farhang, Iva, Pasqualina, Johan L, Johan K, Johan F, Kifla, Moris, Mikael, Nolte, Lei, Ivica and Jan C. I have had a lot of good times and a lot of fun with you people. Of course, I would also like to thank my parents and brothers: Ing-marie, Jan, Bennet and Alexander.

This research was funded in part by CUGS (the National Graduate School in Computer Science, Sweden) and has been supported by the Swedish Foundation for Strategic Research (SSF) via the strategic research centre PROGRESS.

Stefan Bygde
Västerås, February, 2010

Contents

Chapter 1

Introduction

1.1 Real-Time and Embedded Systems

An embedded system can be said to be computer system designed for a specific purpose. Such a computer system differs from a desktop computer in the sense that it interacts with its environment via sensors, buses and other devices rather than a keyboard and monitor. Embedded systems are used in mobile phones, cars, power plants etc. Typically, these systems have resource constraints as they are often small, battery driven or have real-time requirements. Real-time requirements on a system means that if a computation is not finished before a given deadline, the system will either have decreased performance or is considered to have failed. Systems which can not tolerate that a deadline have been missed are called *hard* real-time systems. A missed deadline can in a safety critical hard real-time system have dire consequences, therefore, it is of great importance for these systems to ensure that all software tasks will meet their deadlines. This is ensured by estimating the worst possible execution time for each task in the system, and produce a feasible schedule for them. However, determining the worst case execution time (WCET) of a task or program is far from trivial since it depends on hardware (including complex hardware features such as pipelines, caches, branch prediction etc.) as well as software semantics (i.e., finding the worst possible paths through the program) and the interplay between the two. The solution is to find a *safe estimation* of the WCET of a task. A safe estimation of the WCET is a number which is guaranteed to be equal to or greater than the WCET. However, it is desired that this number is as close to the real WCET as possible, without compromising the safety.

1.1.1 Scheduling in Real-Time Systems

A real-time system typically has a set of software tasks which need to execute on the available processors of the system. A *real-time scheduler* is a piece of software which assigns tasks to processors during different time slots. Tasks with hard real-time constraints are required to execute and finish their execution before their given deadlines. Thus, a real-time scheduler has to make sure that all real-time tasks can meet all their deadlines. For this to be possible, the given execution times of the tasks have to be short enough for all real-time tasks to execute. If the given execution times are too pessimistic, it may be impossible for the scheduler to find a suitable schedule (that is, the system is not schedulable). For this reason, it is essential to obtain WCET estimates which are *safe* (to guarantee that the real-time constraints are met) and at the same time *tight*, i.e., as close to the real WCET as possible (to make the task set schedulable). This thesis investigates a method to find safe and tight bounds for the WCET of programs. This method is fully automatic, flexible and can achieve symbolic upper bounds for the WCET.

1.2 Worst-Case Execution Time Analysis

A lot of research has been done in the area of worst-case execution time analysis, a good overview can be found in [WEE$^+$08]. WCET analysis can roughly be divided into two disciplines, namely static and dynamic WCET analysis. A *dynamic WCET analysis* is done by performing end-to-end measurements of a running program on the target processor (or a simulator). This requires either very extensive measurements to ensure enough coverage or alternatively attempting to enforce the program to execute its worst-case path, which may be very difficult. Dynamic analysis cannot in the general case ensure that the worst-case have actually been found, and may therefore under-estimate the WCET. To come around this problem a safety margin is usually added to the worst found measurement.

The other approach is *static WCET analysis* which computes a safe upper bound of the WCET of a program by statically analysing the program code, its possible inputs and a model of the hardware. A static WCET analysis has to do pessimistic assumptions in uncertain cases to give a safe upper bound, i.e., a bound which is guaranteed to be at least as large as the real WCET. There are also hybrid approaches which in some way combines the static and dynamic analyses. Figure 1.1 displays the relationship between measured execution times, analysed execution times and the actual WCET. The figure also shows

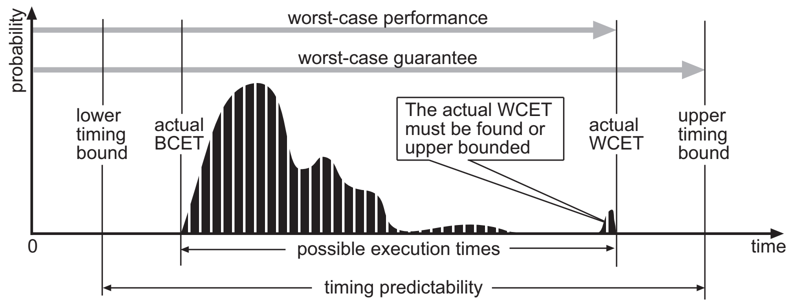


Figure 1.1: Relation between execution times and analysis results (taken from [WEE+08])

the relation to BCET which means Best-Case Execution Time. The upper and lower timing bounds are *safe* estimates of the WCET and BCET respectively.

This thesis will solely focus on static analysis. More specifically, the thesis will investigate a framework for static analysis which is based on counting run-time states to derive a WCET of a program. This framework has two major applications which will be presented in the thesis. The applications are loop bound analysis and parametric WCET analysis.

1.3 Problem Formulation

In this section two common problems associated with WCET analysis are presented. While the two problems may seem to differ quite a lot, this thesis shows that both problems can elegantly be solved by very similar methods. The first problem is to automatically find upper bounds of the length of the execution traces of a program. To be able to give an upper bound of the timing of a program, the program has to execute in a finite number of steps. Typically, programs spends most execution time is in loops, therefore it is essential to find an upper bound for the iterations for each loop. Ideally this should be automatic and quick. Thus, we attempt to answer the question

> How to efficiently and automatically find upper execution bounds for a program loop?

This thesis presents a method to quickly and automatically (that is, without user interaction) derive safe upper bounds for loops in a program.

Static analysis derives, as seen in Figure 1.1, a safe upper bound of the WCET of a program. However, the execution time of a program is affected by a number of things. Very often, the execution time of a program is heavily dependent on input variables and/or configurations/modes of the task or program. The input combination/configuration of the worst-case may be a such which is never used in practice, making the upper bound unnecessarily pessimistic. It might even be too pessimistic to use in practice [BEGL05, SEGL04, CEE+02]. Thus, the main question this thesis tries to solve is the following:

> How to decrease the inherit pessimism introduced from static analysis by assuming the worst-case input combination?

This thesis tries to overcome this pessimism by the realisation that the input of a program may be known at run-time or even at deploy time. This information can be used to derive a re-usable time estimate which is *dependent* on the input variables of a program. That is, rather than expressing the WCET as a constant number, it is expressed as a formula in terms of the values of the input variables.

1.4 Research Results

1.4.1 Loop Bound Analysis

In order to provide a concrete upper bound of the WCET of a program, a static analysis needs to be able to find execution bounds for the different parts of the program. If the number of executions of a certain loop cannot be bound, the analysis fails to give a finite, safe WCET estimation. Thus, it is crucial to have an upper bound for each loop in the program. This can be achieved with manual annotations (i.e., having the programmer annotate the code with loop bounds) or it can be automatically derived by static analysis. A method to quickly and automatically derive loop bounds was presented in [ESG+07], which is based on counting run-time states of a program.

1.4.2 Parametric WCET Analysis

In many cases the execution time of a program highly depends on its input. If the control flow of the program depends on the input data, the execution time will naturally be affected. Since the WCET of a program holds for *all* possible input combinations, it may often be too pessimistic. For example, the program may never be called with the worst-case input in practice, and the real worst case may be much lower than the estimated one. A solution to this is to compute a WCET bound which is symbolic in terms of the input values. Such a bound can then quickly be instantiated by substituting concrete input values for the symbols in the formula. Such a formula then constitutes a reusable upper bound on a program or task which is safe but also more precise since more information about the bound is known. Furthermore, by having a formula of the WCET, mathematical analysis can be applied to the formula to perform things like sensitivity analysis. The investigated framework uses known techniques to symbolically count run-time states in a program and can be used to obtain these kind of formulae.

Parametric WCET analysis is naturally more complex than classical static WCET analysis and should not be used on large systems with millions of lines of code; rather, the parametric estimation is most efficiently used on smaller program parts (like smaller tasks or functions) which have input data dependent execution times. Interesting applications would include *disable interrupt sections*, which are code sections which may not be interrupted and are therefore naturally interesting to find the WCET for. These sections typically needs to be small and are interesting candidates for a parametric WCET [CEE+02]. Another important application of parametric WCET analysis would be in component based software development (CBD) [Crn05, HC01]. In CBD, reusable components designed to interact with each other in different contexts can be analysed in isolation. Since components are designed to function in different contexts, a reusable WCET estimation is desired. Component models designed for embedded systems (such as saveCCM [HÅCT04] or Rubus [Arc09]) typically uses quite small components which makes parametric WCET analysis interesting.

1.5 Summary of Contributions

This thesis is based on a method for parametric WCET analysis presented in [Lis03a], and a method for loop bound analysis presented in [ESG+07]. The concrete contributions of this thesis is the following.

- We have formalised and enhanced the method presented in [Lis03a] and merged it with the method presented in [ESG+07] to obtain a formalised framework on how to perform WCET analysis by counting run-time states.

- A prototype implementing parts of the framework has been developed in order to evaluate the method. This prototype has provided insight and experience with the method, leading to the discovery of bottle-necks and potentials.

- We have proposed a set of simplifications of the method for parametric WCET analysis proposed in [Lis03a], such as reducing the number of variables used in the calculation.
- An enhancement of an abstract domain used in loop bound analysis has been made to make it possible to use it for low level code, which is commonly used in WCET analysis.
- An algorithm for efficient parametric WCET calculation has been proposed, implemented and evaluated.
- Some of the methods and algorithms presented in the thesis have been experimentally evaluated with the prototype mentioned above, and in the static WCET analysis tool SWEET (SWEdish Execution Time tool) [GESL07, WCE09].

1.6 Summary of Publications

This thesis is based on four papers, of which three have been published.

Paper A

Analysis of Arithmetical Congruences on Low-Level Code. Stefan Bygde. Extended abstract NWPT'07 [Byg07].

This paper describes the enhancement of the congruence domain. It provides low-level support for the domain, including low-level abstract operations and an abstraction which works for both signed and unsigned integers. The contents of this paper is covered in Chapter 5.

Paper B

Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis. Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Presented at the WCET workshop in 2007 [ESG$^+$07].

This paper shows how to estimate loop bounds by counting elements of abstract states. The evaluation of this methods also shows that the congruence domain gives more accurate results. As the forth author of the paper, I have been involved in formulating the original idea and provided the analysis with the congruence domain. The contents of this paper is mainly covered in Chapter 4, although the theoretical foundations of the method is outlined in Chapter 3. In addition to the published materials, this thesis goes deeper on some of the theoretical foundations of the approach.

Paper C

Towards an Automatic Parametric WCET Analysis. Stefan Bygde, Björn Lisper. Presented at the WCET workshop in 2008 [BL08].

This paper presents an implementation of the parametric WCET analysis based on counting elements in abstract states introduced in [Lis03a]. The paper presents necessary workarounds to make a functioning implementation as well as some simplifications that can be done to reduce complexity. As first author I have been writing the paper and been the main driver. The contents of the paper is mostly contained in Chapter 6. However, Chapter 6 contains more details than the original publication, including detailed examples.

Paper D

An Efficient Algorithm for Parametric WCET Calculation. Stefan Bygde, Andreas Ermedahl, Björn Lisper. Presented at RTCSA'09 [BEL09]. Best paper award.

This paper introduces a new parametric calculation algorithm called MPA. The paper presents the algorithm and evaluates it on a large set of benchmarks. As first author I have been writing the paper and been the main driver. The contents of the paper are included in Chapter 7, although the chapter contains a more detailed evaluation of the algorithm as well as more theoretical properties of it.

1.7 Thesis Outline

The thesis is outlined as follows:

Chapter 2 gives an overview over the field of WCET analysis and related work.

Chapter 3 provides a formalisation of the proposed framework.

Chapter 4 explains how to use the framework to compute loop bounds and evaluates it.

Chapter 5 introduces necessary developments to perform abstract interpretation on a lower level using the congruence abstract domain.

Chapter 6 explains in detail how to perform a parametric WCET analysis with the framework.

Chapter 7 introduces an efficient algorithm for parametric WCET calculation, and finally,

Chapter 8 presents a summary, conclusions and future work.

Chapter 2

Background and Related Work

This chapter will introduce terminology and concepts used in static WCET analysis and present some related work.

2.1 Analysis Phases

Static WCET analysis can essentially be divided into three independent phases. To put it simple, one phase analyses the software, one phase analyse the hardware and the final phase combines the analysis results to calculate an estimation of the WCET. This estimation is in most cases just the worst-case execution time in milliseconds. Figure 2.1 shows how the different analysis phases relate.

2.1.1 Flow Analysis

Flow analysis or high-level analysis analyses the source or object code of a program. The goal of this process is to find constraints on the program flow and find bounds on the execution counts of different parts of the program. Information about program flow are known as *flow facts*. Several analysis techniques can be applied during this phase to obtain as much information as possible. It needs to be mentioned that exact information about programs in general is

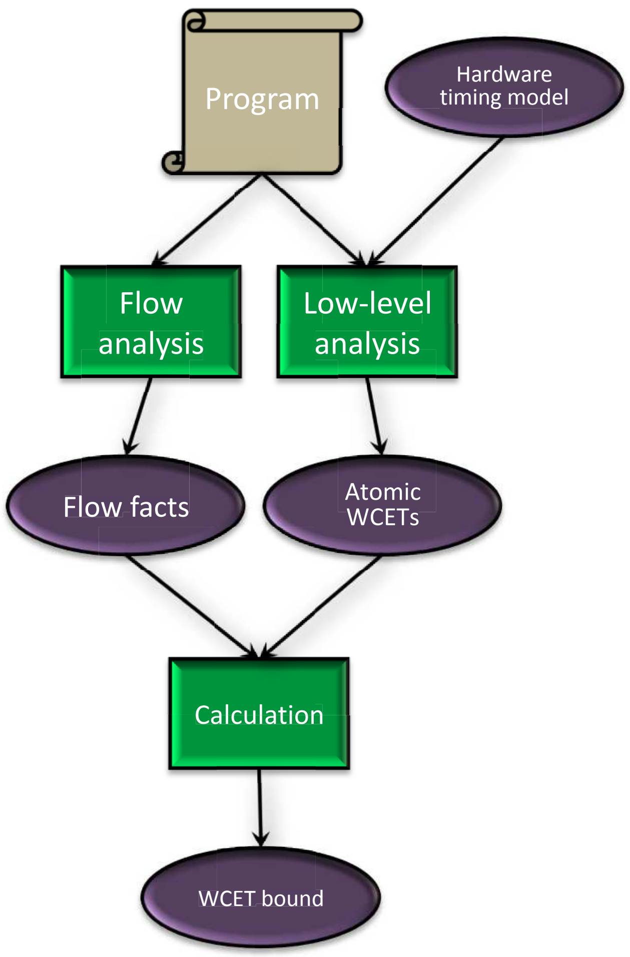


Figure 2.1: Relation between analysis phases

undecidable and many of these techniques need to introduce sound approximations rather than giving precise results.

Loop Bound Analysis

As mentioned in Chapter 1, an important part of WCET analysis, specifically the flow analysis phase, is to find an upper bound for each loop. If a loop can not be bounded the only safe assumption is that the loop will go on forever leading to an unbounded WCET of the program.

There have been some work focusing on the development of efficient and precise loop bound analysers. Healy et. al. [HSRW98], introduced a pattern based approach to find upper and lower bounds on loops. It requires user knowledge and annotation about variable bounds and it is not fully automatic and requires structured loops (although multiple exits are allowed). Another loop bound analysis is suggested in [CM07], it is based on flow analysis and binds loops by finding fixed increments of loop counter variables. It requires structured loops and can handle only loops with fixed increments. In [MBCS08] an efficient loop bound analysis is presented. This analysis requires programs to be run through a code simplifier to make sure that loops are structured and that they have single exits. Gustafsson et. al. [GESL07] presents a method to find loop bounds by a technique called abstract execution, which is simulating the execution of a program over abstract states. Bartlett et. al. [BBK09] presents a method to find exact parametric loop bounds given a certain class of nested loops, however, this requires that several traces of the programs execution is recorded and that the loop expressions are identified and can therefore not be considered as fully automatic.

In our framework we base the loop bound analysis on the method outlined in [ESG$^+$07] where general loops quickly and automatically can be analysed without imposing any restriction on the structure. This method is based on counting possible semantic states in a loop using abstract interpretation, slicing and invariant analysis techniques. Later work by Lokuciejewski et. al [LCFM09] has achieved even better results using very similar techniques but with another abstract domain, acceleration techniques (to avoid iteration in the abstract interpretation) and improved slicing. While we base our work on the earlier publication, the latter work fits quite well into the general framework suggested in this thesis. The advantage by using abstract interpretation in loop bound analysis is that abstract interpretation is completely independent of the structure of loops and works on arbitrary program flow.

Infeasible Path Detection

Essentially, the purpose of the flow analysis is to give as many and exact flow facts as possible to be able to give an accurate WCET bound. Therefore, finding paths that due to semantic constraints cannot be taken is of value to decrease the pessimism of the analysis. To give a simple example, consider the following code

```
if $n > 2$ then
    statement 1
end if
if $n < 0$ then
    statement 2
end if
```

No execution of this piece of code can execute both statement 1 and statement 2 (assuming statement 1 does not change the value of n). Some research efforts, devoted to finding infeasible paths to decrease analysis pessimism, are presented in [Alt96, APT00, HW99, GESL07, CMRS05, Lun02].

2.1.2 Low-Level Analysis

The low-level analysis analyses a mathematical model of the hardware platform. The model should be as detailed as possible, but it has to be conservative, e.g, assume a cache-miss rather than a cache-hit when it is impossible to determine statically. The purpose of the low-level analysis is to derive worst-case execution times for atomic parts of the program. Atomic parts can mean either instructions, basic blocks or some other small easily distinguishable part of a program. Note that our work is mainly concerned with flow analysis and calculation, thus, the related work presented about low-level analysis will be sparse.

Complex Hardware Features

In modern computer architectures it is common to have complex hardware features such as pipelines, caches and branch prediction. While these features greatly improves average performance, they also make the timing behaviour much harder to predict. For a low-level analysis to be precise enough, these complex features have to be taken into account and analysed. This can lead to high over-estimations of the WCET and the synergy effects among the different features may be hard to detect. A lot of work has been published in the area of low-level analysis and how to model hardware features. For instance low-level analysis and modelling has been proposed for caches [HAM$^+$99, LMW99, Rei08, FW99], pipelines [Eng02, Wil05], branch predictors [BR05, BR04b, CP00], multi-core caches [ZY09] etc.

2.1.3 Calculation

When flow facts have been derived from the flow analysis and atomic worst-case execution times have been calculated by the low-level analysis, the results can be combined to obtain a concrete bound of the WCET. This is done in the calculation phase. There have been some different approaches to WCET calculation proposed, like the tree-based (or structure based) approach [PS91, LBJ$^+$95, PPVZ92, BBMP00, CB02] which calculates the WCET by parsing the program structure bottom-up, the path-based approach [HAM$^+$99, SA00, Erm08] which explicitly models the paths of the program to find the worst-case and the perhaps most used approach called IPET introduced in the next subsection.

Implicit Path Enumeration Technique

The Implicit Path Enumeration Technique (IPET) was proposed in [LM95, LM97]. Since the number of paths through a realistically sized program tends to get very large, it is simply infeasible to try to find the worst-case path. Instead, the idea of IPET is to formulate the flow constraints and the atomic costs as an Integer Linear Programming (ILP) problem. This is done by maximising a cost function subject to the constraints obtained from flow analysis. Since the flow facts might not be exact, this calculation may over-approximate the final result.

Much of the research presented in this thesis uses and refers to the IPET method, thus, a detailed presentation of the method is presented here. The idea is to obtain an estimation of the WCET as the maximum of

$$\sum_{q \in \mathcal{Q}_P} c_q x_q$$

where $\mathcal{Q}_P$ is a set of all points in the program P, this may be edges or nodes in a control flow graph, basic blocks, labels or whatever is used to represent a program. The factor c_q represents the worst-case execution time of point q which has been calculated by a low-level analysis. The factor x_q represents an upper bound of the execution count of program point q. This factor is unknown but are subject to a set of constraints which may be obtained by the flow analysis. For example, suppose that flow-analysis has determined that the program point $q_5 \in \mathcal{Q}_P$ can never be visited more than five times. This imposes a constraint looking as follows:

$$x_{q_5} \leq 5$$

Moreover, flow-analysis may have determined that the program point q_5 is visited at least as many times as q_6 since it is dominating q_6. This can be expressed through the constraint:

$$x_{q_6} \leq x_{q_5}$$

Thus, with a objective function to maximise and a set of linear constraints, this can be solved with the simplex algorithm. The simplex algorithm gives a solution to the unknown variables $x_{q \in \mathcal{Q}_\mathcal{P}}$ such that all constraints holds and that the objective function is as large as possible.

Since there exist really efficient ILP solvers, IPET is an effective and widely used technique. IPET is also very flexible, and work has been proposed to enhance the IPET model to analyse, for instance, caches [LMW99, ZY09].

2.1.4 Auxiliary Analyses and Techniques

Some common techniques in WCET analysis are not really part of any analysis phase, but are auxiliary methods to generally facilitate WCET the different analyses.

Program Slicing

Program slicing [Wei81, Wei84] is a process of eliminating certain statements and variables from a program. A program slice is a program where each statement which directly or indirectly affects a set of given variables (slicing criterion) has been removed. In WCET analysis, slicing can be used to produce a program slice where all variables which affect program flow are removed. Flow analysis over such a slice is more efficient since a program slice is smaller than the original program but still contains the same flow facts. Program slicing is an essential part of the framework outlined in this thesis, thus a detailed explanation of the technique follows.

A *program slice* is a minimal representation of a program with respect to a *slicing criterion*. A slicing criterion is a set of variables observed at a certain statement. The slice is then obtained by removing statements and variables from the original program which are guaranteed to not affect the slicing criterion. In general it is undecidable to get a perfect slice (i.e., a slice where *all* irrelevant statements have been removed), so slicing algorithms has to apply some sort of conservative behaviour. We illustrate by an example, consider the following program:

```
$n \leftarrow 10$
$i \leftarrow 0$
$j \leftarrow 1$
**while** $n > 0$ **do**
  $i \leftarrow i + 1$
  $j \leftarrow j * 2$  {“Statement”}
**end while**
```

If this program is sliced with respect to the variable i at “Statement”, this will result in

```
$n \leftarrow 10$
$i \leftarrow 0$
**while** $n > 0$ **do**
  $i \leftarrow i + 1$
**end while**
```

As can be seen, this program has the same semantics as the original program if one only observes i at the program point where “statement” was. However, “statement” itself was irrelevant in this case and was removed by the slicing.

Slicing in the context of WCET analysis has been used in [SEGL06, ESG$^+$07, LCFM09].

Value Analysis

Value analysis is the process of determining a superset of the possible values variables can be assigned to in the program. This can be used to find infeasible paths, loop bounds and dead code among other things. The most common technique to perform value analysis is abstract interpretation [CC77]. Since, as with general flow facts, an exact value analysis would in general be undecidable, abstract interpretation soundly approximates program semantics in order to obtain a set of values which variables can be assigned to. There are many kind of approximations that can be chosen, and the choice is a trade-off between precision and complexity of the analysis.

Manual Annotations

Most of the above mentioned analyses have to introduce approximations and can in most cases not find all possible flow facts. In addition, many analyses may be costly. In some cases it may therefore be worthwhile to have a human manually annotate the source or object code with flow facts, known as *code annotations*. This might be error prone and requires good knowledge about the code, but it can on the other hand introduce flow facts which are impossible for a static analysis to derive.

2.2 Parametric Methods

The parametric WCET analysis framework presented in this thesis is based on the method outlined in [Lis03a] (see also [Lis03b]). The analysis is general, fully automatic and works for arbitrary control flow and can give potentially very complex and detailed formulae expressed in the input variables of a program. In [CB02], a WCET analysis which computes a formula given in some chosen set of function parameters, is presented. In this method, flow constraints has to be manually provided. Two methods of parametric WCET analyses are presented in [VHMW01] and [CHMW07]. They are both parameterised in loop bounds only and they do not take global constraints into consideration. A method similar to the one outlined in [Lis03a] were presented in [AHLW08], but it is using loop and path analyses instead of abstract interpretation. It requires special treatment of loops and is not as accurate as polyhedral abstract interpretation. A method which computes the complexity of a program is presented in [GMC09]. This method derives symbolic bounds of the complexity of the code only and does not take hardware into consideration, and cannot be used to obtain WCET estimations.

Chapter 3

Framework

In this chapter we introduce and formalise a framework for static WCET analysis based on abstract interpretation and counting states. This framework is based on the ideas published in [Lis03a] and [ESG+07]. This chapter will introduce the theoretical foundations of the framework while the two following chapters will go into details how to use the framework in practice.

3.1 Program Syntax

In this thesis, our general notion of a *program* is a piece of software; a task, a function, a full system or even just a loop. In order to have a simple and language-independent representation of programs they are represented by flowcharts. Furthermore, we shall assume that all variables of a program are integer valued. While this may seem like a strong restriction, the control flow of programs are usually governed by integers. Also, it is often easy to gener-

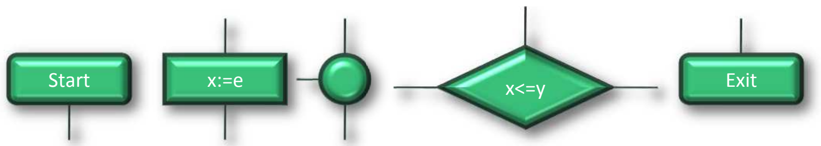


Figure 3.1: Flow Chart Nodes

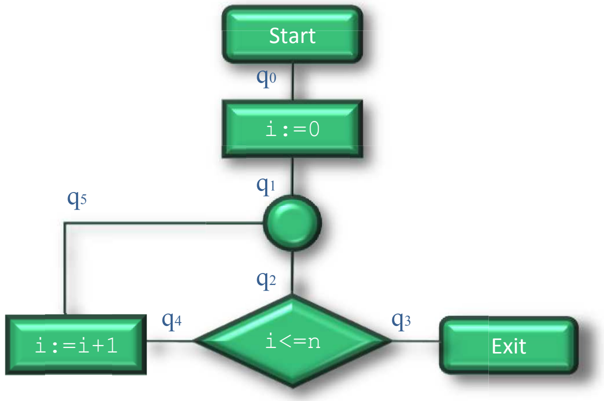


Figure 3.2: An example program L

alise analyses to other data types, but our representation becomes simpler if restricted to integers.

Definition 1. *A* program $P = \langle V_P, \mathcal{Q}_P, \mathcal{V}_P \rangle$ *is a piece of software, represented by a flow chart. The set* V_P *is the set of flow chart nodes (see Figure 3.1). The set* $\mathcal{Q}_P \subseteq V_P \times V_P$ *is the set of arcs in a flow chart, these will be referred to as* program points*. The set* $\mathcal{V}_P$ *denotes a set of program variables.*

Every program is assumed to have single entry and exit points, where the arc immediately connected to the entry (or start) node is referred to as the *initial program point* q_0, and the arc connected to the exit node is called the *final program point*.

As an example, a program $L = \langle V_L, \{q_0, q_1, q_2, q_3, q_4, q_5\}, \{i, n\} \rangle$ is depicted in Figure 3.2. This program will be used as a running example of the analysis techniques throughout the thesis.

3.1.1 Input Parameters

Since the execution time of a program varies with input, and this framework aims to provide a parametric WCET we shall make an important definition.

Definition 2. *Each program* P *is assumed to have a set of* input parameters $\mathcal{I}_P$ *which is a set of symbolic parameter corresponding to concrete values which* *affect the program flow of P.*

Depending on the type of program analysed, the input parameters can mean different things. If a function is analysed, the input parameters may correspond to the values of the formal parameters of the function. For a component, the input parameters may correspond to the data of the input ports of the component. For a loop, an input parameter may correspond to a given loop bound. For a task, the input parameter may correspond to the initial value of a global variables etc. The important thing is that the value of an input parameter in some way affects the timing of the program under analysis.

As an example, consider the program in Figure 3.2. Here, the initial value of n is a suitable input parameter of L, since the execution time of L is dependent on it. Thus, we can assume that the initial value of n is n_0, and consequently $\mathcal{I}_L = \{n_0\}$. Note here that n_0 is treated as a symbolic parameter rather than an absolute constant.

3.2 Program Semantics

The previous section defined how to represent programs without attaching any meaning to them. As the meaning of the different flow charts nodes should be straightforward to understand, we will not attach a formal definition of their semantics. However, in order to be able to reason about programs, we need to be able to reason about the run-time states of a program.

Definition 3. *An* environment *of a program P is a mapping $\sigma_P : \mathcal{V}_P \to \mathbb{Z}$. In other words, an environment is an assignment for every variable to an integer. The set of all environments of a program P is denoted Σ_P.*

Definition 4. *A* state $\langle q, \sigma \rangle \in \mathcal{Q}_P \times \Sigma_P$ *of a program P is a program point associated with an environment. The set of states $\mathcal{Q}_P \times \Sigma_P$ is denoted $\mathcal{S}_P$.*

Informally, an environment can be said to be a memory configuration and a state is a memory configuration together with the program pointer. With Definition 3 and 4 we can now reason formally about the run-time states of programs.

Definition 5. *The* semantic function τ_P *of a program P is a partial mapping $\tau_P : \mathcal{S}_P \hookrightarrow \mathcal{S}_P$, mapping one state to another.*

The semantic function defines the meaning of the program, i.e., formally defines what each flow chart node does to the current state. The mapping is partial since the function is not defined for the final program point. Again, while it is possible to give a formal definition of τ_P for each type of flow chart node, we shall refrain from doing so since it is not significant for the rest of the developments in this section.

3.2.1 Initial and Final States

Any program has a set of initial states $I_P \subseteq \mathcal{S}_P$ and a set of final states $F_P \subseteq \mathcal{S}_P$[1]. The initial states are all associated with the initial program point (i.e., they all have the form $\langle q_0, \sigma\rangle$). Conversely, the final states are all associated with the final program point. The environment associated with the initial state can be any environment[2]. However, some initial values may correspond to *input parameters* $\mathcal{I}_P$ of a program. Such variables will affect the execution and the initial configuration of these variables will therefore lead to different executions.

Definition 6. *The* semantic closure function $\tau_P^* : \mathcal{S}_P \to F_P$ *of a program is recursively defined as*

$$\tau_P^*(s) = \begin{cases} s & \text{if } s \in F_P \\ \tau_P^*(\tau_P(s)) & \text{otherwise} \end{cases}$$

The semantic closure function maps any state into a final state if it terminates. It is not defined for non-terminating programs.

3.2.2 Example

As an example, consider program L from Figure 3.2 again. Below is a demonstration on how to compute the final state from a given initial state. Choosing an initial state consists in determining values for each input parameter in $\mathcal{I}_L$. Since $\mathcal{I}_L = \{n_0\}$, this comes down to choosing an initial value for n, in this example we set n_0 to 2. We denote an environment where n maps to 2 as $[n \mapsto 2]$. Thus, we choose the initial state $\langle q_0, [i \mapsto i_0][n \mapsto 2]\rangle$. Note that the initial value of i does not matter since i is assigned before it is used, hence an arbitrary value i_0 is chosen for i. To compute the semantics of executing L on this initial state, we compute $\tau_L^*(\langle q_0, [i \mapsto i_0][n \mapsto 2]\rangle)$. Without having

[1]Except programs which never terminate, but such programs are uninteresting for analysis purposes.

[2]It is common to assume that the memory state before a program executes is undefined.

defined τ_L formally, the reader should have no problems understanding the following intuitively. We shall as in this example often omit the subscript (in this case L) when no ambiguity occurs.

$$\begin{aligned}
\tau^*(\langle q_0, [i \mapsto i_0][n \mapsto 2]\rangle) &= \tau^*(\langle q_1, [i \mapsto 0][n \mapsto 2]\rangle) \\
\tau^*(\langle q_2, [i \mapsto 0][n \mapsto 2]\rangle) &= \tau^*(\langle q_4, [i \mapsto 0][n \mapsto 2]\rangle) \\
\tau^*(\langle q_5, [i \mapsto 1][n \mapsto 2]\rangle) &= \tau^*(\langle q_2, [i \mapsto 1][n \mapsto 2]\rangle) \\
\tau^*(\langle q_4, [i \mapsto 1][n \mapsto 2]\rangle) &= \tau^*(\langle q_5, [i \mapsto 2][n \mapsto 2]\rangle) \\
\tau^*(\langle q_2, [i \mapsto 2][n \mapsto 2]\rangle) &= \tau^*(\langle q_4, [i \mapsto 2][n \mapsto 2]\rangle) \\
\tau^*(\langle q_5, [i \mapsto 3][n \mapsto 2]\rangle) &= \tau^*(\langle q_2, [i \mapsto 3][n \mapsto 2]\rangle) \\
\tau^*(\langle q_3, [i \mapsto 3][n \mapsto 2]\rangle) &= \langle q_3, [i \mapsto 3][n \mapsto 2]\rangle \,.
\end{aligned}$$

Thus, the semantics of executing L with initial state $\langle q_0, [i \mapsto i_0][n \mapsto 2]\rangle$ is to derive the state $\langle q_3, [i \mapsto 3][n \mapsto 2]\rangle$.

3.2.3 Program Timing

In this thesis we will mainly focus on flow analysis, but for a WCET analysis to estimate realistic times, a low-level analysis is needed. As explained in Chapter 2, a low-level analysis is far from trivial. In this work, it is assumed that a low-level analysis exists and that it can provide worst-case execution times for each atomic part of the program. In reality, these atomic parts might have different timings depending on execution history, that is, they may depend on cache and pipeline contents as well as branch predictors etc. In our framework, we will associate each program point with a worst-case execution time. While this may seem pessimistic, it should be possible for most analyses in the framework to add artificial program points to handle cases such as loop unrolling, cache-hit, cache-miss cases etc. However, in order to stay clear from details, we will assume that each edge in the flow chart has exactly one atomic WCET. Therefore, the results of the low-level analysis will be a function, associating an atomic WCET for each program point:

$$\ell : \mathcal{Q}_P \to \mathbb{Z}$$

The value domain can be milliseconds, clock cycles or whatever measure suitable for the application. Below, some possible values for the atomic WCETs of program L are shown. These values are often referred to in forthcoming examples.

$$\begin{array}{ll} \ell(q_0) = 1 & \ell(q_1) = 3 \\ \ell(q_2) = 1 & \ell(q_3) = 2 \\ \ell(q_4) = 2 & \ell(q_5) = 8 \end{array} \qquad (3.1)$$

3.3 Trace Semantics

Trace semantics [Cou01] is informally defined as all possible execution traces of a program. To formally define the tracing semantics, we need to notion of a *trace*. A trace $\mathcal{S}^+$ is a non-empty, possibly infinite string of states. The *trace closure function* $T : \mathcal{S} \to \mathcal{S}^+$ computes the unique trace corresponding to an initial state. If $T(s_0) = s_0, s_1, ...$ is a trace, then let s_j be any element in $T(s_0)$, then s_j is defined as

$$s_j = \tau(s_{j-1})$$

if $j \geq 1$. Note that if $T(s)$ is a terminating trace (that is, a finite trace ending in a final state), then all but a finite number of states in the trace are undefined. The *length* of a trace is defined as the largest defined index in the string. Having this formal definition of a trace given an initial state, we can define the full trace semantics of a program as

$$\mathcal{TS}_P = \{T(s) \mid s \in I_P\} \ .$$

Thus, the trace semantics $\mathcal{TS}_P$ is the set of all complete execution traces of a program.

3.3.1 Computing the global WCET of a program

Theoretically, if $\mathcal{TS}_P$ could be efficiently computed and all program states were associated with a worst-case execution time, the worst-case execution time of P could be computed by exhaustively computing the cost of each trace in $\mathcal{TS}_P$ and chose the maximum of these.

Table 3.1 shows the computation of the worst-case execution time of a single trace of program L (see Figure 3.2). The first column shows the trace, the second column shows the cost consumed by the particular state (taken from (3.1)), and the last column shows the accumulated cost for the whole trace.

In summary, the trace corresponding to the initial state $\langle q_0, [i \mapsto i_0][n \mapsto 2]\rangle$ has a worst-case execution time of 40. However, there are several reasons why

Trace	Cost	Acc. WCET
$\langle q_0, [i \mapsto i_0][n \mapsto 2]\rangle$	$\ell(q_0) = 1$	1
$\langle q_1, [i \mapsto 0][n \mapsto 2]\rangle$	$\ell(q_1) = 3$	4
$\langle q_2, [i \mapsto 0][n \mapsto 2]\rangle$	$\ell(q_2) = 1$	5
$\langle q_4, [i \mapsto 0][n \mapsto 2]\rangle$	$\ell(q_4) = 2$	7
$\langle q_5, [i \mapsto 1][n \mapsto 2]\rangle$	$\ell(q_5) = 8$	15
$\langle q_2, [i \mapsto 1][n \mapsto 2]\rangle$	$\ell(q_2) = 1$	16
$\langle q_4, [i \mapsto 1][n \mapsto 2]\rangle$	$\ell(q_4) = 2$	18
$\langle q_5, [i \mapsto 2][n \mapsto 2]\rangle$	$\ell(q_5) = 8$	26
$\langle q_2, [i \mapsto 2][n \mapsto 2]\rangle$	$\ell(q_2) = 1$	27
$\langle q_4, [i \mapsto 2][n \mapsto 2]\rangle$	$\ell(q_4) = 2$	29
$\langle q_5, [i \mapsto 3][n \mapsto 2]\rangle$	$\ell(q_5) = 8$	37
$\langle q_2, [i \mapsto 3][n \mapsto 2]\rangle$	$\ell(q_2) = 1$	38
$\langle q_3, [i \mapsto 3][n \mapsto 2]\rangle$	$\ell(q_3) = 2$	40

Table 3.1: Computation of the WCET of a trace

this is not a feasible approach. First of all, the computation of $\mathcal{TS}_P$ is undecidable in general (since it may contain infinite traces for non-terminating programs). Even if the program in question was guaranteed to terminate on all input, the computation of $\mathcal{TS}_P$ would be far too costly to use in practice, due to the often overwhelmingly large number of initial states. The computation of $\mathcal{TS}_P$ would essentially be equivalent to simulating the execution of P on all possible input combinations. That being said, computing $\mathcal{TS}_P$ and calculate the cost for each trace (under the assumption of an exact low-level analysis) would be an exact method of finding the global WCET of a program and will act as an optimal model of our method. However, to make this efficiently computable, a number of abstractions have to be made on top of this.

3.4 Collecting Semantics

Since trace semantics is too complex to use as basis for WCET analysis, a first abstraction is to consider a *set of states* rather then a set of traces. If the order in which states are visited is forgotten and also in which traces the states belongs to, then the problem becomes simpler. That is to say, the problem of computing the set of possible states that may occur during any execution is a simpler problem than to compute the set of possible traces that may occur during any execution. The set of states which may occur during any execution of a program is known as the *collecting semantics* [Cou01]. To define the collecting semantics of a program we use a function $\mathcal{CS}_P : \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S})$ defined as follows

$$\mathcal{CS}_P(\mathcal{S}) = \mathcal{S} \cup \{\tau_P(s) | s \in \mathcal{S}\} \cup I_P$$

This function takes a set of states and adds the immediate successor states of these. Note that it always contains the initial states of P. Using this function, we can formally define the collecting semantics of a program. The following result is stated in [CC77] using results from [Tar55].

Proposition 1. *The following two statements are equivalent*

1. *S contains all states which may occur during execution of P and S does not contain any state which may not occur during an execution of P.*

2. *S is the least set (wrt. inclusion) such that $S = \mathcal{CS}_P(S)$. I.e., S is the* least fixed point *of $\mathcal{CS}_P$.*

Statement 2 above expresses that the collecting semantics can be defined as the least fixed point of the function $\mathcal{CS}_P$. The reason for expressing the semantics as such is that there exist standard techniques for solving fixed point equations, which will be shown in Section 3.6.

3.4.1 Computing the WCET of a Program Using Collecting Semantics

With collecting semantics there is no information about execution traces and we cannot compute the WCET for individual traces using this technique. To be able to compute the worst-case execution time we instead claim the following two things.

- In any finite execution trace, each state occurs at most one time, and as a consequence:

- the number of environments associated with a program point is an upper bound of the number times the program point can be visited in *any* execution.

By using these claims we will be able to give an accurate upper bound of the WCET of a program without having information about the traces. First we will prove that these claims actually hold.

Lemma 1. *In any finite trace $T = s_0, ..., s_{n-1}$, state s_j occurs exactly once in T.*

Proof. Assume for contradiction that $s_i = s_j$ and that $i \neq j$. Then $s_{i+1} = \tau(s_i)$ by definition of a trace. Then we have that $s_{i+1} = \tau(s_i) = \tau(s_j) = s_{j+1}$. By induction we have that for all $m \in \mathbb{N}$ we have that $s_{i+m} = s_{j+m}$. Since T is finite there exist an m such that s_{j+m} is the final state. But since $s_{i+m} = s_{j+m}$, then s_{i+m} must be a final state too. But the assumption says that $i \neq j$ so T must have two different final states, which is a contradiction. $\square$

Lemma 2. *Let $\mathcal{CS}_P$ denote the collecting semantics for P. Partition $\mathcal{CS}_P$ into $|\mathcal{Q}_P|$ partitions $\{\mathcal{CS}_P^q \mid q \in \mathcal{Q}\}$, where each partition $\mathcal{CS}_P^q$ contains environments associated with an element $q \in \mathcal{Q}_P$. Then $|\mathcal{CS}_P^q|$ is an upper bound on the number of times program point q occurs in* any *finite trace T.*

Proof. Since a state s occurs maximum one time in any finite trace T (according to lemma 1), a state in the collecting semantics can be visited *maximum one time per finite trace*. The collecting set CS_P^q contains all states associated with program point q that can be reached during any finite execution trace. Since each state can be visited maximum once per trace, this is naturally an upper limit on how many times q can be visited in a single trace. $\square$

Using the result from Lemma 2, a naïve upper bound of the global WCET of the program can be derived if all traces of the program are finite. By computing the partitions $\mathrm{CS}_P^{q \in \mathcal{Q}}$, we can see that

$$\mathrm{WCET}_P \leq \sum_{q \in \mathcal{Q}} \ell(q) |\mathcal{CS}_P^q| \, . \tag{3.2}$$

The reason for this should be obvious; the execution time cannot be greater than the cost of visiting a program point multiplied with the maximum number of times it may be visited, summed for all program points in the program. This, in itself, may not be a very tight bound since there is little risk that the program visits all program points the maximum time. Therefore, in order to tighten the bound, techniques can be used to ”reconstruct” parts of the traces by using the program structure. As an example, detection of infeasible paths can provide useful information.

The framework presented in this thesis is founded on (3.2), but with additional techniques to find a tighter bound. One subtlety which should not be missed in this context is that this is based on the assumption that for any analysed program P, *all traces are finite*. If not all traces are finite, Lemma 1 is no longer valid (a non-terminating loop may visit the same state an unlimited number of times) and the technique can not be applied. However, the whole problem of finding the WCET of a program which have non-terminating branches is moot anyway, so this is not a major restriction.

Section 3.6 introduces fixed point theory which is a technique used to solve equations like $S = \mathcal{CS}_P(S)$. However, as will be seen, the collecting semantics is not in general computable (even though abstracting the trace semantics) and further approximations will therefore be introduced in Section 3.7.

3.5 Control Variables

A first step to reduce over-approximations which may be introduced in (3.2) is to realise that all variables in $\mathcal{V}_P$ do not need to be present in the computation of the collecting semantics for the purpose of counting the collected states. A *control variable* is a variable which directly or indirectly affects the control flow of a program. In other words, control variables are variables which affect the expressions in conditional nodes. *Non-control* variables are all variables which are not control variables. We will prove that non-control variables can be disregarded in the computation for the purpose of computing the size of states by showing that states which only differ in non-control variables must come from different execution traces.

Definition 7. *Partition $\mathcal{V}_P$ into control C_P and non-control NC_P variables, so that $\mathcal{V}_P = C_P \cup NC_P$. Then two states $s = \langle q, \sigma \rangle$, $s' = \langle q', \sigma' \rangle$ are considered to be* control equivalent, *denoted $s \sim s'$, iff $\forall v \in C_P : \sigma(v) = \sigma'(v) \wedge q = q'$.*

In other words, $s \sim s'$ iff s and s' belongs to the same program point and all control variables map to the same value. Note that $\sim$ is an equivalence relation on $\mathcal{S}_P$.

Lemma 3. *If $s_0 \sim s_1$ then $\tau(s_0) \sim \tau(s_1)$. Furthermore, it also holds that $\tau^n(s_0) \sim \tau^n(s_1)$ for all $n \in \mathbb{N}$*

Proof. Let $s_0 = \langle q_0, \sigma_0 \rangle$ and $s_1 = \langle q_0, \sigma'_0 \rangle$, and let $s_0 \sim s_1$ (by definition of $\sim$, s_0 and s_1 needs to be associated with the same program point q_0). First we prove that if $\tau(s_0) = \langle q_1, \sigma_1 \rangle$, then $\tau(s_1) = \langle q_1, \sigma'_1 \rangle$, i.e., τ maps s_0 and s_1 to the same program point q_1.

Assume for contradiction that τ would map q_0 to q_1 for s_0 and that it would map q_0 to q_2 for s_1 and that $q_1 \neq q_2$, i.e., that two different paths were executed for s_0 and s_1. But since $s_0 \sim s_1$, all control variables maps to the same values, which means that it is impossible for τ to map s_0 and s_1 to different paths, so $\tau(s_0)$ and $\tau(s_1)$ must map to the same program point q_1. Now, let $\tau(s_0) = \langle q_1, \sigma_1 \rangle$ and $\tau(s_1) = \langle q_1, \sigma_1' \rangle$. We will now show that $\forall v \in C_P : \sigma_1(v) = \sigma_1'(v)$. First of all, it holds that $\forall v \in C_P : \sigma_0(v) = \sigma_0'(v)$, since $s_0 \sim s_1$. Assume that there is a $v_0 \in C_P$ such that $\sigma_1(v_0) \neq \sigma_1'(v_0)$. This means that one variable which is *not* in C_P has changed the value of v_0 through the image of τ (since the variables in C_P are the same for σ_1 and σ_1' per assumption). However, the variables in NC_P may not in any way affect the variables in C_P (again, per definition), so this may not happen. Thus, we must reach the conclusion that $\langle q_1, \sigma_1 \rangle \sim \langle q_1, \sigma_2 \rangle$, in other words $\tau(s_0) \sim \tau(s_1)$. As a consequence of the transitivity of $\sim$ we can also draw the conclusion that $\tau^n(s_0) \sim \tau^n(s_1)$ for all $n \in \mathbb{N}$. ☐

Lemma 3 shows that the control equivalent relation holds during the full execution of a trace, which leads to the following important proposition:

Proposition 2. *Let s_0 be a state belonging to a finite trace t_0, and let s_1 be a state belonging to a finite trace t_1. Assume that $s_0 \neq s_1$, then $s_0 \sim s_1 \Rightarrow t_0 \neq t_1$.*

This is to say that two control equivalent states cannot be on the same finite trace.

Proof. Let $s_0 = \langle q, \sigma \rangle, s_1 = \langle q, \sigma' \rangle$. Assume that $s_0 \sim s_1 \wedge s_0 \neq s_1$. Assume for contradiction that s_0 and s_1 belongs to the same trace t. Without loss of generality, we can assume that s_0 precedes s_1 in t. Since s_0 precedes s_1 in t, there exists an n_0 such that $\tau^{n_0}(s_0) = s_1$. By Lemma 3 we have that $\tau^n(s_0) \sim \tau^n(s_1)$ for any $n \in \mathbb{N}$, so $s_1 = \tau^{n_0}(s_0) \sim \tau^{n_0}(s_1) = s_2 = \langle q, \sigma'' \rangle$. Accordingly, we define the state s_k as $\tau^{kn_0}(s_0)$ and deduce that $\tau^{kn_0}(s_0) = \langle q, \sigma_k \rangle$ for all $k \in \mathbb{N}$. This implies that t visits q infinitely many times, and thus t is an infinite trace which never reaches the final state and thus contradicts the assumption that t is a finite trace which both s_0 and s_1 belongs to. ☐

Proposition 2 suggests that two states which differ only in the values of non-control variables must belong to *different traces*. This means, effectively, that when counting the states associated with a program point, states which differs only in non-control variables (i.e., which are control equivalent) need only to be counted *once*, since they by Proposition 2 are *guaranteed* to belong to different traces. The summary of this is that non-control variables can be completely disregarded from analysis, since multiple states with different non-control variables do not contribute to the upper bound of the times which that particular program point can be visited. Program slicing (see Section 2.1.4) can be used to identify and remove all statements and variables which do not affect control flow. This is done by slicing with respect to all conditionals and all variables in the conditionals (see [SEGL06] for details).

3.6 Fixed Point Theory

Section 3.4 introduced the collecting semantics which is the theoretical basis for the framework presented in this thesis. The collecting semantics can be formulated as a fixed point equation (see Proposition 1 on page 24). This section introduces some elementary domain theory in order to develop a method to solve fixed point equations. Details about domain theory can be found in [NNH05, AJ94].

Definition 8. *(Poset)*
A poset *(or partially ordered set) $\langle L, \sqsubseteq_L \rangle$ is a set and a relation such that $\sqsubseteq_L$ is*

- *reflexive:* $\forall l \in L : l \sqsubseteq_L l$
- *anti-symmetric:* $\forall l, m \in L : l \sqsubseteq_L m \land m \sqsubseteq_L l \Rightarrow l = m$
- *and transitive:* $\forall k, l, m \in L : k \sqsubseteq_L l \land l \sqsubseteq_L m \Rightarrow k \sqsubseteq_L m$

Definition 9. *(Upper and lower bounds)*
Let $\langle L, \sqsubseteq \rangle$ be a poset and let $M \subseteq L$. An element $u \in L$ is an upper bound *of M if it holds that $m \sqsubseteq u$ for all $m \in M$. Conversely, an element $l \in L$ is considered to be a* lower bound *if $l \sqsubseteq m$ for all $m \in M$.*

Definition 10. *(Supremum and infimum)*
Let $\langle L, \sqsubseteq \rangle$ be a poset and let $M \subseteq L$. If M has upper bounds and there exist an upper bound u_0 such that for all other upper bounds $u \in L$ it holds that $u_0 \sqsubseteq u$, then u_0 is the supremum *of M and is denoted $\sqcup M$. Similarly, if M has lower bounds and the exist a lower bound l_0 such that for all other lower bounds $l \in L$ it holds that $l \sqsubseteq l_0$, then l_0 is the* infimum *of M and is denoted $\sqcap M$. The supremum or infimum of a subset $M \subseteq L$ is always unique if it exists.*

Note that a subset of a poset does not necessarily have upper and lower bounds, and if they do, they don't necessarily have a infimum or supremum. A poset L such that for all subsets $M \subseteq L$, $\sqcup M$ and $\sqcap M$ exists, is called a *complete lattice*. Since $L \subseteq L$, this also means that a complete lattice has a supremum, which in domain theory is commonly refered to as the *top* element of L, denoted $\top_L$. The infimum of L, conversely, is called the *bottom* element of L and is denoted $\bot_L$.

Definition 11. *(Monotone functions) Let $\langle L, \sqsubseteq_L \rangle$ and $\langle L', \sqsubseteq_{L'} \rangle$ be a posets and let $f : L \to L'$ be a function. Then f is a* monotone *or* order-preserving *function iff*

$$l \sqsubseteq_L m \Rightarrow f(l) \sqsubseteq_{L'} f(m)$$

A well-known and important result of monotone functions on complete lattices is that for any monotone self-map over a complete lattice has a *least fixed point*.

Proposition 3. *(Tarski [Tar55])*
Let L be a complete lattice and $f : L \to L$ be a monotone function. Then the set $\text{fix} f = \{l \in L \mid f(l) = l\}$ *is a complete lattice.*

A consequence of Proposition 3 is that since $\text{fix} f$ is a complete lattice, $\sqcap(\text{fix} f)$ is the least element in this lattice, and consequently the *least fixed point* of f. In order to compute this fixed point, a few more definitions are needed.

Definition 12. *(Chains)*
Let L be a complete lattice, then $M \subseteq L$ is a chain *if it is non-empty and for all elements $m, m' \in M$ either $m \sqsubseteq m'$ or $m \sqsupseteq m'$.*

In other words, a chain is a subset of a complete lattice where the elements are completely ordered. Thus, chains can be described as decreasing or increasing sequences (e.g., $m_0 \sqsubseteq m_1 \sqsubseteq ...$).

Definition 13. *(Continuity)*
A monotone function $f : L \to L$ is Scott-continuous *iff that for every chain $M \subseteq L$, it holds that $f(\sqcup M) = \sqcup \{f(m) \mid m \in M\}$.*

A constructive result on how to compute the least fixed point (lfp) of a continuous function can be presented. This result is due to Kleene, and is not presented in its full generality here.

Proposition 4. *(Kleene [Kle52])*
Let L be a complete lattice and $f : L \to L$ a Scott-continuous function, then

$$\text{lfp} f = \bigsqcup \{f^n(\bot) \mid n \in \mathbb{N}\}$$

This result basically says that starting by $\perp$ and iteratively compute f until a fixed point is reached, will obtain the least fixed point of f. Of course, this requires the ascending sequence $(f^n(\perp))_{n \in \mathbb{N}} = \perp \subseteq f(\perp) \subseteq f(f(\perp)) \subseteq ...$ to reach a fixed point in a finite number of steps to be useful.

3.7 Abstract Interpretation

In this chapter we have introduced the collecting semantics as the theoretical basis for the framework in this thesis. As hinted in Section 3.4, collecting semantics can not in general be computed, so even more abstractions have to be layered on top of it to make it efficiently computable.

Abstract Interpretation [CC77] is a well-known technique to soundly approximate program semantics. The collecting semantics is defined as the smallest possible set of states which can be reached during any execution of a program, while with abstract interpretation it is possible to derive a superset of the collecting semantics (abstract semantics) in a computable and efficient manner. A superset of the collecting semantics may naturally have less exact information since it may contain states which actually never occur during any execution, but the information is still *sound* in the sense that there is *no* state which *may* occur during execution but which is not present in the derived set of states. Abstract interpretation approximates semantics according to some property of choice, this is formalised by choosing an appropriate *abstract domain* to use as abstraction of the semantics. A great variety of abstract domains can be formulated and the choice of domain offers a trade-off between precision and computational complexity. Examples of abstract domains are presented in Section 3.9. The following sections will introduce the theory of abstract interpretation.

3.7.1 Abstraction

The idea of abstract interpretation is to have a certain relationship between two complete lattices. One lattice is referred to as the *concrete domain* L and the other as the *abstract domain* M. The intention is to have the abstract domain approximating the concrete domain. This is done by having a *Galois connection* $\langle L, \alpha, \gamma, M \rangle$ between the two lattices, consisting of an *abstraction* function $\alpha : L \rightarrow M$ and a *concretisation* function $\gamma : M \rightarrow L$. The relationship is depicted in Figure 3.3.

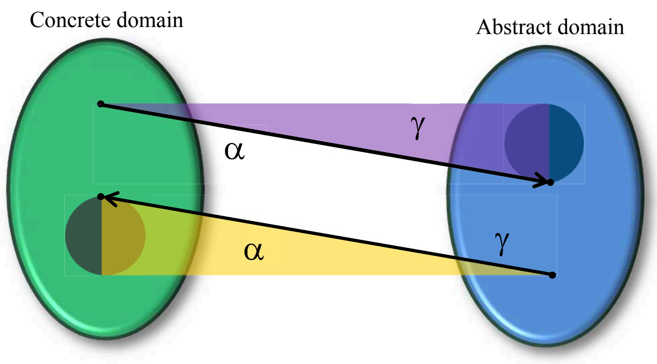


Figure 3.3: Relation between the concrete and abstract domain

Definition 14. *A* Galois connection $\langle L, \alpha, \gamma, M \rangle$ *is a tuple consisting of two complete lattices* L, M *and two monotone functions* $\langle \alpha, \gamma \rangle \in (L \to M) \times (M \to L)$*, such that*

$$\alpha \circ \gamma \sqsubseteq_M \lambda m.m \text{ and } \gamma \circ \alpha \sqsubseteq_L \lambda l.l$$

If it also holds that $\alpha \circ \gamma \sqsupseteq_M \lambda m.m$*, then* $\langle L, \alpha, \gamma, M \rangle$ *is called a* Galois insertion.

In general it is desired to have a Galois insertion rather than a Galois connection since any concrete element has exactly one abstract element describing it.

Example

As an example of a Galois connection, consider $\mathsf{sign} = \langle L, \alpha, \gamma, M \rangle$, where the concrete domain is $L = \langle \mathcal{P}(\mathbb{Z}), \subseteq \rangle$ and the abstract domain is $M = \langle \{\bot, -, 0, +, \top\}, \sqsubseteq \rangle$ with an ordering as shown in Figure 3.4. We then form

$$\begin{tikzcd}
 & \top & \\
- \arrow[ur, "\sqsubseteq"] & 0 \arrow[u] & + \arrow[ul, "\sqsubseteq"'] \\
 & \bot \arrow[ul, "\sqsubseteq"] \arrow[u] \arrow[ur, "\sqsubseteq"'] &
\end{tikzcd}$$

Figure 3.4: The lattice of signs

the following Galois connection:

$$\begin{aligned}
\gamma(\bot) &= \varnothing & \alpha(\varnothing) &= \bot \\
\gamma(-) &= \mathbb{Z}_- & \alpha(A) &= - \text{ iff } \forall a \in A : a < 0 \\
\gamma(0) &= \{0\} & \alpha(\{0\}) &= 0 \\
\gamma(+) &= \mathbb{Z}_+ & \alpha(A) &= + \text{ iff } \forall a \in A : a > 0 \\
\gamma(\top) &= \mathbb{Z} & \alpha(A) &= \top \text{ in all other cases}
\end{aligned}$$

Note that Definition 14 holds for $\langle \alpha, \gamma \rangle$. The intuition behind this is that the set of integers are abstracted by sign by this Galois connection. The α function *abstracts* a set by mapping the set into its minimum representation in the abstract domain. As an example, consider the set $\{1, 2, 3\} \in \mathcal{P}(\mathbb{Z})$. The abstract version of this element is obtained by $\alpha(\{1, 2, 3\}) = +$. The set $\{1, 2, 3\}$ is represented by a ”+” in the abstract domain, meaning that the set is a set of positive integers. The ”meaning” of this set is obtained by mapping this abstract representation back into the concrete domain via γ. We see that $\gamma(+) = \mathbb{Z}_+$. Mapping to the abstract domain and back makes us lose precision; from the concrete set $\{1, 2, 3\}$ of three numbers, ”abstracting” the set and ”concretising” it again gives us only the information that the original set was a set of positive integers.

3.7.2 Abstract Functions

By using abstract interpretation it is possible ”simulate” the usage of functions over a complex lattice by performing the functions over the abstract lattice instead. Doing this may under some assumptions turn undecidable problems decidable, but then naturally with some lost precision. Let $\langle L, \alpha, \gamma, M \rangle$ be a

Galois-connection and let $f : L \to L$ be a monotone function over the concrete lattice L. Then we say that $\widehat{f} : M \to M$ is *approximating* f or that $\widehat{f}$ is an *abstract version* of f, iff

$$\forall l \in L : f(l) \sqsubseteq_L \gamma \circ \widehat{f} \circ \alpha(l) \ .$$

This relation is depicted in Figure 3.5. The idea here is that $\widehat{f}$ gives a correct interpretation of the semantics of f, but with possible loss of information. As an example, consider the Galois connection $\mathrm{sign} = \langle L, \alpha, \gamma, M \rangle$ from Section 3.7.1 again. First, consider the *lifted multiplication operation* $\cdot_P : \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$ defined as

$$A \cdot_P B = \{a \cdot b \mid a \in A \wedge b \in B\} \ .$$

This operation is simply normal multiplication defined over sets of integers, for instance, $\{1,2,3\} \cdot_P \{-1,-2\} = \{-1,-2,-3,-4,-6\}$. This is an operation on our concrete domain $\mathcal{P}(\mathbb{Z})$ and is the operation which we are interested to approximate. Now, we define the abstract multiplication $\widehat{\cdot} : M \to M$ as follows

$$\begin{aligned} +\,\widehat{\cdot}\,+ &= + \\ -\,\widehat{\cdot}\,- &= + \\ -\,\widehat{\cdot}\,+ &= - \\ 0\,\widehat{\cdot}\,a &= 0 \\ \top\,\widehat{\cdot}\,a &= \top \\ \bot\,\widehat{\cdot}\,b &= \bot \end{aligned}$$

where a is any non-bottom element and b is any element. This is a correct definition of an abstract operation, which should be easy to verify. As an example, we see that

$$\begin{gathered} \{1,2,3\} \cdot_P \{-1,-2\} = \{-1,-2,-3,-4,-6\} \sqsubseteq \\ \gamma(\alpha(\{1,2,3\})\,\widehat{\cdot}\,\alpha(\{-1,-2\})) = \gamma(+\,\widehat{\cdot}\,-) = \\ \gamma(-) = \mathbb{Z}_- \end{gathered}$$

When abstract interpretation is applied in static analysis, the abstract functions approximates functions available in the programming language semantics. In this thesis we are restricted to integer valued variables, and will be interested in approximating functions over integers (such as addition, subtraction, multiplication and addition), i.e., functions of the type $f : \mathbb{Z}^n \to \mathbb{Z}$.

$$\begin{tikzcd}
L \arrow[r, "f"] \arrow[d, "\alpha"'] & L \\
M \arrow[r, "\widehat{f}"] \arrow[ur, phantom, "\sqsubseteq"] & M \arrow[u, "\gamma"']
\end{tikzcd}$$

Figure 3.5: Relation between concrete and abstract functions

However, the concrete domain used in abstract interpretation operates over sets of integers rather than integers themselves. Thus, for any n-ary operation, $f : \mathbb{Z}^n \to \mathbb{Z}$, it is possible to define a *lifted* version $f_P : \mathcal{P}(\mathbb{Z})^n \to \mathcal{P}(\mathbb{Z})$ defined as

$$f_P(X_0, ..., X_{n-1}) = \{f(x_0, ..., x_{n-1}) \mid x_i \in X_i \text{ for all } 0 \leq i < n\}$$

In practice, when operations over the integers are used, the concrete domain will be $\mathcal{P}(\mathbb{Z})$, correspondingly, it is the lifted versions of the operations that will be approximated. For this reason, we will from now on use the abusive notation f for f_P in the context of abstract operations. Note that lifted functions are always monotone.

Fixed Points of Abstract Functions

The reason to formulate abstract functions is that abstract interpretation is performed over the abstract functions rather than the concrete ones to obtain a correct result without having to iterate over the concrete and often not practically computable lattice. A basic result from abstract interpretation is that for any monotone functions $f : L \to L$ and $\widehat{f} : M \to M$ such that $\widehat{f}$ is approximating $\widehat{f}$. Then

$$\text{lfp} f \sqsubseteq \gamma(\text{lfp}\ \widehat{f}\ ).$$

This means that the least fixed point of the abstract function is a safe approximation of the least fixed point of the concrete function.

3.7.3 Widening and Narrowing

To find the least fixed point of a monotone operator $f : L \to L$ over a lattice L, two cumbersome requirements are imposed on L and f:

- f must be Scott-continuous.

- The sequence $\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq ...$ stabilises after a finite number of steps.

By "stabilises after a finite number of steps", we mean that there exist a $k \in \mathbb{N}$ such that for the increasing sequence $(f^n(\bot))_{n\in\mathbb{N}} = \bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq$ ... it holds that $f^k(\bot) \sqsupseteq f(f^k(\bot))$. In practice, these two requirements are too restrictive. Thus, a way of approximating the fixed point without these requirements is desired. The solution is to introduce a so-called *widening operator* [CC77].

Definition 15. *A widening operator $\nabla : L \times L \to L$ is an operator over a lattice fulfilling $\forall l, l' : l, l' \sqsubseteq (l \nabla l')$ and for any increasing sequence $l_0 \sqsubseteq l_1 \sqsubseteq ...$, the increasing sequence $l_0 \sqsubseteq l_0 \nabla l_1 \sqsubseteq l_0 \nabla l_1 \nabla l_2 \sqsubseteq ...$ eventually stabilises.*

Thus if the sequence $(f^n(\bot))_{n\in\mathbb{N}} = \bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq ...$ is replaced by the sequence $(f^n_\nabla)_{n\in\mathbb{N}} = \bot \sqsubseteq f(\bot)\nabla\bot \sqsubseteq f(f(\bot))\nabla f(\bot)\nabla\bot \sqsubseteq$ ..., then the sequence will eventually stabilise at $\mathrm{lfp}(f_\nabla)$ and by definition of the widening operation, it will hold that $\mathrm{lfp}(f_\nabla) \sqsupseteq \mathrm{lfp}(f)$. Thus, an approximation of the fixed point can be found in a finite number of steps. This to the cost of possible lost precision; the widening operation is greater or equal to the supremum of its arguments. However, the situation can be improved by having a somewhat dual concept of a *narrowing* operator.

Definition 16. *A narrowing operator $\Delta : L \times L \to L$ is an operator over a lattice fulfilling $\forall l, l' : l \sqsubseteq l' \to l \sqsubseteq (l \Delta l') \sqsubseteq l'$ and for any decreasing sequence $l_0 \sqsupseteq l_1 \sqsupseteq ...$, the decreasing sequence $l_0 \sqsupseteq l_0 \Delta l_1 \sqsupseteq l_0 \Delta l_1 \Delta l_2 \sqsupseteq ...$ eventually stabilises.*

Note that the sequence $(f^n_\nabla)_{n\in\mathbb{N}}$ is stable at $\mathrm{lfp}(f_\nabla)$, so the sequence $\mathrm{lfp}(f_\nabla) \sqsupseteq f(\mathrm{lfp}(f_\nabla)) \sqsupseteq f(f(\mathrm{lfp}(f_\nabla))) \sqsupseteq ...$ is a decreasing sequence, in fact it is a stable sequence, the inequalities could be replaced by equality signs. Thus, any element in this sequence is greater or equal to $\mathrm{lfp}(f)$. Furthermore, the sequence $\mathrm{lfp}(f_\nabla) \sqsupseteq f(\mathrm{lfp}(f_\nabla))\Delta\mathrm{lfp}(f_\nabla) \sqsupseteq ...$ eventually stabilises (by definition of the narrowing operator) at $\mathrm{lfp}(f^\Delta_\nabla)$ and any element in the sequence is greater or equal to $\mathrm{lfp}(f)$ as shown in [NNH05]. Thus,

$$\mathrm{lfp}(f) \sqsubseteq \mathrm{lfp}(f^\Delta_\nabla) \sqsubseteq \mathrm{lfp}(f_\nabla).$$

In summary, to find a good approximation of the least fixed point of f in finite time without the requirements that f is Scott-continuous or that the increasing sequence $(f^n)_{n\in\mathbb{N}}$ should stabilise:

- First compute $\mathrm{lfp}(f_\nabla)$
- Use $\mathrm{lfp}(f_\nabla)$ as starting point and compute $\mathrm{lfp}(f_\nabla^\Delta)$.

3.8 Abstract Interpretation in Static Analysis

In this section we will explain how abstract interpretation usually is applied in static analysis, and in particular how it is applied in our framework. The basic idea is that the power set of program states $\mathcal{P}(\mathcal{S}_P)$ are used as concrete domain in the abstract interpretation, and an *abstract semantic function* $\mathcal{AS}_P$ approximating the collecting semantic $\mathcal{CS}_P$ over a program P is used. The abstract interpretation is then formulated as computing the least fixed point of the abstract semantic function as a correct approximation of the collecting semantics of the program. To abstract the collecting semantics, it is necessary to abstract the semantic function τ_P of a program including all operations used to define τ_P. This consist of choosing an appropriate abstract domain, and lifting all used concrete operations to abstract ones in the chosen domain.

3.8.1 Widening and Narrowing in Static Analysis

If the abstract domain used in static analysis contains infinite ascending chains it is necessary to introduce widening as described in Section 3.7.3. However, the abstract semantic function transfers states from states which are associated with program points in the flow chart. In [CC77] it is shown that it is sufficient to use the widening operator at least once per cycle in the flow chart to ensure termination. Thus, instead of using the sequences described in Section 3.7.3, it is sufficient to introduce the widening operation once per loop, more on this is presented in Section 3.8.4.

3.8.2 Relational vs. Non-Relational Domains

A semantic state of a program P is, as seen in Section 3.2, a program point associated with an environment. Consequently, the set of semantic states of a program is a subset of

$$\mathcal{P}(\mathcal{Q} \times (\mathcal{V}_P \to \mathbb{Z}))$$

Since the actual set of program points is not going to be abstracted, it is sufficient to consider the following set as the concrete domain

$$\mathcal{Q} \to \mathcal{P}(\mathcal{V}_P \to \mathbb{Z}) \ .$$

That is, each program point is associated with a *set* of environments. Then, it is sufficient to use $\mathcal{P}(\mathcal{V}_P \to \mathbb{Z})$ as concrete domain. In practical cases of analysis (see Chapter 5), the real value domain of variables are not the mathematical set of integers $\mathbb{Z}$ but a finite set of integers, limited by the representation of integers in the computer. So by imagining a finite value domain, say $\mathbb{Z}_{2^{32}}$, which means the set of integers modulus 2^{32} and set

$$\mathcal{P}(\mathcal{V}_P \to \mathbb{Z}_{2^{32}}) \ .$$

as the concrete domain. The number of elements in $\mathcal{V}_P \to \mathbb{Z}_{2^{32}}$ is $(2^{32})^{|\mathcal{V}_P|}$, and the full domain has two to the power of $(2^{32})^{|\mathcal{V}_P|}$ as cardinality. Thus, even with a finite domain, the size of possible program states associated with a single program point grows in $O(2^{(2^{32})^n})$ where n is the number of variables. Even if this set is abstracted using an appropriate abstract domain, the possibilities to abstract this set somewhat accurately is getting hard with many variables. To remedy this, the concrete domain can be approximated using a domain which is not affected by the number of variables. Consider a function,

$$\mathcal{V}_P \to \mathcal{P}(\mathbb{Z})$$

as concrete domain. I.e., a function which associates each variable with a *set* of integers. This is, a direct abstraction of $\mathcal{P}(\mathcal{V}_P \to \mathbb{Z})$, as can be seen by formulating the Galois-connection $\langle \mathcal{P}(\mathcal{V}_P \to \mathbb{Z}), \alpha, \gamma, \mathcal{V}_P \to \mathcal{P}(\mathbb{Z}) \rangle$ as follows:

$$\alpha(\Sigma) = \lambda v. \{\sigma(v) \mid \sigma \in \Sigma\}$$
$$\gamma(\sigma) = \{\lambda v.n \mid n \in \sigma(v)\}$$

The loss in this abstraction is that the relation between variables are abstracted away; each variable is associated with a set of values independently of other values, while before abstraction, a set of *functions* of variables were associated with a program point. In this abstraction there is no need to include the variables in the abstraction, it is enough to use the simple set $\mathcal{P}(\mathbb{Z})$ as the concrete domain, and perform analysis once per variable. Since the concrete lattice does not grow in number of variables, the analysis is much less sensitive to the number of variables. To summarise, performing analysis over the concrete domain

$$\mathcal{P}(\mathcal{V}_P \to \mathbb{Z})$$

is called *relational abstract interpretation*, since the relation between variables are preserved through the abstraction. Using the concrete domain

$$\mathcal{V}_P \to \mathcal{P}(\mathbb{Z})$$

is called *non-relational abstract interpretation* because no relation between variables are preserved. The choice of concrete domain (and consequently, also abstract domain) is a trade-off between computational complexity and precision. A non-relational domain is much simpler and faster to analyse but loses more precision than a relational one. In Section 3.9 some abstract domains of both classes are presented.

3.8.3 Terminology in Abstract Interpretation in Static Analysis

It is often necessary to talk about abstraction in different layers, therefore it is necessary to have a terminology for the abstract counterparts of values, environments and states. This terminology in turn depends on if the abstraction is relational or non-relational. In this section we shall make clear definitions of the terms used in the rest of the thesis and what they mean. The first definition mainly concerns non-relational abstract domains.

Definition 17. *Let $\langle \mathcal{P}(\mathbb{Z}), \alpha, \gamma, A \rangle$ be a Galois-connection, then*

- *An element $a \in A$ shall be referred to as an* abstract value.
- *An element $\widehat{\sigma} \in (\mathcal{V} \to A)$ is called an* abstract environment.
- *An element $s' \in \mathcal{Q} \to \mathcal{V} \to A$ is called an* abstract state.

Note that all these sets can be seen as lattices, in particular, they can be seen as abstractions of the concrete counterparts. The ordering of the lattices are given below. Let $\widehat{\sigma}$ and $\widehat{\theta}$ be abstract environments, then

$$\widehat{\sigma} \sqsubseteq_\Sigma \widehat{\theta} \Leftrightarrow \forall v \in \mathcal{V} : \widehat{\sigma}(v) \sqsubseteq_A \widehat{\theta}(v) .$$

Similarly, let s and s' be abstract states, then

$$s \sqsubseteq_{\mathcal{S}} s' \Leftrightarrow \forall q \in \mathcal{Q} : s(q) \sqsubseteq_\Sigma s'(q)$$

For relational domains, a similar terminology is used, but with a slightly different meaning.

Definition 18. *Let $\left\langle \mathcal{P}(\mathcal{V} \to \mathbb{Z}), \alpha, \gamma, \widehat{\Sigma} \right\rangle$ be a Galois-connection, then*

- *An element $\widehat{\sigma} \in \widehat{\Sigma}$ is called an* abstract environment

- *An element $s' \in \mathcal{Q} \to \widehat{\Sigma}$ is called an* abstract state

Note that in a relational domain it does not make sense to talk about "abstract values" since the abstraction is directly on the environments. Also notice that the terms *abstract environment* and *abstract state* are similar enough to use on both non-relational domains and relational domains without causing confusion. So in summary, for non-relational and relational domains we shall use the notation A for the set of abstract values, $\widehat{\Sigma}$ for the set of abstract environments and $\widehat{\mathcal{S}}$ for the set of abstract states, all subscripted with the program under consideration if necessary.

3.8.4 Abstract Interpretation over Flow Charts

Our representation of programs are flow charts, so a static analysis needs to be defined over such. This section will define the abstract semantic function $\widehat{\tau} : \widehat{\mathcal{S}} \to \widehat{\mathcal{S}}$ for the different types of arcs in a flow chart. For any arc $q \in \mathcal{Q}$ we shall denote its predecessor arcs as q_{pre}. For merge nodes, which have two incoming arcs, the second is denoted $q_{\mathrm{pre}'}$. In the following, a partial definition of $\widehat{\tau}$ is given for every type of program point.

Start arc. At the start arc q, nothing is known about the values of variables, except possibly the variables which are corresponding to input parameters. However, assuming that the abstract interpretation is correct for *all* possible combinations of input, we make the assertion that nothing is known about the input parameters[3]. Having said this, the natural definition of an abstract state associated with the initial state should be as follows:

$$\widehat{\tau}(S)(q) = \top_{\widehat{\Sigma}}$$

Assignment arc. An assignment arc is an arc which emerges from an assignment node. An assignment node has an assignment `x := e` associated with it, where `x` is a variable and `e` is an arithmetical formula. The abstract function should equal to the previous abstract environment with the variable `x` updated to the abstract value of `e`, as follows:

$$\widehat{\tau}(S)(q) = S(q_{\mathrm{pre}})[x \mapsto \alpha(e)]$$

where $f[x \mapsto y]$ for a function f means

$$f[x \mapsto y] = \lambda v. \begin{cases} y & \text{if } v = x \\ f(v) & \text{otherwise.} \end{cases}$$

[3]In some cases it can be useful to put known constraints on the initial state though.

Note that the computation of $\alpha(e)$ requires abstract versions of all arithmetical operations in `e`.

Merge arc. A merge arc is an arc emerging from a merge node. A merge node combines the analysis results of the two incoming arcs. The least abstract value which is correct with respect to both incoming values is the supremum of the these. In addition, if the merge node is the entry of a loop, then that is a good place to put the widening, if the abstract domain requires that. Thus, the abstract transition function for merge nodes is

$$\begin{aligned} \widehat{\tau}\,(S)(q) &= S(q)\nabla(S(q_{\text{pre}}) \sqcup S(q_{\text{pre}'})) && \text{if loop merge} \\ \widehat{\tau}\,(S)(q) &= S(q_{\text{pre}}) \sqcup S(q_{\text{pre}'}) && \text{otherwise} \end{aligned}$$

Conditional arcs. The conditional node has two outgoing arcs. While there seems to be no standard approach of handling conditionals for non-relational domains, we shall adopt the same approach as in [AH87]. A more elaborate and constructive method is presented in [Gus00]. Conditionals are resolved by relations, so for an abstract domain it is necessary to have abstract version of all relations. An abstract relation is a function $\widehat{\leq}\,: A^n \to \texttt{bool}_\perp$, where $\texttt{bool}_\perp$ is the lattice shown in Figure 3.6. The outgoing arc then reflects the largest (wrt. $\sqsubseteq$) abstract value which is at least `true` respective `false`. For a conditional `a <= b`, the transfer function for a true-arc q is as follows:

$$\widehat{\tau}\,(S)(q) = \bigsqcup \left\{ \widehat{\sigma}\, \sqsubseteq S(q_{\text{pre}}) \mid \widehat{\sigma}\,(\texttt{a}) \,\widehat{\leq}\, \widehat{\sigma}\,(\texttt{b}) \sqsubseteq \texttt{true} \right\}.$$

Similarly, the transfer function for a false-arc q is:

$$\widehat{\tau}\,(S)(q) = \bigsqcup \left\{ \widehat{\sigma}\, \sqsubseteq S(q_{\text{pre}}) \mid \widehat{\sigma}\,(\texttt{a}) \,\widehat{\leq}\, \widehat{\sigma}\,(\texttt{b}) \sqsubseteq \texttt{false} \right\}.$$

3.8.5 Abstract Interpretation Example

To see how abstract interpretation is used in practice, this section shows a thorough example of applying abstract interpretation to program L depicted in Figure 3.2 on page 18. The sign abstraction presented in Section 3.7.1 is used in the abstract interpretation. The sign abstraction is a non-relational domain

$$\begin{tikzcd}
 & \top & \\
\texttt{true} \arrow[ur, "\sqsubseteq"] & & \texttt{false} \arrow[ul, "\sqsubseteq"'] \\
 & \bot \arrow[ul, "\sqsubseteq"] \arrow[ur, "\sqsubseteq"'] \arrow[uu] &
\end{tikzcd}$$

Figure 3.6: The lattice of Booleans $\texttt{bool}_\bot$

since it is an abstraction of $\mathcal{P}(\mathbb{Z})$. The equations defining $\widehat{\tau}$ is as follows:

$$\begin{aligned}
\widehat{\tau}(S)(q_0) &= \top \\
\widehat{\tau}(S)(q_1) &= S[i \mapsto \alpha(0)] \\
\widehat{\tau}(S)(q_2) &= S(q_2)\nabla(S(q_1) \sqcup S(q_5)) \\
\widehat{\tau}(S)(q_3) &= \bigsqcup\left\{ \widehat{\sigma} \sqsubseteq S(q_2) \mid \widehat{\sigma}(i) \mathrel{\widehat{\leq}} \widehat{\sigma}(n) \sqsubseteq \texttt{true}\right\} \\
\widehat{\tau}(S)(q_4) &= \bigsqcup\left\{ \widehat{\sigma} \sqsubseteq S(q_2) \mid \widehat{\sigma}(i) \mathrel{\widehat{\leq}} \widehat{\sigma}(n) \sqsubseteq \texttt{false}\right\} \\
\widehat{\tau}(S)(q_5) &= S[i \mapsto S(q_4)(i) \mathbin{\widehat{+}} \alpha(1)]
\end{aligned}$$

As seen in the calculation for q_5, an abstract operation for $+$ is needed to evaluate the abstract value. This set of equations is the same for all non-relational abstract domains, to make it more specific for the sign domain, we can specify a more sign-domain specific set of equations,

$$\begin{aligned}
\widehat{\tau}(S)(q_0) &= \top_{\text{sign}} \\
\widehat{\tau}(S)(q_1) &= S[i \mapsto \texttt{0}] \\
\widehat{\tau}(S)(q_2) &= S(q_1) \sqcup S(q_5) \\
\widehat{\tau}(S)(q_3) &= \bigsqcup\left\{ \widehat{\sigma} \sqsubseteq S(q_2) \mid \widehat{\sigma}(i) \mathrel{\widehat{\leq}} \widehat{\sigma}(n) \sqsubseteq \texttt{true}\right\} \\
\widehat{\tau}(S)(q_4) &= \bigsqcup\left\{ \widehat{\sigma} \sqsubseteq S(q_2) \mid \widehat{\sigma}(i) \mathrel{\widehat{\leq}} \widehat{\sigma}(n) \sqsubseteq \texttt{false}\right\} \\
\widehat{\tau}(S)(q_5) &= S[i \mapsto S(q_4)(i) \mathbin{\widehat{+}} +]
\end{aligned}$$

Note here that widening is not necessary since the sign-lattice does not contain any infinite ascending chains. The abstract semantics of L is now specified as the fixed point of the function $\widehat{\tau}$, and can be computed by the iterates of $\widehat{\tau}^{\,n}(\bot)$. So set $S = \lambda q.\bot$, then the first iterate $\widehat{\tau^1}$ will contain the following:

$$\begin{aligned}
\widehat{\tau}^{\,1}(\bot)(q_0) &= \top\\
\widehat{\tau}^{\,1}(\bot)(q_1) &= \bot[i \mapsto \mathsf{0}]\\
\widehat{\tau}^{\,1}(\bot)(q_2) &= \bot[i \mapsto \mathsf{0}]\\
\widehat{\tau}^{\,1}(\bot)(q_3) &= \bot[i \mapsto \mathsf{0}]\\
\widehat{\tau}^{\,1}(\bot)(q_4) &= \bot[i \mapsto \mathsf{0}]\\
\widehat{\tau}^{\,1}(\bot)(q_5) &= \bot[i \mapsto \mathsf{0} \mathbin{\widehat{+}} + = +]
\end{aligned}$$

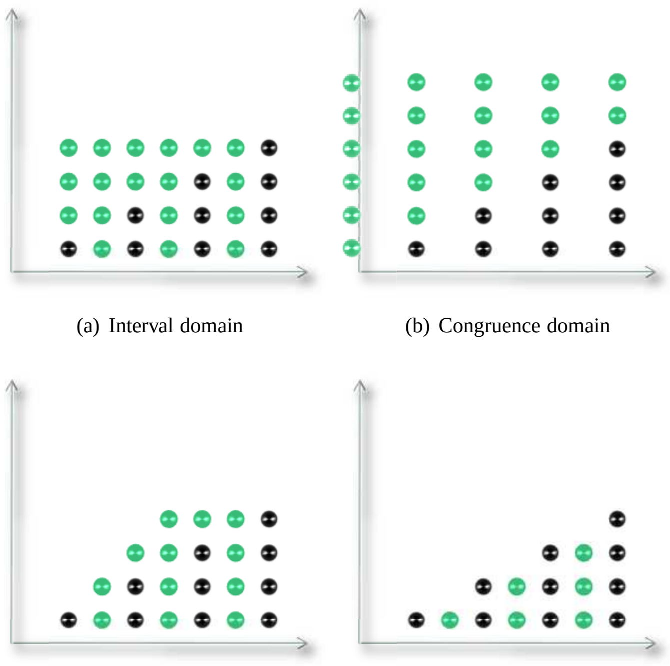
We use Jacobi iteration here, i.e., when computing $\widehat{\tau}^{\,j}(\bot)(q_n)$ we use the values computed for $\widehat{\tau}^{\,j}(\bot)(q_m)$ for all $m < n$, and $\widehat{\tau}^{\,j-1}(\bot)(q_n)$ if $m \geq n$. New here is the calculation of the abstract operation $\widehat{+}$, but it should be obvious that a positive value added to zero is again a positive value. Next iteration:

$$\begin{aligned}
\widehat{\tau}^{\,2}(\bot)(q_0) &= \top\\
\widehat{\tau}^{\,2}(\bot)(q_1) &= \bot[i \mapsto \mathsf{0}]\\
\widehat{\tau}^{\,2}(\bot)(q_2) &= \bot[i \mapsto \top]\\
\widehat{\tau}^{\,2}(\bot)(q_3) &= \bot[i \mapsto \top]\\
\widehat{\tau}^{\,2}(\bot)(q_4) &= \bot[i \mapsto \top]\\
\widehat{\tau}^{\,2}(\bot)(q_5) &= \bot[i \mapsto \top \mathbin{\widehat{+}} + = \top]
\end{aligned}$$

The reason i maps to $\top$ in $\widehat{\tau}^{\,2}(\bot)(q_2)$ is that the union of the values from q_1 and q_5 gives rise to the computation $+ \sqcup \mathsf{0} = \top$. The third iterate will be equal to the second one, meaning that a fixed point is reached and the result is the least fixed point. The sign lattice is a very simple lattice so a fixed point could be reached really fast (3 iterations), but as seen, not much information about the program has been obtained; all that can be said now is that i will always be zero at q_1. Note that nothing can be said about n for any non-relational abstraction since n is never assigned any value in L.

3.9 Abstract Domains

In this section we will give examples of a few abstract domains commonly used in literature. All domains used in this section abstracts sets of integers (since



(a) Interval domain (b) Congruence domain

(c) Octagon domain (d) Polyhedral domain

Figure 3.7: Examples of abstract domains. The black dots correspond to the set which is being abstracted, the others denotes over-approximation.

this is the most common abstraction), and with simple modifications they can all be used to abstract the collecting semantics of a program.

This section is divided into two parts, one which handles non-relational domains, which uses $\mathcal{P}(\mathbb{Z})$ as concrete domain, and the other one presents relational domains, which uses $\mathcal{P}(\mathbb{Z}^n)$ as concrete domain (note that this is equivalent to $\mathcal{P}(\mathcal{V} \to \mathbb{Z})$ if $|\mathcal{V}| = n$).

3.9.1 Non-Relational Abstract Domains

Non-relational domains all have in common that they are abstractions of the set $\mathcal{P}(\mathbb{Z})$. Non-relational domains can be used to quickly determine properties of program variables independently.

The Interval Domain

One of the most commonly used abstract domains is the interval domain [CC77]. The interval domain has a simple representation consisting of the form $[a, b]$ for non-bottom values, where $a, b \in \mathbb{Z} \cup \{-\infty, \infty\}$. The interval domain abstracts a set of integers as an interval of integers ranging from the minimum element to the maximum. The details of the lattice is given below:

$$\begin{aligned}
[a, b] \sqsubseteq [a', b'] &\Leftrightarrow a \geq a' \wedge b \leq b' \\
[a, b] \sqcup [a', b'] &\overset{\text{def}}{=} [\min(a, a'), \max(b, b')] \\
[a, b] \sqcap [a', b'] &\overset{\text{def}}{=} [\max(a, a'), \min(b, b')] \\
\top &\overset{\text{def}}{=} [-\infty, \infty]
\end{aligned}$$

Note that if the result from $\sqcap$ by two intervals results in an interval $[a, b]$ where $a > b$, it is interpreted as $\bot$.

The abstraction and concretisation functions are as follows

$$\begin{aligned}
\alpha(\varnothing) &= \bot \\
\alpha(A) &= [\min(A), \max(A)] \\
\gamma(\bot) &= \varnothing \\
\gamma([a, b]) &= \{n \in \mathbb{Z} \mid a \leq n \leq b\}
\end{aligned}$$

The interval domain contains infinite ascending chains, such as

$$[0, 1] \sqsubseteq [0, 2] \sqsubseteq ... \sqsubseteq [0, \infty]$$

indicating that a widening (and narrowing) operation needs to be introduced. The widening and narrowing suggested in [CC77] are

$$[a, b] \nabla [a', b'] = [\text{if } a' < a \text{ then } -\infty \text{ else } a', \text{if } b' > b \text{ then } \infty \text{ else } b']$$

The narrowing can be defined as follows

$$\begin{aligned}
[a, b] \Delta [a', b'] = [&\text{if } a = -\infty \text{ then } a' \text{ else } \min(a, a'), \\
&\qquad \text{if } b = \infty \text{ then } b' \text{ else } \max(b, b')]
\end{aligned}$$

The interval domain is used to find upper and lower bounds of variables and have been is commonly used in WCET analysis [GESL07, The04]. The domain is depicted in Figure 3.7(a).

The congruence domain

The congruence domain was introduced by Phillippe Granger in [Gra89] and is somewhat of an orthogonal concept to the interval domain. The congruence domain abstracts a set of integers to their least possible congruence class. The abstract elements are on the form $m + k\mathbb{Z}$ for non-bottom values[4]. The details of the lattice is given below:

$$\begin{aligned}
&m + k\mathbb{Z} \sqsubseteq m' + k'\mathbb{Z} \Leftrightarrow m - m' \in k'\mathbb{Z} \wedge k\mathbb{Z} \subseteq k'\mathbb{Z}\\
&m + k\mathbb{Z} \sqcup m' + k'\mathbb{Z} \overset{\text{def}}{=} m + \gcd\{|m - m'|, k, k'\}\mathbb{Z}\\
&m + k\mathbb{Z} \sqcap m' + k'\mathbb{Z}\\
&\overset{\text{def}}{=} \begin{cases} m'' + \operatorname{lcm}(k, k')\mathbb{Z} & \text{if } m'' \in m + k\mathbb{Z} \cap m' + k'\mathbb{Z}\\ \bot & \text{otherwise}\end{cases}\\
&\top = 0 + 1\mathbb{Z}
\end{aligned}$$

Here $\gcd$ and lcm stands for the *greatest common divisor* and the *least common multiple* respectively. Both can be applied to pairs (indicated by parenthesis) or sets (indicated by set-notation). The abstraction and concretisation maps are defined as follows

$$\begin{aligned}
\alpha(\varnothing) &= \bot\\
\alpha(A) &= a_0 + \gcd\{|a - a'| \mid a, a' \in A\}\,\mathbb{Z}\\
\gamma(m + k\mathbb{Z}) &= \{m + kn \mid n \in \mathbb{Z}\}\\
\gamma(\bot) &= \varnothing
\end{aligned}$$

where a_0 is the least non-negative number in A. The intention of the invention of the congruence domain was to aid compilers to perform automatic vectorisation [Gra89]. The congruence domain has successfully been used in conjunction with the interval domain (see Section 3.9.3). The domain is depicted in Figure 3.7(b).

3.9.2 Relational Abstract Domains

Relational abstract domains abstract the set $\mathcal{P}(\mathbb{Z}^n)$ (or equivalently $\mathcal{P}(\mathcal{V} \to \mathbb{Z})$ for $n = |\mathcal{V}|$) where n is a fixed number of dimensions. The number of dimensions usually correspond to the number of variables which should be analysed in the program. The complexity of these domains are directly related to the

[4]This can be read as "all elements which are equal to m modulus k"

number of dimensions. The domains presented in this section are all restricted to preserve only linear relationships between variables. This means that the abstract functions provided for these domain are restricted to linear assignments, meaning that non-linear assignments will in most cases be represented by the respective domain's top value.

The Polyhedral Domain

The polyhedral domain [CH78] approximates a set of integer points in n-dimensional space by the smallest (wrt. inclusion) possible convex polyhedron enclosing all points in the set (see Figure 3.7(d)). A convex polyhedron can be represented in two ways, one being a system of linear inequalities $A\mathbf{x} \leq \mathbf{b}$, the other being a set of vertices and rays $\langle V, R \rangle$ where the vertices represents the extreme points of a polyhedron an rays represents infinite lines in one direction in case the polyhedron is unbounded. Both these representations need to be used in order to efficiently compute the infimum (concatenation of linear inequalities) and supremum (concatenation of rays and vertices), as well as a method for converting between these two representations. The ordering in the lattice of convex polyhedra is simply the set-inclusion operator. Convex polyhedra have been used in several applications [BJT99, HPR94, Ben02, BL08], and there exist a few open source implementations of the domain [New09, Par09, Pol09, Apr09].

The Octagon Domain

The octagon domain was suggested by Miné in [Min01], and is a relational domain where a set of integer points are enclosed by the smallest possible octagon. An octagon can be expressed as a set of constraints on the form $\pm x \pm y \leq c$. The octagon domain is less precise domain than the polyhedral domain, but in compensation it is much more efficient. Octagons can efficiently be represented by *difference bound matrices* and graphs. There is an open source implementation of the octagon domain available [Apr09, Oct09]. An approximation of a set of points using the octagon domain is shown in Figure 3.7(c).

The Linear Congruence Domain

In [Gra91], a generalisation of the congruence domain is presented. This domain derives the congruence relationship between dimensions (variables) and is represented by a system of linear congruence relations $A\mathbf{x} \equiv \mathbf{m} \bmod \mathbf{k}$.

3.9.3 Domain Products

To increase the precision of an abstract domain, it is possible to combine domains to have an abstract domain which captures the properties of several abstract domains. This is known as a *product* of abstract domains [CC79]. The *direct product* of two abstract domains A and B is defined as the Galois connection $\langle C, \alpha, \gamma, A \times B \rangle$ where

$$\alpha(c) = \langle \alpha_A(c), \alpha_B(c) \rangle$$
$$\gamma(\langle a, b \rangle) = \gamma(a) \sqcap \gamma(b).$$

However, this connection does not preserve Galois insertions, meaning that the domains in the product do not take advantage of each other. In practical terms this means that this is equivalent to simply performing the analysis twice; once for each domain and taking the intersection of results. However, in many cases the domains can provide information for each other to give more precise results. This can be achieved via the *reduced product*, which is defined by introducing an equivalence relation on the abstract values and considering the abstract domain modulus this equivalence relation. How this is done technically depends on each pair of domains for which the reduced product is defined. A common reduced product domain is the reduced product of the interval and congruence domains [ESG$^+$07, Min06, VCKL05, BR04a].

3.10 Overview of the Framework

In this section we give an overview of a static WCET analysis framework based on abstract interpretation and counting of abstract environments. This is a framework based on standard static analysis techniques and it conforms to the classical layered model with flow analysis, low-level analysis and calculation. The basic idea is based on computing a set of possible states at each program point (i.e., the collecting semantics of a program) and observing the fact that the number of states is an upper bound of the times that a program can be visited (see Section 3.4.1), and thereby the WCET of a program can be approximated by the equation (3.2) on page 25, subject to constraints obtained from structural analysis and flow analyses. However, since collecting semantics is undecidable in the general case, the states are computed using abstract interpretation. The abstract states are usually represented by well-known mathematical spaces, and by counting the discrete elements in these spaces, an upper bound of the number of program points is obtained. For the interval

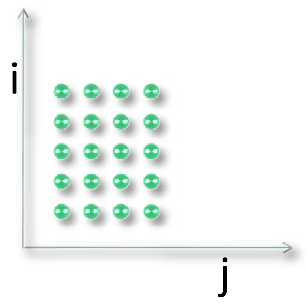


Figure 3.8: Abstract environment associated with the body of a simple nested loop

domain and other finite simple non-relational domain, the counting of discrete points is fairly straightforward, whereas in relational domains such as the polyhedral one, counting integer points might be more complex.

3.10.1 Slicing

While WCET analysis in general can benefit from using program slicing (see for example [SEGL06]), it is an integral part of this framework. The set of possible states in a program serves the basis as upper bounds of program points and the more combination of variable assignments there are, the larger these sets become. However, non-control variables do not contribute to the upper bound of the execution count of any program point as seen in Section 3.5. Thus, slicing can be used to remove all non-control variables.

3.10.2 Overview of Loop Bound Analysis

To derive a loop bound of a loop L the following steps are taken. First, slicing is used to remove statments and variables which do not affect control flow. Here non-control variables refer to variables which do not affect the control of the number of times the loop iterates. In addition, a slicing with respect to the exit conditions of L are also made to remove statements and variables which do not affect the number of times the loop iterates. Then, an invariant analysis is used to remove variables which are invariant (i.e., do not change) in the loop body. After that, abstract interpretation is used to derive a superset of the possible states that are visited in the loop header. Finally, the elements in the abstract states are counted to derive a concrete upper bound of a loop. As a simple example, consider the following program:

```
for $i = 1$ to 5 do
  {Loop 1}
  for $j = 1$ to 4 do
    {Loop 2}
    statement
  end for
end for
```

The program point corresponding to `statement` is bounded by the number of possible environments associated with it. Using abstract interpretation with the interval domain results in the environments shown in Figure 3.8. After slicing on Loop 2, the variable i can be taken out of consideration and it can effectively be bound to 4. The loop bounds are then used as constraints in the calculation phase to derive a concrete WCET. The analysis is described in detail in Chapter 4.

3.10.3 Overview of Parametric WCET Analysis

The parametric WCET framework is based on the same idea as the loop bound analysis but requires some additional steps. The framework is presented here similarly to the presentations in [BL08, BEL09]. The goal of the analysis is to derive a function $\mathrm{PWCET}_P : \mathbb{Z}^{|\mathcal{I}_P|} \to \mathbb{Z}$ as the WCET rather than a constant as in the classical case. The set $\mathcal{I}_P$ is the set of input parameters of P. Thus, the function takes $|\mathcal{I}_P|$ arguments which correspond to concrete values of the input parameters of a program. This function is constructed as the functional composition of two other functions; the *parametric calculation function* (PCF) and the *execution count function*[5] (ECF). The parametric calculation function of a program P

$$\mathrm{PCF}_P : \mathbb{N}^{|\mathcal{Q}_P|} \to \mathbb{N}$$

takes a vector of upper bounds (where $\mathcal{Q}_P$ is the set of program points of P) for *each program point* in the program and returns the worst case execution time given those bounds. This function is generated in the calculation phase of the analysis. Thus, the calculation phase has to be altered to a *parametric*

[5]Named maximum execution count function in [BL08].

calculation. More on this in Chapter 6. The execution count function of a program P

$$\mathrm{ECF}_P : \mathbb{Z}^{|\mathcal{I}_P|} \rightarrow \mathbb{N}^{|\mathcal{Q}_P|}$$

takes a vector of instantiated input parameters and returns a vector of upper bounds for each program point in the program. This function is generated during the flow analysis via abstract interpretation and symbolic state counting.

The composition of PCF and ECF gives a function which takes a vector of instantiated input parameters and returns the worst case execution time. Formally:

$$\mathrm{PWCET}_P : \mathbb{Z}^{|\mathcal{I}_P|} \rightarrow \mathbb{N} = \mathrm{PCF}_P \circ \mathrm{ECF}_P \ .$$

Computing PCF_P

The PCF_P function is computed by a parametric calculation [Lis03a, BL08, BEL09]. A parametric calculation works like the normal calculation phase in WCET analysis; it takes flow constraints and low-level analysis results and calculates the worst-case subject to these constraints. The difference is that the result of the parametric calculation is a formula in terms of *symbolic upper bounds*. These symbolic upper bounds can be loop bounds, upper bounds on paths [Lis03b], or execution bounds on program points. Two methods of parametric calculation have been investigated within this framework: Parametric Integer Programming (PIP) [Fea88] and the Minimum Propagation Algorithm (MPA) [BEL09]. These methods are discussed in detail in Chapter 6 and 7.

Computing ECF_P

The ECF_P function is computed in a very similar fashion as the loop bounds. By performing abstract interpretation to obtain super sets of the collecting semantics of a program, and then counting the number of elements in the abstract state an upper bound of the execution count of the program point has been computed. However, the bounds computed should be in terms of the input parameters of the program. This can be achieved by using a relational abstract domain. In a relational abstract domain information about the relationship between variables are preserved and can be used to compute tighter bounds. Consider the following loop

```
for $i = 1$ to n do
  for $j = 1$ to 4 do
    statement
  end for
```

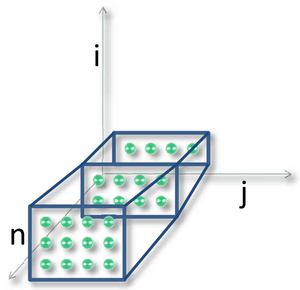


Figure 3.9: A part of an abstract environment associated with the loop body of a simple nested loop

```
  end for
```

where the initial value of n is an input parameter. Figure 3.9 shows a part of an abstract environment associated with `statement` computed using abstract interpretation with the polyhedral domain. As seen, the execution count of `statement` depends on the variable n.

Chapter 4

Finding Loop Bounds

4.1 Introduction

This chapter outlines a methodology for finding loop bounds based on counting elements in abstract states. This methodology was first introduced in [ESG$^+$07], and is shown here to illustrate the principles of counting elements in abstract states and to further motivate the developments of the congruence domain as shown in Section 5.3. A similar method has later been presented in [LCFM09].

The loop bound analysis relies on the principles outlined in Chapter 3. The method is presented using non-relational abstract interpretation since it in general is more efficient that relational ditto. Since the number of elements in an abstract state is an upper bound of the number of times a certain program point can be visited, a loop can easily be bounded by counting the elements of an abstract state corresponding to a program point which is visited in *every loop iteration*. However, some additional techniques are needed to obtain tight and finite loop bounds, namely *slicing* and *invariant analysis*. Invariant analysis [Muc97, ESG$^+$07] is a technique to find variables which are *invariant* in a loop, that is, variables which do not change in the loop body. Finding these variables are essential for two reasons: 1) they may be used to prevent the widening operation to severely over-estimate the result (or even make them unbounded) and 2) they may be used to substantially reduce the size of the resulting abstract states. How this is done is explained in the following sections.

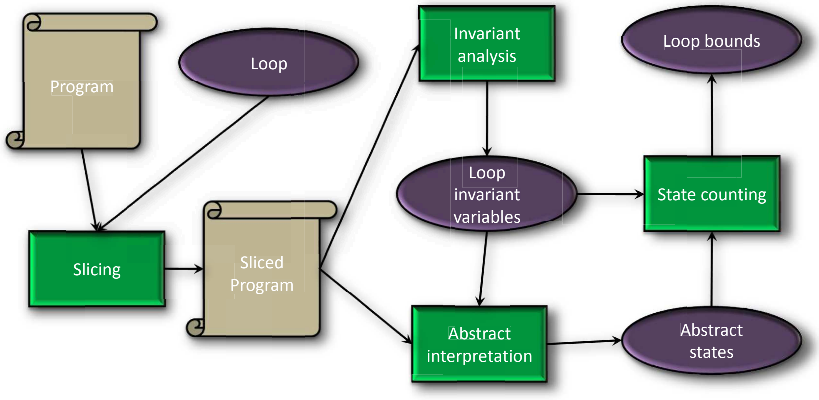


Figure 4.1: Workflow for the loop bound analysis

4.2 Slicing on Loops

To find out the bound of an individual loop, slicing can be applied with respect to the exit conditions of the loop, to obtain a program slice where most statements and variables that do not affect the number of loop iterations have been removed. As an example, consider the following nested loop:

```
for $i = 1$ to 10 do
  {Loop $L1$}
  for $j = 1$ to $i$ do
    {Loop $L2$}
    statement1
  end for
  statement2
end for
```

Assume that `statement1` and `statement2` do not affect i or j directly or indirectly. A slicing on $L1$ would result in the following loop:

```
for $i = 1$ to 10 do
  {Loop $L1$}
end for
```

since neither $L2$, `statement1` nor `statement2` affect the number of iterations of $L1$.

4.3 Loop Invariant Variables

As mentioned in previous section, it is an important step of this loop bound analysis to find loop invariant variables for the analysed loops. Each loop L has a set of loop invariant variables $\mathcal{I}_L$ defined as the set of variables of a program $\mathcal{V}_P$ minus the set of variables which are updated in the loop L.

A simple approach to find loop invariant variables is presented in [ESG+07]; it simply searches for variables which are used in a (sliced) loop body but which are not updated. In addition, the result from abstract interpretation may discover variables which are guaranteed to have only one value (e.g., the interval $[1, 1]$). Such variables must trivially be loop invariant as well.

4.4 Restricted Widening

The widening operation can sometimes yield imprecise results since it may not correspond to the least fixed point. Consequently, the loop bound analysis may not be able to bind some loops. When the widening is placed just before a conditional, as the case is in our flow charts, the widening may prevent the conditional from properly prune the abstract states after the conditional. Figure 4.2 illustrates this problem. In the left part of the figure, a first abstract interpretation iteration with the interval domain has been performed. The second incoming arc to the merge node, is in the first iteration the bottom value. The right part of the figure shows the second iteration where the second incoming arc to the merge node maps i and j to $[1, 9]$. Since $9 \geq 1$ the widening maps (see Section 3.9.1) this to $[1, \infty]$. This causes the true-arc of the conditional to map j to $[1, \infty]$ where $[1, 9]$ would have been more precise, yet still correct. We will solve this problem by using a restricted form of widening. This widening was used in the evaluation of [ESG+07] but was not explained or proved to be correct.

We define the *restricted widening operator* $\nabla_C : (\mathcal{V}_P \to A) \to (\mathcal{V}_P \to A)$ in terms of a widening operator $\nabla : (\mathcal{V}_P \to A) \to (\mathcal{V}_P \to A)$ as:

$$X \nabla_C Y = \lambda v. \begin{cases} Y(v) & \text{if } v \in C \\ X \nabla Y(v) & \text{otherwise} \end{cases}$$

where $C \subseteq \mathcal{V}_P$. To demonstrate the usage of the restricted widening, assume that q is a loop merge arc for a loop L, and that $\mathcal{I}_L$ is the set of loop invariant variables for L. Let p, r be incoming arcs to the merge node, then the corresponding data-flow equation for this node is:

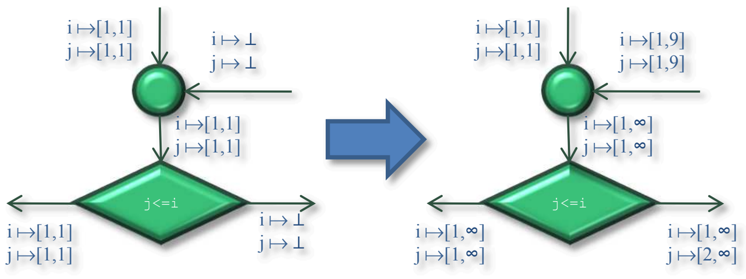


Figure 4.2: A widening is made just after the merge node. This causes the abstract values after the conditional to be grossly over-estimated.

$$\widehat{\tau}_{L}(S)(q) = S(q)\nabla_{\mathcal{I}_L}(S(p) \sqcup S(r))$$

This means that, in a non-relational domain, the widening is performed only over the variables which are possibly changed in the loop.

The reason that this is a valid approach is as follows. Let $(\sigma(v))_{n\in\mathbb{N}}$ be an infinite strictly increasing chain. This means that either the variable v is updated inside a loop, or v directly or indirectly depends on a variable which is updated inside a loop (since v must be updated an infinite number of times in order to cause an infinite strictly increasing chain). Consequently, if v is a loop invariant variable for a loop L and $(\sigma(v))_{n\in\mathbb{N}}$ is associated with a program point inside L, then v must be either be updated inside *another* loop L' or be dependent (directly or indirectly) on a variable v' which is updated in said loop. This means that the $(\sigma(v))_{n\in\mathbb{N}}$ is an infinite strictly increasing chain because it depends on another infinite strictly increasing chain $(\sigma'(v'))_{n\in\mathbb{N}}$ associated with the loop L'. Here, either $v' = v$ or v depends directly or indirectly on v'. However, since v' cannot be a loop invariant of L', applying the restricted widening $\nabla_{\mathcal{I}_{L'}}$ to $(\sigma'(v'))_{n\in\mathbb{N}}$ results in a ascending chain which eventually stabilises, since $v' \notin \mathcal{I}_{L'}$. Applying $\nabla_{\mathcal{I}_{L'}}$ to $(\sigma(v))_{n\in\mathbb{N}}$ in addition, would make also this chain to eventually stabilise, even though $v \in \mathcal{I}_L$ since the reason that $(\sigma(v))_{n\in\mathbb{N}}$ were strictly increasing was the dependency on the strictly increasing chain $(\sigma'(v'))_{n\in\mathbb{N}}$.

4.5 Abstract Interpretation in Loop Bound Analysis

Abstract interpretation is used in the loop bound analysis to find an over-approximation of the set of states reachable inside a loop. We will illustrate the loop bound analysis by using an example program TP, shown in Figure 4.3. This program is not doing anything useful; it is designed to illustrate the techniques. It is assumed to already have been sliced, thus, a slicing would not be able to remove any statements or variables of TP. The program consists of a nested triangular loop, and the developments of this chapter will be devoted on finding the number of iterations of the loop bodies. The two loops of TP are $L_1 = \{q_2, q_3, q_4, q_5, q_8, q_9\}$ and $L_2 = \{q_5, q_6, q_7\}$. However, slicing on L_1 would remove $\{q_4, q_5, q_6, q_7, q_8\}$ from it, leaving $L_1 = \{q_2, q_3, q_9\}$. Slicing on L_2 would not remove any program points.

An invariant analysis may detect the set of loop invariant variables as $\mathcal{I}_{L_1} = \{j\}$ and $\mathcal{I}_{L_2} = \{i\}$.

As seen in Figure 4.1, the next step is the abstract interpretation. We perform the abstract interpretation using three non-relational abstract domains to demonstrate that the achieved precision varies by using different domains. The domains used in this section are the interval domain, the congruence domain and the reduced product of these two. The definition of the abstract semantic function for non-relational domains is defined as follows for TP:

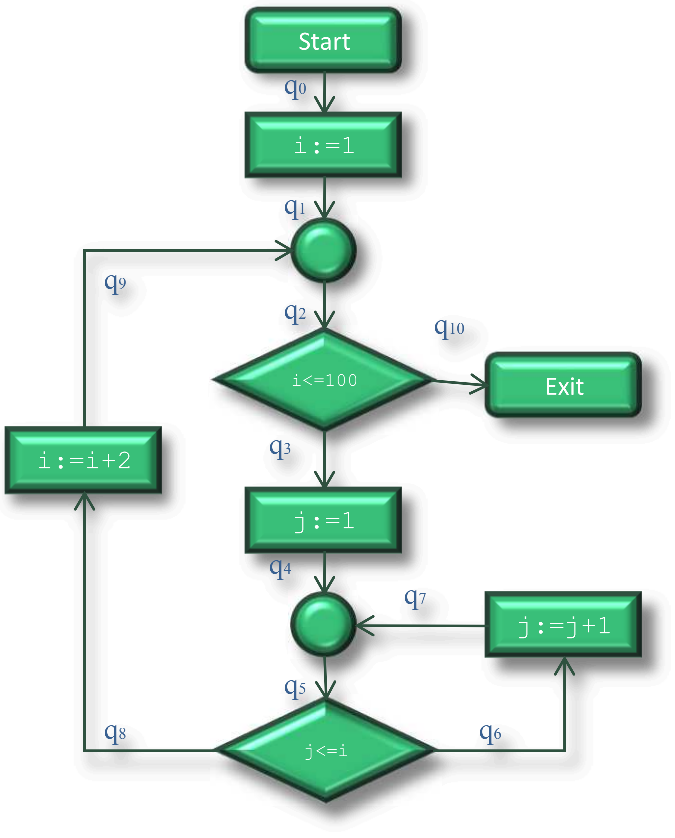


Figure 4.3: Triangular loop program, TP

$$
\begin{aligned}
\widehat{\tau}\,(S)(q_0) &= \top \\
\widehat{\tau}\,(S)(q_1) &= S(q_0)[i \mapsto \alpha(\{1\})] \\
\widehat{\tau}\,(S)(q_2) &= S(q_2)\nabla(S(q_1) \sqcup S(q_9)) \\
\widehat{\tau}\,(S)(q_3) &= \bigsqcup \left\{ \widehat{\sigma} \sqsubseteq S(q_2) \mid \widehat{\sigma}\,(i) \mathrel{\widehat{\leq}} \alpha(\{100\}) \sqsubseteq \texttt{true} \right\} \\
\widehat{\tau}\,(S)(q_4) &= S(q_3)[j \mapsto \alpha(\{1\})] \\
\widehat{\tau}\,(S)(q_5) &= S(q_5)\nabla_{\{i\}}(S(q_4) \sqcup S(q_7)) \\
\widehat{\tau}\,(S)(q_6) &= \bigsqcup \left\{ \widehat{\sigma} \sqsubseteq S(q_5) \mid \widehat{\sigma}\,(j) \mathrel{\widehat{\leq}} \widehat{\sigma}\,(i) \sqsubseteq \texttt{true} \right\} \\
\widehat{\tau}\,(S)(q_7) &= S(q_6)[j \mapsto S(q_6)(j) \mathbin{\widehat{+}} \alpha(\{1\})] \\
\widehat{\tau}\,(S)(q_8) &= \bigsqcup \left\{ \widehat{\sigma} \sqsubseteq S(q_5) \mid \widehat{\sigma}\,(j) \mathrel{\widehat{\leq}} \widehat{\sigma}\,(i) \sqsubseteq \texttt{false} \right\} \\
\widehat{\tau}\,(S)(q_9) &= S(q_8)[i \mapsto S(q_8)(i) \mathbin{\widehat{+}} \alpha(\{2\})] \\
\widehat{\tau}\,(S)(q_{10}) &= \bigsqcup \left\{ \widehat{\sigma} \sqsubseteq S(q_2) \mid \widehat{\sigma}\,(i) \mathrel{\widehat{\leq}} \alpha(\{100\}) \sqsubseteq \texttt{false} \right\}
\end{aligned}
$$

Note that the definition of $\widehat{\tau}\,(S)(q_5)$ uses restricted widening in the loop invariant i. Using Jacobi-iteration over the interval domain results in Table 4.1. The third iteration is a narrowing pass on the final result (i.e., replacing all widenings with narrowings). As can be seen, the use of restricted widening is crucial to find correct finite bounds on both loop counters. If ordinary widening was used in the definition of $\widehat{\tau}\,(S)(q_5)$, the widening would have resulted in $i \mapsto [1, \infty]$ in the second iteration of row 5. This in turn would have prevented the pruning of the conditional in $\widehat{\tau}\,(S)(q_6)$, which would have yielded the abstract state $i \mapsto [1, \infty], j \mapsto [1, \infty]$ instead of $i \mapsto [1, 100], j \mapsto [1, 100]$. This illustrates the usefulness of the restricted widening.

Table 4.2 displays the result of performing abstract interpretation using the congruence domain, and finally, in table 4.3, the reduced product of these two domains is displayed. Note the synergy of the two domains makes the intervals tighter in Table 4.3 compared to using the interval domain in isolation as in Table 4.1.

∇	iteration 1	iteration 2
$\widehat{\tau}^{1,2}(\bot)(0)$	$i \mapsto [-\infty,\infty], j \mapsto [-\infty,\infty]$	$i \mapsto [-\infty,\infty], j \mapsto [-\infty,\infty]$
$\widehat{\tau}^{1,2}(\bot)(1)$	$i \mapsto [1,1], j \mapsto [-\infty,\infty]$	$i \mapsto [1,1], j \mapsto [-\infty,\infty]$
$\widehat{\tau}^{1,2}(\bot)(2)$	$i \mapsto [1,1], j \mapsto [-\infty,\infty]$	$i \mapsto [1,\infty], j \mapsto [-\infty,\infty]$
$\widehat{\tau}^{1,2}(\bot)(3)$	$i \mapsto [1,1], j \mapsto [-\infty,\infty]$	$i \mapsto [1,100], j \mapsto [-\infty,\infty]$
$\widehat{\tau}^{1,2}(\bot)(4)$	$i \mapsto [1,1], j \mapsto [1,1]$	$i \mapsto [1,100], j \mapsto [1,1]$
$\widehat{\tau}^{1,2}(\bot)(5)$	$i \mapsto [1,1], j \mapsto [1,1]$	$i \mapsto [1,100], j \mapsto [1,\infty]$
$\widehat{\tau}^{1,2}(\bot)(6)$	$i \mapsto [1,1], j \mapsto [1,1]$	$i \mapsto [1,100], j \mapsto [1,100]$
$\widehat{\tau}^{1,2}(\bot)(7)$	$i \mapsto [1,1], j \mapsto [2,2]$	$i \mapsto [1,100], j \mapsto [2,101]$
$\widehat{\tau}^{1,2}(\bot)(8)$	$i \mapsto [1,1], j \mapsto [2,2]$	$i \mapsto [1,100], j \mapsto [2,\infty]$
$\widehat{\tau}^{1,2}(\bot)(9)$	$i \mapsto [3,3], j \mapsto [2,2]$	$i \mapsto [3,102], j \mapsto [2,\infty]$
$\widehat{\tau}^{1,2}(\bot)(10)$	$i \mapsto \bot, j \mapsto \bot$	$i \mapsto [101,\infty], j \mapsto [-\infty,\infty]$
Δ	iteration 3	
$\widehat{\tau}^{3}(\bot)(0)$	$i \mapsto [-\infty,\infty], j \mapsto [-\infty,\infty]$	
$\widehat{\tau}^{3}(\bot)(1)$	$i \mapsto [1,1], j \mapsto [-\infty,\infty]$	
$\widehat{\tau}^{3}(\bot)(2)$	$i \mapsto [1,102], j \mapsto [-\infty,\infty]$	
$\widehat{\tau}^{3}(\bot)(3)$	$i \mapsto [1,100], j \mapsto [-\infty,\infty]$	
$\widehat{\tau}^{3}(\bot)(4)$	$i \mapsto [1,100], j \mapsto [1,1]$	
$\widehat{\tau}^{3}(\bot)(5)$	$i \mapsto [1,100], j \mapsto [1,101]$	
$\widehat{\tau}^{3}(\bot)(6)$	$i \mapsto [1,100], j \mapsto [1,100]$	
$\widehat{\tau}^{3}(\bot)(7)$	$i \mapsto [1,100], j \mapsto [2,101]$	
$\widehat{\tau}^{3}(\bot)(8)$	$i \mapsto [1,100], j \mapsto [2,101]$	
$\widehat{\tau}^{3}(\bot)(9)$	$i \mapsto [3,102], j \mapsto [2,101]$	
$\widehat{\tau}^{3}(\bot)(10)$	$i \mapsto [101,102], j \mapsto [-\infty,\infty]$	

Table 4.1: Jacobi-iteration using the interval domain

	iteration 1	iteration 2
$\widehat{\tau}^{1,2}(\bot)(0)$	$i \mapsto 0+1\mathbb{Z}, j \mapsto 0+1\mathbb{Z}$	$i \mapsto 0+1\mathbb{Z}, j \mapsto 0+1\mathbb{Z}$
$\widehat{\tau}^{1,2}(\bot)(1)$	$i \mapsto 1+0\mathbb{Z}, j \mapsto 0+1\mathbb{Z}$	$i \mapsto 1+0\mathbb{Z}, j \mapsto 0+1\mathbb{Z}$
$\widehat{\tau}^{1,2}(\bot)(2)$	$i \mapsto 1+0\mathbb{Z}, j \mapsto 0+1\mathbb{Z}$	$i \mapsto 1+2\mathbb{Z}, j \mapsto 0+1\mathbb{Z}$
$\widehat{\tau}^{1,2}(\bot)(3)$	$i \mapsto 1+0\mathbb{Z}, j \mapsto 0+1\mathbb{Z}$	$i \mapsto 1+2\mathbb{Z}, j \mapsto 0+1\mathbb{Z}$
$\widehat{\tau}^{1,2}(\bot)(4)$	$i \mapsto 1+0\mathbb{Z}, j \mapsto 1+0\mathbb{Z}$	$i \mapsto 1+2\mathbb{Z}, j \mapsto 1+0\mathbb{Z}$
$\widehat{\tau}^{1,2}(\bot)(5)$	$i \mapsto 1+0\mathbb{Z}, j \mapsto 1+0\mathbb{Z}$	$i \mapsto 1+2\mathbb{Z}, j \mapsto 0+1\mathbb{Z}$
$\widehat{\tau}^{1,2}(\bot)(6)$	$i \mapsto 1+0\mathbb{Z}, j \mapsto 1+0\mathbb{Z}$	$i \mapsto 1+2\mathbb{Z}, j \mapsto 0+1\mathbb{Z}$
$\widehat{\tau}^{1,2}(\bot)(7)$	$i \mapsto 1+0\mathbb{Z}, j \mapsto 2+0\mathbb{Z}$	$i \mapsto 1+2\mathbb{Z}, j \mapsto 0+1\mathbb{Z}$
$\widehat{\tau}^{1,2}(\bot)(8)$	$i \mapsto 1+0\mathbb{Z}, j \mapsto 2+0\mathbb{Z}$	$i \mapsto 1+2\mathbb{Z}, j \mapsto 0+1\mathbb{Z}$
$\widehat{\tau}^{1,2}(\bot)(9)$	$i \mapsto 3+0\mathbb{Z}, j \mapsto 2+0\mathbb{Z}$	$i \mapsto 1+2\mathbb{Z}, j \mapsto 0+1\mathbb{Z}$
$\widehat{\tau}^{1,2}(\bot)(10)$	$i \mapsto \bot, j \mapsto \bot$	$i \mapsto 1+2\mathbb{Z}, j \mapsto 0+1\mathbb{Z}$

Table 4.2: Abstract interpretation with the congruence domain

∇	iteration 1
$\widehat{\tau}^{\,1}(\bot)(0)$	$i \mapsto \langle[-\infty,\infty],0+\mathbb{Z}\rangle\,,j \mapsto \langle[-\infty,\infty],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,1}(\bot)(1)$	$i \mapsto \langle[1,1],1+0\mathbb{Z}\rangle\,,j \mapsto \langle[-\infty,\infty],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,1}(\bot)(2)$	$i \mapsto \langle[1,1],1+0\mathbb{Z}\rangle\,,j \mapsto \langle[-\infty,\infty],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,1}(\bot)(3)$	$i \mapsto \langle[1,1],1+0\mathbb{Z}\rangle\,,j \mapsto \langle[-\infty,\infty],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,1}(\bot)(4)$	$i \mapsto \langle[1,1],1+0\mathbb{Z}\rangle\,,j \mapsto \langle[1,1],1+0\mathbb{Z}\rangle$
$\widehat{\tau}^{\,1}(\bot)(5)$	$i \mapsto \langle[1,1],1+0\mathbb{Z}\rangle\,,j \mapsto \langle[1,1],1+0\mathbb{Z}\rangle$
$\widehat{\tau}^{\,1}(\bot)(6)$	$i \mapsto \langle[1,1],1+0\mathbb{Z}\rangle\,,j \mapsto \langle[1,1],1+0\mathbb{Z}\rangle$
$\widehat{\tau}^{\,1}(\bot)(7)$	$i \mapsto \langle[1,1],1+0\mathbb{Z}\rangle\,,j \mapsto \langle[2,2],2+0\mathbb{Z}\rangle$
$\widehat{\tau}^{\,1}(\bot)(8)$	$i \mapsto \langle[1,1],1+0\mathbb{Z}\rangle\,,j \mapsto \langle[2,2],2+0\mathbb{Z}\rangle$
$\widehat{\tau}^{\,1}(\bot)(9)$	$i \mapsto \langle[3,3],3+0\mathbb{Z}\rangle\,,j \mapsto \langle[2,2],2+0\mathbb{Z}\rangle$
$\widehat{\tau}^{\,1}(\bot)(10)$	$i \mapsto \bot, j \mapsto \bot$
∇	iteration 2
$\widehat{\tau}^{\,2}(\bot)(0)$	$i \mapsto \langle[-\infty,\infty],0+\mathbb{Z}\rangle\,,j \mapsto \langle[-\infty,\infty],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,2}(\bot)(1)$	$i \mapsto \langle[1,1],1+0\mathbb{Z}\rangle\,,j \mapsto \langle[-\infty,\infty],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,2}(\bot)(2)$	$i \mapsto \langle[1,\infty],1+2\mathbb{Z}\rangle\,,j \mapsto \langle[-\infty,\infty],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,2}(\bot)(3)$	$i \mapsto \langle[1,99],1+2\mathbb{Z}\rangle\,,j \mapsto \langle[-\infty,\infty],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,2}(\bot)(4)$	$i \mapsto \langle[1,99],1+2\mathbb{Z}\rangle\,,j \mapsto \langle[1,1],1+0\mathbb{Z}\rangle$
$\widehat{\tau}^{\,2}(\bot)(5)$	$i \mapsto \langle[1,99],1+2\mathbb{Z}\rangle\,,j \mapsto \langle[1,\infty],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,2}(\bot)(6)$	$i \mapsto \langle[1,99],1+2\mathbb{Z}\rangle\,,j \mapsto \langle[1,99],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,2}(\bot)(7)$	$i \mapsto \langle[1,99],1+2\mathbb{Z}\rangle\,,j \mapsto \langle[2,100],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,2}(\bot)(8)$	$i \mapsto \langle[1,99],1+2\mathbb{Z}\rangle\,,j \mapsto \langle[2,\infty],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,2}(\bot)(9)$	$i \mapsto \langle[3,101],1+2\mathbb{Z}\rangle\,,j \mapsto \langle[2,\infty],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,2}(\bot)(10)$	$i \mapsto \langle[101,\infty],1+2\mathbb{Z}\rangle\,,j \mapsto \langle[-\infty,\infty],0+1\mathbb{Z}\rangle$
Δ	iteration 3
$\widehat{\tau}^{\,3}(\bot)(0)$	$i \mapsto \langle[-\infty,\infty],0+\mathbb{Z}\rangle\,,j \mapsto \langle[-\infty,\infty],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,3}(\bot)(1)$	$i \mapsto \langle[1,1],1+0\mathbb{Z}\rangle\,,j \mapsto \langle[-\infty,\infty],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,3}(\bot)(2)$	$i \mapsto \langle[1,101],1+2\mathbb{Z}\rangle\,,j \mapsto \langle[-\infty,\infty],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,3}(\bot)(3)$	$i \mapsto \langle[1,99],1+2\mathbb{Z}\rangle\,,j \mapsto \langle[-\infty,\infty],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,3}(\bot)(4)$	$i \mapsto \langle[1,99],1+2\mathbb{Z}\rangle\,,j \mapsto \langle[1,1],1+0\mathbb{Z}\rangle$
$\widehat{\tau}^{\,3}(\bot)(5)$	$i \mapsto \langle[1,99],1+2\mathbb{Z}\rangle\,,j \mapsto \langle[1,100],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,3}(\bot)(6)$	$i \mapsto \langle[1,99],1+2\mathbb{Z}\rangle\,,j \mapsto \langle[1,99],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,3}(\bot)(7)$	$i \mapsto \langle[1,99],1+2\mathbb{Z}\rangle\,,j \mapsto \langle[2,100],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,3}(\bot)(8)$	$i \mapsto \langle[1,99],1+2\mathbb{Z}\rangle\,,j \mapsto \langle[2,100],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,3}(\bot)(9)$	$i \mapsto \langle[3,101],1+2\mathbb{Z}\rangle\,,j \mapsto \langle[2,100],0+1\mathbb{Z}\rangle$
$\widehat{\tau}^{\,3}(\bot)(10)$	$i \mapsto \langle[101,101],101+0\mathbb{Z}\rangle\,,j \mapsto \langle[-\infty,\infty],0+1\mathbb{Z}\rangle$

Table 4.3: The reduced product of the interval and congruence domain

4.6 Counting Elements in Abstract Environments

Having obtained an abstract environment for each program point in the program, we are now ready to extract loop bounds using this information. A safe upper bound of the number of times a loop can iterate can be extracted from an abstract state corresponding to a program point which is visited in each iteration. Since an abstract state safely approximates the set of possible concrete states at that program point, the ”size” of that abstract environment is an upper bound of the number of loop iterations. Let $\langle L, \alpha, \gamma, M\rangle$ be a Galois-connection, then we define the *size* of an abstract value $a \in L$ as follows:

$$|a| = |\gamma(a)| \tag{4.1}$$

The size of an abstract value in the interval domain is therefore

$$\begin{aligned}
&|\bot| = |\gamma(\bot)| = |\varnothing| = 0\\
&|[a,b]| = |\gamma([a,b])| = |\{n \in \mathbb{Z} \mid a \le n \le b\}| = b - a + 1\\
&|\top| = |\gamma(\top)| = |\mathbb{Z}| = \infty
\end{aligned}$$

for the congruence domain the sizes are

$$\begin{aligned}
&|\bot| = |\varnothing| = 0\\
&|m + k\mathbb{Z}| = \begin{cases} |\{m\}| = 1 & \text{if } k = 0\\ |\{m + kn \mid n \in \mathbb{Z}\}| = \infty & \text{otherwise}\end{cases}
\end{aligned}$$

Finally, the size of a value in the reduced product of the interval and congruence domain is

$$\begin{aligned}
|\bot| &= |\varnothing| = 0\\
|\langle [a,b], m + k\mathbb{Z}\rangle| &= |\{m + kn \mid n \in \mathbb{Z} \wedge a \le m + kn \le b\}|\\
&= \left\lceil \frac{b - a + 1}{k} \right\rceil\\
|\top| &= |\mathbb{Z}| = \infty
\end{aligned}$$

The size of an abstract environment is an upper bound of the execution count of its associated program point and thus a loop bound. The size of an environment can be derived by the following formula:

$$|\,\widehat{\sigma}\,| = \prod_{v \in \mathcal{V}} |\,\widehat{\sigma}\,(v)|\;.$$

This corresponds to the actual number of possible environments associated with a program point. To bind a loop, we look at the element count of a *loop representative*, i.e., a program point which is guaranteed to execute exactly once in every loop iteration. To avoid over-estimations, the minimum count of all possible loop representatives should be chosen [1].

4.6.1 Example of Loop Bounding with Intervals

We illustrate the loop bounding process by considering L_1 and L_2 from TP. The set of possible representatives of L_1 is $\{q_3\}$, note that q_2 should *not* be included in this set since q_2 may also execute outside the loop. The set of representatives for L_2 is $\{q_6, q_7\}$, and by similar reasoning, q_5 should not be included in this set. First we compute the upper bound of L_1 using the interval domain.

$$\begin{aligned} q_3 &= |[i \mapsto [1,100], j \mapsto [-\infty,\infty]]| \\ &= |\gamma([1,100])| \text{ (since } j \text{ is disregarded)} \\ &= |\{x \mid 1 \le x \le 100\}| = 100 \end{aligned}$$

By slicing of L_1 and also by invariant analysis, j can be completely disregarded from the loop bound analysis here, resulting in a loop bound of 100.

To compute the bound for L_2 we proceed in a similar fashion, since $\mathcal{I}_{L_2} = \{i\}$, we can disregard i completely for the purposes of element counting. This is because i remains the same during the execution of L_2. Thus, we have

$$\begin{aligned} q_6 &= |[i \mapsto [1,100], j \mapsto [1,100]]| \\ &= |\gamma([1,100])| \text{ (since } i \text{ is disregarded)} \\ &= 100 \end{aligned}$$

$$\begin{aligned} q_7 &= |[i \mapsto [1,100], j \mapsto [2,101]]| \\ &= |\gamma([1,100])| \\ &= 100 \end{aligned}$$

The minimum of these are 100, which is the derived loop bound for L_2.

[1] In [ESG$^+$07] it is suggested to simply choose one representative, which is clearly possible but might lead to over-approximations.

4.6.2 Example of Loop Bounding with Intervals and Congruences

To illustrate the differences between abstract domains, we will also calculate the loop bounds using other abstract domains. The congruence abstract domain is not useful in itself for computing loop bounds, since abstract values in most cases correspond to infinite concrete sets. However, by using it in conjunction with the interval domain via the reduced product, tighter loop bounds can be found. To illustrate this, we make the following computations based on the results of Table 4.3.

$$\begin{aligned} q_3 &= |[i \mapsto \langle [1,99], 1+2\mathbb{Z}\rangle, j \mapsto \langle [-\infty,\infty], 0+1\mathbb{Z}\rangle]| \\ &= |\gamma([1,99], 1+2\mathbb{Z})| = 50 \end{aligned}$$

Thus, we can safely bound L_1 to 50, which in this case is an exact bound. Also L_2 can be bound tighter, as suggested by the following:

$$\begin{aligned} q_6 &= |[i \mapsto \langle [1,99], 1+2\mathbb{Z}\rangle, j \mapsto \langle [1,99], 0+1\mathbb{Z}\rangle]| \\ &= |\gamma(\langle [1,99], 0+1\mathbb{Z}\rangle)| \text{ (since } i \text{ is disregarded)} \\ &= 99 \end{aligned}$$

$$\begin{aligned} q_7 &= |[i \mapsto \langle [3,101], 1+2\mathbb{Z}\rangle, j \mapsto \langle [2,100], 0+1\mathbb{Z}\rangle]| \\ &= |\gamma(\langle [2,100], 0+1\mathbb{Z}\rangle)| \\ &= 99 \end{aligned}$$

So, L_2 can get a slightly more precise upper bound of 99.

4.6.3 Limitation of Non-Relational Domains

The loop bounds derived using the technology outlined in previous sections derives local loop bounds by the use of loop invariant variables. However, an abstract state derived for a program point can also be used to determine an upper bound of the total number of times that that program point can be visited. As an example, counting the number of elements for program point q_5

gives, when using the reduced product, an upper bound of $49 \cdot 100 = 4900$. While this is a safe upper bound, it is not very precise. This is because the abstract interpretation does not take the fact that the loop bound of L_2 changes depending on the value of i at that point. In other words, the loop counter j of L_2 is dependent on the value of i. A non-relational abstract domain fails to take advantage of this fact, which leads to an unavoidable over-approximation. The solution to this is to use a relational abstract domain which tracks some of the relations between variables, and thus can be able to detect these kind of dependencies. A relational abstract domain is more expensive to use in analysis but can capture some of these essential dependencies. The loop bound analysis in [LCFM09] uses the polyhedral abstract domain for this. The use of relational abstract domains will be more thoroughly examined in Chapter 6.

4.7 Evaluation

An evaluation of the method outlined in this chapter was made in [ESG$^+$07]. In this publication, 28 benchmarks from the Mälardalen WCET benchmark suite (see [MDH09]) where analysed. The method was evaluated using the interval domain and the reduced product as described above. The analysis binds 63% of the loops in the benchmarks and 51% of them are bound exactly. For six of the loops, a tighter bound was found by using the reduced product of the interval and congruence domains compared to when using only the interval domain. Table 4.4, taken from [ESG$^+$07], shows the results from the evaluation. Here **#LC** is the lines of code, **#L** is the number of loops, **#B** is the number of loops bound by the analysis **%B** is the percentage of loops bound, **#E** is the number of loops bound exactly, **%E** is the percentage of loops bound exactly and finally, **Time** is the execution time of the analyis (implemented in SWEET [WCE09]) running on a 3 GHz PC running Linux.

Program	**#LC**	**#L**	**#B**	**%B**	**#E**	**%E**	**Time**
`adpcm`	879	27	18	67%	8	30%	48.6
`bs`	114	1	0	0%	0	0%	0.81
`cnt`	267	4	4	100%	4	100%	0.24
`cover`	640	3	3	100%	3	100%	0.32
`crc`	128	6	6	100%	6	100%	0.11
`duff`	86	2	1	50%	1	50%	0.04
`edn`	285	12	12	100%	9	75%	0.71
`expint`	157	3	3	100%	3	100%	0.04
`fac`	21	1	1	100%	1	100%	0.01
`fdct`	239	2	2	100%	2	100%	0.05
`fft1`	219	30	7	23%	3	10%	5.39
`fibcall`	72	1	1	100%	1	100%	0.01
`fir`	276	2	2	100%	1	50%	0.38
`inssort`	92	2	1	50%	1	50%	0.54
`jcomplex`	64	2	0	0%	0	0%	0.04
`jfdctint`	375	3	3	100%	3	100%	0.06
`lcdnum`	64	1	1	100%	1	100%	0.01
`ludcmp`	147	11	6	55%	5	45%	247.6
`matmult`	163	7	7	100%	7	100%	0.51
`ndes`	231	12	12	100%	12	100%	3.11
`ns`	535	4	1	25%	1	25%	91.9
`nsichneu`	4253	1	1	100%	1	100%	1.11
`prime`	535	2	0	0%	0	0%	0.05
`qsort-exam`	121	6	0	0%	0	0%	76.4
`qurt`	166	3	1	33%	1	33%	0.09
`select`	114	4	0	0%	0	0%	19.6
`statemate`	1276	1	0	0%	0	0%	1.00
`ud`	161	11	11	100%	10	91%	0.53
Total	-	164	104	63%	84	51%	-

Table 4.4: Benchmark programs and result of loop bound analysis (taken from [ESG$^+$07])

Chapter 5

The Congruence Domain

5.1 Background

This chapter investigates the congruence domain invented by P. Granger, presented in [Gra89]. The congruence domain was implemented into the static WCET analysis tool SWEET [WCE09] to produce tighter loop bounds (see [ESG$^+$07]) and as a complement to abstract execution [GESL07]. That version of SWEET operated on an intermediate level language called NIC (New Intermediate Code). SWEET uses an internal compiler developed by a research group at Uppsala University which compiles C code into NIC. In the compilation some type information is lost, such as if an integer is signed or unsigned. In addition, the intermediate format commonly uses lower level operations such as bit-shifting and logical bit-operations. This chapter presents necessary developments of the congruence domain in order to able to perform a tight and safe analysis on intermediate or low-level code.

5.2 Analysis on Low-Level and Intermediate-Level Code

The theory of abstract interpretation is, usually in the literature, formulated over abstract representation of programs (like in this thesis), and the abstract domains are based on this abstract model of programs. To avoid special cases and language dependencies, the domains, including their abstract operations are usually given over basic mathematical sets of numbers, like $\mathbb{Z}, \mathbb{N}, \mathbb{Q}$ or $\mathbb{R}$.

While maintaining clarity and language independence, the link to actual program code is getting lost. When analysing real code on an intermediate or object code level, the structures looks highly different. First of all, numbers can not be arbitrary large as they are usually represented as bit strings. In addition, the operations performed on program variables are not restricted to common arithmetical operations; bit-string operations such as shifting and logical operations are often applied. Abstract domains as found in literature usually do not take these things into consideration. This requires extra research to make the domains practically usable when analysis is performed over low or intermediate level code.

In this chapter, we shall focus on developments of the congruence domain (see Chapter 3). Related work to this section is the work of Müller and Olm [MOS07, MOS05] which investigates the congruence domain over a more realistic concrete domain. Another related work with the usage of the reduced product of the congruence domain and the interval domain combined is presented in [RBL06], where the developments requires the two domains in conjunction, while our work is provided for the congruence domain in isolation.

5.2.1 Assumptions

To motivate the developments of this chapter, we shall make a few assumptions about the system for which we want to apply the analysis. These assumptions are safe for most modern systems, independent of source language or platform.

- Integers are represented by a fixed number of bits n (usually $16, 32$ or 64). When an operation results in a larger or smaller number than can which be represented by n bits, either a run-time error is caused or the number is wrapped-around[1].
- An integer represented by a string of n bits can be interpreted as *signed* (in the range $[-2^{n-1}, 2^{n-1} - 1]$) or *unsigned* (in the range $[0, 2^n - 1]$)
- Integers may be computed as a result from and/or as arguments to functions over bit-strings such as shifting or bitwise logical operations

These assumptions have an impact on the formulation of analyses in order for them to be correct and precise. The following sections develop the congruence domain using the above assumptions as basis.

[1]That is, considered modulus 2^n.

5.2.2 Two's Complement

The most common way to represent negative numbers in binary format is the so-called *two's complement*. The two's complement of a binary number B is obtained by first reversing all bits (i.e., performing a logical NOT) and then adding 1. For example, the two's complement of the binary number 0101 is 1011. Negative numbers can then be recognised by the most significant bit; a zero as least significant bit means a positive number, and a one implies a negative number. Thus, 0101 would be interpreted as 5 and 1011 would be interpreted as the negation of the two's complement, that is, $-0110 = -5$. If the two's complement is used to represent negative numbers, then an integer is called *signed*, and if it is not (i.e., if 1011 would be interpreted normally, as 11), then the integer is called *unsigned*.

5.3 The Congruence Domain

The abstract domain of arithmetical congruences was proposed by Philippe Granger in [Gra89]. The domain approximates a set of integers as a residue class, i.e., as a class of integers which are equal to n modulus m. This domain was used in the developments of Chapter 4 to obtain tighter loop bounds. The domain as proposed in [Gra89] has several problems with the assumptions outlined in Section 5.2.1:

- The domain is presented as an abstraction of $\mathcal{P}(\mathbb{Z})$, while we consider integer valued variables which are represented by a fixed number of bits.
- The domain does not take ambiguous interpretation of integers into consideration (such as signed/unsigned).
- In the original presentation, abstract operations were limited to common arithmetic operations such as addition, subtraction, multiplication, division and modulus. In lower level code there are more operations that need to be taken into consideration such as bitwise logical operators, bit-shifting etc.

In this chapter we will enhance the congruence domain by adding support to use it in intermediate or low-level code. This will be done by adopting an abstraction of the lattice $\mathcal{P}(\{0, 1\}^n)$ rather than $\mathcal{P}(\mathbb{Z})$, and developing lower-level abstract operations for the congruence domain. The definition of the congruence domain and its lattice operations are shown on page 45 in

Chapter 3. The lattice of arithmetical congruences is referred to as $C(\mathbb{Z})$ in the following. For completeness, we here present the definition of some arithmetic abstract operations as presented in [Gra89]. We will use the convention that $\gcd(0,m) = m, \mathrm{lcm}(0,m) = 0, \mathrm{lcm}(1,m) = 1, \gcd(m,n) = \gcd(|m|,|n|)$ and $\mathrm{lcm}(m,n) = \mathrm{lcm}(|m|,|n|)$. Let $m_0 + k_0\mathbb{Z}$ and $m_1 + k_1\mathbb{Z}$ be two non-bottom abstract values of the congruence domain, then

$$\begin{aligned}
(m_0 + k_0\mathbb{Z}) \mathbin{\hat{\pm}} (m_1 + k_1\mathbb{Z}) &\overset{\text{def}}{=} m_0 \pm m_1 + \gcd(k_0, k_1)\mathbb{Z} \\
(m_0 + k_0\mathbb{Z}) \mathbin{\hat{*}} (m_1 + k_1\mathbb{Z}) &\overset{\text{def}}{=} m_0 m_1 + \gcd\{m_0 k_1, m_1 k_0, k_0 k_1\}\mathbb{Z} \\
(m_0 + k_0\mathbb{Z})_* \mathbin{\widehat{\mathrm{mod}}} (m_1 + k_1\mathbb{Z})_+ &\overset{\text{def}}{=} m_0 + \gcd\{k_0, m_1, k_1\}\mathbb{Z} \\
(m_0 + k_0\mathbb{Z})_* \mathbin{\widehat{\mathrm{div}}} (m_1 + k_1\mathbb{Z})_+ &\overset{\text{def}}{=} 0 + 1\mathbb{Z}
\end{aligned}$$

where mod is the modulus operator, div is integer division and $S_* \overset{\text{def}}{=} S \cap \mathbb{N}$ and $S_+ \overset{\text{def}}{=} S \cap \mathbb{Z}_+$ for all $S \subseteq \mathbb{Z}$. The operation $\hat{\pm}$ denotes the abstract versions of the operations $+$ and $-$. Sometimes, when one operand is a singleton set, it is possible to give a more precise result.

Let $m + k\mathbb{Z}$ be an abstract value, $a + 0\mathbb{Z}$ be a singleton abstract value, and let $N = k((a - m) \mathrm{~div~} k)) + m$. Then,

$$\begin{aligned}
&(m + k\mathbb{Z})_* \mathbin{\widehat{\mathrm{div}}} a + 0\mathbb{Z} \\
&\overset{\text{def}}{=} \begin{cases} m \mathrm{~div~} a + (k \mathrm{~div~} a)\mathbb{Z} & \text{if } a|k \\ 0 + 1\mathbb{Z} & \text{otherwise} \end{cases} \\
&a + 0\mathbb{Z} \mathbin{\widehat{\mathrm{div}}} (m + k\mathbb{Z})_+ \\
&\overset{\text{def}}{=} \begin{cases} 0 + (a \mathrm{~div~} N)\mathbb{Z} & \text{if } N > 0 \\ 0 + 0\mathbb{Z} & \text{otherwise} \end{cases} \\
&(m + k\mathbb{Z}) \mathbin{\widehat{\mathrm{mod}}} a + 0\mathbb{Z} \\
&\overset{\text{def}}{=} \begin{cases} m \mathrm{~mod~} a + 0\mathbb{Z} & \text{if } a|k \\ m + \gcd(k, a)\mathbb{Z} & \text{otherwise} \end{cases} \\
&a + 0\mathbb{Z} \mathbin{\widehat{\mathrm{mod}}} (m + k\mathbb{Z})_+ \\
&\overset{\text{def}}{=} \begin{cases} a + 0\mathbb{Z} & \text{if } N \leq 0 \\ a + \gcd(m, k)\mathbb{Z} & \text{if } a \mathrm{~div~} N = 1 \\ a + N(a \mathrm{~div~} N)\mathbb{Z} & \text{if } a \mathrm{~div~} N \geq 2 \end{cases}
\end{aligned}$$

5.4 Integer Representation

As mentioned in Section 5.2.1, the concrete domain $\mathcal{P}(\mathbb{Z})$ is not correct with respect to the outlined assumptions, since integers are limited by the reserved memory to represent them and the fact that ”overflows” result in errors or wrap-around effects. Müller and Olm [MOS07, MOS05] have suggested to use an abstraction of the domain $\mathcal{P}(\mathbb{Z}/n\mathbb{Z})$ to remedy the situation. The set $\mathbb{Z}/2^n\mathbb{Z}$ contains all co-sets of the Abelian group $\mathbb{Z}$. That is, all integers are considered modulo 2^n. If n is the number of bits used in the system, the elements of $\mathbb{Z}/n\mathbb{Z}$ has the nice property of simulating wrap-around effects. For instance, in a 32-bit system, elements would represent equivalence classes such as $5 + 2^{32}\mathbb{Z}$, meaning that any operation on this set would still be correct even if out-of-bounds. The drawback of this approach is that the class of congruences detected are limited to a power of two. The reason for this is that most lattice computations involve a computation of the greatest common divisor, and since all classes are powers of two, the domain can only preserve the greatest common divisor when it is a power of two.

Before analysing congruence invariants on low-level code we have to make a decision. We may:

1. Use the less precise analysis over the abstract domain $C(\mathbb{Z}/2^n\mathbb{Z})$. This amounts to deriving invariants of the type $x \in b + 2^k\mathbb{Z}$, which is the same as knowing the k least significant bits of a variable.

2. Rely on the assumption that no overflows (or underflows) may occur. While this in theory could yield unsound results, it is a reasonable assumption to do if the user of the analysis by other means can be sure that no overflow (underflow) can occur in the program to be analysed.

3. We could use the analysis in conjunction with another one to see where possible overflow/underflows can occur. For instance, the reduced product of the congruence domain and an interval domain designed for finite domains can find program points in which overflows/underflows are impossible and for other program points consider the abstract values modulo 2^n.

We will in this thesis use the second item approach, since it gives the most precise result. From now on we will assume that no overflows or underflows are present in the program we wish to analyse. Our suggestion is to use $B^n = \mathcal{P}(\{0,1\}^n)$ as concrete domain, and abstract the congruence domain via functions which interpret bit-strings as integers. Note that this approach is

$$\begin{tikzcd}
 & \mathbb{Z}^n_{\mathrm{S}} \arrow[dr, bend left, "\subseteq"] & \\
B^n \arrow[ur, bend left, "\theta^n_{\mathrm{S}}"] \arrow[dr, bend right, "\theta^n_{\mathrm{U}}"'] & & \mathbb{Z} \arrow[ll, "\phi^n_{\mathrm{S}}"', "\phi^n_{\mathrm{U}}"] \\
 & \mathbb{Z}^n_{\mathrm{U}} \arrow[ur, bend right, "\subseteq"'] &
\end{tikzcd}$$

Figure 5.1: Relation between bit-representations and integers.

actually general and could potentially be applied to other abstract domains to solve similar problems.

5.4.1 Signed and Unsigned Integers

In this section we introduce three Galois connections, all using the set of bit-strings B^n as basis and the lattice of arithmetical congruences $C(\mathbb{Z})$ as abstract domain. The first Galois connection $C_{\mathrm{U}} = \langle \mathcal{P}(B^n), \alpha_{\mathrm{U}}, \gamma_{\mathrm{U}}, C(\mathbb{Z})\rangle$ is used for unsigned integers, the second Galois connection $C_{\mathrm{S}} = \langle \mathcal{P}(B^n), \alpha_{\mathrm{S}}, \gamma_{\mathrm{S}}, C(\mathbb{Z})\rangle$ is used for signed integers, and finally $C_* = \langle \mathcal{P}(B^n), \alpha_*, \gamma_*, C(\mathbb{Z})\rangle$, is a Galois-connection which is safe to use independently of the interpretation of the bit-strings. The definitions of these abstractions are all based on the original domain $C_{\mathbb{Z}} = \langle \mathcal{P}(\mathbb{Z}), \alpha_{\mathbb{Z}}, \gamma_{\mathbb{Z}}, C(\mathbb{Z})\rangle$ presented in [Gra89]. To define these properly, we need to specify formally how to interpret bit-strings.

Definition 19. *Let B^n be the set of all bit-strings of length n. Then we define two interpretation functions $\theta^n_{\mathrm{S}} : B^n \to \mathbb{Z}^n_{\mathrm{S}}$ and $\theta^n_{\mathrm{U}} : B^n \to \mathbb{Z}^n_{\mathrm{U}}$ where $\mathbb{Z}^n_{\mathrm{S}} = \{x \in \mathbb{Z} | -2^{n-1} \leq x \leq 2^{n-1}-1\}$ and $\mathbb{Z}^n_{\mathrm{U}} = \{x \in \mathbb{Z} | \, 0 \leq x \leq 2^n - 1\}$. The function θ^n_{S} is the signed interpretation of a bit-string and θ^n_{U} is the regular (unsigned) interpretation. Also we define the functions $\phi^n_{\mathrm{S}}, \phi^n_{\mathrm{U}} : \mathbb{Z} \to B^n$ which map integers (modulus 2^n) to their respective signed and unsigned bit-string representations.*

Figure 5.1 shows how the interpretation functions and the representations are related in a diagram. To get a set of integers that is safe no matter how we interpret the strings we introduce the function $\theta : \mathcal{P}(B^n) \to \mathcal{P}(\mathbb{Z})$ as

$$\theta^n_* = \lambda B.\theta^n_{\mathrm{S}}(B) \cup \theta^n_{\mathrm{U}}(B).$$

Suppose that we represent integers by 4 bits and that we want to find an abstract value corresponding to the set $\{1110, 0110\}$. This set can be interpreted as $\{-2, 6\}$ or $\{6, 16\}$ depending on if we use signed or unsigned integers (i.e., depending on which version of θ we use). Therefore we define the abstraction functions as follows

$$\begin{aligned} \alpha_S^n &= \alpha_{\mathbb{Z}} \circ \theta_S^n \\ \alpha_U^n &= \alpha_{\mathbb{Z}} \circ \theta_U^n \\ \alpha_*^n &= \alpha_{\mathbb{Z}} \circ \theta_*^n \end{aligned}$$

The last function α_*^n is used as a safe approximation no matter how we interpret the bit-strings, i.e., it is safe to use for both signed and unsigned integers. The concretisation functions are defined as:

$$\begin{aligned} \gamma_S^n &= \phi_S^n \circ \gamma_{\mathbb{Z}} \\ \gamma_U^n &= \phi_U^n \circ \gamma_{\mathbb{Z}} \\ \gamma_*^n &= \phi_*^n \circ \gamma_{\mathbb{Z}} \end{aligned}$$

An example of the usage of the functions is given below:

$$\begin{aligned} \alpha_S^4(\{1110, 0110\}) &= \alpha_{\mathbb{Z}} \circ \theta_S^4(\{1110, 0110\}) \\ &= \alpha_{\mathbb{Z}}(\{-2, 6\}) \\ &= -2 + 0\mathbb{Z} \sqcup 6 + 0\mathbb{Z} \\ &= 6 + 8\mathbb{Z} \end{aligned}$$

$$\begin{aligned} \alpha_U^4(\{1110, 0110\}) &= \alpha_{\mathbb{Z}} \circ \theta_U^4(\{1110, 0110\}) \\ &= \alpha_{\mathbb{Z}}(\{16, 6\}) \\ &= 16 + 0\mathbb{Z} \sqcup 6 + 0\mathbb{Z} \\ &= 6 + 10\mathbb{Z} \end{aligned}$$

As can be seen, different abstract values are obtained depending on the interpretation of the bit strings. Using α_*^4 will obtain an abstract value which is valid for both interpretations.

$$\begin{tikzcd}
\mathcal{P}(B^n) \arrow[r, bend left, "\theta_S^n"] \arrow[r, bend right, "\theta_U^n"'] \arrow[dr, bend right, "\alpha_S^n"'] \arrow[dr, bend left, "\alpha_U^n"] & \mathcal{P}(\mathbb{Z}) \arrow[d, bend left, "\alpha"] \\
& C(\mathbb{Z}) \arrow[u, bend left, "\gamma"]
\end{tikzcd}$$

Figure 5.2: Relation between bit-representation and the abstract domain.

$$\begin{aligned}
\alpha_*^4(\{1110, 0110\}) &= \alpha_{\mathbb{Z}} \circ \theta_*^4(\{1110, 0110\}) \\
&= \alpha_{\mathbb{Z}}(\theta_{\mathrm{S}}^4(\{1110, 0110\}) \cup \theta_{\mathrm{U}}^4(\{1110, 0110\})) \\
&= \alpha_{\mathbb{Z}}(\{-2, 6\} \cup \{14, 6\}) \\
&= \alpha_{\mathbb{Z}}(\{-2, 6, 14\}) \\
&= -2 + 0\mathbb{Z} \sqcup 6 + 0\mathbb{Z} \sqcup 14 + 0\mathbb{Z} \\
&= 6 + 8\mathbb{Z} \sqcup 14 + 0\mathbb{Z} \\
&= 6 + 8\mathbb{Z}
\end{aligned}$$

The relation between the different abstractions are depicted in Figure 5.2. To summarise, we replace the concrete domain $\mathcal{P}(\mathbb{Z})$ by $\mathcal{P}(B^n)$. This makes analysis on intermediate or low-level possible because:

1. The analysis is safe for both signed and unsigned integers, it is also possible to make analysis safe when the interpretation is unknown[2].

2. The analysis can handle overflows since the ϕ^n interpretation of an abstract value only consider the n least significant bits of any integer.

3. The analysis does not assume that the number of integers is infinite.

[2]Using only this abstraction will lead to similar problems as in [MOS07, MOS05], since only modulus which are a power of two can be detected.

5.5 Abstract Bit-Operations

In [Gra89] abstract operations for addition, subtraction, multiplication, division and modulus is presented. This section will in addition provide abstract definitions of the bitwise operations AND, OR and XOR, as well as abstract versions of bit-shifting (left and right). Bit operations are used even in high-level languages as C, whereas bit-shifting operations and truncation are usually found in intermediate representations or object code. Since we are dealing with three different Galois connections $C_{\mathrm{U}}, C_{\mathrm{S}}$ and C_*, we have to provide different abstract functions for them. For an abstract function $\widehat{f}$ we shall use the notations $\widehat{f_{\mathrm{U}}}$, $\widehat{f_{\mathrm{S}}}$ and $\widehat{f_*}$ respectively.

5.5.1 Bitwise NOT

The bitwise NOT operation takes a set of bit-strings and returns a set of bit-strings where all zeroes are replaced by ones, and vice versa. As example $\mathrm{NOT}(\{0110, 1101\}) = \{1001, 0010\}$. If the bit-strings are interpreted as integers, we see that

$$\theta_{\mathrm{S}} \circ \mathrm{NOT}(B) = \{-b - 1 \mid b \in \theta_{\mathrm{S}}(B)\} \tag{5.1}$$

$$\theta_{\mathrm{U}} \circ \mathrm{NOT}(B) = \{2^n - b - 1 \mid b \in \theta_{\mathrm{U}}(B)\} \tag{5.2}$$

Equation (5.1) follows directly from the definition of two's complement, $-b - 1 = (\mathrm{NOT}(b) + 1) - 1 = \mathrm{NOT}\ b$. Bitwise NOT for unsigned integers is simply taking a string of n ones (a string of n ones is represented by the decimal number $2^n - 1$) and subtracting it with the original string, obtaining (5.2). These facts can be used as basis for the definition of the abstract version of bitwise NOT:

Definition 20. *Let $m + k\mathbb{Z}$ be a non-bottom abstract value, then we define*

$$\begin{array}{ll}
\widehat{\mathrm{NOT}}^n_{\mathrm{U}}(m + k\mathbb{Z}) & \overset{def}{=} 2^n - m - 1 + k\mathbb{Z} \\
\widehat{\mathrm{NOT}}^n_{\mathrm{S}}(m + k\mathbb{Z}) & \overset{def}{=} -m - 1 + k\mathbb{Z} \\
\widehat{\mathrm{NOT}}^n_{*}(m + k\mathbb{Z}) & \overset{def}{=} \widehat{\mathrm{NOT}}^n_{\mathrm{U}}(m + k\mathbb{Z}) \sqcup \widehat{\mathrm{NOT}}^n_{\mathrm{S}}(m + k\mathbb{Z}) \\
 & = -m - 1 + \gcd\{k, 2^n\}\mathbb{Z}.
\end{array}$$

Proposition 5. *The abstract operations in Definition 20 are correct approximations of abstract $\mathrm{NOT}_{\mathrm{S}}, \mathrm{NOT}_{\mathrm{U}}$ and NOT_*.*

The proof is trivial, it follows directly from (5.1) and (5.2).

5.5.2 Bitwise Binary Logical Operators

The most common binary bitwise logical operators are $\mathrm{AND}, \mathrm{OR}$ and XOR. These operations all take two bit-strings as arguments, performs the logical connectives bitwise on the two strings, and returns the result. While most such behaviour destroys congruence relations, there are still some cases where a modulus which is a power of two can be preserved. This section defines an abstract version of AND, from which the other two logical operations can be derived by identities (such as De Morgan's laws). The definition of the abstract AND requires some preliminaries. In the following, we shall use ϕ^n to denote either ϕ^n_{S} or ϕ^n_{U} when the result applies for both. Furthermore, if x and y are integers, we shall use the notation $x \,\mathrm{AND}_\phi\, y$ for the expression $\phi^n(x) \,\mathrm{AND}\, \phi^n(y)$.

Lemma 4. *Let $2^n\mathbb{Z} = \{2^n k \mid k \in \mathbb{Z}\}$ and let $\phi^n(2^n\mathbb{Z}) = \{\phi^n(k) \mid k \in 2^n\mathbb{Z}\}$. Then any for any bit-string $b \in \phi^n(2^n\mathbb{Z})$, the n least significant bits of b are zeroes.*

Proof. Take any element $k \in 2^n\mathbb{Z}$, then k can be re-written as $k = 2^n k'$ for some number k'. Multiplying a bit-string by two corresponds to shifting it left one step (both for signed and unsigned integers). Left shifting a bit-string introduces a zero as least significant bit, and since $k = 2^n k'$ the interpreted string $\phi^n(2^n k')$ corresponds to left shifting k', n steps, thus $\phi^n(2^n k') = \phi^n(k)$ has at least n zeroes as least significant bits. □

Definition 21. *Let A be a bit-string. Then we define $L(A)$ as the position of the least significant (rightmost) ”one” of A, and $M(A)$ as the most significant (leftmost) ”one” of A. The least significant bit of a string is considered to be position 0.*

As an example of this, consider the bit-string 1001. Then $L(1001) = 0$ since the least significant ”one” is on position 0. Also, $M(1001) = 3$ since the most significant ”one” is on position three. Furthermore, we have that $L(0110) = 1$ and $M(0110) = 2$.

Lemma 5. *Take the sets $2^k\mathbb{Z}$ and $2^{k'}\mathbb{Z}$ such that $k \geq k'$, then*

$$\theta(2^k\mathbb{Z} \ \mathrm{AND}_\phi \ 2^{k'}\mathbb{Z}) = 2^k\mathbb{Z}.$$

Proof. According to Lemma 4, the set $\phi^n(2^k\mathbb{Z})$ contains only bit-strings that have their k least significant bits as zeroes. The result of ”AND”-ing a zero with anything results again with a zero, so the number of trailing zeroes of k

will be in the result. Note that θ is not subscripted since the result holds for both signed and unsigned integers, in other words, θ can be replace with any of $\theta_{\mathrm{S}}, \theta_{\mathrm{U}}$ or θ_*. $\square$

Proposition 6. *Let $a\mathbb{Z}, b\mathbb{Z}$ be sets such that a and b are odd. Then*

$$\alpha^n(a\mathbb{Z} \ \mathrm{AND}_\phi \, b\mathbb{Z}) = 0 + 1\mathbb{Z}$$

The function α^n represents all of the functions $\alpha^n_{\mathrm{S}}, \alpha^n_{\mathrm{U}}$ and α^n_*.

We will prove the proposition using the following lemma.

Lemma 6. *Let $a\mathbb{Z}, b\mathbb{Z}$ be sets such that a and b are odd. Then*

$$\alpha^n(a\mathbb{Z} \ \mathrm{AND}_\phi \ b\mathbb{Z}) \sqsupseteq 2^k\mathbb{Z}$$

for some non-negative integer k.

Proof. Consider the element $a \in a\mathbb{Z}$. Let $L(\phi^n(a)) = k$ and consider the element $2^k b \in b\mathbb{Z}$ and note that if $2^k b > 2^n$ then $\phi^n(2^k b) = \phi^n(2^k b \bmod 2^n)$ which significantly means that the n least significant bits of $2^k b$ are remains the same. Now, $a \ \mathrm{AND}_\phi \ 2^k b = \phi^n(2^k)$ since position k will be the only position where both $\phi^n(a)$ and $\phi^n(2^k b)$ has a one. This can be generalised to $\phi^n(2^{k+m}) = 2^m a \ \mathrm{AND}_\phi \, 2^{k+m} b$ for all non-negative integers m. Consequently, $\phi^n(2^{m+k}) \in a\mathbb{Z} \ \mathrm{AND}_\phi \ b\mathbb{Z}$ for all integers m. Now,

$$\alpha(\phi^n(\{2^{m+k} \mid m \in \mathbb{N}\})) = 0 + 2^k\mathbb{Z},$$

which implies that $\alpha(\phi^n(2^k\mathbb{Z})) \sqsubseteq a\mathbb{Z} \ \mathrm{AND}_\phi \ b\mathbb{Z}$. $\square$

Using the result of Lemma 6, the proof of Proposition 6 can be given.

Proof. (of Proposition 6)
Let $c = a \, \mathrm{AND}_\phi \, b$, then $\alpha^n(a\mathbb{Z} \ \mathrm{AND}_\phi \ b\mathbb{Z}) \sqsupseteq c{+}0\mathbb{Z}$, since $c \in a\mathbb{Z} \, \mathrm{AND}_\phi \, b\mathbb{Z}$. Now, $\alpha^n(a\mathbb{Z} \ \mathrm{AND}_\phi \ b\mathbb{Z})$ must be equal to or larger than the supremum of the two elements $c + 0\mathbb{Z}$ and $0 + 2^m\mathbb{Z}$ since they are included (the latter by Lemma 6). But, $c + 0\mathbb{Z} \sqcup 0 + 2^m\mathbb{Z} = 0 + \gcd(0, 2^m, 2^m - c)\mathbb{Z}$. Since a and b are odd, c must also be odd, thus, the greatest common divisor of these is one, implying that $\alpha^n(\phi^n(a\mathbb{Z}) \ \mathrm{AND} \ \phi^n(b\mathbb{Z})) \sqsupseteq 0 + 1\mathbb{Z}$. $\square$

Proposition 6 states that the AND operation for abstract values which have a moduli which is odd results in the top value. This justifies the following abstraction, we introduce a *weakening operator* on $C(\mathbb{Z})$. The weakening operator ξ is defined as followed:

$$\xi(m + k\mathbb{Z}) = (m \bmod \gcd(2^n, k)) + \gcd(2^n, k)\mathbb{Z}$$

Note that $m + k\mathbb{Z} \sqsubseteq \xi(m + k\mathbb{Z})$, which follows directly from the definition of $\sqsubseteq$ given in Section 5.3. For any abstract value c, $\xi(c)$ will have a moduli of the form 2^k for some $k \in \mathbb{N}$. Having a moduli as a power of two is beneficial since it means that the set $2^k\mathbb{Z}$ has its k least significant digits as zeroes.

Lemma 7. *If A and B are bit strings such that $A \,\mathrm{AND}\, B = 0$. Then, for any bit string C:*

$$(A + B) \,\mathrm{AND}\, C = A \,\mathrm{AND}\, C + B \,\mathrm{AND}\, C$$

Furthermore, $(A \,\mathrm{AND}\, C) \,\mathrm{AND} (B \,\mathrm{AND}\, C) = 0$.

Proof. Since A and B contains no common ones, a plus between two bits with at least one zero is exactly the same as the OR operation. Thus:

$$(A + B) \,\mathrm{AND}\, C = (A \,\mathrm{OR}\, B) \,\mathrm{AND}\, C$$

Bitwise OR is distributive over bitwise AND, so

$$(A \,\mathrm{OR}\, B) \,\mathrm{AND}\, C = (A \,\mathrm{AND}\, C) \,\mathrm{OR} (B \,\mathrm{AND}\, C)$$

Note that since $A \,\mathrm{AND}\, B = 0$, then for any string C, it holds that

$$(A \,\mathrm{AND}\, C) \,\mathrm{AND} (B \,\mathrm{AND}\, C)$$

as C can only ”remove” ones from A and B. As a consequence,

$$(A \,\mathrm{AND}\, C) \,\mathrm{OR} (B \,\mathrm{AND}\, C) = (A \,\mathrm{AND}\, C) + (B \,\mathrm{AND}\, C)$$

since OR between two bits, where at least one is zero, equals the plus operation. $\square$

Note that if $M(A) < L(A)$ implies that $A \,\mathrm{AND}\, B = 0$. Now we present the definition of the abstract AND operation.

Proposition 7. *Let $m + 2^k\mathbb{Z}$ and $m' + 2^{k'}\mathbb{Z}$ be non-bottom abstract values such that $k' \leq k$. Then,*

$$m + 2^k\mathbb{Z} \,\widehat{\mathrm{AND}}\, m' + 2^{k'}\mathbb{Z} = \begin{cases} m \,\mathrm{AND}_\phi\, m' + 2^{k'}\mathbb{Z} & \textit{if } M(m') < k' \\ m \,\mathrm{AND}_\phi\, m' + 2^k\mathbb{Z} & \textit{otherwise} \end{cases}$$

is a correct abstraction of AND.

Proof. Take $m + 2^k n \in m + 2^k\mathbb{Z}$ and $m' + 2^{k'}n' \in m' + 2^{k'}n'\mathbb{Z}$ such that $k' \leq k$ arbitrarily. Note that $k' \leq k$ is not a restriction since the abstract values could just be reversed. Since we assume that m and m' are minimal representatives for the abstract values respectively it follows that

$$m \,\mathrm{AND}_\phi\, 2^k n = 0, m' \,\mathrm{AND}_\phi\, 2^{k'}n' = 0$$

Thus,

$$m+2^k n \,\mathrm{AND}_\phi\, m' + 2^{k'}n' = (m+2^k n \,\mathrm{AND}_\phi\, m') + (m+2^k n \,\mathrm{AND}_\phi\, 2^{k'}n')$$

by Lemma 7. Furthermore,

$$\begin{aligned}
(m + 2^k n \,\mathrm{AND}_\phi\, m') + (m + 2^k n \,\mathrm{AND}_\phi\, 2^{k'}n') = && (5.3)\\
((m \,\mathrm{AND}_\phi\, m') && (5.4)\\
+(2^k n \,\mathrm{AND}_\phi\, m')) && (5.5)\\
+((m \,\mathrm{AND}_\phi\, 2^{k'}n') && (5.6)\\
+(2^k n \,\mathrm{AND}_\phi\, 2^{k'}n')) && (5.7)
\end{aligned}$$

Now, term 5.5 is equal to zero, since $2^{k'}z \,\mathrm{AND}_\phi\, m' = 0$ for any $z \in \mathbb{N}$. In particular, it holds for $z_0 = 2^{k-k'}n$ (this is an integer since $k \geq k'$), since $2^{k'}z_0 = 2^{k'}2^{k-k'}n = 2^{k'+k-k'}n = 2^k n$.

Term 5.6 has two cases: if $M(m) < L(2^{k'}n')$ then it is equal to zero since m and $2^{k'}n'$ would have no common ones. But if $M(m) \geq L(2^{k'}n')$ then the term may result in a non-zero, where the possible ones are in the position-interval $[L(2^{k'}n'), M(m)]$. Thus it is safe to represent this term by the expression $2^{k'}z$ for some $z \in \mathbb{N}$.

Term 5.7 has its k least significant digits as zeroes, since $k \geq k'$, thus $(2^k n \,\mathrm{AND}_\phi\, 2^{k'}n')$ can be written as $2^k p$ for some $p \in \mathbb{N}$.

Thus, term 5.4 is $m \,\mathrm{AND}_\phi\, m'$, term 5.5 is zero, term 5.6 is zero if $M(m) < L(2^{k'}n')$ and can be written as $2^{k'}z$ for some z otherwise. Term 5.7 can be rewritten as $2^k p$ for some p. Thus we can rewrite (5.3) as

$$m \,\mathrm{AND}_\phi\, m' + 2^k p$$

if $M(m) < L(2^{k'}n')$, and

$$m \,\mathrm{AND}_\phi\, m' + 2^{k'}z + 2^k p$$

otherwise. Since the elements were taken arbitrary it means that *all* elements can be written on this form, with variations on z and p. Thus, it is safe to say

$$m \,\mathrm{AND}_\phi\, m' + 2^k p \in m \,\mathrm{AND}_\phi\, m' + 2^k\mathbb{Z}$$
$$m \,\mathrm{AND}_\phi\, m' + 2^{k'} z + 2^k p \in m \,\mathrm{AND}_\phi\, m' + 2^{k'}\mathbb{Z}$$

The latter can be deduced from the fact that $k' < k$. □

Note that Proposition 7 defines $\widehat{\mathrm{AND}}$ only for abstract values on the form $m + 2^k\mathbb{Z}$. Any abstract value can be converted to this form via the weakening operator however, making a safe abstraction of the value in question. A definition on the regular form $m + k\mathbb{Z}$ is not useful since by Proposition 6 any odd moduli will result in the top value. Now, when we have the bitwise $\widehat{\mathrm{AND}}$ and bitwise $\widehat{\mathrm{NOT}}$ we can easily construct the bitwise $\widehat{\mathrm{OR}}$ and $\widehat{\mathrm{XOR}}$ using identities such as De Morgan's Law. To show that these identities hold for abstract functions we show the following.

Proposition 8. *Let $f, g : \mathbb{Z} \to \mathbb{Z}$ be functions, and let $f_*, g_* : \mathcal{P}(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$ be the lifted functions defined as*

$$f_*(A) = \{f(a) \mid a \in A\}$$
$$g_*(B) = \{g(b) \mid b \in B\}$$

Note that f_* and g_* are monotone w.r.t. $\subseteq$. Now let $\widehat{f}$, $\widehat{g}$ be functions abstracting f_* and g_* respectively, i.e.,

$$f_* \subseteq \gamma \circ \widehat{f} \circ \alpha$$
$$g_* \subseteq \gamma \circ \widehat{g} \circ \alpha$$

Then, it holds that

$$f_* \circ g_* \subseteq \gamma \circ (\widehat{f} \circ \widehat{g}) \circ \alpha.$$

That is, function abstraction is closed under functional composition.

Proof. First,

$$f_* \circ g_* \subseteq f_* \circ (\gamma \circ \widehat{g} \circ \alpha)$$

since f_* is monotone and $g_* \subseteq \gamma \circ \widehat{g} \circ \alpha$ by definition of $\widehat{g}$. Furthermore, since $\widehat{f}$ is abstracting f_* we have:

$$f_* \circ (\gamma \circ \widehat{g} \circ \alpha) \subseteq (\gamma \circ \widehat{f} \circ \alpha) \circ (\gamma \circ \widehat{g} \circ \alpha) \subseteq \gamma \circ (\widehat{f} \circ \widehat{g}) \circ \alpha$$

where the last inequality is the implication of $\lambda l.l \sqsubseteq \alpha \circ \gamma$ by definition of a Galois-connection. $\square$

Thus, the following identities can be used to derive the abstract operations $\widehat{\mathrm{OR}}$ and $\widehat{\mathrm{XOR}}$:

$$(m + k\mathbb{Z})\ \widehat{\mathrm{OR}}\ (m + k\mathbb{Z}) = \\ \widehat{\mathrm{NOT}}((\widehat{\mathrm{NOT}}\ (m + k\mathbb{Z}))\ \widehat{\mathrm{AND}}\ (\widehat{\mathrm{NOT}}\ (m' + k'\mathbb{Z}))$$

and

$$(m + k\mathbb{Z})\ \widehat{\mathrm{XOR}}\ (m + k\mathbb{Z}) \\ = (m + k\mathbb{Z}\ \widehat{\mathrm{AND}}\ \ \widehat{\mathrm{NOT}}\ (m' + k'\mathbb{Z})) \\ \widehat{\mathrm{OR}}\ (\ \widehat{\mathrm{NOT}}\ (m + k\mathbb{Z})\ \widehat{\mathrm{AND}}\ m' + k'\mathbb{Z}).$$

5.5.3 Shifting

Shifting bit-strings is a common low-level operation. Left shifting one step is the process of moving all bits to the left and inserting a zero at the least significant position. A right shift inserts zero from the right instead. An arithmetic right shift inserts a copy of the most significant bit instead of a zero. Shifting left one step is equal to multiplying by two (holds both for signed and unsigned integers) and shifting right one step corresponds to (integer) division by two. We use the notation a LSH b to denote that a should be left shifted b steps to the left. We assume that the right argument is always positive.

Definition 22. *Let $m + k\mathbb{Z}$ and $a + 0\mathbb{Z}$ be non-bottom abstract values such that a is non-negative. Then,*

$$(m + k\mathbb{Z})\ \widehat{\mathrm{LSH}}\ (a + 0\mathbb{Z}) \overset{\mathrm{def}}{=} 2^a m + 2^a k\mathbb{Z}$$

is a correct abstraction for left shifting with a singleton abstract value $a + 0\mathbb{Z}$.

Proof. Take an arbitrary element $m + kn \in m + k\mathbb{Z}$. Left shifting $a + 0\mathbb{Z} = a$ steps corresponds to multiplying $m + kn$ by 2^a. Thus,

$$2^a(m + kn) = 2^a m + 2^a kn \in 2^a m + 2^a k\mathbb{Z}$$

$\square$

Proposition 9. *Let $m+k\mathbb{Z}$ and $m'+k'\mathbb{Z}$ be non-bottom abstract values. Then the definition of the abstract left shift operator* $\widehat{\mathrm{LSH}}$ *:*

$$(m+k\mathbb{Z})\ \widehat{\mathrm{LSH}}\ (m'+k'\mathbb{Z}) \overset{def}{=} 2^{m'}m + 2^{m'}\gcd(k, m(2^{k'}-1))\mathbb{Z}$$

is correctly abstracting LSH.

Proof. Naturally, the best correct approximation of this operation is

$$\bigsqcup\{2^a m + 2^a k\mathbb{Z} \mid a \in m' + k'n' \wedge n' \in \mathbb{Z}\}$$

since the supremum of all constant right shifts should equal it. First we compute

$$\bigsqcup\{2^a m + 2^a k\mathbb{Z} \mid a \in m' + k'n' \wedge n' \in \{0,1\}\}$$

In the case where $n' = 0$ then $a = m'$, when $n' = 1$, then $a = m' + k'$. Thus, we compute

$$\begin{aligned}
&2^{m'}m + 2^{m'}k\mathbb{Z} \sqcup 2^{m'+k'}m + 2^{m'+k'}k\mathbb{Z}\\
&= 2^{m'}m + \gcd\left\{2^{m'}k, 2^{m'+k'}k, |2^{m'+k'}m - 2^{m'}m|\right\}\mathbb{Z}\\
&= 2^{m'}m + \gcd\left\{2^{m'}k, |2^{m'}2^{k'}m - 2^{m'}m|\right\}\mathbb{Z}\\
&= 2^{m'}m + \gcd(2^{m'}k, 2^{m'}m|2^{k'}-1|)\mathbb{Z}\\
&= 2^{m'}m + 2^{m'}\gcd(k, m(2^{k'}-1))\mathbb{Z}
\end{aligned}$$

To conclude the proof, let n' be any integer greater than one, and we take

$$\begin{aligned}
&2^{m'}m + 2^{m'}\gcd(k, m(2^{k'}-1))\mathbb{Z} \sqcup 2^{m'+n'k'}m + 2^{m'+n'k'}k\mathbb{Z}\\
&= 2^{m'}m + \gcd\left\{2^{m'}\gcd(k, m(2^{k'}-1)), 2^{m'+n'k'}k, |2^{m'+n'k'}m - 2^{m'}m|\right\}\mathbb{Z}\\
&= 2^{m'}m + \gcd\left\{2^{m'}k, 2^{m'}(2^{k'}-1)m, 2^{m'}2^{n'k'}k, |2^{m'}2^{n'k'}m - 2^{m'}m|\right\}\mathbb{Z}\\
&= 2^{m'}m + 2^{m'}\gcd\left\{k, m(2^{k'}-1), 2^{n'k'}k, m(2^{n'k'}-1)\right\}\mathbb{Z}
\end{aligned}$$

Now since k divides $2^{n'k'}k$ for any n' and $m(2^{k'}-1)$ divides $m(2^{n'k'}-1)$ for any n', we conclude that

$$\begin{aligned}
&2^{m'}m + 2^{m'}\gcd\left\{k, m(2^{k'}-1), 2^{n'k'}k, m(2^{n'k'}-1)\right\}\mathbb{Z}\\
&\qquad\qquad \sqsubseteq 2^{m'}m + 2^{m'}\gcd(k, m(2^{k'}-1))\mathbb{Z}.
\end{aligned}$$

Since $n' > 1$ was chosen arbitrarily and merged with $n' = 0$ and $n' = 1$ the result holds by merging any other n' as well, so this is a safe approximation.

□

The right-shift operation behaves less well. Right shifting is equivalent to (integer) division by two. As seen in section 5.3 division destroys most congruence relations.

Proposition 10. *Let a be any positive integer and $m + k\mathbb{Z}$ be a non-bottom abstract value, then the definition*

$$(m + k\mathbb{Z})\ \widehat{\mathrm{RSH}}\ a$$

$$\overset{def}{=} \begin{cases} m \ \mathrm{RSH}\ a + 2^{t-a}\mathbb{Z} & \textit{if } k = 2^t \textit{ and } a < t \\ 0 + 1\mathbb{Z} & \textit{otherwise} \end{cases} \tag{5.8}$$

$$(m + k\mathbb{Z})\ \widehat{\mathrm{RSH}}\ (m' + k'\mathbb{Z}) \overset{def}{=} 0 + 1\mathbb{Z} \tag{5.9}$$

is a correct definition of the abstract right shift.

This proposition is stated without proof since (5.8) is just a re-writing of the definition of the abstract integer-division and (5.9) is trivial. Note that we do not claim that (5.9) is the *best* definition, but certainly a correct one. However, it seems unlikely that it is possible to define a *generally* better abstract operation.

Chapter 6

Parametric WCET Analysis

6.1 Introduction

In Chapter 4, we showed how to count elements in abstract environments to calculate bounds for loops. This was done with non-relational abstract interpretation. However, a relational domain can provide more information and can preserve some relations between variables. In this chapter we will show how a relational abstract interpretation and some other techniques can achieve a parametric WCET analysis. A brief overview of this framework was given in Chapter 2, while this chapter presents the details. The general methodology presented in this chapter was first proposed in [Lis03a] and further explained in [Lis03b]. An overview of the parametric framework is shown in Figure 6.1.

6.2 Relational Abstract Interpretation and Input Parameters

As explained in Section 3.10.3 and as seen in Figure 6.1, abstract interpretation and element counting is used to obtain the execution count function ECF. The purpose of the execution count function is to compute an upper bound of the number of times that a program point can be visited, given a set of initial values for the input parameters. Thus, the relation between the possible environments associated with a program point and the values of the input parameters has to be analysed. This is done via abstract interpretation, using a *relational* domain, such as the polyhedral or octagon domain.

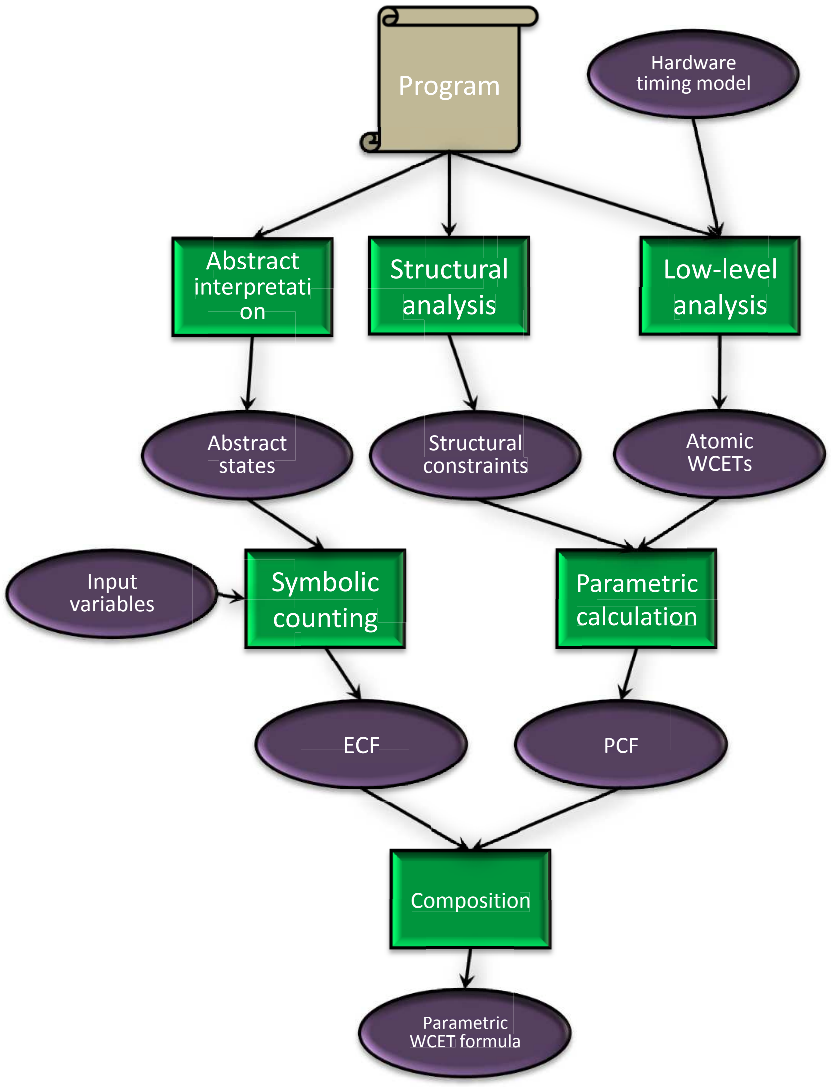


Figure 6.1: The workflow of the parametric framework

Since the input parameters has to be related to the variables in the program, they have to be present in the abstract interpretation. This can be done by adding a set of artificial constants to the program corresponding to the input parameters. Consider the following program:

```
**while** $i > 0$ **do**
    $i \leftarrow i - 1$
**end while**
```

In this program, the initial value i_0 of i would be considered an input parameter. To model this, the artificial constant i_0 has to be added to the abstract interpretation and the artificial statement $i = i_0$ is added to the beginning of the program. Note that these artificial constants and statements don't need to be added manually, it is a straightforward transformation which can be done automatically given a set of input parameters. Indeed this doesn't even have to be applied to the actual code, since it is just a matter of modeling the abstract interpretation. However, in many cases the input parameters correspond to constant variables in the program. In those cases no artificial constants have to be introduced. As an example, in L (see Section 3.1), the input parameter n_0 does not have to be explicitly modelled since $n = n_0$ through the execution.

Consider a program T with $\mathcal{V}_T = \{i\}$ and $\mathcal{I}_T = \{n\}$. Figure 6.2 shows an abstract state which has been derived for $q \in \mathcal{Q}_T$, using the polyhedral domain. Note here that n is not a program variable but an artificial constant corresponding to an input parameter. As can be seen, i has the maximal number of elements when n is 1, namely 7. However, since n corresponds to an input parameter, it should be known before execution. Thus, the bound should be expressed in terms of n. In this case the function $f(n) = \max(0, 9 - 2n)$ would be a precise upper bound when n is known. Note that, if a non-relational domain was used, the number of possible values for i would be independent of n so such a function would not be meaningful. The idea of the relational abstract interpretation is thus

> to derive relational abstract states in a program in order to express their execution counts in terms of input parameters.

6.3 Counting Elements in a Relational Abstract Environment

In Section 4.6, some examples of how to count elements in abstract environments are shown. In this case it was fairly simple since the domains were

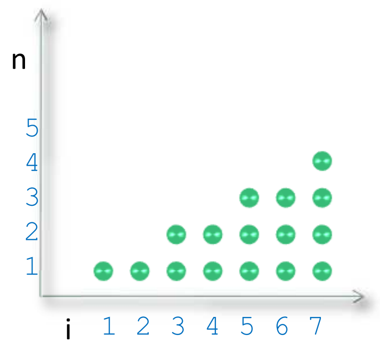


Figure 6.2: A triangular abstract environment

non-relational. In the non-relational case it is sufficient to count the states of an abstraction of $\mathcal{P}(\mathbb{Z})$ individually, and then to multiply the individual counts for each variable with each other. In the relational case, it becomes more complex since we need to count abstractions of environments. The definition of the *size* of an abstract environment for a relational domain is the same as (4.1) given in Section 4.6, i.e.,

$$|\,\widehat{\sigma}\,| = |\gamma(\sigma)|$$

This calculation however, is for most relational domains non-trivial. As an example, the abstract environment depicted in Figure 6.2 is represented the following system of linear inequalities:

$$\widehat{\sigma}_q = \begin{pmatrix} 1 & -2 & 1 \\ -1 & 0 & 7 \\ 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} i \\ n \\ 1 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

The function γ of this abstract environment maps to the set

$$\gamma(\widehat{\sigma}_q) = \{[i \mapsto i'][n \mapsto n'] \mid n' \geq 1 \wedge i' \leq 7 \wedge i' \geq 2n' - 1 \wedge i', n' \in \mathbb{Z}\}$$

To compute the size of this set is non-trivial and is equal to computing the number of integer solutions to a system of linear inequalities. There are known methods of doing this, which is shown in the following sections. However, as shown earlier, in this case the variable n should be disregarded, so we want to compute the number of elements in this set *symbolically*, i.e., in terms of the value of variable n. This makes the computation even more complex. The following subsections reviews some methodologies to compute sizes of this kind of sets.

6.3.1 Ehrhart Polynomials

In [Cla96], a method of symbolically counting the number of integer points inside the union of rational convex polytopes (i.e., bounded polyhedra) is presented. The method uses Ehrhart's theory to find quasi-polynomials (polynomials which has periodic functions as coefficient, which simply can be seen as a finite set of polynomials) which corresponds to the number of integer points in polytopes.

6.3.2 Barvinok's Rational Functions

In [VSB$^+$07], a parametric version of Barvinok's rational functions [BBP99] is presented, using a similar method of [Cla96] to find quasi-polynomials that represent the number of integer points inside polyhedra. The method can be extended to handle general Presburger formulae, but it requires potentially very costly preprocessing using parametric integer programming [Fea88].

6.3.3 Successive Projection

The successive projection method was suggested in [Pug94]. The method is used to count the number of solutions to a Presburger formula, which is more general than unions of polytopes. The method is presented as a set of rules to successively transform a symbolic summation to a formula. Since the method is not an algorithm, but a set of rules, some additional work has to be done to make it computable. This method has been implemented in a prototype tool for the parametric framework presented in this chapter [BL08]. Since Pugh's method has been investigated and implemented, this is the method we make use of to count integer points in polyhedra in this thesis. For this reason, we explain the method more thoroughly.

The method computes the result of generalised sums $(\Sigma V : P : x)$ where V is a set of variables to sum over, P is a Presburger formula (the guard) and x is any formula. The result of such a sum is the sum for all variables $v \in V$ which satisfy P of x. As a simple example, the sum $\sum_{v=l}^{u} v$ is represented by the general summation $(\Sigma\{v\} : l \leq v \leq u : v)$ and the sum $\sum_{i=0}^{n}\sum_{j=i}^{m} 1$ would be represented by $(\Sigma\{i, j\} : 0 \leq i \leq n \wedge i \leq j \leq m : 1)$. The result is then computed by choosing an appropriate projection rule to simplify the formula. The most important rule to reduce a generalised sum $(\Sigma V : P : x)$ is to choose a variable $v \in V$ and compute the general sum

$$(\Sigma V \setminus \{v\} : P' : (\Sigma\{v\} : l \leq v \leq u : x))$$

where P' is P where all information about v is removed. Since $(\Sigma\{v\} : l \leq v \leq u : x)$ is equivalent to $\sum_{v=l}^{u} x$, known formulae of summations over the form of x can be used to simplify it. If $V \setminus \{v\}$ is non-empty another variable is chosen and the procedure is repeated until $V = \varnothing$, and the result is a sum of generalised sums $(\Sigma : G : x')$ which should be read as "x' if G holds, else 0". This result is symbolic in the variables occurring free in x or P but not in V. To exemplify, take the sum $(\Sigma\{i, j\} : 0 \leq i \leq n \wedge i \leq j \leq m : 1)$ again. Applying the rule, projecting the variable i, above would yield

$$\begin{aligned}&(\Sigma\{i, j\} : 0 \leq i \leq n \wedge i \leq j \leq m : 1)\\ &\qquad\qquad = (\Sigma\,\{j\} : i \leq j \leq m : (\Sigma\,\{i\} : 0 \leq i \leq n : 1))\end{aligned}$$

Now, naturally $(\Sigma\,\{i\} : 0 \leq i \leq n : 1) = n+1$ (this is a "known formula"), so we can conclude

$$(\Sigma\,\{j\} : i \leq j \leq m : (\Sigma\,\{i\} : 0 \leq i \leq n : 1)) = (\Sigma\,\{j\} : i \leq j \leq m : n + 1)$$

and continue applying rules until we cannot apply more rules. The situation is however not always this easy; variables can have several lower/upper bounds or be unbounded and bounds can be negative and/or rational. All these cases are handled in [Pug94].

In the special case where the integer points are counted inside a convex polyhedron, the problem becomes a bit easier. In the following, we will assume that it is integer points inside convex polyhedra which are to be counted[1]. We assume two restrictions of the generalised sums; first, rather than having the guard as a Presburger formula, the guard is considered to be a system of linear inequalities in the variables of $\mathcal{V}_P$ (since this is exactly what the polyhedral abstract interpretation will give). The other restriction is that we model the formulae to sum over as polynomials, simplifying both representation and computation. Polynomials can easily be modelled as a sum of terms, where a term is a vector representing an integer coefficient and variable powers. As an

[1]Note that this also is applicable for the octagon domain, since its abstract environments are also abstract environments of the polyhedral domain.

example we can model the polynomial $3a^2b^3 + 5a^4$ (assuming $\mathcal{V}_P = \{a, b\}$) as the sum of the terms $(3\ 2\ 3)$ and $(5\ 4\ 0)$. This also makes arithmetical operations on these vectors straightforward to implement. Furthermore, since the guards are polyhedra, the lower and upper bound of any variable will be sets of linear expressions. Summing a linear expression over a polynomial is again a polynomial, so this model is closed under summations. However, these restrictions sometimes require the result to be slightly over-approximated. The constraint $3a - b \leq 0$ gives an upper bound for a as $a \leq \lfloor \frac{b}{3} \rfloor$, since a and b are integers. As seen, the upper bound is not a polynomial and therefore problematic in our model. Since $\frac{b}{3}$ is a safe upper bound for $\lfloor \frac{b}{3} \rfloor$ and on polynomial form we can use it as approximation. Lower bounds are handled in a similar fashion.

6.4 Obtaining ECF_P

In this section we will give an example on how to obtain the ECF_P function using relational abstract interpretation and counting of elements. The example is performed on the program L in Figure 3.2 on page 18. This example uses abstract interpretation with the polyhedral domain [CH78] and counting of integer points by successive projection [Pug94].

6.4.1 Polyhedral Abstract Interpretation

Abstract interpretation using the polyhedral domain results in the following abstract states of L (presented in a more human readable format than matrices).

$$\begin{array}{ll} \widehat{\mathcal{S}}_0 = \top & \widehat{\mathcal{S}}_3 = \{i \geq 0, i \geq n+1\} \\ \widehat{\mathcal{S}}_1 = \{i = 0\} & \widehat{\mathcal{S}}_4 = \{0 \leq i \leq n\} \\ \widehat{\mathcal{S}}_2 = \{i \geq 0\} & \widehat{\mathcal{S}}_5 = \{1 \leq i \leq n+1\} \end{array} \tag{6.1}$$

6.4.2 Counting Integer Points

The ECF_L function is computed by calculating the size of the abstract states in (6.1). We are interested in computing the sizes *symbolically* in terms of the *input parameters* of L. In practice, using Pugh's method this consists of summing over all non-constant program variables, but not the symbolic constants corresponding to input parameters. For L this means, we sum over $\{i\}$. The guard in the general sums will correspond to the inequalities given for a convex polyhedron. The result of computing the sizes is shown below:

$$
\begin{aligned}
|\,\widehat{\mathcal{S}}_0| &= (\Sigma i : \varnothing : 1) = \infty \text{ (unbounded sum)}\\
|\,\widehat{\mathcal{S}}_1| &= (\Sigma i : i = 0 : 1) = 1\\
|\,\widehat{\mathcal{S}}_2| &= (\Sigma i : i \geq 0 : 1) = \infty\\
|\,\widehat{\mathcal{S}}_3| &= (\Sigma i : i \geq 0 \wedge i \geq n+1 : 1) = \infty\\
|\,\widehat{\mathcal{S}}_4| &= (\Sigma i : 0 \leq i \leq n : 1) = (\Sigma : n \geq 0 : n+1)\\
|\,\widehat{\mathcal{S}}_5| &= (\Sigma i : 1 \leq i \leq n+1 : 1) = (\Sigma : n \geq 0 : n+1)
\end{aligned}
\tag{6.2}
$$

Note that we have used the short hand Σi for $\Sigma\,\{i\}$. By this, the ECF_L function has been computed. The function takes an instance of the input parameter n and returns a vector of upper execution bounds for each program point. Component j of the result will then be interpreted as the upper execution count bound for program point q_j. The formal definition of ECF_L is thus,

$$\mathrm{ECF}_L = \lambda n.\,\langle \infty, 1, \infty, \infty, (n \geq 0\ ?\ n+1\ :\ 0), (n \geq 0\ ?\ n+1\ :\ 0)\rangle$$

where $(n \geq 0\ ?\ n+1\ :\ 0)$ is a compressed if-statement borrowed from C, it should be understood as "if $n \geq 0$ then $n+1$ else 0". However, our assumption about programs suggests that the single entry and exit points of a program will be taken exactly once. Thus, a better definition of ECF_L is to ignore the counting for these program points and define the upper bound for the initial program point q_0 and the final program point q_3 as one, giving

$$\mathrm{ECF}_L = \lambda n.\,\langle 1, 1, \infty, 1, (n \geq 0\ ?\ n+1\ :\ 0), (n \geq 0\ ?\ n+1\ :\ 0)\rangle$$

Note that no finite bound could be found for q_2. This is because the widening in the abstract interpretation yielded an unbounded polyhedron for $\widehat{\mathcal{S}_2}$. However, it is not necessary to have bounds on all program points in order to obtain a WCET bound or loop bound, since it is in general the minimum of the bounds that are interesting (see Chapter 4).

As an example, the upper execution bounds for the six program points in $\mathcal{Q}_L$ for the input parameter instantiated as $n_0 = 2$ equals

$$\mathrm{ECF_L}(2) = \langle 1, 1, \infty, 1, 3, 3\rangle\,.$$

6.5 Obtaining PCF_P

As seen in Chapter 1, the calculation phase of WCET analysis combines the results from low-level analysis and high-level flow analysis to compute a concrete worst-case execution time. If the calculation phase is altered, it can be used to compute a parametric worst-case execution time. In this section we will look at some techniques for computing a WCET which is parametric in the number of times that the program points can be maximally visited. In other words, the WCET is computed in terms of parametric capacities of the flow chart. In this framework, as seen in Figure 6.1 and in Section 3.10.3, parametric calculation is used to obtain the function PCF_P of a program P.

6.5.1 Parametric Calculation

Parametric Calculation can be stated as the general problem of maximising the objective function

$$\sum_{q \in \mathcal{Q}_P} c_q x_q$$

subject to constraints on the program flow

$$A\mathbf{x} + \mathbf{b} \geq \mathbf{0}$$

as well as the *symbolic constraints*

$$x_q \leq p_q$$

where $\mathbf{c}$ is a vector of atomic WCETs obtained from low-level analysis, $\mathbf{x}$ is the solution vector corresponding to execution counts for each program point in the program and $\mathbf{p}$ is a vector of symbolic execution bounds for each program point. The solution of a parametric calculation is a formula expressing the vector $\mathbf{x}$ in terms of the symbols in $\mathbf{p}$.

Two approaches to solve this has been proposed, one based on parametric integer programming [Lis03a, BL08] and one based on propagation of flow constraints [BEL09]. In this chapter we will focus on the first method while the second is handled in Chapter 7.

6.5.2 Parametric Integer Programming

P. Feautrier suggested in [Fea88] an algorithm for parametric integer programming (PIP). Parametric integer programming gives the lexicographical minimum of the set

$$F(\mathbf{z}) = \{A\mathbf{y} + B\mathbf{z} + \mathbf{c} \geq \mathbf{0} \wedge \mathbf{y} \in \mathbb{Z}^n\}$$

in terms of the vector $\mathbf{z}$. In the following, we call $\mathbf{y}$ the *solution vector* and $\mathbf{z}$ the *parameter vector*. The matrices A and B correspond to constraints on the variables and the parameters, respectively. There exists a tool called Piplib, which is an open source implementation of the algorithm [Pip09]. The result of a parametric integer problem is a binary tree where the leaves correspond to linear solutions and the other nodes correspond to linear conditionals. The tree is expressed as a nested `if`-statement (an example of a solution can be seen in Figure 6.3 on page 96).

6.5.3 PIP as Parametric Calculation

Parametric calculation with PIP can be formulated by having the execution count vector $\mathbf{x}$ as solution vector $\mathbf{y}$ in the PIP problem. The matrix A and the vector $\mathbf{c}$ will correspond to program flow constraints, and the vector of symbolic upper bounds $\mathbf{p}$ will be the parameter vector $\mathbf{z}$ in PIP. However, using PIP with this set-up will result in the lexicographical minimum of $\mathbf{x}$. What is desired is to obtain the maximum of an objective function. The solution (which can be found in the Piplib manual [Pip09]) is to introduce a new variable y which represents the objective function. Since y should be maximised rather than minimised, an artificial ”big” parameter B is introduced. The parameter B is considered to be arbitrarily large, and therefore a maximisation problem is achieved by minimising $B - y$. In addition, to actually connect y to the objective function $\sum_{q \in \mathcal{Q}_P} c_q x_q$, an additional constraint $B - y \leq \sum_{q \in \mathcal{Q}_P} c_q x_q$ has to be added. Now, the new variable y has to be added to the solution vector $\mathbf{x}$, as the *first component* of the solution vector. In this sense, y is guaranteed to have the highest priority in the lexicographical ordering. Thus, PIP will attempt to minimise $B - y$ which by the constraint is guaranteed to be less than the objective function.

The function PCF_P is obtained by parametric calculation. The function calculates the worst-case execution times of a program P given a vector of upper bounds for each program point. We illustrate by an example of the program L in Figure 3.2 on page 18. The objective function in this case is obtained by the cost vector presented in (3.1) on page 22. The function PCF_L is then obtained by maximising:

$$x_0 + 3x_1 + x_2 + 2x_3 + 2x_4 + 8x_5 \tag{6.3}$$

subject to some constraints. To get a bounded problem it is enough to provide structural constraints of a program together with the symbolic constraints. This is because the solution will be expressed in terms of the symbolic execution bounds.

The structural constraints are obtained by adding constraints for each program point q. The exact form of constraint is determined by the type of program point. The initial program point q_0 and the final program point q_f will be taken exactly once, so for these the constraints

$$x_0 = 1$$
$$x_f = 1$$

are added. For any program point q succeeding a merge node, the sum of the two incoming edges q' and p' of the merge node will equal the execution count of q. Thus,

$$x_q = x_{q'} + x_{p'}$$

can be added. For a program point q succeeding an assignment node, the execution count simply equals the execution count of the incoming program point to that assigment node q', so

$$x_q = x_{q'}$$

can be added. Finally, for any program point *proceeding* a conditional, the execution count is equal to the sum of the two outgoing program points of the conditional $q_{\texttt{true}}$ and $q_{\texttt{false}}$. Thus,

$$x_q = x_{\texttt{true}} + x_{\texttt{false}}$$

can be added as constraints.

This process is what is referred to as *structural analysis* in Figure 6.1. The structural constraints of L are:

$$\begin{array}{lr}
x_0 = 1 & \text{initial node} \\
x_1 = x_0 & \text{proceeding assignment} \\
x_2 = x_1 + x_5 & \text{proceeding merge node} \\
x_2 = x_3 + x_4 & \text{preceeding conditional} \\
x_5 = x_4 & \text{proceeding assignment} \\
x_3 = 1 & \text{final program point}
\end{array} \qquad (6.4)$$

Finally, the symbolic constraints $x_q \leq p_q$ for all $q \in \mathcal{Q}_L$ have to be added. Maximising (6.3) subject to these constraints with Piplib yields the result shown

```
$\mathrm{PCF}_L = \lambda p_0, p_1, p_2, p_3, p_4, p_5.$
if $p_2 \geq 1$ then
  if $p_0 \geq 1$ then
    if $p_1 \geq 1$ then
      if $p_3 \geq 1$ then
        if $p_2 \leq p_4 + 1$ then
          if $p_2 \leq p_5 + 1$ then
            $11p_2 - 4$
          else
            $11p_5 + 7$
          end if
        else
          if $p_4 \leq p_5$ then
            $11p_4 + 7$
          else
            $11p_5 + 7$
          end if
        end if
      end if
    end if
  end if
else
  $0$
end if
```

Figure 6.3: Result of PCF_L function from Piplib

in Figure 6.3. As can be seen in the figure, this result is the definition of PCF_L. As an example, suppose that $p_0, ..., p_5$ are instantiated as $\langle 1, 1, \infty, 1, 3, 3\rangle$. Then,

$$\mathrm{PCF}_L(1, 1, \infty, 1, 3, 3) = 11 \cdot 3 + 7 = 40$$

which should be easy to see by studying Figure 6.3.

6.6 Obtaining PWCET_P

The final and most interesting function is PWCET_P which computes a WCET bound given an instantiation of all input parameters of a program. The function is obtained by the composition of ECF_P and PCF_P.

$$\mathrm{PWCET}_P = \mathrm{PCF}_P \circ \mathrm{ECF}_P$$

Functional composition is simple in terms of computation, since it constitutes of substituting the result of ECF_P for the arguments in PCF_P. Figure 6.4 shows the composition of ECF_L and PCF_L which were derived in Section 6.4 and 6.5 respectively. As an example, we compute PWCET_L with the argument $n = 2$.

$$\begin{aligned}\mathrm{PWCET}_L(2) &= \mathrm{PCF}_L \circ \mathrm{ECF}_L(2)\\ &= \mathrm{PCF}_L(\langle 1, 1, \infty, 3, 3\rangle)\\ &= 40\end{aligned}$$

Note that this result is also obtained by substituting 2 for n in Figure 6.4. Compare this result to the trace computation in Figure 3.1 on page 23. The worst-case trace was computed as 40 which correspond exactly to the result in this case. So in this case the exact result was obtained from the method.

6.7 Simplifying PWCET_P

The final formula for PWCET_L shown in Figure 6.4 contains a lot of redundancy and is unnecessarily complex for this small example. Of course, this complexity is much worse in realistic examples. Thus, there is a need for simplification. During the composition phase, a lot of redundancy is introduced in the formula, which could easily be removed. Examples of trivial simplifications that can be done:

- All `then` branches can be cut from conditionals on the form $\infty \leq x$.
- All `else` branches can be cut from conditionals on the form $\infty \geq x$.
- All `else` branches can be cut from conditionals on the form $x \leq x$.
- If x, y are constants such that $x \leq y$, then all `else` branches can be cut from conditionals on the form $x \leq y$.

Applying these trivial simplifications to PWCET_L reduces the formula to:

$$\mathrm{PWCET}_L = \lambda n.11(n \geq 0\,?\,n{+}1\ :\ 0){+}7 = \lambda n.\begin{cases} 11n + 18 & \text{if } n \geq 0\\ 7 & \text{otherwise.}\end{cases}$$

```
$\mathrm{PWCET}_L = \lambda n.$
**if** $\infty \geq 1$ **then**
  **if** $1 \geq 1$ **then**
    **if** $1 \geq 1$ **then**
      **if** $1 \geq 1$ **then**
        **if** $\infty \leq (n \geq 0\ ?\ n+1\ :\ 0)) + 1$ **then**
          **if** $\infty \leq (n \geq 0\ ?\ n+1\ :\ 0) + 1$ **then**
            $11 \cdot \infty - 4$
          **else**
            $11(n \geq 0\ ?\ n+1\ :\ 0) + 7$
          **end if**
        **else**
          **if** $(n \geq 0\ ?\ n+1\ :\ 0) \leq (n \geq 0\ ?\ n+1\ :\ 0)$ **then**
            $11(n \geq 0\ ?\ n+1\ :\ 0) + 7$
          **else**
            $11(n \geq 0\ ?\ n+1\ :\ 0) + 7$
          **end if**
        **end if**
      **end if**
    **end if**
  **end if**
**else**
  $0$
**end if**
```

Figure 6.4: The resulting WCET_L function

6.8 Reducing the Number of Variables

PIP has exponential complexity in the number of variables in the worst-case, making scalability problematic. However, the structural constraints of a program produces an under-determined system of equations. In such a system with n variables, the solution space is the span of a set of $n - r$ vectors (where r is the rank of the constraint matrix). Thus, these $n-r$ vectors form a basis for the solution space. The variables can be expressed as linear combinations of this basis, meaning that the problem can be computed using only the basis. Let $A\mathbf{x} = \mathbf{b}$ be a system of structural and possible other linear equations obtained from flow analyses and $y = \mathbf{c}^{\mathrm{T}}\mathbf{x}$ be the cost function. The constraints together with the cost function is

$$\begin{pmatrix} 1 & -\mathbf{c} \\ 0 & A \end{pmatrix} \begin{pmatrix} y \\ \mathbf{x} \end{pmatrix} = \begin{pmatrix} 0 \\ \mathbf{b} \end{pmatrix} \tag{6.5}$$

If we perform Gauss-Jordan elimination on the above (including the right-hand side by augmenting the constraint matrix by $(0\ \mathbf{b})^{\mathrm{T}}$) and re-arrange the columns of A and the components of $\mathbf{x}$ such that all pivot columns are to the left, and $\mathbf{x}$ is re-arranged accordingly, we get

$$\begin{pmatrix} 1 & 0 & -\mathbf{c}' \\ 0 & I_r & A' \end{pmatrix} \begin{pmatrix} y \\ \mathbf{x}_{\mathrm{BV}} \\ \mathbf{x}_{\mathrm{FV}} \end{pmatrix} = \begin{pmatrix} z \\ \mathbf{b}' \end{pmatrix} \tag{6.6}$$

where I_r is the $r \times r$ identity matrix and r is the rank of A. Furthermore, z is the last column of the solution of the augmented matrix after elimination. The vector $\mathbf{x}$ has now been partitioned into two vectors, one partition is the vector of *basic variables* $\mathbf{x}_{\mathrm{BV}}$ with r components, and the other being $n - r$ *free variables* $\mathbf{x}_{\mathrm{FV}}$ (where n is the number of columns of A). Note that this transformation also removes any redundant constraints from the system. From this we can derive two important equations. One is the objective function expressed in terms of the free variables

$$y = z + \mathbf{c}'\mathbf{x}_{\mathrm{FV}} \tag{6.7}$$

and a way to express the basic variables in terms of the free ones

$$\mathbf{x}_{\mathrm{BV}} = \mathbf{b}' - A'\mathbf{x}_{\mathrm{FV}} \tag{6.8}$$

As we model the parametric upper bounds as the constraints $\mathbf{x} \leq \mathbf{p}$, we can now simply model our IPET problem as

$$\begin{pmatrix} z + \mathbf{c}'\mathbf{x}_{\mathrm{FV}} \\ \mathbf{b}' - A'\mathbf{x}_{\mathrm{FV}} \\ \mathbf{x}_{\mathrm{FV}} \end{pmatrix} \leq \begin{pmatrix} y \\ \mathbf{p}_{\mathrm{BV}} \\ \mathbf{p}_{\mathrm{FV}} \end{pmatrix} \tag{6.9}$$

where we have partitioned and re-arranged $\mathbf{p}$ exactly as for $\mathbf{x}$. Now it suffices to solve the IPET with these constraints, thus reducing the number of unknowns by the rank of A. This method of eliminating variables is not restricted to the parametric case, but can be used to reduce the dimensionality of any IPET problem.

6.8.1 Concrete Example of Variable Reduction

As an example on how the variable reduction can be applied we shall perform variable reduction on the program L. First we assemble a matrix like (6.5) by assembling the cost function (6.3) of L and the structural constraints (6.4). Thus, we have

$$\mathbf{c} = \begin{pmatrix} 1 & 3 & 1 & 2 & 2 & 8 \end{pmatrix}$$

and

$$\mathbf{b} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

and

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & -1 \\ 0 & 0 & -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

and finally,

$$\mathbf{x} = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 & x_4 & x_5 \end{pmatrix}$$

The resulting constraint matrix as in (6.5) is:

$$\left(\begin{array}{c|cccccc} 1 & -1 & -3 & -1 & -2 & -2 & -8 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}\right) \left(\begin{array}{c} y \\ \hline x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{array}\right) = \left(\begin{array}{c} 0 \\ \hline 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{array}\right)$$

Now we perform Gauss-Jordan elimination of the above to obtain a matrix as in (6.6). The result is:

$$\left(\begin{array}{c|ccccc|c} 1 & 0 & 0 & 0 & 0 & 0 & -11 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{array}\right) \left(\begin{array}{c} y \\ \hline x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ \hline x_5 \end{array}\right) = \left(\begin{array}{c} 7 \\ \hline 1 \\ 1 \\ 1 \\ 1 \\ 0 \end{array}\right)$$

This means that for L we have:

$$\mathbf{c}' = \begin{pmatrix} 11 \end{pmatrix}, A' = \begin{pmatrix} 0 & 0 & 0 & -1 & 0 & -1 \end{pmatrix}^{\mathrm{T}}$$
$$\mathbf{x}_{\mathrm{BV}} = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 & x_4 \end{pmatrix}^{\mathrm{T}}, \mathbf{x}_{\mathrm{FV}} = \begin{pmatrix} x_5 \end{pmatrix}$$
$$\mathbf{b}' = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 \end{pmatrix}^{\mathrm{T}}, z = 7$$

Thus, we can conclude (6.7) and (6.8) as

$$y = 7 + 11x_5$$

and

$$x_0 = 1, x_1 = 1, x_2 = x_5 + 1, x_3 = 1, x_4 = x_5$$

We can model the IPET problem by (6.9), which gives the following concrete constraints:

$$\begin{gathered} 7 + 11x_5 \le y \\ 1 + x_5 \le p_2 \\ 1 \le p_0, p_1, p_3 \\ x_5 \le p_4 \\ x_5 \le p_5 \end{gathered}$$

thus, we have reduced the problem to one unknown variable (x_5) from the original six. Solving this system with Piplib results in the following, which can be inspected to realise that it is the same as the solution shown in Figure 6.3:

```
$\lambda p_2, p_4, p_5.$
**if** $p_2 \geq 1$ **then**
  **if** $p_4 + 1 \geq p_2$ **then**
    **if** $p_5 + 1 \geq p_2$ **then**
      $y = 11p_2 - 4$
    **else**
      $y = 11p_5 + 7$
    **end if**
  **else**
    **if** $p_4 \leq p_5$ **then**
      $y = 11p_4 + 7$
    **else**
      $y = 11p_5 + 7$
    **end if**
  **end if**
**else**
  $0$
**end if**
```

6.9 Prototype Implementation of the Parametric Framework

A prototype of the parametric framework has been implemented in order to evaluate the approach.

6.9.1 Input Language

The input language for the prototype is a very simple language which translates into flow charts as defined in Chapter 3. The language has the following BNF grammar:

$$
\begin{aligned}
\texttt{Stmt} \rightarrow\; & \texttt{Var} := \texttt{Expr} \\
& \mid \textbf{If}\ \texttt{Expr}\ \textbf{then}\ \texttt{Stmt}\ \textbf{else}\ \texttt{Stmt} \\
& \mid \textbf{While}\ \texttt{BExpr}\ \textbf{do}\ \texttt{Stmt} \\
& \mid \textbf{Skip} \\
& \mid \texttt{Stmt}\ ;\ \texttt{Stmt} \\
\texttt{Expr} \rightarrow\; & \texttt{Num} * \texttt{Var} \\
& \mid \texttt{Num} * \texttt{Var} + \texttt{Expr}' \\
& \mid \mathbf{NL} \\
\texttt{BExpr} \rightarrow\; & \texttt{Expr} >= \mathbf{0} \mid \texttt{Expr} == \mathbf{0}
\end{aligned}
$$

where Num and Var are the syntactic categories for integers and variables respectively. The non-terminal $\texttt{Expr}'$ denotes Expr without the $\mathbf{NL}$ rule. Some syntactic sugar is layered on top of this, but the BNF grammar above illustrates the expressiveness of the input language. This structure is then translated into a flow chart data structure. Thus, the prototype does not consider pointers, arrays, structs or any other types than integers or Booleans. Function calls are supported but are analysed by inlining each function before translating into the BNF above. This means that recursion is not supported. Since the prototype is implemented for relational domains, which in most cases can only handle linear assignments and conditionals, the language in the prototype is restricted to linear conditionals and assignments only. That is all, arithmetic expressions in the language either have the form $\sum_{i=0}^{n-1} a_i v_i$ or the special value $\mathbf{NL}$ denoting a non-linear expression. Any non-linear value is mapped to the top value of the abstract domain.

6.9.2 Implemented Analyses

Since the computational tasks in Figure 6.1 (that is, the boxes) are quite modular, every green box is implemented as an independent program communicating the data (i.e., the ovals) through files. The prototype is using abstract interpretation with the polyhedral domain, using the Parma Polyhedra library [Par09]. The abstract interpretation was implemented in C++. Symbolic counting was implemented using the method outlined in [Pug94] adopted for convex polyhedra as described in Section 6.3.3. The symbolic counting was implemented in

Haskell. The prototype does not implement any low-level analysis, instead, it assumes that all program points has a worst case cost of ten clock cycles. The reason for this is that our focus has not been on low-level analysis, however, to use the framework in a realistic setting, a proper low-level analysis would be needed. Two different parametric calculations have been implemented, one of them is PIP (using the Piplib library [Pip09]). The other one is the Minimum Propagation Algorithm which is explained in Chapter 7. The prototype implements the SIMPLESLICE algorithm, which is a quick flow-insensitive slicing algorithm presented in [SEGL06].

6.9.3 Conclusion and Experiences

The prototype has provided experience with the parametric framework. Running the prototype on toy-examples and translated versions of some of the benchmarks in [MDH09] has provided the following observations:

- The method suggested in [Lis03a] can be implemented, mostly using existing code libraries to apply parametric WCET estimates on simple programs.

- The bottle-neck of the method is clearly the parametric calculation. This is because the parametric calculation essentially solves a set of concrete calculation problems. As an example: IPET is in general NP-complete, thus, a method such as PIP generalising it is naturally even more complex.

- PIP gives very large solutions, even for small programs (as indicated by Figure 6.3. Simplification is clearly needed. The composition part of the framework introduces, in general, a lot of redundancy. Thus, for practical use, simplification is needed both before parametric calculation and after. For this reason, we propose an alternative parametric calculation outlined in Chapter 7.

- PIP fails to produce solutions for larger programs, probably due to high complexity.

- The number of variables of the parametric calculation using PIP can be reduced by the technique outlined in Section 6.8. However, the number of symbolic parameters can not be reduced in the same way since they correspond to upper bounds. What can be done is to introduce one parameter for each basic block[2] and assign it to the minimum of all parameters in that basic block, and use it in the calculation. As an example, let $B = \{q_0, q_1, q_2\}$ be a basic block. Then we can use the parameter $p_B = \min\{p_0, p_1, p_2\}$ as substitute for p_0, p_1 and p_2. This has been implemented into the prototype, but it does not seem to have any significant impact on the complexity of the method.

[2]that is, a series of consecutive program points with a single entry and a single exit point, which is not containing any branches.

Chapter 7

The Minimum Propagation Algorithm

7.1 Introduction

Chapter 6 introduced a framework for parametric WCET analysis by computing two functions ECF_P and PCF_P independently of each other. The function PCF_P is obtained by parametric calculation. In Chapter 6 this was exemplified using parametric integer programming. Unfortunately, the complexity of parametric integer programming is very large and this turns out to be the bottleneck in the framework for the purposes of parametric calculation. Not only the computational complexity is a problem with PIP; the solutions obtained from PIP tends to grow exponentially in size and for large programs Piplib fails to deliver a solution. This chapter evaluates PIP and introduces an alternative algorithm for parametric calculation called *the minimum propagation algorithm* (MPA) introduced in [BEL09]. MPA scales much better than PIP in both computation time and solution size, as seen in Section 7.4.

7.2 The Minimum Propagation Algorithm

The Minimum Propagation Algorithm computes the function PCF_P of a program P given its flow chart and results from low-level analysis. Thus, it may operate as the *parametric calculation* phase as seen in Figure 6.1 in Chapter 6.

Being a parametric calculation, MPA computes the maximum of

$$\sum_{q \in \mathcal{Q}_P} c_q x_q$$

given the flow constraints and the symbolic upper bounds

$$\forall q \in \mathcal{Q}_P : x_q \leq p_q \tag{7.1}$$

The idea of the algorithm is to use the naïve, but correct upper bound

$$\mathrm{PCF}_P = \sum_{q \in \mathcal{Q}_P} c_q p_q \tag{7.2}$$

as basis, and then gradually improve the bound by propagating the flow constraints through the program. When flow chart edges are used as program points (as in our case), the following facts hold

- A program point can never be visited more times than the sum of its predecessors
- A program point can never be visited more times than the sum of its successors

or formally,

$$x_q \leq \sum_{q' \in \mathrm{pred}(q)} x_{q'} \tag{7.3}$$

$$x_q \leq \sum_{q' \in \mathrm{succ}(q)} x_{q'} \tag{7.4}$$

Note here that these are *inequalities*, since an incoming edge to a successor of a programming point may cause the successor to be visited more times than the program point (see Figure 7.1). Now, the execution count x_q for each program point have three upper bounds: (7.1), (7.3) and (7.4). Obviously the smallest one of these is the tightest and most desirable one. The idea of MPA is to use (7.2) as basis, but to substitute t_q for p_q where t_q is a upper bound computed from (7.1), (7.3) and (7.4). The upper bound t_q is computed by propagating the upper bounds through the graph and construct a tree which represents the upper bound.

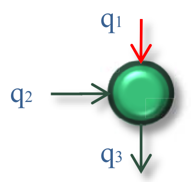


Figure 7.1: The program point q_3 can be visited more times than q_1 even though q_3 is the only successor of q_1

7.2.1 The Min-Tree

The upper bound for a program point needs to be valid for all possible combinations of symbolic execution counts $p_{q \in \mathcal{Q}}$. An upper bound t_q will be represented as a tree with three types of nodes: minimum nodes, plus nodes and leaf nodes. Minimum nodes (denoted $\Diamond$) express the minimum of all its children. Plus nodes (denoted $\oplus$) express the sum of all its children. Leaf nodes (denoted p_q) express the value of p_q. Such a tree will be referred to as a *Min-Tree*. Figure 7.2 depicts an example of a Min-Tree. This tree is in fact representing the upper bound of x_0 in L (see Figure 3.2, on page 18).

7.2.2 The Algorithm

MPA is shown in Algorithm 1. It is a recursive procedure which takes as argument a program point, a context and a set of constraints and returns a Min-Tree as described in previous section. The context is a set of visited program points for internal book keeping. The algorithm is always called with the empty set as context when used. The set of constraints corresponds to that of (7.3) and (7.4) and is obtained directly from the graph structure. The constraint (7.1) is implicit and is not needed in the constraint argument of the algorithm. MPA searches the given constraints and recursively builds a Min-Tree by adding visited nodes as children to a minimum node. It searches all simple paths first, leaving the branches for later. The branches are then recursively computed as children for plus nodes.

The root of the Min-Tree will always be a minimum node, and its children will be all maximum bounds found for the program point under analysis. MPA maintains a worklist and a branch set; the worklist keeps track of visited pro-

Algorithm 1 $\texttt{MPA}(q_i, \texttt{context}, \texttt{constraints})$

```
1: node ← $\texttt{mkMinNode}()$
2: worklist ← $\texttt{push}(\text{NIL}, i)$
3: branch ← $\varnothing$
4: while worklist $\neq$ NIL do
5:     $k$ ← $\texttt{peek}(\text{worklist})$
6:     worklist ← $\texttt{pop}(\text{worklist})$
7:     if $k \notin$ context then
8:         context ← context $\cup \{k\}$
9:         node ← $\texttt{addLeaf}(\text{node}, p_k)$
10:        for all $[x_k \leq x_j] \in$ constraints do
11:            if $j \notin$ context then
12:                worklist ← $\texttt{push}(\text{worklist}, j)$
13:            end if
14:        end for
15:        for all $[x_k \leq \sum_{n \in N} x_n] \in$ constraints such that $|N| \geq 2$ do
16:            if $N \cap$ context $= \varnothing$ then
17:                branch ← branch $\cup \{N\}$
18:            end if
19:        end for
20:    end if
21: end while
22: for all $N \in$ branch do
23:     plusNode ← $\texttt{mkPlusNode}()$
24:     for all $n \in N$ do
25:         child ← $\text{MPA}(n, \text{context}, \text{constraints})$
26:         plusNode ← $\texttt{addChild}(\text{plusNode}, \text{child})$
27:     end for
28:     node ← $\texttt{addChild}(\text{node}, \text{plusNode})$
29: end for
30: return node
```

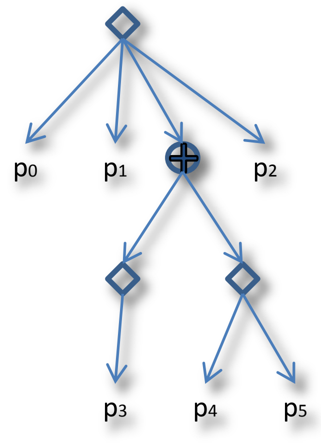


Figure 7.2: A Min-Tree representing the formula $\min(p_0, p_1, p_3 + \min(p_4, p_5), p_2)$.

gram points and the branch set keeps track of pending plus nodes. Whenever a program point has single predecessors and successors, the neighbouring capacities alone constitute as upper bounds for the program point and is therefore put in the worklist for continued processing. In the case of branching program points, the program points are put in the pending branch set for recursive processing as children of a plus node. The reason for this can be read directly from (7.3) and (7.4), where it is obvious that it is the *sum* of the upper bounds of the other program points that needs to be computed.

Detailed Explanation of the Algorithm

Row 1 creates the root of the tree which is always a minimum node, the primitive `mkMinNode` returns a minimum node without children. The Rows 2-3 initialise the worklist and the branch set. The worklist is implemented as a stack and using the stack primitives `push`, `pop` and `peek` (`peek` returns the top element of the stack, `pop` returns the stack with the top element removed) to manipulate it. The loop in rows 4-21 builds the leaves of the minimum node and puts the pending plus nodes in the branch set. Row 7 ensures that nodes which have already been considered (and thus don't contribute to any tighter result) are skipped. Row 9 adds leaves to the minimum node by using the primitive `addLeaf` which takes a node and a leaf and returns the node with the leaf added. Then, in rows 10-14, all single entry/exit constraints are added to the worklist for further processing. Rows 15-19 add the multiple entry/exit constraints to the pending branch set.

When no more program points are present in the worklist, the algorithm has added all leaves to the current min node, and enters the part of the algorithm which builds the plus nodes (row 22). By now, *node* is a minimum node, possibly with a couple of leaves, which are all maximum bounds on the program point i. In other words, the constraints from (7.1) have been added. Left to add are the plus nodes, which correspond to (7.3) and (7.4). This is done in rows 16-23. Each constraint which is corresponding to a branch in the program (i.e., a constraint which is a sum of program points) will produce a plus node (row 23), this is done by the primitive `mkPlusNode` which simply returns a plus node without children. The children of the plus node are then recursively computed from each term in the constraint (row 25), and then added as children to the plus node via the primitive `addChild` (row 26). Finally, each plus node is added as a child of the minimum node (row 28) and the root node is returned (row 30).

7.2.3 Example of MPA

Consider the example program L in Figure 3.2. We will show how to compute a Min-Tree for q_0. The set of constraints obtained from (7.1), (7.3) and (7.4) are the following

$$\begin{aligned}
\forall q \in \mathcal{Q}_L : x_q &\leq p_q \\
x_0 &\leq x_1 \\
x_1 &\leq x_2, x_0 \\
x_2 &\leq x_1 + x_5, x_3 + x_4 \\
x_3 &\leq x_2 \\
x_4 &\leq x_2, x_5 \\
x_5 &\leq x_4, x_2.
\end{aligned}$$

We start by calling $\mathrm{MPA}(q_0, \varnothing, \mathrm{constraints})$. Processing in row 4-21 will generate the following intermediate results:

$\text{analysis}(q_0, \varnothing, \text{constraints})$			
node	worklist	branch	context
min()	[0]	$\varnothing$	$\varnothing$
min(p_0)	[1]	$\varnothing$	$\{0\}$
min(p_0,p_1)	[2]	$\varnothing$	$\{0,1\}$
min(p_0,p_1,p_2)	[]	$\{\{3,4\}\}$	$\{0,1,2\}$

After the worklist has become empty and the main loop has finished, the algorithm is in row 22 and the plus nodes will be evaluated. We have that $N = \{3, 4\}$ and so this leads to two recursive calls: $\text{MPA}(q_3, \{0, 1, 2\}, \text{constraints})$ and $\text{MPA}(q_3, \{0, 1, 2\}, \text{constraints})$. The following tables show the intermediate results for these calls.

$\text{MPA}(q_3, \{0, 1, 2\}, \text{constraints})$			
node	worklist	branch	context
min()	[0]	$\varnothing$	$\{0,1,2\}$
min(p_3)	[]	$\varnothing$	$\{0,1,2,3\}$

$\text{MPA}(q_4, \{0, 1, 2\}, \text{MPA})$			
node	worklist	branch	context
min()	[4]	$\varnothing$	$\{0,1,2\}$
min(p_4)	[5]	$\varnothing$	$\{0,1,2,4\}$
min(p_4,p_5)	[]	$\varnothing$	$\{0,1,2,4,5\}$

The result of these two calls will both be children to a plus node, which in turn will be child to the minimum node that will be returned from the original call. This plus node is then added as child to the previous minimum node. The final Min-Tree for q_0 expresses:

$$\min(p_0, p_1, p_2, \min(p_3) + \min(p_4, p_5))).$$

Computing the Min-Tree t_q for all program points $q \in \mathcal{Q}_L$ results in

$$\begin{aligned}
t_0 &= \min(p_0, p_1, p_3 + \min(p_4, p_5), p_2) \\
t_1 &= \min(p_0, p_1, p_3 + \min(p_4, p_5), p_2) \\
t_2 &= \min(p_2, \min(p_0, p_1) + \min(p_4, p_5), p_3 + \min(p_4, p_5)) \\
t_3 &= \min(p_2, p_3, \min(p_0, p_1) + \min(p_4, p_5)) \\
t_4 &= \min(p_2, p_4, p_5) \\
t_5 &= \min(p_2, p_4, p_5).
\end{aligned}$$

The function PCF_L is then computed by taking

$$\mathrm{PCF}_L = \sum_{q \in \mathcal{Q}_L} c_q t_q,$$

where the atomic WCETs **c** are taken from (3.1) on page 22.

$$\begin{gathered}
\mathrm{PCF}_L = \lambda p_0, p_1, p_2, p_3, p_4, p_5. \\
\min(p_0, p_1, p_3 + \min(p_4, p_5), p_2) \\
+3(\min(p_0, p_1, p_3 + \min(p_4, p_5), p_2)) \\
+\min(p_2, \min(p_0, p_1) + \min(p_4, p_5), p_3 + \min(p_4, p_5)) \\
+2(\min(p_2, p_3, \min(p_0, p_1) + \min(p_4, p_5)) \\
+2(\min(p_2, p_4, p_5)) \\
+8(\min(p_2, p_4, p_5)).
\end{gathered} \tag{7.5}$$

Computing $\mathrm{PCF}_L \circ \mathrm{ECF}_L$ (by substituting execution counts for symbolic parameters) will result in

$$\begin{gathered}
\mathrm{PWCET}_L = \lambda n. \\
\min(1, 1 + (n \geq 0 \; ? \; n+1 \; : \; 0), \infty) \\
+3(\min(1, 1 + (n \geq 0 \; ? \; n+1 \; : \; 0), \infty)) \\
+\min(\infty, 1 + (n \geq 0 \; ? \; n+1 \; : \; 0)) \\
+2(\min(\infty, 1, 1 + (n \geq 0 \; ? \; n+1 \; : \; 0))) \\
+2(\min(\infty, (n \geq 0 \; ? \; n+1 \; : \; 0))) \\
+8(\min(\infty, (n \geq 0 \; ? \; n+1 \; : \; 0))) \\
= \lambda n. \\
4(\min(1, 1 + (n \geq 0 \; ? \; n+1 \; : \; 0))) \\
+1 + (n \geq 0 \; ? \; n+1 \; : \; 0) \\
+2 + 10(n \geq 0 \; ? \; n+1 \; : \; 0) \\
= \lambda n. 7 + 11(n \geq 0 \; ? \; n+1 \; : \; 0) \\
= \lambda n. \begin{cases} 18 + 11n & \text{if } n \geq 0 \\ 7 & \text{otherwise.} \end{cases}
\end{gathered}$$

which equals PWCET_L obtained by PIP.

7.3 Properties of MPA

This section investigates different properties of MPA. In particular, the algorithm is proven to terminate and to be correct. A bound on the complexity of the algorithm is also given. MPA contains five loops referred to as $L_4, L_{10}, L_{15}, L_{22}$ and L_{24}, where the subscript is the row number of the loop header in Algorithm 1.

7.3.1 Termination

In order to prove that MPA terminates, we show that the recursion and all five loops of MPA terminate. First, we see that the loops L_{10} and L_{15} terminate because they iterate over stable and finite sets. For the same reason, the loops L_{22} and L_{24} terminate *provided* that the recursive calls to MPA all terminate. The fact that *branch* is a finite set is a consequence of that L_4 terminates, which is shown below. This means that L_4 executes a finite number of times and thus adding a finite number of elements to *branch*.

In each iteration of L_4 exactly one of the following holds:

1. The set *context* will contain one more element than in the previous iteration.

2. The list *worklist* will contain one less element than in the previous iteration.

Row 7 will execute in every iteration of L_4. If the conditional evaluates to *true*, then k was not part of *context* and will on row 8 be added. Thus, *context* contains one more element than the previous iteration. No more elements are added to *context* in the loop. If the conditional in row 7 evaluates to *false*, no elements will be added to *worklist* (row 12 can not execute). Since row 6 will be executed in any case, the list *worklist* will contain one element less.

By definition, *context* $\subseteq \mathcal{Q}_P$, and if $\mathrm{context}_i = \mathcal{Q}_P$, then L_4 would terminate immediately. This is because the conditional on row 7 will evaluate to false and continue to loop L_{22}. Now, $\mathrm{branch} = \varnothing$, since row 3 was the last assignment of branch. Thus, L_{22} will terminate immediately and L_{24} will never execute. Now, assume for contradiction that there exists an input $I = (q_i, \mathrm{context}_i, \mathrm{constraints}_i)$ such that L_4 does not terminate. Thus, $\mathrm{context}_i \subset \mathcal{Q}_P$. Statement 1 in the list above may only occur a finite number of times since $\mathcal{Q}_P$ is a finite set. This means that there exists an infinite sequence of iterations such that statement 2 occur. However, $\mathrm{worklist}$ is a finite set (since a finite number of elements can be added to it a finite number of times), meaning that $\mathrm{worklist}$ in this infinite sequence of iterations will become empty, but the conditional on row 4 terminates L_4 as $\mathrm{worklist}$ is empty. This contradicts that there exists an input such that L_4 never terminates.

Left to prove is that the recursive calls to MPA eventually terminate. If MPA is called with $\mathrm{context}_i \subset \mathcal{Q}_P$ as input, then MPA may recursively call MPA a finite number of times with $\mathrm{context}_j$ as input. We want to show that for every recursive call $\mathrm{context}_i \subset \mathrm{context}_j$ (i.e., the recursive calls to MPA are called with a context with at least one more element than the current call), which would imply that in a finite number of steps MPA will either have terminated or will be called with $\mathrm{context}_j = \mathcal{Q}_P$, which also means that MPA will terminate. First, note that row 25 is executed only if N is non-empty. The set N is non-empty only if branch is non-empty. Finally, branch is non-empty only if row 17 executes, when the conditional on row 7 evaluates to true, in which case row 8 executes, meaning that $\mathrm{context}$ has at least one more element than before the call (since k was not previously a member). This proves that Algorithm 1 terminates.

7.3.2 Complexity

The complexity of MPA involves a lot of factors. In fact the complexity depends more on the structure of the input program than on the actual size of the program (i.e., the number of program points). While it is possible to derive a worst-case complexity of MPA in terms of the number of program points of a program, it would not be a very useful one. To investigate the complexity of MPA we take an informal alternative approach in which we investigate the behaviour of MPA rather than the algorithm itself. All primitives used in MPA can be implemented so that they take constant time. The first thing that needs to be said about the complexity of MPA is that L_{10} and L_{15} actually do not need to be implemented as loops. This is because there are maximum two constraints associated with a single program point (incoming and outgoing edges). Thus, the constraints can be stored in such a way that each program point can access two constraints, meaning that it is not necessary to loop through all constraints in L_{10} and L_{15}. Consequently, we consider L_{10} and L_{15} to be $O(1)$[1].

Now, the loop L_4 iterates through all program points found in any non-branching path in both directions (row 10-12). All branching paths are put into `branch`, which then recursively call MPA for every branching path (row 25), in a recursive call a previously explored path will not be explored again, since previously explored edges will be stored in `context`. However, an edge may be explored several times in different recursive calls of MPA. In summary, this means that MPA will explore every non-cyclic path of the CFG. Thus, the complexity of MPA is directly proportional to the number of non-cyclic paths in the analysed program.

7.3.3 Correctness of MPA

In this section we will prove that MPA is correct. By correct we mean that edge q is guaranteed to be visited less than or equal to the expression represented by the Min-Tree returned by $\mathrm{MPA}(q, \varnothing, \mathrm{constraints})$. This will be proven by induction over the depth of the MPA tree.

Proposition 11. *Let q be a program point $q \in \mathcal{Q}_P$ and let t be a Min-Tree produced by calling Algorithm 1 for program point q with an empty context. Assume that t consists only of a set of leaf nodes L_t and no plus nodes. Then t is a correct upper bound of the number of times q is visited.*

[1]although the arrangement of constraints would require $O(n)$ of memory, where n is the size of $\mathcal{Q}_P$

The first level of the Min-Tree is the tree produced before any recursive calls to MPA is performed. Thus, a one-level Min-Tree consists only in a Min-Node and a set of leaf nodes.

Proof. All leaf nodes are added by row 9 of Algorithm 1. The first node added is the constraint $x_q \leq p_q$ which is directly taken from (7.1). Then, the constraints $x_q \leq p_r$ where r is any program point which may be in the worklist. The program points that may be in the worklist are the neighbours to q (including q itself) which *must* be or myst have been visited when q is visited (see row 10). All nodes in this set of program points must be visited the same number of times, say m. Since all nodes in this set must be visited every time q is visited, the least capacity of these nodes constitutes maximum bound on m. □

This means that it is safe to terminate the algorithm before any recursive calls. However, to reach a potentially tighter result, the recursive calls may contribute to a tighter, yet still correct bound.

Proposition 12. *Let q be a program point $q \in \mathcal{Q}_P$ and let t be the Min-Tree produced by calling Algorithm 1 for program point q with an empty context. Assume that t consists of a (possibly empty) set of leaf nodes L_t, and a set of plus nodes P_t, where each plus node $p \in P_t$ has a set of n-level Min-Trees. Then, if all n-level Min-Trees are correct (that is, they constitute correct upper bounds of the program points they represent), then t is an $n+1$-level Min-Tree representing a correct upper bound of the number of times q is visited.*

Proof. First of all, the minimum of the set of leaf nodes L_t is a correct upper bound for q as stated by Proposition 11. Now, every set of program points N in the branch set (see row 17) represents a selection of edges in the CFG, that is, exactly one of the program points in N will be taken for every time program point q is visited. This means that the sum of all upper bounds of the program points in N is an upper bound also on the number of times q can be visited. Assuming that all n-level Min-Trees produced by MPA are correct, this also corresponds to a correct upper bound of the number of times q is visited. The proposition holds since the minimum of a set of correct upper bounds (i.e., the bound derived by the leaf nodes and the bound derived by the plus nodes) are again a correct upper bound. □

Proposition 11 and Proposition 12 together proves that MPA produces correct bounds.

7.3.4 Upper Bounds on Tree Depth

An important consequence of the arguments in Section 7.3.3 is that every level of the Min-Tree is a *safe* upper bound of the execution time of a program point. This means that it is safe to skip the computation of any subtree in the Min-Tree, although it may result in a less precise sub tree. Thus, it is possible to set an upper bound on the depth of the produced Min-Trees to ensure a faster termination of the MPA algorithm, to the cost of possible precision loss. However, the deeper a node is in a Min-Tree, the less likely it is to actually contribute to a tighter upper bound. This is because subtrees are children to plus nodes, which in turn probably will give a larger bound than the children of Min-nodes, and since the root node is always a Min-node, the larger nodes will not contribute to the final solution.

To summarise: the first levels in the Min-Tree are the nodes most likely to contribute to the final results. Thus, computing very deep sub trees will in many cases be a waste of computation time. In Section 7.4.3, we will show setting an upper bound of the depth of Min-Trees affects computation time and precision.

7.4 Evaluation

In this section we evaluate MPA in two ways. First, execution time and solution size is compared to that of PIP. This is done by running the two approaches on the prototype described in Chapter 6. Since the input language to this prototype is somewhat limited (requiring source code to be translated by hand), the scalability of MPA is also evaluated by running it in isolation on a larger set of benchmarks. This is possible since parametric calculation can be performed over the structure (control flow graph) and does not need the actual code.

7.4.1 Comparison with PIP

Experiment Set-up

The experiments are run under Windows XP Professional SP3 on an Intel core duo 2.4 GHz with 2.39GB RAM and a 6MB L2-cache. Both Piplib and a C++ implementation of MPA were compiled with GCC 3.4.4 under Cygwin. Since the prototype tool lacks a proper low-level analysis, all program points are assumed to have a constant WCET of 10 clock cycles in this evaluation. The experiments have been performed by analysing some benchmarks using both

Benchmark	#Pps	PIP		MPA	
		Time	Size	Time	Size
edn/fir	11	0.1s	3	0.1s	1
edn/latsynth	7	0.1s	1	0.1s	1
edn/latsynth x2	12	0.1s	2	0.1s	1
edn/latsynth x4	25	0.1s	10	0.1s	3
cnt/initialize	12	0.1s	3	0.1s	1
cnt/initialize x2	23	0.2s	83	0.1s	3
cnt/initialize x3	34	2.6s	1782	0.1s	6
cnt/sum	16	0.3s	80	0.1s	2
cnt/sum x2	31	-	-	0.1s	5
jcomplex	23	-	-	0.1s	7
matmult/Initialize	12	0.1s	3	0.1s	1
matmult/Initialize x2	23	0.3s	83	0.1s	3

Table 7.1: Test results

PIP and MPA. These benchmarks have been manually translated to the simplified analysed language. After analysis, some sample points in the parameters have been chosen and instantiated. Table 7.1 shows the evaluation, and the columns are explained below.

Benchmark

The benchmarks are taken from the Mälardalen benchmarks [MDH09]. These benchmarks are standard WCET benchmarks and are common to use in the field of WCET analysis. We have chosen benchmarks that conform to the limitation of the prototype and which have a timing behaviour which is parametric in some variables or constant macros. One function at a time has been analysed and the name of the function is given as second name in the benchmark column. When a benchmark is marked with x2, x3 etc, it means that the particular function has been called repeatedly and is thus inlined multiple times. This is just to see how the PIP and MPA scale with the number of program points.

Program points

Labelled as *#Pps* in the table. This is the number of arcs in the flow chart.

Execution time

The execution time of the running algorithms. The cases where the time is not given means that Piplib failed to solve the problem due to a too high complexity of the solution. All times have been obtained by the UNIX command `time`.

Size

The size of the solution, given in KB. The measurements comes from the file sizes of the solutions textual representations. Note that Piplib does not scale well, especially not in solution size.

7.4.2 Evaluation of Precision

The precision of MPA compared to PIP is hard to measure since the solutions of MPA and PIP looks so different. The precision has been compared by evaluating PCF_P for some chosen values for some chosen input parameters. As an example, `jcomplex` has been evaluated by choosing instances of the input parameters a and b, and compute a vector of upper bounds by $\mathrm{ECF}_{\texttt{jcomplex}}(a, b)$. The resulting vector of upper bounds has been used as arguments to $\mathrm{PCF}_{\texttt{jcomplex}}$ to derive a concrete WCET. Table 7.2 shows the estimated WCETs of instantiated variables from the two parametric calculation methods. The input parameters have been chosen so they have a parametric behaviour and are instantiated with values somewhat close to their original values in the benchmark programs. The last two columns shows how much the MPA solution differs from the PIP solution in that particular instantiation. As can be seen, MPA gives slightly less precise result compared to PIP. An imprecision of up to 32.3% has been observed (on cnt/sum), but in most cases it is less than one percent.

7.4.3 Evaluation of Upper Bounds on Min-Tree Depth

As mentioned in Section 7.3.4, the first levels of a Min-Tree are the ones most likely to contribute to the final solution. This can be demonstrated by running MPA with different max-depths of the Min-Trees. Interestingly enough, for most of the evaluated programs, it is sufficient to have a maximum depth of one to achieve the precision presented in Table 7.2. To be precise, the following programs gives the same precision as in Table 7.2 when the maximum depth is set to one: edn/fir, edn/latsynth, edn/latsynth x2, edn/latsynth x4, cnt/initialize, cnt/sum and matmult/initialize.

Benchmark	Parameters	PIP result	MPA result	Diff	Percent
edn/fir	N = 100, ORDER = 25	60790	60810	20	0.03%
	N = 100, ORDER = 50	78040	78060	20	0.03%
	N = 100, ORDER = 75	57790	57810	20	0.03%
	N = 200, ORDER = 25	141790	141810	20	0.01%
	N = 200, ORDER = 50	234040	234060	20	<0.01%
	N = 200, ORDER = 75	288790	288810	20	<0.01%
	N = 300, ORDER = 25	222790	222810	20	<0.01%
	N = 300, ORDER = 50	390040	390060	20	<0.01%
	N = 300, ORDER = 75	519790	519810	20	<0.01%
edn/latsynth	n = 50	1520	1520	0	0%
	n =100	3020	3020	0	0%
	n =200	6020	6020	0	0%
edn x2	n = 50	3030	3060	30	0.99%
	n = 100	6030	6060	30	0.5%
	n = 200	12030	12060	30	0.25%
edn x4	n = 50	6050	6160	110	1.82%
	n =100	12050	12160	110	0.91%
	n =200	24050	24160	110	0.46%
cnt/initialize	MAXSIZE=10	4640	4660	20	0.4%
	MAXSIZE=20	17240	17260	20	0.1%
	MAXSIZE=30	37840	37860	20	0.05%
cnt x2	MAXSIZE=10	9270	9810	540	5.83%
	MAXSIZE=20	34470	35510	1040	3.02%
	MAXSIZE=30	75670	77210	1540	2.04%
cnt x3	MAXSIZE=10	13900	15460	1560	11.22%
	MAXSIZE=20	51700	54760	3060	5.92 %
	MAXSIZE=30	113500	118060	4560	4.02%
cnt/sum	MAXSIZE=10	6640	8660	2020	30.4%
	MAXSIZE=20	25240	33260	8020	31.8%
	MAXSIZE=30	55840	73860	18020	32.3%
cnt x2	MAXSIZE=10	-	17810	-	-
	MAXSIZE=20	-	67510	-	-
	MAXSIZE=30	-	149210	-	-
jcomplex	a = 1, b = 1	-	80	-	-
	a = 1, b = 15	-	120	-	-
	a = 1, b = 30	-	110	-	-
	a = 15, b = 1	-	80	-	-
	a = 15, b = 15	-	80	-	-
	a = 15, b = 30	-	30	-	-
	a = 30, b = 1	-	80	-	-
	a = 30, b = 15	-	80	-	-
	a = 30, b = 30	-	30	-	-
matmult/Initialize	UPPERLIMIT = 100	406040	406060	20	<0.01%
	UPPERLIMIT = 150	909040	909060	20	<0.01%
	UPPERLIMIT = 200	1612040	1612060	20	<0.01%
matmult x2	UPPERLIMIT = 100	812070	817110	5040	0.62%
	UPPERLIMIT = 150	1818070	1825610	7540	0.41%
	UPPERLIMIT = 200	3224070	3234110	10040	0.31%

Table 7.2: Precision Comparison

Benchmark	Parameters	Result with max-depth			
		1	2	3	4
cnt/initialize x2	MAXSIZE = 10	∞	9810	9810	9810
cnt/initialize x3	MAXSIZE = 10	∞	17840	15460	15460
cnt/sum x2	MAXSIZE = 10	∞	17810	17810	17810
jcomplex	a=30,b=30	90	80	70	30
matmult/initialize x2	UPPERLIMIT=100	∞	817110	817110	817110

Table 7.3: Comparison of precision with different max-depths

The rest of the benchmarks lost precision when the depth was set to one. Table 7.3 shows how the precision changes when changing the max depth. As can be seen, in all tested programs, the best possible precision can be achieved with the maximum depth set to 4, but in most cases it is sufficient to set it even lower. Note that in these small examples, the execution time of MPA is still neglectable, so there are no obvious benefits on setting a maximum depth for them. However, in larger programs, as will be seen in Section 7.4, the benefits are obvious.

7.4.4 Scaling Properties

Since the translated benchmarks used in previous section are small, they don't show the scaling properties of MPA properly. In order to investigate how MPA scales in more realistic cases, we have run the algorithm in isolation (independent of the parametric framework and the prototype) on the full benchmark suite of [MDH09]. We have used the WCET analysis research prototype SWEET [EG97, WCE09] to obtain control flow graphs for the benchmarks. A control flow graph (CFG) is a graph where each node is a basic block[2]. Note that the MPA is equally valid on control flow graph as on a flow chart, since the premises of the constraints are the same. The CFGs obtained from SWEET are on the *full programs*, that is, it includes all functions and all function calls. In contrast to the evaluation in Section 7.4, the CFGs obtained from SWEET are not inlined; each function call is an edge from the caller to the callee, and each return is an edge from the exit of a function, back to the caller. Since the algorithm in this experiment is not run on the full parametric framework, we cannot examine the precision of MPA in this test, just how resource consuming it is.

[2]that is, a sequence of consecutive instructions ending in a jump

Benchmark	#Pps	MD	Iterations	Calls	Time	Size
adpcm	884	4	-	-	-	-
bs	39	2	249	1538	0.04s	6633
bsort	66	2	1750	9610	0.14s	44152
cnt	93	2	1537	11547	0.16s	48022
cover	1593	121	-	-	-	-
compress	380	5	-	-	-	-
crc	127	2	10543	69017	0.85s	316212
duff	390	9	5937	121369	1.22s	475556
edn	342	2	7202	95585	1.04s	421438
expint	88	2	1028	7983	0.11s	32407
fac	36	2	260	1298	0.08s	5959
fdct	147	2	973	21565	0.22s	79457
fft1	266	4	72572	482486	6.18s	2390541
fibcall	29	2	75	702	0.03s	2567
fir	77	2	779	6828	0.10s	27315
insertsort	39	2	175	1313	0.03s	5310
jcomplex	48	2	792	3289	0.06s	16763
jfdctint	122	2	1038	14726	0.16s	55563
lcdnum	158	17	4042	40164	0.46s	168927
lms	262	4	90864	586176	7.44s	2959463
ludcmp	181	3	5583	35101	0.47s	169103
matmult	97	2	1351	9441	0.13s	40263
minmax	109	3	1926	18881	0.23s	74619
ndes	445	9	1235359	11593218	2m19s	54938649
ns	46	2	562	2838	0.06s	13245
nsichneu	3313	5	-	-	-	-
prime	114	3	11425	79060	0.95s	356992
qsort-exam	153	2	15861	104870	1.30s	501762
qurt	135	4	27658	178578	2.17s	821808
select	136	2	32418	165320	2.25s	842275
sqrt	49	3	896	4717	0.08s	21758
statemate	1287	47	-	-	-	-
ud	150	2	2770	15938	0.23s	78277

Table 7.4: Scalability Properties of MPA

Benchmark	Execution Time of MPA with Min-Tree max-depth				
	2	3	4	5	6
adpcm	3s	17s	30s	53s	-
compress	0.8s	1.6s	3s	4s	7s
cover	1m9s	1m18s	-	-	-
statemate	11s	36s	1m11s	1m16s	-
nsichneu	-	-	-	-	-

Table 7.5: scaling properties with max-depth

Table 7.4 shows the result of the tests. The first column is the benchmark name, second column (#Pps) is the number of program points. The third column (MD) is the maximum degree of a node, i.e. the maximum number of outgoing or incoming edges from a node. As seen in Table 7.4 this property strongly affects the time consumed by MPA. The fourth column (iterations) is the global number of iterations of MPAs main loop (rows 4-21 in Algorithm 1). The fifth column (Calls) is the global number of calls (including recursive calls) to Algorithm 1. The sixth column (Time) is the real time of the algorithm running, obtained by the UNIX command *time*. Finally, the seventh column (Size) is the size of the solution file in bytes.

Note that MPA runs without imposing a limit on the Min-Tree depth. This caused five of the programs to fail the analysis. The reason seems to be the combination of many program points and a high vertex degree on the nodes, resulting in a high number of recursive branches. However, by imposing an upper bound on the depth of the produced Min-Trees, most of these programs can be analysed. Since we do not run the whole framework in these tests, we are not able to see how much precision is lost from doing so, but the tests in the previous section indicated that a maximum depth of four was enough for small benchmarks. For the five programs that failed the unbounded Min-Tree depth we have analysed them with different upper bounds to see how they would scale. The result is shown in Table 7.5. The program ”nsichneu” fails to be analysed even for maximum depth 2 and is the only benchmark which completely fails to be analysed. As seen, the other benchmarks can be analysed, but may or may not be over-approximated.

7.5 The Reason for Over-Estimation

As seen in Table 7.2, MPA often over-estimates the results compared to Pip. This occurs in the cases where MPA fails to derive the tightest possible bounds for a Min-Tree. There are cases where the bounds (7.3), (7.4) and (7.1) are not sufficient to express the tightest possible bound. Consider Figure 7.3 which depicts a nested loop (note that MPA is not concerned with the semantics of the program, so the boxes are intentionally left empty). The Min-Tree of q_7 and q_8 (they are equal) is shown in Figure 7.4. By inspection of Figure 7.3 we can see that the execution of q_3 implies execution of q_7 and q_8, thus p_3 is an upper bound of q_7 (since, by (7.1), q_3 can not execute more than p_3 times, which implies that neither can q_7). As seen in Figure 7.4, MPA does not derive p_3 as an upper bound, instead, the bounds $p_3 + \text{MIN}(p_5, p_6)$ and $p_3 + p_2$ are derived. This is because q_3 is one out of two outgoing edges from q_2 as well as one out of two incoming edges for q_4, giving rise to the constraints $x_4 \leq x_3 + x_6$ (by (7.3)) and $x_1 \leq x_2 + x_3$ (by (7.4)). In summary, the upper bound p_3 cannot be determined for q_7 only by the constraints (7.3),(7.4) and (7.1) only, and hence MPA fails to find it.

As an example, we will show that for a certain instantiation of the symbolic bounds $p_0, ..., p_8$, that none of the bounds (i.e., branches) in Figure 7.4 are sufficiently tight. Assume that the symbolic bounds are instantiated as follows:

$$p_0, ..., p_8 = [1, \infty, 1, 1, \infty, 3, 3, \infty, \infty].$$

Again, by inspection of Figure 7.3 we can conclude that q_7 can maximally be executed once, since $v_3 \leq 1$. But the bounds derived from Figure 7.4 are (from left to right): $p_3 + \text{MIN}(p_5, p_6) = 1 + \text{MIN}(3, 3) = 4$, $p_1 = \infty$, $p_4 = \infty$, $p_7 = \infty$, $p_8 = \infty$ and $p_3 + p_2 = 1 + 1 = 2$. Thus, the tightest bound (the minimum of the above) given from the Min-Tree is 2, which clearly is an over-approximation.

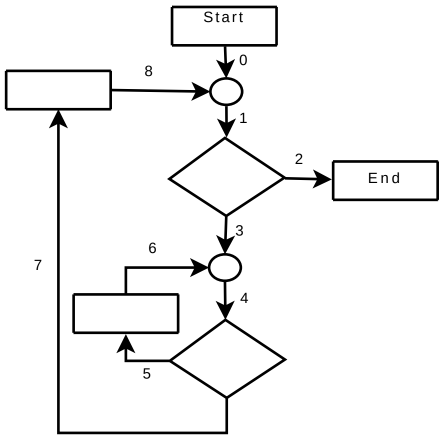


Figure 7.3: A program causing over-estimation in MPA

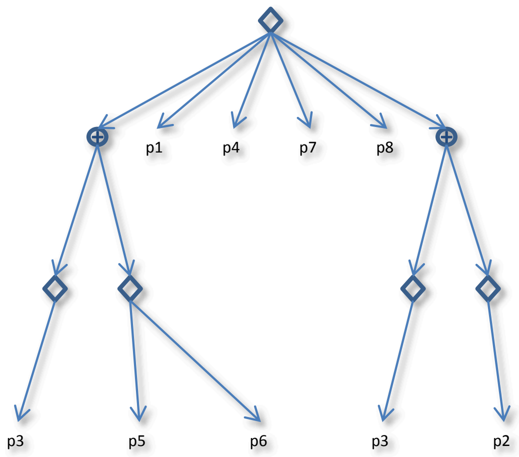


Figure 7.4: The Min-Tree for the program points q_7 and q_8.

Chapter 8

Summary, Conclusions and Future Work

8.1 Summary

In this thesis we have suggested a modular framework for static WCET analysis. The framework is based on the idea of counting semantic states of a program to be able to estimate a tight and correct upper bound of the WCET of a program. Two possible applications of this framework has been presented; one is to quickly and efficiently calculating loop bounds, and the perhaps most useful application: a method of deriving a parametric WCET estimate. The framework presented is based on results presented in previous publications by other authors, but this thesis contributes with important research on how to make the methods practically useful.

8.1.1 Contributions

As summary of the thesis, the individual contributions of this thesis are listed in detail. The following sections correspond to the list of contributions given in Chapter 1 on page 5.

Formalised Framework

The framework of the parametric WCET analysis proposed in [Lis03a, Lis03b] has been investigated, refined and formalised, though there are still some aspects presented in [Lis03b], which has not been investigated (such as using the method to find infeasible paths). The method inspired the publication [ESG$^+$07] which is based on similar techniques, and hence incorporated as a framework for finding loop bounds.

Research Prototype

The research presented in this thesis has been based on a prototype tool implementing the parametric framework. The tool has provided practical experience with the parametric framework and has helped to discover the potentials and bottle-necks of the framework, as well as made it possible to evaluate the approach.

Simplification of the Parametric Framework

In addition to the prototype, some research results considering practical issues such as how to implement Pugh's method for element counting and a method for reducing the number of variables in a IPET problem have been presented.

The Congruence Domain

The congruence abstract domain which is an integral part of the loop bound analysis has been modified to be able to perform analysis on realistic low-level or intermediate level code. This by introducing abstractions over bit strings and introducing abstract low-level operations.

The Minimum Propagation Algorithm

Perhaps the most important contribution of the thesis is the Minimum Propagation Algorithm which is intended to replace Parametric Integer Programming as parametric calculation. The evaluation of the algorithm indicates that the algorithm can be practically used for larger programs compared to PIP, and thus be more useful for realistic analysis.

Evaluation of Algorithms

The general framework have been evaluated through the research prototype. This shows that the framework is possible to use and that it can analyse programs correctly. Furthermore, the PIP and MPA algorithms have been evaluated on a larger set of benchmarks.

8.2 Future Work

8.2.1 Full Evaluation

Future work includes a full evaluation of the parametric framework on more realistic benchmark programs. A final evaluation on real industrial code should be able to give strong indications of the usefulness of the proposed approach. The evaluation is planned to be done by implementing the framework into the research prototype SWEET [GESL07, WCE09], which can analyse C programs which has been compiled to an intermediate format.

8.2.2 The Minimum Propagation Algorithm

While the minimum propagation algorithm works quite well in practice, there are still things that can be done to improve it. For instance, more (memory) efficient data types could possibly be invented. In addition, we plan to investigate the possibility of adding additional constraints (such as infeasible path information) to MPA. Finally, future work is to see if the over-estimation can be reduced.

8.2.3 Abstract Domains

While the congruence domain has been investigated, other domains, and in particular relational domains, are presented on the same assumptions as the congruence domain in literature. For relational domains, the issues as those presented in Chapter 5 will have to be solved, such as handling of finite value domains, signed and unsigned integers, as well as abstract bit-operations. But another problem arises in relational domains, namely, what should be treated as variables (dimensions)? On low-level code, the memory model may not be divided clearly into ”variables”, but as registers, memory positions etc. Further research is needed to practically be able to use relational abstract interpretation on low-level code.

8.2.4 Modifications to the Parametric Framework

The resulting function PWCET_P of a program is meant to be stored as a final result, and should ideally be instantiated at run-time when values of input variables are known. However, to avoid the bottle-neck of the framework, which seems to be the parametric calculation, an alternative approach would be to not compute PCF_P, but to use the result from ECF_P to generate a regular ILP-problem for every instance. The result from ECF_P is a concrete vector $\mathbf{p}$ (instead of a symbolic one), which can be used to generate an ILP-problem based on the structural constraints and the concrete values in $\mathbf{p}$. In this way, the computation of PCF_P can be skipped. Thus, the function PWCET_P will not be computed, but an ILP-problem can be obtained for any instantiation of the input variables. This means that the ILP problem has to be solved to get a concrete WCET estimation, but due to fast solvers this should be fairly efficient (if not efficient enough to instantiate at run-time). This approach would then make the time for instantiation longer, while reducing the time for analysis. Future work is to evaluate this approach and compare it to the outlined one.

8.3 Conclusions

This section will summarise and conclude the experience and impact of the research results found in this thesis.

8.3.1 Parametric WCET Analysis is Possible

The prototype tool presented in Chapter 6 implements the framework formalised in the thesis and most of the methods suggested in [Lis03a] and [Lis03b]. Its main drawback is currently its input language which prevents analysis on large programs. However, the implementation shows that the approach is feasible and that it indeed can produce correct parametric WCET estimates of programs in reasonable time. With the MPA algorithm we believe that parametric WCET analysis is possible to perform on smaller program parts (such as ”disable interrupt”-sections or small embedded system components). We have discovered that the two functions used to compute a parametric WCET estimate, PCF_P and ECF_P are independent of each other and can be used in isolation if needed.

One of the most powerful properties of the proposed framework is that it is general; it is based on the language-independent and program structure-independent abstract interpretation. Furthermore, the proposed framework has a modular work flow which means that most of the individual analyses can be replaced and customised. For instance, the framework is not restricted to a certain abstract domain, calculation method or low-level analysis.

8.3.2 Parametric Calculation is Complex

The most complex and time consuming part of the framework seems to be the parametric calculation. This is because a parametric calculation essentially tries to solve multiple concrete calculation problems. Since the most popular regular calculation methods proposed already have a high complexity, this is not surprising. Thus, the most important results in this thesis is the suggestions to make the parametric calculation more efficient. The following three points apply to parametric calculation in general and can be used for PIP as well as MPA:

- Reduce the number of parameters in the parametric calculation

This was briefly mentioned in Section 6.9.3. The parameters of a parametric calculation problem can be classified into equivalence classes so that a single parameter per basic block can model the set of parameters in the basic block. This typically reduces the number of parameters approximately one third.

- Exploit the results from ECF_P in the parametric calculation

As said in Section 8.2, in the final WCET formula, the bounds computed by ECF_P is substituted for the symbolic bounds of PCF_P (that is, the arguments of PCF_P) after it has been computed. However, some of the substituted parameters may be constant values or unbounded, some of them may also be equal to each other. This can be used to reduce the number of parameters and constraints of the resulting parametric calculation problem. While the benefit of this has not been investigated, it could in combination with the other approaches further simplify the parametric calculation.

- Skip the parametric calculation

Also mentioned in Section 8.2 was that the step of computing PCF_P can be skipped in favour of producing concrete IPET problems as output rather than a parametric formula. However, this may make it impossible to instantiate formulae at run-time and no concrete formula estimating the WCET will be produced.

The last point on how to simplify parametric calculation only applies to PIP:

- Reduce the number of variables in the PIP problem

As shown in Section 6.8, the number of variables of the problem can be reduced significantly. While this simplifies the problem, it does not actually reduce the asymptotic behaviour of the method.

8.3.3 The Minimum Propagation Algorithm Scales

As seen in Section 7.4, PIP does not seem to scale well, even though some of the simplifications in the previous sections have been applied. In the comparison to PIP, the MPA algorithm scales much better. In addition, MPA provides a trade-off in that it can provide correct but possibly less precise results by imposing an upper bound on the depths of the produced Min-Trees. With this possibility MPA is able to analyse all benchmarks from [MDH09] but one.

Bibliography

[AH87] S. Abramsky and C. Hankin. Introduction to abstract interpretation. In S.Abramsky and C. Hankin, editors, *Abstract Interpretation for Declarative Languages*, pages 9–31. Ellis Horwood, 1987.

[AHLW08] Sebastian Altmeyer, Christian Hümbert, Björn Lisper, and Reinhard Wilhelm. Parametric timing analysis for complex architectures. In *Proc. 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 08)*, Kaohsiung, Taiwan, August 2008.

[AJ94] Samson Abramsky and Achim Jung. Domain theory. In *Handbook of Logic in Computer Science*, pages 1–168. Clarendon Press, 1994.

[Alt96] P. Altenbernd. On the false path problem in hard real-time programs. In *EUROMICRO96*, pages 102–107, June 1996.

[Apr09] Apron website, 2009. `apron.cri.ensmp.fr/library/`.

[APT00] H. Aljifri, A. Pons, and M. Tapia. Tighten the computation of worst-case execution-time by detecting feasible paths. In *IPCCC2000*. IEEE, February 2000.

[Arc09] Arcticus Systems homepage, 2009. `www.arcticus-systems.com`.

[BBK09] Mark Bartlett, Iain Bate, and Dimitar Kazakov. Guaranteed loop bound identification from program traces for wcet. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:287–294, 2009.

[BBMP00] I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-level analysis of a portable java byte code WCET analysis framework. *Real-Time Computing Systems and Applications, International Workshop on*, 0:39, 2000.

[BBP99] Alexander Barvinok, Er Barvinok, and James E. Pommersheim. An algorithmic theory of lattice points in polyhedra. In *New Perspectives in Algebraic Combinatorics*, pages 91–147. MSRI Publications, 1999.

[BEGL05] Susanna Byhlin, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Applying static WCET analysis to automotive communication software. In *Proc. 17th Euromicro Conference of Real-Time Systems, (ECRTS'05)*, July 2005.

[BEL09] Stefan Bygde, Andreas Ermedahl, and Björn Lisper. An efficient algorithm for parametric WCET calculation. In *The 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2009*, pages 13–21. IEEE Computer Society, August 2009.

[Ben02] Patricia Mary Benoy. *Polyhedral Domains for Abstract Interpretation in Logic Programming*. PhD thesis, Computing Laboratory, University of Kent, Canterbury, Kent, UK, January 2002.

[BJT99] Frédéric Besson, Thomas P. Jensen, and Jean-Pierre Talpin. Polyhedral analysis for synchronous languages. In *SAS '99: Proceedings of the 6th International Symposium on Static Analysis*, pages 51–68, London, UK, 1999. Springer-Verlag.

[BL08] Stefan Bygde and Björn Lisper. Towards an automatic parametric WCET analysis. In *Worst-Case Execution Time Analysis Workshop*, pages 9–17. Austrian Computer Society, July 2008.

[BR04a] Gogul Balakrishnan and Thomas W. Reps. Analyzing memory accesses in x86 executables. In *CC*, volume 2985 of *Lecture Notes in Computer Science*, pages 5–23. Springer, 2004.

[BR04b] Iain Bate and Ralf Reutemann. Worst-case execution time analysis for dynamic branch predictors. In *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 215–222, Washington, DC, USA, 2004. IEEE Computer Society.

[BR05] Claire Burguiere and Christine Rochange. A contribution to branch prediction modeling in WCET analysis. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 612–617, Washington, DC, USA, 2005. IEEE Computer Society.

[Byg07] Stefan Bygde. Analysis of arithmetical congruences on low-level code (extended abstract). In Olaf Owe and Gerardo Schneider, editors, *Nordic Workshop on Programming Theory*. Oslo University, October 2007.

[CB02] Antoine Colin and Guillem Bernat. Scope-tree: A program representation for symbolic worst-case execution time analysis. *Euromicro Conference onReal-Time Systems*, 0:50, 2002.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4^{th} ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[CEE^{+}02] Martin Carlsson, Jakob Engblom, Andreas Ermedahl, Jan Lindblad, and Björn Lisper. Worst-case execution time analysis of disable interrupt regions in a commercial real-time operating system. In *Proc. 2^{nd} International Workshop on Real-Time Tools*, 2002.

[CH78] Patrick Cousot and Nicholas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. 5th ACM Symposium on Principles of Programming Languages*, pages 84–97, 1978.

[CHMW07] Joel Coffman, Christopher Healy, Frank Mueller, and David Whalley. Generalizing parametric timing analysis. In *LCTES '07: Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools*, pages 152–154, New York, NY, USA, 2007. ACM.

[Cla96] Philippe Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: applications to analyze and transform scientific programs. In *ICS '96: Proceedings of the 10th international conference on Supercomputing*, pages 278–285, New York, NY, USA, 1996. ACM.

[CM07] Christoph Cullmann and Florian Martin. Data-flow based detection of loop bounds. In Christine Rochange, editor, *WCET*, volume 07002 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[CMRS05] Ting Chen, Tulika Mitra, Abhik Roychoudhury, and Vivy Suhendra. Exploiting branch constraints without exhaustive path enumeration. In Reinhard Wilhelm, editor, *Proc. 5th International Workshop on Worst-Case Execution Time Analysis (WCET'2005)*, pages 40–43, Palma de Mallorca, July 2005.

[Cou01] Patrick Cousot. Abstract interpretation based formal methods and future challenges. In Reinhard Wilhelm, editor, *Informatics*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer, 2001.

[CP00] Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2/3):249–274, 2000.

[Crn05] Ivica Crnkovic. Component-based software engineering for embedded systems. In *International Conference on Software engineering, ICSE'05*. ACM, 5 2005.

[EG97] A. Ermedahl and J. Gustafsson. Deriving Annotations for Tight Calculation of Execution Time. In *Proc. Euro-Par'97 Parallel Processing, LNCS 1300*, pages 1298–1307. Springer Verlag, Aug 1997.

[Eng02] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Uppsala, Sweden, April 2002. ISBN 91-554-5228-0.

[Erm08] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. VDM Verlag, 2008.

[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 Christine Rochange, editor, *Proc. 7th International Workshop on Worst-Case Execution Time Analysis (WCET'2007)*, Pisa, Italy, July 2007.

[Fea88] P. Feautrier. Parametric integer programming. *Operationnelle/Operations Research*, 22(3):243–268, 1988. citeseer.ist.psu.edu/feautrier88parametric.html.

[FW99] Christian Ferdinand and Reinhard Wilhelm. Fast and efficient cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2/3):131–181, 1999.

[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*, August 2007.

[GMC09] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 127–139, New York, NY, USA, 2009. ACM.

[Gra89] Philippe Granger. Static Analysis of Arithmetical Congruences. *International Journal of Computer Mathematics*, pages 165–199, 1989.

[Gra91] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT '91: Proceedings of the international joint conference on theory and practice of software development on Colloquium on trees in algebra and programming (CAAP '91): vol 1*, pages 169–192, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[Gus00] Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Dept. of Information Technology, Uppsala University, Sweden, May 2000. Available as report DoCS 00/115.

[HÅCT04] Hans Hansson, Mikael Åkerholm, Ivica Crnkovic, and Martin Törngren. SaveCCM - a component model for safety-critical real-time systems. In *EUROMICRO '04: Proceedings of the 30th EUROMICRO Conference*, pages 627–635, Washington, DC, USA, 2004. IEEE Computer Society.

[HAM$^+$99] Christopher A. Healy, Robert D. Arnold, Frank Mueller, Marion G. Harmon, and David B. Walley. Bounding pipeline and instruction cache performance. *IEEE Trans. Comput.*, 48(1):53–70, 1999.

[HC01] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Professional, June 2001.

[HPR94] Nicolas Halbwachs, Yann-Eric Proy, and Pascal Raymond. Verification of linear hybrid systems by means of convex approximations. In *SAS*, pages 223–237, 1994.

[HSRW98] C. Healy, Mikael Sjödin, V. Rustagi, and David Whalley. Bounding Loop Iterations for Timing Analysis. In *Proc. 4th IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, June 1998.

[HW99] Christopher Healy and David Whalley. Tighter timing predictions by automatic detection and exploitation of value-dependent constraints. In *RTAS '99: Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium*, page 79, Washington, DC, USA, 1999. IEEE Computer Society.

[Kle52] Stephen Cole Kleene. *Introduction to metamathematics*. Bibl. Matematica. North-Holland, Amsterdam, 1952.

[LBJ$^+$95] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong-Sang Kim. An accurate worst case timing analysis for risc processors. *IEEE Trans. Software Eng.*, 21(7):593–604, 1995.

[LCFM09] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 136–146, Washington, DC, USA, 2009. IEEE Computer Society.

[Lis03a] Björn Lisper. Fully automatic, parametric worst-case execution time analysis. In Jan Gustafsson, editor, *Proc. Third International Workshop on Worst-Case Execution Time WCET Analysis*, pages 77–80, July 2003.

[Lis03b] Björn Lisper. Fully automatic, parametric worst-case execution time analysis. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-97/2003-1-SE, Mälardalen University, April 2003.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'95)*, La Jolla, CA, June 1995.

[LM97] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 16(12):1477–1487, December 1997.

[LMW99] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems.*, 4(3):257–279, 1999.

[Lun02] Thomas Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, June 2002.

[MBCS08] Marianne De Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static loop bound analysis of c programs based on flow analysis and abstract interpretation. In *The 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2008*, pages 161–166. IEEE Computer Society, 2008.

[MDH09] The mälardalen benchmark suite, 2009. `www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

[Min01] A. Miné. The octagon abstract domain. In *Proc. of the Workshop on Analysis, Slicing, and Transformation (AST'01)*, IEEE, pages 310–319, Stuttgart, Germany, October 2001. IEEE CS Press. www.di.ens.fr/ mine/publi/article-mine-ast01.pdf.

[Min06] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, pages 54–63, Ottawa, Ontario, Canada, June 2006. ACM Press. www.di.ens.fr/ mine/publi/article-mine-lctes06.pdf.

[MOS05] Markus Müller-Olm and Helmut Seidl. Analysis of modular arithmetic. In Shmuel Sagiv, editor, *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2005.

[MOS07] Markus Müller-Olm and Helmut Seidl. Analysis of modular arithmetic. *ACM Trans. Program. Lang. Syst.*, 29(5):29, 2007.

[Muc97] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997. ISBN: 1-55860-320-4.

[New09] New polka webpage, 2009. `pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/newpolka/index.html`.

[NNH05] Flemming Nielson, Hanne Ries Nielson, and Chris Hankin. *Principles of Program Analysis, 2nd edition*. Springer, 2005. ISBN 3-540-65410-0.

[Oct09] Octagon library website, 2009. `www.di.ens.fr/˜mine/oct/`.

[Par09] Parma polyhedra library website, 2009. `www.cs.unipr.it/ppl/`.

[Pip09] The parametric integer programming's home, 2009. `www.piplib.org/`.

[Pol09] Polylib website, 2009. `icps.u-strasbg.fr/polylib/`.

[PPVZ92] Gustav Pospischil, Peter Puschner, Alexander Vrchoticky, and Ralph Zainlinger. Developing real-time tasks with predictable timing. *IEEE Softw.*, 9(5):35–44, 1992.

[PS91] Chang Yun Park and Alan C. Shaw. Experiments with a program timing tool based on source-level timing schema. *Computer*, 24(5):48–57, 1991.

[Pug94] William Pugh. Counting solutions to presburger formulas: How and why. In *PLDI*, pages 121–134, 1994.

[RBL06] Thomas Reps, Gogul Balakrishnan, and Junghee Lim. Intermediate-representation recovery from low-level code. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 100–111, New York, NY, USA, 2006. ACM.

[Rei08] Jan Reineke. *Caches in WCET Analysis*. PhD thesis, Universität des Saarlandes, Saarbrücken, November 2008.

[SA00] Friedhelm Stappert and Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *J. Syst. Archit.*, 46(4):339–355, 2000.

[SEGL04] D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper. Static timing analysis of real-time operating system code. In *Proc. 1st International Symposium on Leveraging Applications of Formal Methods (ISOLA'04)*, October 2004.

[SEGL06] Christer Sandberg, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Faster WCET flow analysis by program slicing. In *ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES2006)*. ACM, June 2006.

[Tar55] Alfred Tarski. A lattice theoretical fixpoint theorem and its applications. In *Pacific Journal of Math*, pages 285–309, 1955.

[The04] S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[VCKL05] Michael Venable, Mohamed R. Chouchane, Md. Enamul Karim, and Arun Lakhotia. Analyzing memory accesses in obfuscated x86 executables. In Klaus Julisch and Christopher Krügel, editors, *DIMVA*, volume 3548 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2005.

[VHMW01] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric timing analysis. In Jay Fenwick and Cindy Norris, editors, *LCTES'01*, pages 88–93, Snowbird, Utah, June 2001.

[VSB$^+$07] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, 2007.

[WCE09] The WCET analysis project, 2009. `www.mrtc.mdh.se/projects/wcet/`.

[WEE$^+$08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

[Wei81] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[Wei84] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[Wil05] Stephan Wilhelm. Efficient analysis of pipeline models for WCET computation. In *Proceedings of the 5th Intl. Workshop on Worst-Case Execution Time Analysis*, 2005.

[ZY09] Wei Zhang and Jun Yan. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In Patrick Kellenberger, editor, *The 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2009*, pages 455–463. IEEE Computer Society, August 2009.