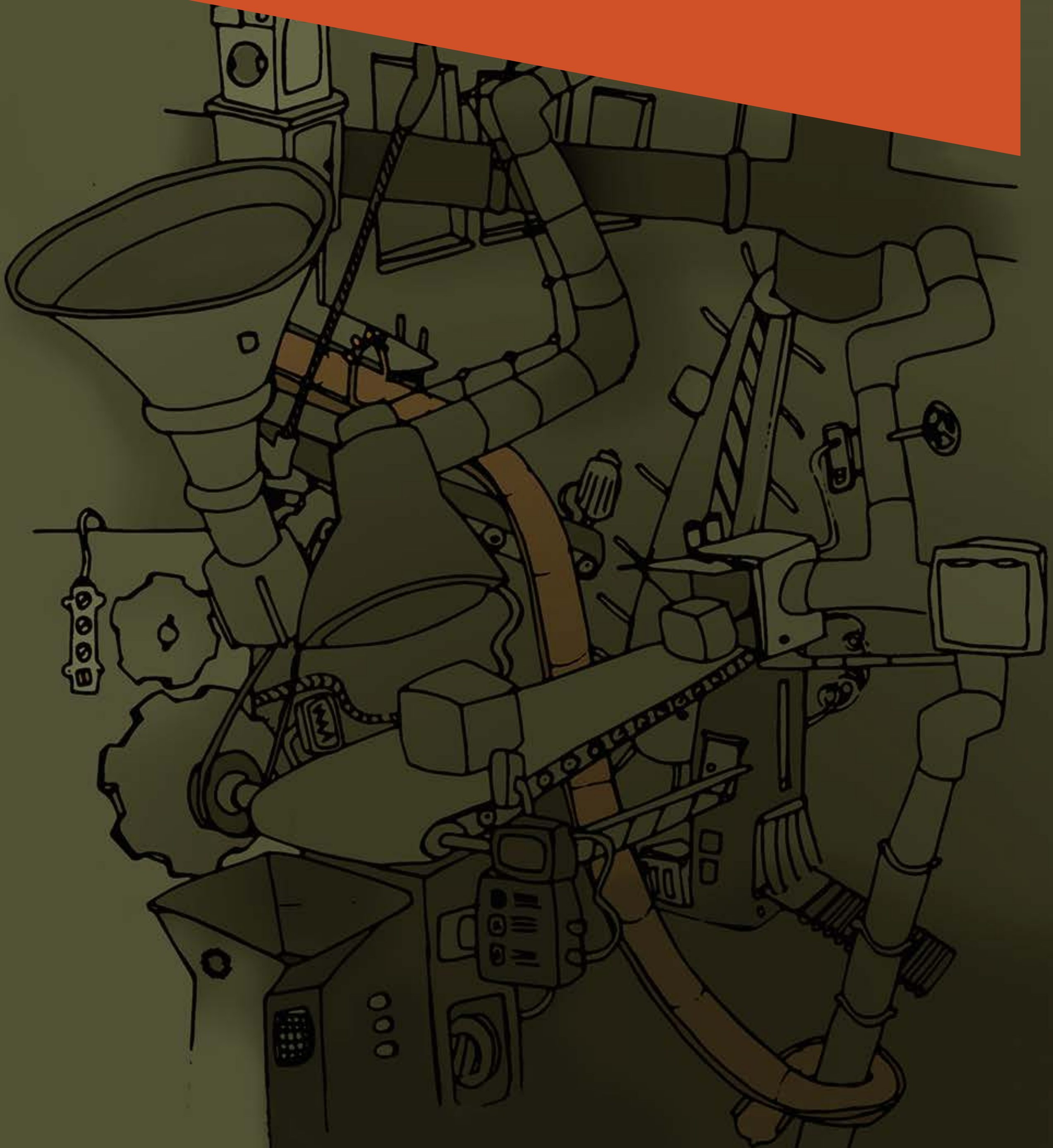
RODY W. J. KERSTEN

SOFTWARE ANALYSIS METHODS FOR RESOURCE-SENSITIVE SYSTEMS



ISBN 978-94-6259-738-9
IPA dissertation series 2015-17

Typeset using $\LaTeX\,2_\varepsilon$
Cover design: Thomas Mennen at Dark Roast Design
Printed by: Ipskamp Drukkers

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics). This research was partially supported by the IOP GenCom GoGreen project, sponsored by the Dutch Ministry of Economic Affairs. This research was partially supported by the CHARTER project, sponsored by the EU Artemis Joint Undertaking. Part of this research was conducted during a visit to the National Institute for Aerospace and the NASA Langley Research Center.

Software Analysis Methods for Resource-Sensitive Systems

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus, prof. dr. Th.L.M. Engelen,
volgens besluit van het college van decanen
in het openbaar te verdedigen op dinsdag 1 september 2015
om 14:30 uur precies

door

Rody Wilhelmus Johannes Kersten

geboren op 29 mei 1983
te Nijmegen

Promotor:

Prof. dr. Marko C.J.D. van Eekelen

Copromotor:

Dr. Sjaak Smetsers

Manuscriptcommissie:

Prof. dr. Frits W. Vaandrager
Prof. Ricardo Peña (Universidad Complutense de Madrid, Spanje)
Prof. dr. Marieke Huisman (Universiteit Twente)
Dr. Neha S. Rungta (NASA Ames Research Center, Verenigde Staten)
Dr. ir. Erik Poll

Software Analysis Methods for Resource-Sensitive Systems

Doctoral thesis

to obtain the degree of doctor
from Radboud University Nijmegen
on the authority of the Rector Magnificus, prof. dr. Th.L.M. Engelen,
according to the decision of the Council of Deans
to be defended in public on September 1, 2015
at 14:30 hours

by

Rody Wilhelmus Johannes Kersten

born on May 29, 1983
in Nijmegen, The Netherlands

Supervisor:

Prof. dr. Marko C.J.D. van Eekelen

Co-supervisor:

Dr. Sjaak Smetsers

Doctoral Thesis Committee:

Prof. dr. Frits W. Vaandrager
Prof. Ricardo Peña (Complutense University of Madrid, Spain)
Prof. dr. Marieke Huisman (University of Twente, The Netherlands)
Dr. Neha S. Rungta (NASA Ames Research Center, United States of America)
Dr. ir. Erik Poll

Acknowledgements

Before starting this thesis, I would like to thank the many people that have helped make it possible. First and foremost, my advisors Marko and Sjaak. Thank you for your valuable advice and guidance. Marko, your optimistic attitude has been instrumental in keeping me motivated and your ambition, pragmatism and enthusiasm are inspiring. Sjaak, your sharp observations and straightforward criticism were paramount in improving my research and my writings. The combination of your optimistic and sceptical personalities has made you the best possible team of advisors I could wish for.

In 2014, I did a two-month internship at NASA Langley Research Center and the National Institute for Aerospace (NIA). I was supervised by Suzette Person, who was a great advisor, as well as an amazing host. Thank you Suzette, for your helpful guidance, your valuable advice and your general kindness. My stay in Hampton was an amazing experience and I have learned a lot from you and your colleagues. During my internship at NASA Langley, we also worked closely together with Neha Rungta and Oksana Tkachuk at NASA Ames Research Center. Thank you Neha and Oksana for your insightful comments and advice. I have greatly enjoyed our collaboration and I hope we can continue this in California. Finally I would like to thank all the other friendly and interesting people I have met at NASA Langley and at NIA, who have made my stay in Hampton very enjoyable. In particular, thank you Alwyn Goodloe, for introducing me to Suzette.

To the members of my thesis committee Frits Vaandrager, Ricardo Peña, Marieke Huisman, Neha Rungta and Erik Poll, thank you for your very useful comments. Your criticism has substantially contributed to improving this thesis.

Thanks to my co-authors Bernard, Olha, Paolo, Manuel, Manu, Marc and Jascha, without whom I would not have succeeded in writing the publications on which this thesis is based. Bernard in particular has collaborated on four of the papers in this thesis and has helped me out with my research many times. Thank you Bernard for our pleasant collaboration, for being a friend and for being a lot of fun to work with. Olha supervised my master thesis back in 2009 and then collaborated with me on my first paper. Thank you Olha for helping me get a great start of my PhD. Paolo, Manuel, Manu, Marc and Jascha, thank you for your contributions and the very enjoyable collaboration.

Thanks to the partners in the GoGreen project, who I enjoyed working with. In particular, thanks to Paul Havinga and Nirvana Meratnia for leading the

project. Also thanks to all the partners in the CHARTER project, from whom I learned a lot about safety-critical software development.

The Digital Security department and ICIS have always been a very pleasant work environment. Thank you for interesting research discussions, as well as countless coffee-breaks, cooking and beer related events and other extracurricular activities: Fabian, Bas Lijnse, Bas Joosten, Freek, Bariş, Joeri, Sven, Wojciech, Leonard, Harald, Jeroen, Irma, Ronny, Engelbert, Bart, Ken, Gerhard, Pim, Roel, Alejandro, Thomas, Julien, Bas Spitters, Merel, Paulan, Gergely, Wouter, the "Quantum Squad" and all others.

To my paranymphs Daan and Roderick, thank you for your help with the organisation of my defence. Moreover, thank you so much for being by my side on this big day, as you were throughout my PhD studies.

I would like to thank my parents, Mieke and Willy, and the rest of my family and friends for their endless support. The last five years have not always been easy, but you have invariably been there for me when I needed you.

Finally, I would like to thank my lovely wife Silke. Thank you for your unconditional support and for your endless patience and understanding in stressful times. I look forward to our new adventure in California and to spending the rest of our lives together. You mean the world to me.

Table of Contents

Chapter 1

Introduction

Software is ubiquitous. It is not only found in obvious places like a desktop computer or a smart phone, but practically every modern electronic device is controlled by software, including household appliances, cars, power plants and sluices. Failure can be lethal. Functional correctness and security are crucial for safe operation. In a world that is quickly depleted of its resources, energy-efficiency is vital.

A wide array of software analysis techniques is available, all aiming to ensure that software indeed exhibits desired functional or non-functional properties. These range from running the program once, to providing a mathematical proof using a theorem prover. More rigorous methods typically require a significantly larger effort. There is no "silver bullet" of software analysis, choice of methods depends highly on circumstance, e.g. the properties of interest and criticality.

Which technique is fit for a certain application also depends on the property of interest. Most of the research on which this thesis is based was done in the context of the GoGreen project, which aims to build a self-learning, secure and energy-efficient smart-home system. Such a system combines wireless sensor nodes to observe a house and the people within its walls with intelligent algorithms that control heating, ventilation, lighting and appliances. We focus on properties that are crucial for such a system: functional correctness, security and efficient use of resources. The latter point is of special interest for the wireless sensor nodes, which are highly resource-sensitive.

Many properties are impossible to automatically analyse *in general*, as this would mean solving the halting problem. However, this does not imply that we cannot analyse such properties in some (most) cases. Thus, when devising a software analysis method, it is often the aim to maximise the set of analysable programs. Alternatively, the aim can be, e.g., to verify stronger properties.

Software analysis methods can be roughly divided into two categories: *dynamic analysis*, where a program is analysed while executing it, and *static analysis*, where the program's source, binary code, or a model is analysed. Mixtures of static and dynamic analysis also exist, such as the loop-bound analysis presented in Chapter 5, in which the results of dynamic analysis are used to guess a loop bound, which is then verified statically. Note that, while the term static analysis for many people is associated with compile-time syntactical checkers – often yielding many false positives and false negatives – we focus on semantic analysis in this thesis.

Dynamic analysis and static analysis are introduced in Section 1.1 and Section 1.2, respectively. Each of these sections also introduces the specific techniques that are used in this thesis. When applying an analysis technique to a system, this increases the trust in the system. Techniques that provide a higher level of trust usually require a greater effort. The order in which techniques are discussed in this chapter is based loosely on the level of trust they can provide. For each technique, an introduction is given, along with an example of an application or extension, and a description of where it is used in this thesis. After introducing the software analysis methods used in this thesis, a motivation of the properties of interest and the thesis outline are given in Section 1.3. Software analysis in this thesis is focused on properties that are essential for the software used in a smart-home system as researched within the GoGreen project. Finally, in Section 1.4, the papers on which this thesis is based are listed and the contributions of the author are highlighted.

1.1 Dynamic Analysis

Dynamic analysis is any type of program analysis in which the program is executed. It is run for certain test input and its behaviour is studied. As such, the effectiveness of dynamic analysis highly depends on the considered input, as it can only provide certainty for behaviours that always occur for the test input.

We first discuss program testing in Section 1.1.1. We then describe debugging and profiling in Section 1.1.2, as this is very similar to the method that is used for loop-bound analysis in Chapter 5 and 6.

1.1.1 Testing

Software testing is perhaps the most ubiquitous program analysis technique. Consequently, there is a large body of research on the subject. A good introduction to software testing is given in [MSB11], which defines testing as “the process of executing a program with the intent of finding errors”. In *black-box testing*, the internal structure of the program is unknown, i.e. the test cases are fully based on the specification of the program. When test cases are based on the program code, we speak about *white-box testing.* In that context we can apply *coverage* measures to assess the adequacy of the test data, such as the percentage of statements covered by the tests.

Example. One relatively extreme coverage criterion is Modified Condition/Decision Coverage (MC/DC). Air traffic in the United States is regulated by the Federal Aviation Administration. As such, this organisation provides aircraft certification. In order to be certified, all airborne software must comply with the DO-178C standard [SC-11]. This quality assurance standard specifies that testing must achieve MC/DC coverage. This is a rigorous coverage criterion, that specifies that “every point of entry/exit in the program has been invoked at least once, every decision has taken all possible outcomes at least once, every condition in a decision has taken all outcomes at least once and every condition in a decision has been shown to independently affect that decision its outcome” [HVCR01]. An empirical evaluation of the MC/DC coverage criterion is presented in [DL00], where it is applied for the control software of the HETE-2 (High Energy Transient Explorer) satellite, built by the MIT Center for Space Research for NASA. In this evaluation, it is found that important errors are detected by a test set that satisfies the MC/DC criterion, that are missed by black-box functional testing or by structural testing using weaker criteria, such as decision coverage and condition/decision coverage.

In this thesis. In Chapter 3, coverage of branches inside loops is discussed, with respect to test data generated using symbolic execution. This extension is described in more detail in Section 1.2.3, where symbolic execution is introduced.

1.1.2 Debuggers and Profilers

To gain more information about the program at run-time, one can run it in a debugging or profiling tool. Such a tool can for instance provide insight into the data-structures allocated in memory at a certain break-point in the program, estimate time-complexity of a function or calculate the percentage of time spent in a certain part of the program. In many cases, the program is instrumented with additional code, to provide more information to the tool. Generally, debugging tools are focused on functional correctness, while profiling tools are mostly concerned with performance and diagnostic issues. A thorough survey of dynamic analysis techniques for program comprehension is given in [CZvD$^+$09].

Example. Software profiling can be used for dynamic energy consumption analysis [SKZS12, WGR13, NRS14]. The profiler is placed at the operating system level and uses an energy model. This eliminates the need for a hardware measurement set-up. Such an energy model must be able to predict energy consumption of operating system calls with sufficient accuracy. Not only do they need to estimate the energy-usage of the processor, but also that of other components (e.g. hard drives and network controllers), which often consume energy based on their internal state (i.e. not synchronous with the system call).

In this thesis. A profiling-like method is used in Chapters 5 and 6, where program loops are instrumented with a counter in order to determine the number of iterations for certain input values. By interpolating the results, loop bounds are inferred. These bounds hold for the tested inputs. Whether or not they are correct for *any* input is verified statically, using a theorem prover. This last step is discussed in more detail in Section 1.2.5, where theorem proving is introduced.

1.2 Static Analysis

Static analysis is any type of program analysis in which the program is not executed. Instead, the source code, binary or a model is used to derive or prove certain properties. Such properties typically do not hold only for a finite set of test inputs, but rather for well-defined classes of inputs (ideally, all inputs).

The static program analysis techniques we discuss in this section are limited to those used in this thesis. Other techniques, including type systems [Pie02] and abstract interpretation [CC77], are omitted. Flow analysis is introduced in Section 1.2.1. Hoare logic is discussed in Section 1.2.2 and symbolic execution in Section 1.2.3. Model-checking is introduced in Section 1.2.4. Finally, theorem proving is discussed in Section 1.2.5.

1.2.1 Flow Analysis

In static analysis, alternative (graph) representations of programs are often constructed, to which standard mathematical methods can be applied. The *Control-Flow Graph* (CFG) has so-called basic blocks as nodes. These are the maximal blocks of code that are always executed in sequence, i.e. there can be only jumps to the first expression in a basic block and only jumps from the last expression. The edges in the CFG represent the jumps between basic blocks. A CFG can for instance be used for *data-flow analysis*, which tries to restrict the set of possible values a variable might have by propagating constraints through the CFG. An excellent introduction to data-flow analysis is given in [NNH99].

Example. Another example where an alternate representation of a program is analysed mathematically is resource analysis through recurrence relation solving, as in COSTA [AAG^{+}08]. In this technique, the program is transformed into a set of recursive formulas expressing consumption of a certain resource, e.g. heap-space. This recurrence relation is then transformed into closed form, capturing the resource consumption of the program in a single formula.

In this thesis. The control-flow graph of a program is used to detect loops at the JAVA bytecode level in Chapter 3. If an edge exists in the CFG from n to h and h dominates n (all paths from the entry node to n go through h), then the edge is the back-edge of a loop with header h. Furthermore, in Chapter 6, COSTA is used for heap-space analysis. A post-processing step is added, where an inaccuracy is corrected. Results are simplified for easier understanding and possibly concretised for a particular JAVA virtual machine. Data-flow analysis is used in Chapter 6 to concretise symbolic stack-usage bounds. The presented tool RESANA first calculates a symbolic stack space usage bound. A concrete number of bytes (upper bound) for this symbolic bound can be calculated by incorporating data-flow analysis from the tool VERIFLUX.

1.2.2 Hoare Logic

Hoare logic or Floyd-Hoare logic is a set of formal rules for reasoning about programs. It was developed by Hoare in [Hoa69], inspired by an earlier work by Floyd [Flo67]. The central concept of Hoare logic is the so-called Hoare triple: $\{P\}\ S\ \{Q\}$, where P is the *precondition*, S is a statement and Q is the *postcondition*. An example of a valid triple is $\{x = 5\}\ x := x + 1\ \{x = 6\}$. Using such triples, a set of axioms for simple statements (e.g. skip or assignment) and inference rules for more complex programming constructs (e.g. composition or a conditional) are defined. Loops are normally handled using a loop invariant, that holds before execution of the loop, after each iteration and after the loop.

Example. Basic Hoare logic does not have rules for many constructs that occur in modern imperative programming languages. Therefore, many extensions to this basic logic exist, e.g. for Java concepts such as dynamic method binding and exception handling [vO01, HJ00]. One extension that has received much attention is *separation logic*, which can be applied for local reasoning about separate parts of the heap [Rey02, ORY01].

In this thesis. In Chapter 4, a Hoare logic for energy-consumption analysis is presented. The language to which this logic is applied is a simple *while* language, extended with a construct for calling functions on specific components. Conditions in the logic are extended with energy-aware component states. When the inference rules of this logic are applied to a program and energy-models for the components it works on, the result is an upper-bound on the energy consumption of each of the components.

1.2.3 Symbolic Execution

In symbolic execution [BEL75, Cla76, Kin76, RHC76, CGK^{+}11, CS13], a program is executed with symbols instead of actual input. For each branch in the program, both choices must be explored, maintaining a *path condition*. Infeasible paths should be excluded. Feasibility can be verified by checking satisfiability of the path condition. Typically, whenever a path condition is extended, satisfiability is checked using an off-the-shelf SMT-solver, e.g. Yices [DdM06], Z3 [dMB08] or CVC4 [BCD^{+}11]. SMT (Satisfiability Modulo Theories) is an extension of SAT with formal theories such as the theory of integers. SAT, or the Boolean Satisfiability Problem, is the problem of determining satisfiability of a Boolean proposition. It is well-known to be NP-complete [Coo71], still many algorithms can solve SAT problem fairly efficiently for practical applications, such as the ones implemented in the SMT-solvers mentioned above.

Example. Many modern symbolic execution techniques mix concrete and symbolic execution, e.g. Symbolic Pathfinder [PR10]. This so-called *concolic execution* executes the program on concrete input and maintains both a concrete state and a symbolic state. The symbolic state only contains values for the subset of program variables marked as symbolic by the user. When concrete execution for the given input is done, concolic execution backtracks and the collected symbolic constraints on the execution path are used to generate inputs that force concrete execution down an alternative feasible path. This process continues until either all paths are explored (which is unlikely, due to path explosion) or some other condition is met (typically a coverage criterion).

In this thesis. Symbolic execution is used in Chapter 3 to generate test-cases. This can be done by finding a model for the path condition of every path through the program. However, input-dependent loops often imply infinite paths. Symbolic execution must therefore be bounded. Furthermore, scalability is limited because of the exponential increase in the number of paths through loops (path explosion), even those that are not dependent on input. Because of the bounding of loops, branching that only occurs after n iterations of a loop might be missed by test data generated using symbolic execution. An implementation of an alternate bounding mechanism is presented, as well as a novel method to concretise symbolic variables in order to execute all branches. Symbolic execution is used in Chapter 6 to infer a so-called *update function* which captures the behaviour of a loop for a certain variable.

1.2.4 Model-Checking

In model-checking [CE82, QS82, CES86, CGP99, BK08, CES09], not the software itself, but rather a finite model representation is exhaustively verified. The model, essentially a finite state machine, is usually written in a special modelling language. In some cases, such as with the tool UPPAAL [BLL+96], the finite state machine can be directly constructed in a graphical interface. The model-checker exhaustively traverses the finite state machine and verifies that a given logical property holds in every reachable state.

Example. Several software analysis methods that are used to verify the reliability of mission-critical flight software at NASA's Jet Propulsion Laboratory are discussed in [GHH+14]. A Flash file-system used in space-craft serves as a case-study. An anomaly in this file-system previously led to a loss of communication that lasted several days, when it was deployed in the Mars Exploration Rover "Spirit" [RN05]. One of the methods applied to the file-system case-study is model-driven verification, in which the actual program code is used for model-checking, instead of a model [HJ04]. A major challenge with model-checking in general and model-driven verification in particular, is scalability, due to the state-space explosion problem. In this case, this is countered by using abstract states, i.e. many concrete states map to the same abstract state, greatly reducing the search space. Because of file-system intrinsics, using sound abstractions did not turn out to be very helpful. Instead, unsound abstractions were used, meaning that this particular approach finds bugs but does not prove their absence.

In this thesis. Model-checking is used in Chapter 2 to find under-specified parameter values to ensure security of a device pairing protocol. In Chapter 3, the tool Symbolic Pathfinder is extended in order to improve the coverage of the test-cases it generates. Symbolic Pathfinder itself builds upon Java Pathfinder, which is a model-checking tool for Java.

1.2.5 Theorem Proving

In theorem proving, the validity of a logical proposition is proved using an (automated) proof assistant. Examples of theorem provers are PVS [ORS92], Coq [BC04], ACL2 [KMM00], Isabelle [Pau94] and KeY [BHS07]. The advantage of using a proof assistant over proving the proposition on paper, is that it rules out human error and potentially automates a large part of the process. Depending on the used proof assistant, the theorem to be proved might be described in classical logic, or including a form of program code. In the latter case, a formalisation of the programming language in the logic used by the proof assistant is required. Each theorem prover typically uses its own form of program logic, often based on classical systems like Hoare logic. This is the case, for instance, in KeY its *dynamic logic*, which includes Java code in its propositions.

Example. In [MSvE10], theorem proving is one of the techniques used to verify correctness of the Rotterdam Storm Surge Barrier (Maeslantkering). This barrier is one of the largest moving structures in the world and protects the southwest part of The Netherlands. Opening and closing of the barrier is controlled fully autonomously by a system called BOS. During formal verification, three discrepancies between the specification of a sub-system of BOS and its program code were found.

In this thesis. The theorem prover KeY is used in Chapters 5 and 6 to prove automatically inferred loop bounds. Dynamically inferred loop bounds are guaranteed to hold for the tested program inputs and expected to hold for *any* program input. A formal proof of their validity is therefore sought with the highly automated theorem prover KeY, which uses Java as input language.

1.3 Motivation and Thesis Overview

Most of the research presented in this thesis was done in the context of the IOP GenCom GoGreen project. This project studies a self-learning energy-preserving smart-home system, which reads various sensors in the house, learns about the household and the environment and actuates lighting, heating, ventilation, air-conditioning, appliances and other electronics. Sensing and actuation is mostly done using Wireless Sensor Nodes (WSNs), which are small, typically battery-powered, devices containing a simple micro-controller, one or more sensors and a wireless transceiver. Such devices are highly resource-sensitive, as they have a finite energy-supply and are equipped with limited memory. The main thesis can be formulated as: *How can we establish software properties, that are of particular interest to resource-sensitive systems?*

Desirable properties for GoGreen software – largely embedded on WSNs – are functional correctness, security, and energy, time and memory efficiency. Each of these properties corresponds to a sub-question for this thesis.

1.3.1 Security

The GoGreen system collects large amounts of sensitive data about the residents (or visitors) of the house in which it is deployed. Most of this data is communicated wirelessly, which must therefore be done in a secure way. Moreover, an attacker should be prevented from influencing communication and thereby gaining control over the actuated devices. Wireless communication can be secured by encrypting the stream of information. To enable this, the communicating parties must agree on a key (for symmetric cryptography) or a pair of keys (for asymmetric cryptography). Normally, this requires the user to enter a password, which is not possible on a WSN, as it does not have a keyboard or a screen. A solution is to use the Push-Button Configuration protocol defined within the Wi-Fi Protected Set-up standard, which enables pairing of two devices by pressing a (virtual) button on both devices within a certain time-frame. However, a series of vulnerabilities of this protocol is presented in [GAZK11]. The same paper presents an extension of this protocol, that protects it against attacks that take advantage of these vulnerabilities: Tamper-Evident Pairing (TEP).

Some of the parameters needed to implement TEP are under-specified. In Chapter 2, model-checking is used to analyse TEP and it is discovered that the protocol is vulnerable to attacks if these parameters are not chosen wisely. Model-checking is applied in an iterative fashion to discover what values of the parameters result in a tamper-evident set-up. From these results, a constraint ensuring security of the protocol from the results is formulated.

1.3.2 Functional correctness

As for all software, it is desirable that the software embedded on WSNs behaves according to specification. For embedded software this is extra important, because it can be very hard to update the software in case errors are detected after release. As most WSNs have uncommon interfaces for programming, GoGreen software running on WSNs in the home is not likely to ever be updated. It must therefore be thoroughly tested.

Chapter 3 presents two extensions to the tool Symbolic Pathfinder, one of which is an implementation of a known method for bounding symbolic execution of loops, the other is a novel method for improving the coverage of test cases generated by symbolic execution for programs with loops. This latter method works by symbolically executing the loop body *out-of-context*, i.e. by disregarding constraints over symbolic variables. This can produce models for symbolic variables that execute all branches within the loop body. The symbolic variables can then be concretised to these values.

1.3.3 Energy efficiency

WSNs are typically battery-powered. Quickly draining their batteries will prevent adoption of the GoGreen system by potential users. Moreover, the GoGreen system must at least be energy-neutral as a whole, but preferably save energy. One of the goals of the GoGreen project is to develop techniques for harvesting energy from surroundings, such as solar or kinetic energy, charging the battery. It is thus crucial that WSNs are energy-efficient. The hardware of the WSNs is controlled by the embedded software. It is therefore essential to use energy-efficient implementations.

Chapter 4 presents a Hoare logic for energy-consumption analysis. Given a model of the energy-related behaviours of the hardware components, this analysis can statically bound the energy consumption of software running on said hardware. The method is sound and is implemented in the tool ECALOGIC.

1.3.4 Time efficiency

WSNs typically wake up for a short time to do their work, then go back into an energy-saving mode again. Their program must be able to execute within this time-frame. It is therefore important to be able to bound execution time. As most of the execution time of typical embedded programs is spent in loops, it is an important step to bound loop iterations. Furthermore, loop bounds are a prerequisite for analysing consumption of *any* resource, as a certain amount of resources (possibly bounded) can be consumed on every iteration. Finally, bounding loops is also an essential step in proving termination, which might be required for proving functional correctness.

Chapter 5 presents a novel method to infer polynomial ranking functions for loops. It instruments the loop with a counter, then runs it for a set of test inputs and interpolates a polynomial over the resulting iteration counts. The set of test inputs is chosen such that it satisfies a condition that guarantees the existence of a unique interpolating polynomial. The presented loop-bound analysis is implemented in the tool RESANA.

1.3.5 Memory efficiency

Wireless sensor nodes need to be produced in a cost-effective manner, such that end-users can afford to place them ubiquitously throughout their house. They are therefore equipped with just as much memory as they need, not more. It is thus important to be able to bound memory consumption of their embedded software. Memory is usually allocated in two separate structures: the stack and the heap. The stack is a last-in-first-out list of *stack frames*. When a function is invoked, a new stack frame is *pushed* on the stack, containing local variables and bookkeeping data. When the function is finished, the corresponding stack frame is *popped* from the stack. The heap is the rest of the memory, which can be used for dynamic allocation. Whereas the stack contains ordered blocks with a well-defined structure, the heap can contain any arbitrary data-structure that the user defines. Due to its unstructured nature, the heap is harder to analyse.

Chapter 6 presents the tool ResAna and the underlying resource analysis methods. The loop bound analysis method from Chapter 5 is further extended with a way to deal with so-called *condition jumping*. A heap-space analysis is developed, using extensions of the resource analysis tool COSTA, which is based on recurrence relation solving. COSTA is extended by applying the polynomial interpolation based ranking function inference method to recurrence relation solving, correcting its results for arrays, simplifying its results and adding a Virtual Machine specialisation step. Furthermore, a stack-space analysis is presented. This analysis uses COSTA to obtain a measure for recursive functions (analogous to a ranking function for loops), then combines this with data-flow analysis and measured stack frame sizes to obtain a concrete upper bound on the consumed stack space.

1.4 Contributions

This section lists the publications that form the basis for this thesis and highlights the contributions of the author. This thesis is based on a total of seven publications, four of which were presented at international workshops, two at international conferences, and one (an extended version of one of the workshop papers) was published in the Journal of Concurrency and Computation: Practice and Experience. Except for moderate restructuring, minor corrections and slight changes in the layout, the content of each chapter is the same as that of the original publication(s). Chapter 4 is based on the contents of two publications. Chapter 5 and Chapter 6, together, are based on the contents of three publications. Note that the chapters are not presented in the chronological order of the original publications. Rather, chapters are bundled with respect to the analysed property type (security, functional correctness, resource consumption).

Chapter 2 presents the analysis of the Tamper-Evident Pairing (TEP) protocol using model-checking. It is based on [KvGD$^+$13]:

> Rody Kersten, Bernard van Gastel, Manu Drijvers, Sjaak Smetsers, and Marko van Eekelen. Using model-checking to reveal a vulnerability of tamper-evident pairing. In *Proceedings of the 5th NASA Formal Methods Symposium*, number 7871 in Lecture Notes in Computer Science, pages 63–77. Springer, May 2013.

Model-checking TEP required a thorough understanding of the protocol in order to build a model which captures the essential security issues. The author made the transition from the protocol description in [GAZK11], an academic paper, to the unambiguous implementation in a Spin model (with the exception of the clock model and the adversary model). The author of this thesis has written most of the publication.

This chapter differs slightly from the original publication. Figures have been improved (two have been merged) and the nature of the discovered vulnerability is described more extensively.

Chapter 3 presents two extensions to the symbolic execution tool Symbolic Pathfinder, that improve the coverage of test cases generated by symbolic execution for programs with loops. This chapter is based on [KPRT15]:

> Rody Kersten, Suzette Person, Neha Rungta, and Oksana Tkachuk. Improving coverage of test-cases generated by Symbolic PathFinder for programs with loops. In *SIGSOFT Software Engineering Notes*, 40(1):1–5. ACM, January 2015.

This work consists of the author's own contributions and was written during an internship at the NASA Langley Research Center in the Formal Methods group. The author received supervision from Suzette Person (NASA Langley), as well as guidance from Neha Rungta and Oksana Tkachuk (NASA Ames).

Chapter 4 presents a Hoare logic for energy-consumption analysis. Given a model of the energy-related behaviors of the hardware components, this analysis can statically bound the energy consumption of software running on said hardware. This chapter is largely based on [KPvv14]:

> Rody Kersten, Paolo Parisen Toldin, Bernard van Gastel, and Marko van Eekelen. A Hoare logic for energy consumption analysis. In *Proceedings of the Third International Workshop on Foundational and Practical Aspects of Resource Analysis*, FOPARA'13, number 8552 in Lecture Notes in Computer Science, pages 93–109. Springer, October 2014.

The author of this thesis has co-developed the presented energy analysis, focusing mostly on the modeling aspect. He has written most of the original publication and the corresponding technical report [PKvv13], and devised the examples.

Chapter 4 also contains elements that were taken from [SNKvE14], mainly the discussion of the implementation of the analysis in the tool ECAlogic:

> Marc Schoolderman, Jascha Neutelings, Rody Kersten, and Marko van Eekelen. ECAlogic: Hardware-parametric energy-consumption analysis of algorithms. In *Proceedings of the 13th Workshop on Foundations of Aspect-oriented Languages*, FOAL'14, pages 19–22. ACM, April 2014.

The tool ECAlogic is an implementation of the presented energy analysis. It was developed by a group of students in the course *System Development*

Research. The author actively supervised this group and wrote the original publication together with students Marc Schoolderman and Jascha Neutelings, guided by Marko van Eekelen.

Chapter 5 presents a novel method to infer polynomial ranking functions for loops. This chapter is based on [SKVE10]:

> Olha Shkaravska, Rody Kersten, and Marko Van Eekelen. Test-based inference of polynomial loop-bound functions. In *PPPJ'10: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, pages 99–108. ACM, September 2010.

The author co-developed the interpolation-based method for loop bound analysis and implemented it in the tool ResAna.

A limited description of several extensions to the basic method is given in [SKVE10]. This section is not included in Chapter 5, in lieu of the more elaborate description in Chapter 6, which also contains further extensions. The section on future work is also not included, as it has become redundant with the publication of [KvGS$^+$14] and the corresponding extensions to the implementation ResAna.

Chapter 6 presents extensions to the loop-bound analysis, a heap-space analysis, a stack-space analysis and their implementation in the tool ResAna. It is based on [KvGS$^+$14]:

> Rody W. J. Kersten, Bernard E. van Gastel, Olha Shkaravska, Manuel Montenegro, and Marko C. J. D. van Eekelen. ResAna: a resource analysis toolset for (real-time) JAVA. *Concurrency and Computation: Practice and Experience*, 26(14):2432–2455. Wiley, September 2014.

This paper was published in a special edition of the *Journal of Concurrency and Computation: Practice and Experience*, with selected papers from the *Workshop on Java Technologies for Real-time and Embedded Systems 2012.* It is an extended version of [KSvG$^+$12]:

> Rody Kersten, Olha Shkaravska, Bernard van Gastel, Manuel Montenegro, and Marko van Eekelen. Making resource analysis practical for Real-Time Java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES'12, pages 135–144. ACM, October 2012.

The author's contributions are the extensions to the loop bound analysis, supportive work for the memory analysis methods and co-authoring and editing the publications.

The contents of this chapter is the exact publication [KvGS$^+$14], except the introduction of the loop bound analysis method. This section from the original publication is omitted, as the loop bound analysis method is described in detail in Chapter 5 of this thesis.

Chapter 2

Using Model-Checking to Reveal a Vulnerability of Tamper-Evident Pairing

Abstract. Wi-Fi Protected Setup is an attempt to simplify configuration of security settings for Wi-Fi networks. It offers, among other methods, Push-Button Configuration (PBC) for devices with a limited user-interface. There are however some security issues in PBC. A solution to these issues was proposed in the form of Tamper-Evident Pairing (TEP). TEP is based on the Tamper-Evident Announcement (TEA), in which a device engaging in the key agreement not only sends a payload containing its Diffie-Hellmann public key, but also sends a hash of this payload in a special manner, that is trusted to be secure. The idea is that thanks to the special way in which the hash is sent, the receiver can tell whether or not the hash was altered by an adversary and if necessary reject it.
Several parameters needed for implementation of TEP have been left unspecified by its authors. Verification of TEA using the Spin model-checker has revealed that the value of these parameters is critical for the security of the protocol. The implementation decision can break the resistance of TEP against man-in-the-middle attacks. We give appropriate values for these parameters and show how model-checking was applied to retrieve these values.

2.1 Introduction

Security protocols aim at securing communications over networks that are publicly accessible. Depending on the application, they are supposed to ensure security properties such as authentication, integrity or confidentiality even when the network is accessible by malicious users, who may intercept and/or adapt existing, and send new messages. While the specification of such protocols is usually short and rather natural, designing a secure protocol is notoriously difficult.

Flaws are often found several years later. One of the sources for the vulnerability of such protocols is that their specification is often (deliberately) incomplete. There are several reasons for the omission of certain details by the designer. For instance, a protocol may depend on properties of the hardware on which it is used. It also leaves some room for the implementer of the protocol to make implementation-dependent choices. The problem with these unspecified parameters is that it can be very hard to analyse the effects of specific choices on the correctness of the protocol itself. Mostly this is due to the fact that the protocol is specified in such a way that both designer and implementer are either convinced that the correctness is not influenced by the concrete values of these parameters, or they assume that theses values are chosen within certain (not explicitly specified) boundaries.

During the last two decades, formal methods have demonstrated their usefulness when designing and analysing security protocols. They indeed provide rigorous frameworks and techniques that allow to discover new flaws. For example, the ProVerif tool [Bla01] and the Avispa platform [ABB$^+$05] are both dedicated tools for automatically analysing security properties. More general purpose model-checkers, such as Spin [Hol97] and Uppaal [BLL$^+$96], are also successfully applied to verify desired properties of protocol specifications. While this model-checking process often reveals errors, the absence of errors does in general not imply correctness of the protocol.

Secure wireless communication is a challenging problem due to the inherently shared nature of the wireless medium. For wireless home networks, the so-called Wi-Fi Protected Setup was designed to provide a standard for easy establishment of a secure connection between a wireless device with a possibly limited interface (e.g. a webcam or a printer) and a wireless access point. The wireless device, once connected to the access point, gets not only internet connectivity, but also access to shared files and content on the network. The standard provides several options for configuring security settings (referred to as *pairing* or *imprinting*). The most prominent ones are PIN and Push-Button Configuration. The PIN method has been shown to be vulnerable to brute-force attacks; see [Vie11]. This method and its weaknesses are briefly discussed in Section 2.5. To establish a secure connection using the Push-Button method, the user presses a button on each device within a certain time-frame, and the devices start broadcasting their Diffie-Hellman public keys [DH76], which are used to agree on the encryption key to protect future communication. In [GAZK11] the authors argue that this protocol only protects against passive adversaries. Since the key exchange messages are not authenticated, the protocol is vulnerable to an active man-in-the-middle (MITM) attack. To protect key establishment against these MITM attacks, [GAZK11] presents a method called *Tamper-Evident Pairing* (TEP), that provides simple and secure Wi-Fi pairing without requiring an out-of-band communication channel (a medium, differing from the communication channel that is used for transmitting normal data). The essence of their method is that the chip-sets used in Wi-Fi devices offer the possibility not only to transmit data, but also to sense the medium to detect whether or not information is communicated. The correctness of the proposed protocol is based on the assumption that an adversary can only change or corrupt data on the medium but not completely remove the data. The TEP protocol is specified in a semi-formal way; its correctness is proven manually (i.e. on paper; not formally using e.g. a theorem prover). However, in the protocol itself some parameters are used that are not fully specified.

In this chapter, we investigate the TEP protocol in order to determine whether its correctness depends on the values chosen for the unspecified parameters. In other words, we analyse the protocol by varying the values of these parameters in order to find out if there exists a combination for which correctness is no longer guaranteed. Our analysis is done using the Spin model-checker. We have modelled the essential part of the protocol (known as the Tamper-Evident Announcement), and used this model to hunt for potentially dangerous combinations of parameters, which indeed appeared to exist. The next step was to explore the vulnerability boundaries, by deriving a closed predicate relating the parameters to each other and providing a safety criterion. The derivation of this predicate, and the verification of the resulting safety criterion, was done by using the model-checker. The contribution of our work is twofold. First, it reveals a vulnerability of a protocol that was 'proven to be correct'. And secondly, it shows how model-checking can be used, not only to track down bugs, but also to establish side-conditions that are essential for the protocol to work properly.

2.2 Tamper-Evident Pairing

The Wi-Fi Alliance has defined the Wi-Fi Protected Setup (WPS) standard in [Wi-06]. The standard provides several options for simple configuration of security settings for Wi-Fi networks (pairing). One of them is Push-Button Configuration (PBC), where two devices (enrollee and registrar) are paired by pressing a (possibly virtual) button on each of the devices within a time-out period of two minutes. Security of this method is enclosed in the fact that the user needs physical access to both devices. However, in [GAZK11], three vulnerabilities are described creating opportunity for man-in-the-middle attacks:

1. **Collision:** An attacker can create a collision with the enrollee's message and send his own message immediately after.
2. **Capture effect:** An attacker can transmit a message at a much higher power than the enrollee. Capture effects were first described in [WJCD00].
3. **Timing control:** An attacker can occupy the medium, prohibiting the enrollee from sending his message, and send his own message in-between.

Gollakota et al. also provide an innovative solution to the PBC security problems in [GAZK11]. Their alternative pairing protocol is named Tamper-Evident Pairing (TEP). It is based on the fact that Wi-Fi devices can not only receive packets, but also simply measure the energy on the channel, as part of the 802.11 standard requirements. This provides the opportunity to encode a bit of information as a time-slot where energy is present or absent on the wireless medium. Under the assumption that an attacker does not have the ability to remove energy from the medium, this means that an attacker cannot turn an on-slot into an off-slot.

Let us start by explicating the attacker model, i.e. the assumptions about the adversary that we are securing the protocol against. The presence of an active adversary is assumed, who is trying to launch a MITM attack. He/she has the following capabilities:

Overwrite data packets The adversary can use any of the three vulnerabilities listed above to overwrite data packets.

Introduce energy on the channel The adversary can introduce energy on the channel. Energy cannot be eliminated from the wireless medium.

2.2.1 The Tamper-Evident Announcement

To facilitate TEP, Gollakota et al. introduce the Tamper-Evident Announcement (TEA) primitive, which is sent in both directions: enrollee to registrar and vice-versa. The structure of a TEA is given in Figure 2.1. It starts with the so-called *synchronisation packet*. This an exceptionally long packet, filled with random data. It is detected by the receiver by measuring a burst of energy on the medium of at least its length (so in a manner similar to the on-off slots). Because this packet is exceptionally long, this uniquely identifies a TEA.

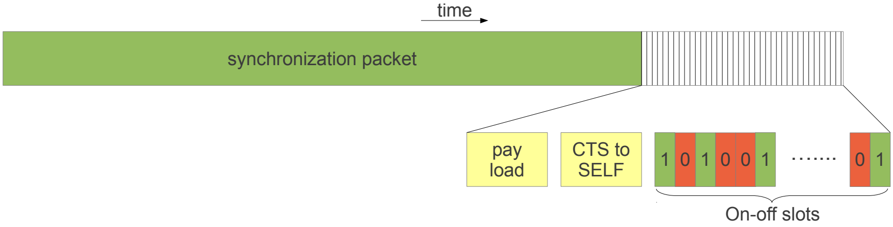


Figure 2.1: The structure of a Tamper-Evident Announcement (TEA)

The synchronisation packet is followed by the payload of the TEA, which contains the Diffie-Hellman public key [DH76] of the sender. Then, a *CTS-to-self* packet is sent. This message is part of the IEEE 802.11 specification and requests all other Wi-Fi devices not to transmit during a certain time period, here the time needed for the remainder of the TEA.

Finally, a hash of the payload is sent by either transmitting or refraining from transmitting during a series of so-called on-off *slots*. An attacker cannot change an on-slot into an off-slot, because he/she cannot remove energy from the medium, but might still do the opposite. To be able to detect this as well, a specially crafted bit-balancing algorithm is applied to the 128-bit hash, prolonging it to 142 bits (71 zeros and 71 ones). Now, when an off-slot is changed into an on-slot, the balance between on and off slots is disturbed, making the tampering detectable. The 142-bit bit-balanced hash is preceded by two bits representing the direction of the TEA (enrollee to registrar or vice-versa). So, in total, 144 slots are sent.

2.2.2 Receiving the Slots

The sender sends out the 144 slots, which take 40 μs each, back-to-back. On the receiver-side the slots are received by measuring energy on the wireless medium. The receiver iteratively measures the energy on the medium, during so-called *sensing windows* of 20 μs. The total number of measurements m during the sensing window is stored, as well as the number of measurements e during which there was energy on the medium. If the *fractional occupancy*, given by e/m, is above a certain threshold then the medium is considered occupied during this particular sensing window.

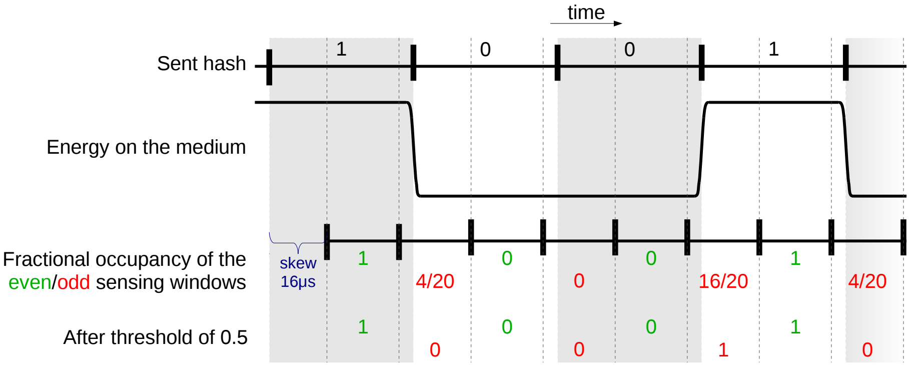


Figure 2.2: Sending and receiving the slots of a 4-bit hash. The even sensing windows have the higher variance here. Therefore, those represent the received hash. Clock skew (16 μs) is shown in blue on the left.

The length of a sensing window is half the slot-length. The reason for this is that now either all the even sensing windows or all the odd sensing windows fall entirely within a slot, i.e. do not cross slot-boundaries. This is shown in Figure 2.2, where the even sensing windows all fall entirely within one of the 40 μs slots. Note that the figure shows the ideal case, where measurements are exact. In reality the measurements will be less than perfect, which motivates the use of a threshold. The use of this special method of receiving the slots is motivated by the fact that there may be a slight clock-skew. This is shown in Figure 2.2 on the lower-left.

After all the measurements are done and after applying the threshold, the receiver verifies that either the even or the odd sensing windows have an equal number of zeros and ones[1]. Then, the receiver checks that the received bits match a calculated hash of the payload packet. If this is not the case, the receiver aborts the pairing process.

2.3 Modelling the Tamper-Evident Announcement in Spin

We use the same attacker model as the authors of [GAZK11], listed in Section 2.2. Given that an adversary can replace the payload packet, we will try to verify that he/she cannot adapt the bits of the hash that are received without being detected. Namely, if the attacker manages to send his/her own payload and adapt the hash such that it matches this payload and contains an equal number of zeros and ones, he/she can initiate a MITM attack. The payload packet itself is therefore not part of the model. We will only model the sending of the bit-balanced hash. The direction bits (i.e. the first two slots) are also not modelled. Gollakota et al. give an informal proof of the security of TEP in [GAZK11]. Effectively, we are challenging Proposition 7.2 of their proof.

We used Spin [Hol97] for the verification of the model. This section contains illustrative fragments from the model only. The full model (including results) can be downloaded from `http://www.cs.ru.nl/R.Kersten/publications/nfm/`.

2.3.1 Model Parameters

The model has a series of parameters that are described in this section.

Hash length The length of the bit-balanced hash to send. All possible hashes of this length that are bit-balanced are tried (the balancing algorithm itself is not part of the model).

Number of measurements per sensing window The number of measurements in each sensing window depends on the Wi-Fi hardware on which the protocol is implemented. The length of the window is 20 μs. During each window, the hardware logs the total number of clock-ticks, as well as the number of clock-ticks during which there was energy on the wireless medium. The number of measurements during each sensing window is thus variable. In the model though, the number of measurements is fixed and given by a parameter. The reason for this is that a variable number of measurements would highly enlarge the state-space, the number of measurements is not something

[1] Actually, the variance of all the even sensing window measurements and that of all the odd sensing window measurements is calculated. The sensing windows with the higher variance will be the correct ones, since on and off slots are balanced. It is however not clear to us what the advantage of this approach is over simply selecting the sensing windows in which the on-off slots are balanced.

that an adversary can influence and that we believe it will be fairly constant in practice. A programmer implementing the protocol could measure or calculate the average number of measurements during a sensing window and use a “safe” approximation (a little lower) in the formula. In our model, the sender puts energy on the wireless medium for the number of clock-ticks it takes to do the measurements for two sensing windows (the sensing window has half the length of an on-off slot). This means that one measurement is the unit for a clock-tick.

Sensing window threshold As explained in Section 2.2.2, bits are received by measuring the *fractional occupancy* during a sensing window. It is determined whether or not the medium was occupied in a sensing window by checking if the fractional occupancy is above a certain threshold. The value of this threshold is not defined in [GAZK11], although it influences the measurements heavily. Since the number of measurements during a sensing window is constant in the model, we can omit the calculation of the fractional occupancy. This means that also the threshold should now be modelled, not as a number between 0 and 1, but as a number between 0 and the number of sensing window measurements and that its unit is clock-ticks (the medium was occupied during e ticks of the discrete clock). If the number of measurements (clock-ticks) in a sensing window where there was energy on the medium exceeds the threshold, then a one is stored for this sensing window.

Skew The reason for the use of pairs of sensing windows for receiving the slots is that there may be an inherent clock skew. It is stated in [GAZK11] that this inherent clock skew may be up to 10 μs, i.e. half the sensing window length. By using pairs of sensing windows, either the even or the odd windows are guaranteed not to cross slot-boundaries. Furthermore, it is stated in [GAZK11] that to detect a TEA it is sufficient to detect a burst of energy “at least as long as the synchronisation packet”. It is not specified which is the exact synchronisation point: the beginning or the end of the energy pulse. Neither is the maximum length of an energy burst that signifies a synchronisation packet. The difference with the given length of 19ms introduces an extra skew. Since an adversary can introduce energy to the wireless medium, he/she can prolong the synchronisation packet and introduce extra skew (the sign of this skew depends on the choice of synchronisation point). The model variable *skew* is the total of the inherent clock skew and this *attacker skew*. Like the number of measurements and the threshold, its unit is also clock-ticks. We only consider positive skew (forward in time) in our model.

These parameters to the model are henceforth referred to as *hash_length*, *sw_measurements*, *threshold* and *skew*, respectively.

2.3.2 Clock Implementation

Timing is essential to modelling the TEA. However, Spin has no inherent notion of time. Luckily, in this case the exact scheduling and execution speed are not important, as the only interaction between the sender and receiver processes is sending energy to and reading the energy-level from the wireless medium. The receiver observes the value of the wireless medium once per clock cycle, the sender updates it at most once.

Due to these properties, we can implement a discrete clock in PROMELA (the modelling language used by SPIN), without the need to use specialised model-checkers with native clock support. We introduce a separate clock process, which waits until all processes using a clock are finished with a clock cycle (Listing 2.1, line 17), before signalling them to continue. Processes are signaled to continue by flipping the Boolean *clock* (line 23). Processes can only continue with the next clock cycle if this variable differs from their local variable *localclock* (line 10), which is also flipped after each clock-tick (line 11). Our clock implementation also supports processes which do not use a clock. A clock-tick in the model corresponds to a measurement taken by the receiver. To avoid the situation that the receiver executes before the sender, we implemented explicit turns for the processes, so the sender always executes first after a clock-tick. The process with the lowest process identifier is always executed first (line 7). We can introduce skew by letting one of the processes wait a number of clock-ticks before starting.

```
 1 byte waiting = 0;
 2 bool clock = false;
 3 #define useClock() bool localclock = false;
 4
 5 inline waitTicks(procID, numberOfTicks) {
 6   byte tick;
 7   for (tick : 0..(numberOfTicks-1)) {
 8     waiting++;
 9     atomic {
10       localclock != clock;
11       localclock = clock;
12       waiting==procID;
13     }
14   }
15 }
16 proctype clockProc() {
17 end:
18   do
19   :: atomic {
20     waiting==NUMBER_OF_CLOCK_PROCESSES;
21     waiting = 0;
22     clock = !clock;
23   }
24   od;
25 }
```

Listing 2.1: Modelling the clock. The `useClock` and `waitTicks` functions must be used in processes that use the clock.

2.3.3 Model Processes

The model begins with a routine that generates all possible hashes of the given length non-deterministically. It then starts four processes:

Clock A simple clock process is used to control the other processes. The clock process is described in Section 2.3.2.

Sender The sender process first initialises and starts the clock. It then iteratively sends a bit of the generated hash (by putting energy on the medium, or not), waits for $2 \cdot sw\_measurements$ clock-ticks, then sends the next bit. When finished sending, the sender must keep the clock ticking, because the receiver process might still be running.

```
proctype sender() {
  useClock();
  waitTicks(0, 1);
  byte i;
  for (i : 0..(HASH_LENGTH-1)) {
    mediumSender = get(i); //send slot
    waitTicks(0, SW_MEASUREMENTS*2);
  }
  doneWithClock(0);
}
```

Listing 2.2: Sender model.

Receiver The receiver also begins with initialising and starting the clock. It then introduces clock skew by waiting *skew* more ticks. Then, it measures energy on the medium *sw_measurements* times (once every clock-tick) and stores the received bit for each sensing window (one if *e* is above *threshold*). Note that the model of the wireless medium consists of two bits: one that is set by the legitimate sender and one that is set by the adversary. The receiver reads energy if either bit is set. Once measurements for all sensing windows are done, the `checkHash()` function verifies if either the even or the odd sensing windows have an equal number of on and off slots.

```
receiver() {
  useClock();
  waitTicks(1, SKEW+1);
  short sw;
  for (sw : 0..(HASH_LENGTH*2-1)) {
    byte e = 0, ticks = 0;
    for (ticks : 0..(SW_MEASUREMENTS-1)) {
      e = e + (mediumSender || mediumAdversary);
      waitTicks(1, 1);
    }
    store(sw, e>THRESHOLD);
  }
  checkHash();
}
```

Listing 2.3: Receiver model.

Adversary The adversary is modelled as a simple process that increases the energy on the medium, then decreases it again. Because processes may be interleaved in any possible way, this verifies all scenarios.

```
proctype adversary() {
end:
  do
  :: mediumAdversary = 1;
     mediumAdversary = 0;
  od
}
```

Listing 2.4: Adversary model.

2.4 Model-Checking Results

Verification of the model means stating the assertion that either 1) the received hash is equal to the sent hash, or 2) the received hash is not equal to the sent hash, but the tampering by the adversary is detected (because the number of ones in the hash is unequal to the number of zeros). It is thus a search for a counter-example.

The expectation was that we might be able to find such a counter-example, but that the freedom with which an adversary could modify the received hash would be limited, probably to just the first or last bit. Model-checking indeed generated a counter-example. Moreover, experimentation with different assertions turned out that the adversary actually has more freedom in modifying the hash than expected. This vulnerability is described in Section 2.4.1. After the vulnerability was discovered, we executed a large series of Spin runs to discover what the exact conditions are that enable such an attack. The results are given in Section 2.4.2.

2.4.1 Revealed Vulnerability in the TEA

Model-checking the TEA model using Spin generated a counter-example to the assertion that a hash that was modified by an adversary will not be accepted by the receiver. In Figure 2.2, where no adversary is active, the even sensing windows still have the higher variance (1001 versus 0010). Thus, those are chosen as the correct slots and the hash 1001 is received, which is equal to the sent hash. In Figure 2.3 a scenario is shown in which an adversary actively introduces energy on the wireless medium. The energy that is introduced by the attacker is shown as a dotted blue line. He/she manages to trick the receiver into choosing the odd sensing windows and consequently receive a modified hash: 1010.

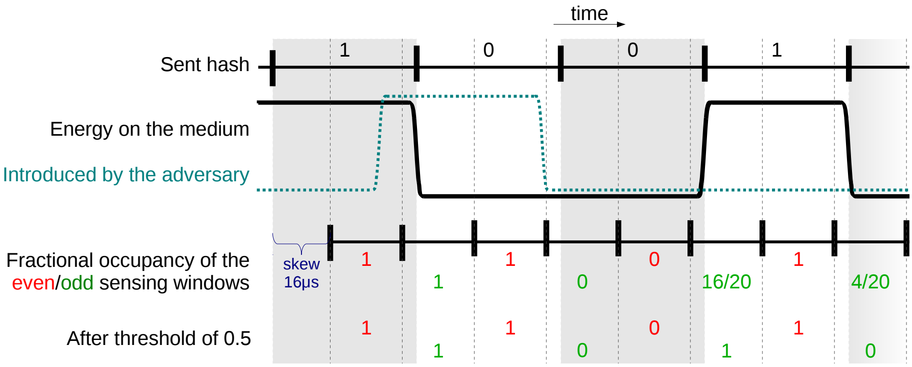


Figure 2.3: The attack found by model checking, for a 4-bit hash. The adversary introduces energy to the medium to change the received hash to 1010.

Experimentation with modified assertions has confirmed our conjecture that an adversary can use this tactic to change a 1 bit in the hash to a 0, *if and only if it is immediately followed by a 0*. Note, however, that the results an attacker can achieve are constrained by the original hash, since he/she can only change the value of *all* bits to the subsequent value *at the same time*. In case the hash starts with a 1-bit (i.e. in 50% of the cases), this modified hash has a single 0-bit extra. In that case, the attacker can place a 1-bit in a location of his/her choice.

2.4.2 Varying the Values of the Model Parameters

After discovering the vulnerability described in the previous section, we wanted to investigate what the exact circumstances are in which the vulnerability occurs. We therefore ran the SPIN model-checker for many different values of the parameters *hash_length*, *sw_measurements*, *threshold* and *skew*[2]. Some of the results are shown in Table 2.1. As it turns out, the length of the hash has no influence on the occurrence of the vulnerability, so this is omitted from the results. Remember that the unit for all three parameters in the table is clock-ticks.

It is obvious from Table 2.1 that the following predicate determines the possibility of an attack:

$$skew \geq sw\_measurements - threshold \tag{2.1}$$

[2] In order to run the SPIN model-checker for various values of defined parameters, we have implemented a small wrapper in the form a of C program. This wrapper can be obtained from `http://www.open.ou.nl/bvg/spinbatch/`.

$threshold = 3$		*skew*										
		0	1	2	3	4	5	6	7	8	9	10
sw_meas.	4	+	-	-	-	-	-	-	-	-	-	-
	5	+	+	-	-	-	-	-	-	-	-	-
	6	+	+	+	-	-	-	-	-	-	-	-
	7	+	+	+	+	-	-	-	-	-	-	-
	8	+	+	+	+	+	-	-	-	-	-	-
	9	+	+	+	+	+	+	-	-	-	-	-
	10	+	+	+	+	+	+	+	-	-	-	-

(a) Results for $threshold = 3$.

$threshold = 5$		*skew*										
		0	1	2	3	4	5	6	7	8	9	10
sw_meas.	6	+	-	-	-	-	-	-	-	-	-	-
	7	+	+	-	-	-	-	-	-	-	-	-
	8	+	+	+	-	-	-	-	-	-	-	-
	9	+	+	+	+	-	-	-	-	-	-	-
	10	+	+	+	+	+	-	-	-	-	-	-

(b) Results for $threshold = 5$.

$threshold = 7$		*skew*										
		0	1	2	3	4	5	6	7	8	9	10
sw_meas.	8	+	-	-	-	-	-	-	-	-	-	-
	9	+	+	-	-	-	-	-	-	-	-	-
	10	+	+	+	-	-	-	-	-	-	-	-

(c) Results for $threshold = 7$.

$threshold = 9$		*skew*										
		0	1	2	3	4	5	6	7	8	9	10
sw_meas.	10	+	-	-	-	-	-	-	-	-	-	-

(d) Results for $threshold = 9$.

Table 2.1: Model-checking results. Pluses indicate that the proposition is not broken. Minuses indicate the occurrence of the vulnerability.

In Figure 2.3, a threshold of 0.5 is used, which is represented by a value for *threshold* of $\frac{1}{2} \cdot$ *sw_measurements* in the model. If the *skew* is large enough to move a number of *threshold* measurements of the even windows over the sensing window boundary, then an adversary might change the received hash. We have model-checked the predicate for all combinations of *sw_measurements* 1–10, *threshold* 1–10 and *skew* 1–10.

2.5 Related Work

Before the Spin model on which this article is based was made, a simple model of the TEA and TEP in Uppaal was made by Drijvers [Dri12]. Uppaal is a tool with which properties about systems modelled as networks of timed automata can be verified [BLL$^+$96]. Because of the simple nature of this model, it did not include clock skew and therefore did not reveal the vulnerability that was later found using Spin. Apart from the TEA, Drijvers made a separate model of the overlying TEP, with which – under the assumption that the TEA is secure – no problems were identified. Since TEP was already successfully model-checked using Uppaal and, contrary to the TEA model, not in a highly abstract form (it is much simpler), we chose not to repeat the modelling for Spin.

In [BD98], a method is proposed for modelling a discrete clock in Promela, without the need to alter Spin. Instead of an alternating Boolean, time is modelled as an integer which negatively impacts the state space explosion. Just as with our approach a separate clock is introduced, which waits until all the other processes are finished, before increasing the discrete clock variable. This waiting is modelled with a native feature of PROMELA, which only continues if no other state-transition can be made (the `timeout` keyword). Therefore, all the processes are implicitly using the modelled clock. Because of our general adversary process, this restriction is too severe for us.

Many approaches to pairing wireless devices are described in the literature. A comparison of various wireless pairing protocols is given in [SVA07]. Often, a trusted *out-of-band* channel is used to transfer (the hash of) an encryption key, e.g. a human [KFR09], direct electrical contact [SA02], Near-Field Communication [MGH07], (ultra)sound [MG07], laser [MW07], visual/barcodes [SEKA06], et cetera. A nice overview is given in [KST$^+$09]. In TEP, a hash of the key is communicated in a trustedly secure manner *in-band*.

When using the PIN method that Wi-Fi Protected Setup provides, one of the devices displays an eight-digit authentication code, which the user then needs to enter on the other device. This method thus requires a screen and some sort of input device. The PIN method has been shown to be vulnerable to feasible brute-force attacks by Viehböck in [Vie11]. The reason for this is that the last digit is actually a check-sum of the first seven digits (i.e. there are only seven digits to verify) and, moreover, that the PIN is verified in two steps. The result of the verification of the first four digits is sent back to the enrollee, which may then send three more digits if this result was positive. This reduces the number of codes to try in a brute-force attack from 10^7 to $10^4 + 10^3$. A successful attack can be executed in approximately two hours on average. CERT-CC has urged users to disable the WPS feature on their wireless access points in response to this vulnerability[3]. A security and usability analysis of Wi-Fi Protected Setup, as well as Bluetooth Simple Pairing, which is similar, is given in [KWP07].

Approaches to model-checking security protocols are described in [Low98] and [ACC09]. In [ML08], a series of XSS and SQL injection attacks is detected using model-checking. Model-checking and theorem proving of security properties are discussed in [Mar98]. In [RA00], security issues that arise from combining hosts in a network are investigated using model-checking. An entire LINUX distribution is model-checked against security violations in [SCW$^+$05].

2.6 Conclusions

The effects of a number of decisions to be made when implementing Tamper-Evident Pairing have been studied. In particular, the sending of a hash by using on-off slots – in which energy is present or absent on the wireless medium – was modelled. The values of several essential parameters of the protocol have not been adequately specified. Model checking proved to be very effective both in uncovering a vulnerability for certain values of these parameters and in finding a predicate on the parameters indicating for which values the vulnerability

[3] `http://www.kb.cert.org/vuls/id/723755`

Future work could include extending the model to cover more of the TEA and constructing a full formal proof that the found vulnerability can only occur if the discovered predicate is satisfied. Furthermore, it would be interesting to investigate the feasability of exploiting the found vulnerability in practice. Unfortunately, there is currently no publicly available implementation of the protocol, so this would require significant software engineering.

Chapter 3

Improving Coverage of Test-Cases Generated by Symbolic PathFinder for Programs with Loops

Abstract. Symbolic execution is a program analysis technique that is used for many purposes, one of which is test-case generation. For loop-free programs, this generates a test-set that achieves path coverage. Program loops, however, imply exponential growth of the number of paths in the best case and non-termination in the worst case. In practice, the number of loop unwindings needs to be bounded for analysis.

We consider symbolic execution in the context of the tool Symbolic Pathfinder. This tool extends the model-checker Java Pathfinder and relies on its bounded state-space exploration for termination. We present an implementation of k-bounded loop unwinding, which increases the amount of user-control over the symbolic execution of loops.

Bounded unwinding can be viewed as a naive way to prune paths through loops. When using symbolic execution for test-case generation, naively pruning paths is likely at the cost of coverage. In order to improve coverage of branches within a loop body, we present a technique that semi-automatically concretises variables used in a loop. The basic technique is limited and we therefore present annotations to manually steer symbolic execution towards certain branches, as well as ideas on how the technique can be extended to be more widely applicable.

3.1 Introduction

Embedded software is hard to update in case errors are detected after release. Moreover, it is often used in safety-critical and mission-critical settings. It is therefore highly important to find errors during the development process. Testing is the most widely used technique for detecting faults in software. Software companies often dedicate over 50% of development time to testing. For safety-critical applications, this number is even larger. Composing an extensive set of test inputs is a complicated task, as the test-designer must achieve some form of *coverage*. For instance, statement coverage requires that all statements in the program have been executed at least once.

Symbolic execution is a well-known technique from program analysis, which can be used for test-case generation. In symbolic execution, a program is executed with symbols in place of concrete input. A *path-condition* is maintained, i.e., updated on each branch, that indicates the constraint under which this path is followed. Effectively, this means that if the generated constraints lie within the set of decidable theories, symbolic execution enumerates all paths through the software and the generated test-cases provide full path-coverage. However, in programs with loops, any extra iteration of a loop introduces a new path, introducing exponential growth in the number of paths. Moreover, in loops that depend on input values, the number of paths may be infinite (81.8% of loops in the applications studied in survey paper [XLXT13] by Xiao et al. are input-dependent). Therefore, in practice, symbolic execution has to be bounded. Since in general, it is impossible to know a priori how many iterations are needed to enter certain branches, this means that likely, any notion of coverage is lost.

We consider symbolic execution in the context of a specific tool, namely Symbolic Pathfinder (SPF) [PR10], which combines the model-checker Java Pathfinder [HP00] with symbolic execution and constraint solving to, among other objectives, generate test-cases. SPF currently implements bounding of the search-space that is explored by the model-checker, rather than bounded unwinding of loops. This means that the number of unwindings of loops is affected by the structure and complexity of the surrounding code. It is therefore hard to predict the number of unwindings for a particular loop, especially if other loops are present. We present the implementation of a more flexible and intuitive bounding mechanism for loops: k-bounded unwinding. Using this algorithm makes it possible to unwind the same number of iterations for each loop. Additionally, we present an annotation to specify k-bounds that are loop-specific.

Both types of bounding (loop bounding and search space bounding) may cause important paths through a program to be missed by SPF. When used to generate a set of test-cases, this means that the test-set will not cover all branches in the loop body. Paths are pruned by naively dropping all with more than k iterations, which can make it very complicated to achieve high coverage. We strive to improve the *object branch coverage* of the set of test-cases generated by SPF.

Definition 1. *A set of test-inputs provides* object branch coverage *if running the program for those test-inputs executes every branch in the object code level control-flow graph.*

Since in the object code level control-flow graph (CFG), evaluation of a condition such as $b1 \wedge b2$ amounts to two CFG nodes, as opposed to a single node in the source code level CFG, object branch coverage implies branch coverage.

Object branch coverage is thus a more rigorous coverage metric than source-level branch coverage.

We present an annotation which can be used to concretise symbolic variables. This can be used to fix symbolic variables to a set of concrete values that cover all branches in the loop body. Furthermore, we present a technique which can infer these cases for loops that are independent of context. The approach has limitations, but represents a step in improving the object branch coverage of the generated test-set. Our contribution is thus threefold:

1. An implementation of k-bounded unwinding of loops in SPF.
2. Annotations to concretise variables upon loop entry.
3. An experimental method to semi-automatically infer such annotations, based on out-of-context symbolic execution of the loop body.

We have implemented k-bounding and concretisation using Java annotations in Java Pathfinder. The source code for this extension can be found here: `http://www.cs.ru.nl/R.Kersten/jpf-symbc-loops.tar.gz`.

3.1.1 Related Work

Symbolic execution for software testing is surveyed in [CGK$^+$11]. Various extensions to classical symbolic execution and state-of-the-art tools are discussed.

Verification of program properties using SPF is discussed in [PV04]. Loops are handled using invariants. Verification of a post-condition of a loop can be simplified to verification of 1) the loop invariant before executing the loop (base case), 2) the loop invariant after a generic iteration using symbolic execution (inductive case) and 3) implication of the post-condition from the invariant. The first two steps prove that the loop invariant is correct. The third step proves the post-condition follows from the invariant.

Gladisch describes a method to generate a test-set with full feasible branch coverage, using the theorem prover KeY in [Gla08]. It requires that strong preconditions, post-conditions and loop invariants are supplied and leverages the theorem prover to replace symbolic execution of a loop by the application of a loop invariant rule. Several types of preconditions are formed for loops, which guarantee the execution of all branches in and after the loop.

Trtík presents a technique for handling loops in symbolic execution in [Trt13]. He introduces *path counters* with update paths (increment by one) and reset paths (set to zero). Symbolic values of program variables can then be expressed in terms of these counters. This theoretically tackles the problem in part, but more complex loops quickly result in non-linear constraints which are expensive to solve or even undecidable.

A survey on loop problems for *dynamic* symbolic execution (DSE) is given in [XLXT13]. DSE executes the program using concrete random inputs and collects the path condition on the side. An interesting result is that 81.8% of loops in the studied applications are dependent on input, thus possibly non-terminating.

The most common way to deal with this type of loops is bounded iteration, solving the termination problem at the cost of completeness of the generated test-set. Search-guiding heuristics can be used to guide symbolic execution to certain "interesting" paths, making the pruning less naive. A more complex approach is to create loop summaries: a set of formulas based on loop invariants and induction variables that summarises the effects of the loop. This is a complex task which is infeasible for many loops. In fact, the state-of-the-art loop summarising algorithm presented in [GL11] can only summarise 6 out of 19 input-dependent loops in their experiment. Single-path symbolic execution is a variation of DSE, in which a set of executions which follow the same control-flow path is considered. It is extended with a mechanism for loops in [SPMS09]. Iteration counters named *trip count variables* are introduced that can be linked to a known input grammar. It is shown that this is a powerful tool for finding problems such as buffer overflow vulnerabilities.

3.2 Bounding Loops in SPF

In this section we present our implementation of k-bounded unwinding of loops in SPF. Consider the method in Listing 3.1. As i is assigned a symbolic value, symbolic execution of this program iterates the loop infinitely many times. In practice, symbolic execution is bounded. SPF currently does not implement bounding itself, but instead relies on the bounded state-space exploration implemented in Java Pathfinder. This means that the number of unwindings of loops is affected by the structure and complexity of the surrounding code.

```
boolean m(int i) {
  boolean b = false;
  while (i > 0) {
    if (i==10)
      b = true;
    i--;
  }
  return b;
}
```

Listing 3.1: Example program with a loop.

To cover all branches, it is desired to unwind the loop in Listing 3.1 at least 10 times. Say we have a simple `main` method that initialises the object and then calls this method on it. If we set the depth-limit on the number of explored states to 4, the loop will be unwound only once, because of the other states on the path related to calling the method from the `main` function. The depth bound is based on the number of `ChoiceGenerator` objects that are encountered by Java Pathfinder. It cannot distinguish between choices related to a loop or other points of non-determinism. It gets harder to estimate the number of unwindings when there are other choice points on the path. Say, if there was a single if-statement after the loop, a depth-limit of 5 would be needed to unwind the loop once. Moreover, the number of choice points may differ between paths. Therefore, a different number of iterations might be unwound for the same loop in different paths. This makes it very hard to control the number of unwindings that will actually be done for a given loop.

3.2.1 K-Bounded Unwinding

We implemented k-bounded unwinding in SPF, in a listener called the `KBoundedSearchListener`. When this class is initialised, the CFG is built and the dominance set is calculated, in order to detect headers and back-edges of loops (a loop header is the single point of entry into the loop). A node n_1 dominates a node n_2 if all paths from the entry node to n_2 go through n_1. If an edge exists in the CFG from n to h and h dominates n, then the edge is the back-edge of a loop with header h. Note that there may be several back-edges into the same loop header. This loop detection algorithm is implemented in the `LoopFinder` class. Headers are stored in objects of the `Location` class, which combines a method name and an instruction position. The instructions of the choice points following the loop headers are also stored, because that is where the actual branching occurs (loop headers typically consist of a load instruction).

The listener then counts the number of unwindings of each loop, using a stack, by listening for registered `ChoiceGenerator` objects. These are objects that Java Pathfinder uses to navigate over decisions, where paths branch. When the k-bound is reached for a certain loop, the remaining paths through this loop are pruned by setting the next `ChoiceGenerator` to done.

Bounded unwinding is activated by adding the `KBoundedSearchListener` to the Java Pathfinder configuration and setting the configuration option `kbound=K`, where `K` is the maximal number of unwindings. The implementation currently is limited to intra-procedural analysis (only loops within the analysed method itself are detected and bound, not those in called methods). An inter-procedural version will be implemented in the near future.

3.2.2 Specifying Loop-Specific Bounds

In some cases, one might want a certain loop to be unwound more than others, or maybe it is clear that unwinding it only once is enough. For those cases we have added an annotation to express loop-specific bounds. It is added to the header of a Java method and has the following syntax:

$$@KBound(k = \{\text{“}N_1 : b_1\text{”}, \ldots, \text{“}N_n : b_n\text{”}\})$$

where each N_i is a loop identifier, determined by the order of loop headers from the top of the method in its source code, and each b_i is an integer bound. For instance, to limit unwinding of the second loop in a method to 1, the following annotation can be used:

$$@KBound(k = \{\text{“}2 : 1\text{”}\})$$

3.3 Concretising Loop Variables

Typically, when symbolically executing a loop, up to a fixed number of k iterations are unwound and the path condition includes propositions expressing the number of unrolled iterations. For example $i > 0 \wedge i - 1 > 0 \wedge i - 2 \leq 0$ signifies two unrolled iterations for the example in Listing 3.1. As the number of iterations of loops is potentially infinite, a selection of paths through loops needs to be pruned. Unwinding of loops up to a given bound can often be ineffective in achieving our testing goals, e.g. object branch coverage.

Since we are considering symbolic execution in the context of test-case generation, we are interested in obtaining test-sets with better coverage. We therefore propose to prune paths through loops based on object branch coverage of the loop body. Our technique is inspired by [PV04], in which a loop body is symbolically executed with fresh symbols in order to prove a loop invariant. It consists of the following steps:

1. Symbolically execute the loop body out-of-context, i.e., with fresh symbols.
2. Solve the generated path conditions to obtain a *model* for each of them.
3. Concretise the values of the variables by adding the concrete values to the path condition (e.g., for models $i = 0$ and $i = 1$ we add $i = 0 \vee i = 1$).

Thanks to using *fresh* symbols for program variables, the search will not be biased to their symbolic values before entering the loop. The result of step 1 is a set of path conditions, capturing all behaviours that one iteration of the loop can exhibit. Models for these path conditions can then be found using off-the-shelf constraint solvers. By concretising the symbolic variables to these models, we prune all other paths, making the search-space finite. Concretisation can thus replace other bounding methods. Note, however, that all iterations corresponding to the bound will need to be unrolled. For instance, if for our running example, a model m is found, the loop needs to be unrolled m times.

Only variables that are used in the loop body or loop guard should be concretised. Otherwise, symbolic execution will settle on a limited set of paths through the entire program. These are also the only variables that need to be fresh for the out-of-context symbolic execution. Variables for which the model is not constrained by the path condition will also not be concretised, as these do not influence the flow of control in the loop.

3.3.1 Example

Consider the Java method in Listing 3.1. If regular k-bounded unwinding is applied to this loop with $k < 10$, not all paths are explored. For example, with

$k = 2$, the following 3 path conditions are generated:

$$i \leq 0 \tag{3.1a}$$

$$i > 0 \land i \neq 10 \land i - 1 \leq 0 \tag{3.1b}$$

$$i > 0 \land i \neq 10 \land i - 1 > 0 \land i - 1 \neq 10 \land i - 2 \leq 0 \tag{3.1c}$$

Using the YICES solver, we get models $i = 0$, $i = 1$ and $i = 2$. The branch where b is assigned the value *true* is missed. Let us now take the loop body out of context. A new JAVA method containing the extracted body is shown in Listing 3.2. The `while` statement has been replaced by an `if`, because we want to consider only the loop body, but still want the resulting models to satisfy the loop guard (except for the single model that we also need in which the loop is not entered at all).

```
void outofcontext(int i, boolean b) {
  if (i > 0) {
    if (i==10)
      b = true;
    i--;
  }
}
```

Listing 3.2: The loop body from Listing 3.1, taken out of context.

Symbolic execution of this extracted loop body results in the following path conditions:

$$i \leq 0 \tag{3.2a}$$

$$i > 0 \land i \neq 10 \tag{3.2b}$$

$$i > 0 \land i = 10 \tag{3.2c}$$

Using the YICES solver, we get models $i = 0$, $i = 9$ and $i = 10$. Because there are no statements before the loop in our program, we can simply use these symbols in the path condition as-is. In general, a mapping to the original symbols is needed, which is explained in Section 3.3.2.

The paths through the loop can now be pruned in a more informed manner which enables object branch coverage by adding the models to the path condition. One can think of this as adding the following assumption before the loop:

```
assume (i==0 || i==9 || i==10);
```

Note that using the path conditions instead of the models would not prune the paths. A k-bound with $k \geq 10$ would still be needed to cover all branches.

3.3.2 Annotations in SPF

Whether determined by out-of-context symbolic execution or by hand, the models for concretisation of loop variables can be expressed with a special annotation.

It is added to the header of a Java method and has the following syntax:

$$@UseModels(models = \{C_1, \ldots, C_k\})$$

where each C_x represents a concretisation string:

$$C_x := \text{``}N_x.[v_1^x, \ldots, v_j^x] \rightarrow [m_1^x, \ldots, m_j^x]\text{''}$$

where N_x is the identifier of a loop (determined by the order of loop headers from the top of the method in its source code), v_a^x is a program variable to be concretised and m_a^x the model to concretise it to. In each of the k concretisation strings, several variables may be concretised that might differ from the other concretisation strings. When only a single variable and model combination is used, brackets may be omitted. As an example, to concretise i to 20 in the first loop of a method one can use the following annotation:

$$@UseModels(models = \{\text{''}1.i \rightarrow 20\text{''}\})$$

When concretising, an equality between the value specified in the model and the symbolic value of the program variable *upon entry to the loop* is added to the path condition. In other words, if the value of a program variable i is $i+3$ before the loop and there is an annotation that i should be concretised to 20, then $20 = i + 3$ is added to the path condition. When symbolically executing nested loops, the variables in the inner loop are concretised and used in the subsequent out-of-context symbolic execution of the outer loops.

3.3.3 Limitations

The concretisation approach to handle loops has two major limitations. We discuss these here, including ideas on how we intend to address them in the future. Given these limitations, it is recommended to use the loop concretisation technique for test-case generation in combination with a coverage checker. Such a tool can check if the test-set achieves object-branch coverage and point to branches that are missed. The user can then go back and add annotations to direct SPF to improve the generated test-set.

Context-dependence. Out-of-context symbolic execution finds models for the out-of-context loop body. In cases such as the running example of this chapter, adding these models to the path condition achieves object branch coverage, because the execution of the loop does not depend on this context. However, when the execution of the loop-body is dependent on the context, the models may be infeasible and the search will back-track. Consider, for instance, the loop in Listing 3.3. This method is taken from the Java prototype of the Airborne Coordinated Conflict Resolution and Detection (ACCoRD) framework developed and maintained by the NASA Langley formal methods group[4]. This is a framework for formal specification and verification of state-based airspace separation assurance algorithms.

[4] `http://shemesh.larc.nasa.gov/people/cam/ACCoRD/`

This specific method estimates the change of vertical speed from a sequence of velocity vectors stored in the containing object. The `numPtsVsRateCalc` parameter specifies the number of data points used in the calculation of the average and the method returns the vertical acceleration. The sign of the return value indicates the direction of the acceleration.

```java
 1 public double avgVsRate(int numPtsVsRateCalc) {
 2   int n = size();
 3   if (numPtsVsRateCalc < 2) numPtsVsRateCalc = 2;
 4   int numPts = Math.min(numPtsVsRateCalc,n);
 5   double vsLast = 0;
 6   double tmLast = 0;
 7   double vsRateSum = 0.0;
 8   for (int i = n-1; i > n-numPts-1 && i >= 0; i--) {
 9     StateVector svt = get(i);
10     double vs = svt.v().vs();
11     double tmTr = time(i);
12     if (i < n-1) {
13       double vsRate = (vs-vsLast)/(tmTr-tmLast);
14       vsRateSum = vsRateSum + vsRate;
15     }
16     vsLast = vs;
17     tmLast = tmTr;
18   }
19   if (numPts < 2) return 0;
20   else return vsRateSum/(numPts-1);
21 }
```

Listing 3.3: Loop for which its execution is dependent on its context, taken from the ACCoRD conflict resolution and detection framework.

If we analyse the body of the loop at lines 8–18 out-of-context, we get the following models (each row corresponds to a path through the loop body; other variables are omitted because they do not influence the control-flow in the loop and are therefore not concretised):

`i`	`n`	`numPts`
47	89	97
0	0	0
-24	-43	89
-77	86	92

The problem with these models is that the value of `numPts` is defined as the minimum of `numPtsVsRateCalc` and `n` on line 4, but none of these models except for the one with all zeros satisfy the consequential constraint `numPts` $\leq$ `n`. Furthermore, the path condition upon entry to the loop will contain a constraint

$\texttt{i} = \texttt{n} - 1$, as this is what `i` is initialised to on line 8. This constraint is also not satisfied by any of the models. Therefore, when we add the constraints for these models to the path condition, symbolic execution will find none of the paths through the loop feasible.

We intend to create a refinement loop, which iteratively refines the constraint to for which a model must be found. The constraint is first set to the path condition of the path we are working on. The model that is found by the solver is then checked against the path conditions of the paths leading to the loop. If all of these conflict with the model, the constraint is strengthened with the conflicting sub-constraint of the path conditions. Complexity lies in finding this conflicting sub-constraint. Furthermore, we will introduce an annotation to specify whether or not loop variable concretisation should be used.

Iteration-count dependence. Consider the example in Listing 3.4. There is only a single path through the loop body. Its path condition is $i > 0$ and a model is $i = 1$. When considering only this path, the path where the method does "something" (line 8) is missed.

```
 1 void m(int i) {
 2   int j = 0;
 3   while (i > 0) {
 4     j++;
 5     i--;
 6   }
 7   if (j==20) {
 8     //do something
 9   }
10 }
```

Listing 3.4: Loop which shows dependence on iteration count.

In this case, the problem can be solved by using a manual annotation that states that i should be concretised to 20. There may also be branching in the loop that only occurs for a particular iteration, the conditional on line 12 of Listing 3.3 is an example of this. The condition used in this case is true for any path, except for the first one. Such a case may be solved by setting a k-bound that is high enough.

The general case, where a branch may occur after n iterations, is equivalent to the halting problem. However, this does not mean that certain cases may not be tackled. An idea to improve this, is to add the iteration count as a variable to symbolic execution or detect induction variables, as is done e.g. in [Trt13] and [GL11]. The symbolic value of variables can then be expressed using the number of iterations of each loop.

3.4 Conclusions

We have presented a series of improvements to the loop-handling capabilities of Symbolic Pathfinder:

- An implementation of k-bounded unwinding of loops.
- The loop concretisation technique, which symbolically executes the loop body out-of-context, then concretises the variables to the found models.
- Novel annotations to direct symbolic execution towards branches that may otherwise be missed.

The loop concretisation technique has two major limitations: 1. when loops are context-dependent, and 2. when program variables are dependent on the number of loop iterations. We suggest some extension ideas to address these limitations.

The problem of pruning paths through loops is a notoriously hard one. A large body of literature on the topic exists and no proposed solution is complete. Our work represents a series of small steps towards better treatment of loops.

Future Work. We will develop the ideas discussed in Section 3.3.3 of this chapter and implement them in SPF. Furthermore, in [PDEP08, MPR12, RPB12, BPRT13], an extension to SPF is discussed that compares software versions and generates test-cases for paths that are impacted by changes only. This *incremental* analysis implies a significant reduction in the number of generated test-cases. Extensions of our method that are specific for incremental analysis can potentially reduce this number even further and improve object branch coverage.

Chapter 4

A Hoare Logic for Energy Consumption Analysis

Abstract. Energy inefficient software implementations may cause battery drain for small systems and high energy costs for large systems. Dynamic energy analysis is often applied to mitigate these issues. However, this is often hardware-specific and requires repetitive measurements using special equipment.

We present a static analysis deriving upper-bounds for energy consumption based on an introduced energy-aware Hoare logic. Software is considered together with models of the hardware it controls. The Hoare logic is parametric with respect to the hardware. Energy models of hardware components can be specified separately from the logic. Parametrised with one or more of such component models, the analysis can statically produce a sound (over-approximated) upper-bound for the energy-usage of the hardware controlled by the software. The analysis is implemented in the tool ECAlogic.

4.1 Introduction

Power consumption and green computing are nowadays important topics in IT. From small systems such as wireless sensor nodes, cell-phones and embedded devices to big architectures such as data centres, mainframes and servers, energy consumption is an important factor. Small devices are often powered by a battery, which should last as long as possible. For larger devices, the problem lies mostly with the costs of powering the device. These costs are often amplified by inefficient power-supplies and cooling of the system.

Obviously, power consumption depends not only on hardware, but also on the software *controlling* the hardware. Currently, most of the methods available to programmers to analyse energy consumption caused by software use dynamic analysis: measuring while the software is running. Power consumption measurement of a system and especially of its individual components is not a trivial task.

A designated measuring set-up is required. This means that most programmers currently have no idea how much energy their software consumes. A static analysis of energy consumption would be a big improvement, potentially leading to more energy-efficient software. Such a static analysis is presented in this chapter.

Since the software interacts with multiple components (software and hardware), energy consumption analysis needs to incorporate different kinds of analysis. Power consumption may depend on hardware state, values of variables and bounds on the required number of clock-cycles.

Related Work There is a large body of work on energy-efficiency of software. Most papers approach the problem on a high level, defining programming and design patterns for writing energy-efficient code (see e.g. [Alb10, Sax10, Ran10]). In [tBMB$^+$13], a modular design for energy-aware software is presented that is based on a series of rules on UML schemes. In [CZSL12] and [SDF$^+$11], a program is divided into "phases" describing similar behaviour. Based on the behaviour of the software, design level optimisations are proposed to achieve lower energy consumption. A lot of research is dedicated to building compilers that optimise code for energy-efficiency, e.g. in GCC [ZBA$^+$09] or in an *iterative* compiler [GCB05]. Petri-net based energy modelling techniques for embedded systems are proposed in [JNM$^+$06] and [NMT$^+$11].

Analyses for consumption of generic resources are built using recurrence relation solving [AAG$^+$08], amortised analysis [HAH11], amortisation and separation logic [Atk10] and a Vienna Development Method style program logic [ABH$^+$07]. The main differences with our work are that we include an explicit hardware model and a context in the form of component states. This context enables the inclusion of state-dependent energy consumption.

Relatively close to our approach are [JML06] and [KLG$^+$13], in which energy consumption of embedded software is analysed for specific architectures ([JML06] for SimpleScalar, [KLG$^+$13] for XMOS ISA-level models), while our approach is hardware-parametric. Several tools perform a static analysis of the energy-consumption of the CPU based on per-instruction measurements, such as JouleTrack [SC01] and Wattch [BTM00]. Furthermore, tools exist for energy profiling of software libraries, i.e. using dynamic analysis [KZ08]. SEProf is an advanced tool that combines dynamic profiling with static estimation of energy consumption [TC12]. One difference is that, while our analysis is geared towards complete systems, SEProf only estimates the energy usage of the CPU. Moreover, while SEProf *estimates* energy-usage, our analysis gives *bounds* that are sound with respect to the hardware model.

In [tMB$^+$14], an abstraction of the resource behaviour of components is presented, called Resource-Utilisation Models (RUMs). Our component models can be viewed as an instantiation of a RUM. RUMs can be analysed, e.g., with the model checker Uppaal. A possible future research direction is to find a way to analyse also algorithms with RUMs as component models.

Our Approach Contrary to the approaches above, we are interested in statically deriving bounds on energy-consumption using a novel, generic approach that is parametrised with hardware models. Energy consumption analysis is an instance of resource consumption analysis. Other instances are worst-case execution time [WEE$^+$08], size [SvEvK09], loop bound [SKVE10, KvGS$^+$14] and memory [KvGS$^+$14] analysis). The focus of this chapter is on energy analysis. Energy consumption models of hardware components are input for our analysis. The analysis requires information about the software, such as dependencies between variables and information about the number of loop iterations. For this reason we assume that a previous analysis (properly instantiated for our case) has been made deriving loop bounds (e.g. [PR04, KvGS$^+$14]) and variable dependency information (e.g. [HTS08]).

Our approach is essentially an energy-aware Hoare logic that is proven sound with respect to an energy-aware semantics. Both the semantics and the logic assume energy-aware component models to be present. The central control is however assumed to be in the software. Consequently, the analysis is done on a hybrid system of software and models of hardware components. The Hoare logic yields an upper bound on the energy consumption of a system of hardware components that are controlled by software. It is implemented in the tool ECALOGIC.

Our contribution The main contributions of this chapter are:

- A novel hardware-parametric energy-aware software semantics.
- A corresponding energy-aware Hoare logic that enables formal reasoning about energy consumption such as deriving an upper-bound for the energy consumption of the system.
- A soundness proof of the derived upper-bounds with respect to the semantics.
- An implementation of the analysis in the tool ECALOGIC.

The basic modelling and semantics are presented in Section 4.2. Energy-awareness is added and the logic is presented in Section 4.3. An example is given in Section 4.4 and the soundness proof is outlined in Section 4.5. Implementation of the analysis in the tool ECALOGIC is discussed in Section 4.6. The chapter is concluded in Section 4.7.

4.2 Modelling Hybrid Systems

Most modern electronic systems consist of hardware and software. In order to study the energy consumption of such hybrid systems we will consider both hardware and software in one single modelling framework. This section defines a hybrid logic in which software plays a central role controlling hardware components. The hardware components are modelled in such a way that only the relevant information for the energy consumption analysis is present. In this chapter, the controlling software is assumed to be written in a small language designed just for illustrating the analysis.

4.2.1 Language

Our analysis is performed on the simple ‘while’ language ECA. The grammar for our language is defined as follows (where $\boxdot \in \{+, -, *, >, \geq, =, \neq, \leq, <, \wedge, \vee\}$):

$$\begin{aligned}
c \in \textsc{Const} &= 0 \mid 1 \mid 2 \mid 3 \mid \ldots \\
id \in \textsc{Ident} &= \text{'a'} \mid \text{'A'} \mid \text{'b'} \mid \text{'B'} \mid \ldots \mid \text{'z'} \mid \text{'Z'} \mid id_1 id_2 \\
x \in \textsc{Var} &= id \\
f \in \textsc{FuncName} &= id \\
C \in \textsc{Component} &= C_{id} \\
e \in \textsc{Expr} &= c \mid x \mid e_1 \boxdot e_2 \mid C :: f(e_1) \mid f(e_1) \mid S, e_1 \\
S \in \textsc{Statement} &= \textbf{skip} \mid S_1; S_2 \mid e \mid x := e_1 \mid \textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end if} \\
&\quad \mid \textbf{while } e \textbf{ do } S \textbf{ end while} \mid F \\
F \in \textsc{Func} &= \textbf{function } f(x) \textbf{ begin } e \textbf{ end}
\end{aligned}$$

This language is used just for illustration purposes, so the only supported type in the language is unsigned integer. There are no explicit Booleans. The value 0 is handled as a **False** value, while all the other values are handled as a **True** value. There are no global variables and parameters are passed by-value, so functions do not have side-effects on the program state. Furthermore, `while` loops are supported but recursion is not. Functions are statically scoped and can be defined anywhere in the program, since they are statements. There are explicit statements for operations on hardware components, like the processor, memory, storage or network devices. By explicitly introducing these statements it is easier to reason about those components, as opposed to, for instance, using conventions about certain memory regions that will map to certain hardware devices. Functions on components have a fixed number of arguments and always return a value. The notation $C_i :: f$ will refer to a function f of a component C_i.

4.2.2 Modelling Components

To reason about hybrid systems we need a way to model hardware components (e.g. memory, hard-disk, network controller) that captures the behaviour of those components with respect to resource consumption. Hence, we introduce a *component model* that consists of a state and a set of functions that can change the state: *component functions*. A component state $C_i :: s$ is a collection of variables of any type. They can signify e.g. that the component is on, off or in stand-by.

A component function is modelled by a function that produces the return value (rv_f) and a function that updates the internal state of the component (δ_f). Both functions are functions over the state variables. The update function $C_i :: \delta_f$ and the return value function $C_i :: rv_f$ take the state s and the arguments *args* passed to the component function and return respectively the new state of the component and the return value. Each component C_i may have multiple component functions. All the state changes in components must be explicit in the source code as an operation, a *component function*, on that specific component.

4.2.3 Semantics

Standard, non-energy-aware semantics can be defined for our language. Full semantics are given in a technical report [PKvv13]. Below, the assignment rule (*sAssign*) and the component function call rule (*sCallCmpF*) are given to illustrate the notation and the way of handling components. The rules are defined over a triple $\langle e, \sigma, \mathit{\Gamma} \rangle$ with respectively a program statement S or expression e, the program state function σ and the component state environment $\mathit{\Gamma}$. The *program* state function returns for every variable its actual value. $\mathit{\Delta}$ is an environment of function definitions. We use the following notation for substitution: $\sigma[x_i \leftarrow n]$. The top-right of the (*sCallCmpF*) thus shows the calculation of $\mathit{\Gamma}'$ from $\mathit{\Gamma}$. With $C_i^{\mathit{\Gamma}} :: s$ we mean the state of component C_i in $\mathit{\Gamma}$. The reduction symbol $\Downarrow^E$ is used for expressions, which evaluate to a value, a new state function and a new component state environment. We use $\Downarrow^S$ for statements, which only evaluate to a new state function and component state environment.

$$\frac{\mathit{\Delta} \vdash \langle e, \sigma, \mathit{\Gamma} \rangle \Downarrow^E \langle n, \sigma', \mathit{\Gamma}' \rangle}{\mathit{\Delta} \vdash \langle x_1 := e, \sigma, \mathit{\Gamma} \rangle \Downarrow^S \langle \sigma'[x_1 \leftarrow n], \mathit{\Gamma}' \rangle} \text{ (sAssign)}$$

$$\frac{\begin{array}{l} \mathit{\Delta} \vdash \langle e, \sigma, \mathit{\Gamma} \rangle \Downarrow^E \langle a, \sigma', \mathit{\Gamma}' \rangle \\ C_i :: rv_f(C_i^{\mathit{\Gamma}} :: s, a) = n \qquad \mathit{\Gamma}' = \mathit{\Gamma}[C_i :: s \leftarrow C_i :: \delta_f(C_i^{\mathit{\Gamma}} :: s, a)] \end{array}}{\mathit{\Delta} \vdash \langle C_i :: f(e), \sigma, \mathit{\Gamma} \rangle \Downarrow^E \langle n, \sigma', \mathit{\Gamma}' \rangle} \text{ (sCallCmpF)}$$

In the following sections we will define energy-aware semantics and energy analysis rules. We used a consistent naming scheme for the different variants of the rules (e.g. **s**Assign, **e**Assign and **a**Assign for the Assignment rule in respectively the **s**tandard non-energy-aware semantics, the **e**nergy aware semantics and the energy **a**nalysis rules).

4.3 Energy Analysis of Hybrid Systems

In this section we extend our hybrid modeling, in order to reason about the energy consumption of programs. We distinguish two kinds of energy usage: *incidental* and *time-dependent*. The former represents an operation that uses a constant amount of energy, disregarding any time aspect. The latter signifies a change in the state of the component; while a component is in a certain state it is assumed to draw a constant amount of energy *per time unit*.

4.3.1 Energy-Aware Semantics

As energy consumption can be based on time, we first need to extend our semantics to be time-aware. We effectively extend all the rules of the semantics with an extra argument, a global timestamp t. Using this timestamp we are able to model and analyse time-dependent energy usage.

We track energy usage for each component individually, by using an accumulator $\mathfrak{e}$ that is added to the component model. For time-dependent energy usage, with each component state change, the energy used while the component was in the previous state is added to the accumulator. To enable calculation of the time spent in the current state, we add τ to the component model, signifying the timestamp at which the component entered the current state. We assume that each component has a constant *power draw* while in a state. Therefore, the component model function $C_i :: \phi(s)$ maps component states onto the corresponding power draw, independent of time. To calculate the power consumed while in a certain state we define the *td* function, with as arguments the component and the current timestamp:

$$td(C_i, t) = C_i :: \phi(s) \cdot (t - C_i :: \tau)$$

We model *incidental energy usage* associated with a component function f with the constant $C_i :: \mathfrak{E}_f$. For each call to a component function we add this constant to the energy accumulator.

A component function call can influence energy consumption in two ways: through its associated incidental energy consumption and by changing the state, thereby influencing time-dependent energy usage. This is expressed by energy-aware semantic rule (*eCallCmpF*) for component functions as defined below, with $C_i :: \mathfrak{T}_f$ representing the time it costs to execute this component function.

$$\frac{\Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^E \langle a, \sigma', \Gamma', \mathfrak{t}'\rangle \qquad C_i :: rv_f(C_i^{\Gamma'} :: s, a) = n \\ \Gamma'' = \Gamma[C_i :: \mathfrak{e} \mathrel{+}= C_i :: \mathfrak{E}_f + td(C_i^{\Gamma'}, \mathfrak{t}), C_i :: s \leftarrow C_i :: \delta_f(C_i^{\Gamma'} :: s, a), C_i :: \tau \leftarrow \mathfrak{t}']}{\Delta \vdash \langle C_i :: f(e), \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^E \langle n, \sigma', \Gamma'', \mathfrak{t}' + C_i :: \mathfrak{T}_f\rangle} \text{(eCallCmpF)}$$

Note the addition of the incidental and time dependent energy usages ($C_i :: \mathfrak{E}_f$ and $td(C_i^{\Gamma}, \mathfrak{t})$ respectively) to the energy accumulator $C_i :: \mathfrak{e}$, the increment of the global time with $C_i :: \mathfrak{T}_f$ and the update of the component timestamp $C_i :: \tau$. Evaluation by $\Downarrow$ in the energy-aware semantics extends the original semantics with a timestamp and an energy accumulator, which are used to calculate the total energy consumption of the evaluation ($\mathfrak{e}_{system}$ as defined below). The full energy-aware semantics are given in Figure 4.1.

The energy accumulator of the components is not always up to date with respect to the current time, as it is only updated in the (*eCallCmpF*) rule. This is done for simplicity; otherwise each rule that adjusts the global time needs to update the energy accumulator of all components.

To calculate the total actual energy usage, the time the components are in their current state should still be accounted for. This means we have to add the result of the *td* function for each component. The total energy consumption of the system can be calculated at any time as follows:

$$\mathfrak{e}_{system}(\Gamma, \mathfrak{t}) = \sum_i C_i^{\Gamma} :: \mathfrak{e} + td(C_i^{\Gamma}, \mathfrak{t})$$

$$\frac{}{\Delta \vdash \langle c, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{E} \langle c, \sigma, \Gamma, \mathfrak{t}\rangle}\ (\text{eConst}) \qquad \frac{}{\Delta \vdash \langle x, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{E} \langle \sigma(x), \sigma, \Gamma, \mathfrak{t}\rangle}\ (\text{eVar})$$

$$\frac{\begin{array}{cc} \Delta \vdash \langle e_1, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{E} \langle n, \sigma', \Gamma', \mathfrak{t}'\rangle & C_{imp} :: \boxdot(n, m) = p \\ \Delta \vdash \langle e_2, \sigma', \Gamma', \mathfrak{t}'\rangle \Downarrow^{E} \langle m, \sigma'', \Gamma'', \mathfrak{t}''\rangle & \Gamma''' = \Gamma''[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_e] \end{array}}{\Delta \vdash \langle e_1 \boxdot e_2, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{E} \langle p, \sigma'', \Gamma''', \mathfrak{t}'' + C_{imp} :: \mathfrak{T}_e\rangle}\ (\text{eBinOp})$$

$$\frac{\begin{array}{c} \Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{E} \langle a, \sigma', \Gamma', \mathfrak{t}'\rangle \qquad C_i :: rv_f(C_i^{\Gamma'} :: s, a) = n \\ \Gamma'' = \Gamma[C_i :: \mathfrak{e} \mathrel{+}= C_i :: \mathfrak{E}_f + td(C_i^{\Gamma'}, \mathfrak{t}), C_i :: s \leftarrow C_i :: \delta_f(C_i^{\Gamma'} :: s, a), C_i :: \tau \leftarrow \mathfrak{t}'] \end{array}}{\Delta \vdash \langle C_i :: f(e), \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{E} \langle n, \sigma', \Gamma'', \mathfrak{t}' + C_i :: \mathfrak{T}_f\rangle}\ (\text{eCallCmpF})$$

$$\frac{\begin{array}{cc} & \Delta(f) = (e_1, \Delta', x) \\ \Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{E} \langle a, \sigma', \Gamma', \mathfrak{t}'\rangle & \Delta' \vdash \langle e_1, [x \leftarrow a], \Gamma', \mathfrak{t}'\rangle \Downarrow^{E} \langle n, \sigma'', \Gamma'', \mathfrak{t}''\rangle \end{array}}{\Delta \vdash \langle f(e), \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{E} \langle n, \sigma', \Gamma'', \mathfrak{t}''\rangle}\ (\text{eCallF})$$

$$\frac{\Delta \vdash \langle S, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{S} \langle \sigma', \Gamma', \mathfrak{t}'\rangle \qquad \Delta \vdash \langle e, \sigma', \Gamma', \mathfrak{t}'\rangle \Downarrow^{E} \langle n, \sigma'', \Gamma'', \mathfrak{t}''\rangle}{\Delta \vdash \langle\ S, e\ , \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{E} \langle n, \sigma'', \Gamma'', \mathfrak{t}''\rangle}\ (\text{eExprConcat})$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{E} \langle n, \sigma', \Gamma', \mathfrak{t}'\rangle}{\Delta \vdash \langle e_1, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{S} \langle \sigma', \Gamma', \mathfrak{t}'\rangle}\ (\text{eExprAsStmt}) \qquad \frac{}{\Delta \vdash \langle \mathbf{skip}, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{S} \langle \sigma, \Gamma, \mathfrak{t}\rangle}\ (\text{eSkip})$$

$$\frac{\Delta \vdash \langle S_1, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{S} \langle \sigma', \Gamma', \mathfrak{t}'\rangle \qquad \Delta \vdash \langle S_2, \sigma', \Gamma', \mathfrak{t}'\rangle \Downarrow^{S} \langle \sigma'', \Gamma'', \mathfrak{t}''\rangle}{\Delta \vdash \langle S_1; S_2, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{S} \langle \sigma'', \Gamma'', \mathfrak{t}''\rangle}\ (\text{eStmtConcat})$$

$$\frac{\Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{E} \langle n, \sigma', \Gamma', \mathfrak{t}'\rangle \qquad \Gamma'' = \Gamma'[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_a]}{\Delta \vdash \langle x := e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{S} \langle \sigma'[x \leftarrow n], \Gamma'', \mathfrak{t}' + C_{imp} :: \mathfrak{T}_a\rangle}\ (\text{eAssign})$$

$$\frac{\begin{array}{cc} & \Delta \vdash \langle S_2, \sigma', \Gamma', \mathfrak{t}'\rangle \Downarrow^{S} \langle \sigma'', \Gamma'', \mathfrak{t}''\rangle \\ \Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{E} \langle 0, \sigma', \Gamma', \mathfrak{t}'\rangle & \Gamma''' = \Gamma''[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_{ite}] \end{array}}{\Delta \vdash \langle \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end\ if}, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{S} \langle \sigma'', \Gamma''', \mathfrak{t}'' + C_{imp} :: \mathfrak{T}_{ite}\rangle}\ (\text{eIf-False})$$

$$\frac{\begin{array}{cc} n \neq 0 & \Delta \vdash \langle S_1, \sigma', \Gamma', \mathfrak{t}'\rangle \Downarrow^{S} \langle \sigma'', \Gamma'', \mathfrak{t}''\rangle \\ \Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{E} \langle n, \sigma', \Gamma', \mathfrak{t}'\rangle & \Gamma''' = \Gamma''[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_{ite}] \end{array}}{\Delta \vdash \langle \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end\ if}, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{S} \langle \sigma'', \Gamma''', \mathfrak{t}'' + C_{imp} :: \mathfrak{T}_{ite}\rangle}\ (\text{eIf-True})$$

$$\frac{\Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{E} \langle 0, \sigma', \Gamma', \mathfrak{t}'\rangle \qquad \Gamma'' = \Gamma'[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_w]}{\Delta \vdash \langle \mathbf{while}\ e\ \mathbf{do}\ S_1\ \mathbf{end\ while}, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{S} \langle \sigma', \Gamma'', \mathfrak{t}' + C_{imp} :: \mathfrak{T}_w\rangle}\ (\text{eWhile-False})$$

$$\frac{\begin{array}{c} \Gamma'' = \Gamma'[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_w] \qquad \Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{E} \langle n, \sigma', \Gamma', \mathfrak{t}'\rangle \qquad n \neq 0 \\ \Delta \vdash \langle S_1; \mathbf{while}\ e\ \mathbf{do}\ S_1\ \mathbf{end\ while}, \sigma', \Gamma'', \mathfrak{t}' + C_{imp} :: \mathfrak{T}_w\rangle \Downarrow^{S} \langle \sigma'', \Gamma''', \mathfrak{t}''\rangle \end{array}}{\Delta \vdash \langle \mathbf{while}\ e\ \mathbf{do}\ S_1\ \mathbf{end\ while}, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{S} \langle \sigma'', \Gamma''', \mathfrak{t}''\rangle}\ (\text{eWhile-True})$$

$$\frac{\Delta[f \leftarrow (e, \Delta, x)] \vdash \langle S_1, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{S} \langle \sigma', \Gamma', \mathfrak{t}'\rangle}{\Delta \vdash \langle \mathbf{function}\ f(x)\ \mathbf{begin}\ e\ \mathbf{end}\ S_1, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{S} \langle \sigma', \Gamma', \mathfrak{t}'\rangle}\ (\text{eFuncDef})$$

Figure 4.1: Energy-aware semantics.

We can now make the distinction between non-energy-aware component state $C_i :: s$, and energy-aware component state, which also includes the time-stamp τ and the energy accumulator $\mathfrak{e}$.

Most energy consuming actions are explicit in our language: $C_i :: consume()$. However, basic language features, such as evaluation of arithmetic expressions, also implicitly consume energy. We capture this behaviour in the C_{imp} component. This component is an integral part of our energy-aware semantics and logic. The C_{imp} component should at least have resource consumption constants defined for the following operations:

- $C_{imp} :: \mathfrak{E}_e$ and $C_{imp} :: \mathfrak{T}_e$ for expression evaluation.
- $C_{imp} :: \mathfrak{E}_a$ and $C_{imp} :: \mathfrak{T}_a$ for assignment.
- $C_{imp} :: \mathfrak{E}_w$ and $C_{imp} :: \mathfrak{T}_w$ for an iteration of a while loop.
- $C_{imp} :: \mathfrak{E}_{ite}$ and $C_{imp} :: \mathfrak{T}_{ite}$ for conditionals.

To capture the resource consumption of these basic operations, we extend the associated rules in the semantics. The energy-aware rule for assignment (*eAssign*) is listed below, with $C_{imp} :: \mathfrak{E}_a$ for the incidental energy usage of an assignment and $C_{imp} :: \mathfrak{T}_a$ for the time it takes to perform an assignment.

$$\frac{\Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^E \langle n, \sigma', \Gamma', \mathfrak{t}'\rangle \qquad \Gamma'' = \Gamma'[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_a]}{\Delta \vdash \langle x := e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^S \langle \sigma'[x \leftarrow n], \Gamma'', \mathfrak{t}' + C_{imp} :: \mathfrak{T}_a\rangle} \text{(eAssign)}$$

All computations of resource consumption and new component states are done symbolically. In the logic, these values are added, multiplied and subtracted or their `max` is taken. Hence, every $\mathfrak{t}$, $\mathfrak{e}$ and τ, as well as the values in component states, are polynomial expressions, extended with the `max` operator, over program variables. Additionally, symbolic states are used, both as input for the program and as start state for the components. The aforementioned polynomials also range over the symbols used in these symbolic states.

4.3.2 Energy-Aware Modelling

Energy-aware models will be used to derive upper-bounds for energy consumption of the modelled system. In order for the energy-aware model to be suited for the analysis the model should reflect an upper-bound on the actual consumption. This can be based on detailed documentation or on actual energy measurements.

To provide a sound analysis, we need to assume that components are modelled in such a way that the component states reflect different power-levels and are partially ordered. Greater states should imply greater power draw. We will use finite state models only to enable fixpoint calculation in our analysis of while loops. The modelling should be such that the following properties hold (in the context of the full soundness proof in [PKvv13] these properties are axioms):

- *Components states form a finite lattice* with a partial order based on the ordering of polynomials (extended with `max`) over symbolic variables. Within the lattice each pair of component states has a least upper-bound.

- *Energy-aware component states are partially ordered.* This ordering extends the ordering on component states in a natural way by adding an energy accumulator and a timestamp. The timestamp stores the time of the latest change to the component state. So, the earliest timestamp reflects the highest energy usage. Therefore, with respect to timestamps the energy-aware component state ordering should be defined such that smaller timestamps lead to bigger energy-aware component states.
- *Power draw functions preserve the ordering*, i.e. larger states consume more energy than smaller states.
- *Component state update functions δ preserve the ordering.* For this reason, δ_f cannot depend on the arguments of f. To signify this, we will use $\delta(s)$ instead of $\delta(s, args)$ in the logic. As a result, component models cannot influence each other. Our soundness proof (Theorem 2 in Section 4.5) requires this assumption.

Severeness of model restrictions There are several restrictions to the modelling that may seem far from reality.

1. *Component state functions take up a constant amount of time and incidental energy.* This is needed for the soundness proof. For instance, when a radio component sends a message, the duration of the function call cannot directly depend on the number of bytes in the message. In most cases this can be dealt with by using a different way of modelling. First, one can use an overestimation. Second, such dependencies can be removed by distributing the costs over multiple function calls. For instance, the radio component can have a function to send a fixed number of bytes. If it internally keeps a queue, the additional costs of sending the full queue can be modelled by distributing it over separate queueing operations. energy consumption of components must remain fixed per component state.
2. *With each component state a constant power draw is associated.* However, some hardware may accumulate heat over time incurring increasing energy consumption over time. Such a 'heating' problem can be modelled e.g. by changing state to a higher energy level with every call of a component function. This is still an approximation of course. In the future, we want to study models with time driven state change or with time-dependent power draw.
3. *Component model must be finite state machines.* Modelling systems with finite state machines is not uncommon, e.g using model checking and the right kind of abstraction for the property that is studied. In our models the abstraction should be such that the energy consumption is modelled as close as possible.
4. *The effect of component state functions on the component states cannot depend on the arguments of the function.* Also, component models cannot influence each other. Both restrictions are needed for soundness guarantee of our analysis. This restricts the modelling. Using multiple component state functions instead of dynamic arguments and cross-component calls is a way

of modelling that can mitigate these restrictions in certain cases. Relieving these restrictions in general is part of future work.

4.3.3 A Hoare Logic for Energy Analysis

This section treats the definition of an energy-aware logic with energy analysis rules that can be used to bound the energy consumption of the analysed system. The full set of rules is given in Figure 4.2. These rules are deterministic; at each moment only one rule can be applied.

$$\dfrac{}{\{\varGamma;\mathfrak{t};\rho\}n\{\varGamma;\mathfrak{t};\rho\}}\ \text{(aConst)} \qquad \dfrac{}{\{\varGamma;\mathfrak{t};\rho\}x\{\varGamma;\mathfrak{t};\rho\}}\ \text{(aVar)}$$

$$\dfrac{\begin{array}{ll}\{\varGamma;\mathfrak{t};\rho\}e_1\{\varGamma_1;\mathfrak{t}_1;\rho_1\} & \\ \{\varGamma_1;\mathfrak{t}_1;\rho_1\}e_2\{\varGamma_2;\mathfrak{t}_2;\rho_2\} & \varGamma_3 = \varGamma_2[C_{imp}::\mathfrak{e} \mathrel{+}= C_{imp}::\mathfrak{E}_e]\end{array}}{\{\varGamma;\mathfrak{t};\rho\}e_1 \boxdot e_2\{\varGamma_3;\mathfrak{t}_2 + C_{imp}::\mathfrak{T}_e;\rho_2\}}\ \text{(aBinOp)}$$

$$\dfrac{\varGamma_1 = \varGamma[C_i::s \leftarrow C_i::\delta_f(C_i::s), C_i::\tau \leftarrow \mathfrak{t}, C_i::\mathfrak{e} \mathrel{+}= C_i::\mathfrak{E}_f + td(C_i,\mathfrak{t})]}{\{\varGamma;\mathfrak{t};\rho\}C_i::f(args)\{\varGamma_1;\mathfrak{t} + C_i::\mathfrak{T}_f;\rho\}}\ \text{(aCallCmpF)}$$

$$\dfrac{\begin{array}{lll}\Delta(f) = (e_1,x) & \{\varGamma;\mathfrak{t};\rho\}e\{\varGamma_1;\mathfrak{t}_1;\rho_1\} & \\ e = a \in \rho & \{\varGamma_1;\mathfrak{t}_1;\rho_1[x' \leftarrow a]\}e_1[x \leftarrow x']\{\varGamma_2;\mathfrak{t}_2;\rho_2\} & x' \text{ fresh in } e_1\end{array}}{\{\varGamma;\mathfrak{t};\rho\}f(e)\{\varGamma_2;\mathfrak{t}_2;\rho_2\}}\ \text{(aCallF)}$$

$$\dfrac{}{\{\varGamma;\mathfrak{t};\rho\}\mathbf{skip}\{\varGamma;\mathfrak{t};\rho\}}\ \text{(aSkip)} \qquad \dfrac{\{\varGamma\}S\{\varGamma_1\} \qquad \{\varGamma_1\}e\{\varGamma_2\}}{\{\varGamma\}S,e\{\varGamma_2\}}\ \text{(aExprConcat)}$$

$$\dfrac{\{\varGamma;\mathfrak{t};\rho\}S_1\{\varGamma_1;\mathfrak{t}_1;\rho_1\} \qquad \{\varGamma_1;\mathfrak{t}_1;\rho_1\}S_2\{\varGamma_2;\mathfrak{t}_2;\rho_2\}}{\{\varGamma;\mathfrak{t};\rho\}S_1;S_2\{\varGamma_2;\mathfrak{t}_2;\rho_2\}}\ \text{(aConcat)}$$

$$\dfrac{\{\varGamma;\mathfrak{t};\rho\}e\{\varGamma_1;\mathfrak{t}_1;\rho_1\} \qquad \varGamma_2 = \varGamma_1[C_{imp}::\mathfrak{e} \mathrel{+}= C_{imp}::\mathfrak{E}_a]}{\{\varGamma;\mathfrak{t};\rho\}x := e\{\varGamma_2;\mathfrak{t}_1 + C_{imp}::\mathfrak{T}_a;\rho_2\}}\ \text{(aAssign)}$$

$$\dfrac{\begin{array}{ll}\{\varGamma;\mathfrak{t};\rho\}e\{\varGamma_1;\mathfrak{t}_1;\rho_1\} & \{\varGamma_2;\mathfrak{t}_1 + C_{imp}::\mathfrak{T}_{ite};\rho_1\}S_1\{\varGamma_3;\mathfrak{t}_2;\rho_2\} \\ \varGamma_2 = \varGamma_1[C_{imp}::\mathfrak{e} \mathrel{+}= C_{imp}::\mathfrak{E}_{ite}] & \{\varGamma_2;\mathfrak{t}_1 + C_{imp}::\mathfrak{T}_{ite};\rho_1\}S_2\{\varGamma_4;\mathfrak{t}_3;\rho_3\}\end{array}}{\{\varGamma;\mathfrak{t};\rho\}\mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end\ if}\{\mathbf{lub}(\varGamma_3,\varGamma_4);\max\{\mathfrak{t}_2,\mathfrak{t}_3\};\rho_4\}}\ \text{(aIf)}$$

$$\dfrac{\begin{array}{ll}\varGamma_1 = \mathbf{process\text{-}td}(\varGamma,\mathfrak{t}) & \varGamma_3 = \varGamma_2[C_{imp}::\mathfrak{e} \mathrel{+}= C_{imp}::\mathfrak{E}_w] \\ \{\mathbf{wci}(\varGamma_1,e;S);\mathfrak{t};\rho\}e\{\varGamma_2;\mathfrak{t}_1;\rho_1\} & \{\varGamma_3;\mathfrak{t}_1 + C_{imp}::\mathfrak{T}_w;\rho_1\}S\{\varGamma_4;\mathfrak{t}_2;\rho_2\}\end{array}}{\{\varGamma;\mathfrak{t};\rho\}\mathbf{while}_{ib}\ e\ \mathbf{do}\ S\ \mathbf{end\ while}\{\mathbf{oe}(\varGamma_1,\mathfrak{t},\varGamma_4,\mathfrak{t}_2,ib);\rho_3\}}\ \text{(aWhile)}$$

Figure 4.2: Energy analysis rules.

Our energy consumption analysis depends on external symbolic analysis of variables and loop analysis. The results of this external analysis are assumed to be accessible in our Hoare Logic in two ways.

First, we restrict the scope of our analysis to programs that are bound in terms of execution. We assume that all loops and component functions terminate on any input. Each loop is annotated with a bound: $\mathbf{while}_{ib}$. The bound is a polynomial over the input variables, which expresses an upper-bound on the number of iterations of the loop. We have added the ib to the while rule in the energy analysis rules to make this assumption explicit. Derivation of bounds is considered out of scope for our analysis. We assume that an external analysis has produced a sound bound.

Second, the symbolic state environment ρ gives a symbolic state of every variable at each line of code, e.g. $\{x_1 := e_1\}x_1 := x_1 + x_2 + x_3\{x_1 := e_1 + x_2 + x_3\}$, plus other non-energy related properties invariants that have previously been proven. In Figure 4.2 we included this prerequisite by denoting ρ, ρ_1, ... explicitly. As these variables represent external input, thet are not bounded by our logic.

All the judgements in the rules have the following shape: $\{\Gamma; \mathfrak{t}; \rho\}S\ \{\Gamma'; \mathfrak{t}'; \rho'\}$, where Γ is the set of all energy aware component states, $\mathfrak{t}$ is the global time and ρ represents the symbolic state environment retrieved from the earlier standard analysis. The notation $\Gamma[n \mathrel{+}= m]$ is a shorthand for $\Gamma[n \leftarrow n+m]$. As (energy-aware) component states are partially ordered, we can take a least upper bound of states $\mathbf{lub}(s_1, s_2)$ and sets of energy-aware component states $\mathbf{lub}(\Gamma_1, \Gamma_2)$.

We will highlight the most relevant aspects of the rules. The (*aCallCmpF*) rule uses the $td(C_i, t)$ function to estimate the time-dependent energy consumption of component function calls. The (*aIf*) rule takes the least upper bound of the energy-aware component states and the maximum of the time estimates.

Special attention is warranted for the (*aWhile*) rule. We study the body of the while loop in isolation. This requires processing the time-dependent energy consumption that occurred before the loop (**process-td**). An overestimation (**oe**) of the energy consumption of the loop will be calculated by taking the product of the bound on the number of iterations and an overestimation of the energy consumption of a single iteration, i.e. the worst-case iteration (**wci**). The worst-case-iteration is determined by taking the least upper-bound of the set of all states that can occur during the execution of the loop. As there are a finite number of states for each component, this set can be determined via a fix point construction (**fix**). The fixpoint is calculated by iterating the component iteration function (**ci**).

In order to support the analysis of statements after the loop, also an overestimation of the component states after the loop has to be calculated. For brevity, in Figure 4.2, this is dealt with in the calculation of **oe**.

Five calculations are needed:

1. Component iteration function **ci**. The component iteration function $\mathbf{ci}_i(S)$ aggregates the (possibly overestimated) effects of S on C_i. It performs the analysis on S, then considers only the effects on C_i. If there are nested loops or conditionals, the effects on the state of C_i are overestimated in the same manner as in the rest of the analysis. By $\mathbf{ci}_i^n(S)$ we mean the component iteration function applied n times: $\mathbf{ci}_i(S) \circ \mathbf{ci}_i^{n-1}(S)$, with $\mathbf{ci}_i^1(S) = \mathbf{ci}_i(S)$.
2. Fixpoint function **fix**. Because component states are finite, there is an iteration after which a component is in a state that it has already been in, earlier in the loop (unless the loop is already finished before this point is reached).

Since components are independent, the behaviour of the component will be identical to its behaviour the first time it was in this state. This is a fixpoint on the set of component states that can be reached in the loop. It can be found using the $\mathbf{fix}_i(S)$ function, which finds the smallest n for which $\exists k.\mathbf{ci}_i^{n+1}(S) = \mathbf{ci}_i^k(S)$. The number of possible component states is an upper bound for n.

3. Worst-Case Iteration function **wci**. To make a sound overestimation of the energy consumption of a loop, we need to find the iteration that consumes the most. As our analysis is non-decreasing with respect to component states, this is the iteration which starts with the largest component state in the precondition. For this purpose, we introduce the worst-case iteration function $\mathbf{wci}_i(S)$, which computes the least-upper bound of all the states up to the fixpoint: $\mathbf{wci}_i(S) = \mathbf{lub}(\mathbf{ci}_i^0(S), \mathbf{ci}_i^1(S), \ldots, \mathbf{ci}_i^{\mathbf{fix}_i(S)}(S))$. The global version $\mathbf{wci}(\Gamma, S)$ is defined by iteratively applying the $\mathbf{wci}_i(S)$ function to each component C_i in Γ.
4. Overestimation function **oe**. This function overestimates the energy-aware output states of the loop. It needs to do three things: find the largest non-energy-aware output states, find the minimal timestamps and add the resource consumption of the loop itself. This function gets as input: the start state of the loop Γ_{in}, the start time $\mathfrak{t}$, the output state from the analysis of the worst-case iteration Γ_{out}, the end time from the analysis of the worst-case iteration $\mathfrak{t}'$ and the iteration bound ib. It returns an overestimated energy-aware component state and an overestimated global time.
Because component state update functions δ preserve the ordering, the analysis of the worst-case iteration results in the maximum output state for any iteration. This, however, does not yet address the case where the loop is not entered at all. Therefore, we need to take the least-upper bound of the start state and the result of the analysis of the worst-case iteration.
To overestimate time-dependent energy usage, we must revert component timestamps to the time of entering the loop. So, if a component is switched to a greater state at some point in the loop, the analysis assumes it has been in this state since entering the loop. Note that the least-upper bound of energy-aware component states does exactly this: maximise the non-energy-aware component state and minimise the timestamp. Taking $\Gamma_{base} = \mathbf{lub}(\Gamma_{in}, \Gamma_{out})$ we find both the maximum output states and the minimum timestamps.
Now, we can add the consumption of the loop itself. We perform the following calculation for each component: $C_i^{\Gamma_{base}} :: \mathfrak{e} = C_i^{\Gamma_{in}} :: \mathfrak{e} + (C_i^{\Gamma_{out}} :: \mathfrak{e} - C_i^{\Gamma_{in}} :: \mathfrak{e}) \cdot ib$. We do something similar for the time consumption: $\mathfrak{t}_{ret} = \mathfrak{t} + ((\mathfrak{t}' - \mathfrak{t}) \cdot ib)$.
5. Processing time-dependent energy function **process-td**. When analysing an iteration of a loop, we must take care not to include any energy consumption outside of the iteration. This would lead to a large overestimation, since it would be multiplied by the (possibly overestimated) number of iterations. Therefore, before analysing the body, we add the time-dependent energy consumption to the energy accumulator for each component and set all timestamps to the current time. Otherwise, the time-dependent consump-

tion before entering the loop would also be included in the analysis of the iteration. We introduce the function **process-td**$(\mathit{\Gamma}, \mathfrak{t})$, which adds $td(C_i, \mathfrak{t})$ to $C_i :: \mathfrak{e}$ and sets $C_i :: \tau$ to $\mathfrak{t}$, for each component C_i in $\mathit{\Gamma}$.

Applying the rules overestimates the sum of the incidental energy consumption and the time-dependent energy consumption. However, the time-dependent energy consumption is only added to the accumulator at changes of component states. So, as for the energy-aware semantics, the time the components are in their current state should still be accounted for by calculating $\mathfrak{e}_{system}(\mathit{\Gamma}_{end}, \mathfrak{t}_{end})$.

4.4 Example: Wireless Sensor Node

To illustrate our analysis, we model a wireless sensor node, which has a sensor C_s and a radio C_r. Furthermore, it has a basic C_{imp} component for the implicit resource consumption. We analyse the energy usage of a program that repeatedly measures the sensor for 10 seconds, then sends the measurement over the radio, shown in Listing 4.1. The example illustrates both *time-dependent* (sensor) and *incidental* (radio) energy usage. We choose a highly abstract modelling to keep the example brief and simple. A more elaborate example can be found in Section 4.6, where two algorithms are compared using an *implementation* of our Hoare logic. A similar analysis, comparing two algorithms, is also done by hand in [PKvv13].

```
while$_n$ n > 0 do
  C$_s$ :: on();
     ... some code taking 10 seconds ...
  x = C$_s$ :: off();
  C$_r$ :: send(x);
  n = n - 1;
end while;
```

Listing 4.1: Example program.

Modelling The sensor component C_s has two states: s_{on} and s_{off}. It does not have any incidental energy consumption. It has a power draw (thus time-dependent consumption) only when on. For this power draw we introduce the constant $\mathfrak{e}_{\text{on}}$. There are two component functions, namely `on` and `off`, which switch between the two component states.

The radio component C_r only has incidental energy consumption. It does not have a state. Its single component function is `send`, which uses $C_r :: \mathfrak{T}_{send}$ time and $C_r :: \mathfrak{E}_{send}$ energy.

The C_{imp} component models the *implicit* resource consumption by various language constructs. For the sake of presentation, we choose a very simple model here, in which only assignment consumes time and energy. We set both the associated constants $C_{imp} :: \mathfrak{T}_a$ and $C_{imp} :: \mathfrak{E}_a$. The other six constants in the C_{imp} model (see Section 4.3.1 for a list) are set to 0.

Application of the energy-aware semantics from Figure 4.1 on the loop body results in a time consumption $\mathfrak{t}_{\text{body}}$ of $10 + C_r :: \mathfrak{T}_{send} + 2 \cdot C_{imp} :: \mathfrak{T}_a$ and an energy consumption $\mathfrak{e}_{\text{body}}$ of $10 \cdot \mathfrak{e}_{\text{on}} + C_r :: \mathfrak{E}_{send} + 2 \cdot C_{imp} :: \mathfrak{E}_a$. Intuitively, the time and energy consumption of the whole loop are $n(\mathfrak{t}_{\text{body}})$ and $n(\mathfrak{e}_{\text{body}})$.

Energy consumption analysis The analysis (Figure 4.2) always starts with a symbolic state. Note that only the sensor component C_s has a state. We introduce the symbol $\mathbf{on^s_0}$ for the symbolic start-state (on or off) of the sensor.

We start the analysis with the while loop. So, we apply the (*aWhile*) rule:

$$\frac{\begin{array}{l}\Gamma_1 = \mathbf{process\text{-}td}(\Gamma_0, \mathfrak{t}_0) \\ \{\mathbf{wci}(\Gamma_1, n > 0; S_{\text{body}}); \mathfrak{t}_0; \rho_0\} n > 0 \{\Gamma_2; \mathfrak{t}_1; \rho_1\} \qquad \{\Gamma_2; \mathfrak{t}_1; \rho_1\} S_{\text{body}} \{\Gamma_{it}; \mathfrak{t}_{it}; \rho_{it}\}\end{array}}{\{\Gamma_0; \mathfrak{t}_0; \rho_0\} \mathbf{while}_n\ n > 0\ \mathbf{do}\ S_{it}\ \mathbf{end\ while} \{\mathbf{oe}(\Gamma_1, \mathfrak{t}_0, \Gamma_{it}, \mathfrak{t}_{it}, \rho(n)); \rho_{end}\}} \text{(aWhile)}$$

Since $C_{imp} :: \mathfrak{E}_w$ and $C_{imp} :: \mathfrak{T}_w$ are 0, we omit them here. We will first solve the **process-td** and **wci** functions, then analyse the loop guard and body (i.e. the part above the line), then determine the final results with the **oe** function.

We first add time-dependent energy consumption and set timestamps to $\mathfrak{t}_0$ for all components using the **process-td** function. If we would not do this, the time-dependent energy consumption *before* the loop would be included in the calculation of the resource consumption of the worst-case iteration. As this would be multiplied by the number of iterations, it would lead to a large overestimation. C_r and C_{imp} do not have a state, so we only need to add the time-dependent consumption of C_s: $td(C_s, \mathfrak{t}_0) = C_s :: \phi(\mathbf{on^s_0}) \cdot (\mathfrak{t}_0 - C_s :: \tau_0)$, where $C_s :: \tau_0$ is the symbolic value of the sensor timestamp before starting the analysis.

We must now find the worst-case iteration, using the **wci** function. For the C_s component we need the $\mathbf{ci}_s(n > 0; S_{\text{body}})$ function. As the other components do not have a state, $\mathbf{ci}_{imp}(n > 0; S_{\text{body}})$ and $\mathbf{ci}_r(n > 0; S_{\text{body}})$ are simply the identity function. The loop body sets the state of the sensor to s_{off}, independent of the start state. So, $\mathbf{ci}_s(n > 0; S_{\text{body}})$ always results in s_{off}. Now we can find the fixpoint. In the first iteration, we enter the loop with symbolic state $\mathbf{on^s_0}$. In the second iteration, the loop is entered with state s_{off}. In the third iteration, the loop is again entered with state s_{off}. We have thus found the fixpoint. The worst-case iteration can now be calculated by $\mathbf{wci}_s(n > 0; S_{\text{body}}) = \mathbf{lub}(\mathbf{ci}^0_s(n > 0; S_{\text{body}}), \mathbf{ci}^1_s(n > 0; S_{\text{body}})) = \mathbf{on^s_0}$. Intuitively, this means that, since after any number of iterations the sensor is off, the symbolic start state, in which it is unknown whether the sensor is on or off, yields the worst-case.

As there are no costs associated with the evaluation of expressions, the analysis of $n > 0$ using the (*aBinOp*) rule does not have any effect on the state. We continue with analysis of the loop body, which starts with a call to component function `on`. We apply the (*aCallCmpF*) rule:

$$\frac{\Gamma_3 = \Gamma_2[C_s :: s \leftarrow C_s :: \delta_{on}(C_s :: s), C_s :: \tau \leftarrow \mathfrak{t}_1, C_s :: \mathfrak{e} \mathrel{+}= td(C_s, \mathfrak{t}_1)]}{\{\Gamma_2; \mathfrak{t}_1; \rho_1\} C_s :: on() \{\Gamma_3; \mathfrak{t}_1; \rho_2\}} \text{(aCallCmpF)}$$

There is no incidental energy consumption or time consumption associated with the call. We must however add the time-dependent energy consumption to the energy accumulator, by adding $td(C_s, \mathfrak{t})$. Since we have just set $C_s :: \tau$ to $\mathfrak{t}_0$ and the evaluation of $n > 0$ costs 0 time, hence $\mathfrak{t}_1 = \mathfrak{t}_0$, $td(C_s, \mathfrak{t}_1)$ results in 0. The function $C_r :: \delta_{\text{on}}$ produces new component state s_{on}. It also saves the current timestamp to the component state, in order to know when the last state transition happened. For simplicity, we omit the application of the concatenation rule (*aConcat*) in the following.

After ten seconds of executing other statements (which we assume only cost time, not energy), the sensor is turned off. The call to the function *off* returns the measurement, which is assigned to x. We must therefore first apply the (*aAssign*) rule. This adds $C_{imp} :: \mathfrak{T}_a$ to the global time and $C_{imp} :: \mathfrak{E}_a$ to the energy accumulator. We now apply the (*aCallCmpF*) rule to the right-hand side of the assignment, i.e. the call to $C_s :: \mathit{off}$. This updates the state of the component to s_{off}. It also executes the $td(C_s, \mathfrak{t}_2)$ function in order to determine the energy cost of the component being on for ten seconds. Because $\mathfrak{t}_2 = C_s :: \tau + 10$ and our model specifies a power draw of $\mathfrak{e}_{\text{on}}$ for s_{on}, this results in $10 \cdot \mathfrak{e}_{\text{on}}$. We add this to the energy accumulator of the sensor component.

We apply the (*aCallCmpF*) rule again, this time to the *send* function of the C_r component. As the transmission costs a fixed amount of energy, all time-dependent constants associated with transmitting are set to zero. So, the (*aCallCmpF*) rule will only add the incidental energy usage specified by $C_s :: \mathfrak{E}_{send}$ and the constant time usage $C_s :: \mathfrak{T}_{send}$. Finally, we apply the (*aBinOp*) rule, which has no costs, and the (*aAssign*) rule, which again adds $C_{imp} :: \mathfrak{E}_a$ and $C_{imp} :: \mathfrak{T}_a$.

Analysis of the worst-case iteration results in global time $\mathfrak{t}_{it}$ and energy-aware component state environment Γ_{it}. We can now apply the overestimation function $\mathbf{oe}(\Gamma_1, \mathfrak{t}_0, \Gamma_{it}, \mathfrak{t}_{it}, \rho(n))$. This takes as base the least-upper bound of Γ_1 and Γ_{it}, which in this case is exactly Γ_1 (note that the state of the sensor is overestimated as $\mathbf{on}_\mathbf{0}^\mathsf{s}$). It then adds the consumption of the worst-case iteration, multiplied by the number of iterations. The worst-case iteration results in a global time of $\mathfrak{t}_0 + 10 + C_r :: \mathfrak{T}_{send} + 2 \cdot C_{imp} :: \mathfrak{T}_a$. So, $\mathbf{oe}$ results in a global time $\mathfrak{t}_{end}$ of $\mathfrak{t}_0 + n \cdot (10 + C_r :: \mathfrak{T}_{send} + 2 \cdot C_{imp} :: \mathfrak{T}_a)$. Note that this is equal to the time consumption resulting from the energy-aware semantics.

A similar calculation is made for energy consumption, for each component. Then, we can calculate $\mathfrak{e}_{system}$. In total, the $\mathbf{oe}$ function results in an energy usage of $\mathfrak{e}_0 + n \cdot (10 \cdot \mathfrak{e}_{\text{on}} + C_r :: \mathfrak{E}_{send} + 2 \cdot C_{imp} :: \mathfrak{E}_a)$. However, we still need to add the time-dependent energy consumption for each component. This is where potential overestimation occurs in this example. Since C_r and C_{imp} do not have a state, we only need to add the time-dependent consumption of C_s. After the analysis of the loop, the state of the sensor is overestimated as $\mathbf{on}_\mathbf{0}^\mathsf{s}$. We must therefore add a consumption of $td(C_s, \mathfrak{t}_{\text{end}}) = C_s :: \phi(\mathbf{on}_\mathbf{0}^\mathsf{s}) \cdot (\mathfrak{t}_{end} - \mathfrak{t}_0) = C_s :: \phi(\mathbf{on}_\mathbf{0}^\mathsf{s}) \cdot n \cdot (10 + C_r :: \mathfrak{T}_{send} + 2 \cdot C_{imp} :: \mathfrak{T}_a)$. This leads to an overestimation only in case $\mathbf{on}_\mathbf{0}^\mathsf{s} = s_{\text{on}}$ and $n > 0$. Otherwise, the result of the analysis is equal to that of the energy-aware semantics.

4.5 Soundness

In this section, we outline a proof of the soundness of the energy-aware Hoare logic with respect to the energy-aware semantics. Intuitively, this means we prove that the analysis over-estimates the actual energy consumption. Here, we present only the fundamental theorems. The reader is referred to [PKvv13] for the full proof. Soundness of the annotations (loop bounds and symbolic states) is assumed in order to guarantee soundness of the final result.

We first show that the logic over-estimates time consumption. In order to establish soundness of the analysis of the energy consumption of the program, we need to establish first soundness of the timing analysis.

Theorem 1 (Timing over-estimation). *If* $\langle S, \sigma, \Gamma, \mathtt{t}\rangle \Downarrow^S \langle \sigma', \Gamma', \mathtt{t}'\rangle$*, then for any derivation* $\{\Gamma; \mathtt{t}; \rho\} S \{\Gamma_1; \mathtt{t}_1; \rho_1\}$ *holds that* $\mathtt{t}_1 \geq \mathtt{t}'$*.*

Proof. Theorem 1 derives from the property that the analysis does not depend on the timestamp in the precondition. For $\{\Gamma_1; \mathtt{t}_1; \rho_1\} S \{\Gamma_2; \mathtt{t}_2; \rho_2\}$, the duration $\mathtt{t}_2 - \mathtt{t}_1$ always over-estimates the duration of every possible real execution of the statement S. Theorem 1 is proved by induction on the energy-aware semantics and the energy analysis rules. The only source of any over-estimation are the rules (*aIf*) and (*aWhile*). The (*aIf*) rule computes a `max` between the final timestamps of then-branch and the else-branch. In the (*aWhile*) rule, the execution time of one iteration of the loop is over-estimated and multiplied by the loop bound, which is an over-estimation of the number of iterations of the loop. □

Over-estimating the component state is fundamental for over-estimating the total energy consumption. A larger component state requires more power and hence consumes more energy.

Theorem 2 (Component state over-estimation). *If* $\{\Gamma; \mathtt{t}; \rho\} S \{\Gamma_1; \mathtt{t}_1; \rho_1\}$ *and* $\langle S, \sigma, \Gamma, \mathtt{t}\rangle \Downarrow^S \langle \sigma', \Gamma', \mathtt{t}'\rangle$ *then* $\Gamma_1 \geq \Gamma'$*.*

Proof. Induction on the energy-aware semantics and the energy analysis rules, yields that the update function δ preserves the ordering on component states (see Section 4.3.2). □

Now, we can formulate and prove the main soundness theorem:

Theorem 3 (Soundness). *If* $\{\mathtt{t}; \Gamma; \rho\} S \{\mathtt{t}_1; \Gamma_1; \rho_1\}$ *and* $\langle S, \sigma, \Gamma, \mathtt{t}\rangle \Downarrow^S \langle \sigma', \Gamma', \mathtt{t}'\rangle$ *then* $\mathfrak{e}_{system}(\Gamma_1; \mathtt{t}_1) \geq \mathfrak{e}_{system}(\Gamma'; \mathtt{t}')$*.*

Proof. By induction on the energy-aware semantics and the energy analysis rules. Theorem 1 ensures that the final timestamp is an over-estimation of the actual time-consumption, hence the calculation of energy usage is based on an over-estimated period of time. Theorem 2 ensures (given that the analysis is non-decreasing with respect to component states, a larger input state means a larger output state) that we find the maximum state (including incidental energy-usage) that can result from an iteration of a loop body with the logic. This depends on the **wci** function determining the maximal initial state for any iteration. It follows, by the definition of $\mathfrak{e}_{system}$ that $\mathfrak{e}_{system}(\Gamma_1; \mathfrak{t}_1) \geq \mathfrak{e}_{system}(\Gamma'; \mathfrak{t}')$. The total energy consumption resulting from the analysis is larger than that of every possible execution of the analysed program. □

4.6 Implementation in ECAlogic

This section presents ECAlogic[5], a tool that implements the static energy consumption analysis. The tool is parametric with respect to a set of hardware component models. Its results are symbolic over the program parameters. It has a web-interface as well as a command-line interface.

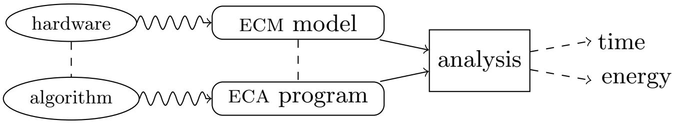


Figure 4.3: Architecture of the ECAlogic tool.

A schematic representation of ECAlogic is shown in Figure 4.3. The algorithm and the hardware on which it will run must first be modelled. To capture the functionality of the algorithm, we offer the simple 'while' language ECA, described in Section 4.6.1. Each hardware component is modelled in a similar language, ECM, which is described in Section 4.6.2.

Component functions explicitly influence energy consumption. Other language constructs, for instance the evaluation of an arithmetic expression, also implicitly consume energy. This is modelled in the special *implicit* component. This component is assumed to be present in any system. It is modelled in ECM and therefore under full user control.

4.6.1 Input Language ECA

For describing algorithms, we use the simple programming language ECA. Because ECAlogic was developed in parallel with the analysis, there are minor differences with the "while" language described in Section 4.2. However, these do not affect the set of algorithms that can be expressed. The grammar of the language is shown in Figure 4.4. A program is represented as a function with input parameters. ECA is a simple "while"-type language, with the usual control structures and function calls. It has two major restrictions:

- All while-loops are bounded in the number of iterations. This upper bound must be specified explicitly and is assumed to be sound. It can either be inferred by a third-party tool or specified directly by the programmer.

[5] http://resourceanalysis.cs.ru.nl/energy/

```
⟨program⟩ ::= {⟨comp-imp⟩ ⟨sep⟩} {⟨fun-def⟩ ⟨sep⟩}
⟨comp-imp⟩ ::= ‘import’ ‘component’ id {‘.’ id} [‘as’ id]
⟨fun-def⟩ ::= ‘function’ id [‘(’ [id {‘,’ id}] ‘)’] ⟨fun-body⟩
⟨fun-body⟩ ::= ‘:=’ ⟨expr⟩
   |  ⟨stat-list⟩ ‘end’ ‘function’
   |  ⟨empty⟩
⟨stat-list⟩ ::= {⟨statement⟩ ⟨sep⟩}
⟨statement⟩ ::= ‘skip’
   |  id ‘:=’ ⟨expr⟩
   |  ⟨fun-call⟩
   |  ‘if’ ⟨expr⟩ ‘then’ ⟨stat-list⟩ ‘else’ ⟨stat-list⟩ ‘end’ ‘if’
   |  ‘while’ ⟨expr⟩ ‘bound’ ⟨expr⟩ ‘do’ ⟨stat-list⟩ ‘end’ ‘while’
   |  ‘{’ ⟨annot-elem⟩ {‘,’ ⟨annot-elem⟩} ‘}’ [⟨statement⟩]
⟨fun-call⟩ ::= [id ‘::’] id ‘(’ [⟨expr⟩ {‘,’ ⟨expr⟩}] ‘)’
⟨annot-elem⟩ ::= id ‘<-’ ⟨expr⟩
⟨expr⟩ ::= ⟨expr⟩ ⟨bin-op⟩ ⟨expr⟩
   |  id
   |  numeral
   |  ⟨fun-call⟩
   |  ‘(’ ⟨expr⟩ ‘)’
⟨bin-op⟩ ::= ‘or’|‘and’|‘=’|‘<>’|‘>’|‘<’|‘>=’|‘<=’|‘+’|‘-’|‘*’|‘/’|‘^’
⟨sep⟩ ::= ‘;’ | end-of-line
⟨id⟩ ::= [:a-zA-Z_:] {[:a-zA-Z0-9_:]}
⟨numeral⟩ ::= [:0-9:] {[:0-9:]}
⟨comment⟩ ::= ‘//’ {⟨any-character⟩} (⟨end-of-line⟩ | ⟨end-of-file⟩)
   |  ‘/*’ {⟨any-character⟩} ‘*/’
   |  ‘(*’ {⟨any-character⟩} ‘*)’
```

Figure 4.4: Grammar of the input language ECA.

- All variables are positive integers. There is no form of structured data. These can however be simulated by modelling them as *component functions*, as we will see below.

A simple program for a wireless sensor node that switches the radio on, takes N measurements and transmits these, is shown in Listing 4.2.

```
function alwaysOn(N)
  Radio::on()
  while N > 0 bound N do
    Value := Sensor::measure()
    Radio::queue(Value)
    Radio::send()
    N := N−1
  end while
  Radio::off()
end function
```

Listing 4.2: Example program for a wireless sensor node.

Here the parameter N acts as an upper bound on the number of iterations of the while loop. It is allowed to use any expression as an upper bound, as long as it can be evaluated in terms of the parameters of a function. In many cases this can be done directly, as above. If, however, the upper bound of a loop references variables whose values are only available at run time, an annotation with a Hoare-style precondition is required to relate each of those variables to the parameters. An example of this is given in Section 4.6.3.

4.6.2 Component Models in ECM

Hardware components models are defined by:

1. A (possibly empty) set of component states
2. A function phi which maps component states to power draw
3. A set of component functions

A simple model for a radio is shown in Listing 4.3.

```
component Radio(active: 0..1)
  initial active := 0

  component function on uses 400 time 400 energy
    active := 1
  end function

  component function off uses 200 time 200 energy
    active := 0
  end function

  component function queue(X) uses 30 time 30 energy
  component function send uses 100 time 100 energy

  function phi := 2 + 200 * active
end component
```

Listing 4.3: A simple model for a radio.

In this example, the radio has two states: off (0) or on (1). There are component functions to turn the radio on/off, to enqueue a measurement for sending and for transmitting the queue. The function on has an incidental energy usage of 30 and changes the state of the component to active. The function phi gives the energy consumption per time-unit, depending on the state of the radio. Note that this is where the timing analysis is needed.

An important constraint on the phi function is monotonicity with respect to the ordering of the states: a higher state implies a higher energy usage. ECAlogic checks whether this constraint holds. Apart from that, component functions have the same expressive power as the ECA language. Hence, more detailed models can easily be constructed.

```
⟨component⟩ ::= {⟨class-imp⟩ ⟨sep⟩} 'component' id ['(' [⟨var-def⟩ {',' ⟨var-def⟩}] ')']
      {⟨member⟩ ⟨sep⟩} 'end' 'component'
⟨class-imp⟩ ::= 'import' 'class' id {'.' id} ['as' id]
⟨var-def⟩ ::= id ':' numeral '..' numeral
⟨member⟩ ::= 'initial' id ':=' numeral
  |  ⟨fun-def⟩
  |  ⟨comp-fun-def⟩
⟨comp-fun-def⟩ ::= 'component' 'function' id ['(' [id {',' id}] ')'] [⟨uses-clause⟩]
      ⟨fun-body⟩
⟨uses-clause⟩ ::= 'uses' numeral 'energy' [numeral 'time']
  |  'uses' numeral 'time' [numeral 'energy']
```

Figure 4.5: Grammar of the component modelling language ECM.

4.6.3 Tool Application

To use the tool, the target platform must first be modelled. This is a complex step, as building a precise model requires measurements of the actual energy consumption. Depending on the goals of the user, educated guessing might suffice when modelling, or a standard ECM model (e.g. for the CPU) taken from a library of component models might be used. If precise results are required, accurate modelling is paramount. If the user wants to compare implementation variants, a less precise modelling will often suffice.

A typical use case is the comparison of different implementations of an algorithm. In the case of a wireless sensor node, a strategy to conserve energy is to send data packets in batches of size B, only turning the radio on right before sending. An alternative implementation of the wireless sensor node example is shown in Listing 4.4.

```
function buffering(N, B)
  while N > 0 bound N/B do
    K := B
    { K <- B }
    while K > 0 and N > 0 bound K do
      Value := Sensor::measure()
      Radio::queue(Value)
      K := K - 1
      N := N - 1
    end while
    Radio::on()
    Radio::send()
    Radio::off()
  end while
end function
```

Listing 4.4: An alternative application for a wireless sensor node, using buffering.

Note that the annotation *{ K <- B }* is necessary to express that the symbolic value of the variable K in terms of the function parameters is *B*. ECALOGIC issues a diagnostic message whenever the symbolic value of a loop bound or function argument cannot be determined.

```
component Implicit
  component function e   uses 10 energy 10 time
  component function a   uses 5  energy 5  time
  component function w   uses 25 energy 25 time
  component function ite uses 25 energy 25 time
end component
```

Listing 4.5: Implicit component model

```
component Sensor
  component function measure uses 40 energy 10 time
  function phi := 3
end component
```

Listing 4.6: Sensor component model

Using the simple implicit component model in Listing 4.5 and sensor model in Listing 4.6, we can now compare the two implementations:

implementation	*time*	*energy*
alwaysOn(N)	$600 + 195 \cdot N$	$83600 + 40200 \cdot N$
buffering(N,B)	$(130 + \frac{740}{B}) \cdot N$	$(1070 + \frac{105640}{B}) \cdot N$

These results also provide information on appropriate values for the block size *B*. If we are interested in a constantly functioning sensor node, we should consider very large *N*. It is then clear that the buffering implementation is more efficient for $B \geq 3$.

4.7 Conclusions and Future Work

We presented a hybrid, energy-aware Hoare logic for reasoning about energy consumption of systems controlled by software. The logic comes with an analysis which is proven to be sound with respect to the semantics. To our knowledge, our approach is the first attempt at bounding energy-consumption statically in a way which is parametric with respect to hardware models. This is a first step towards a hybrid approach to energy consumption analysis in which the software is analysed automatically together with the hardware it controls. The analysis is implemented in the tool ECALOGIC. In the Software Analysis course at Radboud University Nijmegen students have used ECALOGIC for exercises, successfully modelling various algorithms and hardware components.

Future Work Many future research directions can be envisioned: e.g. performing energy measurements for defining component models, modelling of software components and enabling the development of tools that can automatically derive energy consumption bounds for large systems, finding the most suited tool(s) to provide the right loop bounds and annotations for our analysis and study energy usage per time unit on systems that are always running, removing certain termination restrictions.

With regard to the implementation, we aim to improve the ECAlogic tool as follows:

- Since ECAlogic currently only supports the eca language, the software to be analysed must be expressed in this language. When analysing existing software, this is restrictive. Eventually, we want to support a 'real-world' programming language, such as C.
- We want to increase the inter-operability with other analysis tools, such as ResAna [SKVE10] for deriving the necessary loop bounds and a CEGAR-based tool [tMB$^+$14] for deriving component models.
- To increase the practical applicability, we will start an open library where users can submit ecm models for hardware components. An additional advantage of such a library is that users with access to physical measurement tools could take a model from the library and validate it.

Chapter 5

Test-based Inference of Polynomial Ranking Functions for Loops

Abstract. This chapter presents an interpolation-based method of inferring arbitrary degree ranking functions for Java loops. Given a loop, by its “ranking function” we mean a function with the numeric program variables as its parameters, that is used to bound the number of iterations.

Analysis of loop bounds is important in several different areas, including worst-case execution time (WCET) and heap consumption analysis, optimising compilers and termination-analysis. While several other methods exist to infer numerical loop bounds, we know of no other research on the inference of non-linear symbolic loop bounds. Additionally, the inferred bounds are provable using external tools, e.g. KeY.

To infer a ranking function for a given loop, it is instrumented with a counter and executed on a *well-chosen* set of values of the numerical program variables. By well-chosen we mean that using these test values and the corresponding values of the counter, one can construct a unique interpolating polynomial. The uniqueness and the existence of the interpolating polynomial is guaranteed if the input values are in the so-called NCA-configuration, known from multivariate-polynomial interpolation theory. The constructed interpolating polynomial presumably bounds the dependency of the number of loop iterations on arbitrary values of the program variables. This hypothetical loop bound is verified by a third-party proof assistant.

A prototype tool has been developed which implements this method. This prototype can infer piecewise polynomial ranking functions for a large class of loops in Java programs. Applicability of the prototype has been tested on a series of safety-critical case studies. For most of the loops in the case studies, ranking functions could be inferred (and verified using a proof assistant).

5.1 Introduction

Most of the execution time of typical embedded programs is spent in loops. In bounding resource consumption, it is therefore an important step to bound loop iterations. Compiler optimisations that transform loops, e.g. loop-unrolling, may also depend on knowledge about loop-bounds. Furthermore, termination of a program can only be proved if an upper bound on the number of loop iterations exists for each loop in the program.

In this chapter, we describe test-based inference of piecewise polynomial ranking functions for Java loops with first-order numerical loop guards. The *ranking function* (RF) of a loop expresses an upper bound on the number of iterations, depending on (some of) the numerical program variables and data sizes.

Consider, for example, the loop in Listing 5.1. While several other methods exist that are able to infer a numerical loop bound for certain values of `i`, for instance minimal or maximal values, we infer the *symbolic* loop bound `15-i`, which can be used to bound (or in this case, calculate exactly) the number of iterations of this loop for *arbitrary* values of `i`. Only one paper [Gul09] is known to the authors where symbolic bounds are inferred, but in the method described in that paper only a limited use of non-linear terms in bounds is possible. In other articles, soundness is not usually discussed, but the correctness of the RF inferred by the presented method can be checked by external tools. By proving correctness of the bound, termination of the loop is also proved inherently.

```
while (i < 15) {
  i++;
}
```

Listing 5.1: A typical single (i.e. not nested) while-loop.

The schema of our approach is as follows. Given a loop, it is first placed into a testing method. The loop in this method is instrumented with a counter and this testing method outputs the number of iterations of the loop on given values of the program variables. We take into consideration numerical program variables that occur in the loop condition and its body. Then, the method is executed on a *well-chosen* set of inputs, which we call test values or, sometimes, following polynomial-interpolation terminology, test nodes. By well-chosen set of test values we mean that using these test values and the corresponding values of the counter, one can construct a unique interpolating polynomial. The uniqueness and the existence of the interpolating polynomial is guaranteed if the input values are in the so-called NCA-configuration, known from multivariate-polynomial interpolation theory. This issue is highlighted in Section 5.2, explaining application of polynomial interpolation. The obtained results are used to compute the corresponding interpolating polynomial. The inference part of the procedure outputs a method, annotated with the inferred bound, e.g. annotated in JML [CKLP06] by the `decreases`-expression. The correctness of the annotation is verified by an external checking tool. In our prototype implementation the method with the annotated loop is run through KeY [BHS07], which contains a verification-condition generator and a theorem prover.

To obtain concrete loop bounds (i.e. the concrete number of iterations on concrete data), the obtained RF may be applied to the results of data-flow analysis, possibly accelerated by abstract interpretation and/or program slicing. Several examples exist in the literature [ESG$^+$07, LCFM09] indicating how this might be implemented.

This work builds on previous research on size analysis [SvEvK09], where we studied polynomial dependencies of the sizes of output data structures (e.g. the length of a linked list) on the sizes of input data structures. A similar algorithm as is used here for generating loop bounds was used in [vKSvE08] to infer size relations.

The running example throughout this chapter is a while-loop with a quadratic RF, given in Listing 5.2. This bound is successfully inferred by the prototype implementation and can be verified using KeY (although some manual steps are required for such a complex bound).

```
while (x>0 && i>0 && i<x && j>0 && j<=x) {
  if (j==x) { i++; j = 0; }
  j++;
}
```

Listing 5.2: A single loop with the quadratic RF $x^2 - xi - j + 1$.

This research is conducted in the context of the Critical and High Assurance Requirements Transformed through Engineering Rigour (CHARTER) project[6]. The goal of this project is to ease, accelerate, and cost-reduce the certification of safety-critical embedded systems by melding Realtime Java, Model Driven Development, rule-based compilation, and formal verification. It is part of a larger chain of tools developed in this project.

We recapitulate polynomial interpolation in Section 5.2. The RF inference method is introduced in Section 5.3. We then discuss the prototype implementation and a series of case studies in Section 5.4. Related work is discussed in Section 5.5 and the chapter is concluded in Section 5.6.

5.2 Polynomial Interpolation

When the result of a polynomial function is known for certain test values, the values of its coefficients can be derived. Such a polynomial, which `interpolates` the test results, exists and is unique under some conditions on the data, which are explored in polynomial-interpolation theory [CL87].

For 1-variable interpolation this condition is well-known: all the test nodes must be different. Let us recapitulate in more detail. A polynomial $p(z)$ of degree

[6] `http://charterproject.ning.com/`

d with coefficients $a_0, \ldots, a_d$ can be written as follows:

$$a_0 \;+\; a_1 z \;+\; \ldots \;\; + \; a_d\, z^d = \; p(z)$$

The values of the polynomial function in any pairwise different $d{+}1$ points determine a system of linear equations with respect to the polynomial coefficients. More specifically, given the set $\big(z_i, p(z_i)\big)$ of pairs of numbers, where $0 \leq i \leq d$, and coefficients $a_0, \;\ldots\; , a_d$, the system of equations can be represented in the following matrix form, where only the a_i are unknown:

$$\begin{pmatrix} 1 & z_0 & \cdots & z_0^{d-1} & z_0^d \\ 1 & z_1 & \cdots & z_1^{d-1} & z_1^d \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & z_{d-1} & \cdots & z_{d-1}^{d-1} & z_{d-1}^d \\ 1 & z_d & \cdots & z_d^{d-1} & z_d^d \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{d-1} \\ a_d \end{pmatrix} = \begin{pmatrix} p(z_0) \\ p(z_1) \\ \vdots \\ p(z_{d-1}) \\ p(z_d) \end{pmatrix}$$

The determinant of the matrix is called a *Vandermonde* determinant. For pairwise different points $z_0, \ldots,\; z_d$ it is non-zero. This means that, as long as the output values $p(z_i)$ are known for $d + 1$ different inputs z_i, there exists a unique solution for the system of equations and, thus, a unique interpolating polynomial.

The condition under which there exists a unique *multivariate* polynomial $p(z_1, \ldots, z_k)$ that interpolates multivariate data is not trivial. Using the result from [CL87], it is shown how to generate test data for size analysis of *functional programs* in [SvEvK09]. Here we recall the basic facts from these papers. First, a polynomial $p(z_1, \ldots, z_k)$ of a degree d and dimension k (the number of variables) has $N_d^k = \binom{d+k}{k}$ coefficients. Let a set of values f_i of a real function f be given and let $\bar{z}$ denote a vector-variable $(z_1, \ldots, z_k)$. A set $W = \{\bar{w}_i = (z_{i1}, \ldots, z_{ik}) : i = 1, \ldots,\; N_d^k\}$ of points in a real k-dimensional space forms the set of *interpolation nodes* if there is a unique polynomial $p(\bar{z}) = \Sigma_{0 \leq j_1 + \ldots + j_k \leq d} a_{j_1 \ldots j_k} z_1^{j_1} \ldots z_k^{j_k}$ with the total degree d with the property $p(\bar{w}_i) = f_i$, where $1 \leq i \leq N_d^k$. In this case one says that the polynomial p interpolates the function f at the nodes $\bar{w}_i$. The condition on W, which assures the existence and uniqueness of an interpolating polynomial, is geometrical: it describes a node configuration, called *Node Configuration A* [CL87], **NCA** in short, in which the nodes from W should be placed in $\mathcal{R}^k$. The multivariate Vandermonde determinant computed from such points is non-zero. Thus, the corresponding system of linear equations with respect to the polynomial its coefficients has a unique solution. For a two-dimensional polynomial of degree d, the condition on the nodes that guarantees a unique polynomial interpolation is as follows:

N_d^2 nodes forming a set $W \subset \mathcal{R}^2$ lie in a 2-dimensional NCA if there exist lines $\gamma_1, \ldots, \gamma_{d+1}$ in the space $\mathcal{R}^2$, such that $d{+}1$ nodes of W lie on γ_{d+1} and d nodes of W lie on $\gamma_d \setminus \gamma_{d+1}$, $\ldots$, and finally 1 node of W lies on $\gamma_1 \setminus (\gamma_2 \cup \ldots \cup \gamma_{d+1})$.

A typical instance of such a configuration is a 2-dimensional *grid*. An example of a two-dimensional grid based on integers is given in Figure 5.1.

For dimensions $k > 2$ the NCA is defined inductively on k. A set of N_d^k nodes is in *NCA in $\mathcal{R}^k$ if and only if*

- *there is a $(k-1)$-dimensional hyperplane such that it contains some N_d^{k-1} of the given nodes lying in $(k-1)$-dimensional NCA for the degree d,*
- *there is a $(k-1)$-dimensional hyperplane such that it contains some N_{d-1}^{k-1} nodes, lying in $(k-1)$-dimensional NCA for the degree $d-1$, and these nodes do not lie on the previous hyperplane,*
- *in general, for any $0 \leq i \leq d$, there is a $(k-1)$-dimensional hyperplane such that it contains some N_{d-i}^{k-1} nodes, lying in $(k-1)$-dimensional NCA for the degree $d-i$, and these nodes do not lie on the previous hyperplanes,*
- *thus, the remaining* 1 *node lies on the remaining hyperplane and does not belong to the previous ones.*

For instance, for the example in Listing 5.2, we might assume that the ranking function is a quadratic polynomial (so, $d = 2$) and depends on three variables, x, i and j. Recall, that a quadratic function of three variables has $\binom{5}{3} = 10$ coefficients: $p(x, i, j) = a_{200}x^2 + a_{020}i^2 + a_{002}j^2 + a_{110}xi + a_{101}xj + a_{011}ij + a_{100}x + a_{010}i + a_{001}j + a_{000}$. Therefore we need 10 three-dimensional points in $\mathcal{R}^3$-NCA. According to the definition above we need:

- A set of $\binom{2+2}{2} = 6$ points on a plane in $\mathcal{R}^2$-NCA: we take the hyperplane $j = 1$ and the following nodes:

x	i	j
2	1	1
3	1	1
4	1	1
3	2	1
4	2	1
4	3	1

  These points are given (projected on the hyperplane $j = 1$) in Figure 5.1.

- A set of $\binom{2+1}{2} = 3$ points in $\mathcal{R}^2$-NCA on another plane: we take the hyperplane $j = 2$ and nodes

x	i	j
3	1	2
4	1	2
3	2	2

- A single $(\binom{2+0}{2} = 1)$ point on yet another plane in $\mathcal{R}^2$-NCA: we take just $(x = 4, i = 1, j = 3)$.

The corresponding system of linear equations is:

$$\begin{cases}
\begin{aligned}
&2^2a_{200} + 1^2a_{020} + 1^2a_{002} + 2a_{110} + 2a_{101} + \\
&\quad 1a_{011} + 2a_{100} + 1a_{010} + 1a_{001} + 1a_{000} = 2 \\
&3^2a_{200} + 1^2a_{020} + 1^2a_{002} + 3a_{110} + 3a_{101} + \\
&\quad 1a_{011} + 3a_{100} + 1a_{010} + 1a_{001} + 1a_{000} = 6 \\
&4^2a_{200} + 1^2a_{020} + 1^2a_{002} + 4a_{110} + 4a_{101} + \\
&\quad 1a_{011} + 4a_{100} + 1a_{010} + 1a_{001} + 1a_{000} = 12 \\
&3^2a_{200} + 2^2a_{020} + 1^2a_{002} + 3 \cdot 2a_{110} + 3a_{101} + \\
&\quad 2a_{011} + 3a_{100} + 2a_{010} + 1a_{001} + 1a_{000} = 3 \\
&4^2a_{200} + 2^2a_{020} + 1^2a_{002} + 4 \cdot 2a_{110} + 4a_{101} + \\
&\quad 2a_{011} + 4a_{100} + 2a_{010} + 1a_{001} + 1a_{000} = 8 \\
&4^2a_{200} + 3^2a_{020} + 1^2a_{002} + 4 \cdot 3a_{110} + 4a_{101} + \\
&\quad 3a_{011} + 4a_{100} + 3a_{010} + 1a_{001} + 1a_{000} = 4 \\
&3^2a_{200} + 1^2a_{020} + 2^2a_{002} + 3a_{110} + 3 \cdot 2a_{101} + \\
&\quad 2a_{011} + 3a_{100} + 1a_{010} + 2a_{001} + 1a_{000} = 5 \\
&4^2a_{200} + 1^2a_{020} + 2^2a_{002} + 4a_{110} + 4 \cdot 2a_{101} + \\
&\quad 2a_{011} + 4a_{100} + 1a_{010} + 2a_{001} + 1a_{000} = 11 \\
&3^2a_{200} + 2^2a_{020} + 2^2a_{002} + 3 \cdot 2a_{110} + 3 \cdot 2a_{101} + \\
&\quad 2 \cdot 2a_{011} + 3a_{100} + 2a_{010} + 2a_{001} + 1a_{000} = 2 \\
&4^2a_{200} + 1^2a_{020} + 3^2a_{002} + 4a_{110} + 4 \cdot 3a_{101} + \\
&\quad 3a_{011} + 4a_{100} + 1a_{010} + 3a_{001} + 1a_{000} = 10
\end{aligned}
\end{cases}$$

Its solution is $(1, 0, 0, -1, 0, 0, 0, 0, 0, -1, 1)$, which yields the following polynomial: $p(x, i, j) = x^2 - xi - j + 1$.

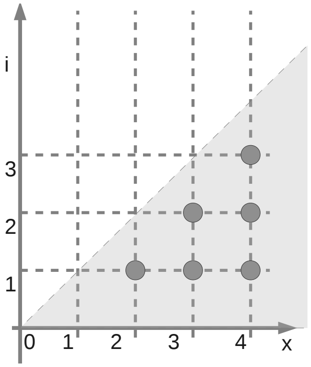


Figure 5.1: An example of well-chosen test nodes for a polynomial $g(x, i) = a_{20}x^2 + a_{11}xi + a_{02}i^2 + a_{10}x + a_{01}i + a_{00}$. These may be used to reconstruct a polynomial $g(x, i) = p(x, i, 1)$, where p is the polynomial bound for our running example. The grey area represents points that satisfy the condition $x < i$.

5.3 Inference of Ranking Functions for Loops

Our method is designed for loops with guards in the form of propositional logic expressions over numerical (in)equalities. Formally:

$$guard := inequality \mid inequality \land guard$$

$$inequality := num_1 \ \mathbf{b} \ num_2$$

where num_i is a numerical program variable or constant and where operator $\mathbf{b} \in \{<, >, =, \neq, \leq, \geq\}$.

For now, we limit our focus to loops where the loop guards are conjunctions over linear (in)equalities. The analysis of loops where the guard contains disjunctions is discussed in Chapter 6. Note that limiting the loop guards to linear (in)equalities does not mean limiting to linear RFs. The loop in Listing 5.2 has a non-linear RF for instance.

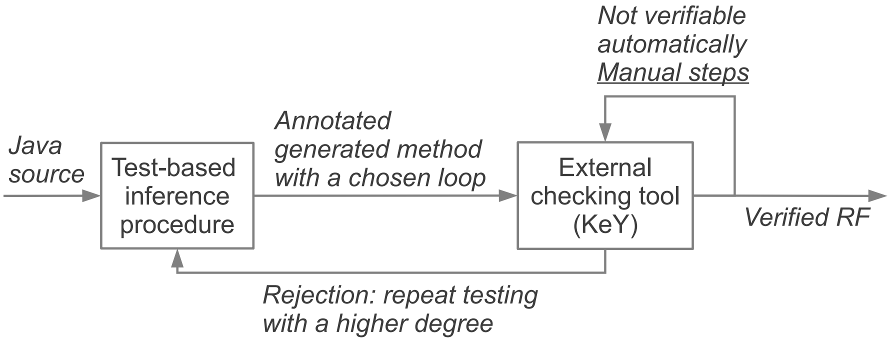
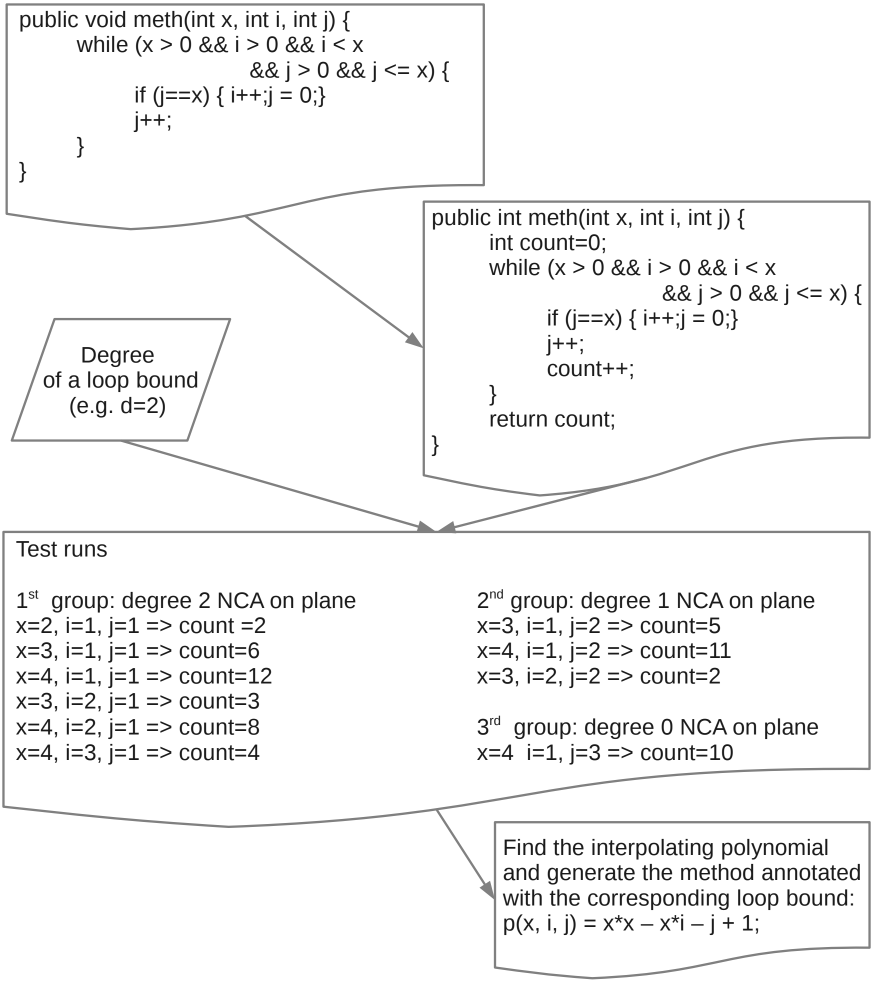


Figure 5.2: Test-based procedure from a bird's-eye view: infer-and-check cycle.

In Figure 5.2 we give a bird's-eye view of the test-based infer-and-check procedure. First, a user inputs the Java source code to the inference procedure. Then the procedure makes a hypothesis of a RF, based on test-runs. This hypothesis is expressed in some conventional annotations, like JML, so the annotated method can be read by an external checker that checks if the inferred bound is correct. Manual steps might be necessary to construct the proof. If the user concludes that such a proof cannot be found, (s)he might go back to the inference procedure and try again with a higher degree of a polynomial RF.

In Figure 5.3, we zoom in on the test-based inference module. We start with Java source code and pick a loop for which one wants to infer a RF. The loop is (automatically) inserted in a new method and instrumented with a counter, which is returned at the end of the method. The parameters of the method are the numerical variables that occur in the loop guard and in its body. The new



```
public void meth(int x, int i, int j) {
        while (x > 0 && i > 0 && i < x
                              && j > 0 && j <= x) {
                if (j==x) { i++;j = 0;}
                j++;
        }
}
```

```
public int meth(int x, int i, int j) {
        int count=0;
        while (x > 0 && i > 0 && i < x
                              && j > 0 && j <= x) {
                if (j==x) { i++;j = 0;}
                j++;
                count++;
        }
        return count;
}
```

Test runs

1st group: degree 2 NCA on plane
x=2, i=1, j=1 => count =2
x=3, i=1, j=1 => count=6
x=4, i=1, j=1 => count=12
x=3, i=2, j=1 => count=3
x=4, i=2, j=1 => count=8
x=4, i=3, j=1 => count=4

2nd group: degree 1 NCA on plane
x=3, i=1, j=2 => count=5
x=4, i=1, j=2 => count=11
x=3, i=2, j=2 => count=2

3rd group: degree 0 NCA on plane
x=4 i=1, j=3 => count=10

Find the interpolating polynomial and generate the method annotated with the corresponding loop bound:
p(x, i, j) = x*x – x*i – j + 1;

Figure 5.3: Test-based inference module in more detail. The choice of test nodes is explained in Section 5.2.

method is now executed for a given degree and an appropriate set of values of these parameters, i.e. on so called *test nodes*. For instance, in our running example $(x = 2, i = 1, j = 1)$ is an admissible test node. A well-chosen complete set of test nodes for this loop is given in the figure. The set consists of 10 nodes, since a polynomial of degree 2 of 3 variables has 10 coefficients: $p(x, i, j) = a_{200}x^2 + a_{020}i^2 + a_{002}j^2 + a_{110}xi + a_{101}xj + a_{011}ij + a_{100}x + a_{010}i + a_{001}j + a_{000}$. The result of a test run is the number of iterations for the corresponding node. For instance, with $(x = 2, i = 1, j = 1)$ the loop body is executed 2 times, so the test method returns `count` = 2. From the results of the test-runs a polynomial over the parameters can be calculated which interpolates the test results.

Multiple tactics are possible to guess the degree of the polynomial. It can be left to the user to supply it as input to the procedure, or an increasing degree can be tried, up to a certain bound. When a degree that is too low is supplied, the method will still find a RF, but the checker will reject it. When a degree that is too high is given, the right polynomial will still be found, but more test-runs are needed.

5.3.1 RF Inference: The Basic Method

This polynomial interpolation method was previously applied to infer output-on-input data-structure size relations in a functional language [SvEvK09, vKSvE08]. The *main challenge* we face when we adjust the interpolation theory to inferring ranking functions for loops in imperative programs is that test data must not only lie on a grid (or more generally, be in NCA), but also satisfy the loop guard C. In Figure 5.1 we show the set of points satisfying the (in)equalities $i < x$, $i > 0$ and $x > 0$. This corresponds to the loop guard in our running example for the fixed $j = 1$. Whenever the loop guard is violated the loop is not executed and the testing method, which is wrapped around the loop, outputs 0. Therefore, if we to construct the interpolation polynomial using a node(s) that does not satisfy the loop guard, we would obtain an incorrect loop bound for sure.

The problem of generating test data for imperative loops is formalised as follows:

Given:

- *a degree d,*
- *the number of variables k, on which the ranking function depends,*
- *a loop condition C,*

find N_d^k nodes in NCA that satisfy C.

We have reduced this task to the following one: *construct an integer grid in $\mathcal{R}^k$, such that it is based on $d+1$ parallel hyperplanes and contains N_d^k nodes, where*

- *there are N_d^{k-1} nodes in $(k-1)$-dimensional NCA for the degree d that lie on one of the hyperplanes and satisfy the corresponding projection of C to this hyperplane,*
- *there are N_{d-1}^{k-1} nodes in $(k-1)$-dimensional NCA for the degree $d-1$ that lie on another hyperplane and satisfy the projection of C on this hyperplane,*
- *there are N_{d-i}^{k-1} nodes in $(k-1)$-dimensional NCA for the degree $d-i$ that lie on a fresh hyperplane and satisfy the projection of C on this hyperplane, $0 \leq i \leq d$,*
- *and the remaining 1 node lies on the remaining hyperplane and satisfies the corresponding projection of C.*

In general terms, our approach is based on a search of the appropriate nodes on hyperplanes $x_1 = i_0, \dots, i_d$. The search is inductive on the number of variables k. To bound the search space one uses an external optimisation procedure solving tasks of the form $f(x_1, \dots, x_k) \to \min$, where $x_1, \dots, x_k$ satisfy the constraint $C(x_1, \dots, x_k)$. Currently, in our prototype, we use a linear programming solver. Therefore, the prototype handles only linear loop guards. Note that this does not limit the generated ranking functions to linear ones. In general, one may use non-linear optimisation software, such as the implementation of the Augmented Lagrangian Genetic Algorithm (ALGA) by MathWorks[7] or the open-source Java package Sigoa[8].

The rest of this subsection is structured as is the inference procedure: generating test-nodes, conducting the tests and interpolating a polynomial RF.

The algorithm for generating test-nodes

1. Run the chosen optimisation procedure for the objective functions $x_i \to$ min, $x_i \to$ max and the constraints constituted from the loop guard and the additional bounds $m_i \leq x_i \leq M_i$, where m_i and M_i are pre-defined minimal and maximal admissible values, respectively, of the variables x_i, with $1 \leq i \leq k$.
   The results define the k-dimensional box that bounds the set defined by C (within the minimal-maximal values). We only look for nodes inside this box, because we know that others do not satisfy the loop condition.
2. Obtain search hyperplanes H_j by cutting the bounding box on d congruent "slices", $j = 0, \dots, d$.
3. Amongst these hyperplanes, search for one that contains N_d^{k-1} nodes in $(k-1)$-dimensional NCA for the degree d and the projection of C on this hyperplane, et cetera, as explained above.
4. If the search succeeds, then stop. Otherwise, refine the grid by increasing the number of hyperplanes (i.e. by decreasing the distance between them) and repeat the search for the refined grid.

This procedure finds test-nodes that both satisfy the loop condition and lie in NCA, if they exist on a grid within the minimal-maximal values m_i and M_i. It finds suitable test-nodes for the case-study examples. To refine the search algorithm, one may add other kinds of NCA configurations than a rectangular grid, such as the *pencil* configuration.

Run tests Now that suitable test nodes have been selected, we can run the tests. Of course, because the investigated loops are actually executed, termination of the inference procedure depends on termination of those loops. The RF inference procedure terminates *if* the considered loop terminates for all inputs.

[7] http://www.mathworks.com/access/helpdesk/help/toolbox/gads/bqf8bdd.html

[8] http://sigoa.sourceforge.net/

Assuming that infinite loops are undesirable in general, but especially for loops for which one seeks to bound the number of iterations, "finding" non-termination for certain inputs is a valuable result in itself. An implementation can never conclude non-termination, but it may quit execution after a particular amount of time has passed and hint the user that there is a large chance that the loop does not terminate on the considered inputs.

Find the interpolating polynomial When all the tests have produced iteration counts (i.e. all have terminated), then we can now fit a polynomial, which interpolates these results. Because the test-nodes satisfy NCA, we know that a single interpolating polynomial exists.

5.3.2 Expressing the RF in JML

In this section we discuss how we can express the found RF in JML, in order to be verified by an external tool.

The result of our method is JAVA code annotated with JML, in which the inferred RF is expressed. Ranking functions for loops are most easily expressed in JML by defining a `decreases` clause on the loop. This is an expression which must decrease by at least 1 on each iteration, and remains greater than or equal to 0, see the JML reference manual [LPC$^+$07]. It therefore forms an upper-bound on the number of iterations.

We want to verify the RF for the case where the loop condition initially holds, otherwise the `decreases`-clause is not guaranteed to be greater than or equal to 0 initially (and the loop will iterate exactly 0 times). Therefore, the loop condition is added as a precondition to the constructed method. The example from Listing 5.2 is shown in annotated form in Listing 5.3.

```
/*@
 requires x>0 && i>0 && i<x && j>0 && j<=x;
 ensures true;
*/
public void meth(int x, int i, int j) {

  //@ assignable i,j;
  //@ loop_invariant true;
  //@ decreases x*x - x*i - j + 1;
  while (x>0 && i>0 && i<x && j>0 && j<=x) {
    if (j==x) { i++; j = 0;}
    j++;
  }
}
```

Listing 5.3: The inferred RF for the example in Listing 5.2 expressed as a JML annotation.

5.3.3 Complexity: Exponential in the Number of Variables

The first sub-procedure in the presented inference method is an external optimisation procedure used to bound the test-nodes search space. Typically, the complexity of optimisation methods depends on the number of (in)equations in the constraints, number of variables (the space's dimension) and complexity of (in)equations. For non-linear constraints the worst-case complexity is, as a rule, exponential, but one often uses "smart search" algorithms providing better average computation time. For instance, in genetic algorithms the search is directed by e.g. the value of a penalty function that decreases when one searches in the "right direction".

For the remaining parts of the inference method we can give independent estimations of complexity. These parts are:

- the search of test nodes that, as one intuitively expects, has the most significant complexity, which we will discuss right now, below,
- the runs ($N_d^k = \binom{d+k}{k}$ times) of the test method on the test nodes,
- solving a system of N_d^k linear equations w.r.t. N_d^k variables that has the worst complexity $O((N_d^k)^3)$; with some advanced matrix-multiplication algorithms the complexity may be between $O((N_d^k)^2)$ and $O((N_d^k)^3)$.

Searching of test nodes is the most time-consuming part of the inference procedure (besides, probably, non-linear optimisation part). Let $\mathcal{N}(d,k)$ denote the time for finding the nodes for a polynomial of the degree d with k variables. Consider its behaviour from the best to the worst case, with $\mathcal{N}_{\min}(d,k)$ denoting the best computation time.

In the *best case* we just cut the k-dimensional cube by $d+1$ hyperplanes of the dimension $k-1$, and find immediately N_{d-i}^{k-1} points on the i-th hyperplane in time $\mathcal{N}_{\min}(d-i,k-1)$, where $0 \leq i \leq d$. Therefore, we may assume that $\mathcal{N}_{\min}(d,k-1) = \mathcal{N}_{\min}(d,k-1) + \mathcal{N}_{\min}(d-1,k-1) + \ldots + \mathcal{N}_{\min}(1,k-1) + \mathcal{N}_{\min}(0,k-1) + (d+1)$ that includes the time for $d+1$ recursive calls. We can show by induction on k that $\mathcal{N}_{\min}(d,k) = O\big(\frac{d^k}{k!}\big)$. Indeed, for $k=1$ we have to pick up $d+1$ different points on the line, so $\mathcal{N}_{\min}(d,1) = d+1 = O(\frac{d}{1})$. For $k=2$ we have $\mathcal{N}_{\min}(d,2) = (d+1) + d + \ldots + 1 + (d+1) = \frac{d(d+1)}{2} + (d+1) = O(\frac{d^2}{2})$. Using the induction assumption, $\mathcal{N}_{\min}(d,k) = \sum_{i=0}^{d} O(\frac{(d-i)^{k-1}}{(k-1)!}) + (d+1) = O\big(\frac{1}{(k-1)!}\sum_{i=0}^{d} j^{k-1}\big) + (d+1) \approx O\big(\frac{1}{(k-1)!}\int_0^d x^{k-1}dx\big) + (d+1) = O(\frac{d^k}{k!})$.

In the *"middle" case* the initial collection of $(d+1)$ hyperplanes does have all the points in the necessary configuration, but, roughly, one has to reorder hyperplanes to get the k-dimensional NCA configuration. That is, the $i=0$-th hyperplane does not contain enough, i.e. N_d^{k-1}, $(k-1)$-dimensional points, so in general we have to look through all $d+1$ hyperplanes. Next, for N_{d-1}^{k-1} points we have to search in d remaining hyperplanes, etc. So, for N_{d-i}^{k-1} points we search in $d+1-i$ hyperplanes. Therefore, $\mathcal{N}(d,k) = \sum_{i=0}^{d}\big((d+1-i)(\mathcal{N}(d-i,k-1)+1)\big)$, including the recursive calls (with "+1" staying for the recursive call of the procedure for $d-i, k-1$). Then, the estimate is

$$\begin{array}{l}
\mathcal{N}(d,k) \leq \\
\sum_{i=0}^{d}\big((d+1-i)(\mathcal{N}(d,k-1)+1)\big) = \\
(\mathcal{N}(d,k-1)+1)\sum_{i=0}^{d}(d+1-i) = \\
(\mathcal{N}(d,k-1)+1)O(\frac{d^2}{2}) \leq \\
(\mathcal{N}(d,k-2)+1)O(\frac{d^4}{4}) + O(\frac{d^2}{2}) = \\
O((\frac{d^2}{2})^k)
\end{array}$$

Now, it is clear that the *the worst-case* computation time of node search is exponential in k. Different versions of the search procedure provide different bases of the exponent or differ by a multiple, that may be quite large. Here we consider one of the versions (implemented in the prototype) with accelerated generation of new collections of hyperplanes. In the worst case, if we fail to find enough nodes w.r.t. the current collection of hyperplanes, we have to generate another collection of $D > d+1$ hyperplanes for a refined grid. Similarly to the estimates above, the estimate is $\mathcal{N}(d,k) = \sum_{i=0}^{d}(D+1-i)(\mathcal{N}(d-i,k-1)+1) \leq (\mathcal{N}(d,k-1)+1)O\big(\frac{D^2}{2}\big) = O\Big(\big(\frac{D^2}{2}\big)^k\Big)$. After failing with the first hyperplane collection, D takes consecutively the values $2(d+1)$, $2^8 2(d+1)$, ... $2^{8i+1}(d+1)$, with $0 \leq i \leq i_{\max}$ and for $i_{\max}$ the following holds. It is such that $2^{8i_{\max}+1}(d+1) \leq M+1$, where M is the (length of the) side of the bounding box, generated by the optimisation procedure on the first step. So, we obtain that $i_{\max} \leq \dfrac{1}{8}(\log_2 \frac{M+1}{d+1} - 1)$. The worst-case time, when we have to go through all the possible cuts, is then

$$\begin{array}{l}
\sum_{i=0}^{i_{\max}}\Big(\big(\frac{(2^{8i+1}(d+1))^2}{2}\big)^k\Big) = \\
O(2^k(d+1)^{2k})\sum_{i=0}^{i_{\max}} O(2^{16k})^i = \\
O\big(2^k(d+1)^{2k}\frac{(2^{16k})^{i_{\max}+1}-1}{2^{16k}-1}\big)
\end{array}$$

Taking into account the estimate for $i_{\max}$ we obtain that $\mathcal{N}(d,k)$ does not exceed

$$O\Bigg(\Big(\frac{2}{2^{16}}(d+1)^2\Big)^k\Big(\frac{M+1}{2(d+1)}\Big)^{2k}\Bigg) = O\Bigg(\Big(\frac{1}{2^{17}}(M+1)^2\Big)^k\Bigg)$$

5.4 Prototype and Case Studies

We have created a prototype implementation of the method in Java. This prototype is embedded in the tool ResAna and can be used to load Java source files, select a loop to analyse, input an expected degree, infer a ranking function for the loop and output Java code containing JML annotations in order to prove this inferred RF using an external tool, for instance KeY [BHS07] or OpenJML [Cok11].

For the prototype, existing software packages were used as much as possible, for instance for bounding the test-node search space and for solving the interpolation matrix. Around 3000 lines of code were added to create a working prototype, including a graphical user interface.

JML annotations can be generated for all of the loops listed in this chapter. We were able to prove all the inferred RFs using KeY. Additionally, we have conducted three case studies of safety-critical Java systems, suggested as test cases by the CHARTER partners.

- **Collision detector case study from [HSST06].** The first case is the collision detector example from the paper "Provably Correct Loop bounds for Realtime Java Programs" by James Hunt et al. This code stems from a safety-critical avionics application.
- **DIANA Package.** This package is developed in the FP6 project *Distributed, equipment Independent environment for Advanced avioNics Applications* (DIANA)[9]. The package is described in detail in [SJL+09].
- **CD$_X$ Collision Detector package.** The CD$_X$ Collision Detector package[10] is a publicly available Realtime Java benchmark. It is described in [KHP+09].

	Nr. of loops	Analysable	Percentage
Hunt et al	2	2	100%
DIANA	4	4	100%
CD$_X$	38	23	61%
Total	44	29	66%

Table 5.1: Summary of the cases studied.

The results are shown in Table 5.1. As can be read from the table, we can handle roughly two-thirds of the loops found in the case studies. This means that we can infer a RF for these loops using our prototype and prove it using KeY. All of the found RFs were linear, i.e. of degree one.

In the case studies, apparently, enough test-nodes are found after just a few cuts of the k-dimensional search space. This leads us to believe that the average complexity of the method lies somewhere around $O(D^{2k})$ for $D = 2^9(d+1)$, rather than near the worst-case complexity. For the examples in the case studies this amounts to approximately one second spent in RF inference. KeY was able to prove all the RFs fully automatically, for which it requires approximately 5 to 10 seconds.

There are 15 examples in the case studies that cannot yet be analysed using our method. In these cases, the loop bound depends on Booleans, array

[9] `http://diana.skysoft.pt/`

[10] `http://adam.lille.inria.fr/soleil/rcd/`

elements, fields of referenced objects, a method invocation or results from a different thread. We do not support such cases at this point. The first four limitations are left for future work. Loops in which results from a different thread are used require a fundamentally different analysis, as loop duration cannot be captured in a ranking function that consideres the loop in isolation.

For examples that can be handled by our method, it usually computes the *exact* RF. An exception to this is when branch-splitting is applied. This means that compared to other methods, our method finds bounds that are equally tight, or tighter. Furthermore, other methods are unable to derive non-linear RFs. This is discussed in more detail in the next section.

5.5 Related Work

Various other research results on bounding the number of loop iterations exist. However, most are concerned with concrete (numerical) bounds, instead of symbolic bounds. Also, most can only handle (tightly) cases where the bound depends linearly on program variables (we can handle the polynomial case). In a sense, our technique is more general than the methods discussed in this section. It may not be the most efficient method for simple loops, but it can be used to handle certain more complex cases. This makes it complementary to the other techniques discussed here.

Another common difference is that other approaches rely on hand-made soundness proofs of their method, while we rely on a verification tool to ensure that the derived RFs are correct.

In [FJ10], pattern-matching on abstract syntax trees (ASTs) is used by Fulara et al. to select one of several syntax-based schemes for generating `decreases`-clauses. If the AST matches a known pattern, it can be used to form a `decreases`-clause. The authors claim to cover 71% of all for-loops in a set of case studies. It is thinkable that their method is used in an implementation for the basic cases and our method is applied when no pattern matches.

Abstract interpretation, program slicing and invariant analysis are used by Ermedahl et al. in [ESG$^+$07] to infer numerical bounds for C programs. The bounds meant here are integers representing the number of times a certain block of code is executed. The method can infer bounds for over 50% of the loops in a set of benchmarks.

A similar approach is taken by Lokuciejewski et al. in [LCFM09], who combine abstract interpretation with polytope models to calculate numerical loop bounds for C programs. Both upper and lower bounds are calculated and the analysis is accelerated by using program slicing. Even though there are restrictive constraints on the loops that can be analysed, the authors claim that they can handle 99% of all for-loops in a set of benchmarks. Soundness or verification of the bounds are not discussed.

Abstract interpretation is also used in [DMBCS08], in combination with flow analysis. Numerical bounds can be found for 84% of the loops in a benchmark suite. The method works on C programs.

Gulwani uses “off-the-shelf linear invariant generation tools” to compute symbolic loop bounds in [Gul09]. The authors experiment with different counter instrumentation methods and a technique they named “control-flow refinement”. Symbolic loop bounds are presented as right-hand sides of the inequations in loop invariants. Inference of invariants is based on linear arithmetic, but some limited use of non-linear terms is possible as well. Given a particular program, the base arithmetic may be extended by a finite set of non-linear operators together with reasoning rules for them. The inference system, first, introduces a fresh variable for each non-linear operator, then deals with linear combinations of such variables (and usual arithmetic variables). The operators and the rules are chosen e.g. by a user, who knows which sort of invariants one can expect in the given code.

In a related article [GJK09], pattern-matching against known loop-iteration lemmas is used to establish bounds for C and C++ programs. This last method can find bounds for 93% of the loops in a significant Microsoft product.

In [BA09], Ben-Amram describes a method to derive *global* ranking functions, based on Size-Change Termination. Such a ranking function is required to decrease in each basic block of the program. He uses an abstraction called Monotonicity Constraints and represents them as graphs. Various algorithms are described that can be applied to these graphs to judge termination and construct ranking functions.

Hunt et al. discuss the expression of manually conceived ranking functions for JAVA loops in JML, their verification using KEY and the combination with data-flow analysis in [HSST06]. This article is an important motivation for our work. What is “missing” in the method is the automated inference of ranking functions for loops, which we supply.

In [AAGP08], Albert et al. describe a system of generating and solving cost recurrence relations. These relations define functions that represent upper bounds on time or memory usage by a program. To solve a recurrence relation means to find a closed, i.e. a recursion-free, form of the corresponding function. Terms in the system represent *monotonic* real functions and, besides monotonically increasing polynomials, may contain the exponent and the logarithmic functions.

5.6 Conclusions

We have presented a method of computing arbitrary degree ranking functions for JAVA loops. By expressing these functions in JML, their correctness can be proved, which is very valuable in safety-critical systems. While various other methods for inferring loop-bounds exist, we are not familiar with any other works on generating non-linear ranking functions for JAVA loops. Moreover, the technique presented herein is largely complementary to other methods, since it is more general and can solve certain more complex cases, such as quadratic bounds. Using a prototype implementation, ranking functions can be inferred for 66% of all loops in a set of case studies from actual safety-critical systems.

Chapter 6

ResAna: A Resource Analysis Toolset for (Real-Time) Java

Abstract. For real-time and embedded systems limiting the consumption of time and memory resources is often an important part of the requirements. Being able to predict bounds on the consumption of such resources during the development process of the code can be of great value. In this paper we focus mainly on memory related bounds.

Recent research results have advanced the state of the art of resource consumption analysis. In this paper we present a toolset that makes it possible to apply these research results in practice for (real-time) systems enabling Java developers to analyse symbolic loop bounds, symbolic bounds on heap size and both symbolic and numeric bounds on stack size. We describe which theoretical additions were needed in order to achieve this.

We give an overview of the capabilities of the ResAna toolset that is the result of this effort. The toolset can not only perform generally applicable analyses, but it also contains a part of the analysis which is dedicated to the developers' (real-time) virtual machine, such that the results apply directly to the actual development environment that is used in practice.

6.1 Introduction

Both in industry and in academia there is an increasing interest in more detailed resource analysis bounds than orders of complexity. In correctness verification for industrial critical systems, the focus is often mainly on functional correctness: does the program deliver the right output with the right input. However, for such systems it is just as important to make sure that bounds for the consumption of time and space are not exceeded. Otherwise, a program may not react within the required time or it may run out of memory and come to a halt (making it vulnerable to a Denial Of Service attack).

Traditionally, the focus has been on performance analysis taking time as resource which is consumed. More recently, several researchers have produced significant results in heap and stack bound analysis. In this chapter we focus on such *memory related resource analysis*. The symbolic loop bound analysis part however may be used both for memory and for time analysis.

Many real-time and embedded systems critically depend on operating within a fixed amount of memory. Clearly, for such systems it can be important to know an upper bound on the consumed memory. For safety critical applications it can be essential. Programmers may be able to guess a bound and to prove it by hand. That activity is quite tedious and error-prone. A tool that in many cases is able to automatically infer bounds and prove them may be very helpful in the software development process. This chapter presents such a tool.

For safety-critical applications often domain specific programming languages are used that have strong support for loop bounding or regular programming languages with strict coding conventions. In the recently finished EU Artemis CHARTER (**C**ritical and **H**igh **A**ssurance **R**equirements **T**ransformed through **E**ngineering **R**igour) project, REALTIME JAVA was considered as possible programming language for safety-critical systems. Reasons for studying REALTIME JAVA include more possibilities for code reuse, more available tools and more programmers that are highly experienced in the use of the language. The RESANA toolset, which is presented in this chapter, is one of the results of the CHARTER project [dRH12, WW12, KSvG$^+$12]. Together, the tools produced by the CHARTER project provide a first step towards the use of general programming languages for safety-critical systems. For full deployment in safety-critical context the CHARTER tool chain should be advanced further. For now, the RESANA toolset can already be used in everyday practice, e.g. for inferring and proving memory consumption properties of existing library functions and of non-critical applications for which memory bounds are relevant like applications for mobile devices. Another usage may be the development of prototype applications with verified resource consumption properties. These prototypes can then be transformed to the language that is in actual use for the safety-critical system. The techniques presented in this chapter can in principle be used for other languages too. Of course, that would require both an adaptation of the front-end of the tool and of the annotation language that is used for expressing the properties.

Even if memory is abundantly available, applications can be hindered significantly when more memory is consumed than expected. Effectively the system may come to a halt due to excessive swapping. Some Denial-Of-Service attacks are based on this principle. A known upper bound of consumed memory may prevent attacks of that kind.

A variety of memory analysis techniques have been developed independently not only on the language level but also on the byte code level [AAGP11]. Researchers use polynomial interpolation [vKSvE08], reachability-bound analysis [GZ10], amortization [HAH11], polynomial quasi-interpretation [Ama05] and new language features such as programmer-controlled destruction and copying of data structures [dDMP10]. Of course, such analyses are undecidable in general. In practice, however, an increasingly large set of problems can be handled.

This research builds upon earlier resource analysis work developed in the Dutch NWO AHA project [vESvK+07], as well as on Chapter 5. In this chapter, we focus on the JAVA language and on resource consumption properties related to heap and stack usage. Using the scoped memory which is offered by REALTIME JAVA one can enforce memory bounds and facilitate simple memory management. However, in order to deal with more complex bounds, a more thorough analysis is needed. While our research mainly focuses on REALTIME JAVA, the techniques and the tool described here are also applicable to regular JAVA programs. The loop bound analysis provided by the RESANA tool can be of further use both for deriving memory bounds and for deriving time bounds. This chapter is an extended version of [KSvG+12]. How this chapter extends [KSvG+12] is described in Section 6.6.

With the goals of making these results applicable in practice, our heap and stack resource analysis goes beyond orders of complexity. We aim at obtaining bounds that are expressions of relevant variables and parameters. If a resource is consumed quadratically with respect to the value of a parameter x, then a typical bound could be e.g. $2x^2 - 4x + 15$ thus indicating the exact dependency of the bound on the variable. In order to achieve that in practice, we developed a tool, RESANA[11], that contains a general process which has two phases.

Inference In the inference phase the RESANA tool analyses the JAVA source of the program in order to propose a possible resource bound for the program. It uses traditional analysis techniques like solving cost-relation systems and a novel polynomial interpolation technique. This interpolation-based approach is very powerful. It allows also non-monotonic polynomial bounds to be derived (the developer does not have to indicate the exact dependencies: they are derived). The obtained result is added to the JAVA program via an annotation using the JML specification language [LPC+07].

Verification Results are achieved by solving cost relations or by interpolating polynomials. Solving cost relations is sound by construction. The use of interpolation is not guaranteed to be sound. Therefore, the results achieved by interpolation must be verified, e.g. by the KEY verification tool [BHS07] or the QEPCAD algebraic decomposition tool [Bro03]. If the tool is not able to verify them, one can proceed with a new inference phase with other user options, such as e.g. trying a higher degree polynomial.

The RESANA tool supports three kinds of analysis.

Loop Bound Analysis An expression that gives a symbolic upper bound for the number times a loop is executed may be derived and verified using the integrated combination of the tools RESANA and KEY, as in Chapter 5.

Heap Bound Analysis An expression for a symbolic upper bound of the consumed heap is derived using RESANA extended with a variant of the external

[11] RESANA is open source software and can be downloaded from `http://resourceanalysis.cs.ru.nl/resana/`.

tool COSTA [AAG^{+}08]. The COSTA tool has been adapted to produce accurate values for OpenJDK, as well as the real-time JamaicaVM virtual machine [Sie02]. Furthermore, the capabilities of the COSTA tool have been enlarged through the internal use of interpolation technology [MSvEPn12].

Stack Bound Analysis An expression for a symbolic upper bound of the space for the stack is derived using ResAna with the enlarged COSTA that provides an upper bound for the depth of recursive calls; this information is used by the VeriFlux tool [HTS08] to obtain a *numeric* stack bound.

These three kinds of analysis are integrated in a common program development environment through an Eclipse plug-in, such that a developer can easily switch between development and verification activities guaranteeing the memory safety of critical real-time software applications.

In Section 6.2 loop bound analysis is described. Section 6.3 presents heap bound analysis and the adjustments that have been made to make it applicable in practice. Analysing stack bounds is discussed in Section 6.4. User experience with ResAna is described in Section 6.5. Finally, in Sections 6.6 and 6.7, related work is discussed and conclusions are drawn.

6.2 Loop-Bound Analysis

In order to prove the termination of a piece of software or, even harder, to calculate bounds on run-time or usage of resources such as heap space or energy, finding bounds on the number of iterations that the loops can make is a prerequisite. While in some cases a loop may iterate a fixed number of times, its execution will often depend on program input. Therefore we consider *symbolic* loop bounds, or *ranking functions*. In Chapter 5, a method for inferring polynomial ranking functions for loops is presented. Here, we present a series of extensions to that *basic method*.

In Section 6.2.1, the basic method is extended in order to deal with ranking functions with rational or real coefficients. Section 6.2.2 presents an extension for loops with branching inside the body. In Section 6.2.3, an extension which handles loop guards which contain disjunctions is discussed. In Section 6.2.4 a limitation to the extension for disjunctional loop guards and a solution are discussed. Another application of our polynomial interpolation method is discussed in Section 6.3.1.

6.2.1 Ranking Functions with Rational or Real Coefficients

The ranking functions inferred by the basic method are polynomials with coefficients that are natural, rational or real numbers. However, when a polynomial has rational or real coefficients, its result is not necessarily a natural number, which, of course, any estimate of a number of loop iterations must be. Consider for instance the loop in Listing 6.1.

```
while (start < end) {
  start += 4;
}
```

Listing 6.1: An example with a loop-bound function that is a polynomial over rational coefficients

The exact number of iterations of this loop is given by $\lceil \frac{\texttt{end}-\texttt{start}}{4} \rceil$. In other words, when $\frac{\texttt{end}-\texttt{start}}{4}$ does not equate to a natural number, for instance to $\frac{3}{4}$, it must be *ceiled*. In general, when the coefficients of an inferred polynomial ranking function $RF(\bar{v})$ are not natural numbers, ceiling should be added as such: $\lceil RF(\bar{v}) \rceil$. Unfortunately, there is no ceiling operator in JML. KeY simply truncates non-integer values after the decimal. We therefore chose to overestimate ceiling by adding one to the KeY truncation: $\lceil RF(\bar{v}) \rceil \leq RF(\bar{v}) + 1$.

When choosing test nodes for the loop in Listing 6.1 naively, for instance (0,1), (1,2) and (1,3), an incorrect ranking function will be the result (in this case the constant 1). We must take into account that if a variable v is updated by increasing or decreasing by a constant *step*, the test-nodes must lie *step* apart. In this example, if we pick test nodes (0,4), (4,8) and (4,12), then the correct ranking function will be found.

6.2.2 Branching inside the loop body

The basic procedure finds correct ranking functions for most loops containing branching, such as for example the one in Listing 5.2 (Chapter 5). However, there are cases in which the basic procedure fails, because the different paths affect the bound in different ways. Such a case is shown in Listing 6.2.

```
while (i > 0)
  if (i > 100) i -= 10;
  else i -= 1;
```

Listing 6.2: Example where the basic method supplies an incorrect ranking function. Therefore, branch-splitting is applied, yielding the pessimistic, but correct ranking function i.

To solve this problem, we have invented *branch-splitting*. This procedure finds ranking functions for loops where the if-statements, if they exist in a loop body, have the following *worst-case computation path (WCCP) property*:

For each loop body, there is an execution path such that, for any collection of values of the loop variables, if one follows this execution path in every loop iteration one reaches the worst-case, i.e. the upper bound on the number of iterations.

The WCCP property is not checked by the loop bound inference procedure. It is given here to specify the class of loops for which the procedure is successful. Soundness of the result is ensured by verification using KeY.

By branch-splitting, we mean that we generate multiple new loops from the original, one for each possible path. We then do the analysis for each of these paths. The ranking function is then the maximum of all the inferred ranking functions. Thanks to the WCCP property, we can easily find the ranking function that always specifies the maximum, by supplying a set of values for the variables (say, all ones) to all the ranking functions. For the example in Listing 6.2, this yields the ranking function i.

6.2.3 Piecewise Ranking Functions for Loops with Disjunctive Guards

In this section, we formally describe an extension to the basic procedure for handling loops with disjunctions in their guards. The set of considered loops is here thus extended to those with as guard *any* propositional logical expression over arithmetical (in)equalities, including disjunctions. We will see that for those loops for which the guard contains disjunctions, the ranking function will become *piecewise*.

Note that in fact, any ranking function for a well-formed loop is a piecewise one, since there is always the piece where the loop guard does not hold and the loop iterates zero times. For instance, for the loop in Listing 5.1 (Chapter 5), the ranking function is actually:

$$\begin{cases} 15 - \mathtt{i} & \text{if } (\mathtt{i} < 15) \\ 0 & \text{else} \end{cases} \tag{6.1}$$

This is of course a trivial case. A more involved example of a loop for which a piecewise ranking function can be defined is shown in Listing 6.3.

```
while ((i>0 && i<20) || i>50) {
  if (i>50) i--;
  else i++;
}
```

Listing 6.3: While loop with a piecewise ranking function.

Its ranking function is the following:

$$\begin{cases} 20 - \mathtt{i} & \text{if } (\mathtt{i} > 0) \wedge (\mathtt{i} < 20) \\ \mathtt{i} - 50 & \text{if } \mathtt{i} > 50 \\ 0 & \text{else} \end{cases} \tag{6.2}$$

We will now formally define a generic method for inferring ranking functions for loops with disjunctive guards. The first step is to transform the guard into disjunctive normal form (DNF), using the laws of distribution and DeMorgan's theorems. Thereafter it has the form:

$$guard := conj \mid conj \vee guard$$
$$conj := inequality \mid inequality \wedge conj$$
$$inequality := num_1 \ \mathbf{b} \ num_2$$

where num_i is a numerical program variable or constant and where operator $\mathbf{b} := \{<, >, =, \neq, \leq, \geq\}$.

Let c_i represent a logical conjunction over numerical (in)equalities. We can now split up the guard by applying the following function:

$$DNFsplit(c_1 \vee \ldots \vee c_n) := \left\{ \bigwedge_{c_i \in CP} c_i \wedge \bigwedge_{c_j \in C_{rest}} \neg c_j \;\middle|\; \begin{array}{l} CP \in \mathcal{P}(\{c_1, \ldots, c_n\}) \backslash \emptyset \\ C_{rest} = \{c_1, \ldots, c_n\} \backslash CP \end{array} \right\}$$

This transforms the condition $c_1 \vee \ldots \vee c_n$ into a set *Pieces* of $2^n - 1$ conjunctive conditions. For instance, $DNFsplit(i > 10 \vee i < 3)$ yields three pieces: $i > 10 \wedge \neg i < 3$, $i < 3 \wedge \neg i > 10$ and $i > 10 \wedge i < 3$. This set may be simplified using a Satisfiability Modulo Theories (SMT) solver. In this case, the negations can be removed from the first two conditions. The third condition is unsatisfiable, thus it may be removed altogether. We refer to the procedure of transforming a guard into disjunctive normal form and separating the pieces as *DNF-splitting*. The set *Pieces* defines the pieces of the piecewise polynomial ranking function.

After DNF-splitting, the basic method can be applied separately to each of the pieces. If RF_p is the polynomial ranking function inferred for a piece $p \in Pieces$, then this yields the following piecewise ranking function:

$$\begin{cases} RF_{p_1} & \text{if } p_1 \\ \ldots & \text{if } \ldots \\ RF_{p_m} & \text{if } p_m \\ 0 & \text{else} \end{cases} \tag{6.3}$$

In this piecewise polynomial ranking function, $m \leq 2^n - 1$, because unsatisfiable pieces have been removed.

6.2.4 Condition Jumping

In this section we define a complication that may arise during DNF-splitting, which we call *condition jumping*. We show how to detect its occurrence and how to infer ranking functions even in the presence of condition jumping.

```
while ((i>0 && i<20) || i>22) {
  if (i>22) i--;
  else i+=4;
}
```

Listing 6.4: While loop with jumping between the disjunctive conditions.

Consider the loop in Listing 6.4. Naively, one could say that its ranking function is the following:

$$\begin{cases} \lceil(\texttt{20} - \texttt{i})/4\rceil & \text{if } (\texttt{i} > \texttt{0}) \wedge (\texttt{i} < 20) \\ \texttt{i} - \texttt{22} & \text{if } \texttt{i} > \texttt{22} \\ 0 & \text{else} \end{cases} \tag{6.4}$$

But, what if `i` is 19, 15, or any $n \in [1, 19]$ with $n \bmod 4 = 3$? Indeed, then there is a shift from the first condition $(0 < i < 20)$ to the second one $(i > 22)$. We call this *condition jumping.* Jumping from the second condition into the first one is not possible in this case.

Because of the presence of condition jumping, regular DNF-splitting does not suffice here. The set of nodes from which condition jumping occurs must be considered as a separate piece, as follows:

$$\begin{cases} \lceil(\texttt{20} - \texttt{i})/4\rceil + 1 & \text{if } (\texttt{i} > \texttt{0}) \wedge (\texttt{i} < 20) \wedge i \texttt{ mod } 4 = 3 \\ \lceil(\texttt{20} - \texttt{i})/4\rceil & \text{if } (\texttt{i} > \texttt{0}) \wedge (\texttt{i} < 20) \wedge i \texttt{ mod } 4 \neq 3 \\ \texttt{i} - \texttt{22} & \text{if } \texttt{i} > \texttt{22} \\ 0 & \text{else} \end{cases} \tag{6.5}$$

In the remainder of this section, we first describe a method to detect condition jumping. This method is then applied in an algorithm which detects all nodes for which jumping occurs, in order to infer a correct piecewise ranking function.

Detection of Condition Jumping using Symbolic Execution and SMT Solving To detect condition jumping in the example in Listing 6.4, we first use symbolic execution [Kin76] to construct an update function, which captures the relation between the values of the program variables pre and post execution of the loop body. We can then use this relation as input to an SMT solver and search for a model for which one part of the loop guard is true pre-execution of the loop body and another part is true post-execution.

Obtaining an update function. We will name the pre/post execution relation for a variable v the $next_v$ function. The function $next_i :: Int \rightarrow Int$ for the loop in Listing 6.4 can be determined by symbolically executing the loop with value α_i for i. This results in the following symbolic post-execution value, which we will name ϕ_i:

$$\phi_i(\alpha_i) = \begin{cases} \alpha_{\texttt{i}} - \texttt{1} & \text{if } \alpha_{\texttt{i}} > \texttt{22} \\ \alpha_{\texttt{i}} + \texttt{4} & \text{if } \neg(\alpha_{\texttt{i}} > \texttt{22}) \end{cases} \tag{6.6}$$

By replacing the α symbol by i, this easily translates to the $next_i$ function we were looking for:

$$next_i(i) = \begin{cases} \texttt{i} - \texttt{1} & \text{if } \texttt{i} > \texttt{22} \\ \texttt{i} + \texttt{4} & \text{if } \neg(\texttt{i} > \texttt{22}) \end{cases} \tag{6.7}$$

In general, such an update function can be derived by symbolic execution of the loop body. Start by giving the variables $v_1 \dots v_n$ symbolic values $\alpha_1, \dots, \alpha_n$. If we restrict our method to loop bodies with polynomial effects, after the symbolic execution of the loop body, each variable v_i will have a value which is a set of polynomials over the symbols $\alpha_1, \dots, \alpha_n$ and constants, with associated *path conditions*, which capture branching. Effectively, this is again a piecewise polynomial. The function $next_{v_i}$ is now obtained by replacing the α's by the corresponding program variables in this piecewise polynomial.

Detecting condition jumping. SMT-LIB is a library of SMT background theories and benchmarks [BST10]. It has a common file format for SMT problems, which can be read by most SMT-solvers. An SMT-LIB script to detect jumping in the example from Listing 6.4 is given in Listing 6.5. The function $next_i :: Int \to Int$ from Equation 6.7 is defined on line 2. Then on line 4 we define the condition expressing that jumping occurs for this example and on line 6 we check satisfiability of this condition.

```
1 (declare-fun i () Int)
2 (define-fun nexti ((x Int)) Int
3   (ite (> x 22) (- x 1) (+ x 4)))
4 (assert (and (and (> i 0) (< i 20))
5   (> (nexti i) 22)))
6 (check-sat)
7 (exit)
```

Listing 6.5: SMT-LIB script to detect jumping in the code of Listing 6.4.

Let us now consider the general case. Condition jumping will be detected pairwise for conditions with multiple disjunctions. Here we thus consider a single condition-pair, i.e. a loop with guard $b_1 \vee b_2$. Here b_1 and b_2 are conditions over $CV \subseteq LV \subseteq PV$, where CV are the program variables in the condition, LV are the program variables in the loop and PV are all program variables.

For each $v_i \in LV$, we can define an associated function $next_{v_i} :: T_{v_1} \to \dots \to T_{v_i} \to \dots \to T_{v_n} \to T_{v_i}$, where T_{v_i} is the type of v_i and $n = |LV|$, which takes the values of all $v \in LV$ as the state and computes the value of v after a single execution of the loop body in that state, by following the procedure described in the previous paragraph. Once these functions have been derived, the question whether jumping from b_1 to b_2 is possible can be answered by any SMT-LIB conforming SMT-solver[12] by determining the satisfiability of $b_1(v_1, \dots, v_n) \wedge b_2(next_{v_1}(LV), \dots, next_{v_n}(LV))$.

Generating Ranking Functions in the Presence of Condition Jumping

The SMT-LIB script in Listing 6.5 can be used to find a *model* for which jumping

[12] For instance Z3, which can be used on-line at `http://research.microsoft.com/en-us/um/redmond/projects/z3/`

occurs by adding the expression `(get-value (i))`. A model is an instantiation of the variables for which the formula for which satisfiability is checked holds. In the SMT-LIB script from Listing 6.5, a model for i is 19.

Subsequently, by adding the expression `(assert (distinct i 19))`, one can search for models *other than* $i = 19$ for which jumping occurs. The answer of the SMT solver is that the combination of propositions in this script is unsatisfiable. Thus, $i = 19$ is the only possible model. We can now see if there are any models from which the state $i = 19$ can be reached in a single iteration, by changing line 6 to `(= ( nexti i) 19)))`. In the example, this will be the model $i = 15$. Subsequently and similarly, we can search for other nodes that can reach the state $i = 19$ in a single step, or that can reach the state $i = 15$. By repeating these steps, we can find the set $J = \{3, 7, 11, 15, 19\}$. These are the models from which jumping can occur.

In general, the method described above can be extended to detect all models from which condition jumping can occur, by first finding all models that can jump directly from b_1 to b_2 and then recursively finding models that can reach a model from this first set. This can be done by implementing the following algorithm around an SMT-solver. In this algorithm, J is the set of models of which it is known that condition jumping occurs and Q is a queue of models. We assume a function $next :: M \to M$ (where M is the type of a model), which applies to each variable v_i in a model $\bar{v}$ its corresponding $next_{v_i}$ function.

1. Is there a model $\bar{v}$ for which $b_1(\bar{v}) \wedge b_2(next(\bar{v})) \wedge \bar{v} \notin J$?
    - SAT → Add $\bar{v}$ to J and Q, goto 1.
    - UNSAT → Goto 2.
2. Q empty?
    - Yes → Done.
    - No → Goto 3.
3. For a model $\bar{q}$ on the queue Q, is there a model $\bar{v}$ for which $b_1(\bar{v}) \wedge next(\bar{v}) = \bar{q} \wedge \bar{v} \notin J$?
    - SAT → Add $\bar{v}$ to J and Q, goto 3.
    - UNSAT → Remove $\bar{q}$ from Q, goto 2.

After execution, J contains exactly all nodes for which jumping occurs. Since here a queue is used, this algorithm implements a breadth-first search. This can easily be adapted to a depth-first search by using a stack. Since the set of models is finite, the algorithm will always terminate. It may however require $|J|$ runs of the SMT-solver, so one may choose to set an upper bound on the size of J and abort ("give up") early.

Now that we know J, we can split condition b_1 into two: $b_1(\bar{v}) \wedge \bar{v} \in J$ and $b_1(\bar{v}) \wedge \bar{v} \notin J$. We can then apply the basic method to each of these disjunctive pieces. This algorithm only detects jumping from one piece into another. It should be applied iteratively over all the pieces, until no more jumping can occur. Note that this approach does not terminate until all condition jumping cases have been found. Since there are loops for which jumping occurs for every value of for instance an integer, it should "give up" after an upper-bound on the number of jumps is reached.

6.3 Heap-Space Usage Analysis

ResAna's heap consumption analysis is based on the COSTA [AAG$^+$08] tool, which provides a generic analysis infrastructure for Java byte code. The symbolic upper bound that COSTA generates for a method depends on the logical sizes of the method's arguments, structures pointed to by the object fields and the costs of the called (library) methods. The (logical) size of an integer is the maximum of the integer and 0, the size of an array is its length, the size of an object is its maximal reference chain. These assumptions constitute the size model in the COSTA terminology. For instance, let a method allocate n objects of class `X`, where integer n is a parameter of the method. Then COSTA generates a symbolic bound of the form $nat(n) * c(size(\mathtt{X}))$, where $nat(n)$ is integer n its logical size: $max\{n, 0\}$ and $c(size(\mathtt{X}))$ is the memory cost of creating an object of type `X`.

COSTA implements different garbage collection models [AGGZ10]. This functionality is retained in ResAna. Inside Java real-time threads no garbage collection is used, so in ResAna a user can select to ignore garbage collection. For normal Java code one can select to use the garbage collection feature of COSTA, which calculates an upper bound for all possible executions of a program. First, for every method, the amount of memory that can escape the method's scope is deduced. Using this information, peak consumption cost relationships are calculated and solved, which give upper bounds on the amount of memory used, even if using garbage collection.

We have added a number of improvements to the existing COSTA tool. Firstly, the recurrence solver was improved with interpolation-based height analysis. Secondly, we have changed the calculation of bounds for arrays, from an under-approximation to an over-approximation. Thirdly, the ability to calculate concrete bounds for a number of Java Virtual Machines, like OpenJDK and JamaicaVM, was added. And finally we added a post-processing step to simplify the expressions, so a programmer can easily interpret the information.

6.3.1 Interpolation-based height analysis for improving a recurrence solver

COSTA's approach to resource analysis is based on the classical method devised by Wegbreit [Weg75], which involves the generation of a recurrence relation capturing the costs of the program being analysed, and the consequtive computation of a closed form (non-recursive cost expression) which bounds the results of this recurrence relation. In COSTA terminology, a recurrence relation is called a *Cost Relation System* (CRS). The main feature that distinguishes CRSs from the classical concept of recurrence relations is non-determinism: a CRS defining the costs of a Java method may be defined by a set of equations guarded by non-disjoint conditions. As an example, consider the loop in Listing 6.6.

```java
while (x <= y) {
  new Object();
  if (...) x = x + 1; else y = y - 2;
}
```

Listing 6.6: Example loop.

We assume that the value of the if condition cannot be determined at compile time. Its memory costs are described by the following (simplified) CRS:

$$T(x, y) = 0 \qquad \{x > y\} \tag{6.8}$$

$$T(x, y) = c + T(x + 1, y) \qquad \{x \leq y\} \tag{6.9}$$

$$T(x, y) = c + T(x, y - 2) \qquad \{x \leq y\} \tag{6.10}$$

where c denotes the constant $c(\mathit{size}(\texttt{java.lang.Object}))$, i.e. the memory cost of creating an instance of `Object`. The COSTA system provides the recurrence solver PUBS [AAGP11], which computes the following closed-form:

$$\begin{gathered} \mathit{nat}(y - x + 1) * c(\mathit{size}(\texttt{java.lang.Object})) \\ + c(\mathit{size}(\texttt{java.lang.Object})) \end{gathered}$$

This is an upper-bound to the values of $T(x, y)$ given above.The resulting closed form corresponds to the worst-case execution of the loop (i.e. when the if condition always holds).

An important issue in the search of a closed-form of a CRS is to approximate the maximum number of unfoldings that must be undergone in order to reach a base case (height analysis). If we consider the CRS as a function being evaluated in a non-deterministic way, the number of unfoldings is closely related with the concept of ranking functions (see Chapter 5). For instance, in the CRS given above we get the following unfolding sequence of length $y - x + 1$:

$$\underbrace{T(x, y) \rightarrow T(x + 1, y) \rightarrow T(x + 2, y) \rightarrow \cdots \rightarrow T(y, y)}_{y-x+1 \text{ unfoldings}}$$

PUBS derives a ranking function for T by applying Podelski and Rybalchenko's method [PR04], which is complete for linear ranking functions. Unfortunately, it fails when the number of unfoldings does not depend linearly on the arguments of the CRS, as the following example shows:

$$R(x, y) = c \qquad \{x = 0, y = 0\} \tag{6.11}$$

$$R(x, y) = c + R(x - 1, x - 1) \qquad \{x > 0, y = 0\} \tag{6.12}$$

$$R(x, y) = c + R(x, y - 1) \qquad \{x \geq 0, y > 0\} \tag{6.13}$$

By equation (6.13) the variable y is decreased in every recursive call, until it reaches zero. Then, by equation (6.12) it is set to $x - 1$, from which it starts decreasing again. The worst-case evaluation of $R(x, y)$ yields a chain of length $\frac{1}{2}x^2 + \frac{1}{2}x + y + 1$, which does not depend linearly on (x, y).

We have extended the PUBS system so that it can infer polynomial ranking functions via testing and polynomial interpolation, as has been explained in Section 6.2. This extension was described in detail in [MSvEPn12]. It is described briefly here, with an additional contribution of verification of the interpolation results. The approach is, essentially, the same: choose a set of points (lying in a NCA) in the domain of the relation defined by the CRS, evaluate the CRS at these points, and find the interpolating polynomial. However, the evaluation of a CRS is more involved than the evaluation of a program instrumented with a counter, as it was done in Chapter 5. The main difficulty lies in non-determinism. Assume we want to evaluate $T(5, 9)$, where T is defined as in the CRS shown in (6.8-6.10). We can unfold the definition of $T(5, 9)$ by using (6.10) until we reach a base case, so we get the following sequence:

$$T(5, 9) \to T(5, 7) \to T(5, 5)$$

This sequence is of length three, which is not maximal, since we could have evaluated T by always using (6.9), so as to obtain a longer sequence:

$$T(5, 9) \to T(6, 9) \to T(7, 9) \to T(8, 9) \to T(9, 9)$$

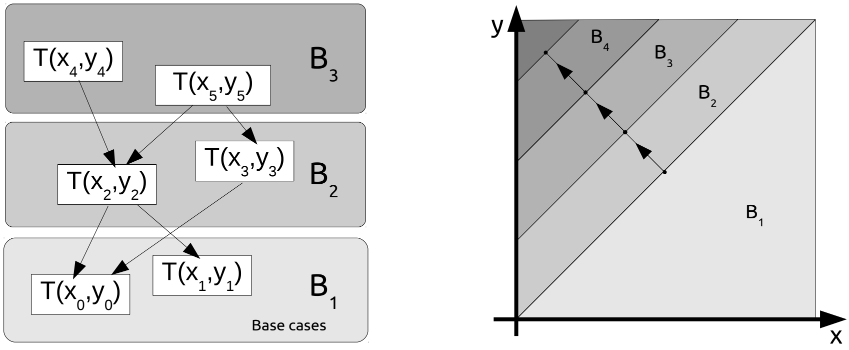


Figure 6.1: Meaning of the B_i sets and their representation as convex polyhedra.

As a consequence of this, we would have to examine all the possible evaluations of $T(5, 9)$ in order to obtain the longest unfolding sequence. However, the number of possible evaluations may be infinite even if the evaluation yields a finite number of results. We have addressed this problem by evaluating the CRS in a bottom-up way (Figure 6.1 left). We start from the set B_1 of pairs (x, y) such that the evaluation of $T(x, y)$ does not fall into a recursive case. The longest obtainable sequence in these cases is of length one. Now let us define the set B_2

of pairs (x, y) such that the evaluation of $T(x, y)$ falls into a recursive case, but the recursive call belongs to B_1. Thus we ensure that the evaluation of these pairs does not require more than two unfoldings. By following this procedure we obtain a sequence of sets $\{B_i\}$ each of which can be described as a disjoint union of convex polyhedra with the help of quantifier elimination techniques. We use a gradient-based approach for selecting the interpolation nodes from the B_i sets (Figure 6.1 right). The algorithm involves the search of *climbing paths* starting at the B_1 set, and minimizing the distance between B_i and B_{i+1} for each $i \in \mathbb{N}$. It is possible that, given a point (x, y) in a set B_i, there are several candidates in the next level B_{i+1} lying at the same distance from (x, y). In this case the climbing path forks, and the next interpolation nodes are searched from all these candidates. The process ends when the interpolating polynomial is uniquely determined.

Once we have found the interpolating polynomial on the set of test nodes, we have to check whether the resulting bound is correct. This can be done as follows: for each CRS the system can derive some predicates, whose satisfiability is a sufficient condition guaranteeing that the polynomial is an upper bound to the values of the CRS. These conditions involve inequalities between polynomial expressions, which are decidable in Tarski's theory of real closed fields. For instance, the system would generate the following logical statement for checking that $y - x + 1$ is an upper bound to $T(x, y)$:

$$\begin{aligned} \forall x, y, x', y' : {} & ((x \le y \wedge x' = x + 1 \wedge y' = y) \vee (x \le y \wedge x' = x \wedge y' = y - 2)) \\ & \Longrightarrow y - x + 1 \ge 1 + y' - x' + 1 \end{aligned}$$

If these generated predicates hold, then $y - x + 1$ is indeed an upper bound to $T(x, y)$. Our extension to PUBS delegates the task of checking such inequalities to the QEPCAD tool [Bro03]. For instance, in our running example $T(x, y)$ the following script is generated[13]:

```
[ Proving correctness of the bound corresponding to simpleLoop ]
(x,y,x’,y’)    -- Variables
0              -- Number of free variables in the formula
(A x) (A y) (A x’) (A y’)
  [[[x >= 0 /\ y >= 0 /\ x’ >= 0 /\ y’ >= 0] /\
  [[[(-1) x + 1 y’ >= (-2) /\ 1 x + (-1) x’ = 0 /\ 1 y + (-1) y’ = 2]  \/
    [(-1) x + 1 y’ >= 0 /\ 1 x + (-1) x’ = (-1) /\ 1 y + (-1) y’ = 0]]]]
     ==> [(-1) x + 1 y + 1 >= 1 + (-1) x’ + 1 y’ + 1]].
finish
```

For this script, QEPCAD yields the message `An equivalent quantifier-free formula: TRUE`, which validates the inferred bound.

6.3.2 Correct array-size analysis

Due to the way memory is handled, an array header will always be included with information about the array. As an array is a regular Java object the

[13] Variables have been renamed for better readability.

array header also includes the normal object header. Almost all architectures impose constraints on the memory allocator, e.g. memory allocators on the x86 architecture will allocate memory blocks in multiples 4 byte words. Although fewer bytes are requested, the memory allocator will add padding to an object that cannot be used for other purposes. This array header and padding need to be taken into account, otherwise the bound would be an under-approximation.

For instance, all JamaicaVM allocations are in (multiple) blocks of 32 bytes, considering the 32-bit version of JamaicaVM. If multiple blocks are needed they are stored in a tree structure with the array content stored in the leafs of the tree. The array header is 16 bytes long, so this leaves up to four pointers to the tree structures. In partial trees (in which the number of elements is not 4×8^n), nodes leading to unused array contents and unused array contents blocks are not stored, e.g. 16 pointers (four bytes each) stored will take only three blocks: two for the leafs and one intermediate block pointing to the leafs [Sie02]. An example array structure is shown in Figure 6.2. COSTA takes into account neither the array header, nor the structure needed to store the contents, nor padding. Only the space needed by the array contents (object references and primitive types) is included in the bound. This results in COSTA producing a bound for `new int[n]` equal to $n * size($`int`$)$, making it indistinguishable from the sequence `new int[1]; new int[n-1];`, so neglecting to account for the extra array header, padding and structure overhead. The (structure) overhead is dependent on the virtual machine used. To deal with these deficiencies we implemented a special mode in COSTA when generating a concrete bound for arrays in JamaicaVM, as explained next.

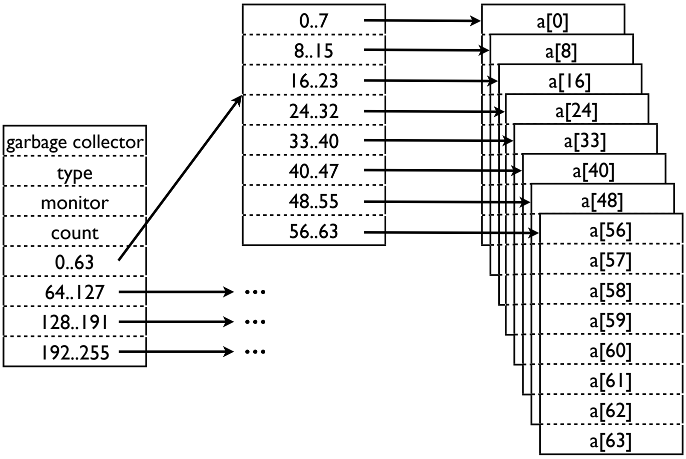


Figure 6.2: Graphical representation of a JamaicaVM array of size n, with $33 \leq n \leq 255$, with $a[i]$ representing the contents of the array. Allocating an array of 63 elements takes 10 blocks.

6.3.3 Virtual-machine specialisation by adding type-size information

COSTA has no knowledge of specific Java Virtual Machines like JamaicaVM. Our approach is to replace in all the symbolic bounds generated by COSTA the symbolic object sizes by the exact sizes of objects in bytes. The exact sizes are retrieved from the target VM by means of a specially generated program. For JamaicaVM, this generated program depends on the Scoped Memory extensions of Realtime Java. For other Java VMs we use the reflection API in Java, which is more general and can be run on any VM which supports the reflection API. We validated this method for OpenJDK, by interfacing directly with the virtual machine by means of a (JNI) plugin.

For generating bounds for arrays allocated in an instance of JamaicaVM, we adjusted COSTA to include an over-approximation. A simple way of calculating the size of arrays, by means of the small recursive function defined in Equation 6.14, could not be implemented in COSTA, because of the manner COSTA represents and calculates the bounds internally. This recursive function is valid for data types of four bytes[14], which correspond to the size of pointers used in the tree structure pointing to the leafs, resulting in a cleaner formula.

$$\text{arrayblocks}(n) = \begin{cases} n & \text{if } n \leq 8 \\ \lceil \frac{n}{8} \rceil + \text{arrayblocks}(\lceil \frac{n}{8} \rceil) & \text{otherwise} \end{cases} \tag{6.14}$$

By transforming the formula to an over-approximation (by replacing $\lceil \frac{n}{8} \rceil$ with $\frac{n+7}{8}$), we were able to solve this new recurrence equation. The results in a new formula, and after integrating adjustments for the start cases, is listed in Equation 6.15. We have implemented this solution in our version of COSTA, which is included in ResAna, so that analysing arrays now gives a correct over-approximation.

$$\text{arrayblocks}(n) \leq \frac{n+5}{7} + (\log_8 n + 7) \tag{6.15}$$

We have a similar formula for arrays in OpenJDK, which uses continuous allocation with a small header by default (an array of n elements uses $4n+8$ bytes, on a 32 bits target architecture). For each new Java VM a new specialisation for arrays needs to be added in order to correctly generate bounds for code using arrays.

6.3.4 Simplification of bounds

COSTA internally calculates the symbolic bounds without considering the format of the expression. The produced expressions are not necessary user friendly, for instance:

[14] These results are valid for data-types with a representation of four bytes. Alternate data-types (e.g. `byte`, `char`, `short`, `double`), can be calculated by multiplying the input n by a factor of $\frac{1}{4}$, $\frac{1}{2}$, $\frac{1}{2}$, 2 respectively.

$$\begin{aligned}
&nat(n)*\\
&\quad(nat(n)*(c(size(\texttt{java.lang.Object},1))+\\
&\qquad\qquad c(size(\texttt{java.lang.Object},2)))+\\
&\quad nat(n)*(c(size(\texttt{java.lang.Object},1))+\\
&\qquad\qquad c(size(\texttt{java.lang.Object},2)))+\\
&\quad nat(n)*(c(size(\texttt{java.lang.Object},1))+\\
&\qquad\qquad c(size(\texttt{java.lang.Object},2))))
\end{aligned}$$

We implemented a recursive descent parser with reductions of mathematical expressions in order to make the expressions generated by COSTA more user readable. The result of an expression is not altered[15], but the formula is reordered and reduced to a more user friendly expression. The expression above is transformed into:

$$6n^2 * size(\texttt{java.lang.Object})$$

One can now easily see that the bound is quadratic. This simplification is built into ResAna and applied to all user-visible expressions.

6.3.5 Example

The complexity of calculating Fibonacci numbers is well known. The run-time complexity (in terms of methods calls) of calculating the nth Fibonacci value using a double recursion is related to the golden ratio $\varphi = \frac{1+\sqrt{5}}{2}$, which results in a complexity of $O(\varphi^n)$ method calls. Standard textbooks on complexity analysis use over-approximation, which results in a complexity of $O(2^n)$ for the same function. By adding an object allocation to each iteration, the heap consumption should be the same as the run-time complexity. The resulting code is given in Listing 6.7. Our tool annotates this function with the bound $(2^n - 1) * size(\texttt{java.lang.Object})$, matching the expected bound.

```
int fib(int n) {
    new Object();
    if (n < 2) return n;
    return fib(n-1) + fib(n-2);
}
```

Listing 6.7: Adaptation of the double recursive Fibonacci function, allocating an object in each call.

The nth Fibonacci number can also be calculated by using a single recursion, for which the complexity should be $O(n)$. The resulting code, with added object allocations, is listed in Listing 6.8. This single recursive function is annotated by

[15] Technically the output is altered a little bit as the allocation order, which only matters internally, is neglected. The allocation order is included in the `size` construct as the second argument. The *nat* function is also omitted for brevity, and should always be applied to variables.

our tool with a bound of $(n+1) * \mathit{size}(\texttt{java.lang.Object})$, also matching the expected complexity bound.

```java
int fib_helper(int a, int b, int n) {
    new Object();
    if (n <= 0) return a;
    return fib_helper(b, a+b, n-1);
}
int fib(int n) {
    return fib_helper(0, 1, n);
}
```

Listing 6.8: Adaptation of the single recursive Fibonacci function, allocating an object in each call.

6.4 Stack-Size Analysis

The proposed method of stack analysis requires global knowledge of the program, including its data. A *data-flow-based static analyser* VeriFlux is used to provide this knowledge [HTS08] (see `http://www.aicas.com/veriflux.html`).

Analysis of *recursive methods* is a challenge in static evaluation of stack consumption. To deal with it, VeriFlux's stack-size analysis relies on recursion-depth annotations. A recursion-depth annotation consists of an expression that evaluates to a natural number that is an upper bound on the number of nested recursive calls. Syntactically, recursion-depth annotations are provided as JML `measured_by` clauses. A `measured_by` expression is a usual symbolic expression like *a.length - 1*. VeriFlux outputs the stack bound in bytes, which is the number computed from the annotations and the input data of the main method. If VeriFlux discovers recursive methods that do not carry a recursion depth annotation, it uses a default recursion depth, which is a positive natural number or infinity. This number can be configured in the tool's GUI. In case the default recursion depth is configured to be infinity, the stack size analysis will report an infinite stack size for all threads that call recursive methods that do not carry a recursion-depth annotation.

Expressions for `measured_by` annotations are obtained using COSTA, which computes both:

- A symbolic upper bound on the depth of recursion (i.e. a "ranking function" for recursive calls) for a given method
- A symbolic upper bound on the number of calls of the method from itself.

The former corresponds to the height of the call tree, the latter represents the number of the nodes in the call tree. For instance, the depth of recursion for a typical implementation of the n-th Fibonacci number calculation belongs to $O(n)$, whereas the number of call belongs to $O(2^n)$. Both a ranking function and a bound on the number of recursive calls, can be used as `measured_by` expressions. The former and the latter coincide if the recursion branching factor $b < 2$. The number of calls leads to exponential over-approximation when $b \geq 2$.

Initially, COSTA did not output ranking functions, even though they were a part of the tool its internal computations. The tool has been adjusted within the CHARTER project by adding an option that allows ranking functions to be shown.

Consider the method `fib`, computing the n-th Fibonacci number, in Listing 6.7. As expected, COSTA produces the ranking function $nat(n-1)$. This represents the depth of the recursion tree. It is transformed by ResAna into the annotation `measured_by n-1`. The upper bound on the number of recursive calls that COSTA generates is $2 * (2^{nat(n-1)} - 1)$. This corresponds to the total number of nodes in the recursion tree.

A Java VM has two stacks: a Java stack and a native one. Interpreted code and dynamically generated code execute on the Java stack. External C libraries, JIT compiled (Java) code and Java functionality implemented natively execute on the native stack. Both have different stack usage characteristics. We consider Java stack usage while running the virtual machine in interpreted mode. While methods utilizing the native stack cannot be analysed automatically, the user can specify bounds in their JML contracts.

Java applications typically call methods from libraries. To obtain good stack-consumption bounds for such applications, one should provide stack-consumption bounds for library methods. In principle, library methods are analysed by CHARTER methodology in the same manner as applications, i.e. as the example above. However, analysis of libraries requires additional technical overhead, because of two issues: libraries are large and library methods may call native routines.

6.4.1 Adjustments for analysis of libraries

Since a call to a library-method typically amounts to long chains of calls to other methods, the corresponding call graph becomes very large. The COSTA analysis is based on call graphs, so obtaining resource bounds in this case becomes unfeasible. Computations take too much time and/or at the end one obtains a huge unreadable symbolic expression. Therefore, when performing the stack analysis on programs with library calls, it is best to begin with analysis of the methods belonging to one strongly-connected component of the call graph[16]. From our experience, COSTA performs it in reasonable time. After that, methods that call the already analysed ones can be analysed. The annotations of the already analysed methods can now be used as *contracts*. Eventually, all the library is analysed in a bottom-up manner.

Technically, *native stacks* are needed to cope with methods that are compiled to native machine code (for optimization purposes) and with native methods that are called through the Java Native Interface JNI (in order to access

[16] Recall that a strongly connected component of a directed graph is a sub-graph in which for any two nodes a and b, there is a path from a to b and vice versa.

services provided by platform-specific native libraries). VeriFlux does not address StackOverflowErrors due to overflows of native stacks. Since verification of C native methods is beyond of scope of this work, one has to rely on the known information about the behavior of these methods, i.e. corresponding contracts.

```
String toString(int i) {
   if (i==Integer.MIN_VALUE) return "-2147483648";
   int size = (i < 0) ? stringSize(-i) + 1 : stringSize(i);
   char[] buf = new char[size];
   getChars(i, size, buf);
   return MyString.valueOf(buf, 0, size);
}
```

Listing 6.9: The `toString` method from the `Integer` class in the Java standard library.

As an example for both issues, consider the `toString` method, which belongs to the `Integer` class and maps an integer number to a string, shown in Listing 6.9. Before running COSTA, place this method in the abstracted class `MyInteger`, that contains only `toString` and the methods called from it. Create the abstracted versions of the classes `StringIndexOutOfBoundsException` and `String`, that contain the methods called from `toString`, and the ones called from them, et cetera. COSTA produces a ranking function that symbolically depends on the costs of two native methods: `copyChars` and `cast2string`. If their contracts say that they do not call Java methods (which is, indeed, the case for this example), their costs are turned into zeros by ResAna and the final `measured_by` expression is 0. This result can be approved by an accurate data-flow analysis of the method `toString` using pen and paper.

6.4.2 Stack-size analysis by VeriFlux

In this section we consider the principles on which the stack analysis of VeriFlux is based. VeriFlux computes an invocation graph, in which nodes correspond to methods and edges represent method invocations. Recursive method calls correspond to cycles in the graph. In order to eliminate cycles, one first computes the strongly connected components (SCCs) of the invocation graph. Each SCC with more than zero nodes is then replaced by a single node that is annotated by *the sum of the sizes of all stack frames that correspond to nodes (i.e., method invocations) in that SCC, multiplied by the maximal recursion depth over all the nodes* (i.e., method invocations) in that SCC. The recursion depths are computed by evaluating the `measured_by` annotations of invoked methods or using the default recursion depth for methods that do not carry these annotations. All nodes that are not in an SCC with more than zero nodes are simply annotated by the size of the stack frame of the corresponding method invocation.

After merging each SCC, one is left with a directed acyclic graph (DAG), where each node is annotated with a positive integer. Let this annotation be called the stack-frame size of the node. To obtain the final result, VeriFlux adds the stack frame size of the node to the maximum of the (recursively computed) stack sizes of its successor nodes. This can be achieved, for all nodes, in a depth-first traversal of the DAG.

From the user perspective, VeriFlux performs stack analysis in the following way. The tool starts from the main method and evaluates the `measured_by` annotations of all called methods in an abstract environment. Variables (and expressions) in this environment are evaluated to intervals that represent all possible values they may have according to data-flow analysis. For instance a variable n is replaced with the interval $[0, 21]$ if data-flow analysis shows that `fib`(n) will be called on n from 0 to 21.

The value that VeriFlux outputs is an upper bound on the used stack in bytes, computed from the symbolic `measured_by` expressions and the input data of the main method. Note that VeriFlux's computation of the abstract environment is approximate. In the worst case, VeriFlux may have computed the abstract value ’Any’ for some of the variables that occur in the `measured_by` expression. Then the concrete value of the `measured_by` expression evaluates to ’Any’ as well. If a symbolic `measured_by` expression is not given, then a concrete default bound is involved, given by the user. The correctness of this given numerical upper bound is not checked, VeriFlux simply uses this value in the analysis. The upper bounds computed by VeriFlux are not tight, i.e., they may be higher than necessary.

Now, proceed with the Fibonacci example. Let it be called from the main method in Listing 6.10.

```java
public static void main(String [] args) {
  fib(21);
}
```

Listing 6.10: Main method calling the `fib` method.

VeriFlux computes the depth of recursion, which, as expected, is equal to 20. The upper bound on consumed stack space computed by VeriFlux is 1156 bytes. This consists of 20 stack frames for the `fib` method, which use 56 bytes each, plus 36 bytes of stack space needed to call the method. Calling the same method with $n = 22$ results in a bound of 1212 bytes. This means that a stack overflow will not occur if 1156 and 1212 bytes of stack space are reserved for the main thread in the first and in the second case respectively.

To deal with virtual method invocations, VeriFlux has an option ”resolve opaque calls”. When switched on, it considers all possible implementations or subclasses of a given interface or a superclass. If the analysis cannot resolve which virtual method is actually called, the maximum over the stack sizes of all those methods that are possibly called is used. Conceptually, the invocation graph will then have edges from the caller to all possibly called methods.

6.5 User Experience

We have combined all the CHARTER verification tools in a VirtualBox image for easy installation. This image, the Eclipse plug-in and the source code, can be downloaded from the ResAna website[17].

The Dutch National Aerospace Laboratory NLR has used the VirtualBox image in the development of a safety-critical avionics application. Their experience is described in [WW12]. They have selected the Environment Control System (ECS) on board an aircraft for evaluating the CHARTER tool-chain. The ECS is responsible for air conditioning and air pressurization. The application is written in Realtime Java and runs on JamaicaVM.

Before using the CHARTER tools, NLR did not determine any ranking functions for loops or memory-usage bounds, because manually devising them would require a very large effort. Now, thanks to ResAna, these bounds can be inferred relatively quickly, so the programmers now have a better understanding of the workings and hardware-requirements of their software. They applied ResAna for loop bound and heap space analysis. The tool was found to be easy to use. Their industrial user feedback has led to several (small but important) improvements of the ResAna tool. NLR has used the complete CHARTER tool-chain in their evaluation. The use of the tool set resulted in a 21% decrease of the required software engineering effort.

We ourselves have also conducted several case studies with respect to the loop bound analysis. These are discussed in Section 5.4. Furthermore, during a course on software analysis, for several consecutive years, we have asked Master students to perform a series of exercises using ResAna. Students successfully used the tool to infer ranking functions, heap bounds and stack bounds for various examples. Also, they performed a small case study on the code of Pygmy (a small web server). Again, ranking functions could be generated for roughly two thirds of the loops. Similar exercises were also given to PhD students at the 2013 IPA[18] school on Software Engineering and Technology, who found the tool to be very useful.

6.6 Related Work

The polynomial interpolation based technique was successfully applied in the analysis of output-on-input data-structure size relations for functions in a functional language in [vKSvE08], [SvKvE07], [SvET11], [SvEvK09], [TSv09] and [GSvE13]. This method can, for instance, be used to determine that if the `append` function gets two lists of lengths n and m as input, it will return a list of length $n + m$.

[17] `http://resourceanalysis.cs.ru.nl/resana/`

[18] Institute for Programming research and Algorithmics: http://www.win.tue.nl/ipa/

6.6.1 Loop-Bound Analysis

Hunt et al. discuss the expression of manually conceived ranking functions in JML, their verification using KeY and the combination with data-flow analysis in [HSST06]. What is “missing” in the method is the automated inference of ranking functions, which ResAna supplies.

In [ABG$^+$11], an approach that is similar to ours is taken, in the combination of COSTA with the KeY tool. The results that COSTA gives are output as JML annotations, that may then be verified using KeY.

Various other research results on bounding the number of loop iterations are described in the literature. However, most approaches generate concrete (numerical) bounds [ESG$^+$07, LCFM09, DMBCS08], as opposed to *symbolic* bounds. The methods that are able to infer symbolic loop bounds are limited to either bounds that depend linearly on program variables (the procedure used in ResAna infers polynomial bounds) [PR04] or that are constructed from monotonic subformulae [Gul09, GZ10].

Several syntactical methods are discussed [FJ10, GJK09], which will be more efficient for simple cases, but less general. Our procedure can be seen as complementary to those methods. In case a syntactical method is not applicable to a certain loop, our more general method can be used.

To generate algebraic loop invariants, Sharma et al. [SGH$^+$13] use a procedure which, as our loop-bound inference algorithm, employs interpolation and separated inference and verification phases. They refer to their algorithm as *guess-and-check*, as it employs a non-sound inference phase and a verification phase. In the inference phase, the program is executed on data from unit tests and results are interpolated. For checking the invariants they use an SMT solver. The main difference to our work is that they search for so-called algebraic *invariants*, which are defined as algebraic *equalities* over program variables, whereas we search for a specific *variant* (a ranking function) specifying the number of remaining iterations of the loop, the value of which is required to decrease on each iteration. This ranking function implies an algebraic *inequality* as invariant.

6.6.2 Time Performance Analysis

There are a number of parallels of our work with time performance analysis. This can be average execution time analysis or, more common, Worst Case Execution Time (*WCET*) analysis. As already mentioned, loop-bound inference can be used for time analysis, in particular for WCET analysis. Depending on the cost function associated with each iteration of the loop, one can compute a memory bound or a timing bound. To properly use this for WCET analysis one has to incorporate extra analysis of e.g. cache behaviour, context switches, et cetera, to precisely approximate the WCET as is done in [WEE$^+$08, RGBW07].

As memory allocators and cache policies are rather slow and unpredictable, the number and the amount of memory allocated have an impact on performance [RGBW07]. One has to resort to special means to alleviate these problems [HBHR11]. Our heap analysis can also be used to gain insight into the allocations of a program. This can help reduce the number and amount of allocations in a program, which can lead to smaller worst case execution times.

6.6.3 Heap-Space Usage Analysis

We have taken the COSTA system [AAG$^+$08] as our point of reference. The authors have recently improved [AGM11] the precision of PUBS, its recurrence solver, by considering upper and lower bounds to the cost of each loop iteration. In a different direction, COSTA has improved its memory analysis in order to take different models of garbage collection into account [AGGZ10]. However, the authors claim that this extension does not require any changes to the recurrence solver PUBS. Thus, the techniques presented in Section 6.3.1 should fit with these extensions.

In the field of functional languages, a seminal paper on static inference of memory bounds is [HJ03]. A special type inference algorithm generates a set of linear constraints which, if satisfiable, specify a safe linear bound on the heap consumption. One of the authors extended this type system in [HH10, HAH11] in order to infer multivariate polynomial bounds. Surprisingly, the constraints resulting from the new type system are still linear.

6.6.4 Stack-Size Analysis

In practice, stack usage in Java is often measured by instrumenting or transforming the source code so that it counts consumed resources (and computes other relevant information) on the inputs of the original code. To our knowledge, there are two commercial tools that perform Java stack analysis: *Coverity Static Analyzer* and *Klockwork*, with its *kwstackoverflow*. Another tool, *GNAT-Stack*, analyses object-oriented applications, automatically determining maximum stack usage on code that uses dynamic dispatching in Ada and C++.

In [WQQC10], a static stack-bound analysis for abstract Java bytecode is described. The described method considers Java bytecode with recovered high-level control structures (conditionals and while-loops). The inference process is divided into three key stages: frame-bound inference, abstract-state inference and stack-bound inference. Recall that a frame is a piece of stack reserved for each method invocation. Each stage applies a corresponding set of inference rules. In these rules the authors use *Presburger (linear) arithmetic formulae* to describe states of programs. It is stated that an implementation is under development.

6.7 Conclusions and Future Work

To assist in making resource analysis practical, we have introduced new techniques and combined these techniques in our new tool, ResAna. Complex loop, heap and stack bounds can be inferred in an integrated way within the Eclipse IDE. Bounds can be inferred that are specific for the underlying virtual machine (shown both for JamaicaVM and OpenJDK).

Obviously, a full resource analysis tool would also need to build in an elaborate time analysis. For now, we will rely on other tools to provide such information. The ability to infer resource bounds contributes to improving the development process of producing real-time safety-critical systems both with respect to ease of development and with respect to improved reliability. The Dutch National Aerospace Laboratory (NLR) has successfully used ResAna in the development of a demonstrator safety-critical Realtime Java avionics application.

Future Work. A more thorough evaluation of ResAna would be very valuable. A practical case study could point out weak points of the different analyses and suggest directions for improvement. Furthermore, the capabilities of timing analysis tools could be incorporated in our tool or it could be made easy to exchange information between our tool and timing analysis tools. Another direction of future research could be to include work on other kinds of resources that are consumed, e.g. also inferring and proving energy related properties of Java programs might be important. Furthermore, one could define, instead of a single overall memory bound for the complete run-time of a program, a time-dependent memory bound which gives a bound for the consumption on a certain moment in the execution of a program. Such a time-dependent bound is called a *live* memory bound. Together with information on synchronisation moments, this opens up the possibility to derive more precise memory bounds by adding upper bounds of processes in the periods between synchronisation moments.

Acknowledgements

We would like to thank James Hunt, Isabel Tonin and Christian Haack of Aicas for their help and insights in developing the stack-size analysis. Also, we would like to thank Gosse Wedzinga and Klaas Wiegmink at NLR for their useful feedback on ResAna use in practice. We thank Sebastiaan Joosten for his valuable advice on elements of this chapter. We would like to thank Samir Genaim for his technical support regarding COSTA and we thank the anonymous reviewers for their important suggestions.

Chapter 7

Conclusions

This thesis contributes a series of software analysis methods. The methods aid in establishing program properties that are of particular interest for resource-sensitive systems, such as wireless sensor nodes. Desirable properties are security, functional correctness and efficient use of resources.

Security. It is shown how model-checking can be applied to reveal security vulnerabilities. A condition on implementation parameters of the Tamper-Evident Pairing protocol is discovered that excludes a discovered vulnerability, enabling secure implementation.

Functional correctness. A method is presented to improve the coverage of test-cases that are generated using symbolic execution. Automatic test-case generation greatly reduces the manual effort required to adequately test (critical) software. The presented method increases the proportion of program behaviours that are activated by a generated test-set.

Efficient use of resources. First, an energy analysis method is presented, that is based on Hoare logic. It is sound and implemented in the tool ECALOGIC. It represents a step towards energy-consumption analysis of software that is attainable to average programmers. In this scenario, a complex and expensive measurement set-up is only required to build hardware models, not to analyse the software itself. Second, a method to infer polynomial ranking functions for loops is presented. This method advances the state of the art by inferring non-monotonic polynomial loop bounds. Finally, a heap space analysis and a stack space analysis are presented, that enable bounding memory consumption of JAVA programs. The presented heap space analysis, stack space analysis and loop bound analysis are implemented in the tool RESANA. This represents a step towards practical applicability of resource analysis. The Dutch National Aerospace Laboratory (NLR) has successfully applied RESANA in the development of a demonstrator safety-critical REALTIME JAVA avionics application.

It can be critical to establish such properties of software, but it is a very complex task in general. There is no “silver bullet” of software analysis, we must pick the right tool for the right application. For most properties, automatic inference is undecidable in general. It is therefore the aim to maximise the set of analysable programs, inputs and properties.

Various aspects of this thesis will be combined in my new endeavour at Carnegie-Mellon University. There, I will research the application of memory and time analysis to detect security vulnerabilities within the Integrated Symbolic Execution for Space-Time Analysis of Code (ISSTAC) project. Symbolic execution will be combined with measures and cost models to compute worst-case behaviour with respect to time and memory consumption. By solving the constraint (path condition) for a program to exhibit a certain worst-case behaviour, vulnerabilities to denial-of-service attacks can be detected. By comparing time and memory usage of different program paths, vulnerabilities to side-channel attacks can be found. In this way, ISSTAC applies symbolic execution in the context of resource analysis, in order to increase security. Within this project, I expect to develop new applications and improvements for the analysis methods presented in this thesis, as well as many novel methods, further advancing the state of the art of practical software analysis.

Bibliography

[AAG$^+$08] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and implementation of a cost and termination analyzer for Java bytecode. In F. de Boer, M. Bonsangue, S. Graf, and W. de Roever, editors, *Proceedings of the 6th International Symposium on Formal Methods for Components and Objects (FMCO'07)*, volume 5382 of *Lecture Notes in Computer Science*, pages 113–132. Springer, 2008. Cited on pages 4, 40, 80, 87 and 100.

[AAGP08] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *Proceedings of 15th International Static Analysis Symposium (SAS'08)*, pages 221–237, 2008. Cited on page 76.

[AAGP11] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, February 2011. Cited on pages 78 and 88.

[ABB$^+$05] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Drielsma, P. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In K. Etessami and S. Rajamani, editors, *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer, 2005. Cited on page 14.

[ABG$^+$11] E. Albert, R. Bubel, S. Genaim, R. Hähnle, G. Puebla, and G. Román-Díez. Verified resource guarantees using COSTA and KeY. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '11, pages 73–76. ACM, 2011. Cited on page 99.

[ABH$^+$07] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resources. *Theor. Comput. Sci.*, 389(3):411–445, December 2007. Cited on page 40.

[ACC09] A. Armando, R. Carbone, and L. Compagna. LTL model checking for security protocols. *Journal of Applied Non-Classical Logics*, 19(4):403–429, 2009. Cited on page 25.

[AGGZ10] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric inference of memory requirements for garbage collected languages. In J. Vitek and D. Lea, editors, *ISMM'10*, pages 121–130. ACM, 2010. Cited on pages 87 and 100.

[AGM11] E. Albert, S. Genaim, and A. N. Masud. More precise yet widely applicable cost analysis. In R. Jhala and D. A. Schmidt, editors, *VMCAI'11*, volume 6538 of *Lecture Notes in Computer Science*, pages 38–53. Springer, 2011. Cited on page 100.

[Alb10] S. Albers. Energy-efficient algorithms. *Commun. ACM*, 53(5):86–96, 2010. Cited on page 40.

[Ama05] R. M. Amadio. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae*, 65(1-2):29–60, August 2005. Cited on page 78.

[Atk10] R. Atkey. Amortised resource analysis with separation logic. In *Proceedings of the 19th European Conference on Programming Languages and Systems (ESOP'10)*, volume 6012 of *Lecture Notes in Computer Science*, pages 85–103, 2010. Cited on page 40.

[BA09] A. M. Ben-Amram. Size-change termination, monotonicity constraints and ranking functions. In *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2009. Cited on page 76.

[BC04] Y. Bertot and P. Castéran. *Interactive theorem proving and program development. Coq'Art: the calculus of inductive constructions.* Springer, 2004. Cited on page 7.

[BCD$^+$11] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. Cited on page 5.

[BD98] D. Bošnacki and D. Dams. Integrating real time into Spin: a prototype implementation. In D. Bošnacki, editor, *Enhancing State Space Reduction Techniques for Model Checking.* Springer, 1998. Cited on page 24.

[BEL75] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT—A formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–245, June 1975. Cited on page 5.

[BHS07] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Artificial Intelligence.* Springer, 2007. Cited on pages 7, 63, 73 and 79.

[BK08] C. Baier and J. P. Katoen. *Principles of Model Checking.* MIT Press, 2008. Cited on page 6.

[Bla01] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society. Cited on page 14.

[BLL$^+$96] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proceedings of the Third Workshop on Verification and Control of Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer–Verlag, 1996. Cited on pages 6, 14 and 24.

[BPRT13] J. Backes, S. Person, N. Rungta, and O. Tkachuk. Regression verification using impact summaries. In E. Bartocci and C. Ramakrishnan, editors, *Model Checking Software*, volume 7976 of *Lecture Notes in Computer Science*, pages 99–116. Springer, 2013. Cited on page 37.

[Bro03] C. W. Brown. QEPCAD B: a program for computing with semi-algebraic sets using CADs. *SIGSAM Bull.*, 37(4):97–108, December 2003. Cited on pages 79 and 90.

[BST10] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, 2010. Cited on page 85.

[BTM00] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. *SIGARCH Comput. Archit. News*, 28(2):83–94, May 2000. Cited on page 40.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252. ACM, 1977. Cited on page 4.

[CE82] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1982. Cited on page 6.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986. Cited on page 6.

[CES09] E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: Algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, November 2009. Cited on page 6.

[CGK$^+$11] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1066–1071. ACM, 2011. Cited on pages 5 and 29.

[CGP99] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking.* MIT press, 1999. Cited on page 6.

[CKLP06] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363. Springer, 2006. Cited on page 62.

[CL87] C. K. Chui and M.-J. Lai. *Vandermonde determinants and Lagrange interpolation in R^s*, volume 107 of *Lecture Notes in Pure and Applied Mathematics*, pages 23–35. CRC Press, 1987. Cited on pages 63 and 64.

[Cla76] L. A. Clarke. A program testing system. In *Proceedings of the 1976 Annual Conference*, pages 488–491. ACM, 1976. Cited on page 5.

[Cok11] D. R. Cok. Openjml: Jml for java 7 by extending openjdk. In M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 472–479. Springer, 2011. Cited on page 73.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158. ACM, 1971. Cited on page 5.

[CS13] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013. Cited on page 5.

[CZSL12] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy types. *SIGPLAN Not.*, 47(10):831–850, October 2012. Cited on page 40.

[CZvD$^+$09] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on*, 35(5):684–702, Sept 2009. Cited on page 3.

[DdM06] B. Dutertre and L. de Moura. The Yices SMT solver. Tool presentation paper at `http://yices.csl.sri.com/tool-paper.pdf`, August 2006. Cited on page 5.

[dDMP10] J. de Dios, M. Montenegro, and R. Peña. Certified absence of dangling pointers in a language with explicit deallocation. In *8th International Conference on Integrated Formal Methods, IFM 2010*, LNCS 6396, pages 305–319. Springer, 2010. Cited on page 78.

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644 – 654, nov 1976. Cited on pages 14 and 16.

[DL00] A. Dupuy and N. Leveson. An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In *Proceedings of the 19th Digital Avionics Systems Conference*, 2000. Cited on page 3.

[dMB08] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. Cited on page 5.

[DMBCS08] M. De Michiel, A. Bonenfant, H. Cassé, and P. Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *RTCSA '08: Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 161–166, Washington, DC, USA, 2008. IEEE Computer Society. Cited on pages 75 and 99.

[dRH12] M. J. de Mol, A. Rensink, and J. J. Hunt. Graph transforming Java data. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE 2012), Talinn, Estonia*, volume 7212 of *Lecture Notes in Computer Science*, pages 209–223, London, March 2012. Springer. Cited on page 78.

[Dri12] M. Drijvers. Model checking Tamper-Evident Pairing. Bachelor thesis, Radboud University Nijmegen, 2012. Cited on page 24.

[ESG$^+$07] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In C. Rochange, editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. Cited on pages 63, 75 and 99.

[FJ10] J. Fulara and K. Jakubczyk. Practically applicable formal methods. In *SOFSEM '10: Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science*, pages 407–418. Springer, 2010. Cited on pages 75 and 99.

[Flo67] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967. Cited on page 5.

[GAZK11] S. Gollakota, N. Ahmed, N. Zeldovich, and D. Katabi. Secure in-band wireless pairing. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, Berkeley, CA, USA, 2011. USENIX Association. Cited on pages 8, 10, 14, 15, 18 and 19.

[GCB05] S. V. Gheorghita, H. Corporaal, and T. Basten. Iterative compilation for energy reduction. *Journal of Embedded Computing*, 1(4):509–520, 2005. Cited on page 40.

[GHH$^+$14] A. Groce, K. Havelund, G. Holzmann, R. Joshi, and R.-G. Xu. Establishing flight software reliability: testing, model checking, constraint-solving, monitoring and learning. *Annals of Mathematics and Artificial Intelligence*, 70(4):315–349, 2014. Cited on page 6.

[GJK09] S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI'09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 375–385. ACM, 2009. Cited on pages 76 and 99.

[GL11] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 23–33. ACM, 2011. Cited on pages 30 and 36.

[Gla08] C. Gladisch. Verification-based test case generation for full feasible branch coverage. In *Sixth IEEE International Conference on Software Engineering and Formal Methods, 2008. SEFM '08.*, pages 159–168, 2008. Cited on page 29.

[GSvE13] A. Gobi, O. Shkaravska, and M. van Eekelen. Higher-order size checking without subtyping. In H.-W. Loidl and K. Hammond, editors, *Proceedings of the 13th International Symposium on Trends in functional Programming (TFP2012)*, volume 7829 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2013. Cited on page 98.

[Gul09] S. Gulwani. SPEED: Symbolic complexity bound analysis. In *CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification*, pages 51–62. Springer, 2009. Cited on pages 62, 76 and 99.

[GZ10] S. Gulwani and F. Zuleger. The reachability-bound problem. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI'10, pages 292–304. ACM, 2010. Cited on pages 78 and 99.

[HAH11] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. In T. Ball and M. Sagiv, editors, *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'11, pages 357–370. ACM, 2011. Cited on pages 40, 78 and 100.

[HBHR11] J. Herter, P. Backes, F. Haupenthal, and J. Reineke. CAMA: A predictable cache-aware memory allocator. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS'11).* IEEE Computer Society, July 2011. Cited on page 99.

[HH10] J. Hoffmann and M. Hofmann. Amortized resource analysis with polynomial potential. A static inference of polynomial bounds for functional programs. In *ESOP'10*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2010. Cited on page 100.

[HJ00] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental*

Approaches to Software Engineering, volume 1783 of *Lecture Notes in Computer Science*, pages 284–303. Springer, 2000. Cited on page 5.

[HJ03] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'03, pages 185–197. ACM Press, 2003. Cited on page 100.

[HJ04] G. Holzmann and R. Joshi. Model-driven software verification. In S. Graf and L. Mounier, editors, *Model Checking Software*, volume 2989 of *Lecture Notes in Computer Science*, pages 76–91. Springer, 2004. Cited on page 6.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969. Cited on page 5.

[Hol97] G. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279 –295, May 1997. Cited on pages 14 and 18.

[HP00] K. Havelund and T. Pressburger. Model checking Java programs using Java Pathfinder. *Int. Journal on Softw. Tools for Tech. Transfer*, 2(4):366–381, 2000. Cited on page 28.

[HSST06] J. J. Hunt, F. B. Siebert, P. H. Schmitt, and I. Tonin. Provably correct loops bounds for realtime Java programs. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 162–169. ACM, 2006. Cited on pages 74, 76 and 99.

[HTS08] J. J. Hunt, I. Tonin, and F. B. Siebert. Using global data flow analysis on bytecode to aid worst case execution time analysis for real-time Java programs. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 97–105. ACM, 2008. Cited on pages 41, 80 and 94.

[HVCR01] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson. *A practical tutorial on modified condition/decision coverage*. NASA Langley Research Center, Hampton, VA, USA, 2001. NASA Technical Memorandum TM-2001-210876. Cited on page 3.

[JML06] R. Jayaseelan, T. Mitra, and X. Li. Estimating the worst-case energy consumption of embedded software. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 81–90. IEEE, 2006. Cited on page 40.

[JNM$^+$06] M. N. O. Junior, S. Neto, P. R. M. Maciel, R. M. F. Lima, A. Ribeiro, R. S. Barreto, E. Tavares, and F. Braga. Analyzing software performance and energy consumption of embedded systems by probabilistic modeling: An approach based on coloured Petri nets. In S. Donatelli and P. Thiagarajan, editors, *Petri Nets and Other Models of Concurrency - ICATPN 2006*, volume 4024 of *Lecture Notes in Computer Science*, pages 261–281, 2006. Cited on page 40.

[KFR09] R. Kainda, I. Flechais, and A. W. Roscoe. Usability and security of out-of-band channels in secure device pairing protocols. In *Proceedings of the 5th Symposium on Usable Privacy and Security*, pages 11:1–11:12. ACM, 2009. Cited on page 25.

[KHP$^+$09] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. CD$_X$: a family of real-time Java benchmarks. In *Proceedings of the 7th*

International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2009, Madrid, Spain, September 23-25, 2009, pages 41–50. ACM, 2009. Cited on page 74.

[Kin76] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976. Cited on pages 5 and 84.

[KLG$^+$13] S. Kerrison, U. Liqat, K. Georgiou, A. S. Mena, N. Grech, P. Lopez-Garcia, K. Eder, and M. V. Hermenegildo. Energy consumption analysis of programs based on XMOS ISA-level models. In G. Gupta and R. Peña, editors, *Proceedings of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)*. Springer, 2013. Cited on page 40.

[KMM00] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, Norwell, MA, USA, 2000. Cited on page 7.

[KPRT15] R. Kersten, S. Person, N. Rungta, and O. Tkachuk. Improving coverage of test cases generated by Symbolic PathFinder for programs with loops. *SIGSOFT Softw. Eng. Notes*, 40(1):1–5, January 2015. Cited on page 11.

[KPvv14] R. Kersten, P. Parisen Toldin, B. van Gastel, and M. van Eekelen. A Hoare logic for energy consumption analysis. In *Proceedings of the Third International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'13)*, volume 8552 of *Lecture Notes in Computer Science*, pages 93–109. Springer, 2014. Cited on page 11.

[KST$^+$09] A. Kobsa, R. Sonawalla, G. Tsudik, E. Uzun, and Y. Wang. Serial hook-ups: a comparative usability study of secure device pairing methods. In *Proceedings of the 5th Symposium on Usable Privacy and Security*, pages 10:1–10:12. ACM, 2009. Cited on page 25.

[KSvG$^+$12] R. Kersten, O. Shkaravska, B. van Gastel, M. Montenegro, and M. van Eekelen. Making resource analysis practical for Real-Time Java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 135–144. ACM, 2012. Cited on pages 12, 78 and 79.

[KvGD$^+$13] R. Kersten, B. van Gastel, M. Drijvers, S. Smetsers, and M. van Eekelen. Using model-checking to reveal a vulnerability of Tamper-Evident Pairing. In G. Brat, N. Rungta, and A. Venet, editors, *Proceedings of the 5th NASA Formal Methods Symposium*, number 7871 in Lecture Notes in Computer Science, pages 63–77. Springer, May 2013. Cited on page 10.

[KvGS$^+$14] R. W. Kersten, B. E. van Gastel, O. Shkaravska, M. Montenegro, and M. C. van Eekelen. ResAna: a resource analysis toolset for (real-time) JAVA. *Concurrency and Computation: Practice and Experience*, 26(14):2432–2455, 2014. Cited on pages 12 and 41.

[KWP07] C. Kuo, J. Walker, and A. Perrig. Low-cost manufacturing, usability, and security: An analysis of Bluetooth Simple Pairing and Wi-Fi Protected Setup. In S. Dietrich and R. Dhamija, editors, *Financial Cryptography and Data Security*, volume 4886 of *Lecture Notes in Computer Science*, pages 325–340. Springer, 2007. Cited on page 25.

[KZ08] A. Kansal and F. Zhao. Fine-grained energy profiling for power-aware application design. *SIGMETRICS Perform. Eval. Rev.*, 36(2):26–31, August 2008. Cited on page 40.

[LCFM09] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *CGO '09: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 136–146, Washington, DC, USA, 2009. IEEE Computer Society. Cited on pages 63, 75 and 99.

[Low98] G. Lowe. Towards a completeness result for model checking of security protocols. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pages 96 –105, June 1998. Cited on page 25.

[LPC^{+}07] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual. Draft Revision 1.200*, February 2007. Cited on pages 71 and 79.

[Mar98] F. Martinelli. Partial model checking and theorem proving for ensuring security properties. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pages 44 –52, 1998. Cited on page 25.

[MG07] R. Mayrhofer and H. Gellersen. On the security of ultrasound as out-of-band channel. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, pages 1–6, 2007. Cited on page 25.

[MGH07] R. Mayrhofer, H. Gellersen, and M. Hazas. Security by spatial reference: Using relative positioning to authenticate devices for spontaneous interaction. In J. Krumm, G. Abowd, A. Seneviratne, and T. Strang, editors, *UbiComp 2007: Ubiquitous Computing*, volume 4717 of *Lecture Notes in Computer Science*, pages 199–216. Springer, 2007. Cited on page 25.

[ML08] M. Martin and M. S. Lam. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *Proceedings of the 17th USENIX Security Symposium*, pages 31–43. USENIX Association, 2008. Cited on page 25.

[MPR12] E. Mercer, S. Person, and N. Rungta. Computing and visualizing the impact of change with Java PathFinder extensions. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, November 2012. Cited on page 37.

[MSB11] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing.* John Wiley & Sons, 2011. Cited on page 2.

[MSvE10] K. Madlener, S. Smetsers, and M. van Eekelen. A formal verification study on the Rotterdam storm surge barrier. In J. Dong and H. Zhu, editors, *Formal Methods and Software Engineering*, volume 6447 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2010. Cited on page 7.

[MSvEPn12] M. Montenegro, O. Shkaravska, M. van Eekelen, and R. Peña. Interpolation-based height analysis for improving a recurrence solver. In *Proceedings of the 2nd Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'11)*, volume 7177 of *Lecture Notes in Computer Science*, pages 36–53. Springer, 2012. Cited on pages 80 and 89.

[MW07] R. Mayrhofer and M. Welch. A human-verifiable authentication protocol using visible laser light. In *Proceedings of the Second International Conference on Availability, Reliability and Security*, pages 1143 –1148, April 2007. Cited on page 25.

[NMT^{+}11] B. Nogueira, P. Maciel, E. Tavares, E. Andrade, R. Massa, G. Callou, and R. Ferraz. A formal model for performance and energy evaluation of embedded systems. *EURASIP Journal on Embedded Systems*, pages 2:1–2:12, January 2011. Cited on page 40.

[NNH99] F. Nielson, R. H. Nielson, and C. L. Hankin. *Principles of Program Analysis.* Springer, second printing, 2005 edition, 1999. Cited on page 4.

[NRS14] A. Noureddine, R. Rouvoy, and L. Seinturier. Monitoring energy hotspots in software. *Automated Software Engineering*, pages 1–42, 2014. Cited on page 3.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Automated Deduction—CADE-11*, pages 748–752. Springer, 1992. Cited on page 7.

[ORY01] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001. Cited on page 5.

[Pau94] L. C. Paulson. *Isabelle: A generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994. Cited on page 7.

[PDEP08] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT'08/FSE-16, pages 226–237. ACM, 2008. Cited on page 37.

[Pie02] B. C. Pierce. *Types and programming languages*. MIT press, 2002. Cited on page 4.

[PKvv13] P. Parisen Toldin, R. Kersten, B. van Gastel, and M. van Eekelen. Soundness Proof for a Hoare Logic for Energy Consumption Analysis. Technical Report ICIS–R13009, Radboud University Nijmegen, October 2013. Cited on pages 11, 43, 46, 51 and 54.

[PR04] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 465–486. Springer, 2004. Cited on pages 41, 88 and 99.

[PR10] C. S. Păsăreanu and N. Rungta. Symbolic PathFinder: Symbolic execution of Java bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180. ACM, 2010. Cited on pages 5 and 28.

[PV04] C. S. Păsăreanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. In S. Graf and L. Mounier, editors, *Model Checking Software*, volume 2989 of *Lecture Notes in Computer Science*, pages 164–181. Springer, 2004. Cited on pages 29 and 32.

[QS82] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982. Cited on page 6.

[RA00] R. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 156 –165, 2000. Cited on page 25.

[Ran10] P. Ranganathan. Recipe for efficiency: principles of power-aware computing. *Commun. ACM*, 53(4):60–67, 2010. Cited on page 40.

[Rey02] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. *Proceedings of the 17th Annual IEEE Symposium on Symposium on Logic in Computer Science*, pages 55–74, 2002. Cited on page 5.

[RGBW07] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, November 2007. Cited on page 99.

[RHC76] C. Ramamoorthy, S.-B. F. Ho, and W. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, SE-2(4):293–300, December 1976. Cited on page 5.

[RN05] G. Reeves and T. Neilson. The mars rover Spirit FLASH anomaly. In *Aerospace Conference, 2005 IEEE*, pages 4186–4199, March 2005. Cited on page 6.

[RPB12] N. Rungta, S. Person, and J. Branchaud. A change impact analysis to characterize evolving program behaviors. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM'12)*, pages 109–118, Sept 2012. Cited on page 37.

[SA02] F. Stajano and R. Anderson. The resurrecting duckling: security issues for ubiquitous computing. *Computer*, 35(4):22 –26, apr 2002. Cited on page 25.

[Sax10] E. Saxe. Power-efficient software. *Commun. ACM*, 53(2):44–48, 2010. Cited on page 40.

[SC-11] SC-205/WG-71 Plenary. *DO-178C Software Considerations in Airborne Systems and Equipment Certification*, December 2011. Cited on page 3.

[SC01] A. Sinha and A. P. Chandrakasan. JouleTrack: A web based tool for software energy profiling. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 220–225. ACM, 2001. Cited on page 40.

[SCW$^+$05] B. Schwarz, H. Chen, D. Wagner, G. Morrison, J. West, J. Lin, and W. Tu. Model checking an entire Linux distribution for security violations. In *Computer Security Applications Conference, 21st Annual*, pages 10–22, 2005. Cited on page 25.

[SDF$^+$11] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: approximate data types for safe and general low-power computation. *SIGPLAN Not.*, 46(6):164–174, June 2011. Cited on page 40.

[SEKA06] N. Saxena, J.-E. Ekberg, K. Kostiainen, and N. Asokan. Secure device pairing based on a visual channel. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 6 pp. –313, May 2006. Cited on page 25.

[SGH$^+$13] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. Nori. A data driven approach for algebraic loop invariants. In M. Felleisen and P. Gardner, editors, *Programming Languages and Systems*, volume 7792 of *Lecture Notes in Computer Science*, pages 574–592. Springer, 2013. Cited on page 99.

[Sie02] F. Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages.* PhD thesis, University of Karlsruhe, 2002. Cited on pages 80 and 91.

[SJL$^+$09] T. Schoofs, E. Jenn, S. Leriche, K. Nilsen, L. Gauthier, and M. Richard-Foy. Use of PERC Pico in the AIDA avionics platform. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 169–178. ACM, 2009. Cited on page 74.

[SKVE10] O. Shkaravska, R. Kersten, and M. Van Eekelen. Test-based inference of polynomial loop-bound functions. In A. Krall and H. Mössenböck, editors, *PPPJ'10: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, ACM Digital Proceedings Series, pages 99–108, 2010. Cited on pages 12, 41 and 60.

[SKZS12] S. Schubert, D. Kostic, W. Zwaenepoel, and K. Shin. Profiling software for energy consumption. In *Proceedings of the 2012 IEEE International Conference on Green Computing and Communications (GreenCom)*, pages 515–522, November 2012. Cited on page 3.

[SNKvE14] M. Schoolderman, J. Neutelings, R. Kersten, and M. van Eekelen. ECAlogic: Hardware-parametric energy-consumption analysis of algorithms. In *Proceedings of the 13th Workshop on Foundations of Aspect-oriented Languages*, FOAL '14, pages 19–22. ACM, 2014. Cited on page 11.

[SPMS09] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 225–236. ACM, 2009. Cited on page 30.

[SVA07] J. Suomalainen, J. Valkonen, and N. Asokan. Security associations in personal networks: A comparative analysis. In F. Stajano, C. Meadows, S. Capkun, and T. Moore, editors, *Security and Privacy in Ad-hoc and Sensor Networks*, volume 4572 of *Lecture Notes in Computer Science*, pages 43–57. Springer, 2007. Cited on page 25.

[SvET11] O. Shkaravska, M. van Eekelen, and A. Tamalet. Collected size semantics for functional programs over lists. In *Proceedings of the 20th international conference on implementation and application of functional languages*, IFL'08, pages 118–137. Springer, 2011. Cited on page 98.

[SvEvK09] O. Shkaravska, M. van Eekelen, and R. van Kesteren. Polynomial size analysis of first-order shapely functions. *Logic in Computer Science*, 2:10(5), 2009. Cited on pages 41, 63, 64, 69 and 98.

[SvKvE07] O. Shkaravska, R. van Kesteren, and M. van Eekelen. Polynomial Size Analysis for First-Order Functions. In S. R. D. Rocca, editor, *Typed Lambda Calculi and Applications (TLCA'2007), Paris, France*, volume 4583 of *Lecture Notes in Computer Science*, pages 351–366. Springer, 2007. Cited on page 98.

[tBMB$^+$13] S. te Brinke, S. Malakuti, C. Bockisch, L. Bergmans, and M. Akşit. A design method for modular energy-aware software. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1180–1182. ACM, 2013. Cited on page 40.

[TC12] S.-L. Tsao and J. J. Chen. SEProf: A high-level software energy profiling tool for an embedded processor enabling power management functions. *Journal of Systems and Software*, 85(8):1757 – 1769, 2012. Cited on page 40.

[tMB$^+$14] S. te Brinke, S. Malakuti, C. M. Bockisch, L. M. J. Bergmans, M. Akşit, and S. Katz. A tool-supported approach for modular design of energy-aware software. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC'14. ACM, 2014. Cited on pages 40 and 60.

[Trt13] M. Trtík. *Symbolic Execution and Program Loops.* PhD thesis, Faculty of Informatics, Masaryk University, 2013. Cited on pages 29 and 36.

[TSv09] A. Tamalet, O. Shkaravska, and M. van Eekelen. Size analysis of algebraic data types. In P. Achten, P. Koopman, and M. Morazán, editors, *Trends in Functional Programming*, volume 9, pages 33–48. Intellect, 2009. Cited on page 98.

[vESvK$^+$07] M. van Eekelen, O. Shkaravska, R. van Kesteren, B. Jacobs, E. Poll, and S. Smetsers. AHA: Amortized heap space usage analysis. In M. Morazán, editor, *Selected Papers of the 8th International Symposium on Trends in Functional Programming (TFP'07), New York, USA*, pages 36–53. Intellect Publishers, UK, 2007. Cited on page 79.

[Vie11] S. Viehböck. Brute forcing Wi-Fi protected setup. `http://sviehb.files.wordpress.com/2011/12/viehboeck_wps.pdf`, 2011. Cited on pages 14 and 25.

[vKSvE08] R. van Kesteren, O. Shkaravska, and M. van Eekelen. Inferring static non-monotonically sized types through testing. In *16th International Workshop on Functional and (Constraint) Logic Programming (WFLP'07), Paris, France*, volume 216C of *Electronic Notes in Theoretical Computer Science*, pages 45–63, 2008. Cited on pages 63, 69, 78 and 98.

[vO01] D. von Oheimb. Hoare Logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001. Cited on page 5.

[WEE$^+$08] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008. Cited on pages 41 and 99.

[Weg75] B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975. Cited on page 87.

[WGR13] C. Wilke, S. Götz, and S. Richly. JouleUnit: A generic framework for software energy profiling and testing. In *Proceedings of the 2013 Workshop on Green in/by Software Engineering*, GIBSE '13, pages 9–14. ACM, 2013. Cited on page 3.

[Wi-06] Wi-Fi Alliance. *Wi-Fi Protected Setup Specification, version 1.0h*, 2006. Cited on page 15.

[WJCD00] C. Ware, J. Judge, J. Chicharo, and E. Dutkiewicz. Unfairness and capture behaviour in 802.11 adhoc networks. In *Proceedings of the 2000 IEEE International Conference on Communications*, pages 159–163, 2000. Cited on page 15.

[WQQC10] S. Wang, Z. Qiu, S. Qin, and W.-N. Chin. Stack bound inference for abstract Java bytecode. In *Proceedings of the 2010 4th IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 57–66, 2010. Cited on page 100.

[WW12] G. Wedzinga and K. Wiegmink. Using CHARTER tools to develop a safety-critical avionics application in Java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 125–134. ACM, 2012. Cited on pages 78 and 98.

[XLXT13] X. Xiao, S. Li, T. Xie, and N. Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. In *IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 246–256, 2013. Cited on pages 28 and 29.

[ZBA$^+$09] D. Zhurikhin, A. Belevantsev, A. Avetisyan, K. Batuzov, and S. Lee. Evaluating power aware optimizations within GCC compiler. In *GROW-2009: International Workshop on GCC Research Opportunities*, 2009. Cited on page 40.

Summary

Practically every modern electronic device is controlled by software. It is important to establish quality characteristics of this software, such as functional correctness, security or the ability to function with limited resources. Failure of critical systems can be financially costly or even lethal. Functional correctness and security are crucial for safe operation. In a world that is quickly depleted of its resources, energy-efficiency is vital. This thesis presents a series of software analysis methods, each of which aids the verification or inference of one or more of the aforementioned quality characteristics. It focuses on semantic analysis at compile-time and builds upon existing techniques where possible.

Most of this research was done in the context of the GoGreen project, in which a self-learning, secure and energy-efficient smart-home system is studied. Such a system combines wireless sensor nodes to observe a house and the people within its walls with intelligent algorithms that control heating, ventilation, lighting and appliances. The focus in this thesis is on properties that are crucial for such a system: functional correctness, security and efficient use of resources.

To protect the system and the privacy of its users, communication between devices in the GoGreen system must be secured by encrypting the stream of information. To enable this, the communicating parties must agree on a (pair of) key(s). Entering a secret key on a wireless sensor node is impossible, due to its limited interface. A solution is to use the Tamper-Evident Pairing protocol, which enables pairing of two devices by pressing a (virtual) button on both devices within a certain time-frame. However, some of the parameters needed to implement Tamper-Evident Pairing are under-specified. In Chapter 2, model-checking is used to analyse Tamper-Evident Pairing and it is discovered that the protocol has a security vulnerability if these parameters are not chosen wisely. Model-checking is applied in an iterative fashion to discover what values of the parameters result in a tamper-evident set-up. A constraint is formulated from the results that excludes occurence of the vulnerability.

For embedded software it is extra important that it behaves according to specification, because it can be very hard to update the software in case errors are detected after release. It must therefore be thoroughly tested. Chapter 3 presents a method to improve coverage of test cases generated by symbolic execution for programs with loops. It works by disregarding constraints over symbolic variables while symbolically executing the loop body, i.e. by taking the loop body out-of-context. This can produce models for symbolic variables that execute all branches within the loop body. The symbolic variables can then be concretised to these values.

Wireless sensor nodes are typically battery-powered. The GoGreen system as a whole is intended to save energy or at least be energy-neutral. It is thus crucial that wireless sensor nodes are energy-efficient. The hardware of the wireless sensor nodes is

controlled by embedded software. It is therefore preferable to use energy-efficient implementations. Chapter 4 presents a Hoare logic for energy-consumption analysis. Given a model of the energy-related behaviours of the hardware components, this analysis can statically bound the energy consumption of software running on said hardware. The method is sound and implemented in the tool ECALOGIC.

Wireless sensor nodes typically wake up for a short time to do their work, then go back into an energy-saving mode. Their program must be able to execute within this time-frame. It is therefore important to bound execution time. As most of the execution time of typical embedded programs is spent in loops, it is an important step to bound loop iterations. Furthermore, loop bounds are a prerequisite for analysing consumption of any resource, as a certain amount of resources (possibly bounded) can be consumed on every iteration. Chapter 5 presents a novel method to infer polynomial ranking functions for loops. It instruments the loop with a counter, then runs it for a set of test inputs and interpolates a polynomial over the resulting iteration counts. The set of test inputs is chosen such that it satisfies a condition that guarantees the existence of a unique interpolating polynomial.

To be able to cheaply produce wireless sensor nodes, they must be equipped with just as much memory as needed, not more. It is therefore important to be able to bound memory consumption of their embedded software. Chapter 6 presents the tool RESANA and the underlying resource analysis methods. The loop bound analysis method is further extended with a way to deal with so-called *condition jumping.* A heap-space analysis is developed, using extensions of the resource analysis tool COSTA, which is based on recurrence relation solving. COSTA is extended by applying the interpolation-based ranking function inference method to recurrence relation solving, correcting its results for arrays, simplifying its results and adding a specialisation for AICAS JAMAICAVM. Furthermore, a stack-space analysis is presented. This analysis uses COSTA to obtain a measure for recursive functions (analogous to a ranking function for loops), then combines this with data-flow analysis results and measured stack frame sizes from AICAS VERIFLUX to obtain a concrete upper bound on the consumed stack space.

This thesis contributes a series of software analysis methods. The methods aid in the establishment of program properties that are of particular interest for resource-sensitive systems, such as security, functional correctness and efficient use of resources. All the presented automatic analysis methods have been implemented in tools.

Samenvatting

Praktisch elk modern elektronisch apparaat wordt aangestuurd door software. Het is belangrijk om bepaalde kwaliteitseigenschappen van deze software vast te stellen. Hierbij kan gedacht worden aan functionele correctheid, aan het bestand zijn tegen aanvallen van hackers en aan de mogelijkheid te functioneren binnen systeem dat beperkt is met betrekking tot geheugen, energie en tijd. Het falen van bepaalde systemen kan kostbaar of zelfs dodelijk zijn. Functionele correctheid en bestandheid tegen aanvallen zijn dus cruciaal voor de veiligheid. In een wereld waarin grondstoffen in rap tempo verbruikt worden is energie-zuinigheid geboden.

Het grootste gedeelte van dit onderzoek is gedaan binnen het GoGreen project. In dit project wordt een zelf-lerend, veilig en energie-efficiënt domoticasysteem bestudeerd. Een dergelijk systeem observeert een huis en de mensen die er wonen door middel van draadloze sensormodules. De verzamelde data dient als invoer voor intelligente algoritmes die verwarming, ventilatie, verlichting en huishoudelijke apparatuur aansturen. Dit proefschrift presenteert een serie software analysemethoden die bijdragen aan het vaststellen van kwaliteitseigenschappen die van belang zijn voor een dergelijk systeem: functionele correctheid, bestandheid tegen aanvallen en efficiënt gebruik van beschikbare tijd, energie en geheugenruimte.

Om het systeem te beveiligen en de privacy van zijn gebruikers te waarborgen moet de communicatie tussen de verschillende draadloze apparaten versleuteld worden. Om dit mogelijk te maken, dient er een sleutel(paar) afgesproken te worden. Het invoeren van een geheime sleutel op een draadloze sensor-module is onmogelijk vanwege de interfacebeperkingen. Een oplossing voor dit probleem is het gebruik van het Tamper-Evident Pairing protocol, waarbij een gebruiker op beide apparaten een (virtuele) knop indrukt binnen een kort tijdsbestek. Helaas is een aantal parameters, die nodig zijn om dit protocol te implementeren, niet gespecificeerd. In hoofdstuk 2 wordt een kwetsbaarheid ontdekt die optreedt indien deze parameters onzorgvuldig gekozen worden. Deze kwetsbaarheid is gevonden door iteratieve toepassing van de model-checking techniek. Er wordt een propositie opgesteld over de desbetreffende parameters, welke het optreden van de kwetsbaarheid uitsluit.

Het kan erg lastig zijn om apparaten zoals draadloze sensormodules van nieuwe software te voorzien. Voor software die is ingebed in een dergelijk apparaat is het daarom bijzonder belangrijk dat deze functioneel correct is. Deze software dient dus uitgebreid getest te worden. De symbolische executietechniek kan worden gebruikt om een testset te genereren. In hoofdstuk 3 wordt een methode gepresenteerd die de dekking van de gegenereerde testset met betrekking tot de geteste code verbetert voor programma's die loops bevatten. Deze aanpak werkt door de condities die zijn opgebouwd rondom de symbolische variabelen even te negeren en de romp van de loop dus symbolisch te executeren buiten zijn context. Dit leidt tot waarden voor de symbolische variabelen die het verwerken van alle paden binnen de loop garanderen. De symbolische variabelen kunnen vervolgens naar deze waarden worden geconcretiseerd.

Draadloze sensormodules worden meestal van stroom voorzien door middel van een batterij. Het is daarom cruciaal dat deze modules energie-zuinig functioneren. De hardware van een dergelijke module wordt aangestuurd door software. Het verdient daarom de voorkeur om energie-zuinige implementaties te gebruiken. In hoofdstuk 4 wordt een Hoare logica gepresenteerd waarmee een bovengrens kan worden bepaald voor het energiegebruik van software, uitgaand van modellen van de hardware waarop deze software wordt uitgevoerd. De *soundness* van de methode is wiskundig bewezen en de methode is geïmplementeerd in de tool ECAlogic.

Draadloze sensormodules moeten hun werk meestal doen in korte actieve periodes, waarna ze weer naar een slaapstand schakelen. Hun programma moet binnen dit korte tijdsbestek uitvoerbaar zijn. Het is daarom van belang om een bovengrens aan de uitvoertijd te bepalen. Een belangrijke stap hierin is het afleiden van een bovengrens aan het aantal iteraties van loops. Bovendien is deze bovengrens een vereiste om de consumptie van andere middelen als geheugen of energie binnen de loop te kunnen bepalen. Tijdens iedere iteratie kan er immers een bepaalde hoeveelheid van deze middelen gebruikt worden. In hoofdstuk 5 wordt een nieuwe methode gepresenteerd, waarmee polynomiale bovengrenzen aan het aantal iteraties van loops kunnen worden afgeleid. Er wordt hiervoor een teller aan de loop toegevoegd, waarna de loop wordt uitgevoerd voor een set invoerwaardes, waarna de resultaten worden geïnterpoleerd. De set invoerwaarden wordt slim gekozen, waardoor er een unieke polynomiale interpolatie bestaat.

Om draadloze sensormodules goedkoop te kunnen produceren moeten deze worden uitgerust met zo weinig mogelijk geheugen. Het is daarom van belang een bovengrens te bepalen voor het geheugengebruik van hun software. In hoofdstuk 6 wordt de tool ResAna gepresenteerd, evenals de onderliggende software analysemethoden. De loop analyse uit hoofdstuk 5 wordt verder uitgebreid. Er wordt een *heap* analyse ontwikkeld die is gebaseerd op de tool COSTA. Deze tool is werkt middels het oplossen van recurrente betrekkingen en kan worden gebruikt voor de analyse van het gebruik van geheugen, tijd en andere middelen, op basis van een kostenmodel. COSTA wordt uitgebreid met een toepassing van de interpolatiemethode op het oplossen van recurrente betrekkingen. Verdere toevoegingen zijn een correctie van de resultaten voor arrays, een simplificatie van de resultaten en een specialisatie voor de Aicas JamaicaVM. Er wordt ook een *stack* analyse gepresenteerd. Deze gebruikt COSTA om een symbolische bovengrens aan de recursie van methoden te bepalen. Deze wordt gecombineerd met resultaten van data-flow analyse en gemeten groottes van stack-frames vanuit Aicas VeriFlux om tot een concrete bovengrens te komen.

Dit proefschrift presenteert een serie software analysemethodes. Deze methodes dragen bij aan de vaststelling van functionele correctheid, van bestandheid tegen aanvallen van hackers en van efficiënt gebruik van middelen als geheugen, energie en tijd. Alle gepresenteerde automatische methodes zijn geïmplementeerd in software tools.

Curriculum Vitae

Rody Kersten

Born in Nijmegen, the Netherlands on May 29, 1983.

September 1995 - August 2001
Pre-university secondary education (VWO)
Nijmeegse Scholengemeenschap Groenewoud

September 2001 - January 2010
Master's degree in Computer Science
Radboud University Nijmegen

September 2007 - June 2008
Software developer
EntiQ B.V.

June 2008 - April 2009
Freelance software developer

February 2010 - January 2011
Scientific programmer
Radboud University Nijmegen

February 2011 - August 2015
PhD candidate
Radboud University Nijmegen
Supervised by Prof. dr. Marko van Eekelen
Co-supervised by Dr. Sjaak Smetsers

July 2014 - September 2014
Research intern
NASA Langley Research Center, Hampton, VA, USA
Supervised by Dr. Suzette Person

February 2015 - August 2015
Assistant Professor
Open University of the Netherlands

October 2015 -
Research associate
Carnegie Mellon University, Silicon Valley Campus, Mountain View, CA, USA

Titles in the IPA Dissertation Series since 2009

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers.* Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software.* Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra.* Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications.* Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services.* Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols.* Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates.* Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement.* Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components.* Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors.* Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory.* Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution.* Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes.* Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains.* Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations.* Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice.* Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars.* Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns.* Faculty of Mathematics and Natural Sciences, UL. 2011-24

J. Wang. *Spiking Neural P Systems.* Faculty of Mathematics and Natural Sciences, UL. 2011-25

A. Khosravi. *Optimal Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2012-01

A. Middelkoop. *Inference of Program Properties with Attribute Grammars, Revisited.* Faculty of Science, UU. 2012-02

Z. Hemel. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

T. Dimkov. *Alignment of Organizational Security Policies: Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

S. Sedghi. *Towards Provably Secure Efficiently Searchable Encryption.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference.* Faculty of Science, Mathematics and Computer Science, RU. 2012-06

K. Verbeek. *Algorithms for Cartographic Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2012-07

D.E. Nadales Agut. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation.* Faculty of Mechanical Engineering, TU/e. 2012-08

H. Rahmani. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2012-09

S.D. Vermolen. *Software Language Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

L.J.P. Engelen. *From Napkin Sketches to Reliable Software.* Faculty of Mathematics and Computer Science, TU/e. 2012-11

F.P.M. Stappers. *Bridging Formal Models – An Engineering Perspective.* Faculty of Mathematics and Computer Science, TU/e. 2012-12

W. Heijstek. *Software Architecture Design in Global and Model-Centric Software Development.* Faculty of Mathematics and Natural Sciences, UL. 2012-13

C. Kop. *Higher Order Termination.* Faculty of Sciences, Department of Computer Science, VUA. 2012-14

A. Osaiweran. *Formal Development of Control Software in the Medical Systems Domain.* Faculty of Mathematics and Computer Science, TU/e. 2012-15

W. Kuijper. *Compositional Synthesis of Safety Controllers.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16

H. Beohar. *Refinement of Communication and States in Models of Embedded Systems.* Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards.* Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. *Interactive mathematical documents: creation and presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. *Composition and synchronization of real-time components upon one processor*. Faculty of Mathematics and Computer Science, TU/e. 2013-08

J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins*. Faculty of Mathematics and Computer Science, TU/e. 2013-09

S. van der Burg. *A Reference Architecture for Distributed Software Deployment*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

J.J.A. Keiren. *Advanced Reduction Techniques for Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2013-11

D.H.P. Gerrits. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points*. Faculty of Mathematics and Computer Science, TU/e. 2013-12

M. Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

M.J.M. Roeloffzen. *Kinetic Data Structures in the Black-Box Model*. Faculty of Mathematics and Computer Science, TU/e. 2013-14

L. Lensink. *Applying Formal Methods in Software Development*. Faculty of Science, Mathematics and Computer Science, RU. 2013-15

C. Tankink. *Documentation and Formal Mathematics — Web Technology meets Proof Assistants*. Faculty of Science, Mathematics and Computer Science, RU. 2013-16

C. de Gouw. *Combining Monitoring with Run-time Assertion Checking*. Faculty of Mathematics and Natural Sciences, UL. 2013-17

J. van den Bos. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics*. Faculty of Science, UvA. 2014-01

D. Hadziosmanovic. *The Process Matters: Cyber Security in Industrial Control Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

A.J.P. Jeckmans. *Cryptographically-Enhanced Privacy for Recommender Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

C.-P. Bezemer. *Performance Optimization of Multi-Tenant Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04

T.M. Ngo. *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

A.W. Laarman. *Scalable Multi-Core Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

J. Winter. *Coalgebraic Characterizations of Automata-Theoretic Classes*. Faculty of Science, Mathematics and Computer Science, RU. 2014-07

W. Meulemans. *Similarity Measures and Algorithms for Cartographic Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2014-08

A.F.E. Belinfante. *JTorX: Exploring Model-Based Testing.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

A.P. van der Meer. *Domain Specific Languages and their Type Systems.* Faculty of Mathematics and Computer Science, TU/e. 2014-10

B.N. Vasilescu. *Social Aspects of Collaboration in Online Software Communities.* Faculty of Mathematics and Computer Science, TU/e. 2014-11

F.D. Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2014-12

N. Noroozi. *Improving Input-Output Conformance Testing Theories.* Faculty of Mathematics and Computer Science, TU/e. 2014-13

M. Helvensteijn. *Abstract Delta Modeling: Software Product Lines and Beyond.* Faculty of Mathematics and Natural Sciences, UL. 2014-14

P. Vullers. *Efficient Implementations of Attribute-based Credentials on Smart Cards.* Faculty of Science, Mathematics and Computer Science, RU. 2014-15

F.W. Takes. *Algorithms for Analyzing and Mining Real-World Graphs.* Faculty of Mathematics and Natural Sciences, UL. 2014-16

M.P. Schraagen. *Aspects of Record Linkage.* Faculty of Mathematics and Natural Sciences, UL. 2014-17

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World.* Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction.* Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems.* Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness.* Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

S. Dietzel. *Resilient In-network Aggregation for Vehicular Networks.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

E. Costante. *Privacy throughout the Data Cycle.* Faculty of Mathematics and Computer Science, TU/e. 2015-08

S. Cranen. *Getting the point — Obtaining and understanding fixpoints in model checking.* Faculty of Mathematics and Computer Science, TU/e. 2015-09

R. Verdult. *The (in)security of proprietary cryptography.* Faculty of Science, Mathematics and Computer Science, RU. 2015-10

J.E.J. de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols.* Faculty of Science, Mathematics and Computer Science, RU. 2015-11

Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems.* Faculty of Mathematics and Computer Science, TU/e. 2015-12

J. Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars.* Faculty of Science, UU. 2015-13

S. Picek. *Applications of Evolutionary Computation to Cryptology*. Faculty of Science, Mathematics and Computer Science, RU. 2015-14

C. Chen. *Automated Fault Localization for Service-Oriented Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

S. te Brinke. *Developing Energy-Aware Software*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16

R.W.J. Kersten. *Software Analysis Methods for Resource-Sensitive Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2015-17