FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Symbolic Methods for the Timing Analysis of Programs

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht von

Jakob Zwirchmayr
Matrikelnummer 0155901

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Laura Kovács

Diese Dissertation haben begutachtet:

(Laura Kovács)	(Jens Knoop)	(Christine Rochange)

Wien, September 1, 2013

(Jakob Zwirchmayr)

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Symbolic Methods for the Timing Analysis of Programs

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Jakob Zwirchmayr
Registration Number 0155901

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Laura Kovács

The dissertation has been reviewed by:

(Laura Kovács)	(Jens Knoop)	(Christine Rochange)

Wien, September 1, 2013

(Jakob Zwirchmayr)

Erklärung zur Verfassung der Arbeit

Jakob Zwirchmayr
Alliiertenstrasse 16/23, 1020 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Abstract

Today, embedded software systems (ESS) are “ubiquitous” technical devices used on a daily basis by a vast number of people. We rely on these systems to work, that is to be *functionally correct*, and thus to operate as expected. Therefore, time and effort is invested to provide the expected quality for the ESS. For *real-time* ESS *timing-properties* are of special importance: such systems need not only be functionally correct, but also deliver results in time. In some applications, such as a video stream, timing violations result in an acceptable, but probably bad quality of the service. For a *hard* real-time system, results are useless once the expected timing behaviour is violated. A *safety-critical* real-time system, such as air and car control devices, however, need not only be functionally correct, but also has to ensure a correct timing behaviour. Missing a deadline is not an option for safety-critical real-time ESS, since an error can have disastrous economic and social consequences. Identifying and correcting such errors is therefore a challenging research topic, both in academia and industry. *In this thesis we address this challenge and describe rigorous methods for the timing analysis of programs.*

One of the most important ESS timing properties is the *Worst-Case Execution Time* (WCET) of the system, that is the maximal running time of the program on the specified hardware. WCET analysis needs to provide formal guarantees that the system under analysis exhibits a proper timing behaviour. This requires computing safe and tight bounds for the execution times of programs. State-of-the-art WCET analysis tools usually apply data-flow analysis, abstract interpretation and/or model-checking techniques on the program in order to infer relevant WCET information. Though very sophisticated and powerful, their results are sometimes not precise enough to conclude a safe and tight timing-behaviour for the system under analysis. By design, none of the existing tools is able to guarantee that a bound that corresponds to the execution time of a concrete system execution is found: often, due to the employed abstractions, the WCET is actually computed for a spurious trace in the program, giving no hint whether the bound is precise (i.e. a bound for an actual execution) or it is an over-estimation of a spurious system execution.

In contrast to these state-of-the-art techniques, in this thesis we present a method for *proving WCET bounds precise.* Our approach guarantees that the WCET bound is actually computed for a feasible trace in the system. Moreover, our method is able to improve the WCET analysis quality by reducing the required manual annotations and the imprecision of WCET estimates, automatically tightening the WCET bound until precise when necessary. For doing so, we apply and combine symbolic techniques

from algorithmic combinatorics, computer algebra, automated theorem proving and formal methods. Such symbolic techniques usually yield good analysis information but are computationally expensive: symbolic execution, for example, requires path-wise execution of the whole program. In order to overcome the computationally expensive costs of symbolic methods, our challenge is to identify and apply symbolic methods only on relevant program parts. This way, the precision of symbolic techniques is fully exploited while avoiding the computational costs of the underlying approaches.

The algorithm we propose to prove WCET bounds precise is an *anytime algorithm*, that is, it can be stopped at any time without violating the soundness of its results. Ultimately, if run until termination, WCET bounds can be proved precise as the algorithm iteratively refines the abstraction, resulting in tighter WCET bounds, until termination of the approach. Then, any WCET over-estimation is due to the (imprecise) hardware model used in the WCET computation. Our approach relies on an initial WCET analysis and terminates fast if the supplied results of the initial WCET analysis are tight.

In order to *improve performance of our approach*, we combine our method with an automatic technique for computing tight upper bounds on the iteration number of special classes of program loops. These upper bounds are further used in the initial WCET analysis of the programs. Our automatically inferred loop bounds reduce the effort of manual annotations and increase the safety of the timing analysis of programs. Even with exact loop bounds, the WCET computed by an analyzer is usually not tight, leaving a gap between the actual and the computed WCET of a program. This gap (over-estimation) is due to the abstraction that is used for the WCET computation as well: it usually encodes numerous spurious paths due to infeasible branching decisions for conditionals. A benefit of our approach is that we automatically infer additional constraints from spurious traces that allow to further tighten the WCET bound. The improved WCET derived by our work therefore significantly improves the precision of the timing analysis.

To evaluate our symbolic techniques *we implemented our approach in the r-TuBound WCET toolchain.* We tested and evaluated r-TuBound on a large number of benchmarks from the WCET community. To make our algorithm available to other WCET analyzers, we addressed the *portability of our method* and extended r-TuBound to support the XML based intermediate flow fact format FFX. This way, analyzers supporting FFX can make use of the underlying infrastructure of our work. To this end, we compared r-TuBound with state-of-the-art WCET tools, including the TuBound and the OTAWA/oRange toolchains. Our experimental results underline the advantage of using symbolic methods for proving WCET bounds precise, at a moderate cost.

Contents

CHAPTER 1

Introduction

Today, embedded software systems (ESS) are “ubiquitous” technical devices used on a daily basis by a vast number of people. We rely on these systems to work, and thus to operate as expected. Embedded software systems are usually composed from some application-specific hardware device (i.e. a microprocessor) that has access to the world via sensor data or some input system, and processes the input data by running software tasks on the system. There are certain expectations that one can impose on such systems: most importantly they should be *functionally correct*. Functional correctness means, on the one hand, that the device does not crash, that is the underlying hardware is not faulty and the software does not contain defects. On the other hand, it also means that the device operates and functions as expected. For example, a GPS device in a car is expected not only not to crash when turned on, but also to describe an acceptable travel route for a supplied start- and endpoint, at least when the underlying hardware is not faulty. Ensuring functional correctness and thus providing the expected quality of an embedded software system is a challenging research topic, both in academia and industry. Rigorous approaches based on formal methods have been developed to address and solve this challenge, including for example extensive testing of systems until one is sufficiently confident that no errors exist or formally verifying the underlying hardware and software system for proving the absence of errors.

When it comes to *real-time* embedded software systems, additionally to functional correctness, ensuring *timing-properties* of such systems are of utmost importance. That is, such systems need not only be functionally correct, but also need to adhere to time restrictions. Timing-properties are properties of the system that are not related to the correctness or proper functioning of the system, but are additional constraints on the time within which results of the system are expected. As a consequence, timing-properties are usually dependent on the underlying hardware model. For example, the timing behaviour of real-time embedded systems is influenced by the operating frequency that the underlying processor is run at, the caches used in the hardware system and the scheduling decisions made on the level of the real-time operating system.

Depending on the application context, the timing behaviour of real-time embedded systems can be critical or not. An example of an uncritical real-time application is a Voice-Over-IP application, i.e. an audio or video stream, where the order in which video or audio packets arrive is important. That is, packets that were sent from the source in some order are expected to arrive at the destination in a similar order. As different packets might travel different routes through the system network, it might happen that an earlier sent packet travels a route that makes it arrive later than a later sent packet. This way, the packets arriving at the source arrive out of order, yielding a timing violation enforcing the video or audio stream to suffer from artifacts, hang, lag or stutter. Such a timing violation results in a bad quality of the service, but it can still be acceptable if the timing violations are within certain limits (i.e. one is still able to see and understand the person on the other end of the audio/video stream). However, such a timing behaviour is absolutely unacceptable in the case of *hard real-time systems*. For a hard real-time system, results are useless once the expected timing behaviour is violated. For example, if too many packets arrive out-of-order in a video stream, at some point the video stream will be unusable since very few required frames arrive in order. Nevertheless, a failing video stream is usually not considered to be safety-critical, as most probably a failing video call will not bring a disastrous social consequence. The situation is, however, very different for *safety-critical hard real-time systems* upon which social safety, if not even human lives, depend on. Examples of such ESS usually come from the avionics and automotive industry, for example in fly-by-wire and drive-by-wire technologies where there are no mechanical links between the control column and the steering gear of an aircraft or car [45].

Safety-critical hard real-time systems need thus to be both functionally correct and have a correct timing behaviour. Missing a deadline is not an option for safety-critical real-time ESS, as an error can have tragical economic and social consequences. For example, the system that is responsible for opening the airbags in a car in the case of an accident needs to react within a time limit of around 40 ms. That is, when the sensor data (e.g. from the accelerometer and pressure sensors) registers an accident condition, the system must compute its required actions and supply the appropriate commands to the respective actuators to open the airbag within the 40 ms time limit.

If under some circumstances the system requires longer to open the airbag, the system can be considered erroneous (and therefore, dangerous): a user (car driver) could already have been hurt severely, while the airbag is still inflating after more than 40 ms. Therefore, for such systems, ensuring functional correctness as well verifying the timing behaviour is equally important. One of the most crucial research tasks in the timing analysis of ESS is defined as the *Worst-case Execution Time* (WCET) problem: the WCET problem is to find an upper bound on the maximal execution time of the system under analysis. With a safe WCET analysis method yielding a WCET bound, any execution of the system is guaranteed to be below the inferred WCET bound. In the airbag example considered above, if a WCET analyzer derives a WCET bound of 30 ms, a safe timing behaviour of the system is ensured.

WCET analysis is typically performed using either static or dynamic approaches.

Static WCET analyzers inspect the program without running it, usually using abstractions of the program to ease analysis effort, often resulting in a WCET over-estimation. Dynamic WCET analyzers on the other hand use concrete input data to run and measure systems executions. Finding a safe WCET bound using dynamic analysis tools therefore requires to measure all program executions, which is an infeasible task in practice. Therefore, dynamic WCET analyzers only measure the execution time of some subset of actual program runs, and usually under-estimate the WCET of the program. By design, neither any of the dynamic approaches for WCET analysis nor any of the static approaches is able to conclude a precise timing behaviour of the system, or even more importantly, to prove or disprove that its computed WCET bound is precise. For example, whenever an unsafe timing behaviour is reported, it is unclear whether this behaviour results from a concrete but too high WCET, or whether the system is actually safe but the reported WCET bound is over-estimated. In such situations, *a proof of WCET precision would be especially useful.* With such a proof at hand, we could either (i) establish the precision of an overly high WCET bound, and hence conclude that a required smaller WCET bound cannot be reached; or (ii) prove that the current WCET bound is not precise, and hence might conclude system safety when an improved and precise bound is finally derived.

In this thesis we address the WCET problem by making use of the advantages of both static and dynamic WCET analysis. We describe an ***automated framework for proving WCET bounds precise****, by combining symbolic techniques from algorithmic combinatorics, computer algebra, automated theorem proving and formal methods.* The results of our method are, by design, out-of-scope of traditional static WCET analyzers, as static WCET methods cannot infer whether a computed WCET bound is an over-estimation or a precise WCET of the program on a particular hardware model. Our method proves the precision of a computed WCET bound, extending thus the state-of-the-art in static WCET analysis.

Given an initial WCET bound computed by an arbitrary WCET bound approach, our approach iteratively refines this bound until it proves it precise, as follows. As the (initial) WCET bound is computed on an abstraction of the concrete program, program paths exhibiting the WCET bound of the abstract program might become infeasible once the concrete behaviour of the program is considered, yielding thus spurious WCET program traces. Our method therefore maps the (initial) WCET bound to one or more concrete program traces exhibiting this bound. Mapping the current WCET bound back to a trace in the program allows us to use a decision procedure to check the feasibility of this trace. If the trace inferred from the program abstraction is also feasible in the concrete program, the computed WCET bound of the trace is precise and the timing behaviour is indeed exhibited by the program. The proof of precision is given by a satisfiable path feasibility check of the decision procedure, formulated as a satisfiability modulo theory (SMT) problem in the theory of linear integer arithmetic, bit-vectors and arrays. On the other hand, if the decision procedure reports the infeasibility of the trace, we conclude that the current WCET bound describes a spurious program behaviour which is exhibited only in the abstract program model but not in the concrete one as well. In this situation, the WCET bound is thus an imprecise WCET over-estimation, and our proof of this imprecision is given by the path infeasibility result of the decision procedure.

Our approach allows to iteratively refine the WCET bound while searching for a proof of precision. This means, if the WCET bound reported by a WCET analyzer is over-estimated, our method will refine the WCET bound until it is proven precise. If the initial WCET bound was highly over-estimated, then using our algorithm to achieve WCET precision comes, in general, at a high cost as well. However, due to the maturity of state-of-the-art WCET analyzers, such a scenario is quite unlikely to happen. State-of-the-art WCET analyzers keep the over-estimation of WCET bounds low (around 25% higher than measured execution times), even for complex architectures [64]. Hence, it is safe to assume that the over-estimation of the initial WCET bound is low or almost none, and therefore using our method to achieve proven precision of WCET involves a moderate cost as well, as demonstrated by our experimental results.

1.1 Problem Statement

In this thesis we study the WCET problem from the timing analysis of embedded software systems. Unlike other exsiting approaches, our work introduces *a novel and automated framework for proving precise WCET bounds.* For doing so, the thesis addresses and solves the following three research problems.

1. Proof – Precision of WCET bounds. We present an algorithm that allows to *decide whether a WCET bound reported by a WCET analyzer is precise.* Our algorithm can be used in conjunction with any static WCET analyzer implementing the implicit path enumeration (IPET) techniques. The result of our method is either a proof of precision or a counter-example, that is a proof of the infeasibility of the program path exhibiting the current WCET bound. Our method, by design, is out-of-scope of state-of-the-art WCET analysis tools, as current static and dynamic WCET analyzers are not able to infer whether a computed WCET bound is precise or over-estimated.

WCET Squeezing as a Proof Procedure. Our method for proving precise WCET bounds is called *WCET Squeezing.* WCET squeezing automatically tighten WCET results by inferring additional constraints from infeasibility proofs of program paths. The constraints derived by WCET Squeezing can be used in further WCET computations, and thus proving precise WCET bounds.

Based on the WCET Squeezing algorithm, we define the *pragmatic WCET problem* to be the following decision problem. It is the problem to decide whether an insufficient WCET bound reported by a static analyzer can be refined and proved to be below some required threshold. Solving the pragmatic WCET problem requires to decide whether a WCET bound is precise: if the bound is precise and above the threshold, the answer to the pragmatic WCET problem is “no”. If the bound can be tightened and hence proved to be below the threshold, the answer to the pragmatic WCET problem is “yes”.

We propose WCET Squeezing as a *decision procedure solving the pragmatic WCET problem.* WCET Squeezing can be run as an anytime algorithm, until the WCET bound is below the required threshold, and thus allows to decide whether an insufficient WCET bound reported by a static analyzer can be refined and proved below some required threshold. WCET Squeezing iteratively refines WCET bounds and even a single iteration of WCET Squeezing might allow to verify (user-supplied) WCET bounds, and terminating WCET Squeezing at an arbitrary iteration still guarantees an improved WCET bound. As tight initial WCET analysis results reduce the effort to prove precision of a WCET bound, in our quest to *efficiently decide* the pragmatic WCET problem, we investigated the following problem in traditional WCET analysis.

2. Performance — Symbolic Methods in Proving Precise WCET bounds. In our approach of proving precise WCET bounds, we rely on sophisticated techniques from symbolic computation, automated theorem proving and formal methods. These methods are used to derive better (initial) WCET bounds and hence to accelerate WCET Squeezing for proving precise WCET bounds. In particular, we make use of recurrence solving approaches from algorithmic combinatorics and combine these techniques with symbolic execution and SMT procedures for analysing program loops and proving feasibility of program paths in the theory of linear integer arithmetic, bit-vectors and arrays. While the deployed symbolic techniques, especially those from symbolic computation and symbolic execution, have in general a very high computational cost. We show that *careful application of these techniques can improve the quality of WCET analyzers at a moderate cost.* For example, instead of exploring all program paths, in WCET Squeezing we apply symbolic execution only on those program traces that could exhibit the reported WCET bound. On the other hand, instead of applying general recurrence solving algorithms, we propose a computationally cheap pattern-based recurrence solving approach to improve the performance of computing WCET bounds, as detailed below.

Loop bound computation for accelerating WCET Squeezing. As proving precise WCET bounds, and in particular WCET Squeezing, crucially depends on the quality of the initial WCET bound, computing nice WCET bounds is a challenging task especially for programs with loops and/or recursion. We have addressed the performance of our method of proving precise WCET bounds, and proposed an *automated approach to derive precise bounds on the number of loop iterations, called loop bounds, for certain, yet general enough classes of program loops.* For doing so, we model program loops by algebraic relations expressing recurrence relations over program variables and loop counters, and combine pattern-based recurrence solving with SMT-based reasoning over linear integer arithmetic to derive loop bounds that cannot be obtained by other loop bound analysis techniques used in WCET analysis. The class of programs our method is able to handle implements non-trivial arithmetic and complicated control flow, for example conditional updates to loop counters within a loop body. For such programs, our method derives tight loop bounds which can be further uses as trusted annotations for ensuring precision of WCET analysis.

3. Portability – Distributing WCET Squeezing by Enabling Interoperability of WCET Analyzers. In this thesis we present *an approach to make our proving precise WCET approach portable for immediate usage in arbitrary WCET tools.* We achieve portability by making tools of different WCET analyzers interoperable by means of introducing a flexible and extensible intermediate language providing a common interface for exchanging analysis information.

For doing so, we adapted an intermediate flow fact format employed in another WCET analysis toolchain, FFX. Interoperability not only makes WCET Squeezing available to other WCET tools, adapting this common interface helps in our attempt to reduce the overhead for finding a proof of precision for WCET bounds: we try to leverage advantages of available WCET tools, for example, tighter loop bounds for program loops for which our loop bound computation method fails.

As different WCET analyzers perform better on different problems, our approach of proving precise WCET bounds might benefit from using and exchanging results between existing WCET tools. Moreover, a major problem in WCET analysis is the lack of comparability and interoperability of WCET tools. Usually, WCET analyzers all "live on their own islands", with their own internal and intermediate formats. Additionally, focus on certain types of program analysis often diverges the results of WCET analyzers for different types of systems. WCET analysis, and hence our methods, would clearly benefit from combining the advantages from different tools.

With the FFX extension at hand, we made the first attempt towards *comparability and interoperability between WCET analysis toolchains.* Additionally, the common annotation language format of FFX can be used to improve WCET Squeezing, by supplying it with a more precise initial WCET bound derived by some existing WCET analyzer, thus decreasing the effort to find a proof of precision.

4. Practice – Implementation and Experiments. The overall approach of proving precise WCET bounds is implemented in the r-TuBound tool. Our implementation offers an automated and uniform framework to refine WCET bounds by WCET Squeezing, compute loop bounds by symbolic methods, and exchange timing analysis information with other WCET toolchains using the FFX format. We have evaluated r-TuBound on a large number of WCET benchmarks, and compared our results against the TuBound and the oRange/Ottawa WCET analyzers. Our results show that exchanging information between WCET tool chains gives much better results when applying WCET Squeezing, and WCET Squeezing in conjunction with the automated computation of loop bounds significantly improves the precision of WCET bounds.

1.2 Contributions

Summarizing, the present thesis brings the following contributions in the area of timing analysis.

1. Proof – WCET Squeezing as Proof Procedure. We present an effective procedure that is able to prove or disprove if the WCET bound computed by a state-of-the-art WCET analyzer is precise, and, if not, to tighten the time bound it delivers until eventually the tightened bound is proven precise. This procedure is called WCET Squeezing and iteratively refines the WCET by excluding infeasible program paths until the precise WCET is obtained or until an additional treshold is reached.

2. **Performance – Accelerating WCET Squeezing.** To make WCET Squeezing highly performant, it is essential to empower the used WCET analyzer. In this thesis we focus on the use of symbolic methods for accelerating WCET Squeezing, and describe an automated approach for static loop analyses, in particular for computing tight bounds on loop iterations. To this end, we combine techniques from symbolic computation, automated theorem proving and formal methods in a restricted, yet efficient and powerful way.

3. **Portability – Distributing WCET Squeezing among WCET Analyzers.** We describe a flexible and extensible approach to make our proof procedure portable for immediate usage in arbitrary WCET analyzer toolchains. This is achieved by making tools of different WCET analyzers interoperable by introducing and extending a common annotation language, called FFX.

4. **Practice – Implementation and Experiments.** Our work has been implemented in the r-TuBound WCET toolchain, and evaluated and compared against the TuBound and oRange/OttawaWCET WCET analyzers.

CHAPTER 2

Preliminaries

This section contains a brief overview of the programming model, WCET analysis, symbolic execution and algebraic techniques, relevant for the later chapters of the thesis. We first present our programming model and fix some notation. We then overview static analysis, IPET based WCET analysis and symbolic execution; these techniques are the basis of our approach to prove WCET bounds precise. In order to improve the performance of the approach, we introduce a pattern-based recurrence solving method to compute loop bounds. To make our approach portable and available for other high-level analyzers, we adapt a common intermediate flow fact format.

2.1 Programming Model

Throughout this thesis, $\mathbb{N}$ and $\mathbb{R}$ denote the set of natural and real numbers, respectively.

We rely on the work of [51] and represent programs as control flow graphs (CFG), $CFG := ((V, E), S, X)$, where V is the set of nodes, E is the set of directed edges representing program blocks, $S \in V$ is the start-node, and $X \in V$ is the end-node. For each edge $e \in E$ an edge weight $w(e)$ is assigned denoting the execution time of e, and we have $w : E \to \mathbb{N}$. To ease readability, we will omit edge weights wherever possible. Every node n, different than S and X, has incoming $inc(n)$ and outgoing edges $out(n)$; the node S has only outgoing edges $out(S)$ and no incoming ones; whereas the node X has only incoming edges $inc(X)$ but no outgoing ones. Conditional nodes C split the flow depending on the runtime evaluation of a boolean condition $c(C)$, where we refer to $c(C)$ as the path-condition. For simplicity, we sometimes write C instead of $c(C)$. Execution times for $c(C)$ are assumed to be added to the successor edge weights. Edges taken when the condition C evaluates to *true* are called *true-edges* (*true-blocks*) and are denoted by t. Similarly, edges taken when the condition evaluates to *false* are called *false-edges* (*false-blocks*) and are denoted by f. To make explicit that t and f result from the evaluation of C, we write t_C and f_C to mean that these are the true-, respectively `false`-edges of C. Further, t_C and f_C are called the *conditional-edges* of C.

A path in the CFG is a sequence of nodes and edges and a program execution trace is a path from S to X. The evaluation of all path conditions in an execution trace defines the *branching behavior* of the trace, i.e. the evaluation of all conditions along the trace. It is thus a sequence of branching decisions that can be encoded as a sequence of bits, where each bit b_i represents the result of evaluating the i^{th} branch condition in the trace: b_1 is 1 if the condition holds when executing the `true`-edge, and it is 0 if the `false`-edge is executed. A program loop in the CFG is modeled by a loop header *lh*, loop body *lb* and loop exit *le* node, with an edge from *lb* to *lh*. Each loop is annotated in the CFG with a loop bound ℓ. A valid path including a loop thus contains *lb* at most $lh * \ell$ times, each time *lh* is contained. The number of times an edge e is taken in a path is given by its frequency $freq(e)$, where $freq : E \to \mathbb{N}$ and $freq(e)$ gives the sum of executions of the edge e in an execution trace.

2.2 Static WCET Analysis

Efficient and precise WCET analysis relies on program analysis and optimization techniques. In this section we overview only those static WCET and program analysis techniques on which our framework crucially depends on. A detailed survey about WCET and program analysis techniques can be found in [28, 54].

By a WCET analysis toolchain of a WCET analyzer we mean the full collection of tools used to analyze a program. Usually, a WCET analysis toolchain consists of an architecture-independent high-level analyzers inferring information (i.e. flow facts) about the program flow. This part of a WCET analysis tool chain is called as the WCET front-end. As an example of WCET front-ends we only name here the oRange [5] and the r-TuBound [44] tools. The parts of a WCET analysis toolchain that operate on the architecture-dependent lower level are called WCET back-ends, for example the Otawa [5] and the CalcWCET167 [40] back-ends.

Essential flow facts include loop bounds and execution frequencies (i.e. worst-case execution counts) of conditional edges in the program. Both of these flow facts are required to calculate the WCET of programs with loops. However, flow facts about programs might not always be precise. For example, loop bounds are usually not tight but are over-approximated, since they depend on the results of a so-called interval analysis that is itself imprecise. Therefore, the abstraction used to compute a WCET bound usually encodes numerous spurious program paths, some of these spurious program paths exhibiting high execution frequencies. As a consequence, the WCET bound computed using these imprecise flow facts is only an over-estimation of the actual WCET. The precision of static WCET analysis therefore crucially depends on the quality of the used flow facts.

When computing WCET estimates, the underlying hardware architecture needs to be analyzed for inferring execution times of program blocks. Additional hardware features, such as cache-configuration and pipeline layout, also need to be taken into account. In this thesis we are concerned with high-level WCET analysis only. In the sequel, we overview the most important high-level analysis steps use in our method.

(1) Interval and points-to analysis implements a forward directed data flow interval analysis, yielding variable values and aliasing information for all program locations. Additionally, a (unification based) flow-insensitive points-to analysis helps dealing with aliased pointer and array data.
(2) Counter-based loop analysis derives bounds for simple loops with incremented/decremented updates, by constructing symbolic equations that are solved using pattern matching and term rewriting. It is necessary to acquire concrete values for variables involved in the loop initialization, condition and increment expression. Generally, if no values are available for variables in the loop initialization, condition, and increment expression, then the analysis fails in deriving loop bounds. Note that in certain cases variables with unknown values can be discarded, and thus loop bounds can be derived even though interval analysis failed to produce concrete variable values.
(3) Constraint-based loop analysis models and enumerates the iteration space of nested loops. To this end, a system of constraints is constructed that reflects dependencies between iteration variables and thus yields better results than just multiplying the bounds of nested loops. The solution to the constraint system is computed by a constraint logic programming solver over finite domains.

It is not hard to argue that the techniques of (2) and (3) can fail on a large class of programs, either because of unhandled loop updates, complex nesting of loops or the arithmetic and data structures used in conditional expressions. One contribution of this thesis addresses this problem and introduces an automated approach overcoming some limitations of the methods of (2) and (3); based on this framework, we were even able to automatically derive loop bounds for some complex loops that stem from search- and sorting-algorithms.
(4) Implicit path enumeration technique (IPET) first maps the program flow to a set of graph flow constraints describing relationships between execution frequencies of program blocks. Execution times for expressions are evaluated by a low-level WCET-analysis. The execution times are then summed up to get execution times for program blocks. Finally, the longest execution path (exposing the WCET) of the program is found by maximizing the solution of the constraint system. In other words, the path exhibiting the WCET is derived by maximizing the executions times over program paths. This maximization problem can naturally be formulated as an integer linear program, shortly referred to as an ILP problem. We illustrate the IPET in Example 2.1 below.

Example 2.1 Consider the program given in Figure 2.1. Its program flow is given in Figure 2.2. The dependency relations between execution frequencies of program blocks are extracted from Figure 2.2 and are listed in Figure 2.3. The execution times of simple program expressions are either inferred by a low-level timing analysis or are manually supplied. For simplicity, we assume that all basic expressions take one time unit to execute. Therefore, execution of one loop iteration takes at most 4 time units: 1 unit to check each boolean condition `b` and `c`, and 1 unit to execute each statement `d` and `e`. The longest path through the loop is the node sequence $b \to c \to d \to e$. We consider[1]

[1] L can be either inferred by a loop analysis step or given as a manual annotation.

```
void func (void) {
  a;
  while (b) {
    if (c)
      d;
    e;
  }
  f;
}
```

Figure 2.1: Abstracted ANSI- C program, where `b` and `c` are boolean expressions and `a`, `d`, `e`, and `f` denote program statements. The WCET of a statement or a block can be computed from the timing information of the expressions it consists of. The timing information is acquired either by a low-level timing analysis that abides the underlying architecture or supplied by hand (estimates).

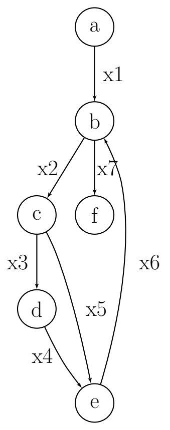


Figure 2.2: Graph representation of program flow. The execution frequencies are listed on edges. Timing information from a low-level analysis on graph edges is omitted. The node numbers correspond to the expression identifiers used in the annotations of Figure 2.1.

the loop bound `L` to be 10. The WCET for `func` is further derived by maximizing the sum of block costs in the process of satisfying the dependency relations of Figure 2.3. As a result, the WCET of `func` is computed to be 42 time units.

As shown in Example 2.1, the tighter the loop iteration bounds are (i.e. L), the more precise are the derived WCET. The accuracy of a WCET limit crucially depends on the precision of the available loop bounds. It is necessary to compute bounds for loops as exact as possible in order to get an accurate WCET. The difficulty of inferring precise upper bounds on loop iterations comes from the presence of complex loop arithmetic and control flow.

$$x1 = 1 \tag{2.1}$$
$$x2 = x1 * L \tag{2.2}$$
$$x3 = x2 - x5 \tag{2.3}$$
$$x4 = x3 \tag{2.4}$$
$$x5 = x2 - x3 \tag{2.5}$$
$$x6 = x5 + x4 \tag{2.6}$$
$$x7 = 1 \tag{2.7}$$

Figure 2.3: Execution frequencies of program expressions. Equations (2.1) and (2.7) state that the program is entered and exited once. Equation (2.2) states that the loop body is executed L times, where L denotes the number of loop iterations. For this example, L is set to be 10. The maximum solution of the equations using the appropriate timing information is the WCET of the program under analysis.

Our approach of proving WCET bound precise crucially relies on the IPET method, and makes use of the following IPET steps. IPET first translates the CFG of a program into an ILP problem. Next, it computes an ILP solution corresponding to the path with the highest edge-weight. For doing so, the following constraints on the CFG are used: (i) the program is entered and exited once, that is $\sum out(S) = \sum inc(X) = 1$; (ii) the execution frequency of incoming edges is equal to the execution frequency of outgoing edges, that is $\sum in(n) = \sum out(n)$; (iii) for each loop, the loop body is executed ℓ times the loop header, that is $\sum in(lb) = \ell * \sum out(n)$. The maximum solution to the above system of ILP constraints corresponds to the (estimation of the) program's WCET bound. Note that the ILP solution fixes the execution frequencies of program blocks, resulting in an *ILP branching behavior*, induced by the ILP, that encodes one or more execution traces in the CFG. The execution traces resulting from the ILP branching behavior are called *WCET candidates* and if they are feasible, they exhibit the calculated WCET.

A single ILP branching behavior can result in one or more execution traces in the CFG, as information about the exact sequence of edge executions for edges in loops is not available. Note that without additional constraints, IPET will always select the maximum execution frequency for those edges of conditional nodes with higher edge-weight. Thus, the solution of IPET in absence of additional constraints always encodes a single execution trace.

Let us consider now an execution trace containing a loop with a conditional in the loop body, such that the loop is executed ℓ times. Assume that the frequency of the false-edge f is m, for some $m \in \mathbb{N}$. Therefore, the frequency of the corresponding `true`-edge t is constrained to $\ell - m$. The ILP branching behavior then encodes multiple execution traces. For conditional-edges $e, e' \in \{t, f\}$, we write ee' to mean that the execution of edge e is followed by the execution of e'. Then, the set of branching behaviors for

$m = 1$ is $\{(f^1t^2t^3 \dots t^{\ell-1}), (t^1f^1t^2 \dots t^{\ell-1}), \dots, (t^1t^2t^3 ... t^{\ell-1}f^1)\}$, where t^i (respectively, f^i) denotes that the `true`-edge t (respectively, `false`-edge f) which was taken in the ith iteration of the loop. In the sequel, we will refer to the branching behavior describing a single execution trace as a path-expression. A single element in a path-expression is a branching decision for a conditional-node C. Note that traces given above are valid execution traces in the CFG. They are however not necessarily valid execution traces in the original program, where each condition is evaluated at runtime.

2.3 Symbolic Execution

We briefly overview the main ingredients of symbolic execution upon which our work relies on, and refer to [20, 72, 10] for details.

Symbolic execution uses symbolic instead of concrete input data to symbolically execute a program. To do so, (input) variables of the program are assumed to be "symbolic", which means that they can have an arbitrary value, conforming to the specified data-type. The same notions of path, execution trace, branching behavior and path expression defined for CFGs also apply for path-wise symbolic execution: a sequence of CFG branching decisions at the same time encodes a symbolic execution trace. Note, that a symbolic execution trace usually encodes multiple concrete execution traces.

If a conditional statement splits the control-flow of the program, symbolic execution follows both successor edges of the conditional, restricting possible values of symbolic variables according to the condition. For example, if a conditional executes the *true*-edge of the condition only if a variable has a certain constant value, then symbolic execution assumes the constant value for the variable when following this edge. Thus, symbolic variable values are restricted by path conditions or assumptions involving the respective variable. This allows to track complex constraints for each variable, and at the same time rely on solvers that can reason about the constraints to infer analysis results.

Symbolic execution of programs with conditionals (and loops) leads to the problem of path explosion, as the number of paths needed to be executed symbolically increases exponentially with the number of conditionals in the program. Hence, full symbolic coverage of larger applications is infeasible in practice. The problem of path explosion can be addressed in different ways, for example, by using heuristics for computing only partial symbolic coverage of the program, for instance in the context of test-case generation and bug-hunting.

2.4 Algebraic Considerations.

This section overviews some algebraic methods from symbolic computation, in particular from algorithmic combinatorics. We rely on these methods in our approach of automatically inferring loop bounds. A more detailed survey can be found in [28].

Let $\mathbb{K}$ be a field of characteristic zero (e.g. $\mathbb{R}$) and $f : \mathbb{N} \to \mathbb{K}$ a *univariate sequence* in $\mathbb{K}$. Consider a rational function $R : \mathbb{K}^{r+1} \to \mathbb{K}$. A *recurrence equation* for the sequence $f(n)$ is of the form $f(n + r) = R(f(n), f(n + 1), ..., f(n + r - 1), n)$, where $r \in \mathbb{N}$ is the *order* of the recurrence. We denote by $\mathbb{K}[n]$ the polynomial ring with indeterminante n and coefficients in $\mathbb{K}$. Further, for $n \in \mathbb{N}$, we denote by $n!$ the factorial of n.

Given a recurrence equation of $f(n)$, one is interested to compute the value of $f(n)$ as a function depending *only* on the summation variable n and some given initial vales $f(0), \ldots, f(n-1)$. In other words, a *closed form* solution of $f(n)$ is sought. Although solving arbitrary recurrence equations is an undecidable problem, special classes of recurrence equations can be effectively decided using algorithmic combinatorics techniques.

In our work of WCET analysis, we are interested in solving one particular class of recurrence equations, called *C-finite recurrences.* An *inhomogeneous* C-finite recurrence equation is of the form $f(n+r) = a_0 f(n) + a_1 f(n+1) + ... + a_{r-1} f(n+r-1) + h(n)$, where $a_0, \ldots, a_{r-1} \in \mathbb{K}$, $a_0 \neq 0$, and $h(n)$ is a linear combination over $\mathbb{K}$ of exponential sequences in n with polynomial coefficients in n. The C-finite recurrence is *homogeneous* if $h(n) = 0$. C-finite recurrences fall in the class of decidable recurrences. In other words, closed forms of C-finite recurrences can always be computed, as illustrated in the example below.

Example 2.1 Consider the C-finite recurrence $f(n+1) = 3f(n)+1$ with initial values $f(0) = 1$. By solving $f(n)$, we obtain the closed form $f(n) = \frac{3}{2} * 3^n - \frac{1}{2}$.

CHAPTER 3

Proof – WCET Squeezing as Proof Procedure

This section describes our approach to proving the precision of WCET bounds, which, to the best of our knowledge, is out-of-scope of any other state-of-the-art WCET analyzer. Our method, called *WCET Squeezing*, extends static approaches of WCET analysis by an automated and novel algorithm that allows not only to tighten a WCET bound reported by a static analyzer, but more importantly, it allows to conclude and proof that a WCET bound computed by an analyzer is indeed the actual WCET of the program. That is, any over-estimation of the bound is due to an imprecise modelling of the underlying hardware and not due to an infeasible path being analyzed.

Our WCET Squeezing algorithm works by viewing the problem specification and the results of an IPET-based WCET bound computation as an abstraction of the actual program. As the abstract behaviour of the program might not be exhibited in the concrete program, the program abstraction applied in the IPET step of our method can yield to an imprecise WCET bound: the computed WCET bound might be derived for a program path that is feasible in the abstract program but not in the concrete one. Therefore, in WCET Squeezing we map the abstract program path exhibiting the computed WCET bound to the concrete program and check whether the abstract program path is feasible in the concrete program. In other words, we check whether there exists a concrete program path that corresponds to the execution of the abstract path. For doing so, we propose a novel combination of symbolic execution with WCET analyis. If such a concrete path does not exist, the computed WCET bound was derived for an infeasible program path, and infeasibility of the computed WCET bound is also concluded. Therefore, we exclude the program trace exhibiting the infeasible WCET bound from the program abstraction, and re-iterate the ILP WCET bound calculation, deriving this way a new, and most importantly, refined (tightened) WCET bound. However, if the abstract program path encodes a feasible path in the concrete program, we can conclude that the WCET bound is exhibited by a concrete behaviour of the program (modulo imprecision in the underlying hardware model). Thus, with a proof for the feasibility of the path in the concrete program, we have a proof for the precision of the WCET bound. For proving feasibility of program paths, we use satisfiability modulo theory (SMT) reasoning in the theory of linear integer arithmetic, bit-vectors and arrays. We believe that the combination of symbolic execution, SMT reasoning and WCET analysis makes our method novel and highly-efficient; we are not aware of any other WCET analyzer exploiting the power of these techniques in a combination. Moreover, our approach overcomes the computational limitations of symbolic execution as follows. WCET Squeezing does not apply symbolic execution on every program path, usually resulting in the expensive problem of path explosion. Rather than that, the WCET bound derived at an arbitrary iteration of WCET Squeezing is used to apply symbolic execution selectively, that is only on the program paths exhibiting the WCET bound.

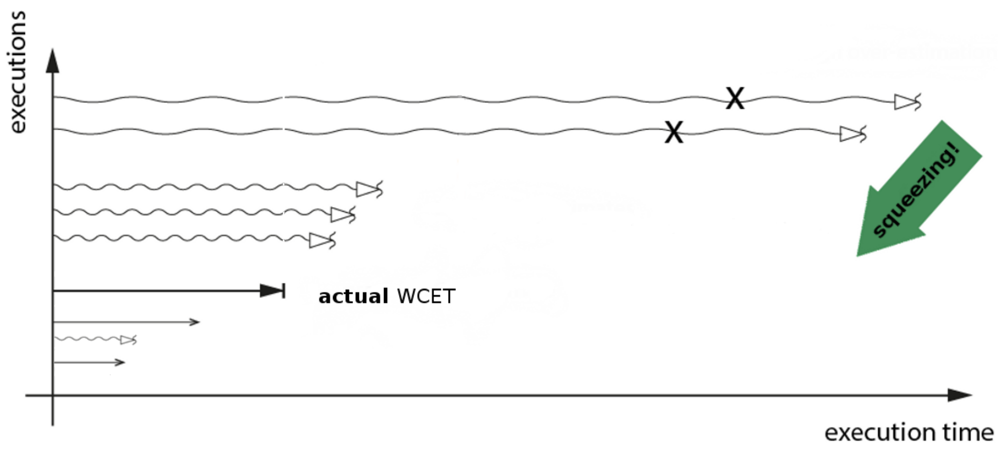


Figure 3.1: WCET Squeezing illustrated: long-waved (crossed-out) arrows represent infeasible paths with highly over-estimated WCET bounds. Short-waved arrows represent paths that exhibit an acceptably over-estimated WCET. The bold straight arrow represents the actual WCET of the program. The goal of WCET Squeezing is to prove infeasibility of the long-waved arrows and finding a WCET estimate for one of the acceptable short-waved arrows. In the best-case scenario, when WCET Squeezing terminates, the actual WCET of the program is found and proved.

To emphasize the advantage of WCET Squeezing in WCET analysis, let us consider Figure 3.1 summarizing different execution times of some arbitrary program, as computed by a WCET analyzer. The long-waved arrows represent executions that take much longer to execute than other executions, for example, the short-straight and short-waved executions. Due to an imprecise program analysis step, the WCET analyzer might only find one of the long-waved arrows as WCET, which is clearly an over-estimation, as the actual WCET of the program is much lower (represented by the bold short-straight arrow). WCET Squeezing automatically finds and excludes the long-waved over-estimated WCET bounds, and hence tightens the WCET bound of the program to be the WCET bound of one of the shorter-waved arrows.

In the traditional use of WCET analysis, one can only infer the WCET of a program. The traditional WCET problem hence answers the question: “what is the WCET of the program?”, but gives no hints or answers whether the derived WCET bound is precise or over-estimated. By using our WCET Squeezing method, the precision of the computed WCET bound can be however proved. In other words, WCET Squeezing allows us to strengthen the traditional task of a WCET problem, by also answering the questions of: “can the WCET of the program be tightened to a value X?” and “is the computed WCET bound precise?”. Based on WCET Squeezing algorithm, we can therefore define the pragmatic WCET problem as the decision problem of proving precise WCET bounds. That is, we consider the pragmatic WCET problem as the problem to decide whether an insufficient WCET bound reported by a static WCET analyzer can be refined and proved to be below some required threshhold. WCET Squeezing solves the pragmatic WCET problem by proving the precision of a WCET bound. In case the WCET bound is precise, our approach comes up with a proof of precision. In case the bound is over-estimated, our method returns a counterexample, proving the infeasibility of the WCET bound. The counterexample can further be used to improve the WCET bound in a next iteration of WCET Squeezing. WCET Squeezing can be iterated until either the WCET bound is precise and still above some required threshhold, or until the required thresshold has been reached.

WCET Squeezing is an *anytime* algorithm, meaning that it can be stopped at any time without violating the soundness of its results. This anytime property allows one to apply WCET Squeezing also when a program needs only be proven to be fast enough, that is, to be below some required limit. If the (initially) computed WCET bound of the program is above this limit, WCET Squeezing can be stopped when the refined WCET bound is either below the imposed limit (proving that the program meets the required timing constraint) or it is tight but above the considered limit (proving it can’t meet the timing constraint). Moreover, WCET Squeezing can also be used until a given time budget is exhausted to compute a tight(er) WCET bound for a program if there is need to. WCET Squeezing therefore opens new venues in WCET analysis, by solving WCET problems that are beyond the scope of other state-of-the-art approaches. These problems, and hence our contributions by using WCET Squeezing in WCET analysis, can be summarized as follows.

- WCET Squeezing is a method to *automatically tighten* the WCET estimate computed by a state-of-the-art WCET analyzer. To this end, our algorithm combines in a unique and novel way traditional IPET-based WCET analysis techniques with (selective) symbolic execution.

- WCET Squeezing works on-demand and can be used in three different modes, each of them being out of the scope of traditional WCET analyzers:

  (i) *Precision-controlled:* Computing a WCET that comes as close as possible to or even coincides with the actual WCET of a program.

(ii) *Threshold-controlled:* Squeezing the WCET as much as necessary in order to prove or disprove that a program meets a pre-defined deadline.

(iii) *Cost-controlled:* Tightening a WCET estimate within a pre-defined time budget allowed for WCET Squeezing.

One may argue that mode (i) is similar to the task of a traditional WCET problem, however, unlike traditional WCET bound approaches, WCET Squeezing allows one to also prove precision of the derived WCET bound. On the other hand, the tasks formulated in modes (ii) and (iii) are along the goals of the pragmatic WCET problem.

- WCET Squeezing addresses deadline-controlled and effort-controlled WCET analysis. We consider these two variants instances of the pragmatic WCET problem. WCET Squeezing also addresses the traditional WCET problem by tightening, and ultimately proving precise, some initially computed WCET bound.
- We present a new *selective symbolic execution framework* that is guided by WCET Squeezing using WCET bounds and program paths exhibiting WCET bounds.
- WCET Squeezing is fully automatic and requires no a-priori specified WCET templates or predicates. We *implemented our approach* in the r-TuBound tool and *evaluated* WCET Squeezing on the Mälardalen benchmarks of the WCET community [34]. Our experiments demonstrate that WCET Squeezing can significantly tighten the WCET estimates of programs, and often succeeds to compute a proven tight bound, at moderate costs (see Chapter 6 for details).

In the rest of the section, we first illustrate our method on concrete example and then formally introduce our WCET Squeezing algorithm.

3.1 Example

We illustrate WCET Squeezing on the example of Figure 3.2, taken from the `lcdnum.c` example of the Mälardalen bencmark suite [34].

This example consists of a for-loop with a conditional statement calling a function named `num_to_lcd`. An initial WCET analysis of this program infers a loop bound of 10, and yields a WCET bound of 24320 cycles, with the execution frequency of 10 for the `then`-branch (`true`-block) of the conditional inside the loop. By mapping back this WCET bound to a the concrete program, we only obtain the program path calling `num_to_lcd` in each loop iteration. WCET Squeezing uses this initial WCET bound and program path as a starting point for the first symbolic execution and concludes the infeasibility of this program path. Therefore, in the next step (second iteration) of WCET Squeezing, an IPET constraint excluding this infeasible program path is derived and added to the IPET-based WCET analysis framework. This way, a new WCET bound

```
// ...
for(i = 0; i < 10; i++) {
  if(i < 5) {
    a = a & 0x0F;
    OUT = num_to_lcd(a);
  }
}
// ...
```

Figure 3.2: `lcdnum.c`, simplified.

of 23420 cycles is then derived, with an execution frequency of 9 for the `true`-block in the loop, which constitutes a WCET tightening of 3.7%.

In the second iteration of WCET Squeezing, the new WCET bound 23420 is used.

This time, there are multiple program paths exhibiting the WCET bound in the concrete program, since the `else`-branch (`false`-block) of the conditional can be also taken in a loop iteration. By using symbolic execution, none of these program paths are however found feasible. A third iteration of WCET Squeezing is therefore taken, ruling out the infeasible program paths with a `true`-block frequency of 8. The WCET bound of the program is hence tigthened to 22520 cycles. Note that, compared to the initial WCET bound, an accumulated WCET improvement of 7.4% is obtained, after excluding 11 program paths.

The number of program paths explored by symbolic execution increases in the following iterations of WCET Squeezing. WCET Squeezing finally terminates in its sixth iteration, when it finds a feasible program path with execution frequency of 5 for the `true`-block. This feasible program path results in a *precise* WCET bound of 19820 cycles. Compared to the initial WCET estimate, the precise WCET bound yields an improvement of 18.5%.

3.2 WCET Squeezing for Proving Precise WCET Bounds

We now describe our *WCET Squeezing* algorithm for proving precise, and if necessary tightening, the WCET bound provided by some off-the-shelve WCET analyzer.

WCET Squeezing applies on-demand WCET feasibility refinement, and proceeds as follows. It takes as input the result of an a-priori WCET analysis of the program and refines, that is *squeezes*, this WCET bound by using symbolic execution [19, 18, 72] in combination with the Implicit Path Enumeration Technique – IPET approach of [58], where the IPET problem is ultimately encoded as an Integer Linear Program (ILP). To squeeze the computed WCET bound of a program, we map the result of the IPET analysis to a program trace and symbolically execute this trace to decide whether the path is feasible or not. To this end, the following two steps are applied. (i) If the program path is feasible, the computed WCET bound is tight: any remaining WCET over-estimation is due to a conservative hardware model. (ii) If the program path is infeasible, we extend the original IPET problem by a new constraint excluding the infeasible path. The new IPET problem is again encoded as an ILP problem, which is then solved, resulting in a tighter WCET bound. The new ILP problem yields a new program path for the WCET bound, which is then used in the WCET Squeezing. These two steps of WCET Squeezing can be iteratively run until termination, that is until a feasible program path exhibiting the WCET bound is found. It can also be run with an optional time-limit after which the algorithm might terminate without finding a feasible trace.

WCET Squeezing avoids the short-comings of IPET and symbolic execution, namely the lack of program knowledge beyond flow facts for IPET and the non-scalability of symbolic execution for a fast growing number of paths. Even more, when evaluated on examples coming from the Mälardalen benchmark suite [34], our experiments show that WCET Squeezing is very efficient. For example, we achieved an improvement of WCET bound of up to 9% after two iterations of WCET Squeezing. Improvements of up to 90% are possible, if WCET Squeezing is run until termination.

WCET Squeezing has similarities to the Counter-example Guided Abstraction Refinement (CeGAR) approach of [24]. In the CeGAR method, an initial abstraction of the program is analyzed for reachability of error states. If an error state is spurious, that is reachable in the abstraction but not in the program, the abstraction is refined to exclude reachability of the error state. The refined abstraction is used in the next iteration of CeGAR. Similarly, in WCET Squeezing, we start with an initial abstraction of the program, where the abstraction is encoded as an ILP problem. A solution of the ILP problem represents a WCET program path of the program. This path might however only be feasible in the considered abstraction and not in the concrete program. Therefore, we next symbolically execute the path and, if found infeasible, the abstraction is refined to exclude the path. This refined abstraction is used in the next iteration of WCET Squeezing. Note that the used program abstraction gets more precise in each iteration of our WCET Squeezing algorithm, yielding a more precise WCET estimate for the program.

WCET Squeezing - The Algorithm

Our WCET Squeezing algorithm is given in Algorithm 3.1. Algorithm 3.1 takes as input the result of an a-priori WCET analysis of the program. That is, Algorithm 3.1 takes as input the ILP problem `ilp_problem` resulting from applying IPET on the CFG of the program under study. Remember that a maximum solution of the `ilp_problem` gives an (initial) WCET bound of the problem. Algorithm 3.1 then iteratively refines the WCET bound of the program, in a *precision-controlled* mode, by reducing the control flow of the program (i.e. excluding infeasible program paths). In addition to the ILP problem, an optional parameter can also be supplied to guarantee termination of the algorithm within a certain time-limit `time`, that is in a *cost-controlled* mode: if during that time a program path exhibiting the current WCET bound is excluded, the WCET bound is improved, unless another program path also exhibits the current WCET bound. Alternatively, when running WCET Squeezing with a pre-defined *threshold*-value, our method allows to solve the pragmatic WCET problem, that is to answer whether a pre-defined deadline can be met by a program: WCET Squeezing is run until either the improvement in the WCET bound reaches the required value, reporting then *yes*, or until it terminates due to a feasible program path, reporting then *no* as the answer of the pragmatic WCET problem. This mode of WCET Squeezing corresponds to a *threshold-controlled* mode.

Let us emphasize that WCET Squeezing is guaranteed to terminate, even without using a pre-defined time- and/or threshold-limit. This is so because the ILP problems during WCET Squeezing encode only a finite number of WCET trace candidates. In the worst-case scenario, WCET Squeezing terminates after symbolically executing all program traces. It is however important to note that the WCET bound is improved at every iteration of Algorithm 3.1, and hence the WCET bound, reported upon the termination of Algorithm 3.1, is improved and proved to be precise.

Algorithm 3.1 WCET Squeezing Algorithm
Input: ILP problem `ilp_problem`
Output: ILP solution `ilp_solution`
Assumption: threshold- or cost-limit `limit`

```
1  begin
2  do
3    $ilp\_solution := ILPsolve(ilp\_problem)$
4    $wcet\_candidates := extractCandidates(ilp\_problem, ilp\_solution)$
5    $counter\_ex := symbolicExecution(wcet\_candidates)$
6    if no $counter\_ex$ then return $ilp\_solution$
7    $ilp\_problem := encodeConstraint(ilp\_problem, counter\_ex)$
8  forever or [optional] until limit is reached
9  return $ilp\_solution$
10 end
```

The main steps of Algorithm 3.1 are as follows. First, a solution `ilp_solution` of the ILP problem `ilp_problem` is computed (line 3), by using an off-the-shelve ILP solver [8]. Based on the computed ILP solution, the corresponding ILP branching behavior is mapped back to the CFG of the program and program paths as WCET trace candidates are extracted (line 4), as presented further. These WCET trace candidates are next symbolically executed (line 5), in a selective manner, as later discussed. The result of symbolic execution on WCET trace candidates is stored in the counterexample `counter_ex`: if a trace candidate is feasible, the `ilp_solution` corresponding to this trace is returned (line 6) as the precise WCET bound of the program under study. If all WCET trace candidates are infeasible, the ILP branching behavior is infeasible as well. The WCET bound corresponding to the *ilp_solution* is hence not exhibited by the program and the infeasible ILP branching behavior is excluded by adding a constraint to the ILP problem (line 7), as addressed in the later part of this section. A next iteration of WCET Squeezing is further applied on the new ILP problem, yielding a tighter WCET bound of the program (line 3). Algorithm 3.1 for WCET Squeezing terminates when a feasible and refined WCET bound is derived (line 6). Even more, this WCET bound is guaranteed to be precise, as a feasible WCET trace exhibiting the bound was also identified.

In what follows, we overview the ingredients of Algorithm 3.1 in more detail. We describe our approach to extracting WCET trace candidates, selective symbolic execution, and encoding of ILP constraints.

WCET Trace Candidates

To construct WCET trace candidates, a mapping from the ILP branching behavior to program execution traces is needed. WCET trace candidates can be specified by a branching behavior, i.e. a sequence of branching decisions. The ILP branching behavior, denoted by *ilp_bb*, is initially generated by mapping all executed edges, that is edges with an execution frequency greater than 0, from the first ILP solution to a trace in the CFG of the program and selecting all conditions executed in the trace. The ILP branching behavior is represented as a sequence of executed conditions C_i, where C_i is the ith condition of the trace, and it has the execution frequencies $\mathit{freq}(t_{C_i})$ and $\mathit{freq}(f_{C_i})$ of its conditional-edges associated with it. The values of $\mathit{freq}(t_{C_i})$ and $\mathit{freq}(f_{C_i})$ are as given in the first ILP solution.

From the ILP branching behavior *ilp_bb*, WCET trace candidates are constructed by specifying their branching behavior *bb*. A branching behavior *bb* forms an execution trace, where the ith element of *bb*, denoted by $bb[i]$, stores the evaluation of the executed path-condition C_i. We refer to $bb[i]$ as the ith branch decision of *bb*. Depending on the execution frequencies of the conditional-edges $e \in \{t_{C_i}, f_{C_i}\}$ of C_i in *ilp_bb*, multiple branching behaviors *bb* can be constructed from *ilp_bb*, as follows.

Case 1. One conditional-edge of C_i is executed once. If only one of the conditional-edges e of C_i is executed with $\mathit{freq}(e)$, no interleaving among branch-conditions is possible. Therefore, a single WCET trace candidate is constructed whose $\mathit{freq}(e)$ positions are set either to t or f results. Assuming that the execution frequency of t_{C_i} (respectively, f_{C_i}) is 1, the path-condition C_i is assumed to evaluate to `true` (respectively, `false`), hence the branching behavior at position i is set to t (respectively, f). Using our previous notation, in this case we have:

$$bb[i] = \begin{cases} t, & \text{if } \mathit{freq}(t_{C_i}) = 1 \\ f, & \text{if } \mathit{freq}(f_{C_i}) = 1 \end{cases}$$

Case 2. One conditional-edge of C_i is executed repeatedly. If the conditional-edge e of C_i is executed with a frequency higher than 1, we need to encode the multiple executions of e. To this end, multiple positions in *bb* are set to either t or f, as follows:

$$\begin{cases} bb[i+j] = t \quad \text{with} \quad 0 \le j \le \mathit{freq}(t_{C_i}), \\ \qquad \text{if } \mathit{freq}(t_{C_i}) > 0 \text{ and } \mathit{freq}(f_{C_i}) = 0 \\ bb[i+j] = f \quad \text{with} \quad 0 \le j \le \mathit{freq}(f_{C_i}), \\ \qquad \text{if } \mathit{freq}(f_{C_i}) > 0 \text{ and } \mathit{freq}(f_{C_i}) = 0 \end{cases}$$

That is, a sequence of t (respectively, f) of length $\mathit{freq}(t_{C_i})$ (respectively, $\mathit{freq}(f_{C_i})$) is set starting from *ilp_bb*$[i]$. Note that for multiple conditionals inside a loop, the values of *bb* must be set such that their indices coincide with the corresponding branching-decision in the trace.

Case 3. Both conditional-edges of C_i are executed repeatedly. If the ILP branching behavior specifies the execution of both conditional-edges of C_i inside a loop, the branching-decisions can interleave, and hence *ilp_bb* encodes multiple WCET trace candidates. The number of WCET trace candidates constructed from *ilp_bb* is then given by the number of all possible permutations over the set of edges
$\mathcal{S} = \{ \underbrace{t \dots t}_{\mathit{freq}(t_{C_i}) \text{ times}}, \underbrace{f \dots f}_{\mathit{freq}(f_{C_i}) \text{ times}} \}$. Using the results of [63], the number of permutations over S, and thus the number of loop branching behaviors is: $p = \frac{(\mathit{freq}(t_{C_i}) + \mathit{freq}(f_{C_i}))!}{(\mathit{freq}(t_{C_i})!) \, * \, (\mathit{freq}(f_{C_i})!)}$.

In this case, we take care of the multiple branching behavior as follows. We construct p copies of the current *bb*, that is we take $bb_1, \dots, bb_p$ branching behaviors where each bb_x is a copy of *bb*. Next, each bb_x is continued by one loop branching behavior, as given below:

$$bb_x[i+j] = \mathit{permutation}_x\{ \underbrace{t, \dots, t}_{\mathit{freq}(t_{C_l}) \text{ times}}, \underbrace{f, \dots, f}_{\mathit{freq}(f_{C_l}) \text{ times}} \}$$
$$\text{with} \quad 0 \le j \le \mathit{freq}(t_{C_i}) + \mathit{freq}(f_{C_i}) \quad \text{and} \quad 0 \le x \le l,$$

where $\mathit{permutation}_x\{S\}$ gives the xth permutation over S.
Note that for *bb* the correspondence between an index i and an executed condition C_i is not one-to-one; previous conditions C_k with $k < i$ might have already set multiple positions, including *bb*$[i]$, in *bb*.

Selective Symbolic Execution

Selective symbolic execution supports the analysis of a computable or measurable property (i.e. WCET) of a program under study, while exploring only the relevant program parts (i.e. trace candidate) for analyzing the property. Our goal with selective symbolic execution is to minimize the number of symbolic executions required in order to improve on analysis results. The WCET Squeezing approach combines a symbolic execution engine with a WCET analysis toolchain and uses WCET bounds to guide selective symbolic execution, by symbolically executing only those traces that might exhibit the WCET bounds.

Using the branching behavior *bb* of the WCET trace candidates, the symbolic execution engine of Algorithm 3.1 directs the program execution along these traces and, for each trace, checks the feasibility of the conditions on each branching point. A symbolically evaluated execution trace is feasible if the conjunction of all path conditions is satisfiable, meaning that the execution trace is a feasible program execution trace. As the symbolic execution engine is precise, it serves as an oracle to decide whether the ILP branching behavior is a feasible branching behavior in the concrete program.

For doing so, our symbolic execution step in line 5 of Algorithm 3.1 proceeds as follows. It takes as input the source code of the program and the branching behavior *bb* of one of the WCET trace candidates. The symbolic execution engine then constructs a satisfiability module theory (SMT) representation [7] of the program execution, according to the branching behavior together with the source. A branching behavior *bb* of length n specifies the evaluations of n path-conditions, which can be analyzed for satisfiability in the SMT representation provided by the symbolic execution engine. That is, if the specified evaluation of the path-condition is unsatisfiable at some point, the trace $\pi(bb)$ is infeasible. Using our previous notations, we conclude that $\pi(bb)$ is infeasible iff the boolean expression:

$$\begin{aligned} &\mathit{symbolicEval}(C_0, bb[0]) \ \wedge \ \mathit{symbolicEval}(C_1, bb[1]) \wedge \\ &\ \ldots \ \wedge \ \mathit{symbolicEval}(C_i, bb[i])) \end{aligned} \tag{3.1}$$

is unsatisfiable, for $i \leq n$. To reason about the unsatisfiability of this condition, we use SMT reasoning in the theory of linear integer arithmetic, bit-vectors and arrays (see Chapter 6 for further details).

If the branching behavior *bb* gives a satisfiable evaluation of (3.1), the WCET trace candidate corresponding to *bb* yields a successful symbolic execution. Hence, the WCET trace candidate is feasible and exhibits the current WCET bound. Therefore, no further WCET refinements are possible (line 6 of Algorithm 3.1).

Otherwise, if the symbolic execution of a trace candidate fails, some path-condition C_i in (3.1) is unsatisfiable for some i. This condition C_i can be mapped to its conditional nodes, resulting in an ILP encoding of an infeasible WCET trace candidate. The encoding gives thus a counter-example that needs to be excluded from the ILP branching behavior in the next iteration of WCET Squeezing (line 7 of Algorithm 3.1). The constraint constructed from this counter-example involves all symbolically executed conditions, as detailed in the next paragraph.

ILP Constraint Encoding

New ILP constraints are derived from infeasible WCET trace candidates. If a WCET trace candidate induced by an ILP branching behavior is infeasible, the trace must be excluded from from further WCET computations. This is done by adding an ILP constraint to exclude the current maximum solution from the new ILP problem.

The construction of the ILP constraint is such that it decreases the total sum of execution frequencies of all conditional-edges that were symbolically executed until infeasibility was inferred (including the unsatisfiable one). That is, for an infeasible WCET trace candidate π, the ILP constraint constructed involves all conditional-edges corresponding to C_i from (3.1). Using our above notation, recall that $bb[i]$ gives the conditional-edge

```
void f () {
    if (C1) ...
    if (C2) ...
}
```

Figure 3.3: Conditions `C1` and `C2` are mutually exclusive.

(t or f) from the ith position of bb. $bb[i]_{C_i}$ is the conditional edge of C_i, denoted t_i or f_i in the ILP. Then, the conditional-edges of (3.1) over which a new ILP constraint is constructed are given by $bb[0]_{C_0}$, $bb[1]_{C_1}$, …, $bb[i]_{C_i}$. To ensure that the execution frequencies of these conditional-edges is decreased, the new ILP constraint we add to the ILP problem is:

$$bb[0]_{C0} + bb[1]_{C1} + \cdots + bb[i]_{Ci} \leq$$
$$\mathit{freq}(bb[0]_{C0}) + \mathit{freq}(bb[1]_{C1}) + \cdots + \mathit{freq}(bb[i]_{Ci}) - 1.$$

We illustrate our approach to ILP construction on the example below.

Example 3.2 Consider Figure 3.3, where conditions C_1 and C_2 are mutually exclusive. The (abstracted) CFG representation of Figure 3.3 is given in Figure 3.4(a). The initial ILP solution yields a WCET trace candidate with branching behaviour tt, i.e. an execution frequency that enables the execution of both $t1$ and $t2$ (both with execution frequency 1). However, as conditions C_1 and C_2 are mutually exclusive, only one of the conditions can be true. Therefore, symbolic execution will set C_1 to true, execute $t1$, and infer that the evaluation of $C_2 =$ true is unsatisfiable, hence execution of $t2$ is infeasible. The constraint constructed from the result of symbolic execution will specify in the resulting ILP problem that either C_1 or C_2 (but not both) is valid, by decreasing the combined execution frequency of all conditionals edges executed, i.e. of $t1$ and $t2$.

In more detail, the new ILP constraint added to the ILP problem of Figure 3.3 specifies the following properties: (i) the entry-edge n and the exit-edge x of the CFG of Figure 3.4(a) is executed at most once, that is $n \leq 1$ and $x \leq 1$; (ii) the conditional-edges of Figure 3.4(a) are executed at most once, that is $t1 + f1 \leq n$ and $t2 + f2 \leq n$; (iii) the total execution frequency of the *true*-edges of C_1 and C_2 is restricted to 1, that is $t1 + t2 \leq 1$. The derived ILP problem is thus the conjunction of the afore-listed five integer inequalities.

As illustrated in the example above, the ILP constraints constructed from infeasible WCET trace candidates restricts the total sum of execution frequencies of all conditional-edges involved in the trace. It therefore excludes the ILP branching behavior and an ILP solution picks at least one conditional-edge differently, due to the *semantic-based* encoding.

Alternatively, the *syntactic-based* encoding introduces additional ILP variables and corresponds to a CFG transformation (Figure 3.4) that removes only the infeasible trace from the CFG. On the graph representation of the CFG, the following two transformations are applied: (i) the branching decision is made explicit in the CFG by copying

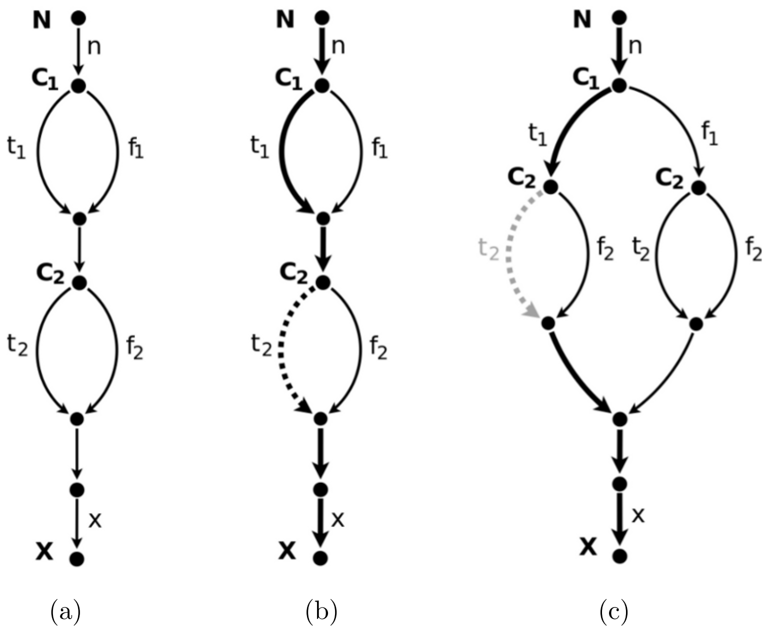


Figure 3.4: (a) CFG representation of Figure 3.3; (b) Infeasible WCET trace candidate (bold) with an unsatisfiable conditional edge (dotted); (c) Transformed CFG, excluding the infeasible trace of (b).

and pulling up the unsatisfiable condition into the predecessor conditional node; (ii) the unsatisfiable edge for the prefix of the infeasible WCET trace candidate is removed.

Example 3.3 Consider again the program code given in Figure 3.3 and the CFG representation in Figure 3.4(a). Let $t1$ and $t2$ respectively denote the true-edge of C_1 and C_2. Assume that the ILP branching behavior for Figure 3.3 is initially tt, that is a WCET trace candidate executing $t1$ followed by $t2$. This WCET trace candidate is inferred infeasible by symbolic execution because the evaluation of condition C_2 to `true` (i.e. executing conditional edge $t2$) is unsatisfiable. Therefore, the trace is removed from the CFG of the program: the conditional node of the unsatisfiable edge is copied pulled up into the last conditional node and the *true*-decision edge is removed for the prefix of the candidate.

Implementation Pragmatics of ILP Encodings in the Presence of Loops. The difference between the syntactic- and the semantic-based encoding in the above examples only effects the number of ILP variables and constraints. In the presence of loops, using the syntactic-based encoding also increases the number of ILP variables and constraints, while using the semantic-based encoding increases the number of WCET trace candidates.

Remember that the execution frequencies of edges inside loops are constrained to their execution frequency times the loop bound. That is, for an edge that is executed m times inside a loop, the following constraint is generated: $\mathit{freq}(e) \leq \ell * m$, This needs to be taken into consideration for both encodings when computing the total execution frequency for edges inside loops, as illustrated in the following example.

```
void main () {
  int i;
  bool exec = false;
  if (*)
    exec = true;       // t1
  for (i = 0; i < 5; i++)
    if (exec == false) {
      expensive();     // t2
      exec = false;
    } else
      exec = true;     // f2
}
```

Figure 3.5: Executing the *true*-edge of the first conditional restricts the execution of the *true*-edge inside the loop. `*` denotes non-deterministic choice, `f1` is not executing `t1`

```
       n <= 1;
      c1 <= n;
 t1 + f1 <= c1;
loopHead <= 1;
loopBody <= loopHead * 5;
loopBody <= t2 + f2;
loopExit <= loopHead;
       x <= loopExit;
```

Figure 3.6: ILP problem after WCET analysis of the example in Figure 3.5. The branching behavior imposed by the ILP solution is *tttttt*.

Example 3.4 Consider Figure 3.5 and assume that the first true-block (edge t_1, marked as `t1`) has an execution time of 1, whereas the second true-block (edge t_2, marked as `t2`) has 10. All other costs are ignored. The initial ILP solution selects all true-blocks to be executed, the t_1 with frequency 1 and the t_2 with frequency 5. The reported WCET estimate amounts to $1 * 1 + 5 * 10 = 51$, the branching behavior for the trace is *tttttt* (i.e. $t_1 t_2^1 \dots t_2^5$).

Symbolically executing this trace yields the unsatisfiability of the second condition (`exec == false`), in the first iteration of the loop. Thus, if the first branching decision is t, the execution frequency of the true-edge in the loop is only 4. Just reducing the frequency in the ILP problem to 4 yields an invalid result. The WCET estimate would be restricted to $1 * 1 + 4 * 10 = 41$, even though there exists a path exposing a higher WCET: Executing the *false*-branch of the first conditional (branching behavior *fttttt*) leads to a WCET of $0 * 1 + 5 * 10 = 50$ and is allowed in the original problem.

We denote the situation illustrated in Example 3.4 as *candidate-flip*. If the prefix to a loop flips, then the constraints about loop iterations need to be inferred again since it increases the number of symbolically executed WCET trace candidates. Both the syntactic- and the semantic-based encoding can handle candidate-flips, but based on our experience, a combination of the two encodings handles them the best.

On CFG level, the syntactic-based approach peels-off a loop iteration [50] and introduces a new conditional for the loop condition in the first iteration and children for the loop body. Similar to Example 3.3, the last conditional is split, introducing copies for all following edges, and then the infeasible trace is removed. On the ILP level, the same technique is applied: additional variables for the copies of the condition and the loop-peel are introduced in the ILP problem. At the same time, the number of WCET trace candidates is smaller, as branching in different iterations is explicitly encoded.

The semantic-based encoding constructs, as before, a constraint that decrements the total execution frequency of all conditional-edges up to the loop and the execution frequency of the conditional-edge in the loop. Additionally to the original ILP problem (Figure 3.6), the constraint restricts the combined execution frequency of the *true*-edge (`t1`) of condition `c1` and the *true*-edge (`t2`) of the condition in the loop body, i.e. `t1 + t2 <= 5`. (Figure 3.7). This constraint introduces no new ILP variables, but the ILP branching behavior encodes more WCET trace candidates.

The combined encoding uses the syntactic-based encoding to peel one loop iteration and decrements the execution frequency of edges inside the loop. It does not split conditional nodes, instead a semantic-based encoding is used to restrict the total execution frequency of the loop prefix and the peeled condition, such that the branching decision in the first iteration of the loop is explicit for the loop prefix. The branching decision in the first iteration of the loop is not restricted for other executions. Figure 3.8 depicts the resulting ILP problem where the solver will return the correct WCET estimate, i.e. branching behavior *fttttt*, exhibiting a WCET of 50. The advantage of the combined encoding is that it only introduces ILP variables and constraints for the peeled loop iteration and, at the same time, it also makes explicit the branching behavior, thus reducing the number of symbolically executed WCET candidates.

```
       n <= 1;
      c1 <= n;
 t1 + f1 <= c1;
loopHead <= c1;
loopBody <= loopHead * 5;
loopBody <= t2 + f2;
loopExit <= loopHead;
 t1 + t2 <= 5;
       x <= loopExit;
```

Figure 3.7: Semantic constraint.

```
            n <= 1;
           c1 <= n;
      t1 + f1 <= c1;
     peelCond <= c1;
peelT + peelF<= peelCond;
  t1 + peelT <= 1;
   loopEntry <= peelCond;
    loopBody <= loopEntry * 4;
    loopBody <= t2 + f2;
    loopExit <= loopEntry;
           x <= loopExit;
```

Figure 3.8: Combined constraint.

Symbolic Execution Overhead and Benefits.

In contrast to formal symbolic techniques, such as model checking, that analyze the program as a whole, symbolic execution reasons in a path-local manner. Symbolic execution is especially suited for automated testing but also found applications in program verification and bug-hunting [19, 72]. It allows for precise analysis of programs but the number of program paths that need to be analyzed increases exponentially with the number of conditionals. Therefore, applications of symbolic execution often target only partial symbolic coverage of the program, e.g. generating test-cases that achieve high line coverage [18].

One of the major advantages of symbolic execution engines is the amount of information they can infer about a program, as they implicitly carry all this information along when they explore a program. Applications of symbolic execution for loop bound refinement in WCET analysis is briefly addressed in our work presented in [44]. There, symbolic execution is used to infer and validate arithmetic properties about program loops, in some cases allowing to refine the computed loop bound.

Unlike these methods, our selective symbolic execution approach in WCET Squeezing relies on the tight interaction between a traditional static WCET analysis toolchain applying IPET and a symbolic execution engine that allows to select symbolic execution traces for reasoning precisely about path-conditions. We exploit the fact that loop bounds are implicitly supplied by the initial WCET analysis. Compared to traditional flow-fact analysis, the symbolic execution component of WCET Squeezing infers precise constraints for paths, that yield further WCET bounds. In each iteration of WCET Squeezing, solving the new ILP problem allows to refine the WCET bound. Compared to traditional symbolic execution engines, WCET Squeezing offers a way to identify precisely which paths need to be symbolically executed in order to improve the analysis.

Limitations of WCET Squeezing

The current major limitations of WCET Squeezing include the following:

(1) WCET Squeezing requires a successful initial WCET analysis of the program. That is, we assume that the program can be analyzed in a fully automatic way and loop bound for every program loop are computed by an initial WCET. These assumptions, especially the latter one, might to be too strong when it comes to programs with complicated control flow, including loops and recursion. If for such cases, no WCET analyzer cannot be used, our WCET Squeezing algorithm would also fail.

(2) For checking feasibility of WCET trace candidates, we rely on SMT reasoning the combined theory of linear integer arithmetic, bit-vectors and arrays. This means, that programs handling more complicated arithmetic operations and/or other data types cannot yet be handled by our method. For example, programs implementing complex polynomial operations over integers or programs with floating point operations are limitations to WCET Squeezing. Extending our method with more sophisticate WCET analysis methods, for example by the work of Astree [13], is an interesting task to be further investigated.

(3) We perform intra-procedural symbolic execution on the source level of the program. It is therefore crucial that both the source and the binary code of the program exhibit a compatible branching behaviour. While this assumption imposes limitations on WCET Squeezing, we believe that overcoming them could be done, for example, by performing symbolic execution on the binary level, or by prohibiting compiler optimizations that introduce incompatible branching behaviours on the source and binary code.

CHAPTER 4

Performance – Accelerating WCET Squeezing by Empowering WCET Analyzers

The WCET Squeezing algorithm described in Chapter 3 empowers state-of-the-art WCET analyzers by a novel approach of proving precise the WCET bound computed by a WCET analyzer. While effective, the performance of WCET Squeezing crucially depends on the analysis strength of the used WCET analyzer, in particular on the ability of the WCET analyzer to infer challenging and precise flow facts about virtual method calls, aliasing of pointer or array variables, and upper bounds on the number of program loop executions. To make WCET Squeezing as proof procedure highly performant, it is therefore important to improve WCET analysis with the ultimate goal of computing program flow facts of high quality.

In this section we address this challenge and present a new kind of static loop analysis for *inferring tight upper bounds, i.e. loop bounds, on the number of program loop iterations*. Our method relies on symbolic computation and automated theorem proving methods and introduces an efficient and automated framework for computing loop bounds. To this end, our work addresses special classes of loops with assignments and conditionals, where updates over program variables are linear expressions. For such loops, we deploy recurrence solving and theorem proving techniques and automatically derive tight iteration bounds, as follows.

(i) A loop with multiple paths arising from conditionals is first transformed into a loop with only one path. We call a loop with multiple paths, respectively with a single path, a multi-path loop, respectively a simple loop. To this end, the control flow of the multi-path loop is analyzed and refined using *SMT reasoning over arithmetical expressions*. The resulting simple loop soundly over-approximates the multi-path loop. Iteration bounds of the simple loop are thus safe iteration bounds of the multi-path loop.

(ii) A simple loop is next rewritten into a set of *recurrence equations* over those scalar variables that are changed at each loop iteration. To this end, a new variable denoting the loop counter is introduced and used as the summation variable. The recurrence equation of the loop iteration variable captures thus the dependency between various iterations of the loop.

(iii) Recurrence equations of loop variables are next solved and the values of loop variables at arbitrary loop iterations are computed as functions of the loop counter and the initial values of loop variables In other words, the *closed forms of loop variables* are derived. Our framework overcomes the limitations of missing initial values by a simple *over-approximation of non-deterministic assignments* .

We note that solving arbitrary recurrence equations is undecidable. However, in our approach we only consider loops with linear updates. Such loops yield C-finite recurrences, and hence our method always succeeds in computing the closed forms of loop variables.

For solving C-finite recurrences we deploy a *pattern-based recurrence solving algorithm.* In other words, we instantiate unknowns in the closed form pattern of C-finite recurrences by the symbolic constant coefficients of the recurrence to be solved. Unlike powerful algorithmic combinatorics techniques that can solve complex recurrences, our framework hence only solves a particular class of recurrence equations. However, it turned out that in WCET analysis the recurrences describing the iteration behavior of program loops are not arbitrarily complex and can be solved by our approach (see Chapter 6 for experimental results).

(iv) Closed forms of loop variables together with the loop condition are used to express the value of the loop counter as a function of loop variables. The upper bound on the number of loop iterations is finally derived by computing the *smallest value of the loop counter* such that the loop is terminated. To this end, we deploy SMT reasoning over arithmetical formulas. The inferred iteration bound is further used to infer an accurate WCET of the program loop.

We believe that our work advances the state-of-the-art in WCET analysis by a conceptually new and fully automated approach to loop bound computation. Moreover, our approach extends the application of program analysis methods by integrating WCET techniques with recurrence solving and SMT reasoning. We implemented our approach in the r-TuBound toolchain [44] and evaluated it on a large number of examples coming from the WCET community. Our results give practical evidence of the efficiency of our method - we refer to Chapter 6 for details.

In what follows, we first give an example illustrating our approach to loop bound computation, and then formally describe the main steps of our method.

4.1 Example

Consider the program in Figure 4.1 manipulating a two-dimensional array `a`, where `a[k][j]` denotes the array element from the `k`th row and `j`th column of `a`. Between the program lines 5-21 , the method `func` iterates texttta row-by-row and updates the elements of `a`, as follows. In each visited row `k`, the array elements in columns 1, 4, 13, and 53 are set to 1 according to the C-finite update of the simple loop from lines 6-9. Note that the number of visited rows in `a` is conditionalized by the non-deterministic assignment from line 2. Depending on the updates made between lines 6-9, the multi-path loop from lines 10-14 conditionally updates the elements of `a` by -1. Finally, the abrupt termination of the multi-path loop from lines 15-18 depends on the updates made throughout lines 6-14.

```
1   void func()
2   {
3     int i = nondet();
4     int j, k = 0;
5     int a[32][100];
6     for ( ; i > 0; i = i >> 1) {
7       for (j = 1; j < 100; j = j * 3 + 1) {
8         a[k][j] = 1;
9         #pragma wcet_loopbound(4)
10      }
11
12      for (j = 0; j < 100; j++) {
13        if (a[k][j] == 1) j++;
14        else a[k][j] = -1;
15        #pragma wcet_loopbound(100)
16      }
17
18      for (j = 0; j < 100; j++) {
19        if (a[k][j] != -1 && a[k][j] != 1)
20          break;
21        #pragma wcet_loopbound(100)
22      }
23       k++;
24       #pragma wcet_loopbound(32)
25    }
26  }
```

Figure 4.1: Program loops annotated with the result of loop bound computation.

Computing a tight WCET bound of the `func` method requires thus tight loop bound for the loops between lines 5-21. The difficulty in computing the number of loop iterations comes with the presence of the non-deterministic initialization and shift updates of the loop from lines 5-21; the use of C-finite updates in the simple loop from lines 6-9; the conditional updates of the multi-path loop from lines 10-14; and the presence of abrupt termination in the multi-path loop from lines 15-18. Our approach overcome these difficulties as follows.

- We apply a pattern-based recurrence solving algorithm, and the iteration bound of the loop from lines 6-9 is inferred to be precisely 4.
- We deploy SMT reasoning to translate multi-path loops into simple ones, and derive 100 to be the over-approximated loop bound of the loop from lines 10-14, as well as of the loop from lines 15-18.
- We over-approximate non-deterministic initializations, and infer the value 31 as the upper bound of the loop from lines 5-21.

The loop bounds inferred automatically by our approach are listed in Figure 4.1, using the program annotations `♯pragma wcet_loopbound(...)`.

4.2 Automated Generation of Loop Bounds for Empowering WCET Analyzers

This section overviews our approach to automatically derive tight bounds on the number of iterations for special classes of program loops. Automated generation of loop bounds is a challenging research topic, which has received considerably attention both in program and WCET analysis. One line of research closed to our method uses powerful symbolic computation algorithms to derive loop bounds, see e.g. [12]), but makes very little, if any, progress in integrating these loop bounds in the program analysis environment of WCET. Another line of research makes use of abstract interpretation based static analysis techniques to provide good WCET estimates; however, often loop bounds are assumed to be a priori given, in part, by the user, see e.g. [57, 32, 49]. Unlike these approaches, we compute loop bounds to be used in the WCET analysis of programs. Our approach relies on recent advances in symbolic computation and automated reasoning and is fully automated. That is, it requires no user guidance in providing loop bound templates or additional poperties.

The loop bounds inferred by our method tightens the WCET bounds of programs, and hence significantly improve the performance of WCET Squeezing in timing analysis. Our method can be summarized as follows. We identify special classes of loops. We over-approximate non-deterministic initializations of programs and translate multi-path loops with abrupt termination and monotonic conditional updates into simple loops. We then apply a pattern-based recurrence solving algorithm to infer precise loop bounds for simple loops with linear arithmetic updates, computing this way tight loop bounds for multi-path loops. The rest of this section describes the main steps of our method.

Loop Bound Computation of Simple Loops

We start presenting our approach to computing loop bounds by first considering a special class of simple loops. The algebraic notions and notations used in this section are as given in Chapter 2. The simple loops on which our method can be applied is described by the following class of *simple loops with linear updates and conditions* given in Figure 4.2.

```
for (i = a; i < b; i = c * i + d)
```

Figure 4.2: Simple loops with linear updates and conditions, where a, b, c, d, with $c \neq 0$, are constants from the field $\mathbb{K}$ and do not depend on i.

Note that for loops described by Figure 4.2, the loop iteration variables (i.e. i) are bounded by symbolic constants and updated by linear expressions over iteration variables. Observe that, when it comes to the WCET analysis of Figure 4.1 discussed in Section 4.1, the inner loop given between lines 6-9 of Figure 4.1 is an instance of the class of loops defined by Figure 4.2

Given a loop as in Figure 4.2, we derive a *precise* upper bound on the number of loop iterations using pattern-based recurrence solving, as follows.
(i) We model the loop iteration update as a recurrence equation over a new variable $n \in \mathbb{N}$ denoting the loop counter. To do so, we write $i(n)$ to denote the value of variable i at the nth loop iteration. The recurrence equation of i corresponding to Figure 4.2 is given below.

$$i(n+1) = c * i(n) + d \quad \text{with the initial value} \quad i(0) = a. \tag{4.1}$$

Note that (4.1) is a C-finite recurrence of order 1 as variable updates of Figure 4.2 are linear.
(ii) Next, the recurrence equation (4.1) is solved and the closed form of i as a function over n is derived. More precisely, we use *pattern-based recurrence solving* depending on the value of c and compute the closed form of $i(n)$ as given below.

$$\begin{array}{l} i(n) \;=\; \alpha * c^n + \beta, \quad \text{if } c \neq 1 \\ \text{where} \\ \begin{cases} \alpha + \beta & = \; a \\ \alpha * c + \beta & = \; a * c + d \end{cases} \end{array} \quad \text{and} \quad \begin{array}{l} i(n) \;=\; \alpha + \beta * n, \quad \text{if } c = 1 \\ \text{where} \\ \begin{cases} \alpha & = \; a \\ \alpha + \beta & = \; a + d \end{cases} \end{array} \tag{4.2}$$

(iii) The closed form of $i(n)$ is further used to derive a tight upper integer bound on the number of loop iterations of Figure 4.2. To this end, we are interested in finding the value of n such that the loop condition holds at the nth iteration and is violated at the $n+1$th iteration. We are thus left with computing the *(smallest) positive integer value* of n such that the below formula is satisfied:

$$n \in \mathbb{N} \;\wedge\; i(n) < b \;\wedge\; i(n+1) \geq b. \tag{4.3}$$

```
for (j = 1; j < 100;
     j = j * 3 + 1) ;
```

Figure 4.3: Loop with C-finite update.

```
for (i = nondet(); i > 0;
     i = i >> 1) ;
```

Figure 4.4: Loop with non-deterministic initialization.

The smallest n derived yields a tight upper bound on the number of loop iterations of Figure 4.2. This upper bound is further used in the WCET analysis of programs containing a loop matching Figure 4.2.

Example 4.1 Consider Figure 4.3, which is a simplified version of the loop between lines 6-9 of our motivating example from Figure 4.1. Updates over j describe a C-finite recurrence, whereas the loop condition is expressed as a linear inequality over j. Let $n \in \mathbb{N}$ denote the loop counter. Based on (4.2), the value of j at arbitrary loop iteration n is $j(n) = \frac{3}{2} * 3^n - \frac{1}{2}$. Using (4.3), the upper bound on loop iterations is further derived to be 4 – we consider 0 to be the starting iteration of a loop.

Let us note that, instead of adapting our pattern-based recurrence solving approach, the recurrence equation (4.1) can also be done by using more powerful symbolic computation packages, such as [52, 37]. These packages are implemented on top of computer algebra systems (CAS), for example the Mathematica system [70]. Integrating a CAS with program analysis tools is however problematic due to the complexity and closed-source nature of CAS. Moreover, the full computational power of CAS algorithms is hardly needed in applications of program analysis and verification. Therefore, for automatically inferring exact loop bounds for Figure 4.2 we designed a *pattern-based recurrence solving* algorithm which is not based on CAS. Our method relies on the crucial observation that in our approach to WCET analysis we do not handle arbitrary C-finite recurrences. We only consider loops matching the pattern of Figure 4.2, where updates describe C-finite recurrences of order 1. Closed forms of such recurrences are given in (4.2). Therefore, to compute upper bounds on the number of loop iterations of Figure (4.2) we do not deploy the general C-finite recurrence solving algorithm given in [28], but instantiate the closed form pattern (4.2). In other words, whenever we encounter a loop of the form Figure 4.2, the closed form of the iteration variable is derived by instantiating the symbolic constants $a, b, c, d, \alpha, \beta$ of (4.2) with the concrete values of the loop under study. Hence, we do not make use of general purpose C-finite recurrence solving algorithms, but handle Figure 4.2 by pattern-matching C-finite recurrences of order 1. However, our approach can be further extended to handle loops with more complex linear updates than those in Figure 4.2.

Finally let us make the observation that while reasoning about Figure 4.2, we consider the iteration variable i and the symbolic constants $a, b, c, d, \alpha, \beta$ to have values in $\mathbb{K}$. That is, when solving recurrences of Figure 4.2, the integer variables and constants are safely approximated over $\mathbb{K}$. However, when deriving upper bounds of Figure (4.2), we restrict the satisfiability problem (4.3) over integers.

```
for (i = nondet();                    for (i = nondet();
     i > d; i >>= m) ;                     i < d; i <<= m) ;

          (a)                                   (b)
```

Figure 4.5: Simple shift-loops, where $d \in \mathbb{K}$ does not depend on i and $m \in \mathbb{N}$.

Program Flow Refinement for Loop Bound Computation of Simple Loops with Non-Deterministic Behaviour

In this case we describe a small generalization of the class of simple loops presented in Figure 4.2, for which our method succeeds to automatically infer loop bounds. We call a loop a *shift-loop* if updates over the loop iteration variables are made using the bit-shift operators $\ll$ (left shift) or $\gg$ (right shift). Let us recall that the operation $i \ll m$ (respectively, $i \gg m$) shifts the value of i left (respectively, right) by m bits. Note that the outermost loop between lines 5-21 of our motivating example given in Figure 4.1 is a shift-loop.

Consider now a *simple shift-loop* with iteration variable i, where i is shifted by m bits. Hence, updates over i describe a C-finite recurrence of order 1. Upper bounds on the number of loop iterations can thus be derived as described for Figure 4.2, whenever the initial value of i is specified as a symbolic constant. However, the initial value of i might not always be given or derived by interval analysis. A possible source of such a limitation can, for example, be that the initialization of i uses non-deterministic assignments, as given in Figure 4.5(a) and (b).

For shift-loops matching Figure 4.5, our approach described in the previous section for simple loops with linear updated and conditions would thus fail in deriving loop upper bounds. To overcome this limitation, we proceed as below.
(i) We soundly approximate the non-deterministic initial assignment to i by setting the initial value of i to be the integer value that allows the maximum number of shift operations within the loop (i.e. the maximum number of loop iterations). To this end, we distinguish between left and right shifts as follows.

- If i is updated using a right-shift operation (i.e. Figure 4.5(a)), the initial value of i is set to be the maximal integer value, yielding a maximum number of shifts within the loop. The initial value of i is hence assumed to have the value 2147483647, that is $010\ldots0$ in a 32-bit binary representation (the most significant, non-sign, bit of i is set).
- If i is updated using a left-shift operation (i.e. Figure 4.5(b)), we assume the initial value of i to be the integer value resulting in the maximum number of shift operations possible: i is set to have the value 1, that is $0\ldots01$ in a 32-bit binary representation (the least significant bit of i is set).

(ii) The upper bound on the number of loop iterations is then obtained by first computing the difference between the positions of the highest bits set in the initial value of i and

d, and then dividing this difference by m. If no value information is available for m, we assume m to be 1.

Example 4.2 Consider the shift-loop of Figure 4.4 with right shift updates. The initial value of i is set to `INT_MAX`. The upper bound on the number of loop iteration is then derived to be 31.

Let us note that the treatment of non-deterministic initial assignments described above is not only restricted to simple shift-loops, but can be also extended to the more general class of Figure 4.2. For doing so, one would however need to investigate the monotonic behavior of C-finite recurrences in order to soundly approximate the initial value of loop iteration variables.

Program Flow Refinement for Loop Bound Computation of Multi-Path Loops

We now describe our method in the presence of multi-path loops. As paths of multi-path loops can interleave in a non-trivial manner, deriving tight loop upper bounds, and thus accurate WCET bound, for programs containing multi-path loops is a challenging task. In our approach to WCET analysis, we identified special classes of *multi-path loops with only conditionals* which can be translated into simple loops by refining the control flow of the multi-path loops. Loop bounds for the obtained simple loops are then further derived as discussed in the previous sections, yielding thus loop bounds of the original multi-path loops. In what follows, we detail the class of multi-path loops our approach can automatically handle and overview the flow analysis techniques deployed in our work. For simplicity, in the rest of this chapter we only multi-path loops with only 2 paths, arising from conditionals.

Multi-path loops with abrupt termination. One class of multi-path loops that can automatically be analyzed by our framework is the class of linearly iterating loops with abrupt termination arising from non-deterministic conditionals, as given in Figure 4.6 (a)–(c).

```
int a;
for ( ; i < b;
     i = c * i + d)
  if (nondet())
    break;
```

(a)

```
int i = nondet();
for ( ; i > e;
     i = i >> m)
  if (nondet())
    break;
```

(b)

```
int i = nondet();
for ( ; i < e;
     i = i << m)
  if (nondet())
    break;
```

(c)

Figure 4.6: Multi-path loops with abrupt termination, where $a, b, c, d, e \in \mathbb{K}$, with $c \neq 0$, do not depend on i and $m \in \mathbb{N}$.

Note that we are interested in computing the *worst-case* execution time of a loop from Figure 4.6. Therefore, we safely over-approximate the number of loop iterations of

```
for (j = 0; j < 100; j++)
  if (nondet())
    break;
```

Figure 4.7: Loop with abrupt termination.

```
for (j = 0; j < 100; j++)
  if (nondet())
    j++;
```

Figure 4.8: Loop with monotonic conditional update.

```
for (i = a; i < b; i = c * i + d)
   if (B) i = f1(i);
   else i = f2(i);
```

Figure 4.9: Multi-path loops with monotonic conditional updates.

Figure 4.6 by assuming that the abruptly terminating loop path is not taken. In other words, the non-deterministic conditional statement causing the abrupt termination of Figure 4.6 is ignored, and we are left with a simple loop as in Figure (4.2) or Figure 4.5. Upper bounds on the resulting loops are then computed as previously described, from which an over-approximation of the WCET bound of Figure 4.6 is derived.

Example 4.3 Consider Figure 4.7. Assuming that the abruptly terminating loop path is not taken, we obtain a simple loop as in Figure 4.2. We thus get the loop bound 100.

Multi-path loops with monotonic conditional updates. By analyzing the effect of conditional statements on the number of loop iterations, we identified the class of multi-path loops as given in Figure 4.9, with the following constraints. The symbolic constants a, b, c, d, with $c > 0$, are elements of $\mathbb{K}$ and do not depend on i, whereas B is a boolean condition over loop variables. Further, we impose that $f_1, f_2 : \mathbb{K} \to \mathbb{K}$ are monotonically increasing function such that:

$$\begin{cases} i < c * i + d \text{ and} i \leq f_1(i) \text{ and } i \leq f_2(i) & \text{if } i \geq a \\ i > c * i + d \text{ and} i \geq f_1(i) \text{ and } i \geq f_2(i) & \text{if } i \leq a. \end{cases}$$

We refer to the assignments $i = f_1(i)$ and $i = f_2(i)$ as *conditional monotonic assignments (or updates)*, as their execution depends on the truth value of B.

We refer to the assignments $i = f_1(i)$ and $i = f_2(i)$ of Figure 4.9 as *conditional monotonic assignments (or update)*, as their execution depends on the truth value of B. Let $g : \mathbb{K} \to \mathbb{K}$ denote the function $i \mapsto c * i + d$ describing the linear updates over i made at *every* iteration of Figure 4.9. Note that the monotonic behavior of g depends on c and coincides with the monotonic properties of f_1 and f_2.

To infer loop bounds for Figure 4.9, we aim at computing the *worst-case* iteration time of Figure 4.9. To do so, we ignore B and transform Figure 4.9 into a simple loop by *safely over-approximating* the multi-path behavior of Figure 4.9, as given below. In what follows, let $\Delta = |g(i+1) - g(i)|$, $\Delta_1 = |f_1(i+1) - f_1(i)|$, and $\Delta_2 = |f_2(i+1) - f_2(i)|$, where $|x|$ denotes the absolute value of $x \in \mathbb{K}$.

(i) If c is positive, let $m = \mathtt{min}\{\Delta + \Delta_1,\ \Delta + \Delta_2\}$. That is, m captures the minimal value by which i can be increased during an arbitrary iteration of Figure 4.9.

Alternatively, if c is negative, we take $m = \mathtt{max}\{\Delta + \Delta_1,\ \Delta + \Delta_2\}$. That is, m captures the maximal value by which i can be decreased during an arbitrary iteration of Figure 4.9.

(ii) The multi-path loop of Figure 4.9 is then over-approximated by the simple loop (4.4) capturing the worst-case iteration time of (4.9).

$$\begin{cases} \texttt{for (i =a; i<b; \{i=c*i+d;i=}f_1\texttt{(i)\})}, & \text{if } c > 0 \text{ and } m = \Delta + \Delta_1 \\ \texttt{for (i =a; i<b; \{i=c*i+d;i=}f_2\texttt{(i)\})}, & \text{if } c > 0 \text{ and } m = \Delta + \Delta_2 \\ \texttt{for (i =a; i<b; \{i=c*i+d;i=}f_1\texttt{(i)\})}, & \text{if } c < 0 \text{ and } m = \Delta + \Delta_1 \\ \texttt{for (i =a; i<b; \{i=c*i+d;i=}f_2\texttt{(i)\})}, & \text{if } c < 0 \text{ and } m = \Delta + \Delta_2 \end{cases} \tag{4.4}$$

Hence, the control flow refinement of Figure 4.9 requires checking arithmetic constraints over f_1 and f_2. We automatically decide this requirement using arithmetic SMT queries.

(iii) We are finally left with computing loop upper bounds of (4.4). To this end, we need to make additional constraints on the monotonic functions f_1 and f_2 of (4.9) so that the approach used for Figure 4.2 can be applied. Namely, we restrict f_1 (respectively, f_2) to be a linear monotonic function $i \mapsto u * i + v$, where $u, v \in \mathbb{K}$ do not depend on i and $u > 0$. As linear monotonic functions are closed under composition, updates over the iteration variable i in (4.4) correspond to C-finite recurrences of order 1. Upper bounds on loop iterations of (4.4) can thus be derived as presented for the simple-loop of Figure 4.2.

Note that the additional constraints imposed over f_1 and f_2 restrict our approach to the following multi-path loops with linear conditional updates.

```
for (i=a; i<b; i=c*i+d)
{
  if (B) i=u1*i+v1;                                    (4.5)
  else  i=u2*i+v2;
}
```

where $a, b, c, d, u_1, u_2, v_1, v_2 \in \mathbb{K}$ do not depend on i, and $c, u_1, u_2 \neq 0$,
B is a boolean condition over loop variables, and

$$\begin{cases} u_1 > 0 \text{ and } u_2 > 0, & \text{if } c > 0 \\ u_1 < 0 \text{ and } u_2 < 0, & \text{if } c < 0 \end{cases}$$

```
for (fcode = (long) hsize;
     fcode < 65536L; fcode *= 2L)
    hshift++;
```

Figure 4.10: Loop using multiplication in the update expression.

```
while (tries_left > 0) {
  ...
  tries_left--;
  if (confirm_hit_result == 0)
    tries_left = 0;
}
```

Figure 4.11: `health.c` program from the Debie benchmark suite, loop with a conditional update to the loop counter.

Loops (4.5) form a special case of Figure 4.9. Let us however note, that extending the simple loop approach of Figure 4.2 to handle general (or not) C-finite recurrences with monotonic behavior would enable our framework to compute upper bounds for arbitrary loops in Figure 4.9.

Example 4.4 Consider Figure 4.8. The conditional update over j is linear, and hence the multi-path loop is transformed into the simple loop `for (j = 0; j < 100; j++)`. The loop bound of Figure 4.8 is therefore derived to be 100.

Challenging Examples and Benchmarks

In this section we address further extension of our method for loop bound computation. For doing so, we give examples of some loops taken from various benchmarks, which illustrate the main ingredients and advantage of our symbolic loop bound computation technique.

Figure 4.10 lists a loop with a C-finite update. It can be characterized by the following linear recurrence relation:

$$\mathit{fcode}(n+1) = 2 * \mathit{fcode}(n), \quad \text{where } n \geq 0 \text{ denotes the loop iteration counter.}$$

Hence, the value of $fcode$ at an arbitrary iteration n can be derived by instantiating the closed form solution of Figure 4.2. As a result, the value of fcode is expressed as a function of n. Note that the loop condition $\mathit{fcode} < 65536L$ holds at any loop iteration, and therefore $\mathit{fcode}(n) < 65536L$ is a valid formula. A precise loop bound for Figure 4.10 is then derived by computing the smallest value of n such that the loop terminates. In other words, the (smallest) value of n is inferred such that the formula $(\mathit{fcode}(n) < 65536L) \wedge (\mathit{fcode}(n+1) \geq 65536L)$ is satisfiable. Let us note that the closed form representation of $i(n)$ in equation 4.3 involves, in general, exponential sequences in n. Therefore, to compute the value of n such that 4.3 holds, we make use of the logarithm, floor and ceiling built-in functions of Prolog.

Consider the loop from Figure 4.11. This loop fits the loop pattern given in Figure 4.6(b). Deriving a tight loop bound for Figure 4.11 requires reasoning about the conditional update to *tries_left*. To this end, the loop iterates over *tries_left* either by monotonically decreasing the value of *tries_left*, or by setting the value of *tries_left* to 0. As in both cases the value of *tries_left* decreases[1], the conditional statement of Figure 4.11 is omitted and Figure 4.11 is approximated by a simple loop of Figure 4.2. Further, a precise loop bound for the simple loop is computed, yielding thus a tight loop bound for Figure 4.11.

Figure 4.12 lists an abruptly terminating loop. The loop fits the loop pattern given in Figure 4.6(a). The difficulty in deriving a tight loop bound for Figure 4.12 comes from the presence of the conditional statement yielding an abrupt termination of the loop. To this end, the control flow of Figure 4.12 is refined by abstracting away the *break* statements. This way, Figure. 4.12 is approximated by a simple loop with C-finite updates. A precise loop bound of the simple bound is next derived, from which a tight loop bound for Figure 4.12 is obtained.

```
for (i = 0; i < M; i++) {
  ...
  if (A[i] == 0) {
    failed = 1;
    break;
  }
}
```

Figure 4.12: Loop with abrupt termination, from `array.c` in the Scimark benchmark suite.

```
while (i > 0) {
  int p = (i - 1) / 2;
  if (nondet())
    break;
  i = p;
}
```

Figure 4.13: Abruptly terminating loop with C-finite update.

```
while (i < size) {
  int j = 2 * i + 1;
  if (nondet())
    j++;
  if (nondet())
    break;
  i = j;
}
```

Figure 4.14: Abruptly terminating loop with C-finite and conditional updates.

The examples given so far described loops matching exactly one of the loop patterns from Figures 4.2, 4.6(a), and 4.6(b). Our experiments show however that these loop patterns are used in their combination (see Chapter 6). To this end, let us consider Figure 4.13 and Figure 4.14.

The loop from Figure 4.13 requires reasoning about abrupt termination and C-finite updates. A tight loop bound for Figure 4.13 is inferred by first applying flow refinement to transform Figure 4.13 into a simple loop with C-finite update, and then derive a precise loop bound for the obtained simple loop.

The program from Figure 4.14 implements an abruptly terminating loop with C-finite and conditional linear updates. After flow refinement and establishing the monotonicity property of the conditional update, Figure 4.14 is rewritten into a simple loop with only C-finite updates. A precise loop bound of the simple loop is next computed by applying our pattern-based recurrence solving approach, and a loop bound of Figure 4.14 is finally inferred.

Limitations. We investigated examples on which our approach fails to derive loop bounds. We list some of the failing loops in Figure 4.15. Note that the arithmetic used

[1]note that the loop condition assumes that *tries_left* is a positive non-zero symbolic scalar

```
while (abs(diff) >= 1.0e-05) {
  diff = diff * -rad * rad /
   (2.0 * inc) *
   (2.0 * inc + 1.0);
  inc++;
}
```
(a)

```
while (k < j) {
  j -= k;
  k /= 2;
}
```
(b)

```
while (((int)p[i]) {
  i++;
```
(c)

Figure 4.15: Limitations of r-TuBound.

in the simple loop Figure 4.15(a) requires extending our framework with more complex recurrence solving techniques, and deploy SMT solving over various numeric functions, such as the absolute value computations over floats or integers. The complex arithmetic update cannot be captured by the C-finite updates of Figure 4.2 and would require solving recurrences with non-constant polynomial coefficients in the loop variables. Algorithmic methods are available for solving such recurrences – see e.g. [61, 37]. On the other hand, Figure 4.15(b) suggests a simple extension of our method to solving blocks of C-finite recurrences, as the evaluation of the loop condition depends on a loop variable that is modified in each iteration of the loop. Finally we note that Figure 4.15(c) illustrates the need of combining our approach with reasoning about array contents, as the loop condition depends on array values. To this end, we plan to extend our method with support for arrays by using SMT or first-order reasoning in the array theory.

Further, extending our method does not yet support multi-path nested loops. However, we believe that using our current flow refinement approaches in a combined manner would yield interesting results for the study of nested loops. We also plan to investigate our approach in conjunction with the technique of [12], where a large number of nested loops are handled using symbolic computation techniques over loop indexes.

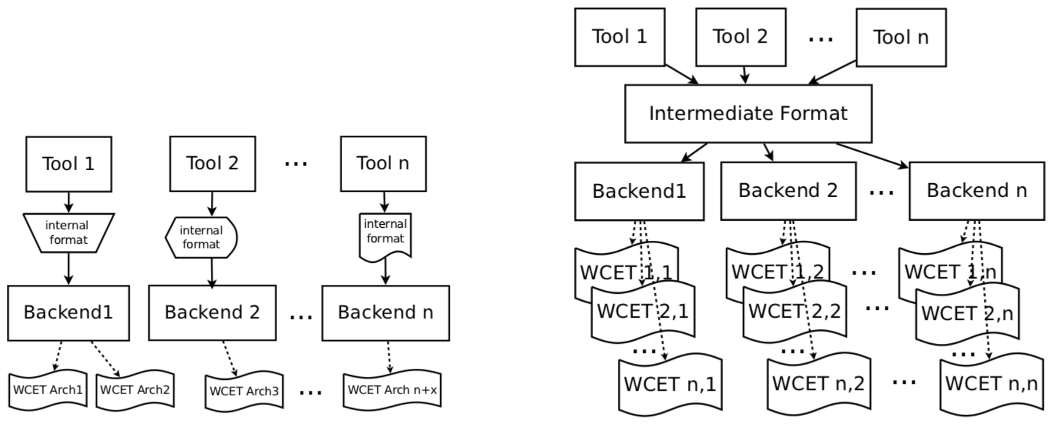
CHAPTER 5

Portability – Distributing WCET Squeezing by Enabling Interoperability of WCET Analyzers

Static WCET analysis relies on various flow fact information about the program under analysis, e.g. loop bounds. Such information may be given manually by the developer or inferred automatically by a flow fact analyzer. Well-known WCET tools, such as aiT [2], Bound-T [67] or SWEET [32], use so-called annotation languages in order to carry gathered information. In [41] the authors define and summarise ingredients of an annotation language in order to classify the information needed by the WCET analysis performed by various WCET tools. Following this line, the work of [42] emphasizes the need of a common annotation language for comparing various WCET tools.

In this chapter we address this problem and propose a *flexible and extensible intermediate language* that provides a common interface for exchaning flow facts between WCET analyzers. For doing so, we adapt a portable annotation language, called Flow Facts in XML, shortly referred to as *FFX*. Our work relies on the FFX format proposed in [21] as the internal format of the OTAWA WCET analyzer, and extends FFX with additional features needed for making it available for other state-of-the-art WCET analyzers. In particular, we address the interopality between the OTAWA [5] and r-TuBound [44] WCET toolchain by using FFX as an intermediate annotation format.

To understand the advantage of a common intermediate annotation languages, let us consider Figure 5.1. Figure 5.1(a) illustrates the difference between The traditional workflow of WCET analyzers relying on an internal format to exchange information between their front- and back-end is illustrated in Figure 5.1(a). On the other hand, the workflow of WCET analyzers proposed by our work using a portable annotation



(a) **(b)**

Figure 5.1: (a) shows the traditional workflow of WCET analyzers using different internal formats. (b) shows the abstract setup of the FFX experiment for an arbitrary number of WCET analysis toolchains. It depicts toolchain 1 translating its internal format to the intermediate format, toolchain 2 supporting it natively and toolchain n dropping its internal format in favor of the intermediate format.

format is presented in Figure 5.1(b). As WCET analyzers might support different architectures, Figure 5.1 suggests that a portable annotation language allows one to extend the usage of various WCET tools to different platforms and architectures. This way, a fair comparison between WCET tools can be made and results and different methods between WCET analysis toolchains can be shared. A benefit of such a sharing can, for example, be identified for WCET Squeezing, by making WCET Squeezing available for various high-level WCET analyzers via FFX. As WCET Squeezing is implemented in our r-TuBound WCET toolchain supporting now also the FFX format, any other WCET analysis toolchain understanding FFX can supply its high-level results to the r-TuBound back-end – see Figure 5.2. Hence, the results of high-level analyzers supporting FFX can be used to infer further program flow facts about, whose precision can be then proved by WCET Squeezing on any WCET back-end implementing the FFX format. Moreover, by representing results of different WCET analyzers in the FFX format, the quality of initial WCET bounds used in WCET Squeezing can be improved, turning WCET Squeezing into a more efficient proof procedure. The situation illustrated in Figure 5.2 can be extended to any other WCET technique which can be applied after a high-level WCET analysis. Even more, sharing the entire WCET analysis back-end to apply results of WCET analyzers on other platforms can also materialize using a common annotation language.

Our intermediate FFX annotation language hence supports interoperability of WCET analyzers, by providing a common interface for exchanging analysis information. The main advantages of FFX, together with our contributions, are summarized below.

- We propose the FFX format as an open portable and expandable annotation for-

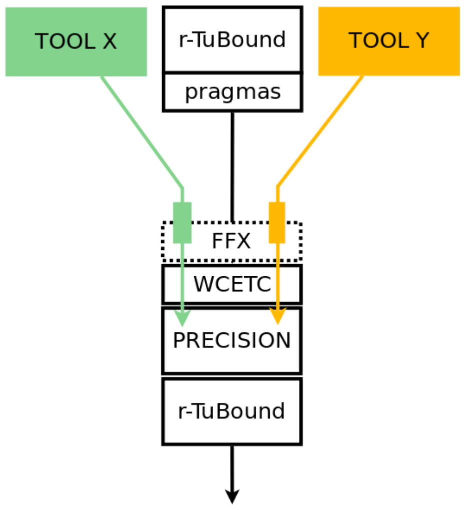


Figure 5.2: Portability of the WCET Squeezing proof procedure by a common (FFX) interface.

mat, and use FFX as an intermediate format for WCET analysis. Such an annotation format is needed for a fair comparison of various WCET results. For instance, during the WCET tool challenge [69] almost all participants had problems annotating the benchmarks with the supplied flow constraints: the restrictions, often formulated in a natural language, can be hard to be correctly understoof and it is even harder to annotate them in a given flow fact language. We believe that with FFX at hand, it is easier to tackle the WCET tool challenge and possibly open the opportunity for other tools to participate.

- The FFX format allows combining flow fact information from different high-level tools. WCET analysis is usually two-fold: the timing analysis part is architecture-dependent, whereas the flow fact analysis is not. For this reason one cannot compare results from tools that do not support a common architecture, e.g. ARM or PowerPC, in the WCET tool challenge. By using the FFX format, we believe tools have the opportunity to extend their flow fact analysis to other architectures that are not supported by their original toolchain.

- FFX decreases the implementation effort when integrating different WCET analysis toolchains. One could argue that integration of two WCET analysis toolchains by supporting the native format of the other toolchain has the same implementation effort. Nevertheless, when using the intermediate format FFX, the effort decreases with a higher number of toolchains involved. Consider Figure 5.1 again. Adding another tool, that is Tool n+1, to Figure 5.1(a) requires to implement n translations to the native formats of the other tools. Using Tool n+1 in Figure 5.1(b), with FFX as intermediate format, reduces the implementation effort to *exactly* 2 translations. Moreover, the FFX intermediate format comes with the advantage that a change in the internal format of a specific tool does not require all the other tools to update their translation. Because of this decoupling, it is only necessary to update the translation from FFX to the modified internal format for one tool [25].

- Introducing a common annotation language is important in order to tighten and compare WCET results. FFX allows to perform the timing analysis by using an annotation file from arbitrary source, for example provided by users or inferred by a flow fact analyzer. To improve the WCET accuracy, a tool could use annotations provided by another flow fact analyzer. It is possible to merge the results of several flow fact analyzers in order to obtain the most accurate information available. This approach is already used partially in the OTAWA tool in order to collect and merge analysis information from several sub-analyzers.

- FFX can be used for quality assurance to test and validate new analysis techniques and tools against. This is achieved by using FFX as a flow fact storage or knowledge base about benchmarks. Therefore, we believe that having the same FFX annotation language also helps real-time systems developers. They will be able to adapt their choice of WCET tool to their targeted architecture or type of programs without learning a new format or tool.

- We introduce an FFX support for the r-TuBound tool and perform experiments with FFX using both r-TuBound [44] and oRange/Otawa [49, 5] . To this end, we instantiate Figure 5.1(b) with the r-TuBound tool and the oRange tool as high-level tools, and use FFX as intermediate format. OTAWA [5] and CalcWCET167 [40] serve as WCET back-ends for the toolchains. We thus get two sets of comparable FFX flow facts and two WCET estimates for each of the architectures supported by the two tool chains. Additionally, it is possible to combine the flow facts gathered by the tools to get a more accurate WCET estimate. Our experiments emphasize the practical and theoretical benefits of using the FFX common format for exchanging flow facts and back-ends in the WCET analysis of systems (see Section 6.3 for experimental details).

In the rest of this section we first overview the basic constructs and principles of the FFX annotation language. We then present our approach and extensions to FFX, making it feasible to various WCET analyzers, in particular to the OTAWA and r-TuBound toolchains.

5.1 The FFX Annotation Language

In this section we present the most relevant FFX elements, for more details about the format we refere to [21]. We use EBNF notation to present the format and focus on high-level flow fact elements, omitting low-level elements (for example, FFX elements used to describe cache or other hardware configurations).

At the current state of development, FFX permits to address interesting program parts and to express information to resolve complex control structures, e.g. indirect branches due to switch statements that are compiled to function tables or function pointer calls. An FFX file may include only the code parts for which flow facts have been derived.

Location Attributes

Location attributes (Figure 5.3) allow to identify code either in the binary or in the source representation of the program. This is done using different sets of attributes depending on the constructs used.

```
LOCATION-ATTRS ::=
| IDENTIFICATION
| ADDRESS-LOCATION
| LABEL-LOCATION
| SOURCE-LOCATION
```

Figure 5.3: FFX Location Attributes

`IDENTIFICATION`: Instead of identifying concrete locations in the program, it allows to make references to parts in the code represented by the other location attributes. A typical usage is to annotate the execution count of a piece of code inside a control constraint.

`ADDRESS-LOCATION`: Location by an address is the simplest scheme. A target location is represented as an address stored in an attribute. The drawback is that the location may be invalidated each time the application is compiled.

`LABEL-LOCATION`: The label location provides more flexibility. It encodes locations either by a label, or by a label and an offset. This scheme does not support recompilation but relinking of libraries, as the code is not modified. It only requires the translation of the label address from one position to another.

`SOURCE-LOCATION`: Source locations support recompilation and relinking but not source modifications. It defines the location as a source file name together with a line number in the file. To be applicable, it requires either a source representation or a binary representation embedding debugging information. It is preferable to have only one statement per line, otherwise ambiguity can emerge between locations in certain circumstances. It is necessary to obtain an XML tree corresponding to the call graph, which is guarantied if there is only one statement per line.

Context Elements

A context element (Figure 5.4) defines a condition on the information it contains. Information embedded in unsatisfied contexts is not considered, i.e. only information contained in valid contexts is used in further analysis.

```
  CONTEXT ::=
    <context name="TEXT">
      TOP-LEVEL-ITEM*
    </context>
```

Figure 5.4: FFX Context Element

```
<context name="arm">
  <function name="f">
    <loop maxcount="10"/>
  </function>
</context>

<context name="arm">
  <context name="task_1">
    <function name="f">
      <loop maxcount="5"/>
    </function>
  </context>
</context>
```

Figure 5.5: FFX Context Example

Consider, for example, Figure 5.5: if the only valid context is `arm`, the loop bound will be 10. If both context `arm` and `task_1` are valid, the bound 5 will be used for further analysis.

Note that context names may be classified and prefixed using the following constructions below: `hard:TEXT` for hardware contexts, `task:TEXT` for task contexts and `scen:TEXT` for scenario contexts.

A hardware context represents different behavior of an application, depending on the underlying hardware. For example, the number of possible iterations of a loop counting the number of one-bits in a word depends on the size of the word. It will be 32 on 32-bit machines and 64 on 64-bit machines.

Functions composing an application may be called from different tasks that make up the real-time system. Depending on the task calling the functions, some flow properties may have different values. One could just take the worst-case behavior to characterize the functions, independent of the context. Nevertheless, this would come at the price of a loss of precision and an overestimation of the WCET.

Finally, flow information may depend on a chosen scenario. For example, an execution configuration chosen by the user, or a system that exhibits state variables controlling the running mode of the application. These might bring the application to a “running state”, “failure state” or “critical state”. It may be interesting to examine the different properties of a task according to the running mode, as the scheduling decisions can also change accordingly. In addition, the properties defined in a scenario may also be used to force the behavior of the task, for example, by fixing the value of the state variables.

Control Flow Elements

A function element (Figure 5.6) represents the static code location for the given function. It can contain statements and thus allows to identify and access dynamic locations inside the function.

```
FUNCTION ::=
  <function LOCATION-ATTRS  INFORMATION-ATTRS>
    STATEMENT*
  </function>
```

Figure 5.6: FFX Functions

The LOCATION-ATTRS identify the static location of the function in the code. INFORMATION-ATTRS represent generic hooks where any flow information can be attached to (frequency, execution time, etc.).

Different statements are supported inside functions (Figure 5.7) and represent the flow structure of the code. They can be composed to express dynamic locations that depend on a specific context.

```
STATEMENT ::= BLOCK | CALL | CONDITION | LOOP
```

Figure 5.7: FFX Statements

`BLOCK`: A block element identifies a piece of code, possibly composed of several execution paths, but with a single entry point only.

`CALL`: A call element identifies the call to a function. Its location represents the caller and it must contain a function element representing the callee. Multiple call elements that embed functions allow to represent call-chain locations.

`CONDITION`: A condition element represents a condition with several alternatives. In the C language, it applies to both `if` statements and `switch` statements.

`LOOP` (Figure 5.8): A loop element matches a loop construct in the code. It may contain *iteration* elements in order to represent properties that are valid only during certain iterations of the loop.

The iteration `number` i can be positive, identifying the ith iteration, or negative, identifying the ith iteration counted from the last iteration. A Loop bound attribute is an INFORMATION-ATTRIBUTE but is limited to loop elements (Figure 5.9).

The attribute `executed` is set to `false` if the element is never executed. `maxcount` and `totalcount`, attributes of `loop` elements, denote the maximum number of loop

```
LOOP ::=
  <loop LOCATION-ATTRS INFORMATION-ATTRS>
    STATEMENT*
  </loop>
| <loop LOCATION-ATTRS INFORMATION-ATTRS>
    <iteration number="INT">
      STATEMENT*
    </iteration>
  </loop>
```

Figure 5.8: FFX Loops

```
LOOP-ATTR ::=
| maxcount="INT|NOCOMP"?
| mincount="INT|NOCOMP"?
| totalcount="INT|NOCOMP"?
| exact="BOOL"?
| executed="BOOL"?
| expmaxcount="TEXT"?
| exptotalcount="TEXT"?
```

Figure 5.9: FFX Loop Attributes

iterations for each loop entry and the maximum number of loop iterations in relation to the outer loop scope. Those attributes can either be integer values, `NOCOMP` (not computable) or parameterized expressions. The `loop` attribute `exact` is set to true if the `totalcount` attribute is exact, i.e., no overestimation. `expmaxcount`, respectively `exptotalcount`, is a formula computing the value of `maxcount`, respectively `totalcount`, if no concrete value can be inferred for `maxcount`, respectively `totalcount`. These attributes represent the iteration count as a formula, using a syntax similar to that of C arithmetic expressions.

The power of FFX comes with the fact that it is expandable. One could add custom tags in order to make a certain tool more efficient. The custom tags will simply be ignored by tools that do not support them. As an example, consider a WCET back-end that relies on the *Implicit Path Enumeration Technique* (IPET) that usually only handles linear constraints, whereas FFX could introduce arbitrary constraints. Nevertheless, the back-end can just ignore these non-linear constraints and find a solution for the remaining constraints. A similar situation is the `iteration` construct: it is unused, and hence not written, by both (static analyzers) oRange [49] and r-TuBound [44], but emitted by the measurement based analyzer rapiTa [60], where it is much more valuable to inspect single iterations of loops.

5.2 Interoperability via FFX

The FFX format has been introduced for and with the oRange/Otawa WCET analysis tool chain [21]. FFX is an XML-based file format that is used to represent flow facts. Such information is either used to help to achieve the computation, or to exhibit WCET in particular situations. The main concepts and ideas that drive the design and the development of FFX include – *freedom*: tools using FFX do not need to support all features. In the worst-case, unsupported features induce a loss of precision but never invalidate information; *expandability*: the use of XML allows to easily insert new elements or new attributes without breaking the compatibility with other tools; *soundness*: all provided information must be ruled by a precision order, ensuring that, at least, information in most generic cases must not be more precise than in particular cases.

One of the goals of FFX is to have an annotation language that can be extended by constructs that yield improvements in the WCET computation, e.g. flow facts (loop bounds, exclusive paths, indirect function calls, control flow constraints, etc.), platform configuration (I/O, caches, etc.), target processor, tasks or entry points, configurations of analyzers, data domain information and more.

Another motivation for FFX is to have the capability to merge the result files of different tools, allowing the tools to interchange information, e.g. between the cache analyzer and the flow fact analyzer. The choice to carry all that information in a single file is made in order to take into account the enormous number of paths and context sensitive information in a program. Carrying all this information directly in the source would clutter the source under analysis. Obviously this comes with a drawback, storing the information external to the source file could lead to divergence between code and information. This however, is a matter of bookkeeping, as one can, e.g. use version strings or source hashes to avoid divergence. FFX offers means to store that information, for example, in constructs comparable to the already mentioned platform configuration part. Additionally it eases communication between tools that apply the analysis results on different representations, be it on source or binary level.

The XML nature of the format allows to collect information provided by different tools[1]. XML is standardized, easy to read and write for the developer as XML is a textual format and it offers great tool and library support. Its tree structure is particularly interesting when representing control flow graphs, as it can implicitly represent a call graph.

In what follows, we describe our contribution in making FFX as a common annotation language used in the WCET analyzers oRange/Otawa and r-TuBound. We also illustrate the use of FFX on concrete examples.

FFX in oRange/Otawa

oRange [49] is the flow fact analyzer for OTAWA. It performs a static analysis based on flow analysis and abstract interpretation of C programs. Currently oRange does not use

[1]XInclude `http://www.w3.org/TR/xinclude/`

FFX as input but produces FFX files as output. It generates flow fact information for functions, calls, conditions and loops starting from a supplied entry point.

Figure 5.10 shows an excerpt of a typical FFX file generated by oRange for the analysis of the **bs.c** benchmark from the Mälardalen suite [34]. As mentioned, the structure of the FFX file represents the structure of the C program, as it is comparable to a call graph representation of the program.

```
<loop loopId="1" line="67" source="bs.c" exact="false" maxcount="5"
      totalcount="5"
      maxexpr="floor((log(14-0)-log(ceil(0+epsilon)))/log(1/0.5)+1)+1"
      totalexpr="floor((log(14-0)-log(ceil(0+epsilon)))/log(1/0.5)+1)+1">
  <conditional>
    <condition varcond="IF-1" line="85" source="bs.c"
               isexecuted="true" expcond=""
               expcondinit="(data[mid]).key==x"></condition>
      <case cond="1"  executed="true"></case>
      <case cond="0" executed="true">
        <conditional>
          <condition varcond="IF-2" line="79" source="bs.c"
                     isexecuted="true" expcond=""
                     expcondinit="(data[mid]).key>x">
          </condition>
          <case cond="1"  executed="true"> </case>
          <case cond="0" executed="true"> </case>
        </conditional>
      </case>
        </conditional>
</loop>
```

Figure 5.10: FFX flow facts output by oRange

Some elements defined in FFX are currently not supported by oRange, others that are supported by oRange are not supported by OTAWA. Due to the composable nature of FFX, OTAWA combines analysis results of different sub-tools prior to WCET computation.

FFX in CalcWcet167/r-TuBound

r-TuBound extends the WCET analysis tool TuBound [57, 56] by combining symbolic computation techniques with the timing analysis of programs (see Section 6.1 for implementation details). Inputs are arbitrary C/C++ programs. The r-TuBound toolchain consists of a high-level source analyzer, a WCET-aware compiler and a low-level WCET analyzer. The low-level WCET analyzer relies on flow facts inferred by the high-level source analyzer. The high-level source analyzer implements interval analysis, points-to

```
for (i = 1;
     i < 100;
     i = i*2+1)
  ...
```
(a)

```
for (i = 1; i < 100;
     i = i * 2 + 1) {
#pragma wcet_marker location1
#pragma wcet_loopbound (4..4)
  ...
}
```
(b)

```
for (i = 1; i < 100;
     i = i * 2 + 1)
maximum 4 iterations {
marker location1
...
}
```
(c)

Figure 5.11: C program analysed for WCET.

analysis and loop bound computation using abstract interpretation, symbolic computation and a model checking extension. The flow facts derived during the high-level source analyses are further used as source code annotations in the low-level WCET analysis, in the form of `#pragma` declarations. The WCET analysis of the system is performed for the Infineon C167 microprocessor. The WCET-aware compiler and the low-level WCET analyzer of r-TuBound are using the WCETC language [39]. WCETC is similar to the ANSI C, extending it by constructs for WCET path annotations. These annotations in WCETC allow one to specify loop bounds and to use markers to express restrictions in the runtime behavior of the program, e.g., relational constraints on the execution frequencies of program blocks. This is illustrated in Figure 5.11. The loop bound computation step derives the loop bound of the program fragment in Figure 5.11(a) to be 4. Figure 5.11(b) shows the loop bound as a `pragma` annotation in the source. The annotated source code is then transformed to the WCETC program given in Figure 5.11(c). This WCETC code is finally compiled and statically analysed using CalcWCET167.

As mentioned for OTAWA/oRange, also r-TuBound/CalcWCET167 does not support all constructs offered by FFX. Nevertheless, this is not an obstacle, as unsupported constructs can be ignored by tools in further analysis. An example are loop attributes: FFX allows to annotate the `totalcount` of loop iterations, i.e., the maximal number of iterations of the loop during the whole execution of the program, as well as the maximum number of iterations each time the loop is entered. r-TuBound only supports the latter, but can just ignore the other attribute.

Immediate Benefits of FFX

Implementing FFX support for r-TuBound allows us to use OTAWA as WCET back-end for r-TuBound and to use CalcWCET167 as WCET back-end for oRange. This way, both tools are able to analyze code for a new architecture. Further, it is possible to compare flow facts in a common format and to compare WCET estimates on different platforms in a unified way.

In particular, for r-TuBound we overcome this way a major hurdle when participating in the WCET tool challenges. This is due to the WCET back-end used in the r-TuBound toolchain that does not support the ARM and the PowerPC platform used in the challenge, see [69].

While updating the WCET back-end to support a new platform usually requires heavy engineering effort, the approach we followed here, i.e., introducing in r-TuBound the WCET annotation language FFX, turned out to be very light-weight.

In fact, extending r-TuBound with an FFX support required only modest effort, as r-TuBound outputs annotated source code after each analysis step. We extract from the annotated source code all necessary analysis results and store the results in FFX format to an FFX file. The tree-like nature of FFX supports this approach, allowing for the construction of the XML representation as yet another layer in the cascade of high-level analyses, executed before the low-level WCET analysis. No changes to either the high-level analyzer nor the low-level back-end are necessary. This advantage extends to other toolchains when using FFX as an intermediate format: developers of WCET tools only need to implement a translation from their internal format to FFX, without changing the internal format of their tools. Another benefit of exporting to FFX instead of directly translating to a specific tool format emerges when additional tools participate in such a tool-cooperation: Using FFX reduces the implementation effort to two transformers, one for the high level to translate flow facts to FFX and one for the low level to translate from FFX to the back-ends native format, no matter how many tools participate. Nevertheless, one gains, for each high-level tool, support for all platforms supported by any of the back-ends. Using the direct translation would require to write a transformer for each of the back-ends. Additionally, such a decoupling from the tools native formats results in robustness towards changes in native formats, as only one FFX translator needs to be adapted.

Flow facts annotated as `pragma`s are translated to the following list of FFX elements: `flowfacts`, `functions`, `loops`, `calls` and `conditionals`.

Further, the following FFX attributes are required for successful WCET analysis: the `line` attribute which is used to specify the source location of constructs. r-TuBound uses `#pragma wcet_marker` that need to be translated to line numbers. The most important flow information that r-TuBound infers are loop bounds. The `maxcount` attribute of the `loop` tags is used to encode those. The `totalcount` and the `exact` attribute are currently unsupported by r-TuBound. The elements and attributes are extracted from the annotated source that r-TuBound emits after each analysis step.

The following constructs are not used in the `pragma` translations; they could, however, be used to refine the flow information and thus tighten the WCET estimate: the `executed` attribute allows to constrain the execution of paths and/or calls. It could be used to encode `#pragma wcet_constraint` of a specific form. The `numcall` attribute could be extracted using r-TuBounds static profiler but is currently not considered (it encodes the total number of calls to a function). Some WCETC constructs cannot yet be translated to FFX, for example the `#pragma wcet_constraint` construct which limits the execution count of a program block.

The example in Figure 5.12 presents a snippet of code from the Mälardalen [34] benchmark `bs.c`, annotated with flow information as used in r-TuBound, and its representation in FFX format. In r-TuBound `#pragma wcet_marker` is used to identify blocks and associate analysis information with them (e.g. the `#pragma wcet_loopbound`). FFX represents locations as FFX elements with line number attributes associated with them instead of markers. The loop bound is annotated as `maxcount` attribute of the `loop` element. Most of the additional information (`source`, `condition`, `extern`) can be extracted from the source. Expressing flow facts not only in WCETC but also in FFX offers numerous advantages: most important, we can compare WCET tools on multiple levels. Further, it allows to specify flow facts in an unambiguous way for different tools and to keep analysis information persistent. It offers the possibility of merging FFX files from different tools, i.e. acquiring a tighter WCET estimate by using the most exact information available, e.g., the tightest loop bounds.

```
    ...
// main calling
// binary_search
int binary_search(int x) {
#pragma wcet_marker(label30)
  ...
  while (low <= up) {
#pragma wcet_marker(label23)
    mid = ((low + up) >> 1);
    if (data[mid].key == x) {
#pragma wcet_marker(label18)
      up = (low - 1);
      fvalue = (data[mid].value);
    } else  {
      if ((data[mid].key) > x) {
#pragma wcet_marker(label21)
        up = (mid - 1);
      } else {
#pragma wcet_marker(label22)
        low = (mid + 1);
      }
    }
#pragma wcet_loopbound(8..8)
  }
  return fvalue;
}
```

```
...
<call name="binary_search" numcall="1"
  line="54" source="bs.c" executed="true"
  extern="false">
  <function name="binary_search"
    executed="true" extern="false">
    <loop loopId="0" line="67" source="bs.c"
      exact="false" maxcount="8">
      <conditional>
        <condition varcond="IF-1" line="70"
          source="bs.c" isexecuted="true"
          expcond="data[mid].key==x;" />
          <case cond="1" executed="true" />
          <case cond="0" executed="true">
            <conditional>
              <condition varcond="IF-2"
                line="79" source="bs.c"
                isexecuted="true"
                expcond="data[mid].key>x;"/>
              <case cond="1"executed="true"/>
              <case cond="0"executed="true"/>
            </conditional>
          </case>
      </conditional>
    </loop>
  </function>
</call>
...
```

Figure 5.12: Part of the Mälardalen benchmark s.c: on top, the original annotations as output after high-level analysis by r-TuBound, on bottom, the FFX translation.

Based on our experience with FFX we are confident that FFX is a suitable open format to store, exchange and collect flow fact information for later use in the WCET analysis of systems.

One major advantage of FFX is, illustrated in our case study, that source level analyzers supporting the FFX format can interchange WCET back-ends. At the same time it offers a way of comparing WCET tools and it allows to refine and possibly tighten WCET results by merging flow facts from different tools. In future work, we will introduce an order on flow facts and other FFX constructs that allows to determine in which manner FFX files should automatically be merged to gain better accuracy for WCET analysis. At the same time one could introduce consistency rules for relevant flow fact information. These rules can then be used to check validity and precision of gathered flow facts.

FFX allows to specify flow facts in an unambiguous way, therefore as future work, we propose to extend FFX in a way that makes it possible to encode problems from the WCET tool challenge, as this would allow for more exact problem specifications and tool comparisons. Additionally, we plan on investigating FFX for a larger experiment with additional WCET and flow fact analyzers involved.

CHAPTER 6

Practice – Implementation & Experimental Results

Our overall approach to proving precise WCET bounds is *implemented in the r-TuBound WCET tool chain* [43, 44]. That is, our implementation in r-TuBound offers automated support for WCET Squeezing, selective symbolic execution, loop bound computation, and interoperability among WCET analyzers using FFX annotations.

In the following, we describe the r-TuBound tool, present its overall workflow, give implementation details and illustrate the usage of r-TuBound via examples. We then present our experimental evaluation of using r-TuBound in the timing analysis of programs, by using a large number of challenging WCET benchmakrs. To this end, we report on our experimental results by using r-TuBound for WCET Squeezing, selective symbolic execution, and loop bound computation. We also evaluated r-TuBound in comparison with the Otawa/oRange WCET analysis tool chain, by using the FFX annotation languages. Finally, we summarize our results from using r-TuBound in the WCET Tool Challenge 2011.

6.1 r-TuBound: Overview and Implementation

We have implemented the work reported in this thesis in the *WCET analysis toolchain r-TuBound*. Our implementation uses the general WCET framework of the TuBound tool [56], and extends TuBound with automated techniques for proving precise WCET bounds, computing tight loop bounds and annotating flow facts based on the FFX format. The mains steps of r-TuBound are illustarted in Figure 6.1, and detailed below.

r-TuBound uses a C/C++ source-to-source transformer based on the ROSE compiler framework [47]. It also uses the static analysis libraries of SATIrE [62] and TERMITE [53] for implementing a forward-directed data flow interval analysis, a points-to analysis and a simple constraint-based loop bound analysis. Further, r-TuBound relies on

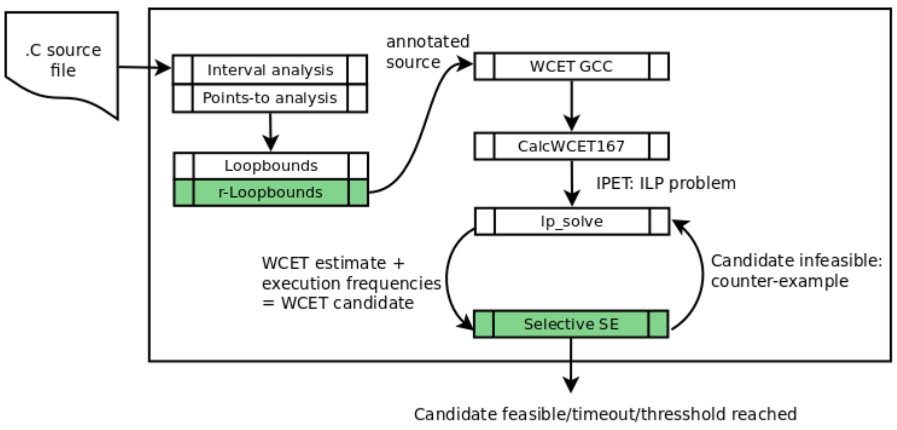


Figure 6.1: Overall workflow of the r-TuBound WCET toolchain.

a WCET-aware C compiler, based on the GNU C compiler 2.7.2.1 ported to the Infineon C167CR architecture with added WCET analysis functionality, and the CalcWCET167 static WCET analysis tool [56] which supports the Infineon C167CR as target processor.

Inputs to r-TuBound are arbitrary C/C++ programs The input program is parsed and analysed. As a result all loops and unstructured goto statements of the input code are extracted. To this end, an initial WCET analysis supported by CalcWCET167 is applied, such as parsing the EDG C/C++ frontend, building the abstract syntax trees and the control-flow graph of the program, interval analysis over program variables, and points-to analysis. Next, loop bounds are inferred by r-TuBound using pattern-based recurrence solving in conjunction with SMT solving, as presented in Chapter 4. These loop bounds are then used by the CalcWCET167 static analyzer and an initial WCET bound is derived. Further, r-TuBound applies WCET Squeezing to tighten and prove precision of the WCET bound, as discussed in Chapter 3. As a result, r-TuBound outputs a precise WCET bound for its input program.

Our implementation is available at `www.complang.tuwien.ac.at/jakob/tubound/`. To invoke r-TuBound, one uses the following command:

Command: `rTuBound program.c`
Input: C/C++ program
Output: precise WCET of functions in `program.c`

Example 6.1 Consider the `test.c` program given in Figure 6.2. By running r-TuBound on this example, the precise WCET bound of the function is computed.

Loop Bound Computation in r-TuBound

After the initial CFG construction and interval and points-to analysis step, r-TuBound proceeds to the automated computation of loop bounds. The loop bound computation

```
void test() {
  int i = 0, a[16];
  while (i < 100) {
    if (a[i] > 0)
      i = i * 2 + 2;
    else
      i = i * 2 + 1;
    i = i + 1;
  }
}
```

Figure 6.2: Input C program `test.c`

```
void test() {
  int i, a[16];
  for (i = 0; i < 100; i = i + 1)
    if (a[i] > 0)
      i = i * 2 + 2;
    else
      i = i * 2 + 1;
}
```

Figure 6.3: C program `test.c`

step of r-TuBound is summarized in Figure 6.4. The extracted loops and goto statements are rewritten, whenever possible, into the format given in Chapter 4– Figure 4.5. Doing so requires, among others, the following steps: rewriting while-loops into equivalent for-loops, rewriting if-statements into if-else statements, translating multi-path loops with abrupt termination into loops without abrupt termination, and approximating non-deterministic variable assignments. The aforementioned steps for parsing, analysing and preprocessing C/C++ programs are summarized in the `loopExtraction` part of Figure 6.4. If a program loop cannot be converted into Figure 4.5 by the `loopExtraction` part of r-TuBound, step of `loopRefine`, the flow refinement in `loopRefine` fails. As a further result, r-TuBound does not compute a loop bound and hence the WCET computation step of r-TuBound will fail. Next, the loops extracted in the format given in equation (4.5) are analysed and translated into equation (4.4) of Section 4.2, required by the loop bound computation engine of r-TuBound. This step is performed in the `loopRefine` part of Figure 6.4. As a result of `loopRefine`, the multi-path loops of equation (4.5) are translated into the simple loops presented in equation (4.4).

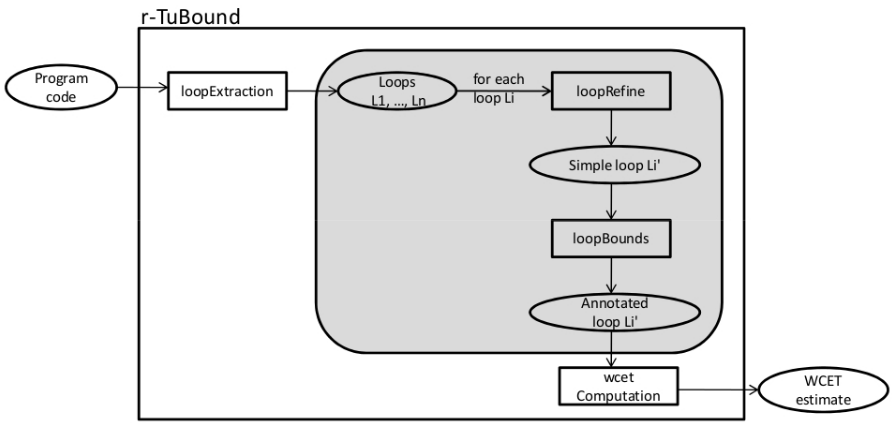


Figure 6.4: Loop bound computation in r-TuBound.

```
for (i = 0; i < 100;
     i = i + 1) {
  if (a[i] > 0)
    i = i * 2 + 2;
  else
    i = i * 2 + 1;
}
```

Figure 6.5: Multi-path loop from `test.c` (`mpath1.c`)

```
for (i = 0; i < 100;
     i = 2 * i + 3) {
}
```

Figure 6.6: Over-approximation (`simple1.c`)

```
for (i = 0; i < 100;
     i = 2 * i + 3) {
  #wcet\_loopbound(6)
}
```

Figure 6.7: Annotated with loop bound (`annot1.c`)

For deriving loop bounds in the `loopBounds` step of Figure 6.4, each loop is analysed separately and bounds are inferred using recurrence solving. The computed loop bounds are added as annotations to the loops and are further used by the `wcetComputation` engine of Figure 6.4 to calculate the WCET of the input program, as illustrated in Figure 6.1 and detailed later.

Example 6.2 Consider again the `test.c` program of Figure 6.2. The while-loop of Figure 6.2 is first translated into equation (4.5), as given in Figure 6.3. The multi-path loop of Figure 6.3 is then over-approximated by a simple loop with linear updates and conditionals, over which pattern-based recurrence solving is applied. As a result, r-TuBound derives the (over-approximated) loop bound of Figure 6.2 to be 6 iterations.

Given a multi-path loop (4.5), the `loopRefine` part of r-TuBound translates (4.5) into a simple loop, such that the loop bound of the simple loop is also a loop bound of the multi-path loop (4.5). To this end, the multi-path behavior of (4.5) is safely over-approximated by using the minimal conditional update m in (4.4). The task of choosing m such that, it yields the minimal increase over i, is encoded in r-TuBound as a set of SMT queries. For doing so, we interfaced `loopRefine` with the Boolector SMT solver [17]. To this end, for each variable in the program, a bit vector variable is introduced by `loopRefine`. An array is used to model the values of the variables involved. This representation allows `loopRefine` to capture the loop behavior in a symbolic manner, by using symbolic values to model the updates to the loop counter.

Example 6.3 For translating the multi-path loop given in Figure 6.5, `loopRefine` infers that the conditional update corresponding to the else-branch of the conditional statement of `mpath1.c` yields the minimal update over i. The multi-path loop of Figure 6.5 is thus rewritten into the simple loop given in Figure 6.6, for which loop bounds are then further inferred. Figure 6.7 shows the result of loop bound computation for Figure 6.6, where the annotation `#wcet_loopbound(6)` specifies that `loopBounds` computed 6 as the loop bound of Figure 6.6.

The pattern-based recurrence solving algorithm and the satisfiability checking of SMT formulas for computing tight loop bounds in r-TuBound are implemented in `loopBounds` on top of Prolog.

loopBounds operates on the TERM representation offered by the Termite library [53]. Let us note that the closed form representation of loop iteration variables involve, in general, exponential sequences in the loop counter. Therefore, to compute the smallest value of the loop counter yielding tight loop bounds (i.e. satisfying equation (4.3)), loopBounds makes use of the logarithm, floor and ceiling built-in functions of Prolog.

The program analysis framework loopExtraction, the loop refinement step loopRefine and the WCET computation part wcetComputation of r-TuBound are written in C++. The loop bound computation engine loopBounds of r-TuBound is implemented on top of the Termite library of Prolog. The loopRefine part of r-TuBound comprises about 1000 lines of C++ code, whereas the loopBounds engine of r-TuBound contains about 350 lines of Prolog code. The loopRefine and loopBounds parts of r-TuBound are glued together using a 50 lines shellscript code.

WCET Squeezing in r-TuBound

After the loop bound computation step, a WCET estimate is derived using CalcWCET167. Running CalcWCET167 on the annotated assembly of a function produces an IPET problem specification formulated as ILP problem. An ILP solver is used to infer an ILP solution that yields the WCET bound and execution frequencies of program blocks.

WCET Squeezing is then applied to prove the computed WCET bound precise: the ILP problem specification and solution and the annotated assembly are supplied to a modified version of CalcWCET167. The modified CalcWCET167 then extracts the branching behaviour from the assembly and the ILP. The branching behaviour is used to construct WCET candidates (c.f. Sect.3).

```
unsigned char IN;
unsigned char OUT;
void main () {
  int i;
  unsigned char a;
  for(i = 0; i < 10; i++) {
#pragma wcet_marker(labelFOR)
    a = IN;
    if(i < 5) {
#pragma wcet_marker(labelIF)
      a = a & 15;
      OUT = a;
    }
  }
}
```

Figure 6.8: WCET Squeezing example, lcdnum.c

```
l_LP_int = f3; 1 = f3; f3 = f4;
f4 = f5; f5 = mlab13; f5 = f6;
f6 = f7; f7 = mLOOP_MRK_0;
f7 + t19 = f8 + t8; t9 = f20;
f8 = t9; t8 = f10; f10 = f11;
f11 = mLOOP_MRK_1; f11 = f12;
f12 = mlab11; f12 = f13;
f13 = mlabFOR; f13 = f14 + t14;
f14 = f15; f15 = mlab10;
f15 = f16; f16 = mlabIF;
f16 = f17; f17 + t14 = f18;
f18 = t19; p14 <= t14;
p8 <= t8; p19 <= t19;
p19 <= p14+p8+mLOOP_MRK_0;
mLOOP_MRK_1 <= 10mLOOP_MRK_0;
f20 = t22; t22 <= 1;
```

Figure 6.9: The ILP problem.

```
Value of objective function: 21680
    mLOOP_MRK_0 1 mLOOP_MRK_1 10
    f3          1 f4           1
    f5          1 f6           1
    f7          1 f8           1
    p8          9 t8          10
    t9          1 f10         10
    f11        10 f20          1
    f12        10 f13         10
    f14        10 f15         10
    f16        10 f17         10
    f18        10 p14          0
    p19        10 t22          1
    t14         0 t19         10
    mlabFOR    10 mlab10      10
    mlab11     10 mlab13       1
    l_LP_int    1 mlabIF      10
```

Figure 6.10: The ILP solution: a WCET bound (21680) and block execution frequencies.

Example 6.4 Consider for example the program given in Figure 6.8 and its IPET problem specification and the ILP solution acquired by the initial WCET analysis step in Figure 6.10. The branching behaviour specified by the ILP yields the WCET candidate `TtTtTtTtTtTtTtTtTtTtF` (for readability, uppercase letters represent loop conditions, lowercase letters conditionals). The specified trace executes the `true`-branch of the conditional in every iteration. This candidate trace is the checked for feasibility using selective symbolic execution, as described later in the SmacC engine of r-TuBound.

Command: `smacc lcdnum.c -sp ttttttttttttttttttttf`
Input: C file and branching behaviour to be checked
Output: Feasibility result in Figure 6.11

```
[IF] @ (19,4) [IF] @ (25,8)    [IF] @ (19,4) [IF] @ (25,8)
[IF] @ (19,4) [IF] @ (25,8)    [IF] @ (19,4) [IF] @ (25,8)
[IF] @ (19,4) [IF] @ (25,8)    [IF] @ (19,4) [IF] @ (25,8)
[UNREACHABLE] abandoning path.
```

Figure 6.11: Feasibility check of branching behaviour `ttttttttttttttttttttf`. The path is infeasible, the output therefore a counterexample.

```
TfTfTtTtTtTtTtTtTtTtF,   TtTtTtTtTtTfTfTtTtTtF,   TtTfTfTtTtTtTtTtTtTtF,
TtTtTtTtTtTtTfTfTtTtF,   TtTtTfTfTtTtTtTtTtTtF,   TtTtTtTtTtTtTtTfTfTtF,
TtTtTtTfTfTtTtTtTtTtF,   TtTtTtTtTtTtTtTtTfTfF,   TtTtTtTtTfTfTtTtTtTtF,
TfTtTtTtTtTtTtTtTtTfF
```

Figure 6.12: WCET trace candidates in the second iteration of WCET Squeezing.

If the trace that exhibits the WCET estimate is infeasible, WCET Squeezing infers an additional constraint from the counterexample. We currently rely on a manually verified mapping between source and assembly, therefore we need to manually encode the resulting counterexample in the ILP, by adding `mlabelIF <= 9` to the ILP problem. The relevant part of the mapping is `[IF] @ (19,4) = mlabelIF` in this case. Solving the ILP again yields a tighter WCET bound and new block execution frequencies that are used in the next iteration of WCET Squeezing. In the second iteration of WCET Squeezing, the extracted branching behaviour specifies a set of trace candidates to be checked for feasibility, listed in Figure 6.12.

We omit here further iterations of WCET Squeezing but note that in the 5th iteration the WCET bound is proven precise.

The changes required to support WCET Squeezing in r-TuBound/CalcWCET167 were minor modifications, i.e extracting the branching behaviour from the ILP solution. Currently, we are working on ROSE source-to-source transformers to automatically construct the mapping between assembly and source code. We further note that, To achieve portability of WCET Squeezing, we use the annotation language FFX. To this end, we use the TinyXML [66], a C++ XML parser, and ROSE source-to-source transformers, as well.

6.2 SmacC: Selective Symbolic Execution in r-TuBound

In this section we describe the selective symbolic execution engine of r-TuBound, which is used in WCET Squeezing for proving precise WCET bounds. Selective symbolic engine of in r-TuBound in implemented in C, yielding a standalone tool, called SmacC, which can be also used independently of r-TuBound. This section overviews the main ingredients of SmacC.

SmacC is a retargetable symbolic execution engine for C programs and is an acronym for *SMT Memory-model and Assertion Checker for C.* Retargetability, a term borrowed from [29] which inspired the front-end implementation of SmacC, refers to its capability of being retargetable to conceptually quite different applications in program analysis. SmacC applies path-wise symbolic execution of a supplied C program that lies in the supported subset of (ANSI) C. SmacC derives verification conditions for program statements and expressions, expressed as satisfiability modulo theory (SMT) formulas in the logic of bit-vectors with arrays. This allows bit-precise reasoning about the program,

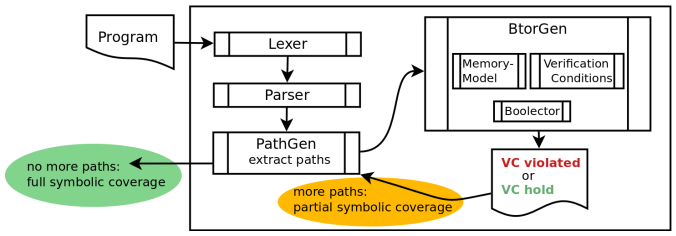


Figure 6.13: Architecture of SmacC: path-wise execution leads to *partial symbolic coverage* if there are more paths to be executed. Exhaustive execution of all paths yields *full symbolic coverage*.

including reasoning about memory accesses and arithmetic overflow. The generated verification conditions precisely captures the memory-model of the program. Proving them to hold guarantees that the supported runtime- and memory-errors cannot occur. Violations in the symbolic representation constitute actual violations.

Beside WCET Squeezing in r-TuBound, SmacC can be applied in a number of program analysis settings. It can prove absence of runtime-errors if full symbolic coverage is achieved. Further, it allows to perform bounded model checking by exhaustive symbolic execution up to a provided bound. Functional correctness, e.g. equivalence checking, is supported via assertions. Generated verification conditions can be dumped to files and used as SMT benchmarks for testing or performance evaluation of SMT solvers.

SmacC: Architecture and Overview

Figure 6.13 shows the architecture of SmacC. SmacC reads a C program as input file, which is then tokenized (`Lexer`) and parsed to abstract syntax trees according to the C expression grammar (`Parser`). The abstract syntax trees are stored as elements of a code-list. Paths through the program are extracted (`PathGen`) and symbolically executed (`BtorGen`), which consists of updating the symbolic representation of the executed path. This symbolic representation is used to generate verification conditions in form of SMT formulas, which express runtime-safety of statements occurring on the path. We use the SMT solver Boolector [17] for checking these SMT formulas in the quantifier-free logic of bit-vectors with arrays. In the sequel, we overview briefly overview the main ingredients of SmacC, give a usage example and then present the architecture and capabilities in more details.

In the path-generation phase (`PathGen`), in order to remove loops, the code-list is flattened, by unwinding program loops up to a certain bound. This way, for each program path, a code-list is constructed. Conditionals, which require to split the control-flow, will produce two paths to explore both branches of the condition. Each fully extracted path is then symbolically executed in `BtorGen`. This step constructs a symbolic SMT representation of the memory used in the program, faithfully covering the semantics of each statement on the program path. Additionally, verification conditions are constructed as SMT formulas. The *program memory* is a collection of symbolic values and modeled by a contiguous array. The memory layout, e.g. the set of declared addresses, is represented by bit-vector variables indexing the memory array. Additional bit-vector variables symbolically track allocated memory regions. Unwritten memory is treated as uninitialized. Verification conditions supported by SmacC include reasoning about *return* statements (check if the program can or returns a specified value), *path conditions* (check satisfiability of conditionals), *division by zero*, and *overflow* of arithmetic operations. Our bit-precise memory-model allows us to construct verification conditions for memory accesses as follows: an access is considered *out-of-allocated* if the address can evaluate to an unallocated array index, i.e. outside the region constrained by *global_beg*, *global_end*, *heap_beg*, *heap_end*, *stack_beg* and *stack_end*.

SmacC produces as output a textual report for each statement symbolically executed along all analyzed paths. For each verification condition, the tool reports whether the property is safe or violated on a specific path. If a verification condition is violated on at least one path, then the corresponding property can be violated by an actual run. If the verification condition holds on all paths, then the corresponding program property cannot be violated by any actual run.

Front-End and Supported C Subset by SmacC

Programs supplied to SmacC must compile with an ANSI C compatible compiler, erroneous programs cannot be handled. Gcc was used as compiler to build SmacC and to compile C examples against which the behaviour of SmacC was tested. In general, a program supplied to SmacC should compile with gcc without warnings, with extra warning flags enabled. The following listing summarizes supported constructs:

- A valid translation unit may only contain global variable declarations of the supported types and one function declaration
- `if-else`, `for`, `assert`, `malloc`, `free`, `sizeof`, `return`, `#include`
- Non-augmented assignment statements, compound statement, valid C expressions (some restrictions)

The front-end gets as input a C file that contains a translation unit which lies in the supported subset of C. The lexer tokenizes the input stream and the parser creates ASTs according to the expression grammar, organizing them as statement elements stored in a code list.

Back-End of SmacC

The back-end gets as its input the code list that was generated by parsing the translation-unit. It extracts and executes paths through the program symbolically by writing to and reading from the BTOR array representing the memory of the program. It generates SMT formulas for the memory layout and verification conditions for statements. These SMT formulas are further used in the SMTSolver Boolector [17]. Figure 6.14 illustrates the BTOR format and basic usage of Boolector.

Example 6.1 Boolector prints a (partial) model in the satsifiable case of formulas, when supplying `-m`, while `-d` enables decimal output. In line 1 an array with element width 8 bit and index width 32 bit is constructed. Line 2 declares a 32 bit bit-vector variable named `index`. Line 3 declares an 8 bit bit-vector constant with value 0 that is written to array 1 on position index (2) in line 4, constructing a new array. Line 5 states that array 1 is equal to array 4. Line 6 sets line 5 as root node such that the formula can be checked with Boolector stand-alone version. Boolector returns 'satisfiable' because it is possible that the element at index `index` of array 4 has the same value as the element at index `index` in array 1.

```
$> cat example.btor
1 array 8 32
2 var 32 index
3 const 8 00000000
4 write 8 32 1 2 3
5 eq 1 1 4
6 root 1 5
$>
```

```
$> boolector example.btor -m -d
sat
index 0
1[0] 0
$>
```

Figure 6.14: BTOR file `example.btor` and output of invoking Boolector.

First, the Path-Generation phase extracts paths from the code list by unrolling loops up to a supplied bound, splitting the path on branching points (i.e. conditionals) and adding an assumption about the evaluation of the branching condition.

For supporting WCET Squeezing, we extended the Path-Generation phase by a method to selectively execute paths through the program, by supplying a set of branching decisions to SmacC. That is, at branching points, SmacC will follow only one evaluation of the conditional, as specified by the branching decisions, instead of following both. This way, WCET path candidates that are extracted from the IPET solution as a set of branching decisions, can directly be supplied to SmacC for symbolic execution. For example, supplying the branching behaviour `'t'` to SmacC in 6.15 would result in executing only `path 0` instead of executing both paths.

Second, the Btor-Generation phase constructs the BTOR representation for statements, the memory and verification conditions.

Example 6.2 Consider the C Program in Figure 6.16. The BTOR instance for the return statement `return 0;` is depicted on the right: line 2 represents the constant 0, line 4, 5 and 6 represent the BTOR variables necessary to construct the memory model. The BTOR formula for the return statement is constructed in line 7. The rest of the lines form the constraints for the memory layout and are conjuncted with line 7 and selected as root in line 18. Lines 8 to 10 are used negated in line 13 to 15 to formulate the properties that the end of stack, global and heap area must be greater or equal to the beginning of stack, global and heap area. Initially the addresses that represent the end of the memory areas are equal to the addresses that represent the begin of the

```
int main () {          path 0:                   path 1:
  int cond;            CSENTER @ (1,10)          CSENTER @ (1,10)
  if (cond)              CSENTER @ (1,12)          CSENTER @ (1,12)
    return cond;           CDECLL @ (2,8)            CDECLL @ (2,8)
  return 0;                CIF @ (4,0)               CELSE @ (5,0)
}                            CBBEG @ (4,0)           CRET @ (5,8)
                               CRET @ (4,11)       CSEXIT @ (6,0)
                             CBEND @ (5,0)       CSEXIT @ (6,0)
                           CRET @ (5,8)
                         CSEXIT @ (6,0)
                       CSEXIT @ (6,0)
```

Figure 6.15: Translation-unit and both paths through the program.

```
int main () {           2 const 32 000...000     11 ult 1 5 6
  return 0;             4 var 32 stack_beg       12 ult 1 6 4
}                       5 var 32 global_beg      13 and 1 7 -8
                        6 var 32 heap_beg        14 and 1 13 -9
                        7 eq 1 2 2               15 and 1 14 -10
                        8 ult 1 5 5              16 and 1 15 11
                        9 ult 1 4 4              17 and 1 16 12
                       10 ult 1 6 6              18 root 1 17
```

Figure 6.16: A C Program and the BTOR representation of the return statement.

memory areas. Line 10 and 11 establish the general memory layout which requires that the highest global address is smaller than the lowest heap address which is smaller than the last lowest stack address. Line 17 is the conjunction of the properties mentioned and the formula specifying the return value to be equal to zero.

The Memory Model is inspired by the memory model usually used in UNIX systems. It is established by an SMT formula that constrains the array variable which models the memory of the program. This allows to check whether memory accesses in the program are valid. If a memory access is invalid for the SMT representation it is also invalid for the real program. The UNIX memory model divides memory for a process into three segments [65]:

- Text Segment: machine instructions, executable code
- Global / Data Segment: global variables, constant strings, but also dynamic memory
- Stack Segment: local variables, parameter variables, grows from high address to low address

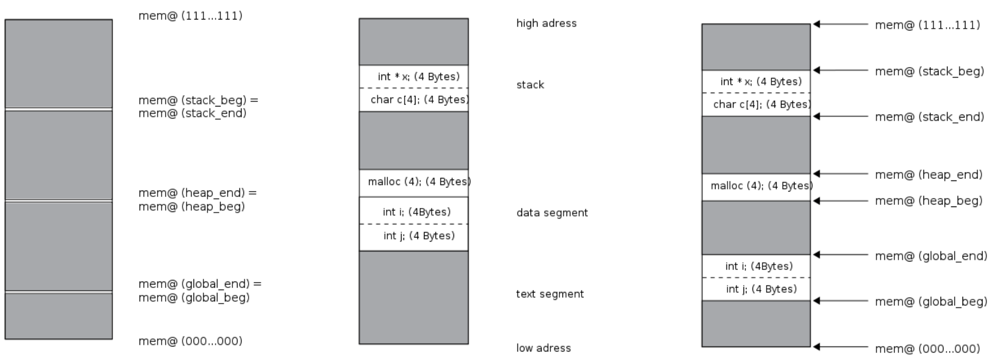


Figure 6.17: Simplified view of the UNIX memory-model of a C program and its representation in SmacC. In the left example no variables are declared. In the right example the program has integers `i` and `j` declared as global variables, integer pointer `x` and character array `c` as local variables and 4 bytes allocated on the heap by a call to `malloc`.

SmacC simplifies the UNIX memory model, there is no text segment, the data segment is called global area and is only used for global variables. Memory that is allocated in the data segment by calls to `malloc` is modeled by the heap area.

The left-hand side of Figure 6.17 is a visualization of the memory layout right after initialization, no variables declared, represented by the following formula:

$$\begin{aligned} global\_beg \leq global\_end \wedge global\_end < heap\_beg \wedge heap\_beg \leq heap\_end \wedge \\ heap\_end < stack\_end \wedge stack\_end \leq stack\_beg \wedge global\_beg = global\_end \wedge \\ stack\_beg = stack\_end \wedge heap\_beg = heap\_end \end{aligned}$$

When variables are declared or dynamic memory is allocated the memory-model needs to be updated to include constraints about the variable. Consider the right-hand side of Figure 6.17, visualizing the memory model after a few variables were declared, represented by the following updates to the memory model:

$$\begin{aligned} i = global\_beg \wedge j = global\_beg + 4 \wedge global\_end = global\_beg + 8 \\ heap\_v1 = heap\_beg \wedge heap\_end = heap\_beg + 4 \\ p = stack\_beg - 4 \wedge c = stack\_beg - 4 - (4 * 1) \wedge stack\_end = stack\_beg - 8 \end{aligned}$$

SmacC considers the following memory accesses invalid:

- Access out of valid memory: an access is considered out of valid memory if it accesses indices that are not indices representing stack area, global area or heap area. Invalid regions are marked grey in Figure refmem.

```
void main () {
  int i;
  assert (i);
}
```

btor_vars $=$
$\{$
$global\_beg, global\_end,$
$heap\_beg, heap\_end,$
$stack\_beg, stack\_end,$
mem, i
$\}$

layout $:= global\_beg \leq global\_end \wedge$
$global\_end < heap\_beg \wedge$
$heap\_beg \leq heap\_end \wedge$
$heap\_end < stack\_end \wedge$
$stack\_end \leq stack\_beg \wedge$
$global\_beg = global\_end \wedge$
$heap\_beg = heap\_end \wedge$
$i = stack\_beg - 4 \wedge$
$stack\_end = stack\_beg - 4$
assert $:= read(mem, i) = 00000000 \wedge$
$read(mem, i + 1) = 00000000 \wedge$
$read(mem, i + 2) = 00000000 \wedge$
$read(mem, i + 3) = 00000000 \wedge$

Figure 6.18: On the left: assertion statement in a C program and declared Boolector variables. On the right: assumptions about memory layout and the formula representing the assertion.

- Access out-of-bounds: an access is considered out-of bounds if it crosses boundaries of data elements, for example when data from two valid regions is read or written. Out of bounds access can happen at all addresses.

SmacC constructs and verifies the following additional verification conditions:

- Assertion statement: verify that assertion statement cannot fail,
- Return statements: check if the program returns a specified value in all cases or check if a specified return value is possible,
- Path conditions: check if an if / else condition is unsatisfiable
- Division by zero: checks if division by zero is possible,
- Overflow: checks for overflow on arithmetic operations,
- Arithmetic Properties about conditional updates.

In the following we will illustrate verification conditions for assertions, memory safety and for verifying arithmetic properties about conditional updates in loops.

Assertions.

Variations of assertion checks are used to verify program return values and to check for division by zero. Consider the example in Figure 6.18.

The conjunction of formulas **layout** $\wedge$ **assert** must be unsatisfiable, otherwise the assertion might fail.

abf_invalid :=
$abf > stack\_beg \wedge$
$abf > global\_end \wedge$
$abf < heap\_beg \wedge$
$abf > heap\_end \wedge$
$abf < stack\_end \wedge$
$abf < global\_beg$

abf_freed :=
$abf \geq free\_var_i \wedge$
$abf < free\_var\_i + free\_var_i\_size$

check :=
$abf = addr$

Figure 6.19: Basic Memory Check: constraining a variable to be outside valid memory or equal to a freed address.

Memory Safety: access out of valid memory.

Checks whether a memory access can reference undeclared memory. This is done by constraining a symbolic address *abf* to reference undeclared memory and asserting that the referenced address *addr* is different from *abf*.

Clearly, because of the constraints on *abf*, if the SMT formula (**abf_invalid** $\vee$ **abf_freed**) $\wedge$ **check** is satisfiable for any byte of *addr*, then invalid memory is accessed.

Arithmetic Properties of Conditional Updates.

The loop bound computation step in r-TuBound relies on the assumption that the loop counter is not modified in the body of the loop. If the loop counter is modified, loop bound computation fails to compute a loop bound. This is an overly restrictive assumption and a safe loop bounds can actually be computed when updates only increase (decrease) the loop counter. In some cases, the updates can be merged to a minimal update. In order to verify this property, we use SmacC to verify whether $(i' = upd(i)) \geq i$.

Applications of SmacC

We have successfully applied SmacC to verify C programs and generate SMT benchmarks using our precise memory-model [72]. We illustrate the bit-precise memory-model and generation and proving of verification conditions using the examples in Figure 6.20(a) and (b) below. We also integrated SmacC with r-TuBound to support timing analysis, and show its use on Figure 6.20(c).

Example 6.3 The variable declarations in the program of Figure 6.20(a) in lines 1 and 3 (a:1,3), result in the following SMT variable declarations, where variables that do not occur in the source are used to track allocated memory: $global\_beg$, $global\_end$, $heap\_beg$, $heap\_end$, $stack\_beg$, $stack\_end$, mem, i, x, a, where mem is an array and models memory. Symbolic execution of a path tracks declared memory constructing the formula $(a = global\_beg) \wedge (global\_end = global\_beg + 16) \wedge (heap\_end = heap\_beg) \wedge (i = stack\_beg) \wedge (stack\_end = stack\_beg - 4)$, while $(read(mem[i]) < 0 \ldots 100)$ is the verification condition for the assertion (a:8). The assertion holds for any variable assignment valid on the current path if the conjunction of the formulas is unsatisfiable. Figure 6.20(b), taken from

```
int a[4];
int main () {
  int i;
  a[0] = 1;
  for(i = 0; i < 4; i++)
    if (a[i] > 0)
      i = i + 1;
  assert(i >= 4);
}
```

(a)

```
int main () {
  int x, y;
  if (x > 0) {
    y = x * x;
    if (y == 0)
      assert(0);
  }
}
```

(b)

```
int main() {
  int i, flag;
  for(i = 0; i < 5; i++)
    if(i == 4 && flag) {
      i = 0;
      flag = 0;
    }
}
```

(c)

Figure 6.20: (a) a program with an assertion and a conditional update; (b) a program with a reachable, failing, assertion; (c) SmacC finds the loop bound, CBMC keeps unwinding the loop.

[11], illustrates the need for a bit-precise memory-model: both conditions (b:3,5) must evaluate to *true* to reach the failing assertion (b:6). When reasoning about unbounded integers the assertion is unreachable due to unsatisfiable path conditions. SmacC infers overflow for the multiplication and thus a satisfiable path condition guarding the failing assertion, therefore the failing assertion is reachable.

SmacC has successfully been used in a number of applications, ranging from program verification to high-level WCET analysis. A key feature of SmacC is its bit-precise symbolic execution which enables it to find a number of typical and important program errors and to functionally verify programs via assertions. Verification conditions that exhibit high solving time can be dumped and used as regression and performance tests for SMT solvers. High-level WCET analysis turned out to be a another promising application field of SmacC and we successfully retargeted SmacC and the underlying memory-model towards integration into a WCET analysis toolchain, improving high-level analysis results. SmacC is implemented in 10Klocs of C and is available at `http://www.complang.tuwien.ac.at/jakob/smacc/`

6.3 Experimental Results using r-TuBound

In the following we report on our experimental evaluation of symbolic methods for WCET analysis. We first present the results of WCET Squeezing on a number of examples, followed by an evaluation of our loop bound computation techniques. We then evaluate the use of FFX as a common annotation language in r-TuBound and OTAWA/oRange. Finally, we present the results of applying SmacC as a stand-alone program analysis tool.

WCET Squeezing Experiments

We evaluated WCET Squeezing on 10 examples taken from the Mälardalen WCET benchmark suite in detail. We summarize our results in Table 6.1. Column 1 of Table 6.1 reports the benchmark's name and Column 2 lists the relevant functions in the benchmark. Column 3 gives the initial WCET estimate as reported after WCET analysis by r-TuBound. Column 4 describes the WCET estimate obtained after running WCET Squeezing. Column 5 lists how many iterations of WCET Squeezing were executed and Column 6 gives the number of execution traces that were excluded. Columns 7 and 8 report on the obtained WCET improvements, as follows: Column 7 describes the achieved improvement, whereas Column 8 denotes the maximum improvement, i.e., the improvement necessary for a precise estimate. Column 9 describes whether the WCET estimate for the function of Column 2 is proven precise.

For the functions `prime`, `cl_block`, and `icrc1` the initial WCET candidate is feasible, hence the initial bound is proved to be precise, which was unknown before, and no improvement is possible. For the functions `adpcm`, `duff`, `expint`, and `fibcall` continuous improvement is achieved until WCET Squeezing terminates, this time proving precision of the squeezed WCET. For the functions `expint` and `janne_complex`, both can be improved by more than 90%, we report the impact of a single iteration of WCET Squeezing to demonstrate the different impacts of excluding a single WCET trace candidate: while the impact amounts to roughly 1% for the first program, it amounts to almost 6% for the second. This reflects the fact of how much the excluded conditional block contributes to the WCET estimate of the function. Similarly, this holds for `lcdnum` and `nsichneu`: executing two iterations of WCET Squeezing results in an improvement of more than 7% for `lcdnum` and less than 0.5% for `nsichneu`. For `expint`, the high over-estimation of the WCET is due to the fact that the WCET toolchain initially assumes the inner loop to be executed in every iteration. Squeezing reveals that the inner loop is only executed in the last iteration. Similarly, in `janne_complex` over-estimation is due to a complex interleaving of nested loops, ultimately inferred by WCET Squeezing.

Loop Bound Computation Experiments

We evaluated the loop bound computation part of r-TuBound on examples coming from the WCET community [59], the SciMark scientific repository [1], and from industrial applications of Dassault Aviation. We report on our results, as follows.

In the following tables, column 1 ("BM") denotes the name of the benchmark file/program, and column 2 gives the number of loops in the corresponding file ("#L"). Column 3 lists how many of those loops were successfully analyzed by TuBound ("TB"), whereas column 4 lists how many loops were analyzed by the symbolic loop bound computation technique of r-TuBound("r-TB"). Column 5 ("G") shows which loop pattern (cf. the notation used in Table 6.6) that could only be handled by r-TuBound[1]. In the sequel we give a detailed analysis of our experiments.

[1]Subtracting column 5 from column 4 yields the number of simple loops with constant increments/decrements (i.e. $c = 1$ in Figure 4.2)

BM	function	WCET	squeezed	#iters	#π	%imp	%prec	note
prime	prime	784860	784860	1	0	0	0	PPO
compress	cl_block	8440	8440	1	0	0	0	PPO
crc	icrc1	18060	18060	1	0	0	0	PPO
adpcm	logsch	5560	5380	1	1	3.24	3.24	PPW
	uppol2	12260	12040	2	2	9.87	9.87	PPW
duff	duffcopy	85940	79949	5	5	7.5	7.5	PPW
fibcall	fib	79840	75040	2	2	6.39	6.39	PPW
expint	expint	1.24E7	1.22E7	1	1	0.94	93	I
janne-cmp	complex	694980	653380	1	1	5.9	93.3	I
lcdnum	main	24320	22520	2	11	7.4	18.5	I
nsichneu	main	4.95E6	4.94E6	2	2	0.28	-	*I

Table 6.1: Proving precision of WCET estimates by running WCET Squeezing until termination. The lower part focuses on the impact of only a few iterations. PPO stands for "precision proved without refinement", PPW denotes "precision proved with refinement", "I" denotes an imprecise WCET bound and "*" denotes that the actual WCET is unknown.

WCET Benchmarks. We investigated two benchmark suites that originate from the WCET community: the Mälardalen Real Time and the Debie-1d benchmark suites [36].

From the Mälardalen suite we investigated 31 files, containing 207 loops. Table 6.2 reports on these experiments. r-TuBound derived loop bound for 165 loops, whereas TuBound was able to analyze 163 loops. The two additional loops analyzed by r-TuBound required recurrence solving and control flow refinement. By analyzing our results, we observed that 120 out of these 121 loops involved only incrementing/decrementing updates over iteration variables, and 1 loop required more complex C-finite recurrence solving as described in Section 4.2. When compared to TuBound [56], we noted that TuBound also computed bounds for the 120 Mälardalen loops with increment/decrement updates. However, unlike r-TuBound, TuBound failed on analyzing the loop with more complex C-finite behavior.

Table 6.3 shows the performance of r-TuBound on examples from the Debie benchmark used in the WCET tool competition. Altogether, we used 8 example files containing 75 loops. TuBound was able to compute loop bounds for 58 loops. In addition to these 58 loops, r-TuBound also inferred loop bound to one additional shift-loop. By analyzing the results of r-TuBound, we conclude the following. r-TuBound successfully analyzed 59 loops. These 59 loops can be classified as follows: 57 simple loops with increment/decrement updates, 1 shift-loop with non-deterministic initialization, and 1 multi-path loop with conditional update. When compared to TuBound, we observed that TuBound could analyze the 58 simple loops and failed on the multi-path loop. Moreover, r-TuBound derived a tighter loop bound for the shift-loop than TuBound.
Scientific Benchmarks. Table 6.4 gives the performance of r-TuBound on 34 loops

BM	#L	TB	r-TB	G	BM	#L	TB	r-TB	G
adpcm	18	17	17	-	qurt	1	1	1	-
bs	1	0	0	-	select	4	0	0	-
bsort100	3	3	3	-	statemate	1	0	0	-
cnt	4	4	4	-	sqrt	1	1	1	-
cover	3	3	3	-	fft1	11	6	6	-
crc	3	3	3	-	lms	10	6	6	-
duff	2	1	1	-	whet	11	11	11	-
edn	12	12	12	-	ludcmp	11	11	11	-
expint	3	3	3	-	compress	7	2	3	CF
fibcall	1	1	1	-	fir	2	1	1	-
janne_cmp	2	0	0	-	minver	17	16	16	-
nsichneu	1	0	0	-	qsort_exam	6	0	1	AT
insertsort	2	0	0	-	fdct	2	2	2	-
jfdctint	3	3	3	-	lcdnum	1	1	1	-
matmult	5	5	5	-	ndes	12	12	12	-
ns	4	4	4	-	**sum**	207	163	165	2

Table 6.2: Evaluation of r-TuBound on the Mälardalen benchmarks.

BM	#L	TB	r-TB	G
class	2	2	2	SH
debie	1	0	0	-
harness	45	34	34	-
health	11	9	10	CU
hw_if	3	2	2	-
measure	6	4	4	-
tc_hand	3	3	3	-
telem	4	4	4	-
sum	75	58	59	1

Table 6.3: Evaluation of r-TuBound on the Debie benchmarks.

from the SciMark scientific benchmark suit. Among other arithmetic operations, SciMark2 makes use of fast Fourier transformations and matrix operations. r-TuBound derived loop bounds for 26 loops, whereas TuBound could analyze 24 loops. The 2 loops which could only be handled by r-TuBound required reasoning about abrupt-termination and C-finite updates.

Industrial Benchmarks. We also evaluated r-TuBound on 77 loops coming from Dassault Aviation, as summarized in Table 6.5. Out of the 77 loops, r-TuBound derived loop bounds for 46 loops, whereas TuBound analyzed 39 loops. The 7 loops that could only be analyzed by r-TuBound included 4 loops with C-finite and conditional updates,

BM	#L	TB	r-TB	G	BM	#L	TB	r-TB	G
array	6	5	6	AT	scimark	0	0	0	-
fft	8	4	5	CF	sor	3	3	3	-
kernel	9	4	4	-	sparsecomprow	3	3	3	-
montecarlo	1	1	1	-	stopwatch	0	0	0	-
random	4	4	4	-	**sum**	34	24	26	2

Table 6.4: Evaluation of r-TuBound on the SciMark2 examples.

```
for (i = x; i < 65536;
     i = i * 2) ;
```

(a) Loop with C-finite update.
Iteration bound is 12.

```
int s = x;
while (s)
  s >>= 1;
```

(b) Shift-loop.
Iteration bound is 31.

Figure 6.21: Examples from[36] (1).

```
while (i > 0)
  if (i >= c)
    i = -c;
  else
    i -= 1;
```

(c) Loop with conditional updates.
With no initial value information on
i, the bound is `INT_MAX`

```
int M, N;
M = 12;
for (i = 0; i < M; i++) {
  A[i] = malloc (N);
    if (!A[i])
      break;
```

(d) Abruptly terminating loop.
Iteration bound is 12

Figure 6.22: Examples from[36] (2).

2 abruptly terminating loops with C-finite and conditional updates, and 1 abruptly terminating loop with C-finite updates. When compared to TuBound, the success of r-TuBound hence lies in its power to handle abrupt termination, conditional updates, and C-finite behavior.

Summary of Experiments. Altogether, we ran r-TuBound on 4 different benchmark suites, on a total of 393 loops and derived loop bounds for 296 loops. Out of these 296 loops, 286 loops were simple and involved only C-finite reasoning, and 10 loops were multi-path loops which required the treatment of abrupt termination and conditional updates. TuBound could handle 284 simple loops only. Figures 6.21 and 6.22 show examples of loops that could be analyzed by r-TuBound, but not by TuBound.

Table 6.6 lists a summary of the experimental results obtained by using r-TuBound on the aforementioned four benchmark suites. Column 1 lists the benchmark suite, column 2 the number of loops contained, columns 3 and 4 list respectively the number of loops

BM	#L	TB	r-TB	G	BM	#L	TB	r-TB	G
all_zeros	3	1	1	-	min_sort	2	2	2	-
array_ptr	3	3	3	-	m_sort	2	2	2	-
asmmset2	2	1	1	-	muller	2	2	2	-
behavior	1	1	1	-	nb_occ	1	1	1	-
b_s_o	1	0	0	-	negate	1	1	1	-
break	3	1	1	-	octvspoly	1	0	0	-
bresenham	1	1	1	-	permsrch2	1	0	0	-
bsearch	1	0	0	-	permsrch	1	0	0	-
bts-bis	3	3	3	-	r_strcpy	1	0	0	-
bts	3	3	3	-	strconst	1	0	0	-
continue	3	0	3	CU,CU,CU	structhack	3	0	0	-
copy	1	0	0	-	sum1	1	1	1	-
count_bits	1	0	0	-	sum2	2	2	2	-
dillon4	1	1	1	-	t5_floats	1	0	0	-
division	1	0	0	-	trace	2	2	2	-
dowhile	1	1	1	-	vamos	2	0	0	-
fs253	1	0	0	-	vieira1	2	2	2	-
fs256	1	1	1	-	vieira2	1	0	0	-
fs350	1	1	1	-	weber1	1	1	1	-
ghost_lbl	1	1	1	-	weber3	1	0	0	-
heap	2	0	2	AT, CF-AT-CU	weber4	1	0	0	-
heapsort	3	1	2	CF-AT-CU	weber5	1	0	0	-
inv_p_min	2	1	1	-	weber6	1	0	0	-
loop_eq	1	0	0	-	weber8	1	0	0	-
loop_inv	1	0	1	CU	weber9	1	1	1	-
malloc	1	1	1	-					
					sum	77	39	46	7

Table 6.5: Evaluation of r-TuBound on examples sent by Dassault Aviation.

analyzed by TuBound and r-TuBound. Column 5 describes the type of loop and why they could only be analyzed by r-TuBound. To this end, we distinguish between simple loops with C-finite updates (CF), shift-loops with non-deterministic initializations (SH), multi-path loops with abrupt termination (AT), and multi-path loops with monotonic conditional updates (CU). Column 5 also lists, in parenthesis, how many of such loops were encountered. For example, among the loops sent by Dassault Aviation 4 multi-path loops with monotonic conditional updates, denoted as CU(4), could only be analyzed by r-TuBound. Some loops require combinations of the proposed techniques, for exampe, multi-path loops with C-finite conditional updates and abrupt termination; such loops

BM	**#Loops**	**TuBound**	**r-TuBound**	**Types**
Mälardalen	207	163	165	AT, CF
Debie	75	58	59	SH, CU
Scimark	34	24	26	AT, CF
Dassault	77	39	46	AT, CF-CU-AT (2), CU(4)
Total	393	284	296	AT (3), SH, CU (5), CF (2), CF-CU-AT (2)

Table 6.6: Experimental results and comparisons with r-TuBound and TuBound.

are listed in Table 6.6 as CF-CU-AT.

Table 6.6 shows that 75.31% of the 393 loops were successfully analyzed by r-TuBound, whereas TuBound succeeded on 72.26% of the 338 loops. That is, when compared to TuBound, the overall quality of loop bound analysis within r-TuBound has increased by 3.05%. This relatively low performance increase of r-TuBound might thus not be considered significant, when compared to TuBound.

Let us however note that the performance of r-TuBound, compared to TuBound, on the WCET and scientific benchmarks was predictable in some sense. These benchmarks are used to test and evaluate WCET tools already since 2006. In other words, it is expected that state-of-the-art WCET tools are fine tuned with various heuristics so that they yield good performance results on loops occurring in ”classical” WCET benchmarks, including Debie-1D, Mälarden, or even Scimark.

The benefit of r-TuBound wrt TuBound can be however evidenced when considering new benchmarks, where loops have more complicated arithmetic and/or control flow. When compared to TuBound, the overall quality of loop bound analysis within r-TuBound has increased by 3% (72% to 75%). However, TuBound already performed well on Mälardalen and Debie, and therefore the increase given by r-TuBound is only of 1% (78% to 79% in Mälardalen and 77% to 78% in Debie). For the SciMark2 and the examples from Dassault, the increase in performance given by r-TuBound is of 6% (70% to 76%) and 9% (50% to 59%), respectively. The practical importance of r-TuBound can thus be better evidenced on examples with more complex arithmetic and/or control-flow.

The programs which can only be handled by r-TuBound require reasoning about multi-path loops where updates to scalars yield linear recurrences of program variables (in many cases, with $c \neq 1$ in (4.2)). These recurrences cannot be solved by the simple variable increment/decrement handling of TuBound. Moreover, TuBound fails in handling multi-path loops. Based on the results obtained on these new benchmark suite, we believe that our pattern-based recurrence solving approach in conjunction with SMT reasoning for flow refinement provides good results for computing bounds for complex loops with r-TuBound.

We note that in our experiments we did not use r-TuBound in conjunction with the software model checking extension of [55]. The results reported below were obtained by only deploying the symbolic loop bound computation framework of Section 4.2 implemented in r-TuBound.

FFX Experiments

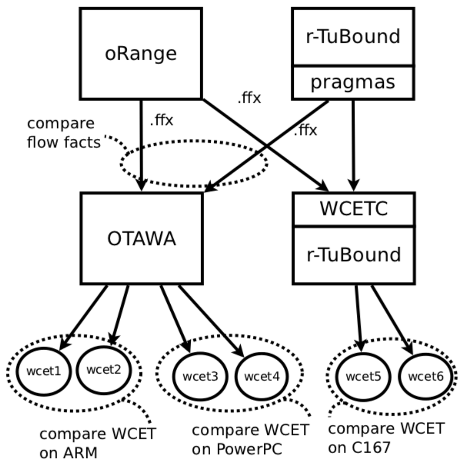


Figure 6.23: Current use of FFX as intermediate format for r-TuBound/CalcWCET167 and OTAWA/oRange. The experiments were performed for the ARM and C167 architecture.

Our experimental case study described in this section focuses on the aspect of comparability, information exchange and extending WCET results to previously unsupported platforms by translating the most important flow facts to FFX and WCETC, respectively. Extensions to this work could focus on using a larger subset of FFX to also allow other tools to participate in information and WCET back-end interchange.

The translation from WCETC to FFX and vice versa allows to investigate differences in flow facts derived from the high-level (e.g. loop bounds) and to study their effect on the tightness of the WCET calculation for a specific platform.

In our case study about FFX-based WCET analysis, we use and compare r-TuBound/CalcWCET167 and oRange/Otawa. Figure 6.23 illustrates the concrete setup of the experiments: the leftmost and the rightmost arrows represent the default workflow and flow of information for each toolchain. Arrows crossing from one toolchain to the other correspond to flow fact translation, either from FFX to WCETC or from `pragmas` to FFX. There exist multiple locations where the tools can exchange information and where the output of the tools can be compared. On flow facts level, depicted in the top left of the diagram, one can compare the FFX output of oRange with the FFX output translated from r-TuBound `pragmas`. Similarly, on WCETC/`pragma` level. On the low level, at the bottom of the diagram, one can compare the WCET estimates for architectures supported by OTAWA and CalcWCET167 by supplying them with translated r-TuBound and oRange flow facts, respectively.

We will present a synthetic example to illustrate the precision gain from combining flow facts from different analyzers. Additionally, we perform experiments on three WCET benchmarks taken from the Mälardalen [34] benchmark family, **bs.c**, **cnt.c** and

minver.c. All of them are small enough to manually inspect and compare the tool outputs. We choose benchmark **bs.c** because oRange performs better flow fact analysis than r-TuBound, **cnt.c** because oRange and r-TuBound infer the same flow facts and **minver.c** because r-TuBound performs better flow fact analysis on the benchmark than oRange. We summarize our results in Table 6.7. The columns X+Y denote the WCET obtained using X as the flow fact analyzer (r for r-TuBound and o for oRange) and the Y back-end (C167 for CalcWCET167 and ARM for ARM Otawa back-end). Therefore, only columns with same Y back-end (columns 2 and 3 or 4 and 5) are meaningful to compare.

BM	**Fct**	**r+C167**	**o+C167**	**r+ARM**	**o+ARM**	**Notes**
bs	main	920	920	1220	**815**	
	b_s	29220	**19140**	1180	**775**	bound 8 vs. 5
cnt	Init	216020	216020	13175	13175	
	InitS	920	920	45	45	
	main	1120	1120	31620	31620	
	RandI	1840	1840	35	35	
	Sum	39700	39700	18265	18265	
	Test	12360	12360	31530	31530	
	ttime	920	920	35	35	
minver	main	167760	167760	**140920**	-	
	minver	**910640**	-	**98905**	-	bound for `while`
	mmul	474880	474880	39145	39145	
	mfabs	7500	7500	250	250	

Table 6.7: BM denotes the benchmark, **Fct** lists functions in the benchmark, the next 4 columns denote the WCET result on different platforms using the given flow fact analyzer and WCET back-end. The last column points out the difference in the flow facts.

(a) The WCET resulting from oRange flow facts on the C167 platform for function `b_s` is tighter than the WCET for r-TuBound flow facts (19140 vs. 29220). The WCET when using r-TuBound flow facts is tighter for function `minver` (910640 vs. unbound). (b) The WCET estimate on ARM is tighter when using oRange flow facts for the analysis of function `b_s` (775 vs. 11890), and tighter for function `minver` when using r-TuBound flow facts (98905 vs. unbound). Thus, in this case, one can merge the flow facts to achieve a better WCET than with the original toolchain (Table 6.8). The difference in WCET in this case is only due to differences in the loop bounds. We are currently investigating whether and how much the WCET result will diverge for larger benchmarks with additional differences in the FFX files (e.g. differences in flow information about infeasible paths).

The piece of code from Figure 6.24 shows the gain from combining flow facts extracted by different flow fact analyzers. The example is synthetic and its only purpose

```
struct DATA { int key; int value; };
struct DATA data[15] = {
  {1, 100}, {5,200}, {6, 300},
  {7, 700}, {8, 900}, {9, 250},
  {10, 400}, {11, 600}, {12, 800},
  {13, 1500}, {14, 1200}, {15, 110},
  {16, 140}, {17, 133}, {18, 10}
};

void main (void) {
  int i, nondet;
  int mid, up, low, x;

  for (i = 1; i < 100; i++) {
    if (nondet)
      i = i * 2 + 1;
    else
      i = i * 2 + 2;
    low = 0;
    up = 14;

    while (low <= up) {
      mid = (low + up) >> 1;
      if (data[mid].key == x)
        up = low - 1;
      else if (data[mid].key > x)
        up = mid - 1;
      else
        low = mid + 1;
    }
  }
}
```

Figure 6.24: A synthetic example where r-TuBound helps oRange to infer a tighter bound on the total number of loop iterations. The example is constructed partly from the `bs` benchmark (inner loop) and examples presented in [43] (outer loop).

to illustrate the theoretical capabilities of merging flow facts: When analyzing the example, r-TuBound can make use of its loop refinement capabilities, thus refining the loop bound of the outer loop to 6. For the inner loop, r-TuBound infers an over-approximated loop bound of 8. Using these loop bounds for further analysis, a `totalcount` (the maximal number of executions of the inner loop when running the program) of 48. On the

BM	Fct	r/o+C167	r/o+ARM	Improvement
bs	main	920	815	reduces to 88.5% of r-TuBounds original WCET on ARM
	b_s	19140	775	65.5% of r-TB WCET on C167, 65.6% of r-TB WCET on ARM
minver	main	167760	140920	unbound for oRange/Otawa
	minver	910640	98905	unbound for oRange/Otawa

Table 6.8: WCET analysis using merged FFX files. Functions that did not change compared to the last table are omitted. **Improvements** denote the improvements in the WCET estimate of the tool configuration that performs better compared to the WCET of the original toolchain.

other hand, oRange would calculate a loop bound of 50 for the outer loop but find a tighter loop bound of 4 for the inner loop. Thus, the `totalcount` inferred is 200. In both cases, the `totalcount` is an over-approximation of the actual `totalcount`. Merging the flow facts allows to infer a safe and tighter `totalcount`: using r-TuBounds loop bound for the outer loop together with oRanges loop bound for the inner loop, results in a `totalcount` of 24!

Symbolic Execution Experiments with SmacC

We analyzed a *memcopy* and a *stringcopy* implementation for bounded runtime-and memory-safety (with bounded array-size 50, respectively 40), verified the functional correctness of a *palindrome check* and checked equality of two *power-of-3 implementations*. Path-wise verification of the *memcopy* implementation up to bound 50 takes approximately two hours. Functional correctness for the *palindrome check* (bounded by word length 16) exhibits high run-times (4.5h), and complete equality checking of two *power-of-3 implementations* (with 32bit int) times out (10h). Varying the bound of the input problems and dumping a conjunction of the verification conditions thus allows to generate SMT benchmarks with varying runtime.

We also integrated the memory-model of SmacC in r-TuBound and extended verification conditions to express arithmetic properties about conditional updates to the loop counter. This allows us to compute loop bounds in cases where the loop bound computation step of r-TuBound would fail. For example, the loop counter i in Figure 6.20(a) is conditionally updated, therefore no safe loop bound can be computed initially. Verifying that the conditional update can never decrease the loop counter allows us to use the constant increment in the loop header to compute a safe over-approximation. For the conditional update $i' = upd(i)$, e.g. `i = i + 1` in Figure6.20(a), (a:7), we verify that executing it can only increase the loop counter for the next iteration i', i.e. $i' < i$ must be unsatisfiable for arbitrary values of i, as for example in Figure 6.20(a) where a loop bound of 4 can be computed using the update `i++` (a:5) in the loop header.

Figure 6.20(a) illustrates another usage of SmacC for loop bound detection. Here,

Benchmark	**Bound**	**Boolector**	**SmacC**	**CBMC**
memcpy.c, array size 30	30	287s	1496s	0.25s
memcpy.c, array size 40	40	565s	5595s	0.33s
memcpy.c, array size 50	50	1114s	7350s	0.34s
palindrome check, n 11	11	639s	3718s	0.18s
palindrome check, n 15	15	1614s	13406s	0.22s
palindrome check, n 16	16	3344s	16220s	0.26s
strcpy array, n 20	20	231s	timeout	0.11s
strcpy array, n 30	30	1430s	timeout	0.15
strcpy array, n 40	40	7684s	timeout	0.20s
power 3 equality	3	timeout	timeout	timeout

Table 6.9: Benchmarks were run on an Intel®CPU at 2.66GHz with 2GB main memory. Time was measured using the UNIX *time* command. The table compares Boolector stand-alone version to library usage in SmacC and to CBMC.

SmacC is called with an initial loop bound. If it reports that the negation of the loop condition is satisfiable along a path, the bound is increased. Upon termination, no execution of the program exhibits a higher loop bound. The loop counter i in Figure 6.20(c) is reset in iteration 5 (c:5), therefore the loop is executed 4 more times. SmacC infers the exact loop bound 9, while a WCET analysis using the model checker CBMC [23] without SmacC does not terminate and keeps unwinding the loop.

The following C files and algorithms were transformed to a BTOR representation, and can be used as benchmarks, timing results are presented in Tab. 1.

- Memcopy: A simple memcpy implementation, copying memory from the source buffer to the destination buffer. Assert that destination buffer contains the same elements as the source buffer after copying.
- Palindrome: implements algorithm to check if a string is a palindrome. If the algorithm concludes that a string is a palindrome, assert that the string fulfills palindrome properties.
- Stringcopy: Similar to memcpy but omitting the third parameter, the number of bytes that must be copied. The loop terminates if null character is read in source buffer which is then copied to the target buffer.
- Power of 3 equality: Compares if a method to compute n^3 using a loop always yields the same result as a method without a loop.

6.4 r-TuBound and the WCET Tool Challenge

This section reports on our results obtained by participating with r-TuBound in the WCET Tool Challenge 2011. From this tool challenge competition, we gained valuable insight for the further development of r-TuBound, and we also hit typical problems in WCET analysis: the Tool Challenge 2011 offered two categories to perform in, a simple architecture (ARM) and the complex architecture (PowerPC). At the time of the challenge, r-TuBound was not able to infer WCET bounds for any of the architectures. Nevertheless, we participated in the Challenge, analyzing the software for the supported C167 platform. This flaw was an important inspiration to implement FFX support, as it allows to use the r-TuBound high-level analyzers but acquire WCET bounds for all back-end supported platforms. Other issues in the challenge included that in some cases it was not possible to annotate the input constraints, because there is no support for them in r-TuBound. For example, r-TuBound does neither support path annotations specifying "the first run" (or in general the xth run), or constraints that specify that "function f is executed once before g". Additionally, the interval analysis does not support arbitrary user supplied value annotations. Some of the input constraints we could, nevertheless, annotate manually. For the cases where the input constraints could not be annotated fully, we reported the worst-case result. Therefore, for example, when the WCET of "the first run" of a function is asked, we calculate the WCET of the function and use it as result. If there are constrained inputs that we cannot model, we again compute the (general) WCET of this function and report it as an over-approximation of the WCET of the run in question.

Another difficulty stems from the supplied assembler code: we cannot perform WCET calculation for the assembler code, because we do not support the target architecture. Therefore we could not, for example, find out the WCET of interrupt routine `__vector_10`. One important feature r-TuBound is currently missing is floating point support: interval analysis does not consider float values, those are used, for example, in parts of the PapaBench inputs. The upper loop bound problems in PapaBench all involved floats, which we don't handle in our interval analysis, even though basically the loops could be bound by our loop analyzers.

We gained valuable experience from the benchmarking part of the challenge, that took place at Daimler Germany. Even though r-TuBound does not entirely support the target architecture of these benchmarks, our managed to infer flow information from some of the sources. This experiments performed at Daimler showed the need for a shippable binary version of r-TuBound, a task we plan to address as a future work.

CHAPTER 7

Related Work

The WCET analysis of embedded systems is a challenging and actively studied research field. In this section we overview only those approaches that are directly related to the the research directions carried out within this thesis.

Proving WCET bounds Precise

To the best of our knowledge, our WCET Squeezing method is the first approach which refines and improves the WCET bound of a program after an initial WCET analysis in an anytime-manner. WCET Squeezing makes use of both symbolic execution and WCET analysis, and is different from state-of-the-art approaches for the following reasons.

Symbolic execution originally was used for test-case generation but recently found more and more applications in program verification or automatic exploit generation for applications. Symbolic execution has been successfully applied for test-case generation for programs in [18] and [19], where, for example, the work of [19] describes a symbolic execution engine for bug-hunting. In these approaches the focus lies on speeding up symbolic execution by optimization and caching of queries discharged to the underlying solver or tracking only currently relevant information. In contrast, symbolic execution in WCET Squeezing offers little optimizations and tracks as much information as possible. In general, precise information needs high run-times, a problem we counteract by applying selective symbolic execution only.

Static WCET analysis is performed using timing analysis tools which need flow- fact information about the program under analysis. Such information may be given manually by the developer or inferred automatically by a flow fact analyzer and includes information about execution frequencies of blocks and loop bounds for program loops. Modern static WCET analyzers, see e.g. [44, 5, 32], typically rely on the IPET technique [58] to calculate a WCET estimate. IPET usually over-estimates the WCET bound, as the constructed ILP problem encodes numerous spurious program executions that are infeasible in the program. WCET Squeezing can be used in addition to IPET-based WCET analysis, overcoming this deficiency.

Our WCET Squeezing approach has similarities to the counterexample guided abstraction refinement (CeGAR) of [24]. It is an anytime-algorithm [14] that allows to refine or prove precise a WCET estimate after an initial IPET-based WCET analysis. A related idea is presented in [6], where an ILP encoding of the program is used to check whether partial solutions of a specific size to the ILP problem yield infeasible program paths. Feasibility of solutions is checked using model checking. In contrast to our approach that applies path-wise symbolic execution, [6] encodes the feasibility check as program assertions in the original program, thereby losing the advantage of path-local reasoning.

In general, inferring precise program information comes with high computational costs, a problem which we avoid by using selective symbolic execution: WCET Squeezing applies symbolic execution only when information about the program is too coarse or when other analysis methods fail. A similar idea is presented in [15] where symbolic execution is used to refine spurious def-use results via a path feasibility analysis. In [15] branching decisions are determined at compile time and used to identify and remove infeasible paths. This method can be seen as a light-weight on-demand symbolic execution of conditional nodes, whereas symbolic execution in WCET Squeezing always executes single paths.

Symbolic execution for WCET analysis is also used in [38] and avoids some typical pitfalls of symbolic execution. For example, loops are not unfolded and hence multiple executions of the same block are omitted. We note that [38] analyses each program block whereas our selective symbolic execution approach only analyses relevant program blocks and paths.

Measurement-based timing analysis techniques can be seen complementary to static WCET analysis tools. Measurement-based tools require test inputs that cover a sufficient portion of the program executions to results with high confidence. In [71] the authors present a tool that systematically generates test data using techniques like heuristics and model checking. In contrast to this technique that generates test-cases for arbitrary executions of the program the approach we propose in WCET Squeezing generates test-cases that lead to executions along the WCET candidate path(s).

A different approach to WCET analysis [22] relies on segment- and state-based abstract interpretation [26], and is denoted quantitative abstraction refinement. The state-based approach has some similarities to counter-example encoding in WCET Squeezing, which makes control-flow decisions explicit in loop iterations. Related, though conceptually different is the approach of [16]. Here, irrelevant program parts are identified via the criticality of basic blocks, which denotes the relation between the longest path through the basic block and the WCET bound of the program. Eliminating the irrelevant parts from the program allows usage of a more precise but computationally more expensive WCET analyzer that then might come up with a more precise WCET bound. Another related approach is the abstract execution framework of [33], where context-sensitive abstract interpretation is applied to analyse loop iterations and function calls in separation. Instead of applying a fix-point analysis, abstract operations on abstract values are used in [33], where an abstract value can, e.g. , be represented as an interval. When abstract values prevent the evaluation of a conditional, both branches need to be followed. Abstract states can be merged at join points to prevent the path explosion problem. As a result, a single abstract execution can represent execution of multiple concrete paths. This is not the case in the traditional use of symbolic execution. Compared to WCET Squeezing, abstract execution in [33] analyses the entire program, whereas our method applies more costly symbolic execution only to relevant parts of the program.

In [68] the authors construct parameterized formulas to model the WCET of program functions or loops. To this end, closed form formulas that depend only on the number of loop iterations are generated. Unlike our approach, the inferred closed forms are evaluated at run-time to improve the performance of the system under analysis, by allowing optimized scheduling decisions and task selections.

Loop Bound Computation

Our work is closely related to the invariant generation and loop bound computation of program loops. Loop invariants describe loop properties that are valid at any, and thus also at the last loop iteration. Invariant generation techniques therefore can be used to infer bounds on program resources, such as time and memory – see e.g. [30, 31, 3, 12, 35].

WCET analysis is usually two-tiered. A low-level analysis estimates the execution times of program instructions on the underlying hardware and computes a concrete time value of the WCET. On the other hand, a high-level analysis is, in general, platform-independent and is concerned, for example, with loop bound computation. In [57] iteration bounds are obtained using data flow analysis and interval based abstract interpretation. However, state-of-the-art WCET analysis tools, including [32, 49, 57], compute loop bounds automatically only for loops with relatively simple flow and arithmetic [36]. For more complex loops iteration bounds are supplied manually in the form of auxiliary program annotations. procedures manually. Unlike these approaches, we require no user guidance but automatically infer iteration bounds for special classes of loops with non-trivial arithmetic and flow.

In [48] a framework for parametric WCET analysis is introduced, and symbolic expressions as iteration bounds are derived. Instantiating the symbolic expressions with specific inputs yields then the WCET of programs. The approach presented in [49] automatically identifies induction variables and recurrence relations of loops using abstract interpretation [4]. Recurrence relations are further solved using precomputed closed form templates, and iteration bounds are hence derived.

The work described in [30] instruments loops with various counters at different program locations. Then an abstract interpretation based linear invariant generation method is used to derive linear bounds over each counter variable. Bounds on counters are composed using a proof-rule-based algorithm, and non-linear disjunctive bounds of multi-path loops are finally inferred. The approach is further extended in citerbp to derive more complex loop bounds. For doing so, disjunctive invariants are inferred using abstract interpretation and flow refinement. Next, proof-rules using max, sum, and product operations on bound patterns are deployed in conjunction with SMT reasoning in the theory of linear arithmetic and arrays. As a result, non-linear symbolic bounds of multi-path loops are obtained. Abstract interpretation based invariant generation is also used in [3] in conjunction with so-called cost relations. Cost relations extend recurrence relations and can express recurrence relations with non-deterministic behavior which arise from multi-path loops. Iteration bounds of loops are inferred by constructing evaluation trees of cost relations and computing bounds on the height of the trees. For doing so, linear invariants and ranking functions for each tree node are inferred. Unlike the aforementioned techniques, we do not use abstract interpretation but deploy a recurrence solving approach to generate bounds on simple loops. Contrarily to [30, 31, 3], our loop bound computation method is limited to multi-path loops that can be translated into simple loops by SMT queries over arithmetic.

Recurrence solving is also used in [12, 35]. The work presented in [35] derives loop bounds by solving arbitrary C-finite recurrences and deploying quantifier elimination over integers and real closed fields. To this end, [35] uses some algebraic algorithms as black-boxes built upon the computer algebra system (CAS) Mathematica [70]. Contrarily to [35], we only solve C-finite recurrences of order 1, but, unlike [35], we do not rely on computer algebra systems and handle more complex multi-path loops. Symbolic loop bounds in [12] are inferred over arbitrarily nested loops with polynomial dependencies among loop iteration variables. To this end, C-finite and hypergeometric recurrence solving is used. Unlike [12], we only handle C-finite recurrences of order 1. Contrarily to [12], we however design flow refinement techniques to make our approach scalable to the WCET analysis of programs.

In [9] the authors suggest algebraic techniques to construct a symbolic formula that characterizes the WCET of a function. For doing so, powerful computer algebra systems (CAS) are used to construct and simplify symbolic formulas. The described approach could be extended by symbolic summation in order to derive loop bounds and handle nested loops. The work presented in [9] relies on annotated programs and deploys a CAS, whereas we use a pattern-based recurrence solving approach to symbolic summation and deploy pre-computed closed form solutions. Unlike these method, we reduce the computationally expensive techniques of a CAS, by deploying pattern-based recurrence solving for computing loop bounds.

Common Annotation Format

The majority of existing WCET tools come with their own annotation language to carry flow facts. For example, the SWEET tool [32] uses a flow information format called "context-sensitive valid-at-entry-of flow facts", the aiT tool uses the "aiS" format [2], and the Bound-T tool uses the "Bound-T Assertion Language" [67].

SWEET offers "value annotations" to specify constraints on possible input values of a program, and allows to annotate flow facts in both local and global contexts. Additionally, flow information can be marked to hold only in certain function call-string contexts [32]. On the other hand, aiT understands both source level annotations as well as binary annotations, using the aiS format. It uses program points to identify source locations as addresses, routine and file names, and it combines so-called atoms to construct more complex flow facts. Bound-T uses source-code mark positions and carries additional assertions in a text file. Assertions are statements about the program that bound certain aspects of the behaviour, e.g. loop bounds or information about the stack usage. Bound-T assertions are then valid flow information at certain program points, identified by the markers.

Basic annotations, e.g. loop bounds, are supported by all aforementioned annotation languages. Nevertheless, the formats of deployed annotations are quite different. In what follows, we argue that our intermediate flow fact format FFX allows to specify important flow information in most WCET analyzers. We therefore believe, that even if some flow facts from different tools cannot yet be formulated in FFX, our work already supports many properties of state-of-the-art annotation languages. Thus, FFX can be used to make fair comparisons in order to gain comparability between the tools. Further, it allows to exchange and interchange information and back-ends between tools, resulting in better WCET estimates when using the tools in collaboration.

The approach of [54] argues that source based annotations are portable, easy to use and flexible to integrate in existing toolchains. FFX follows this line of argument by focusing especially on the flexible integration within existing toolchains.

The strength of FFX has already been shown to some extent in [5], as it is the internal format of the oRange/Otawa tool. oRange/Otawa benefits from the fact that all components of the toolchain use a common format to carry analysis information.

Additionally, in the Merasa[1] project, the FFX format was successfully used in order to compare results of OTAWA [5] with RapiTime[2]. FFX thus already proved to be a suitable annotation language as the internal format of the OTAWA/oRange WCET toolchain [5]. Our FFX support in r-TuBound/CalcWCET167 shows that FFX is also suitable for intermediate flow fact representation. Moreover, our FFX format extends [5] and supports most of the ingredients proposed in [41]. Unlike [41], where only the theoretical benefits of a common annotation language are discussed, in work we propose FFX as a common annotation language. We address both theoretical and practical details of FFX as a common annotation language and present initial experiments with FFX. As the format is expandable, we believe that all the required ingredients of an annotation language can be supported without breaking compatibility with tools not supporting new information.

WCET Tools

We now compare our implementation in r-TuBound to other state-of-the-art WCET toolchains.

The SWEET tool [33] applies a technique called abstract execution which is basically a form of symbolic execution based on abstract interpretation (value analysis). It uses abstract values for program variables and executes the program in the abstract domain,

[1]`http://www.merasa.org`
[2]`http://www.rapitasystems.com/`

using abstract operators. This technique allows for deriving context sensitive information about the program, e.g. by using execution counters for blocks, incremented when entering the block. The results of the value analysis for these counters derive execution frequencies of blocks – the ones located at loop headers are used to derive loop bounds and also allow to derive lower bounds on the number of iterations. The abstract execution approach taken by SWEET seems comparable to the symbolic execution approach pursued with r-TuBound. Whereas r-TuBound relies on symbolic execution using SMT over bitvectors and arrays, SWEET relies on abstract interpretation based symbolic execution and is especially powerful for deriving flow facts (execution behaviour of the program). Compared to SWEET, r-TuBound itself is not able to handle loops that involve floating point values (there exists a model-checking extension, though). Another restriction is that r-TuBound cannot handle loops where the loop condition relates two variables that are both modified in the loop body. Additionally, the pattern based approach of r-TuBound limits the types of loops that r-TuBound can analyze, whereas an approach based on abstract operators seems more versatile. On the other hand, the over-approximation techniques might allow for analyzing loops where abstract interpretation based techniques fail.

The OTAWA [5] uses the companion tool oRange [49] to gather flow facts, including loop bounds. The approach pursued by oRange is based on abstract interpretation and execution flow analysis [59]. It works as follows: an annotated context tree consisting of loop and function call nodes is constructed. For each loop node, two expressions are constructed, one, representing the total number of loop iterations, and one that counts the maximum number of loop iterations for each startup. An extension to Ammarguellats method [4] for the recognition of induction variables and recurrence relations is applied to evaluate abstract stores, which are finally evaluated to compute the above mentioned total and maximum number of loop iterations. The operators that oRange can handle are very similar to the operators r-TuBound can handle. The approach can deal with certain loops that contain boolean connectives in the condition and it is able to handle nested loops.

When comparing the symbolic execution engine SmacC of r-TuBound to the CBMC [23] bounded model checker for ANSI C and C++ programs, an important difference is that SmacC performs path-wise symbolic execution. That is, instead of verifying that a property holds for the whole program, SmacC checks the property for each path through the program. This situation, often denoted as the path explosion problem of symbolic execution, on the one hand might prevent a full-fledged verification of a program due to a high number of paths, but at the same time it comes with the advantage of precise path-wise analysis information, that, as we have shown, is very relevant for our approaches to WCET analysis.

CBMC allows verifying array bounds, pointer safety, exceptions and user-specified assertions [46]. It takes as input C files and translates the program, merging function definitions from the input files. Instead of producing a binary for execution, CBMC performs symbolic simulation on the program [27]. CBMC translates refined programs to SAT instances and uses the MiniSAT SAT solver to verify properties. Unlike CBMC, our symbolic execution engine SmaC does not establish a full representation for the memory of the program and its layout, instead it uses intermediate variables when accessing variables. CBMC unwinds loops and recursive function calls and transforms the program until it only consists of `if` instructions, assignments, assertions, labels and `goto` instructions [23]. An assertion for each loop verifies that the unwinding bound [23] is large enough, otherwise the bound is increased. Then it is transformed into static single assignment form, consisting of bit-vector equations for constraints and verification conditions. The conjunction of the constraints and the negation of the property is checked for satisfiability. If the conjunction is satisfiable, the property is violated.

CHAPTER 8

Conclusion & Perspectives

The present thesis addresses new and uncoventional problems in the WCET analysis of programs.

(1) Precision. We first describe an effective procedure that is able to prove whether the WCET bound inferred by a static WCET analyzer is precise, and if not, to tighten the WCET bound until eventually the bound is proved to be precise. Our procedure is called *WCET Squeezing*. It works by iteratively refining the program model by excluding infeasible program paths from the model until the time bound delivered by a re-application of the WCET bound calculation to the refined model can be proved precise.

WCET Squeezing combines symbolic execution together with the implicit path enumeration technique, and uses SMT reasoning together with integer linear programming techniques for computing precise WCET bounds. Symbolic execution offers a precise framework for program analysis and tracks complex program properties by analyzing single program paths in isolation. This path-wise program exploration of symbolic execution is, however, computationally expensive, which often prevents full symbolic analysis of larger applications: the number of paths in a program increases exponentially with the number of conditionals, a situation denoted as the path explosion problem. Therefore, for applying symbolic execution WCET Squeezing, we used WCET analysis as guidance for symbolic execution in order to avoid full symbolic coverage of the program. The resulting symbolic execution framework is thus selective in the sense that it explores only those program paths that (possibly) exhibit the WCET bound of the program. *Selective symbolic execution* in WCET Squeezing makes a compromise between the precision and computational cost of symbolic execution, with the ultimate cost of making WCET Squeezing practically efficient.

Such a compromise between precision and efficiency is also present in other approaches to the WCET analysis of programs: a successful WCET analyzer usually requires a balance between the speed and the precision of the deployed analysis. Precision of the analysis is gained by applying powerful program analysis techniques that gather information about the program and pass it to further analysis and computation steps.

Precision of the analysis yields tight WCET bounds, however, at the cost of high computational efforts; this sometimes prevents the analysis to terminate within a given time-limit. Therefore, precision of the analysis is often traded for its speed, that is faster analysis with possibly imprecise WCET bounds are in general instead of a precise but inefficient analysis. Automated methods for refining imprecise WCET results into tight WCET bounds are therefore crucial in making WCET analysis practically useful.

In this thesis we argue that combining selective symbolic execution with traditional WCET analysis in WCET Squeezing yields such automated method for deriving precise WCET bounds. We show that, when using selective symbolic execution in WCET analysis, a partial symbolic coverage of the program is sufficient to tighten and, eventually, prove precise the WCET bound of the program. Selective symbolic execution comes thus with the advantage of avoiding the path explosion problem of traditional symbolic execution, as it applies costly symbolic execution only for those program parts that influence the WCET bouds.

(2) Performance. Our results show that efficiency of WCET Squeezing crucially depends on the quality of the initial WCET bound derived by a static WCET analyzer. To make WCET Squeezing as proof procedure highly performant, it is thus key to empower the used WCET analyzer as much as possible. Therefore, our thesis also addressed the problem of improving the strength WCET analyzers. To this end, we described an automated approach for *computing tight bound, called loop bounds, on the number of program loop itarations.* Our method relies on symbolic computation techniques, in particular on recurrence solving approaches, and combines them with new ways of program flow refinement and SMT reasoning. We demonstrate that loop bounds derived by our work improve quality of WCET bound, and hence the performance of WCET Squeezing in general.

Further, we can use the selective symbolic execution for deriving loop bounds for programs where using only symbolic computation methods fails. For doing so, we consider the program loop and use only variable declarations relevant this program loop. The such *reduced* program is next symbolically executed, by taking increasingly finite loop unrollings; the initial loop bound is hence set to 0. If symbolic execution reports that the negation of the loop condition is unsatisfiable on the unrolled loop path, the loop bound is increased until the negation of the loop condition is satisfiable. Upon termination, no execution of the program exhibits a higher loop bound. Such a use of symbolic execution in loop bound computation is especially useful for bit-precise reasoning over the program.

(3) Portability. In this thesis we also present an approach to make our WCET Squeezing method portable for immediate usage in arbitrary WCET analyzer toolchains. This is achieved by making tools of different WCET analyzers interoperable by means of introducing a *flexible and extensible intermediate language providing a common interface for exchanging analysis information.* This language is called Flow Facts in XML, shortly referred to as *FFX*. An immediate benefit of FFX is, for example, quality assurance for WCET analysis, FFX can be used to test and validate new analysis techniques and tools against using a common annotation language.

(4) Practice. Last, but not least, the thesis reports on the implementation and experimental evaluation of our results. To this end, we imlemented the overall framework of the thesis in the r-TuBound WCET tool change and tested r-TuBound on a large number of examples coming both from academia and industry. Our experiments of using r-TuBound for WCET Squeezing, selective symbolic execution, loop bound computation, and comparing WCET analyzers based on the FFX format highlight the practical benefits of our work. It is not unusual, for example, that in a single iteration of WCET Squeezing, the precision of WCET bound is improved by 3%.

Conclusions and Further Work. The techniques developed in this thesis emphasize the advantage of using symbolic methods, including symbolic execution, symbolic computation, and theorem proving, in the WCET analysis of programs. We believe that our results empower the strength of existing WCET analysis techniques, for the following reasons.

- We use symbolic execution in WCET Squeezing on reduced program fragments, and analyze programs which could not be analyzed so far by other methods.
- We deploy symbolic execution and symbolic computation to compute loop bounds that are further used in improving the efficiency of WCET Squeezing. Some of the loop bounds derived by our approach could not be obtained by previous methods.
- Based on the implicit path enumeration technique, we use the result of an initial WCET analyzer and apply symbolic execution in WCET Squeezing to prove precise initial WCET estimates. To the best of our knowledge, WCET Squeezing is the first approach able to prove precision of WCET bounds inferred by a static WCET analyzer in a fully automated way.
- WCET Squeezing can be used as a general proof procedure in WCET analysis, by relying on the FFX format to exchange information between WCET analyzers.
- Our overall approach is implemeted in the r-TuBound WCET toolchain. When applying the selective symbolic execution framework of r-TuBound on a large number of benchmarks, our results show that selective symbolic execution proves, or if necessary tightens, the precision of WCET bounds by keeping low the computational overhead for symbolic execution.

We conclude, that symbolic methods, if applied efficiently as proposed in this thesis, are a useful addition to program analysis techniques and particularly suited for WCET analysis of programs. We therefore believe that extending our results with more sophisticated symbolic methods is challenging challenging task to be further investigated. In particular, we are interested in design new methods deriving precise WCET bounds for programs with complex control flow and data structures, including floating points, arrays, and pointers. Additionally, improving the performance of selective symbolic execution by symbolic testing approach is another topic worth to be studied.

Bibliography

[1] SciMark2 C Benchmark Suite. `math.nist.gov/scimark2/index.html`.

[2] AbsInt Angewandte Informatik GmbH. aiT. `http://www.absint.com`, 2007.

[3] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *J. Automated Reasoning*, 46(2):161–203, 2011.

[4] Z. Ammarguellat and W. L. Harrison, III. Automatic Recognition of Induction Variables and Recurrence Relations by Abstract Interpretation. In *Proc. of PLDI*, pages 283–295, 1990.

[5] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: an Open Toolbox for Adaptive WCET Analysis. In *Proc. of IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, Austria, October 2010. Springer.

[6] H. J. Bang, T. H. Kim, and S. D. Cha. An Iterative Refinement Framework for Tighter Worst-Case Execution Time Calculation. In *Proc.of ISORC*, pages 365–372, 2007.

[7] C. Barrett, A. Stump, C. Tinelli, S. Boehme, D. Cok, D. Deharbe, B. Dutertre, P. Fontaine, V. Ganesh, A. Griggio, J. Grundy, P. Jackson, A. Oliveras, S. Krstić, M. Moskal, L. D. Moura, R. Sebastiani, T. D. Cok, and J. Hoenicke. The SMT-LIB Standard: Version 2.0. Technical report, 2010.

[8] M. Berkelaar, K. Eikland, and P. Notebaert. lp_solve. Software, 2004. `http://lpsolve.sourceforge.net/5.5/`.

[9] G. Bernat and A. Burns. An Approach To Symbolic Worst-Case Execution Time Analysis. In *In 25th IFAC Workshop on Real-Time Programming*, 2000.

[10] A. Biere, J. Knoop, L. Kovács, and J. Zwirchmayr. SmacC: A Retargetable Symbolic Execution Engine. In *Proc. of ATVA*, 2013. To appear.

[11] N. Bjørner, L. de Moura, and N. Tillmann. Satisfiability Modulo Bit-precise Theories for Program Exploration. In *Proc. of CFV*, 2008.

[12] R. Blanc, T. Henzinger, T. Hottelier, and L. Kovács. ABC: Algebraic Bound Computation for Loops. In *Proc. of LPAR-16*, pages 103–118, 2010.

[13] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In T. Mogensen, D. Schmidt, and I. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, LNCS 2566, pages 85–108. Springer-Verlag, Oct. 2002.

[14] M. S. Boddy. Anytime Problem Solving Using Dynamic Programming. In *Proc. of AAAI*, pages 738–743, 1991.

[15] R. Bodík, R. Gupta, and M. L. Soffa. Refining Data Flow Information Using Infeasible Paths. *SIGSOFT Softw. Eng. Notes*, 22(6):361–377, Nov. 1997.

[16] F. Brandner and A. Jordan. Refinement of Worst-Case Execution Time Bounds by Graph Pruning. 2013. under submission.

[17] R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Lecture Notes in Computer Science (LNCS)*, volume 5505. Springer, 2009. TACAS'09.

[18] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, pages 209–224, 2008.

[19] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *Proc. of CCS*, pages 322–335, 2006.

[20] C. Cadar and K. Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, 2013.

[21] H. Cassé and M. De Michiel. FFX: Flow Facts in XML. Rapport de recherche IRIT/RR–2012-5–FR, IRIT, Université Paul Sabatier, Toulouse, April 2012.

[22] P. Cerny, T. Henzinger, and A. Radhakrishna. Quantitative Abstraction Refinement. In *Proc. of POPL*, pages 115–128, 2013.

[23] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *Proc. of TACAS*, pages 168–176. Carnegie Mellon University, 2004.

[24] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proc. of CAV*, pages 154–169, 2000.

[25] M. E. Conway. Proposal for an UNCOL. *Commun. ACM*, 1(10):5–8, Oct. 1958.

[26] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *In Proc.of POPL*, pages 238–252, 1977.

[27] CProver. The CProver User Manual. http://www.cprover.org/cprover-manual/cbmc.shtml.

[28] G. Everest, A. van der Poorten, I. Shparlinski, and T. Ward. *Recurrence Sequences*, volume 104 of *Mathematical Surveys and Monographs*. American Mathematical Society, 2003.

[29] C. Fraser and D. Hanson. *lcc, A Retargetable C Compiler for ANSI C*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, 1995.

[30] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. of POPL*, pages 127–139, 2009.

[31] S. Gulwani and F. Zuleger. The Reachability-Bound Problem. In *Proc. of PLDI*, pages 292–304, 2010.

[32] J. Gustafsson. SWEET Manual. http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/out/webhelp/index_frames.html.

[33] J. Gustafsson. SWEET: SWEdish Execution Time tool. http://www.mrtc.mdh.se/projects/wcet/sweet.html, 2001.

[34] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In *Proc. of WCET*, pages 136–146, 2010.

[35] T. Henzinger, T. Hottelier, and L. Kovács. Valigator: A Verification Tool with Bound and Invariant Generation. In *Proc. of LPAR-15*, pages 333–342, 2008.

[36] N. Holsti, J. Gustafsson, G. Bernat, C. Ballabriga, A. Bonenfant, R. Bourgade, H. Cassé, D. Cordes, A. Kadlec, R. Kirner, J. Knoop, P. Lokuciejewski, N. Merriam, M. de Michiel, A. Prantl, B. Rieder, C. Rochange, P. Sainrat, and M. Schordan. WCET 2008 - Report from the Tool Challenge 2008. In *Proc. of WCET*, 2008.

[37] M. Kauers. SumCracker: A Package for Manipulating Symbolic Sums and Related Objects. *J. of Symbolic Computation*, 41(9):1039–1057, 2006.

[38] D. Kebbal and P. Sainrat. Combining Symbolic Execution and Path Enumeration in Worst-Case Execution Time Analysis. In *Proc. of WCET*, 2006.

[39] R. Kirner. User's Manual – WCET-Analysis Framework based on WCETC. http://www.vmars.tuwien.ac.at/~raimund/calc_wcet/, 2001.

[40] R. Kirner. The WCET Analysis Tool CalcWcet167. In T. Margaria and B. Steffen, editors, *ISoLA (2)*, volume 7610 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2012.

[41] R. Kirner, A. Kadlec, A. Prantl, M. Schordan, and J. Knoop. Towards a Common WCET Annotation Language: Essential Ingredients. In *WCET*, 2008.

[42] R. Kirner, J. Knoop, A. Prantl, M. Schordan, and A. Kadlec. Beyond Loop Bounds: Comparing Annotation Languages for Worst-case Execution Time Analysis. *Software and System Modeling*, 10(3):411–437, 2011.

[43] J. Knoop, L. Kovács, and J. Zwirchmayr. Symbolic Loop Bound Computation for WCET Analysis. In *Proc. of PSI*, pages 116 – 126, 2011.

[44] J. Knoop, L. Kovaćs, and J. Zwirchmayr. r-TuBound: Loop Bounds for WCET Analysis. In *Proc. of LPAR*, volume 7180 of *LNCS*, pages 435 – 444, Mérida, Venezuela, 2012.

[45] L. Kovács. Cutting-Edge Timing Analysis Techniques for Safety-Critical Real-Time Systems (CeTAT). Project application at Vienna University of Technology.

[46] D. Kroening. Bounded Model Checking for ANSI-C. `http://www.cprover.org/cbmc/`.

[47] L. L. N. Laboratory. The Rose Compiler: an Open Source Compiler Infrastructure. http://www.rosecompiler.org/.

[48] B. Lisper. Fully Automatic, Parametric Worst-Case Execution Time Analysis. In *Proc. of WCET*, pages 99–102, 2003.

[49] M. D. Michiel, A. Bonenfant, H. Cassé, and P. Sainrat. Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation. In *Proc. 14^{th} IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, (RTCSA 2008)*, Taiwan, 2008.

[50] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[51] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[52] P. Paule and M. Schorn. A Mathematica Version of Zeilberger's Algorithm for Proving Binomial Coefficient Identities. *J. of Symbolic Computation*, 20(5-6):673–698, 1995.

[53] A. Prantl. The Termite Library. `http://www.complang.tuwien.ac.at/adrian/termite/Manual/`.

[54] A. Prantl. *High-level Compiler Support for Timing-Analysis.* PhD thesis, TU Vienna, 2010.

[55] A. Prantl, J. Knoop, R. Kirner, A. Kadlec, and M. Schordan. From Trusted Annotations to Verified Knowledge. In *Proc. of WCET*, pages 39–49, 2009.

[56] A. Prantl, J. Knoop, M. Schordan, and M. Triska. Constraint Solving For High-level WCET Analysis. *CoRR*, abs/0903.2251, 2009.

[57] A. Prantl, M. Schordan, and J. Knoop. TuBound - A Conceptually New Tool for Worst-Case Execution Time Analysis. In *In Proc.of WCET'08*, volume 8, 2008.

[58] P. P. Puschner and S. A. V. Computing Maximum Task Execution Times – A Graph-Based Approach. *Real-Time Systems*, 13(1):67–91, July 1997.

[59] R. von Hanxleden et al. The WCET Tool Challenge 2011: Report. In *Proc. of WCET*, 2011. under journal submission.

[60] Rapita Systems Ltd. RapiTime Explained – White Paper. `http://www.rapitasystems.com/downloads/rapitime_explained_white_paper`.

[61] C. Schneider. Symbolic Summation with Single-Nested Sum Extensions. In *Proc. of ISSAC*, pages 282–289, 2004.

[62] M. Schordan, G. Barany, A. Prantl, and V. Pavlu. SATIrE - The Static Analysis Tool Integration Engine. `http://www.complang.tuwien.ac.at/satire/`.

[63] S. S. Skiena. *The Algorithm Design Manual.* Springer Publishing Company, Incorporated, 2nd edition, 2008.

[64] J. Souyris, E. L. Pavec, G. Himbert, V. Jégu, and G. Borios. Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. In *Proc. of WCET*, pages 21–24, 2005.

[65] A. S. Tannenbaum. *Modern Operating Systems, 3rd Edition.* Pearson, Prentice Hall, Upper Saddle River, New Jersey 07458, 2007.

[66] L. Thomason. TinyXML. `http://sourceforge.net/projects/tinyxml/`.

[67] Tidorum Ltd. Bound-T. `http://www.tidorum.fi/bound-t`, 2005.

[68] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric Timing Analysis. In *Proc. of LCTES*, pages 88–93, 2001.

[69] R. von Hanxleden, N. Holsti, B. Lisper, E. Ploedereder, R. Wilhelm, A. Bonenfant, H. Casse, S. Bünte, W. Fellger, S. Gepperth, J. Gustafsson, B. Huber, N. Islam, D. Kästner, R. Kirner, L. Kovacs, F. Krause, M. de Michiel, M. C. Olesen, A. Prantl, W. Puffitsch, C. Rochange, M. Schoeberl, S. Wegener, M. Zolda, and J. Zwirchmayr. WCET Tool Challenge 2011: Report. In *Proc. 11th International Workshop on Worst-Case Execution Time Analysis (WCET 2011)*, 2011.

[70] S. Wolfram. *The Mathematica Book. Version 5.0.* Wolfram Media, 2003.

[71] M. Zolda and R. Kirner. Compiler Support for Measurement-based Timing Analysis. In *Proc. of WCET*, pages 62–71, 2011.

[72] J. Zwirchmayr. A Satisfiability Modulo Theories Memory-Model and Assertion Checker for C. Master's thesis, JKU Linz, Austria, 2009.