ASSESSING SUSTAINABILITY OF SOFTWARE

analysing correctness, memory and energy consumption

Bernard van Gastel

Op deze wijze wil ik graag al mijn lieve familie, vrienden en collega's bedanken voor hun grote steun de afgelopen tijd. Maar bovenal wil ik Floor bedanken, mijn maatje, die dag en nacht voor mij klaar staat, met haar onbeperkte energie, waardoor mijn leven een stuk meer kleur krijgt.

Assessing sustainability of software

analysing correctness, memory and energy consumption

proefschrift ter verkrijging van de graad van doctor
aan de Open Universiteit op gezag van de rector magnificus
prof. mr. dr. A. Oskamp
ten overstaan van
een door het College voor promoties ingestelde commissie
in het openbaar te verdedigen
op vrijdag 28 oktober 2016 te Heerlen om 13.30 uur precies door

Bernard Erik van Gastel
geboren op 12 maart 1983 te Arnhem

promotor

prof. dr. M.C.J.D. (Marko) van Eekelen, Open Universiteit en Radboud Universiteit

overige leden beoordelingscommissie

prof. dr. J.H. (Herman) Geuvers, Radboud Universiteit

prof. dr. P. (Patricia) Lago, Vrije Universiteit Amsterdam

prof. R. (Ricardo) Peña, Universidad Complutense de Madrid, Spanje

prof. dr. E. (Erik) Barendsen, Open Universiteit en Radboud Universiteit

prof. dr. A. (Lex) Bijlsma, Open Universiteit

paranimfen

Geert-Jan Besjes

Petra Molenaar

CONTENTS

INTRODUCTION 1

1

CONTEXT *Many aspects of modern life depend on, and are controlled by, software. When one wakes up, the alarm clock that wakes them is controlled by software. During the day, almost every device one uses contains software, including fridges [Tro15], cars, televisions, elevators, and perhaps even coffee machines. Just like a driver has a huge impact on the efficiency, behaviour and needed maintenance of a car, software has a huge impact on the efficiency, behaviour and needed maintenance of the device running that software. With Earth's dwindling resources, it is important these devices, including their software, function in a sustainable manner. That is further strengthened by the fact that software is essential to let our contemporary society function. From tax returns, online theft reports, applications for benefits, to construction work and of course communications: all are handled by software. In this thesis, methods to analyse energy consumption and memory usage of (parts of) software are proposed. The obtained results can be used to optimise energy consumption of software and the devices it runs on, therefore making these devices more sustainable. Correctness of software is also considered, as misbehaving software can have a significant negative impact on our environment. As much as energy can be wasted by software, our quality of life can also be adversely affected by software's (lack of) quality.*

photo 1.1 Launch of an Ariane 5, by ESA.

If software demonstrates erroneous behaviour, undesirable effects can occur. These effects can result in having negative impact on users of the software and devices, or to the environment of the devices running the software. In case of safety-critical systems in for example cars or airplanes, the consequences can lead to serious injury or even loss of life. An early example of problems that arouse from using software is the medical device **Therac-25**, administrating doses for radio therapy. Between 1985 and 1987, (concurrent) programming bugs resulted in massive radiation overdoses, over 100 times the intended dose, eventually killing five people and leaving others serious injured. Another notorious example with a catastrophic ending is the launch of the first **Ariane 5** rocket in 1996 (see photo 1.1), which resulted in a gigantic explosion that polluted the environment significantly. The cause was determined to be a number conversion errors (overflows), which originated from the reuse of existing software. This software did function correctly aboard the Ariane 4, but changing circumstances were not taken into account: the Ariane 5 had a greater horizontal acceleration.

However, the software was not tested under these new circumstances. Had such testing occurred, it would have shown that the increased horizontal acceleration invalidated a condition of the used software about the flight path. Although methods to verify correctness existed at the time, they were not used in practice. The aftermath resulted in one of the first large scale effective industrial applications of automated (static) analysis methods (using abstract interpretation) [Lac+98]. Nowadays, much research is dedicated to researching safety critical systems, with numerous research programmes.

Recently, another example of (lack of) correctness was made public. General Motors recalled 4.3 million vehicles on September 9th, 2016, due to a software bug [Tim16]. The control system of the safety systems in the car, controlling the airbags and seat belt systems, is running software. Due to specific driving conditions, a diagnostic mode was triggered in this system, preventing actual deployment of the airbags, and preventing properly securing the seat belts. The vehicles were sold over a period of four years leading up to the announcement. The bug resulted in at least one death and three injures. General Motors recalls all affected models, which is a big operation that costs a lot, being energy, manpower, gasoline, replacement parts, etc. To avoid wasting resources, we consider correctness a prerequisite of sustainable software.

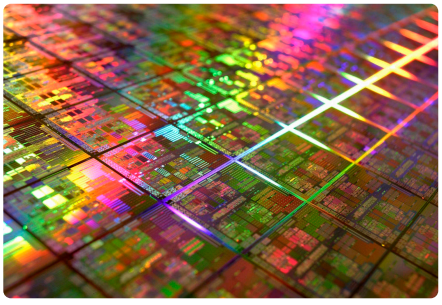

photo 1.2 A silicon wafer with multiple AMD Phenom II processors, by AMD.

In order to ensure that software is correctly executed on a processor, one must verify this hardware component. A modern (Intel Haswell-EX Xeon E7 v3, summer 2015) processor consists of 5.7 billion transistors, stacked in multiple layers, designed by hundreds of people in a period of up to decades. A small error can cost a hardware company an enormous amount of money and work. In 1994, Intel faced a problem with their newly introduced Pentium processors. During certain floating point operations, the processor produces an incorrect result, from the fourth significant decimal. All affected processors were recalled, and Intel reported a $475 million loss in earnings in 1995.

However, correctness can also extend over other quality aspects of software. Limited memory usage is important for the correct working of software, especially in devices intended for daily use (embedded systems). If a system runs out of memory, the program or system could crash or enter a reduced-performance state known as trashing. This

trashing also costs a significant amount of energy. If the result of analysing a program yields that it will only uses a maximum of memory at any given time, these devices can be built with exactly the right amount of memory, rather than a huge and expensive surplus of memory. In addition to offering correctness guarantees, this in turn reduces the cost for producing the system. Such an analysis can also give valuable insight to a programmer, so they can optimise the memory usage of a program. An additional benefit from this optimisation is that the amount of memory used has a significant impact on the energy consumption of the device running the software. When reducing the memory consumption of a program, the energy consumption will also be reduced. In order to be sustainable, a prerequisite to software is therefore that it uses only a limited amount of memory.

photo 1.3 Measuring pollution of a Volkswagen Golf 2.0, by Patrick Pleul/AFP.

As traditionally many savings did occur in the hardware side of a computer, energy consumption is a blind spot when developing software. Each next hardware generation consumed less energy for performing the same amount of work. However, recently this development has lost its pace. At the same time, it becomes more and more clear that software has a huge impact on the behaviour and the properties of devices it runs on. A recent example of software influencing the working of a device is the Volkswagen scandal. The car manufacturer used software to detect if the car was being tested. If this was found to be the case, the diesel motor was programmed to operate in such a way that it exhausted less toxic gases and fumes (see photo 1.3). In [OZH16] it is calculated that 44,000 years of human life are lost in Europe because of the fraud, which lasted at least 6 years. Another example are fridges from Panasonic, which could detect if a test was going on and suppressed energy intensive defrost cycles during this test. These are negative examples, but they do make clear that the software is in control of the device and its (energy) behaviour.

Although software is evidently in control of the devices, there is almost no time dedicated in most computer science curricula to energy efficiency of software. This is peculiar since energy is of vital importance to modern (software) industry. For years, data centres have been located at places where the energy is cheap, and since the rise of the smartphone more software engineers recognise that in order to get good user reviews, their software should not rapidly deplete the battery charge of the user's phone. As a result, most aspiring

programmers never learn to produce energy efficient code. Software engineers have trouble assessing how much energy will be consumed by their software on a target device, especially when the software is run on a multitude of different systems. With the advent of the internet of things, where software is increasingly embedded in our daily life, it is increasingly important that the **software industry as a whole becomes aware of their energy footprint**, and methods are developed to reduce this footprint.

Furthermore, the combination of many individual negative effects can also affect our society at large. Although this effect is less direct, it is no less essential. If devices that are present in large quantities in our society all exhibit the same negative behaviour, such as incurring needlessly a too high energy consumption, they can impact public utilities and our economy, and will consume the finite resources of Earth even faster. Governments increasingly recognise this societal effect, as indicated by the new laws in the European Union issuing ecodesign requirements for all kind of devices. One of the aims of these requirements is to make devices more energy efficient. Examples of product categories with ecodesign requirements include vacuum cleaners, electrical motors, lightning, heaters, cooking appliances [EU-1], televisions [EU-2] and coffee machines [EU-3]. Even requirements leading to relative small improvements in energy efficiency can yield large results at scale, even in the case of devices of which one would expect no significant electricity savings to be possible. Although it might not necessarily run software, a vacuum cleaner can be such a device. By reducing the energy consumption of vacuum cleaners **alone**, **20 TWh** of electricity can be saved in 2020, by estimation of the European Union itself [EU-4]. To put that number into context, the Netherlands used 96.8 TWh of electrical energy in 2013 [EU-5].

Over the past years, worldwide electricity generation continued to increase, year by year: **from 6,131 TWh in 1973 to 23,322 TWh in 2013** [IEA15a], representing a **growth of 280%**. The population growth from 3,909,722,120 to 7,181,715,139 people during the same period was 84% [UN16]. More and more energy per capita is consumed around the world. Most of the growth in energy generation per capita is attributed to China (97%), and the regional attribution of energy generation shows that Africa, Asia and the Middle East are taking a larger share of the worldwide production. The average energy consumption per capita is distributed unevenly. According to [IEA15a] in 2013 the United States consumed 12,987 kWh/capita of electricity, Germany 7,022 kWh/capita, the Netherlands 6,823 kWh/capita, Spain 5,404 kWh/capita, and Italy 5,124 kWh/capita. However, in comparison, for example only 783 kWh/capita in India and 3778 kWh/capita in China are consumed. As a society

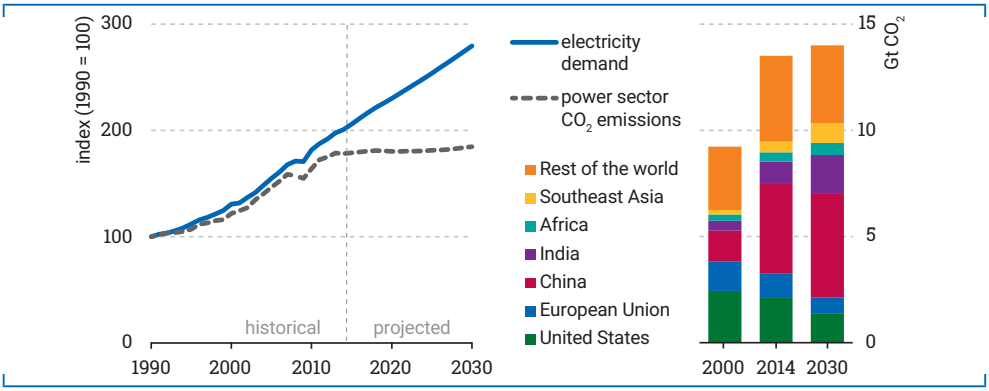

figure 1.4 Growth in world electricity demand and related CO2 emissions since 1990 (left) and related CO_2 emissions by region (right). Source: figure 2 in [IEA15b].

we have already difficulty with meeting our energy appetite, which will only increase if people in countries as China and India would consume the same amount of energy as in the United States and Europe. According to [IEA15b], the outlook appears the same, with a significant shift in regional CO_2 emissions, as can be found in figure 1.4.

This context is the basis for the argument that it is important to reduce energy consumption, even though the potential savings are relatively small: even small savings in many devices will yield a big societal and environmental impact if combined. We consider a low energy consumption of a device and its software a prerequisite for it to be used in practice, as this is required for these devices to be applicable on a large scale in our society.

In this thesis we focus on three aspects of software: **analysing its correctness**, **analysing its memory usage** and **analysing its energy-consumption**. Although at first sight they look different, these share the same (sociality) context. Moreover, they share many techniques, and build on a common foundation. Some parts of this thesis relate to all aspects, as these parts describe general techniques that can be used for multiple intents.

Using these proposed analysis methods one can verify that certain correctness problems are absent in a piece of software, analyse how much memory the (Java) software will use when executed, and how much energy devices controlled by (ECA) software will consume. A programmer can use these analysis methods to compare algorithms and implementations, and asses which one is the most sustainable. That assessment will depend heavily on the context and evental usage of a device. We continue with discussing the techniques used in more detail.

1.1 Shared techniques

A common method the aspects share is **resource analysis**. The method starts with identifying a certain resource, and continues with analysing all the usages of this particular resource. In this thesis the resources studied are **memory** and **energy**. A third resource, time, is needed in order to analyse energy. We will discuss time only briefly. Resource analysis is a **static analysis method**, meaning the program is analysed without running it. This has a number of advantages as opposed to measuring the energy consumption in a test setup. During the development phase, without actually building the product, the energy usage can be predicted. Once the product is actually built, benchmarking energy usage can be expensive, both in terms of energy and cost. That holds especially for large industrial applications, e.g. a whole factory or a large communications infrastructure.

Another common theme in the following chapters is the **transformation of programs**. By transforming the program into another one analysing the program becomes feasible. This transformed program abstracts from all kinds of details that are in the original program, while being equivalent for one particular property, allowing it to be analysed. That property can be memory usage or energy consumption.

Another method is, instead of making an transformation, to **overapproximate** a certain property. This is a powerful techniques but it has its drawbacks. On the one side, the use of sound overapproximations, allows one derive properties that hold always regardless of the input, i.e. a maximum. These are most of the time symbolic: input variables can occur in the resulting expression, as opposed to a concrete value. In order for the results to be correct, stronger assumptions on the input and/or the structure of the program are needed. Another drawback is, as the name implies, imprecision: the derived approximated maximum can be a significant factor worse than the actual maximum.

In all these shared techniques **assumptions** play an important role. Stronger assumptions lead to easier to derive properties and higher-quality properties. It is therefore key to clearly identify those properties that are important to a certain case, and make a balanced consideration of which techniques to apply.

1.2 Correctness

Analysing a piece of code for a certain property is often non-trivial. A large part is deciding which properties to verify, but also verifying the property itself can be hard. For code doing a concrete calculation, correctness can be defined by equivalence with the mathematical version of the calculation. One can check conformance to this equivalence by hand, or automatically with model checking, or computer assisted with interactive theorem proving. However, this is not enough to ensure correctness. Once calculated, one also must ensure that the values are not modified. As an example: in the case of the programming language C, there can be no accidental unintended writes (e.g. 'dangling pointers') anywhere in the program. Static verification can provide the means to ensure that these values are not modified after initialisation. However, one can also apply model checking or interactive theorem proving to ensure no modifications, using different models as for checking correctness of the actual calculation.

In a concurrent setting, when e.g. the loop on line 16 of listing 1.5 is executed concurrently, the next step is to assure proper access control is applied to variables shared between concurrent running threads. Mutual exclusion can be used to ensure that no race conditions occur. Vital to the used mutual exclusion algorithm is the absence of deadlock and absence of starvation. Otherwise no progress is made although the device running the software is consuming energy, a huge waste of resources. The absence of deadlock and absence of starvation is analysed for one specific mutual exclusion algorithm, the readers-writers algorithm, in chapter 2.

A general assumption that is often made is the assumption that no errors occur in the transformation step from program to hardware instructions. This step is done by the compiler, and to ensure absence of errors a formally verified compiler can be used.

However, zooming in on the **hardware level**, many errors can occur. A production error could impact the behaviour of the processor, resulting in unintended behaviour. Just as in software, errors in (hardware) designs can also induce unintended behaviour in software. Although the largest part of this thesis is about software, chapter 3 is devoted to hardware. Hardware is essential to software: all derived prerequisites about software hold under the assumption that the hardware running the software functions correctly, making correctly functioning hardware a necessary condition for software. In chapter 3 a method for analysing the correctness conditions of (parts of) processors is proposed.

RUNNING EXAMPLE *To illustrate the prerequisites to sustainable software, the C++ example code in listing 1.5 is used as a running example. This is part of an implementation of the Bayesian sets algorithm as described in [GH06], which can be used as a recommender system. To give a concrete example: in the beer app 'BierApp' (available on iOS and Android) developed by the author, the algorithm is used to recommend new beers to the users, based on earlier ratings of other beers. The algorithm is fast and works offline. After the preprocessing step, when doing a search (the term used for executing the recommendation algorithm), the* `alpha` *and* `beta` *values are used often so it is efficient to calculate them once. All properties of an object must be available, however because the input of the algorithm is groups of related beers, a mapping* `objectProperties` *is constructed which maps objects to properties. Moreover, the algorithm has to work on various devices, with the lowest powered ones only having 20 MiB of memory available for the app (including the interface). This poses a serious challenge.*

The values calculated must be calculated according to the algorithm in [GH06]. Static verification can validate no values are written to the variables after initialisation, through pointers that are faulty or in any other way.

```
template <class E>
class BSets {
    static const int C = 2;
    std::vector<E> objects;
    std::multimap<E, int> objectProperties;
    int propCount;
    double* alpha = nullptr;
    double* beta = nullptr;
  public:
    BSets(const std::vector<E> &elements, const std::vector<std::set<E>> &properties) :
            objects(elements),
            propCount(properties.size()),
            alpha(new double[propCount]),
            beta(new double[propCount]) {
        double n = (double)elements.size();
        for (int i = 0; i < properties.size(); ++i) {
            double m = properties[i].size() / n;
            alpha[i] = C * m;
            beta[i] = C * (1-m);
            for (const auto& object : properties[i])
                objectProperties.insert(std::make_pair(object, i));
        }
    }
    ...
}
```

listing 1.5 Preprocessing step performed in an C++ implementation of Bayesian sets [GH06].

1.3 Memory

As already mentioned, if a program uses a lot of memory, the system can enter a worst case state known as trashing. This has a huge effect on performance, and therefore also on energy usage. Keeping the memory usage of your program in check also keeps your energy bill in check.

Besides trashing, memory allocators have an overhead associated with them. There is a lot of book keeping involved, to manage all the free parts of memory, and join continuous blocks of memory together so larger allocations can use them. By reducing the number of memory allocations, this overhead is also reduced. This saves both execution time and energy, besides having better worst case behaviour.

RUNNING EXAMPLE *The* `alpha` *and* `beta` *arrays of doubles are allocated on lines 13 and 14 of listing 1.5, both of size* `propCount`*, which is equal to* `properties.size()`*. The element to property map is build on line 21, which allocates for each* ⟨object, property⟩ *tuple (in which* object *can be of any type, as it is a template type, and* property *is a number signifying a property) an internal storage node of the* std::multimap *class. Depending on the implementation, the number of bytes allocated for the two objects can actually differ. Symbolically the amount of heap memory used after the constructor has finished is listed below, with* |T| *denoting the memory taken when allocating an instance of datatype* T*.*

$$2 \cdot \texttt{properties.size()} \cdot |\text{double}| + \sum_{g \in \texttt{properties}} g\texttt{.size()} \cdot |\text{std::multimap<E, int>::node}|$$

For the programmer, a symbolic bound abstracts from platform specific details, and results in additional insight in the workings of the algorithm. This insight can be used to change the program to allow for more memory efficient behaviour. An approach to practical memory analysis of Java programs is proposed in chapter 4, which aids a programmer in their day-to-day work. In addition to heap memory usage, stack memory usage is also considered.

1.4 Energy

As already mentioned, energy increasingly plays an important role in the software industry. Both memory and performance can be an indicator for energy consumption of a program. However, it is also possible to analyse energy consumption more directly by means of energy analysis on the source code level.

RUNNING EXAMPLE *Returning to the example of the preprocessing step in the Bayesian Sets algorithm, it is useful to analyse the effect of the preprocessing of* `alpha` *and* `beta` *on the energy usage. In the preprocessing step, four arithmetic operations and two size lookups are performed. If the energy usage of an arithmetic operation is equal to* a*, and a lookup equal to* l*, the energy cost of a single preprocessing step is thus* $4a + 2l$*. In the* `search()` *method of* BSets*, for each unique property calculations are executed depending on a value in* `alpha` *and* `beta` *respectively. If the number of searches is* s*, the amount of energy saved by preventing redundant calculations is* $(s-1) \cdot (4a + 2l)$*. In practice this number differs, because of caching, extra energy to maintain extra values in memory, and several other reasons. However, like the resulting memory expressions, it gives a programmer insight in the energy behaviour of the program.*

An important aspect in order to analyse energy is the construction and validity of energy models of hardware. As it is the hardware that consumes energy, a realistic energy model is needed. However, energy consumption is not in all cases easily modelled, as it can depend on ambient temperature, wear of the hardware, radio signals, and various other variables. To be applicable, constraints are required on the energy models of hardware.

Lately, research in the relation between energy consumption and software has gained traction, with numerous different approaches and views on the subject (see related work of chapters 5 and 6). Two methods for analysing energy on source code level are proposed in chapters 5 and 6. These chapters focus on software that controls energy consuming hardware. The processor is not necessarily studies as a processor features many implicit behaviours affecting the energy consumption. Examples of control software that can be analysed with the methods in these chapters are control software for thermostats, industrial equipment, environment control systems, and household appliances, amongst others.

1.5 Outline

All **source code** created for this thesis is **available online** at the webpage listed below and through the QR-code on the right. The source code is released under open licenses, and like this thesis, available for all, free of charge.

`https://www.bitpowder.com/~bvgastel/phdthesis`

In chapter 2 a **correctness analysis** of a readers-writers mutual exclusion algorithm is described. This class of algorithms is used when **concurrency** at process level, or within a process using threads, is employed. The algorithm is used in the Qt library and therefore is present in many highly-distributed applications. The algorithm is first analysed using **model checking** in the Spin tool. Using interactive **theorem proving** with the PVS framework it is proved correct.

This chapter is based on [BvG-1] and based on [BvG-2]. The author found the initial problem in the Qt framework, analysed the algorithm using model checking, corrected the algorithm, and suggested the modifications to the makers of Qt.

In chapter 3 an analysis of hardware is described. A new technique to verify correctness of an essential part of a processor is explored in this chapter. The various parts of a processor are connected by a communication network, which increasingly contains more logic. This makes **correctness analysis** of hardware both harder and more important. For each entry point of the communication network, the kind of packets that are injected is specified. Using **symbolic execution** the flow of the packets is calculated, i.e. for each channel **type inference** is applied. As a result, for each channel in the network all the packets that can reach that channel are known. This is a prerequisite for **specification checking**, which validates if the network conforms to a specification that is given for each packet at each entry point of the network. A prototype written in C++ demonstrating the feasibility of this approach is available.

This chapter is based on [BvG-11]. The author came up with the initial idea, worked out the theoretical framework, and fully implemented the prototype. For this thesis the chapter is extended with respect to the publication with specification checking, a new case study and a more elaborate verification of the existing case study.

In chapter 4 a memory analysis is proposed. A symbolic bound is derived for both heap and stack memory usage using overapproximation. This analysis is implemented in the prototype ResAna, for the Java language. This prototype partly builds on the foundation of the prototype build in the COSTA project. This chapter resulted from the European Union CHARTER project, in which our part was to make memory analysis practical for real time Java programs.

This chapter is based on [BvG-4] and based on [BvG-10]. The author worked on the heap and stack analysis, also implementing it in the prototype. To make the specialisations for the JamaicaVM virtual machine, the author travelled to Karlsruhe (Germany) to talk to the creators of the virtual machine. During the validation of the heap memory analysis, the author discovered a modelling error of arrays in COSTA, and proposed a solution.

In chapter 5 a method is proposed to deduce an overapproximated symbolic energy consumption bound from programs written in the demonstration language ECA. This language has explicit hardware interaction embedded in the language, enabling easier analysis of the interactions. The overapproximation is based on a Hoare logic, which is applied on the source-code level. Due to the nature of overapproximations, several extra assumptions on the models are required. The approach is hardware parametric: it works for any external hardware device. Several implementations of hardware components can be easily exchanged in this energy analysis.

This chapter is based on [BvG-9]. The author took part in creating the idea, the Hoare logic, and the theoretical foundations. For this thesis, the notation is improved considerably, and a discussion on validity is added.

In chapter 6 an alternative method to analyse programs written in the ECA language is proposed, using program transformations. The program transformation is based on dependent types. The approach is also a hardware parametric energy analysis. The alternative approach has a number of advantages, among others several limitations on the modelling of components and the ECA language from chapter 5 do not apply. The Hoare logic approach suffers both from lack of compositionality and from significant overshoot in the derived bounds. This method derives a concrete precise result.

This chapter is based on [BvG-14]. The author thought of the idea, worked on the type system, and wrote the article. For this thesis, recursion is added to the ECA language.

DEADLOCK AND STARVATION FREE REENTRANT READERS-WRITERS

2

ABSTRACT *The classic readers-writers problem has been extensively studied. This holds to a lesser degree for the reentrant version, where it is allowed to nest locking actions. Such nesting is useful when a library is created with various procedures each starting and ending with a lock operation. Allowing nesting makes it possible for these procedures to call each other.*

We consider an existing widely used industrial implementation of the reentrant readers-writers problem. Staying close to the original code, we model and analyse it using the Spin model checker resulting in the detection of a serious error: a possible deadlock situation. The code and model was improved and checked satisfactorily for a fixed number of processes. To achieve a correctness result for an arbitrary number of processes the model is converted to a specification that is proven with the PVS theorem prover. Using model checking we found a starvation problem. We have also fixed the problem and checked the solution. Combining model checking with theorem proving appeared to be very effective in reducing the time of the verification process, by quickly getting acquainted with the problem, but also to check invariants needed for proving the theorems.

based on [BvG-1] based on [BvG-2]

correctness analysis **concurrency** **model checking** **theorem proving**

2

It is generally acknowledged that the historical growth in processor speed is reaching a hard physical limitation. This has led to a revival of interest in concurrent processing. Also in industrial software, concurrency is increasingly used to improve efficiency [Sut05]. It is notoriously hard to write correct concurrent software. Finding bugs in concurrent software and proving the correctness of (parts of) this software is therefore attracting more and more attention, in particular where the software is in the core of safety critical or industrial critical applications.

However, it can be incredibly difficult to track down concurrent software bugs. In concurrent software, bugs are typically caused by infrequent 'race conditions' that are hard to reproduce. Also the ability to encounter bugs depends heavily on the testing environment used. A kernel update can trigger race conditions previously not encountered. For concurrent software, it is necessary to thoroughly investigate 'suspicious' parts of the system in order to improve these components in such a way that correctness is guaranteed.

Three commonly used techniques for checking correctness of such systems are **testing**, **static (code) analysis** and **formal verification**. In practice, testing is widely and successfully used to discover faulty behaviour, but it cannot assure the absence of bugs. In particular,

for concurrent software testing is less suited due to the typical characteristics of the bugs (infrequent and hard to reproduce). In contrast with testing, static analysis is performed directly and fully automatically on the source code, without actually executing it. The information obtained from the analysis are, for example, common coding errors and suspicious control flow (e.g. leading to **null** pointer exceptions or lock order violations). There are roughly two approaches to formal verification: **model checking** and **theorem proving**. Model checking [CES83; QS82] has the advantage that it can be performed automatically, provided that a suitable model of the software (or hardware) component has been created. Furthermore, in the case a bug is found, model checking yields a counterexample scenario. A drawback of model checking is that it suffers from the state-space explosion problem and typically requires a closed system. In principle, theorem proving can handle any system. However, creating a proof may be hard and it generally requires a large investment of time. It is only partially automated and mainly driven by the user's understanding of the system. Besides, when theorem proving fails this does not necessarily imply that a bug is present. It may also be that the proof could not be found by the user.

We will consider the **reentrant readers-writers** problem as a formal verification case study. The classic readers-writers problem [CHP71] considers multiple processes that want to have read and/or write access to a common resource (a global variable or a shared object). The problem is to set up an access protocol such that no two writers are writing at the same time and no reader is accessing the common resource while a writer is accessing it. The classic problem is studied extensively [Pan+06]; the reentrant variant (in which locking can be nested) has received less attention so far although it is used in Java, C# and C++ libraries.

We have chosen a widely used industrial C++ library (Qt by Qt Group Plc, formerly Trolltech and Nokia) that provides methods for reentrant readers-writers. For this library a serious bug is revealed and fixed. This case study is performed in a structured manner combining the use of a model checker with the use of a theorem prover exploiting the advantages of these methods and avoiding their weaknesses. The main achievement of this approach is that it significantly improves the time effectiveness of the verification process itself.

This paper can be seen as an extended version of [BvG-1]. There are two main differences. Firstly, in this version we managed to keep the model much **closer to the code** using Promela and Spin in stead of Uppaal. The model contains more of the details present in the C++ program and it looks like the C++ program, but is still at approximately the

same abstraction level as the model in [BvG-1]. We have **manually** translated both the original C++ code into Spin models and the Spin models into PVS specifications. However, by keeping the model and the specification so close to the C++ code, we have shown that our approach lends itself for tool support, i.e. the used translations indicate ways of performing the conversion in a (semi) automatic way. Secondly, in this paper we also study **starvation**.

2

In section 2.1 we will introduce the abstract readers-writers problem. The studied Qt implementation is discussed in 2.2. Its model will be defined, improved and checked for a fixed number of processes in section 2.3. Using a theorem prover the model will be fully verified in section 2.4. Finally, related work, future work and concluding remarks are found in sections 2.5 and 2.6.

2.1 The readers-writers problem

If in a concurrent setting two threads are working on the same resource, synchronisation of operations is (often) necessary to avoid errors. A **test-and-set** operation is an important primitive for protecting common resources, commonly implemented in hardware. This atomic (i.e. non-interruptible) instruction is used to both test and (conditionally) write to a memory location. To ensure that only one thread is able to access a resource at a given time, these processes usually share a global boolean variable that is controlled via **test-and-set** operations, and if a process is currently performing a test-and-set, it is guaranteed that no other process may begin another test-and-set until the first process is done. This primitive operation can be used to implement **locks**. A lock has two operations: lock and unlock. The lock operation is done before the critical section is entered, and the unlock operation is performed after the critical section is left. However, implementing a lock with just an atomic test-and-set operation is impracticable. More realistic solutions will require support of the underlying operating system: threads acquiring a lock already occupied by some thread should be de-scheduled until the lock is released. A variant of this way of locking is called **condition locking**: a thread can wait until a certain condition is satisfied, and will automatically continue when **signalled** (notified) that the condition has been changed. An extension for both basic and condition locking is **reentrancy**, i.e. allowing nested lock operations by the same thread.

A so-called **read-write** lock functions differently from a normal lock: it either allows multiple threads to access the resource in a read-only way, or it allows one, and only one, thread at any given time to have full access (both read and write) to the resource [Goe+06]. These locks are used in databases and file systems. Several kinds of solutions to the classical readers-writers problem exist. In this chapter we will consider a **read-write** locking mechanism with the following properties:

writers preference Most solutions give priority to write locks over read locks because write locks are assumed to be more important, smaller, exclusive, and to occur less frequently. The main disadvantage of this choice is that it results in the possibility of reader starvation: when constantly there is a thread waiting to acquire a write lock, threads waiting for a read lock will never be able to proceed.

reentrant A thread can acquire the lock multiple times, even when the thread has not fully released the lock. Note that this property is important for modular programming: a function holding a lock can use other functions which possibly acquire the same lock. We distinguish two variants of reentrancy:

weakly reentrant only permit sequences of either read or write locks;

strongly reentrant permit a thread holding a write lock to acquire a read lock. This allows the following sequence of lock operations: `write_lock()`, `read_lock()`, `unlock()`, `unlock()`. Note that the same function is called to unlock both a write lock and a read lock. The sequence of a read lock followed by a write lock is not permitted because of the evident risk of a deadlock (e.g. when two threads both performs the locking sequence `read_lock()`, `write_lock()` they can both read but none of them can write).

2.2 Qt's implementation of readers-writers locks

In this section we show the C++ implementation of weakly reentrant read-write locks being part of the multi-threading library of the Qt development framework, version 4.3. The code is not complete; parts that are not relevant to this presentation are omitted. This implementation uses other parts of the library: threads, mutexes and conditions. Like for example in Java, a condition object allows a thread that owns the lock but that cannot

```
struct QReadWriteLockPrivate {
  QReadWriteLockPrivate()
  : accessCount(0),
    currentWriter(0),
    waitingReaders(0),
    waitingWriters(0)
  { }

  QMutex mutex;
  QWaitCondition readerWait,
                 writerWait;

  Qt::HANDLE currentWriter;
  int accessCount,waitingReaders,
      waitingWriters;
};

void QReadWriteLock::lockForRead() {
  QMutexLocker lock(&d->mutex);
  while (d->accessCount < 0 ||
         d->waitingWriters) {
    ++d->waitingReaders;
    d->readerWait.wait(&d->mutex);
    --d->waitingReaders;
   }
  ++d->accessCount;
  Q_ASSERT_X(d->accessCount>0,
        "...","...");
}

void QReadWriteLock::lockForWrite() {
  QMutexLocker lock(&d->mutex);
  Qt::HANDLE self =
     QThread::currentThreadId();
  while (d->accessCount != 0) {
    if (d->accessCount < 0 &&
        self == d->currentWriter) {
      break; // recursive write lock
    }
    ++d->waitingWriters;
    d->writerWait.wait(&d->mutex);
    --d->waitingWriters;
  }
  d->currentWriter = self;
  --d->accessCount;
  Q_ASSERT_X(d->accessCount<0,
        "...","...");
}

void QReadWriteLock::unlock() {
  QMutexLocker lock(&d->mutex);
  Q_ASSERT_X(d->accessCount!=0,
        "...","...");
  if ((d->accessCount > 0 &&
         --d->accessCount == 0) ||
      (d->accessCount < 0 &&
         ++d->accessCount == 0)) {
    d->currentWriter = 0;
    if (d->waitingWriters) {
      d->writerWait.wakeOne();
    } else if (d->waitingReaders) {
      d->readerWait.wakeAll();
    }
   }
}
```

listing 2.1 The QReadWriteLock class of Qt 4.3.

proceed, to wait until some condition is satisfied. When a running thread completes a task and determines that a waiting thread can now continue, it can call a signal on the corresponding condition. This mechanism is used in the C++ code listed in figure 2.1.

The structure QReadWriteLockPrivate contains the attributes of the class QReadWriteLock. These attributes are accessible via an indirection named `d`. The attributes `mutex` (of type QMutex), `readerWait` (of type QWaitCondition) and `writerWait` (of type QWaitCondition) are used to synchronise access to the other administrative attributes, of which accessCount keeps track of the number of locks acquired (including reentrant locks) for this lock. A negative value is used for write access and a positive value for read access. The attributes

`waitingReaders` and `waitingWriters` (both of type `int`) indicate the number of threads requesting a read respectively write permission, that are currently pending. If some thread owns the write lock, `currentWriter` contains a `HANDLE` to this thread; otherwise `currentWriter` is a `null` pointer.

The code itself is fairly straightforward. The locking of attribute `mutex`, of type `QMutex`, is performed via the constructor of the wrapper class `QMutexLocker`. Unlocking this mutex happens implicitly in the destructor of this wrapper. A write lock can only be obtained when the lock is completely released (`d->accessCount == 0`), or the thread already has obtained a write lock (`d->currentWriter == self`, a reentrant write lock).

The code could be polished a bit, e.g. one of the administrative attributes can be expressed in terms of the others. However, we have chosen not to deviate from the original code, except for the messages in the assertions which were, of course, more informative.

2.3 Model checking readers-writers with Spin

Spin is an explicit state model checker with support for assertions and Linear Temporal Logic (LTL), including liveness properties. Spin converts a model written in the specification language Promela to a checker written in C. By compiling and running the checker, properties can be checked, e.g. see [Hol04; Ben08].

In the previous version of this paper [BvG-1] we used Uppaal for modelling the system. An advantage of Uppaal is its intuitive and easy to use graphical interface. However, we have decided to switch to Spin for mainly two reasons: First, the input language Promela resembles C, which allows us to model the code in a direct and clear way. Second, compiled models generated by Spin appear to be more efficient than equivalent models specified in Uppaal. This enables us to enlarge the examined state space of the model significantly.

A few general notes can be made about modelling code in Promela. Promela is not a (general-purpose) programming language, and therefore it lacks some features that are found in common language like C or Java. For instance, there are no functions that return values in Promela. For simple non-recursive procedures, one can use the `inline` construct instead. Moreover, Promela does not support object oriented programming. In our translation, we will will represent the attributes of objects as structs, and non-static methods as (inline) functions, having `this` as en explicit argument.

A feature of Spin is the ability to embed C code directly. With a couple of special Promela statements C code can be inserted in the model and is executed atomically in the model. Spin tracks the memory used by these statements and include the memory regions in the state space. One can easily convert source code to a Promela model by wrapping all C code in the proper Promela statements. This method is not applicable to our case study: the mutexes are system calls which modify memory outside the process space. The content of these (kernel) memory regions can not be rolled back by Spin as the state space is explored. So we have to model the whole program in Promela.

2.3.1 Modelling the basics

The Qt implementation of the QReadWriteLock class is based on two other classes: QMutex and QWaitCondition. These components are platform dependent. In our case study we use the Linux version, in which QMutex and QWaitCondition are built on the pthread_mutex and pthread_cond components of the POSIX thread library. This library is part of the operating system. Creating a code based model of these components would require the treatment of operating system dependent details making the whole system too complex. Instead we will use abstract versions of these components.

When using the 2.6 version of the Linux kernel, the default behaviour for POSIX components is not starvation free. Starvation free behaviour of these components can be activated by setting the SCHED_FIFO flag when creating threads. Qt, however, uses the default behaviour. This is, of course, an important observation when we are considering the absence of starvation of the locking mechanism. In that case we will assume that the threads are scheduled fairly and that the underlying basic locking primitives use a first-in-first-out (FIFO) lock assignment strategy, see section 2.3.6. However, below we study the default behaviour of the POSIX components first.

```
1  typedef pthread_mutex_t {
2    bool locked = false
3  };
4
5  inline pthread_mutex_unlock(this) {
6      assert(this.locked);
7      this.locked = false;
8  }
10 inline pthread_mutex_lock(this) {
11   atomic {
12     !this.locked;
13     this.locked = true;
14   }
15 }
```

listing 2.2 Abstract model in Promela of the non-reentrant pthread_mutex.

We start with modelling the basic pthread_mutex component. The two main functions operating on this component are `pthread_mutex_lock()` and `pthread_mutex_unlock()`, which both can be specified easily in Promela; see figure 2.2. The lock itself is represented as a single boolean, named `locked` and initially set to false. The `pthread_mutex_lock()` function is an atomic operation that waits until `locked` is false before setting the variable to true. Waiting can be expressed in Promela just be using boolean expressions as statements. If, during the execution of the model such a statement is encountered, the corresponding computation branch is suspended until the expression becomes true. The `pthread_mutex_unlock()` function resets `locked` to false. To check for incorrect use, an assertion is added to the code verifying that no lock is released if it has not been obtained before. By wrapping the `locked` variable in a typedef (named pthread_mutex_t) we can use this pthread_mutex component in the same manner as in the C++ code.

We now model the pthread_cond component, listed in figure 2.3. This component allows a thread owning the lock to wait until some condition is satisfied (while releasing the lock). When another running thread completes a task and determines that a waiting thread can now continue, it can wake up this thread by calling a signal on the corresponding condition. Actually, two kinds of signal functions are available working on pthread_cond: `pthread_cond_signal()` (waking one thread) and `pthread_cond_broadcast()` (waking all threads). Our abstract version of pthread_cond uses a basic synchronisation mechanism of Promela: (synchronous) rendez-vous channels. The `pthread_cond_wait()` func-

```
typedef pthread_cond_t {
  byte waiting = 0;
  chan cont = [0] of {bit};
};

inline pthread_cond_signal(this) {
  atomic {
    if
    :: this.waiting > 0 ->
          this.waiting--;
          this.cont?_;
    :: else
    fi;
  }
}

inline pthread_cond_broadcast(this) {
  atomic {
    do
    :: this.waiting > 0 ->
          this.waiting--;
          this.cont?_;
    :: else -> break;
    od;
  }
}

inline pthread_cond_wait(this,mutex) {
  this.waiting++;
  pthread_mutex_unlock(mutex);
  this.cont!1;
  pthread_mutex_lock(mutex);
}
```

listing 2.3 Abstract model in Promela of pthread_cond.

```
typedef QWaitCondition {
   pthread_mutex_t mutex;
   pthread_cond_t cond;
   int waiters = 0;
   int wakeups = 0;
};

inline QWaitCondition_wakeOne(this) {
   pthread_mutex_lock(this.mutex);
   this.wakeups = min(this.wakeups + 1,
                      this.waiters);
   pthread_cond_signal(this.cond);
   pthread_mutex_unlock(this.mutex);
}

inline QWaitCondition_wakeAll(this) {
   pthread_mutex_lock(this.mutex);
   this.wakeups = this.waiters;
   pthread_cond_broadcast(this.cond);
   pthread_mutex_unlock(this.mutex);
}
```

```
inline QWaitCondition_wait(this, m) {
   pthread_mutex_lock(this.mutex);
   this.waiters++;
   QMutex_unlock(m);
   do
   :: this.wakeups == 0 ->
      pthread_cond_wait(this.cond,
                        this.mutex);
   :: else ->
      break;
   od;
   this.waiters--;
   this.wakeups--;
   pthread_mutex_unlock(this.mutex);
   QMutex_lock(m);
}
```

listing 2.4 Concrete model in Promela of QWaitCondition.

tion uses a send operation on the rendez-vous channel `cont`. The thread invoking this method will be blocked until another thread execute a receive operation. The contents of the message sent over this channel are irrelevant, only the timing of the message counts. On the receiver side this is specified by using an anonymous write-only variable (in Promela: `cont?_`), and on the sender side by choosing some arbitrary value (in our case the value 1, sent with the statement `cont!1`). Before waiting on the channel the wait function has to unlock the mutex and, after continuing, to lock the mutex again. To be able to wake all the waiting threads, the condition keeps track of the number of waiting threads in the field `waiting`. For correctness atomic blocks are used to limit the interleaving of processes (otherwise the tests `waiting > 0` and `waiting--` could be interrupted). Just like pthread_mutex the variables are wrapped in a new type pthread_cond_t.

The implementation of QMutex class appears to be rather complex, due to some optimisations that have been performed. As a consequence, the code base is large and it is outside the scope of this article, to model this part faithfully. Instead we will use pthread_mutex to provide the locking mechanism, because it has the same functional behaviour as QMutex. Hence QMutex is a wrapper around around pthread_mutex. The implementation of QWaitCondition, on the other hand, is much shorter, and can therefore be converted to Promela straightforwardly. The result is listed in figure 2.4. Again, the

attributes of this class are wrapped in a struct. As one can see, the class depends on pthread_mutex, pthread_cond (appearing as attribute types), and on QMutex (passed as an argument to the method QWaitCondition_wait()). According to the comments in the source code 'many vendors warn of spurious wake-ups from pthread_cond_wait(), especially after signal delivery'. Both the attribute wakeups and the loop in wait() method are used to counter the described spurious wake-ups. In this way, a thread can only finish the wait() method if a signal is received. The variable wakeups is used to keep track of the number of threads allowed to wake up and is bound by the number of waiting threads, as contained in the attribute waiters. Both the wakeOne() and the wakeAll() methods increase the wakeups attribute, and the wait() method decreases the variable as threads are woken. The pthread_mutex used in QWaitCondition is needed because QMutex does not use a pthread_mutex, and such a mutex is needed for the pthread_cond_wait() function. The parameter m of the wait() method is a mutex. This mutex is released until a signal is received.

2.3.2 Modelling readers-writers

Now we have modelled all the components on which the QReadWriteLock class depends, we can convert the QReadWriteLock itself to Promela. All class attributes can be expressed directly (the type Qt::HANDLE is converted to the Promela type pid, both identifying a specific process or thread). In figure 2.5 the variables of the class and the code of lockForRead() are listed, on the left the original C++ code, and on the right the conversion in Promela. Methods are converted to inline definitions.

The QMutexLocker is a convenience wrapper around a lock, obtaining a lock when the object is constructed and releasing the lock implicitly (via its destructor) when the object is deallocated. When used as a local (stack) object, QMutexLocker obtains the lock during its initialisation and releases the lock when this local object gets out of scope. This implicit destructor invocation is converted to an explicit call of QMutexUnlock().

The translation of the code for the lockForRead() method is performed instruction-wise. A while-loop is converted into a do ... od statement (which can be thought of as for in C++). The loop is ended with a break in one of the **condition blocks** (prefixed by a double colon, ::). Normally, a block with a true condition is chosen non-deterministically for execution, though in our case only one of these conditions can possibly hold at a given time.

```
struct QReadWriteLockPrivate {
   QMutex mutex;
   QWaitCondition readerWait,
                  writerWait;
   Qt::HANDLE currentWriter;
   int accessCount,
     waitingReaders,
     waitingWriters;
};

void QReadWriteLock::lockForRead() {
  QMutexLocker lock(&d->mutex);
  while (d->accessCount < 0 ||
         d->waitingWriters) {
      ++d->waitingReaders;
      d->readerWait.wait(&d->mutex);
      --d->waitingReaders;
   }
  ++d->accessCount;
  Q_ASSERT_X(d->accessCount > 0,
                     "...", "...");
}
```

```
typedef QReadWriteLock {
   QMutex mutex;
   QWaitCondition readerWait;
   QWaitCondition writerWait;
   pid currentWriter = NT;
   int accessCount = 0;
   int waitingReaders = 0;
   int waitingWriters = 0;
};

inline QReadWriteLock_lockForRead(this) {
  QMutex_lock(this.mutex);
  do
  :: this.accessCount < 0 ||
        this.waitingWriters > 0 ->
      this.waitingReaders++;
      QWaitCondition_wait(this.readerWait,
                          this.mutex);
      this.waitingReaders--;
  :: else -> break;
  od;
  this.accessCount = this.accessCount + 1;
  assert(this.accessCount > 0);
  QMutex_unlock(this.mutex);
}
```

listing 2.5 Part of QReadWriteLock (Qt 4.3 version) in C++ (left) and Promela (right).

2.3.3 Modelling usage of the lock

In order to check properties we will simulate all possible usages of the QReadWriteLock. For this reason we will define a number of threads, each (sequentially) executing a finite number of read and/or write locks, and matching unlocks, in a proper sequence (i.e. no unlocks if the lock is not obtained first by the thread and no write lock requests if the thread already has a read lock). Eventually each thread relinquishes all locks, so other threads are allowed to proceed. The variable maxLocks indicates how many locks a thread may request before it relinquishes all locks. We model these threads by Promela processes as shown in figure 2.6. Here, THREADS indicates the number of threads the model is checked with. Note that the do statement chooses one of the options non-deterministically. The readNest variable is used to exclude the case in which a (reentrant) write lock is performed after a read lock is already obtained. Both readNest and writeNest are used to control unlocking. Both are updated in the 'methods' of QReadWriteLock. As the 'methods' are in fact just inlined code, they can access and update these variables.

```
active[THREADS] proctype user() {
   int readNest = 0;
   int writeNest = 0;
   int maxLocks;
   do
   :: maxLocks = MAXLOCKS;
      do
      :: maxLocks > 0 ->
         maxLocks--;
         if
         :: readNest == 0 -> QReadWriteLock_lockForWrite(rwlock);
         :: QReadWriteLock_lockForRead(rwlock);
         fi;
      :: writeNest + readNest > 0 ->
         QReadWriteLock_unlock(rwlock);
      :: maxLocks != MAXLOCKS && writeNest + readNest == 0 ->
         break;
      od;
   od;
}
```

listing 2.6 Promela process of QReadWriteLock usage.

There are three kinds of properties to be checked, each invoked differently by Spin. The absence of deadlock property is checked implicitly when running the verifier for assertion violations. Each time a non-end state is encountered and no transitions out of the state are valid an 'invalid end state' error is reported. The second type of properties we check are safety properties, which are valid in each state of the model (specified as LTL formulas beginning with the G operator). Most of the informal correctness properties specified in section 2.1 are of this type. The last type are liveness properties, guaranteeing that each process can make progress of some sort. Spin has special support for liveness properties, called **progress states**, but they can also be checked with LTL properties. We continue with checking for deadlock and assertions.

2.3.4 Checking for deadlock and assertions

As stated before, deadlock detection is done implicitly when checking for assertions. Each state not marked as an end state and with no outgoing transitions is reported. Also all assertions in the model are checked. Besides the assertions that were present in the original code, there is one assertion in the method `lockForWrite()` that has been added, to verify that no thread gets write access when readers are busy.

```
pan: invalid end state (at depth 188)
pan: wrote qreadwritelock43.usage.trail
...
pan: reducing search depth to 32
...
   0: enter lockForRead
   0: leave lockForRead
      1: enter lockForWrite
      1: waiting
   0: enter lockForRead
   0: waiting
spin: trail ends after 34 steps
#processes: 2
   rwlock.mutex.m.lockedBy = 255
   rwlock.mutex.m.count = 0
   rwlock.readerWait.waiters = 1
   rwlock.readerWait.wakeups = 0
   rwlock.readerWait.waiting = 1
   rwlock.writerWait.waiters = 1
   rwlock.writerWait.wakeups = 0
   rwlock.writerWait.waiting = 1
   rwlock.accessCount = 1
   rwlock.currentWriter = 255
   rwlock.waitingReaders = 1
   rwlock.waitingWriters = 1
   readers = 1
   writers = 0
 34:    proc  0 (user) line  19 "qwaitcondition.abs" (state 29)
 34:    proc  1 (user) line  19 "qwaitcondition.abs" (state 187)
```

listing 2.7 Output of Spin when checking for a deadlock.

Running our model resulted immediately in the detection of a deadlock. The output of Spin is given in figure 2.7. It starts with an iterative search for the shortest error trail. After that the debug output of the shortest trail is printed. The values of all variables in the last state are showed, and the output ends with a message in which state the processes are. The situation reported by Spin occurs when a thread already having a read lock requests another one, while another thread is waiting for a write lock. The deadlock is clear: the first thread is never going to proceed with the reentrant reader because there is a writer waiting. The second thread is never going to proceed because the lock is never released. A change to the algorithm is needed to avoid this deadlock.

The solution to the deadlock stated above is to let a reentrant lock always proceed. To check if a lock request is a reentrant operation, for each thread the number of calls to the specific lock should be kept track of. If this number is positive the lock operation should always succeed. An extra attribute count of type QHash⟨Qt::HANDLE, int⟩ is introduced in

the original C++ code, mapping thread identifiers to numbers. In our translated model we represented this hash table by an integer array `count` in which `count[pid]` is the number of reentrant locks of process `pid`. In Promela the array is declared with the statement `int count[THREADS]`.

Furthermore, we take this opportunity to change the strange use of the `accessCount` variable: the sign of the value of accessCount indicates whether active locks are read locks or write locks. This distinction between readers and writers appears to be superfluous. In fact, leaving out this distinction provides that our implementation is strongly reentrant. Moreover, we changed the name of the variable into `threadCount` to indicate it actually contains the number of different threads that are currently holding the lock.

After the adjustments to the model, Spin reports no assertion violations and no invalid end states for a parameterised model with three threads and a maximum of five locking operations. So the model is shown to be free of deadlocks with these parameters.

We reported the deadlock to Qt Group Plc. Recently, Qt Group Plc released a new version of the thread library (version 4.4) in which the deadlock was repaired. However, the new version of the Qt library is still only weakly reentrant, not admitting threads that have write access to do a read lock. This limitation unnecessarily hampers modular programming.

2.3.5 Checking LTL safety properties

To check the properties we introduce auxiliary variables in the model to track the number of threads having write locks (called `writers`) and having read locks (called `readers`). The code needed to keep track of these auxiliary variables is inserted at appropriates place in the 'methods' of `QReadWriteLock`. The `readers` and `writers` variables are only incremented on a non-reentrant call of a thread, and therefore decremented only on the final unlock. The other variables stated in the properties are attributes of `QReadWriteLock`.

We now continue with checking LTL safety properties of the algorithm. These properties are checked by querying Spin with a LTL expression. We removed a deadlock in the previous subsection, but the algorithm was not checked for conceptually flawed behaviour, for example allowing both a reader and a writer enter the critical section at the same time.

```
typedef QReadWriteLock {
  QMutex mutex;
  QWaitCondition readerWait;
  QWaitCondition writerWait;

  int threadCount = 0;
  int waitingReaders = 0;
  int waitingWriters = 0;

  pid currentWriter = NT;
  int count[THREADS] = 0;
}

inline QReadWriteLock_lockForRead(this) {
  QMutex_lock(this.mutex);
  if
  :: this.count[_pid] == 0 ->
     do
     :: (this.currentWriter != NT ||
         this.waitingWriters > 0) ->
        this.waitingReaders++;
        QWaitCondition_wait
            (this.readerWait,this.mutex);
        this.waitingReaders--;
     :: else -> break;
     od;
     this.threadCount++;
     assert(this.waitingWriters == 0);
  :: else
  fi;
  this.count[_pid]++;
  ... update model variables ...
  QMutex_unlock(this.mutex);
}

inline QReadWriteLock_lockForWrite(this) {
  QMutex_lock(this.mutex);
  if
  :: this.currentWriter != _pid ->
     do
     :: this.threadCount != 0 ->
        this.waitingWriters++;
        QWaitCondition_wait
            (this.writerWait,this.mutex);
        this.waitingWriters--;
     :: else -> break;
     od;
     this.currentWriter = _pid;
     this.threadCount++;
  :: else
  fi;
  assert(this.threadCount == 1 &&
         this.currentWriter == _pid);
  this.count[_pid]++;
   ... update model variables ...
  QMutex_unlock(this.mutex);
}

inline QReadWriteLock_unlock(this) {
  QMutex_lock(this.mutex);
  this.count[_pid]--;
  // is it the last unlock by this thread?
  if
  :: this.count[_pid] == 0 ->
     this.threadCount--;
     // is it the last unlock of the lock?
     if
     :: this.threadCount == 0 ->
        this.currentWriter = NT;
        if
        // if available wake one writer,
        :: this.waitingWriters > 0 ->
           QWaitCondition_wakeOne
                       (this.writerWait);
        // otherwise wake all readers
        :: else ->
           if
           :: this.waitingReaders > 0 ->
              QWaitCondition_wakeAll
                       (this.readerWait);
           :: else
           fi;
        fi;
     :: else
     fi;
  :: else
  fi;
  ... update model variables ...
  QMutex_unlock(this.mutex);
}
```

listing 2.8 Updated Promela model of readers-writers algorithm.

A predicate called outsideCS is introduced, indicating that **no** change can occur inside the lock structure. In other words no thread has locked the mutex, as indicated by the negation of the boolean attribute locked from attribute mutex fromQReadWriteLock.

Formalisation of the properties stated in section 2.1 is now straightforward. The resulting invariants are listed below. The waitingReaders and waitingWriters variables used are attributes of the QReadWriteLock object.

$G\,(\texttt{readers} = 0 \;\vee\; \texttt{writers} = 0)$

There are not simultaneously writers and readers allowed.

$G\,(\texttt{writers} \leq 1)$

No more than one writer is allowed.

$G\,(\texttt{outsideCS} \rightarrow (\texttt{waitingWriters} > 0 \rightarrow (\texttt{readers} > 0 \;\vee\; \texttt{writers} > 0)))$

States that the only possibility of waiting writers is when there are readers or writers busy, but only when there is no change to the lock.

$G\,(\texttt{outsideCS} \rightarrow (\texttt{waitingReaders} > 0 \rightarrow (\texttt{writers} > 0 \;\vee\; \texttt{waitingWriters} > 0)))$

States that the only possibility of waiting readers is when there are writers waiting or writers busy, but only when there is no change to the lock.

The third and fourth invariant do not hold for this algorithm. We detected this issue during model checking. There exists a state in which the proposition outsideCS is true, there are no readers and no writers, but there are readers and/or writers waiting. The third and fourth stated safety property are therefore violated. This occurs if a thread has just called the unlock() method, and another thread intends to continue with acquiring a read or a write lock. The invariants are not easily fixed, as these states can not be easily excluded. In the next subsection, a change is proposed to avoid starvation. This change also avoids the state mentioned above. Therefore we postpone verifying these invariants to the next subsection.

2.3.6 Checking for absence of starvation

We continue with ensuring the absence of starvation in the algorithm. In section 2.1 we stated that the design decision to give preference to writers results in a possible reader starvation. Therefore it only makes sense to check the property for writers. In Spin one

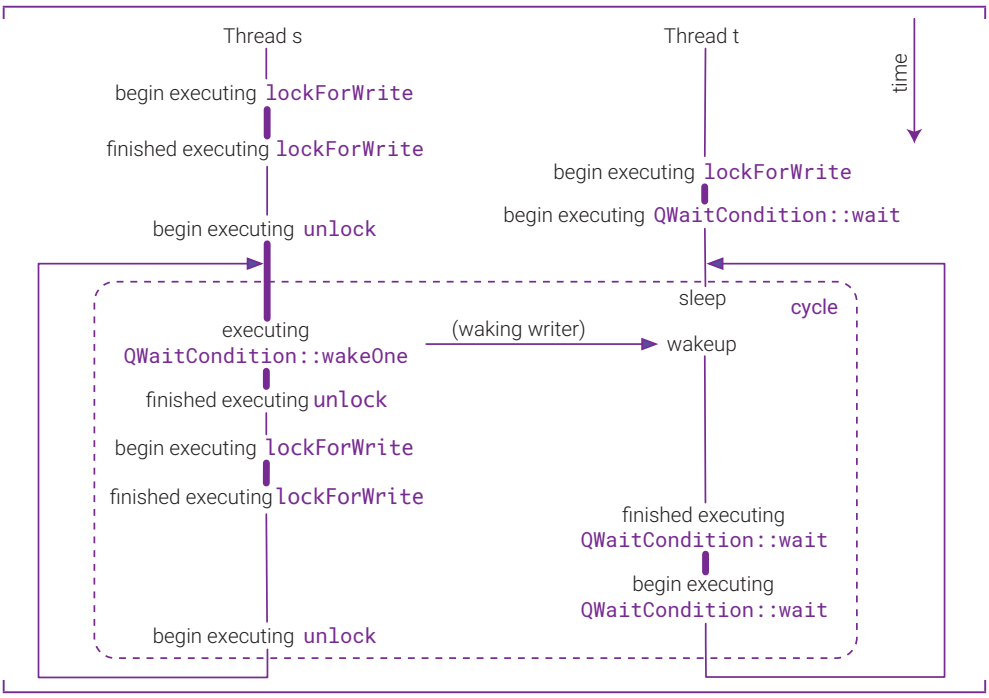

figure 2.9 Graphical representation of the counterexample indicating a starvation problem. The thick line indicates that the mutex is hold by a thread.

can verify starvation properties by using **progress states**. A looping process obtaining and releasing write locks, but no read locks, is added and labelled with a **progress label**. When checking the model, it is verified that all execution cycles (i.e. an execution path on which the same state occurs twice) contain this progress label.

As noted earlier, the original readers-writers algorithm has a starvation problem because Qt uses the default behaviour of POSIX on Linux. However, we continue as if a fair scheduling policy would have been used. To avoid starvation in the underlying `pthread_mutex` and `pthread_cond` models, these were replaced by starvation free versions that use a FIFO mechanism. Despite of these changes, the model still contains the possibility of writers starvation. This appeared when we checked the model for absence of progress, and Spin found an execution cycle with no progress states. A graphic representation of this cycle is shown in figure 2.9. The problem is caused by the `wait()` method of `QWaitCondition`; see figure 2.4. When thread t calls `QWaitCondition_wait()`, it will suspend execution (by calling `pthread_cond_wait()`) until thread s signals that thread t can continue its execu-

tion. However, at that time, thread t has no longer locked the mutex `this.mutex`. Each other thread, in the figure thread s, can now lock this mutex (by calling `lockForWrite()`) just before t does, effectively stealing the turn of t.

This problem can be avoided by ensuring that no thread can get the mutex before the signalled thread (t in the above example) can start executing again. This can be done by atomically transferring the lock on the mutex from the signalling thread to the signalled thread. Also, all stated invariants are valid, as the states mentioned in the previous subsection do not exists anymore because of the atomic transfer of the lock between threads. To accommodate this behaviour we have adjusted the QWaitCondition and QMutex parts of our Spin model. Although we were able to find a solution, the solution is rather large and complex. The solution also includes a way to create a starvation-free condition variables out of one starvation-free mutex and two starvation-prone condition variables. This is needed because starvation-free condition variables are not available on most POSIX platforms, including Linux, Mac OS X, and FreeBSD. For brevity, we will not present the adjusted Spin model, but take the improvement into account in the next section. For the complete solution and a more extensive report of our experiments, see [BvG-0]. The adjusted version is verified free of deadlock and starvation and not violating the safety properties, for a model with three threads with a maximum of four lock operations (actually we were able to verify the model free of starvation for a maximum of six reentrant lock operations, but not the other properties).

2.3.7 Results

In these experiments we have verified absence of deadlock and starvation and a number of safety properties for a maximum of three threads, and for a maximum of four lock operations. Although the absence of starvation was verified for six lock operations, the safety properties and absence of deadlock were only verified for four lock operations. For these parameters, the experiments runs in about four hours (94 minutes for deadlock checking, and 35 minutes for starvation, and 128 minutes for the safety properties), using 127.6 gigabytes of memory. If we increase these values slightly, the execution time worsens drastically and/or the memory usage increases above 128 GiB, the memory limit for our machines. For a complete correctness result, we have to proceed differently.

2.4 Generalised reentrant readers-writers model

In this section we will formalise the Spin model in PVS [ORS92]. We prove that the reentrant algorithm is free from deadlock and writer starvation when we generalise to **any** number of processes. We use the PVS interactive theorem prover for the generalised model.

The PVS system consists of a specification language and a theorem prover. The specification language of PVS is based on classical, typed higher-order logic. It resembles common functional programming languages, such as Haskell, LISP or ML. The choice of PVS as the theorem prover to model the readers writers locking algorithm is purely based upon the presence of local expertise. The proof can be reconstructed in any reasonably modern theorem prover, for instance Isabelle [NPW02] or Coq [BC04].

The earlier translation of an Uppaal model of the algorithm to PVS [BvG-1] was specific to that particular model. In order to derive the PVS specification from the Spin model we use a more methodical approach, suitable for other models as well. Furthermore, this methodical approach offers more opportunities for tool support.

2.4.1 Readers-writers model in PVS

There is no implicit notion of state or processes in PVS specifications. So, we construct a state transition system that explicitly keeps track of a system state. This state consists of the global variables of the Spin model, thread information, and a variable indicating which thread is currently active. For each thread a program counter and the state of the local variables are also part of the global transition system. Moreover, whether a thread is allowed to be scheduled is kept by means of a `ThreadState`. When it is `Running` the scheduler will allow the thread to progress. However, when it is `Sleeping`, it will not be permitted to run until woken up. A thread can have an `atomic` flag set. This flag tells the scheduler that only this thread can be executed. The `atomic` flag is set whenever the atomic primitive is used in Spin and is reset when the atomic block ends. Each critical section in the Spin model starts with a `QMutex_lock()` and ends with a `QMutex_unlock()` (see figure 2.8). These method calls enforce mutual exclusion of access to all the global variables in the Spin model. We abstract away from these method calls by setting the `atomic` flag when a thread enters its critical section and resetting the flag once it leaves the critical section. This is semantically the same as using the mutual exclusion mechanism, because threads use only local variables outside of their critical sections.

A thread can transfer its atomic status to another one, say with the ThreadID tid, by setting the field to tid. Only tid will be able to be scheduled next. With NT denoting the total number of processes, we get a general representation of threads, listed below. The kind of local and global variables that are used is left open. These can be instantiated for each particular Spin model with a model specific collection of local and global variables.

```
Threads[NT:nat, PC:TYPE, LV:TYPE, GV:TYPE] : THEORY
BEGIN
  ThreadID : TYPE = below(NT)
  ThreadStateType : TYPE = { Running, Waiting, Terminated }
  ThreadState : TYPE = [# state : ThreadStateType
                       , local : LV
                       , PC : PC
                       , atomic : boolean #]
  Threads : TYPE = ARRAY[ ThreadID → ThreadState]
  System : TYPE = [# threads : Threads
                   , currentTID, transfer : ThreadID
                   , global : GV #]
END Threads
```

The predicate interleave simulates parallel execution of threads, see below. A thread is only allowed to switch its context when it is not atomic or when the lock is transferred from one thread to another. With isNull is tested whether transfer contains a valid ThreadID. This thread becomes the next current thread. Only Running threads are scheduled. The predicate only holds for a subset of the System data type, signified by the validState? predicate, further explained in section 2.4.3.

```
interleave(s1,s3:(validState?)) : boolean =
∃ (s2:System) : chain_atomic(s1,s2)
 ∧ IF isNull(s2‘transfer)
   THEN ∃ (tid:ThreadID) : s3 = s2 WITH [ ‘currentTID := tid ]
                         ∧ s3‘threads(tid)‘state = Running
   ELSE s3 = s2 WITH [ ‘currentTID := s2‘transfer
                     , ‘transfer := NT ]
   ENDIF
```

Before possibly switching its context, the current thread performs a series of execution steps using the chain_atomic relationship. It is assumed that a next relation is provided, representing a single step in the execution of a thread. The non-deterministic choice which thread gets to execute is modeled by the existential quantifier that states that any thread can become the next current thread, unless there is an explicit lock transfer.

A single step, as described by the next relation, is atomic by definition. A sequence of such steps is executed recursively until the thread has released its atomic flag. The predicate for this is listed below. This recursive relationship terminates because there are no cycles in the progression of states a thread can transfer to with its atomic flag set.

```
chain_atomic(s1:System, s2:System) : RECURSIVE boolean =
    ¬s2‘threads(s2‘currentTID)‘atomic
∧ (next(s1,s2)∨ ∃ (s:System): next(s1,s)∧ s1‘currentTID = s‘currentTID
                    ∧ s1‘threads(s1‘currentTID)‘atomic
                    ∧ s1‘threads(s1‘currentTID)‘atomic = s‘threads(s‘currentTID)‘atomic
                    ∧ chain_atomic(s,s2))
MEASURE s1 BY state_order
```

The PVS specification used here is semantically slightly different from the one used in Qt. This model not only wakes up a process, but also passes the lock on with the transfer field to one of the woken threads to avoid writer starvation, mentioned as a solution to the starvation problem at the end of section 2.3.6. Note that this is only possible if a thread immediately leaves its critical section after synchronisation. The model is based on a FIFO queue that holds all processes, such that they will be woken in the order that they have been put to sleep.

```
QWaitCondition : TYPE = list[ThreadID]
NEQWaitCondition : TYPE = {wc:QWaitCondition | length(wc) > 0 }
wait(s:System, q:QWaitCondition) : [System, QWaitCondition] =
  (s WITH [ ‘threads(s‘currentTID)‘state := Waiting
          , ‘threads(s‘currentTID)‘atomic := false ]
          , append(q, cons(s‘currentTID, null)))
wakeOne(s:System, q:NEQWaitCondition) : [System, QWaitCondition] =
  (s WITH [ ‘threads(car(q))‘state := Running
          , ‘threads(s‘currentTID)‘atomic := false
          , ‘threads(car(q))‘atomic := true
          , ‘transfer := car(q)], cdr(q))
wakeAll(s:System, q:NEQWaitCondition) : [System, QWaitCondition] =
  LET newthreads = λ (p:ThreadID) : s‘threads(p)
                   WITH [ state := IF member(p,q)
                                   THEN Running
                                   ELSE s‘threads(p)‘state
                                   ENDIF ] IN
  (s WITH [ ‘threads := newthreads
          , ‘threads(s‘currentTID)‘atomic := false
          , ‘threads(car(q))‘atomic := true
          , ‘transfer := car(q)], null)
```

listing 2.10 PVS model for QWaitCondition.

The type QWaitCondition, as listed in listing 2.10, is a list that holds the ThreadID s of all threads that are put to sleep. The wait function takes a wait queue and changes the state of the current thread to Waiting and releases the atomic flag. The wakeOne and wakeAll functions are used to wake up one waiting writer and all waiting readers respectively. Their states are set to Running, so they can be scheduled and the lock is transferred to the process that is first in the queue.

2.4.2 Translation from Spin to PVS

After having defined all the components, the total state of the model is defined by all the local and global variables. These are exactly the same as in the original Spin model as defined in figure 2.8. The ProgramCounterStates refer to the locations of the program counter as the Spin model executes. For instance, the start of the outer do loop in the user() function defined in figure 2.6 contributes user05 to ProgramCounterStates. ProgramCounterStates, defined below, instantiates PC in the theory Threads and similarly, both LocalVariables and GlobalVariables instantiate LV and GV respectively.

```
1  ProgramCounterStates : TYPE = { lockForRead17, ... , user05 }
2  LocalVariables : TYPE = [# readNest, writeNest, maxLocks : nat #]
3  QReadWriteLock : TYPE =
4       [# readerWait, writerWait : QWaitCondition
5        , count : [ThreadID → nat]
6        , currentWriter : ThreadID
7        , threadCount, waitingReaders, waitingWriters : nat #]
8  GlobalVariables : TYPE =
9       [# readers, writers : nat, rwlock : QReadWriteLock #]
```

The relation next(s1,s2 : System) : boolean specifies the global state transitions, which can be found in listing 2.11. The body of this function is derived directly from the Spin model using the following method:

- at each position where there can be a context switch in the Spin model, there is a location added to the program counter type.
- control structures like do are translated by setting the program counter to the appropriate location. Location labels are derived from the function names, appended with the line numbers in the Spin source code.
- assignments are translated to modifications of the variables in the state.

```
next(s1:System, s2:System) : boolean =
[ .. removed some code for brevity .. ]
CASES s1`threads(currentTID)`PC OF
user05:
  s2 = s1 WITH [ `threads(currentTID)`local`maxLocks := MAXLOCKS
               , `threads(currentTID)`PC := user07 ],
user07:
  ( IF maxLocks > 0
    THEN s2 = s1 WITH [ `threads(currentTID)`local`maxLocks := maxLocks-1
                      , `threads(currentTID)`PC := user10]
    ELSE FALSE ENDIF
  ∨ IF writeNest + readNest > 0
    THEN s2 = s1 WITH [ `threads(currentTID)`PC := unlock67]
    ELSE FALSE ENDIF
  ∨ IF writeNest + readNest = 0
    THEN s2 = s1 WITH [ `threads(currentTID)`PC := user05 ]
    ELSE FALSE ENDIF ),
user10:
  ( IF readNest = 0
    THEN s2 = s1 WITH [ `threads(currentTID)`PC := lockForWrite42]
    ELSE FALSE ENDIF
  ∨ s2 = s1 WITH [ `threads(currentTID)`PC := lockForRead17] ),
 [ .. transition relation continues with cases for lockForRead, etc .. ]
```

listing 2.11 PVS model of the user function.

- non-deterministic choices are modelled as disjunctions in the transition relation. There is one disjunct for each non-deterministic choice.
- function calls are done by setting the program counter to the location of the function. Since no function is called from more than one location, using a return address or even using a stack for more than one return address has been omitted.

The auxiliary variables `readNest`, `writeNest` and `MAXLOCKS` restrict the Spin model to a maximum number of nested reads and writes. They also prevent unwanted sequences of lock/unlock operations, e.g. when a write lock request occurs after a read lock has already been obtained. This `user()` function from figure 2.6 is directly coded in the state transition model, where each label corresponds to the line number in the original.

As an example we provide the transition model derived from the Spin code in figure 2.8 for the `lockForRead()` function by using the rules specified earlier, shown in listing 2.12. After obtaining a read lock, the variable `readNest` is increased, corresponding with the code that updates the model variables in the original Spin model. The transition model starts out with all threads in a `Running` state and with the local variables at their initial values. Also, the global variables are all initialised and all queues are empty.

```
CASES s1`threads(currentTID)`PC OF
  [ .. removed some code for brevity .. ]
lockForRead17:
  s2 = s1 WITH [ `threads(currentTID)`atomic := true % QMutexLock
               , `threads(currentTID)`PC := lockForRead18 ],
lockForRead18:
  IF count(currentTID) = 0
  THEN s2 = s1 WITH [ `threads(currentTID)`PC := lockForRead20 ]
  ELSE s2 = s1 WITH [ `threads(currentTID)`PC := lockForRead35 ]
  ENDIF,
lockForRead20:
  IF currentWriter ≠ NT ∨ waitingWriters > 0
  THEN LET s = s1 WITH [ `global`rwlock`waitingReaders
                                          := waitingReaders + 1
                       , `threads(currentTID)`PC := lockForRead26 ]
       IN LET (s_upd,q_upd) = wait(s, readerWait)
          IN s2 = s_upd WITH [ `global`rwlock`readerWait := q_upd ]
  ELSE %¬(s1`global`currentWriter ≠ NT) ∨ s1`global`waitingWriters > 0)
       s2 = s1 WITH [ `threads(currentTID)`PC := lockForRead31 ]
  ENDIF,
lockForRead26:
  s2 = s1 WITH [ `global`rwlock`waitingReaders := waitingReaders - 1
               , `threads(currentTID)`PC := lockForRead20 ],
lockForRead31:
  s2 = s1 WITH [ `global`rwlock`threadCount := threadCount + 1
               , `threads(currentTID)`PC := lockForRead35 ],
lockForRead35:
  S2 = s1 WITH [ `global`rwlock`count(currentTID)
                                := count(currentTID) + 1
                 , `threads(currentTID)`atomic := false
                 , `threads(currentTID)`PC := incReadNest01],
```

listing 2.12 PVS model of the readLock function.

2.4.3 System invariants

In a system state, not every combination of variables will be reached during normal execution of the program. A certain amount of redundancy is present in the set of variables in the model. For instance, the number of writers waiting can be deduced both from the `waitingWriters` variable as well as the length of the wait queue. Also, variables are maintained that keep track of the total amount of processes that occupy the critical section and of the number of processes that are waiting for a lock. We express the integrity of the values of those variables by using a `validState?` predicate. This is an invariant on the global state of all the processes and essential in proving that the algorithm is deadlock free. We want to express in this invariant that the global state is sane and safe at the time a context switch can take place. Sanity is defined as:

- the value of the `waitingReaders` should be equal to the total number of processes with a status of `Waiting` and that are a member of the `readerWait` queue. Counting the members of the wait queue is done by the recursive `waitingReaders` function.

```
waitReadInv(s:System) : bool =
    s‘global‘rwlock‘waitingReaders = waitingReaders(s)
```

- the value of the `waitingWriters` should be equal to the total number of processes with a status of `Waiting` and that are a member of the `writerWait` queue. The `waitingWriters` function counts the waiters in the queue.

```
waitWriteInv(s:System) : bool =
    s‘global‘rwlock‘waitingWriters = waitingWriters(s)
```

- the value of the `threadCount` variable should be equal to the number of processes with a lock count of 1 or higher and at the same time this equals the total number of `readers` and `writers`. Again, recursively defined in the `count` function.

```
countInv(s:System) : bool =
     s‘global‘rwlock‘threadCount = count(s‘threads)
```

Besides the redundant variables having sane values, we also prove that the invariant implies that a waiting process does not have a lock, indicated by having a `count` of zero. If it has obtained a lock, it must necessarily be `Running`.

```
statusInv(s:System): bool = ∀(tid:ThreadID):
  LET thr = s‘threads(tid) IN
  thr‘state = Waiting ⇒ s‘global‘rwlock‘count(thr) = 0
 ∧ s‘global‘rwlock‘count(thr) > 0 ⇒ thr‘state = Running
```

Part of the invariants defined in section 2.4.3 are defined as `safetyInv`:

```
safetyInv(s:System) : bool =
 (readers = 0 ∨ writers = 0 ) ∧ writers ≤ 1
```

When a process has obtained a write lock, only that process is in the critical section:

```
writeLockedByInv(s:System) : bool =
  currentWriter ≠ NT ⇒ threadCount = 1 ∧
  count(currentWriter) > 0 ∧
  ∀(tid:ThreadID): tid ≠ currentWriter ⇒ count(tid) = 0))
```

The combination of all these invariants makes up a valid state.

```
validState?(s:System) : bool = countInv(s) ∧ waitWriteInv(s) ∧
  statusInv(s) ∧ writeLockedByInv(s) ∧ safetyInv(s) ∧ waitReadInv(s)
```

The definition of `interleave` generates a type correctness condition that will guarantee that if we are in a valid state, we will transition with an interleaving to another state that is still valid. We also show that the starting state is a valid state. The proof of this correctness condition is a straightforward, albeit large, case distinction.

2.4.4 Freedom from deadlocks and livelocks

The theorem-prover PVS does not have an innate notion of deadlock. If, however, we consider the state-transition model as a directed graph, in which the edges are determined by the `interleave` function, deadlock can be determined by identifying states in the state transition graph having no outgoing edges. This interpretation of deadlock, however, can be too limited. If, for example, there is a situation where a process alters one of the state variables in a non terminating loop, a deadlock will not be detected, because each state has an outgoing edge. There still can be **livelock**; transitions are possible, but there will be no progress. To prove there can be no livelock, we define a well founded ordering on the all valid system states and show that for each state reachable from the starting state (except for the starting state itself), there exists a transition to a smaller state according to that ordering. The smallest element within the order is the starting state. This means that for each reachable state there exists a path back to the starting state and consequently it is impossible for any process to get stuck in a such a loop indefinitely. Moreover, this also covers the situation in which we would have a **local deadlock** (i.e. several but not all processes are waiting for each other).

We create a well founded ordering by defining a state to become smaller if the number of waiting processes decreases or alternatively, if the number of waiting processes remains the same and the total count of the number of processes that have obtained a lock is decreasing. Well foundedness follows directly from the well foundedness of the lexicographical ordering on pairs of natural numbers.

```
smallerState(s2, s1 : (validState?)) : bool =
  numberWaiting(s2) < numberWaiting(s1) ∨
  numberWaiting(s2) = numberWaiting(s1) ∧
                        totalCount(s2) < totalCount(s1)
```

The numberWaiting function is a function on the array of thread-states that yields the number of processes that have a Waiting status. The totalCount function computes the sum of all the elements of the count array.

Once we have established that each state transition maintains the validState? invariant, all we have to prove is that each transition has outgoing states and that all of these states (except for the starting state) will possibly result in a state that is smaller. This is the noDeadlock theorem.

```
noDeadlock: THEOREM
∀(s1: (validState?)) : ∃(s2: (validState?)) :
  interleave(s1, s2)∧ (¬startingState(s1) ⇒ smallerState(s2, s1))
```

All that is needed to prove this theorem is a case distinction and inductive proofs of auxiliary lemmas that state that the recursively defined counting functions used in the invariant definitions are only decreased and increased if certain preconditions are met.

The proofs of the absence of deadlock proceeds analogously to the proof demonstrated for an earlier, more abstract, version of this model by the same authors in [BvG-1].

2.4.5 Freedom from starvation

There is no builtin notion of starvation in PVS either. We define the absence of starvation as a theorem stating that if a thread intends to acquire a lock, it will eventually obtain it. The intention is identified by the thread entering the lockForWrite part of the code.

```
noWriterStarvation: THEOREM
∀ (s1:(validState?)) :
  s1‘threads(s1‘currentTID)‘PC = lockForWrite42 ⇒
    lock_on_trace(s1, s1‘currentTID)
```

Eventually obtaining the lock is defined using the observation that for all traces of possible interleavings, the thread wanting to acquire a lock will become the current writer.

```
lock_on_trace(s1:System, lockTID:ThreadID) : RECURSIVE boolean =
∀ (s2:(ValidState?)) : interleave(s1,s2)∧
  (s2‘global‘rwlock‘currentWriter = lockTID∨
        lock_on_trace(s2, lockTID))
MEASURE s1 BY lock_on_trace_measure(lockTID)
```

This recursive relationship is well-founded, since the measure defined in this function guarantees termination. Proving that for each interleaving the measure decreases, again, is done by a massive case distinction. The complete proof, including the proof of the absence of writer starvation is available digitally.

2.5 Related and future work

Several studies investigated **either** the conversion of code to state transition models, as is done in [Eek+08] with mCRL2, **or** the transformation of a state transition model specified in a model checker to a state transition model specified in a theorem prover, as is done e.g. in [Kat01] using VeriTech. With the tool TAME one can specify a time automaton directly in the theorem prover PVS [AHS98]. For the purpose of developing consistent requirement specifications, the transformation of specifications in a model checker (Uppaal [LPY97]) to specifications in PVS has been studied in [Gro08].

In [Pan+06] model checking and theorem proving are combined to analyse the classic non-reentrant (in contrast to the reentrant version studied in our paper) readers-writers problem. The authors do not start with actual industrial source code but they start from a tabular specification that can be translated straightforwardly into Spin and PVS. Safety and clean completion properties are derived semi-automatically. [HS96] reports on experiments in combing theorem proving with model checking for verifying transition systems. The complexity of systems is reduced abstracting out sources for unboundedness using theorem proving, resulting in a bounded system suited for being model checked.

The verification framework SAL [Sha00] combines different analysis tools and techniques for analysing transition systems. Besides model checking and theorem proving it provides program slicing, abstraction and invariant generation.

In [Ha+04] part of an aircraft control system is analysed, using a theorem prover. On a single configuration this was previously studied with a model checker. A technique called **feature-based decomposition** is proposed to determine inductive invariants. It appears that this approach admits incremental extension of an initially simple base model making it better scalable than traditional techniques.

Directly operating on Java source code is Java Pathfinder (JPF) [Vis+03], which makes source code transformation superfluous. If the code studied would have been written in Java, JPF would have been the foremost candidate tool for this case study. This can be done directly within JPF or, if that is desirable, even by generating Promela code as was done originally in [HP00]. It would be interesting to compare the effort, ease of modelling and ease/performance of model checking of tools for different languages by taking the case study of this paper and performing it also for the same algorithm written in Java using e.g. the extension of JPF with symbolic execution [APV07]. Alternatively, Bandera [Cor+00] could be used for such a comparative case study. Bandera includes support for abstractions which may be very useful in such a case study. It translates Java programs to the input languages of SMV and Spin. There is an interesting connection between Bandera and PVS. To express that properties do not depend on specific values, Bandera provides a dedicated language for specifying abstractions, i.e. concrete values are automatically replaced by abstract values, thus reducing the state space. The introduction of these abstract values may lead to prove obligations which can be proven in PVS.

In [Rob+06] a model checking method is given which uses an extension of JML [LKP07] to check properties of multi-threaded Java programs. With Zing [And+04] on the one hand models can be created from source code and on the other hand executable versions of the transition relation of a model can be generated from the model. This has been used successfully by Microsoft to model check parts of their concurrency libraries.

Future work The methodology used (creating in a structured way a model close to the code, model checking it first and proving it afterwards [SE08]) proved to be very valuable. We found a bug, improved the code, extended the capabilities of the code and proved it correct. One can say that the model checker was used to develop the formal model which was proven with the theorem prover. This decreased significantly the time investment of the use of a theorem prover to enhance reliability. However, every model was created manually. We identified several opportunities for tool support and further research:

deep versus shallow embedding A complete specification of the semantics and syntax of Promela in PVS was avoided in our construction of the PVS model. We focused on methodically translating between the two models. Greater confidence of the translation may be achieved by using a translation that preserves the structure of the original Promela code instead.

bounded model related to source code Tool support could be helpful here: not only to 'translate' the code from the source language to the model checker's language. It could also be used to record the abstractions that are made. In this case that were: basic locks → lock process model, hash tables → arrays, threads → processes and some name changes. A tool that recorded these abstractions, could assist in creating trusted source code from the model checked model.

relation of finite to unbounded model It would be interesting to prove that the model in the theorem prover and the model checked are properly related, e.g. by establishing a refinement relation [Bar05] between them. Interesting methods to do this would be using a semantic compiler, as was done in the European Robin project [Tew+08], or employing a specially designed formal library for models created with a model checker, e.g. TAME [AHS98].

relation of unbounded model to source code Another interesting future research option is to investigate generating code from a fully proven PVS model. This could be code generated from code-carrying theories [JSW07] or it could be proof-carrying code through the use of refinement techniques [Bar05].

2.6 Discussion

We have investigated Qt's widely used industrial implementation of the reentrant readers-writers problem. Model checking revealed an error in the implementation (version 4.3). Qt Group Plc was informed about the bug, after which Qt Group Plc released a new version of Qt (version 4.4) in which the error was repaired. However, the new version of the Qt library is still only weakly reentrant, not admitting threads that have write access to do a read lock. This limitation unnecessarily hampers modular programming.

The improved readers-writers model described in this paper is **deadlock free** and **strongly reentrant**. The model was first developed and checked for a limited number of processes using a model checker. Then, the properties were proven for any number of processes using a theorem prover. We also studied the **absence of starvation**. With model checking a starvation problem was revealed. We created a starvation-free implementation and verified it with model checking. The outline of a proof for this implementation was sketched in PVS.

3

INFERRING TYPES AND CHECKING CORRECTNESS OF ON-CHIP COMMUNICATION NETWORKS

ABSTRACT *For software to function correctly, the hardware running the software also has to be correct. In the multi-core era, on-chip communication networks are key to system correctness and performance. To deal with their growing complexity, micro-architectural models capture the intent of architects and provide means for formal analysis. However, the analysis of such micro-architectural models is restricted to non-scalable and/or very specific approaches. We present a novel scalable approach using symbolic execution on large micro-architectural models described in the xMAS language proposed by Intel. Symbolic channel types are inferred by an algorithm that computes all possible messages that can occur in a communication channel, treating their payload symbolically. These symbolic types are used to verify absence of misrouting. These results can be used for further analysis such as deriving inductive invariants and deadlock detection. We illustrate our approach with a prototype on a 2D mesh, and a Spidergon network developed at STMicroelectronics.*

3

based on [BvG-11]

correctness analysis | hardware | symbolic execution | type inference

The communication network of a processor is crucial to the overall correctness and performance of a modern multi-processor Systems-on-Chip (SoC). When the number of interconnected system elements increases, performance of bus-based architecture degrades [BM02]. Networks-on-Chip (NoC) designs have emerged as solid high performance communications architectures. Recently, Intel proposed a language – called xMAS, for eXecutable Micro-Architectural Specification – to capture the intent of architects [CKO10; CKO12]. These models are executable and also amenable to formal verification. It is possible to extract high-level information about a Verilog design from its xMAS abstraction. This information can then be used to improve model checking the Verilog description [CK10; GCK11; CK12]. Efficient deadlock verification on large xMAS models was demonstrated in [VS11; VS12]. Although many concrete simulation techniques exist, they lack scalability because they simulate a NoC with clock cycle precision, and are therefore not applicable to a communication-centric SoC [RJE03]. We place ourselves in the context of the scalable verification at the level of xMAS models.

Our main contribution is to extend the analysis of xMAS models with the inference of channel types. Our approach is based on a symbolic propagation algorithm. This effectively computes the typing information of all channels, i.e. for each channel it computes the set of packets that can possibly traverse this channel. We define two symbolic types,

enumeration and interval range. Every component/packet/function in the network is modelled in terms of sets and operations on sets of these symbolic types. Due to this modelling, a symbolic packet can be split and eventually end up in multiple sinks with different payloads. Our network representation closely matches the formal semantics of xMAS networks as described in [BvG-5], in order to make it feasible to use this approach in a generic formal proof. We implemented our algorithm in C++. Starting from a representation of an xMAS network (including the description of what kind of packets each node may inject), our procedure is fully automatic. We demonstrate the application and scalability of our algorithm on the Spidergon design from STMicroelectronics, and a 2D mesh.

3.1 The xMAS language

A model in xMAS is a network of primitives connected via typed **channels**. A channel is connected to an **initiator** and a **target** primitive. A channel is composed of three signals. Channel signal $c.\mathrm{irdy}$ and $c.\mathrm{trdy}$ indicates respectively whether the initiator is ready to write to channel c, and indicates whether the target is ready to read from channel c. Channel signal $c.\mathrm{data}$ contains data that are transferred from the initiator output to the target input if and only if both signals $c.\mathrm{irdy}$ and $c.\mathrm{trdy}$ are set to true. The eight primitives of the xMAS language are listed in figure 3.1. A **function** primitive manipulates data. Its parameter is a function that produces an outgoing packet from an incoming packet. Typically, functions are used to convert packet types and represent message dependencies inside the fabric or in the model of the environment. A **fork** duplicates an incoming packet to its two outputs. Such a transfer takes place if and only if the input is ready to

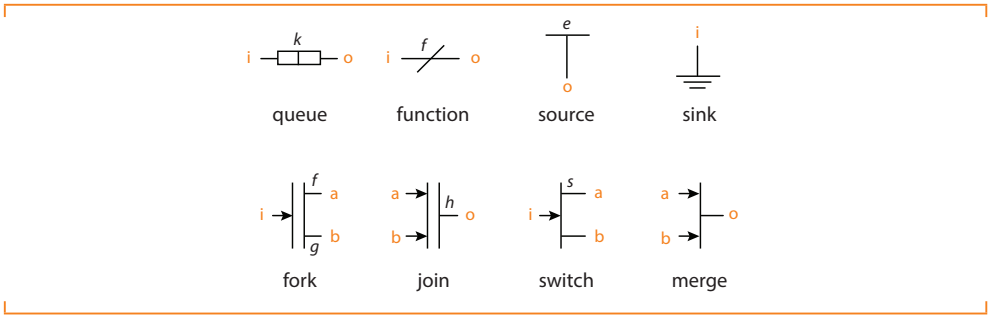

figure 3.1 The eight xMAS primitives.

send and the two outputs are both ready to read. The dual of a fork is a **join**. The function parameter determines how the two incoming packets are merged. Transfers take place if and only if the two inputs are ready to send and the output is ready to read. A **switch** uses its function parameter to determine to which output an incoming packet must be routed. A **merge** is an arbiter. It grants its output to one of its inputs. The arbitration policy is a parameter of the merge. A **queue** stores data. Messages are non-deterministically produced and consumed at **sources** and **sinks**. Sources and sinks may process multiple packet types.

The execution semantics of an xMAS network consists in a combinatorial and a sequential part. The combinatorial part updates the values of channel signals. The sequential part is the synchronous update of all queues according to the values of the channel signals. A simulation cycle consists of a combinatorial and a sequential update. A sequential update only concerns queues, sinks, and sources. We denote these primitives as **sequential primitives**. Other primitives are denoted as **combinatorial**.

For each output port o, signal $o.\mathbf{irdy}$ is set to true if the primitive can transmit a packet towards channel o, i.e. port o is ready to transmit to its target. For each input port i, signal $i.\mathbf{trdy}$ is set to true if the primitive can accept a packet from input channel i, i.e. the target of channel i is ready to receive. In a sequential primitive, the values of output signals depend on the values of the input signals and an internal state. Queues accept packets only when they are empty. A source and a sink produces or consumes a packet according to an internal oracle modelling non-determinism.

3.2 The Spidergon design

Spidergon is a Network-on-Chip developed at STMicroelectronics [Cop+09]. The basic architecture consists of eight nodes connected in a ring with across links, as can be seen in figure 3.2. A popular routing algorithm for this network is **across first**. The idea is that if a packet needs to use across links to minimise the travel distance an across hop is always performed first.

A micro-architectural model of a Spidergon router is shown in figure 3.3. The router has four inputs and four outputs. Packets are coming from either other routers or from input A, i.e. the local input. Switches are used to route packets towards their respective output ports. Arbiters are used to handle conflicting requests for output ports.

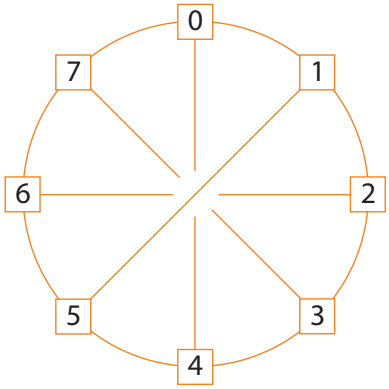

figure 3.2 Topology of an eight node Spidergon network

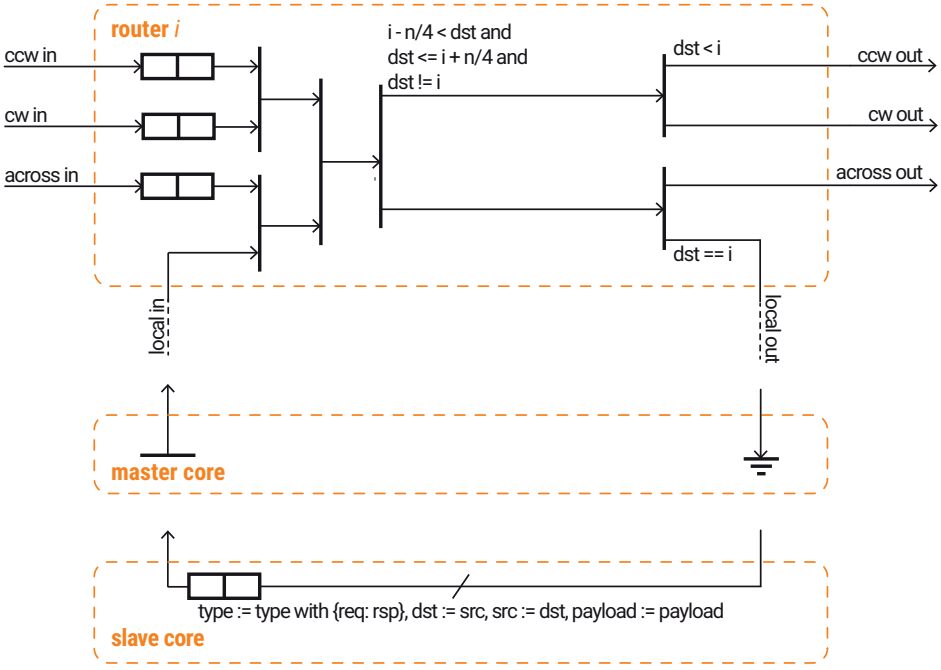

figure 3.3 Internals of a Spidergon router with across, cw (clockwise) and ccw (counterclockwise) incoming and outgoing connections, and two sorts of local cores (master and slaves).

Different kinds of cores can be connected to the routers (see figure 3.3). For sake of an example that includes message dependencies [HGR07] we have a setup with masters and slaves. Masters inject requests and consume responses. Slaves transform requests into responses. For a topology of n nodes, a master injects packets that contain the following fields:

dst An integer ranging from 0 to n that represents the destination of the packet. This destination is always a slave.

src An integer ranging from 0 to n representing the original injection point of the packet. This is used to send the packet back after arriving at its destination.

colour Either request (when it is injected) or response (when the packet has visited its destination and is returning to its original injection point).

payload Some 32 bit integer.

In the remainder of this paper, we will use the Spidergon design as a running example. We will show how one can define the routing functions within the switches and specify at sources what kind of packets can be injected. We will use our algorithm to compute all typing information for configurations going from 8 to 1024 nodes.

3.3 Specification of packets

A simple expression based language is needed to express the intent and effect of primitives. Packets consist of a number of fields. Each field is either an integer (e.g. $\mathbf{dst}$ and $\mathbf{src}$ in the Spidergon example) or an enumeration (e.g. the $\mathbf{colour}$ field). There are two kinds of expressions supported: **matching expressions** and **modifying expressions**. Matching expressions are used in xMAS sources and xMAS switches to determine which packets are injected. Modifying expressions are used in the xMAS function to express how a packet is altered.

For example, the following matching expression (automatically generated) corresponds to the xMAS switch connected to the local input a (see figure 3.3). This switch decides whether the incoming packet is routed across. We consider the fourth node in the network, meaning that packets destined for $0, 1,$ and 7 should be routed across. The following

expression generates two intervals: $[-1\,..\,1]$ and $[7\,..\,9]$. As the dst field signifies a node number (which in the example is between 0 and 7), this expression only matches for nodes 0, 1, and 7, as desired:

```
(dst > 4 ? dst > 6 : dst > -2) && (dst > 4 ? dst < 10 : dst < 2)
```

The following modifying expression transform requests into responses in a slave node:

```
dst := src, colour := colour with {req: rsp}
```

This expression yields a new packet with as destination the original source. The new colour is obtained by means of a mapping, which turns requests into responses.

We use the syntax described by the following BNF grammar for **matching expressions**:

⟨*expr*⟩ ::= ⟨*enum-match*⟩ | ⟨*integer-match*⟩
| '(' ⟨*expr*⟩ ')'
| '!' ⟨*expr*⟩
| ⟨*expr*⟩ '?' ⟨*expr*⟩ ':' ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*logical-op*⟩ ⟨*expr*⟩

⟨*logical-op*⟩ ::= 'and' | '&&' | 'or' | '||'

⟨*enum-match*⟩ ::= ⟨*variable*⟩
| ⟨*variable*⟩ 'in' '{' ⟨*enum-contents*⟩ '}'
| ⟨*variable*⟩ 'not' 'in' '{' ⟨*enum-contents*⟩ '}'

⟨*enum-contents*⟩ ::= ⟨*label*⟩ | ⟨*label*⟩ ',' ⟨*enum-contents*⟩

⟨*integer-match*⟩ ::= ⟨*variable*⟩
| ⟨*variable*⟩ ⟨*compare-op*⟩ ⟨*constant*⟩
| ⟨*variable*⟩ 'in' '[' ⟨*constant*⟩ '..' ⟨*constant*⟩ ']'
| ⟨*variable*⟩ 'not' 'in' '[' ⟨*constant*⟩ '..' ⟨*constant*⟩ ']'

⟨*compare-op*⟩ ::= '<' | '<=' | '>=' | '>' | '==' | '!='

⟨*constant*⟩ ::= ⟨*integer*⟩
| ⟨*constant*⟩ ⟨*constant-op*⟩ ⟨*constant*⟩
| '(' ⟨*constant*⟩ ')'

⟨*constant-op*⟩ ::= '+' | '-' | '*' | '/' | '%' | '^'

For defining **modifying expressions** we support another syntax, as they express another kind of intent. The following BNF grammar describes these expressions:

```
⟨expr⟩             ::= ⟨field-definition⟩
                     | ⟨expr⟩ ',' ⟨field-definition⟩
⟨field-definition⟩ ::= ⟨variable⟩ ':=' ⟨value-expr⟩
⟨value-expr⟩       ::= ⟨variable⟩ | ⟨integer⟩ | '(' ⟨value-expr⟩ ')'
                     | ⟨value-expr⟩ ⟨arithmetic-op⟩ ⟨value-expr⟩
                     | ⟨value-expr⟩ 'with' '{' ⟨substitution-def⟩ '}'
⟨arithmetic-op⟩    ::= '+' | '-' | '*' | '/'
⟨substitution-def⟩ ::= ⟨label⟩ ':' ⟨label⟩
                     | ⟨label⟩ ':' ⟨label⟩ ',' ⟨substitution-def⟩
```

If a substitution is defined on an expression, a special label '_' can be defined, which is the default (fail-over) case of the substitution.

3.4 Symbolic representation of packets

The basic principle of our symbolic computation is to represent a set of concrete packets in a significantly smaller set of symbolic packets. The computation of all paths for concrete packets is effectively done by propagating their symbolic representations. Fields of the integer kind are symbolically represented by intervals, fields of the enum kind are represented by abstract enumeration labels.

To manipulate symbolic packets, we assume the following common operations on **fields**:

intersection of two sets.

difference of set a and b, defined as $a \cap \overline{b}$. This can be computed without having the means to take the complement of a set.

subset to check if a set is a subset of another set.

Note that the union operation is not explicitly listed, as we can represent the union of two symbolic packets by associating these two symbolic packets with the same channel independently of each other.

These operations are used for the manipulation of symbolic packets, by applying them in a field-wise fashion. We assume that each packet-field has a unique **label**. For each field operation $a \square b$ ($\square$ is intersection or difference) and for each label l of the first argument, if the label is also present in the second argument, perform $a_l \square b_l$. If the label is not present, the resulting field for label l in the result is equal to the field in the first argument a.

Consider again the Spidergon example. For an eight node Spidergon network, the source and destination fields will be in a range of 0 to 7. The payload can be any value between 0 and $2^{32} - 1$. All possible concrete packets of these fields, can be represented in the following singleton set containing one symbolic packet:

$$\left\{ \begin{array}{ll} \text{dst} \rightarrow [0..7] & \text{colour} \rightarrow \{\text{request}, \text{response}\} \\ \text{src} \rightarrow [0..7] & \text{payload} \rightarrow [0..2^{32} - 1] \end{array} \right\}$$

We continue with describing the field operations that are specific for each field kind.

3.4.1 Enumerate kind

This kind is the standard enumeration type, where the labels are kept symbolic. The only operation defined on this kind is a mapping operation, converting one value in another one. We denote an enumerate field by $\{a, b, c, d\}$.

3.4.2 Interval range kind

Intervals have a lower bound l and an upper bound h, both represented as an integer for which $l \leq h$. A number of integer arithmetic operations are supported on these types, such as addition and subtraction on intervals, and comparisons with a concrete number, like the operators greater than, less than, etc. Multiplication is supported, but as it can result in multiple non-continuous regions it is prone to state explosion. An interval field is denoted by $[l..h]$.

The addition of two bounds is defined by adding the lower bounds together for the new lower bound, and doing the same for the upper bound. Formally, we use the following definition:

$$[a\,..\,b] + [c\,..\,d] = [(a+c)\,..\,(b+d)]$$

The unary minus operation is defined as:

$$-\,[a\,..\,b] = [-b\,..\,-a]$$

Combining these two definition, subtraction is defined as follows:

$$[a\,..\,b] - [c\,..\,d] = [(a-d)\,..\,(b-c)]$$

Multiplication results in multiple nonadjacent intervals, and is defined as follows:

$$[a\,..\,b] \times [c\,..\,d] = \forall_{i\in[a\,..\,b]}\forall_{j\in[c\,..\,d]}\,[(i\times j)\,..\,(i\times j)]$$

Division is defined below. As division by zero is not defined on integers, we also consider division by an interval that includes zero an error as it should not occur.

$$[a\,..\,b] \,/\, [c\,..\,d] = \left[\left\lfloor\frac{a}{d}\right\rfloor\,..\,\left\lceil\frac{b}{c}\right\rceil\right]$$

3.4.3 Symbolic semantics for xMAS

Queues, merges, forks and **sinks** do not modify packets, i.e. all packets going into the primitive propagate to all the output channels unmodified. Regarding the remaining primitives, they can either filter packets (xMAS switch), modify the packets (xMAS function), combine two packets together (xMAS join) or serve as an injection point to the network (xMAS source). We now detail the semantics of these primitives.

xMAS switch A switch has a switching condition associated with it. This condition describes a set of packets that should be routed to output a. Since a symbolic packet signifies a set of possible values, a symbolic packet sw can be used to represent the switching condition.

The effect of the xMAS switch on packets in the input channel can be expressed using set operators, specifically the intersection and difference operators. The intersection of p with sw will yield the resulting packets that will be propagated to the channel connected to the a output: $p \cap sw$. Calculating which packets will propagate to the b output can be done by taking the intersection of p with the complement of sw: $input \cap \overline{sw}$.

The switching condition is described using the syntax for matching expressions as described in section 3.3. We show by example how we derive a symbolic packet sw from such an expression. The expression `a <= 10` describes the field a interval $[0\,..\,10]$. Combining this restriction with another can be expressed by `a <= 10 && a >= 5`, which will yield the intersections of $[0\,..\,10]$ and $[5\,..\,\infty]$, and result in the interval $[5\,..\,10]$. The conditional expression can also be expressed as a series of set operations: we define the semantic of `c?a:b` as $(c \cap a) \cup (\bar{c} \cap b)$, which results in two symbolic packets stored in the associated channel (as the union operation is not explicitly defined). Stating conflicting types for the same label in one expression, e.g. `a = {req,rsp} && a < 5`, is considered a type error and should be handled as such. A runtime typing error can occur if a field is accessed that does not exists (e.g. a switching function depends on a field that does not exists in a symbolic packet that is being propagated through the switch).

EXAMPLE *A switch routes packets based on the colour of the packet, as seen in figure 3.4. From the matching expression* `colour in {R}` *a symbolic representation is derived:* $\{\text{colour} \rightarrow \{\text{R}\}\}$. *If the symbolic packet* $\{\text{colour} \rightarrow \{\text{R},\text{G},\text{B}\}\}$ *is located in* c_0, *the effect of propagating the packet can be calculated using set operators. The resulting type in* c_1 *will be the intersection of the input packet with the symbolic representation of the switching condition:* $\{\text{colour} \rightarrow \{\text{R},\text{G},\text{B}\}\} \cap \{\text{colour} \rightarrow \{\text{R}\}\}$, *resulting in* $\{\text{colour} \rightarrow \{\text{R}\}\}$. *Likewise, the resulting type in channel* c_2 *will be* $\{\text{colour} \rightarrow \{\text{R},\text{G},\text{B}\}\} \cap \overline{\{\text{colour} \rightarrow \{\text{R}\}\}}$, *resulting in* $\{\text{colour} \rightarrow \{\text{G},\text{B}\}\}$.

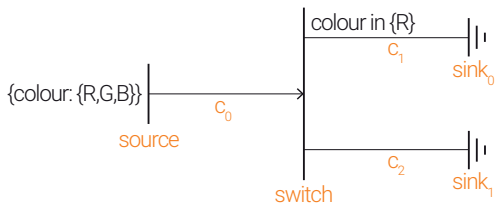

figure 3.4 Network demonstrating a switching function.

xMAS function The **function** primitive applies a function on a packet going through the primitive. Although the function is defined over concrete packets, the same function can be applied to symbolic packets. The effect of a xMAS function is specific to a kind of a field.

EXAMPLE *A network containing a function* `result := x + y` *(with* `result`, `x`, `y` *of the interval kind) transforms concrete packets going into the function. We can also calculate the effect of the function symbolically by using the symbolic semantics on intervals. If the function is applied to a symbolic packet* $\{x \to [0..16], y \to [8..32]\}$*, the result will be combining the two intervals. The resulting packet will be* $\{\text{result} \to [8..48]\}$.

xMAS join A join primitive combines the available symbolic packets of the two inputs into one packet, with each field prefixed with `a_` or `b_` to make clear from which input channel the field came. This is a somewhat different semantics as opposed to the original xMAS paper [CKO12], as we split their join in our join and a normal xMAS function. This eliminates the need to separately handle the function part of the xMAS join, so we do not need to define any additional syntax for joining expressions.

xMAS source An expression on a xMAS source primitive signifies which symbolic packets might be inserted at this point, which constitute a set of possible values that can be injected. These expressions are also described using the syntax for matching expressions, the same as for the xMAS switch.

3.5 Type inference algorithm

The type inference algorithm, listed in listing 3.5, is based on iteratively propagating a symbolic packet from a channel to the next channel, connected by a primitive. Propagation continues until a fix point has been reached, where no new inference can be performed. The algorithm outputs for each channel the set of symbolic packets describing which concrete packets can occur at the channel.

```
inject source types into channels;
while not all types in the network are marked as propagated do
    forall channels in the network do
        normalise types in channel;
        forall types in the channel do
            if type is not marked as propagated then
                propagate type;
                mark type as propagated;
```

listing 3.5 The basic propagation algorithm

A normalisation procedure is required to reach a fix point. If packets are added to a channel, all the available packets in the channel are normalised. The normalisation consists of two parts: **eliminating** symbolic packets that are already contained in other symbolic packets, and **combining** two symbolic packets together if possible. Both parts of the normalisation are essential.

The **elimination step** checks whether a symbolic packet is already described by another symbolic packet. If so, the symbolic packet can be removed with no effect on correctness. A symbolic packet is assumed to be described by another symbolic packet, if for each field in the symbolic packet the other symbolic packet contains a superset of the values allowed by that field.

The second step of the normalisation is the **combination step**. A symbolic packet is combinable if and only if all fields are equal, except for one field, and if this one field is combinable independently of the other fields. For enumeration field kinds, two fields are always combinable. For interval fields a and b are combinable if and only if

$$(a.\text{min} \leq b.\text{min} \wedge b.\text{min} \leq a.\text{max} + 1) \vee (a.\text{min} \leq b.\text{max} \wedge b.\text{max} \leq a.\text{max})$$

i.e. if one of the bounds of b lies in the interval as represented by a. For example, an interval of $[a\,..\,b]$ and $[(b+1)\,..\,c]$ can be combined into one interval of $[a\,..\,c]$. This step is used to reduce the runtime of the algorithm as it reduces the number of propagation steps that are needed.

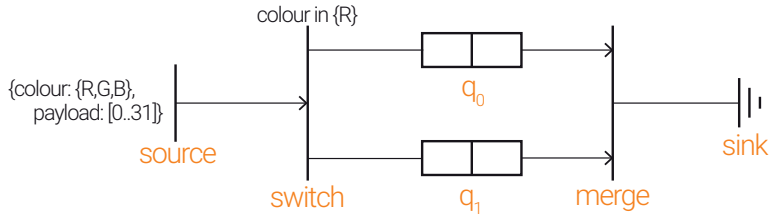

figure 3.6 Network demonstrating the combination step.

EXAMPLE *We demonstrate the combination step using a small network as shown in figure 3.6. In this network, the channel at the source and the sink have the same symbolic type* $\{\text{colour} \to \{R,G,B\},\ \text{payload} \to [0..31]\}$. *In the input channel of* q_0 *there are only symbolic packets with* $\{\text{colour} \to \{R\},\ \text{payload} \to [0..31]\}$, *as the switching condition only matches packets that are of* ***colour*** R. *The remaining packets are routed to the lower route with* q_1. *The merge propagates all the packets from the channels of the queues to the last channel. The type occurring at the sink is the result of combining the type*

$$\{\text{colour} \to \{R\},\ \text{payload} \to [0..31]\}$$

with the type from the bottom route

$$\{\text{colour} \to \{G,B\},\ \text{payload} \to [0..31]\}$$

3.6 Checking correctness of specification

The derived channel types can be used to check if a network conforms to a given specification. With each matching expression at a xMAS source we can add a specification: multiple destinations (xMAS sinks) with the corresponding expected packet as it should arrive at this destination. These ⟨destination, symbolic packet⟩ tuples can be **joined together with logical and and or operators**. xMAS primitives that modifying packets (function and join) are supported by specifying explicitly the symbolic packet as it should arrive in the specified sink. We extend the BNF grammar for matching expressions to support this construct. We add the following rules to the grammar, which in essence supports a ⟨*packet*⟩ '**->**' '**(**' ⟨*destination-node*⟩ '**,**' ⟨*packet*⟩ '**)**' form at each xMAS source.

⟨*source*⟩ ::= ⟨*source-expr*⟩
| ⟨*source-expr*⟩ ',' ⟨*source*⟩

⟨*source-expr*⟩ ::= ⟨*expr*⟩
| ⟨*expr*⟩ '->' ⟨*spec*⟩

⟨*spec*⟩ ::= '(' ⟨*node*⟩ ',' ⟨*expr*⟩ ')'
| ⟨*spec*⟩ ⟨*logical-op*⟩ ⟨*spec*⟩

First lets give an example of a specification, before continuing with describing the workings of the specification checking algorithm. For Spidergon networks with n nodes, with the first $\frac{n}{4}$ nodes slaves, typically one would inject at a given source in master node i request packets (which are represented as packet with the **colour** set to **req**) having the **dst** field set to a slave node. We add an additional field **src** to indicate to which node the packets should be returned. This will result in the following expression:

```
colour in {req} && dst < n/4 && src == i
```

Slave nodes can convert a request to a response with the following expression, keeping information to which slave node was visited. If a response is received by a slave node, the substitution statement with is defined to raise an error.

```
colour := colour with {req: rsp}, dst := src, src := dst
```

We can convert this into a specification which enforces that a packet injected at a node traverses the network to the specified slave node, and returns to the originating master node with the content of the packet modified as supposed to. The following expression is specified at the fifth node, a master, in a eight node Spidergon network (which has 2 slave nodes):

```
colour in {req} && dst == 0 && src == 5
    -> (sink5, colour in {rsp} && dst == 5 && src == 0),
colour in {req} && dst == 1 && src == 5
    -> (sink5, colour in {rsp} && dst == 5 && src == 1)
```

The algorithm constructs formulas by traversing the xMAS network for each given source specification. The formula contains ⟨destination, symbolic packet⟩ tuples combined with and and or operators. The current symbolic packet is propagated, and upon reaching a sink such a tuple is constructed (with the current sink and current symbolic packet). This tuple is then propagated backwards along the same route.

The manner of constructing the formulas depend on the xMAS primitive being traversed. A source, queue and merge just (back) propagates the packet/tuple. A function modifies the packet, and back propagates the tuple unmodified. A switch will propagate the packet, modified to account for the switching condition, and constructs an or formula from the tuples that are back propagated. A fork works likewise, and constructs an and formula. A join is special, as it needs the symbolic types inferred in the previous step. As packets are joined with the other channel from the join, the symbolic types that can occur in that channel are used to calculate the new symbolic packet that gets propagated. In a join, the return tuple is back propagated unmodified.

3.7 Implementation

We have implemented the aforementioned algorithms in a prototype called sym-xmas written in C++, without any external library dependencies. Our tool chain consists of this checker and a separate graphical editor for xMAS networks, as shown in figure 3.7. The network representation is outputted by this graphical tool. This representation can be read by our sym-xmas tool. For easy of testing we have included a few options in sym-xmas to generate networks in memory, to avoid large input files.

The syntax of expressions used in functions, switches and joins is defined in section 3.3. This syntax is read using a recursive descent parser. For matching expressions a set of possible values is generated by evaluating the expressions during parsing. We also added a constant expression evaluator for convenience of the user, so constant expressions with addition, subtraction, multiplication are supported. For modifying expressions we use an internal representation which applies the operations as defined in section 3.4.

Our tool defines some extra operations (e.g. hash function and printing a string representation) that are not essential to the algorithm but aid in execution speed and the debugging effort. All kinds of typing errors are detected, both conflicting restrictions on fields as well as applying expressions on non existent fields.

We developed our tool to be a common foundation to deploy all sorts of algorithms to analyse various correctness properties. Therefore it is important to maximise the flexibility of the network data structure. Two aspects are important. Foremost, the data structure supports the **visitor design pattern** [PJ98], enabling the decoupling of the algorithm executing on a data structure and the data structure itself. It enables add methods to the

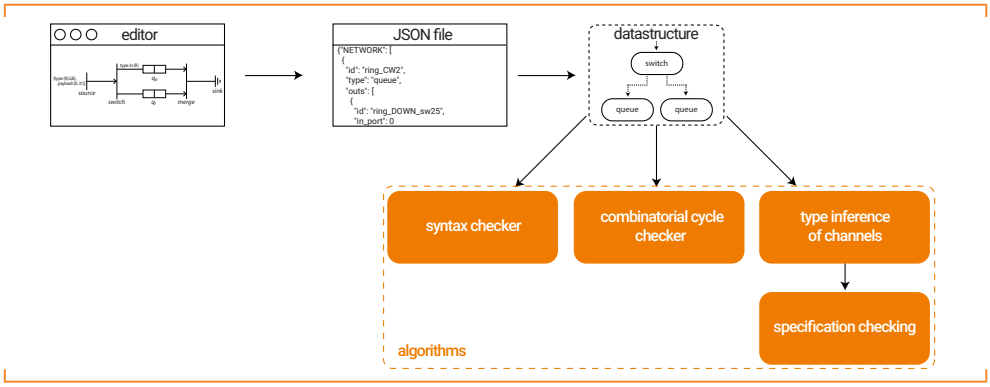

figure 3.7 The tool chain.

data structure without modifying the data structure itself, so a new algorithm or features can be added without modifying all the code depending on this data structure. Secondly, each object of the network can have algorithm specific data structure attached to it. This avoids costly lookups in mappings and enables easier algorithm design. In the future other algorithms can be based on this foundation. In fact our specification checking is implemented using this design.

There are two basic correctness requirements to xMAS networks: syntactic correctness and absence of combinatorial cycles. Before we symbolically infer types we check that an input xMAS network satisfies these correctness properties, otherwise the results of the type inference are not sound. Both are implemented using the data structure and the visitor pattern as mentioned before. The first one is easily checked by ensuring all ports are connected and output ports are only connected to input ports. This is a quick superficial check. The latter requirement is described in the paper introducing xMAS [CKO12]. In short it ensures a stable state can be reached between clock cycles, ensuring the data transferred is deterministic. A combinatorial cycle is a cycle of dependencies of **irdy** and **trdy** wires. Absence thereof can be verified using a standard cycle detection algorithm for a directed graph [Cor+01].

We implemented most steps single threaded, as the execution time was either small or making it multithreaded showed no savings. However, specification checking was faster multithreaded, because the shared data was only used to read. Therefore for the specification checking the number of cores directly influenced the result. In the next section we will describe two small case-studies of our tool.

3.8 Application to Spidergon

Using our sym-xmas tool, the symbolic types of channels in the Spidergon design were inferred. In our Spidergon example, the nodes of the first quadrant are slaves. All other nodes are masters. All packets have a payload of a 32-bits integer. In order to attain a notion of correctness, we keep the original injection point in the responses. This allows us to validate that the packets that are evacuated from the network are indeed correctly handled by the communication fabric. At the sinks in the network, the types inferred are stated below, with n the concrete value of the node number the sink is located in:

$$\left\{ \begin{array}{ll} \text{dst} \to [n\,..\,n] & \text{colour} \to \{\text{response}\} \\ \text{src} \to [n\,..\,n] & \text{payload} \to [0\,..\,2^{32}-1] \end{array} \right\}$$

This expression shows that node n only receives responses destined for n, that were originally injected at that same node. In other words, in the network requests and responses are correctly handled. By using specifications as described in section 3.6, we can automate this checking process. For larger networks, automatic checking is essential.

Actually during development of our algorithm, the version we analysed contained (without our knowledge) a wrong switching expression for the counter clockwise channel, resulting in packets misrouted to the wrong node. This error was easily detected and corrected. The source of the problem was a **off-by-one** error in a switching condition of a switch in the router.

The experiment was conducted on a 2 GHz quad core Intel Core i7 2635QM running Mac OS X 10.11.5, with the Apple LLVM clang 703.0.31 compiler. Results are listed in table 3.8.

nodes	primitives	type inference	specification checking	memory
8	88	0.002 s	0.001 s	1.48 MiB
16	176	0.013 s	0.004 s	2.41 MiB
32	352	0.091 s	0.023 s	5.46 MiB
64	704	0.529 s	0.156 s	19.35 MiB
128	1408	4.325 s	1.187 s	65.16 MiB
256	2816	40.635 s	10.038 s	254.22 MiB

table 3.8 Results of the experiments on a Spidergon design using a 2 GHz quad core Intel Core i7 2635QM.

3.9 Application to a 2D mesh

A topology in Network-on-Chips that is studied frequently is a two dimensional mesh of nodes. Each node has a router component and local core. The router component connects the local core with the nodes adjacent to this node. If the node is on the border of the mesh, those connections are left out, simplifying the router logic of the node. We implemented the router component as shown in figure 3.10, which works with with XY routing.

Like the previous case study, we could apply the same approach to the 2D mesh design. Packets are composed of five fields: the coordinates of the source and the destination, a bit indicating the type/colour (request or response) of the packet, and a payload of 32 bits. Masters inject requests and consume responses, similar to the Spidergon case study. Slaves swap the source and destination coordinates to produce a response sent back to its sender. The layout is such that masters are in the left half of the mesh and slaves are in the right half of the mesh. The results of our tool are listed in table 3.9.

size	primitives	type inference	specification checking	memory
20 x 20	6880	14.223 s	16.719 s	133.46 MiB
22 x 22	8360	24.664 s	27.327 s	201.21 MiB
24 x 24	9984	42.972 s	43.095 s	268.19 MiB
26 x 26	11752	74.165 s	64.279 s	350.53 MiB
28 x 28	13664	120.361 s	94.117 s	468.26 MiB
30 x 30	15720	206.980 s	135.788 s	594.95 MiB
32 x 32	17920	336.727 s	186.666 s	738.76 MiB
34 x 34	20264	507.672 s	246.822 s	956.00 MiB
36 x 36	22752	802.077 s	329.945 s	1141.55 MiB
38 x 38	25384	1248.430 s	439.082 s	1389.69 MiB
40 x 40	28160	3136.850 s	560.885 s	1907.86 MiB
42 x 42	31080	2890.370 s	731.258 s	2070.29 MiB

table 3.9 Results of the experiments on a 2D mesh design using a 2 GHz quad core Intel Core i7 2635QM.

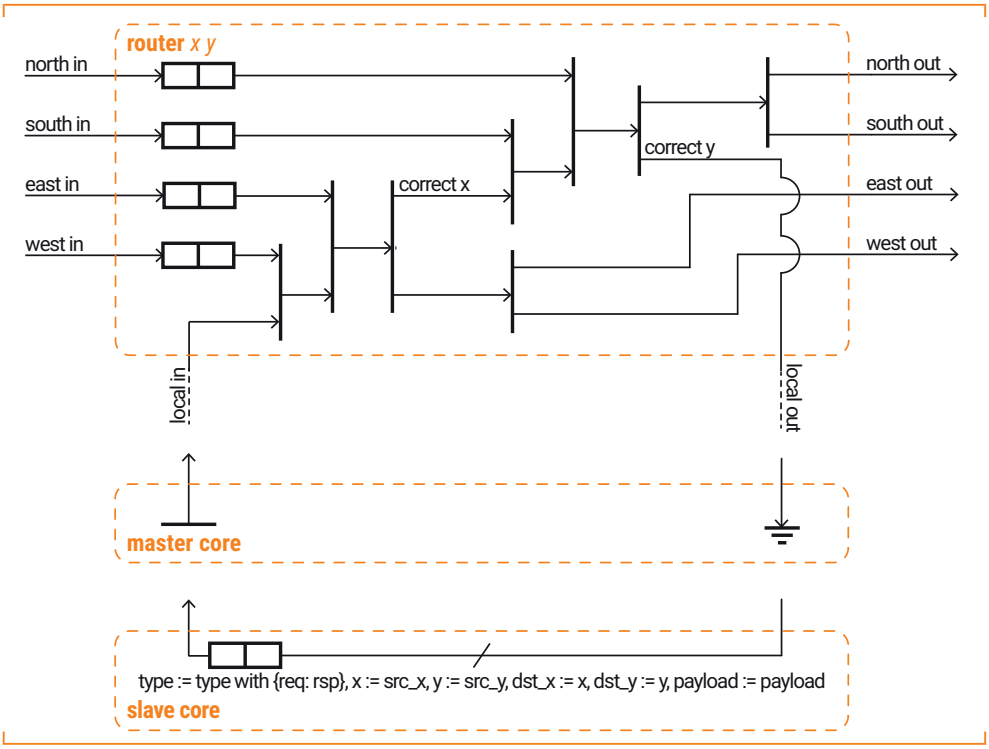

figure 3.10 Hermes router with incoming channels from four directions and the local core, and corresponding outgoing connections. There are two sorts of cores implemented: masters and slaves.

3.10 Discussion

For large networks, the explicit simulation of all possible values is not feasible. For such networks, our approach provides a scalable alternative. We presented an algorithm that efficiently infers the type of all channels in large networks described in the xMAS language proposed by Intel. By 'channel type', we denote the information about which packet can traverse the channel. This information is key in the analysis of xMAS networks. Our type inference algorithm produces at every sink, the set of packets which can possibly reach that sink. We also introduced an algorithm which, using the inferred types, checks whether the network corresponds to the specification of a designer, proving the absence of misrouting. We demonstrated this approach on two case studies, the Spidergon design and a two dimensional mesh, using a prototype implementing the proposed algorithms.

An future direction is the automatic generation of Register Transfer Level descriptions from xMAS networks. In this context, the knowledge of channel types can be used to determine the bit-width of all wires. Verification of deadlock freedom [VS11] also depends on Inferring channel types. Currently, all possible channel types in the mentioned freedom of deadlock verification method are constructed one by one trying out all values, with a concrete representation. The symbolic execution technique discussed in this chapter should be incorporated in the freedom of deadlock verification method. A soundness proof for the technique presented in this chapter should be constructed. This can be done using monotone frameworks, or by other means.

The method used in this chapter, symbolic execution, has seen wide application on software. This shows there are similarities between hardware and software, although they differ greatly. Software has often a reduced concurrency level, however, hardware features highly parallel designs. One could envision incorporating state of the art techniques of symbolic execution, e.g. in order to avoid complexity in the symbolic simulation of loops in the circuit designs. Applying other techniques traditionally associated with software analysis would be interesting, for example abstract interpolation, or other techniques.

PRACTICAL RESOURCE ANALYSIS FOR (REAL-TIME) JAVA

4

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 chapter we focus mainly on memory related bounds. Freedom of deadlock and starvation, as covered in chapter 2, is a condition for properly deriving memory bounds. Correctly functioning hardware is also assumed, which can for example be analysed with the method covered in chapter 3.*

Recent research results have advanced the state of the art of resource consumption analysis. In this chapter we present a toolset ResAna that makes it possible to apply these research results in practice enabling 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. Our toolset works on real-time systems written in Java.

We give an overview of the capabilities of this toolset. 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.

4

based on [BvG-4] based on [BvG-10]

memory analysis **symbolic** **overapproximation**

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 European Union 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 [MRH12; WW12; BvG-4]. 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 [Alb+11a]. Researchers use polynomial interpolation [KSE08], reachability-bound analysis [GZ10], amortisation [HAH11], polynomial quasi-interpretation [Ama05] and new language features such as programmer-controlled destruction and copying of data structures [DMP10]. 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 [Eek+07]. 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 [BvG-4]. How this chapter extends [BvG-4] is described in section 4.5.

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, 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 [Lea+13].

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.

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 tool COSTA [Alb+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 [Mon+12].

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 4.1 loop bound analysis is described. Heap bound analysis and the adjustments that have been made to make it applicable in practice are presented in section 4.2. Analysing stack bounds is discussed in section 4.3. User experience with ResAna is described in section 4.4. Finally, in sections 4.5 and 4.6, related work is discussed and conclusions are drawn.

4.1 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**.

A loop ranking function is a function over (some of) the program variables used in the loop, that decreases at each iteration and is bounded by zero. A simple while loop is shown in listing 4.1. Although $100 - i$ is a perfectly fine ranking function as well, the most precise one for this loop is $15 - i$. This gives the exact number of iterations the loop will make, for arbitrary i (given that $i < 15$, see section 4.1.6).

In this section, we present a method for the automatic inference of polynomial ranking functions for loops, based on polynomial interpolation. The basic procedure was first presented in [SKv10]. It can infer **polynomial** ranking functions, whereas other methods are limited to linear symbolic or concrete bounds. Note that to derive concrete bounds from symbolic bounds, the analysis could be combined with data-flow analysis. To derive concrete upper and lower bounds on the number of iterations of a loop, upper and lower bounds have to be known statically for all the program variables in the symbolic bound.

We introduce polynomial-interpolation-based ranking function inference in section 4.1.1. In section 4.1.2, a quadratic example is given. The soundness of the method is discussed in section 4.1.3. Then, extensions to the basic method are discussed in section 4.1.4 (ranking functions with rational or real coefficients), section 4.1.5 (branching inside the loop body) and section 4.1.6 (disjunctional loop guards). In section 4.1.7 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 4.2.1.

```
1 while (i < 15)
2   i++;
```

listing 4.1 A simple while loop, with most precise ranking function $15 - i$.

4.1.1 Test-based inference of polynomial ranking functions for loops

In [SKv10] a method for the inference of polynomial ranking functions for loops is presented. Only loops in which the guards are conjunctions over arithmetical (in)equalities are considered. These have the following form, where $\langle num \rangle$ is a numerical program variable or constant and $\boxdot := \{<, >, =, \neq, \leq, \geq\}$:

$$\langle guard \rangle \quad ::= \quad \langle inequality \rangle \mid \langle inequality \rangle \wedge \langle guard \rangle$$

$$\langle inequality \rangle \quad ::= \quad \langle num \rangle \boxdot \langle num \rangle$$

The method proceeds in the following steps:

1. instrument the loop with a counter
2. run tests on a **well-chosen** set of input values
3. find **the** polynomial interpolation of the results

In this context **well-chosen** means that test-nodes have to be picked such that there exists a unique interpolating polynomial. This is the reason we can refer to **the** polynomial interpolation in step 3. Remember that a polynomial $p(z_1, \ldots, z_k)$ of degree d and dimension k (the number of variables) has $N_d^k = \binom{d+k}{k} = \frac{(d+k)!}{d! \cdot k!}$ coefficients. This is the number of test-nodes that we need. To ensure the existence of a unique interpolation, the test nodes are chosen to lie in so-called Node Configuration A (NCA). This condition was first presented in [CL87]; its application to loop-bound analysis is described in [SKv10]. Besides lying in NCA, test-nodes must also satisfy the guard of the considered loop. An algorithm for node search is presented in [SKv10].

In the current version of ResAna, the ranking function can contain primitive data types, object field access and array access. Note that in theory, the method could also handle loops for which the ranking function depends on for instance the height of a tree. However, since this height is not readily available in a program variable, this would require the addition of such a variable expressing the tree height by the programmer.

4.1.2 Quadratic example

Consider the example in listing 4.2. The most precise ranking function for this loop is the degree 2 polynomial $a \cdot b - c + 1$.

```
while (a > 0 && c <= b && c > 0) {
  if (c == b) { a--; c = 0; }
  c++;
}
```

figure 4.2 A while loop with degree 2 ranking function $a \cdot b - c + 1$.

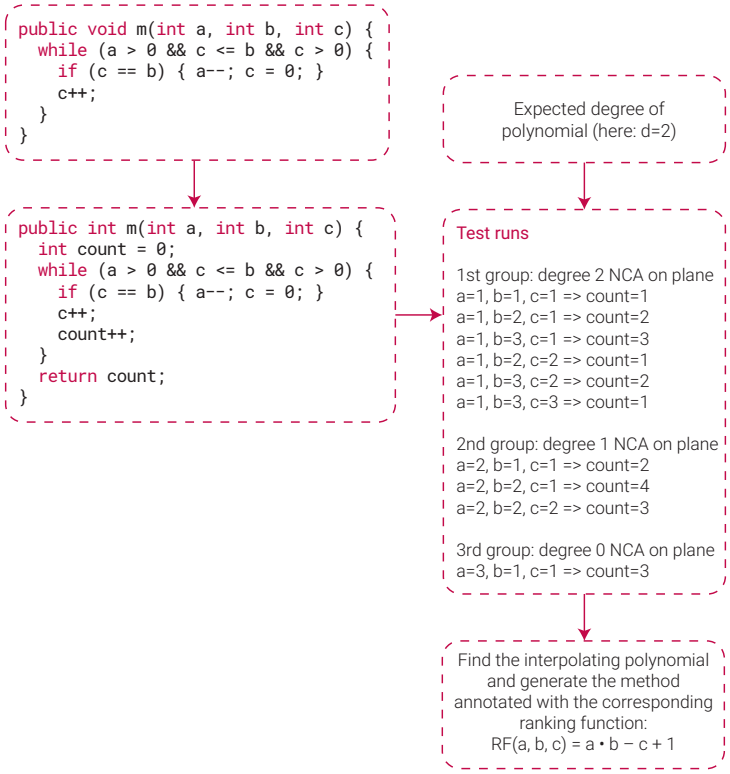

figure 4.3 Test-based inference method applied to the example from listing 4.2.

```
//@ decreases (a > 0) && (c <= b) && (c > 0) ? a*b-c+1 : 0;
while (a > 0 && c <= b && c > 0) {
  if (c == b) { a--; c = 0; }
  c++;
}
```

listing 4.4 The loop from listing 4.2, annotated with its ranking function.

The inference of a ranking function for the loop in listing 4.2 is depicted in figure 4.3. First, the loop is instrumented with a counter. The user inputs the expected degree 2 of the polynomial ranking function. Since there are 3 variables, a set of $N_2^3 = \frac{(2+3)!}{2!\cdot 3!} = 10$ test-nodes in NCA is generated. By interpolating the results from test runs using these input values, the most precise quadratic ranking function $a \cdot b - c + 1$ is found.

4.1.3 Soundness

The presented method infers a **hypothetical** ranking function. It is not sound by itself, but requires an external verifier. The Java Modelling Language (JML) is used to express the ranking functions [Cha+06]. Inferred ranking functions are 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 has a value greater than or equal to 0, see the JML reference manual [Lea+13]. It therefore forms an upper-bound on the number of iterations of the loop. An example is shown in listing 4.4.

When the loop condition does not hold, the loop iterates zero times. Therefore the shown annotation actually expresses the maximum of $a \cdot b - c + 1$ and 0. In general, a ranking function $\mathrm{RF}(\vec{v})$ for a loop with condition b can be expressed as follows: `decreases` b `?` $\mathrm{RF}(\vec{v})$ `:` `0`. Such JML annotations can be verified by a variety of tools, for instance KeY [BHS07]. The procedure described here should be used in conjunction with such a prover to provide soundness.

A bird's eye view of the overall procedure is depicted in figure 4.5. After a ranking function is inferred, the Java sources are annotated and sent to the verification tool (KeY). The verifier might be able to prove correctness of the annotation automatically, manual steps may be needed for complex ranking functions (non-linear, rational coefficients, et cetera)

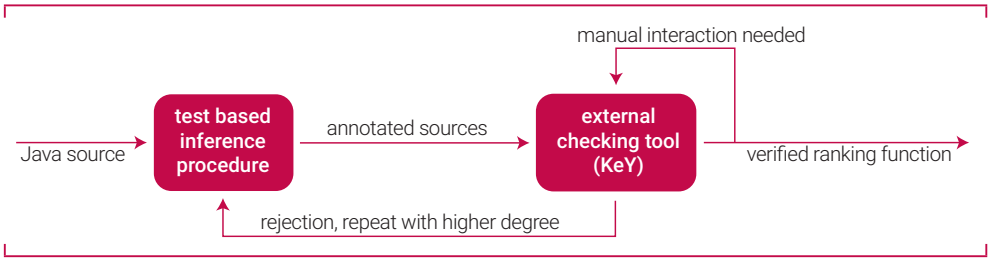

figure 4.5 The basic inference procedure from a bird's eye view: infer-and-check cycle.

or the user may not be able to construct a proof at all. In the latter case, the user can go back and try the procedure for a higher expected degree of the polynomial ranking function. If an expected degree higher than the actual degree of the polynomial is used, the correct result will still be found. There will however be a performance penalty on the analysis.

4.1.4 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 4.6.

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 $\mathrm{RF}(\bar{v})$ are not natural numbers, ceiling should be added as such: $\lceil \mathrm{RF}(\bar{v}) \rceil$. 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:

$$\lceil \mathrm{RF}(\bar{v}) \rceil \leq \mathrm{RF}(\bar{v}) + 1$$

When choosing test nodes for the loop in listing 4.6 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 $\mathbf{step}$, the test-nodes must lie $\mathbf{step}$ apart. In this example, if we pick test nodes (0,4), (4,8) and (4,12), then the correct ranking function will be found.

```
1 while (start < end) {
2   start += 4;
3 }
```

listing 4.6 An example with a loop-bound function that is a polynomial over rational coefficients.

```
1 while (i > 0) {
2     if (i > 100)
3         i -= 10;
4     else
5         i -= 1;
6 }
```

listing 4.7 The basic method supplies an incorrect ranking function. Branch-splitting is applied, yielding the pessimistic, but correct, ranking function i.

4.1.5 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 4.2. 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 4.7.

To solve this problem, we introduce **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 4.7, this yields the ranking function i.

4.1.6 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**.

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

listing 4.8 A while loop with a piecewise ranking function.

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 4.1, the ranking function is actually:

$$\begin{cases} 15 - i & \text{if } i < 15 \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

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 4.8. Its ranking function is:

$$\begin{cases} 20 - i & \text{if } (i > 0) \wedge (i < 20) \\ i - 50 & \text{if } i > 50 \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

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, resulting in a form as given below, with $\boxdot \in \{<, >, =, \neq, \leq, \geq\}$, and $\langle num \rangle$ a variable or constant:

$$\begin{aligned} \langle guard \rangle &::= \langle conj \rangle \mid \langle conj \rangle \vee \langle guard \rangle \\ \langle conj \rangle &::= \langle inequality \rangle \mid \langle inequality \rangle \wedge \langle conj \rangle \\ \langle inequality \rangle &::= \langle num \rangle \boxdot \langle num \rangle \end{aligned}$$

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

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

This transforms the condition $c_1 \vee \ldots \vee c_n$ into a set $\mathbf{Pieces}$ of $2^n - 1$ conjunctive conditions. For instance, $\mathbf{DNFsplit}(i > 10 \vee i < 3)$ yields three pieces:

1 $i > 10 \wedge \neg i < 3$ 2 $i < 3 \wedge \neg i > 10$ 3 $i > 10 \wedge i < 3$

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 $\mathbf{Pieces}$ 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 $p \in \mathbf{Pieces}$, the piecewise ranking function listed below is yielded. In this piecewise polynomial ranking function, $m \leq 2^n - 1$, because unsatisfiable pieces have been removed.

$$\begin{cases} \mathrm{RF}_{p_1} & \text{if } p_1 \\ \ldots & \text{if } \ldots \\ \mathrm{RF}_{p_m} & \text{if } p_m \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

4.1.7 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.

Consider the loop in listing 4.9. Naively, one could say that its ranking function is the following, which is an oversimplification:

$$\begin{cases} \lceil (20 - i)/4 \rceil & \text{if } (i > 0) \wedge (i < 20) \\ i - 22 & \text{if } i > 22 \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

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

listing 4.9 Jumping between the disjunctive conditions in a **while** loop.

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 separately:

$$\begin{cases} \lceil (20-i)/4 \rceil + 1 & \text{if } (i > 0) \wedge (i < 20) \wedge i \bmod 4 = 3 \\ \lceil (20-i)/4 \rceil & \text{if } (i > 0) \wedge (i < 20) \wedge i \bmod 4 \neq 3 \\ i - 22 & \text{if } i > 22 \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

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.

To detect condition jumping in the example in listing 4.9, 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 $\mathbf{next}_v$ function. The function $\mathbf{next}_i :: \mathbf{Int} \to \mathbf{Int}$ for the loop in listing 4.9 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:

$$\phi_i(\alpha_i) = \begin{cases} \alpha_i - 1 & \text{if } \alpha_i > 22 \\ \alpha_i + 4 & \text{otherwise} \end{cases} \tag{4.6}$$

By substituting the α_i symbol with i, we obtain the $\mathbf{next}_i$ function we are looking for:

$$\mathbf{next}_i(i) = \begin{cases} i-1 & \text{if } i > 22 \\ i+4 & \text{otherwise} \end{cases} \tag{4.7}$$

In general, such an update function can be derived by symbolic execution of the loop body. Start by giving the variables $v_1 \ldots v_n$ symbolic values $\alpha_1, \ldots, \alpha_n$. 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 $\alpha_1, \ldots, \alpha_n$ and constants, with associated **path conditions**, which capture branching. Effectively, this is again a piecewise polynomial. The function $\mathbf{next}_{v_i}$ is now obtained by replacing the α's by the corresponding program variables in this piecewise polynomial.

Detecting condition jumping We use SMT-LIB, 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 figure 4.9 is given in figure 4.10. The function $\mathbf{next}_i :: \mathbf{Int} \to \mathbf{Int}$ from equation 4.7 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.

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 $\mathbf{next}_{v_i} :: T_{v_1} \to \ldots \to T_{v_i} \to \ldots \to T_{v_n} \to T_{v_i}$, 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

```
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 4.10 SMT-LIB script to detect jumping in the code of figure 4.9.

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 (for instance Z3 by Microsoft) by determining the satisfiability of $b_1(v_1, \ldots, v_n) \wedge b_2(\mathbf{next}_{v_1}(LV), \ldots, \mathbf{next}_{v_n}(LV))$.

The SMT-LIB script in figure 4.10 can be used to find a **model** for which jumping 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 4.10, 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 the comparison on 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 $\mathbf{next} :: M \to M$ (where M is the type of a model), which applies to each variable v_i in a model $\bar{v}$ its corresponding $\mathbf{next}_{v_i}$ function. The algorithm is listed in listing 4.11.

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.

```
forall v̄ that satisfy b1(v̄) ∧ b2(next(v̄)) ∧ v̄ ∉ J do
  add v̄ to J and Q
while Q ≠ ∅ do
  q̄ = first item of Q
  forall v̄ that satisfy b1(v̄) ∧ next(v̄) = q̄ ∧ v̄ ∉ J do
    add v̄ to J and Q
  remove q̄ from Q
```

listing 4.11 Condition jumping detection algorithm.

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.

4.2 Heap-space usage analysis

The heap consumption analysis of ResAna is based on the COSTA [Alb+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 $\mathbf{nat}(n) \times \mathbf{c}(\mathbf{size}(\mathsf{X}))$, where $\mathbf{nat}(n)$ is the logical size $\mathbf{max}(n, 0)$, and $\mathbf{c}(\mathbf{size}(\mathsf{X}))$ is the memory cost of creating an object of type X.

COSTA implements different garbage collection models [AGG10]. 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 overapproximation. 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.

4.2.1 Interpolation-based height analysis for improving a recurrence solver

The approach of COSTA 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 consecutive 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 CRS s 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 4.12. 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, where c denotes the constant $\mathrm{c}(\mathrm{size}(\text{java.lang.Object}))$, i.e. the memory cost of creating an instance of Object.

$$T(x,y) = 0 \qquad \{x > y\} \tag{4.8}$$

$$T(x,y) = c + T(x+1,y) \qquad \{x \leq y\} \tag{4.9}$$

$$T(x,y) = c + T(x,y-2) \qquad \{x \leq y\} \tag{4.10}$$

The COSTA system provides the recurrence solver PUBS [Alb+11a], which computes the following closed-form:

$$\mathrm{nat}(y - x + 1) \times \mathrm{c}(\mathrm{size}(\text{java.lang.Object})) + \mathrm{c}(\mathrm{size}(\text{java.lang.Object}))$$

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

listing 4.12 Example loop.

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 section 4.1). For instance, in the CRS given above we get the following unfolding sequence of length $y - x + 1$:

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

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:

$$R(x,y) = c \qquad \{x = 0, y = 0\} \tag{4.11}$$

$$R(x,y) = c + R(x-1,x-1) \qquad \{x > 0, y = 0\} \tag{4.12}$$

$$R(x,y) = c + R(x,y-1) \qquad \{x \geq 0, y > 0\} \tag{4.13}$$

By equation (4.13) the variable y is decreased in every recursive call, until it reaches zero. Then, by equation (4.12) 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 4.1. This extension was described in detail in [Mon+12]. 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 section 4.1.1. 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 (4.8-4.10). We can unfold the definition of $T(5,9)$ by using (4.10) until we reach a base case:

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

This sequence is of length three, which is not maximal, since we could have evaluated T by always using (4.9), so as to obtain a longer sequence:

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

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 4.13 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

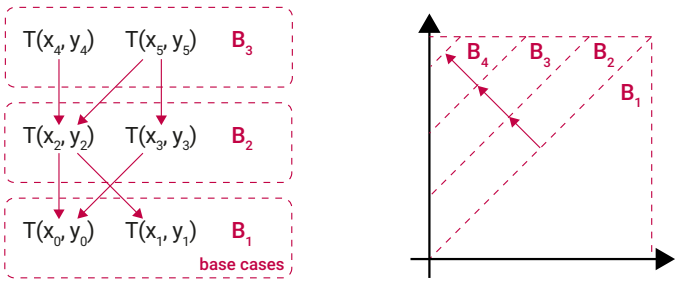

figure 4.13 Meaning of the B_i sets and their representation as convex polyhedra.

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 4.13 right). The algorithm involves the search of **climbing paths** starting at the B_1 set, and minimising 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 $T(x,y)$:

$$\forall x,y,x',y' : ((x \leq y \wedge x' = x + 1 \wedge y' = y) \vee (x \leq y \wedge x' = x \wedge y' = y - 2)) \\ \Longrightarrow y - x + 1 \geq 1 + y' - x' + 1$$

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 our running example $T(x,y)$, the script in figure 4.14 is generated. For this script, QEPCAD yields **true** as an equivalent formula, validating the inferred bound.

```
[ 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
```

listing 4.14 QEPCAD script for our running example $T(x,y)$. Variables have been renamed for better readability.

4.2.2 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 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 4.15. COSTA takes into account neither the

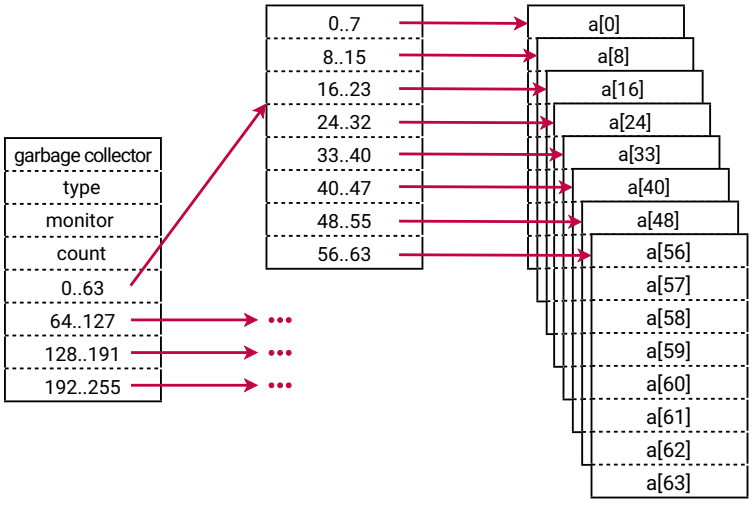

figure 4.15 Graphical representation of a JamaicaVM array of size n, with $33 \leq n \leq 255$, with $a[i]$ representing the contents of the array. Allocating an array of 63 elements takes 10 blocks.

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 $n \times \mathbf{size}(\mathsf{int})$, 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, as explained next.

4.2.3 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 virtual machine by means of a specially generated program. For JamaicaVM, this generated program depends on the Scoped Memory extensions of realtime Java. For other Java virtual machines we use the reflection interface in Java, which is more general and can be run on any virtual machine which supports the reflection interface. 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 overapproximation. A simple way of calculating the size of arrays, by means of the small recursive function defined in equation 4.14, could not be implemented in COSTA, because of the manner COSTA represents and calculates the bounds internally.

$$\mathbf{arrayblocks}(n) = \begin{cases} n & \text{if } n \leq 8 \\ \lceil \frac{n}{8} \rceil + \mathbf{arrayblocks}(\lceil \frac{n}{8} \rceil) & \text{otherwise} \end{cases} \tag{4.14}$$

This recursive function is valid for data types of four bytes, which correspond to the size of pointers used in the tree structure pointing to the leafs, resulting in a cleaner formula. 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).

By transforming the formula to an overapproximation (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 4.15. We have implemented

this solution in our version of COSTA, which is included in ResAna, so that analysing arrays now gives a correct overapproximation.

$$\mathrm{arrayblocks}(n) \quad \leq \quad \frac{n+5}{7} + (\log_8 n + 7) \tag{4.15}$$

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 virtual machine a new specialisation for arrays needs to be added in order to correctly generate bounds for code using arrays.

4.2.4 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:

$$\begin{aligned}\mathrm{nat}(n) \times (\mathrm{nat}(n) \times (\mathrm{c}(\mathrm{size}(\mathsf{java.lang.Object}, 1)) + \mathrm{c}(\mathrm{size}(\mathsf{java.lang.Object}, 2))) + \\ \mathrm{nat}(n) \times (\mathrm{c}(\mathrm{size}(\mathsf{java.lang.Object}, 1)) + \mathrm{c}(\mathrm{size}(\mathsf{java.lang.Object}, 2))) + \\ \mathrm{nat}(n) \times (\mathrm{c}(\mathrm{size}(\mathsf{java.lang.Object}, 1)) + \mathrm{c}(\mathrm{size}(\mathsf{java.lang.Object}, 2))))\end{aligned}$$

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 in essence, but the formula is reordered and reduced to a more user friendly expression. 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. The expression above is transformed, and listed below. One can now easily see that the bound is quadratic. This simplification is built into ResAna and applied to all user-visible expressions. The transformed expression:

$$6n^2 \times \mathrm{size}(\mathsf{java.lang.Object})$$

4.2.5 Example

The run-time complexity, in terms of methods calls, of calculating the nth Fibonacci number is well known. The complexity of a function using a double recursion is $\mathcal{O}(\varphi^n)$ method calls, which is related to the golden ratio $\varphi = \frac{1+\sqrt{5}}{2}$. Standard textbooks on complexity

```
1 int fib(int n) {
2     new Object();
3     if (n < 2)
4         return n;
5     return fib(n-1) + fib(n-2);
6 }
```

listing 4.16 Adaptation of a double recursive function calculating Fibonacci numbers, allocating an object in each call.

```
1 int _fib(int a, int b, int n) {
2     new Object();
3     if (n <= 0)
4         return a;
5     return _fib(b, a+b, n-1);
6 }
7 int fib(int n) { return _fib(0, 1, n); }
```

listing 4.17 Adaptation of a single recursive function calculating Fibonacci numbers, allocating an object in each call.

analysis use overapproximation, which results in a complexity of $\mathcal{O}(2^n)$ for the same function. By adding an object allocation to each method call, the number of allocations equals the number of method calls. Therefore, in this example the heap consumption should be related to the run-time complexity. Our tool annotates this function, listed in listing 4.16, with the following bound, matching the expected bound:

$$(2^n - 1) \times \text{size}(\text{java.lang.Object})$$

The nth Fibonacci number can also be calculated by using a single recursion, for which the complexity is $\mathcal{O}(n)$ method calls. The code, like the previous example with added object allocations, is listed in listing 4.17. This single recursive function is annotated by our tool with the following bound, also matching the expected complexity bound:

$$(n + 1) \times \text{size}(\text{java.lang.Object})$$

4.3 Stack-size analysis

The proposed method of stack analysis requires global knowledge of the program, including its data. VeriFlux [HTS08], a static data-flow analyser, is used to provide this.

Analysis of **recursive methods** is a challenge in static evaluation of stack consumption. To deal with it, the stack-size analysis of VeriFlux 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 measured_by clauses in the JML syntax. 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 user interface of the tool. 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 for a given (recursive) method:

- a symbolic upper bound on the depth of recursion (i.e. a 'ranking function').
- 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 $\mathcal{O}(n)$, whereas the number of call belongs to $\mathcal{O}(2^n)$. 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 overapproximation 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 4.16. As expected, COSTA produces the ranking function $\mathbf{nat}(n-1)$. 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 $2 \times (2^{\mathbf{nat}(n-1)} - 1)$. This corresponds to the total number of nodes in the recursion tree.

A Java virtual machine 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 utilising 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.

4.3.1 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. 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. 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 optimisation purposes) and with native methods that are called through the Java Native Interface (JNI, used to access the kernel and platform-specific native libraries). VeriFlux does not address StackOverflowError s 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 behaviour 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 4.18 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 4.18. Before running COSTA, place this method in the abstracted class MyInteger, that contains only `toString()` method and the methods called from it. Create the abstracted versions of the classes String and StringIndexOutOfBoundsException, 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 validated by an accurate data-flow analysis of the method `toString()` using pen and paper.

4.3.2 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 a SCC with more than zero nodes are simply annotated by the size of the stack frame of the corresponding method invocation.

```
1 public static void main(String[] args) {
2   fib(21);
3 }
```

listing 4.19 Calling the `fib()` method from the `main()` method.

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()` 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, we proceed with the Fibonacci example, called from the main method in listing 4.19. 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. Performing the same method for `fib(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.

4.4 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, is available for free online. 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 their environment control system on board an aircraft for evaluating the CHARTER tool-chain. This system is responsible for air conditioning and air pressurisation. 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.

```
1 synchronized (lock) {
2   while (producedFrames != consumedFrames) {
3     try {
4       lock.wait();
5     } catch (Exception e) { }
6   }
7 }
```

listing 4.20 Example loop from CD_X located in the class immortal.FrameSynchronizer. No ranking function can be inferred for this loop.

```
1 for (int i = 0; i < arrays.length; i++) {
2   Object o = arrays[i];
3   if (o != null)
4     currentRetention += ((byte[])o).length;
5 }
```

listing 4.21 Example loop from CD_X located in the class heap.MemoryAllocator. The ranking function inferred by ResAna, and proven by KeY, is `arrays.length - i`.

	# loops	analysable	percentage
Hunt et al.	2	2	100%
DIANA	4	4	100%
CD_X	38	23	61%
Total	44	29	66%

table 4.22 Summary of the cases studied.

We also have conducted several case studies. For loop bound analysis three case studies, suggested by our CHARTER partners, were performed. These are parts of safety-critical Java systems. Roughly two-thirds of the loops found in the case studies are handled, i.e. ranking functions are inferred for these loops using our prototype and prove it using KeY. Almost all of the loops for which no ranking function could be inferred are loops for which the guard depends on a different thread, e.g. in listing 4.20. Analysing the temporal behaviour of such loops would require a fundamentally different analysis, which should take into account the code of all threads as well as the scheduling of threads. An example of a loop from the case studies for which a ranking function could be inferred and proved is given in listing 4.21. The results are shown in table 4.22. The case studies are:

collision detector case study The first case is the collision detector example from the paper "Provable Correct Loop bounds for Realtime Java Programs" by James Hunt et al [Hun+06]. This code stems from a safety-critical avionics application.

package DIANA This package [Sch+09] is developed in the project Distributed, equipment Independent environment for Advanced avioNics Applications (DIANA).

collision detector package CD_X The CD_X collision detector package is a publicly available real-time Java benchmark. It is described in [Kal+09].

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 Institute for Programming research and Algorithmics (IPA) school in 2013 on 'Software Engineering and Technology' [BvG-19], who found the tool to be very useful.

4.5 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 [KSE08; SKE07; SET11; SEK09; TSv09] and [GSE13]. 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 $n + m$.

4.5.1 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 [Hun+06]. What is 'missing' in the method is the automated inference of ranking functions, which ResAna supplies.

In [Alb+11b], 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 [Erm+07; Lok+09; Mic+08], 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. [Sha+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 a 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.

4.5.2 Timing analysis

There are a number of parallels of our work with timing 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 [Wil+08; Rei+07].

As memory allocators and cache policies are rather slow and unpredictable, the number and the amount of memory allocated have an impact on performance [Rei+07]. One has to resort to special means to alleviate these problems [Her+11]. 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.

4.5.3 Heap and stack size analysis

We have taken the COSTA system [Alb+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 [AGG10]. However, the authors claim that this extension does not require any changes to the recurrence solver PUBS. Thus, the techniques presented in section 4.2.1 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.

Deduced heap and stack bounds can be certified, based on the work in [DP11]. As an analysis can be unsound, or an implementation can be incorrect, verification of the bounds can be needed. An Isabelle/HOL proof needs to be performed once, and for each program a set of premisses needs to be verified. This method can also be applied for bounds deduced by hand.

In [MPS15], by iteratively reapplying the analysis with the previous bound, the result can be improved. The derived bounds are sound, and deal with both heap and stack memory usage for a functional language called Safe.

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, GNATStack, analyses object-oriented applications, automatically determining maximum stack usage on code that uses dynamic dispatching in Ada and C++.

In [Wan+10], 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.

4.6 Discussion

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.

5

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. Therefore it can be essential for the success of a system under development to be able to derive and optimise its resource consumption. Dynamic energy analysis is often applied to mitigate these issues. However, this is often hardware-specific and requires repetitive measurements using special equipment. Using similar techniques as used in chapter 4, a static analysis is introduced to analyse energy consumption instead of memory.*

We present a static analysis overapproximating the energy consumption based on an energy-aware Hoare logic. To achieve this, 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 (overapproximated) upper-bound for the energy-usage of the hardware controlled by the software.

based on [BvG-9]

energy analysis **symbolic** **Hoare logic** **overapproximation**

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.

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 offer a big improvement, potentially leading to improvements in the energy-efficiency of software.

Energy consumption may depend on hardware state, values of variables and bounds on the execution time. An energy analysis should incorporate these, in order to yield a realistic result.

Related work There is a large body of work on energy-efficiency of software. Most publications approach the problem on a high level, defining programming and design patterns for writing energy-efficient code, see e.g. [Alb10; Sax10; Ran10]. In [Bri+13], a modular design for energy-aware software is presented that is based on a series of rules on UML schemes. In [Coh+12] and [Sam+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 [Zhu+09] or in an **iterative compiler** [GCB05]. Petri-net based energy modelling techniques for embedded systems are proposed in [Jun+06] and [Nog+11].

Analyses for consumption of generic resources are built using recurrence relation solving [Alb+08], amortised analysis [HAH11], amortisation and separation logic [Atk10] and a Vienna Development Method style program logic [Asp+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 [Ker+13a], in which energy consumption of embedded software is analysed for specific architectures (SimpleScalar in [JML06], and [Ker+13a] 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 [Bri+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 RUM s 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 [Wil+08], size [SEK09], and loop bound and memory analysis as in chapter 4.

The focus of this chapter is on energy analysis. Energy consumption models of hardware components constitute the 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. chapter 4 or [PR04], and variable dependency information, e.g. [HTS08].

Essentially our approach is 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 software, however, has the central control. 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.

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.

The basic modelling and semantics are presented in section 5.1. Energy-awareness is added and the logic is presented in section 5.2. An example is given in section 5.3 and the soundness proof is outlined in section 5.4. The chapter is concluded in section 5.5.

5.1 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 programming language, with (energy) semantics, in which hardware components can be explicitly controlled. The hardware components are modelled in such a way that only the relevant information about the energy consumption and the behaviour influencing the software is present.

5.1.1 Language

Our analysis is performed on the simple while language ECA. The grammar for our language is defined in listing 5.1. This language is used 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,

```
⟨fun-def⟩    ::= 'function' ⟨function-name⟩ '(' ⟨var⟩ ')' 'begin' ⟨expr⟩ 'end'

⟨bin-op⟩     ::= '+' | '-' | '*' | '>' | '>=' | '==' | '!=' | '<=' | '<' | 'and' | 'or'

⟨expr⟩       ::= ⟨const⟩ | ⟨var⟩
               | ⟨var⟩ ':=' ⟨expr⟩ | ⟨expr⟩ ⟨bin-op⟩ ⟨expr⟩
               | ⟨component⟩ '.' ⟨function-name⟩ '(' ⟨expr⟩ ')'
               | ⟨function-name⟩ '(' ⟨expr⟩ ')'
               | ⟨stmt⟩ ';' ⟨expr⟩

⟨stmt⟩       ::= 'skip' | ⟨stmt⟩ ';' ⟨stmt⟩ | ⟨expr⟩
               | 'if' ⟨expr⟩ 'then' ⟨stmt⟩ 'else' ⟨stmt⟩ 'end'
               | 'while' '[' ⟨bound⟩ ']' ⟨expr⟩ 'do' ⟨stmt⟩ 'end'
               | ⟨fun-def⟩ ⟨stmt⟩
```

listing 5.1 Grammar for the ECA language, with the while rule.

for instance, using conventions about certain memory regions that will map to certain hardware devices. Functions on components have one argument and always return a value. The notation $C.f$ will refer to a function `f()` operating on a component `C`.

5.1.2 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 $\mathbf{s}$ 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 $\mathbf{rv}_f$ that produces the return value and a function δ_f that updates the internal state of the component. Both functions are functions over the state variables. The update function δ_f and the return value function $\mathbf{rv}_f$ take the state s and the argument a passed to the component function and return respectively the new state of the component and the return value. These functions are part of the function environment Δ (which is extended later on with language functions). Each component C 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.

5

5.1.3 Regular semantics

Standard, non-energy-aware semantics can be defined for our language. Full semantics are listed in figure 5.2, and can also be found together with more explanation in our technical report [BvG-20]. The assignment rule **sAssign** and the component function call rule **sCmpF** are given below to illustrate the notation and the way of handling components. The rules are defined over triples $\langle S, \sigma, \Gamma \rangle$ and $\langle e, \sigma, \Gamma \rangle$ with respectively a program statement S or a expression e, the program state function σ and the component state environment Γ. The **program** state function returns for every variable its actual value. Δ is an environment of function definitions. $\sigma[x_i \leftarrow n]$ is used as notation for substitution.

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma \rangle \xrightarrow{e} \langle n, \sigma', \Gamma' \rangle}{\Delta \vdash \langle x := e_1, \sigma, \Gamma, t \rangle \xrightarrow{e} \langle n, \sigma'[x \leftarrow n], \Gamma' \rangle} \text{(sAssign)}$$

$$\frac{}{\Delta \vdash \langle c, \sigma, \Gamma\rangle \xrightarrow{e} \langle c, \sigma, \Gamma\rangle}\ (\text{sConst}) \qquad \frac{}{\Delta \vdash \langle x, \sigma, \Gamma\rangle \xrightarrow{e} \langle \sigma(x), \sigma, \Gamma\rangle}\ (\text{sVar})$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma', \Gamma'\rangle \qquad \Delta \vdash \langle e_2, \sigma', \Gamma'\rangle \xrightarrow{e} \langle m, \sigma'', \Gamma''\rangle}{\Delta \vdash \langle e_1 \boxdot e_2, \sigma, \Gamma\rangle \xrightarrow{e} \langle n \overline{\boxdot} m, \sigma'', \Gamma''\rangle}\ (\text{sBinOp})$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma', \Gamma'\rangle}{\Delta \vdash \langle x := e_1, \sigma, \Gamma, t\rangle \xrightarrow{e} \langle n, \sigma'[x \leftarrow n], \Gamma'\rangle}\ (\text{sAssign})$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle a, \sigma', \Gamma'\rangle}{\Delta, C.f = (\delta_f, \text{rv}_f) \vdash \langle C.f(e_1), \sigma, \Gamma\rangle \xrightarrow{e} \langle \text{rv}_f(\Gamma'(C), a), \sigma', \Gamma'[C \leftarrow \delta_f(\Gamma'(C), a)]\rangle}\ (\text{sCmpF})$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle a, \sigma', \Gamma'\rangle \qquad \Delta \vdash \langle e_f, [x_f \leftarrow a], \Gamma'\rangle \xrightarrow{e} \langle n, \sigma'', \Gamma''\rangle}{\Delta, f = (e_f, x_f) \vdash \langle f(e_1), \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma', \Gamma''\rangle}\ (\text{sCallF})$$

$$\frac{\Delta \vdash \langle S_1, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma', \Gamma'\rangle \qquad \Delta \vdash \langle e_1, \sigma', \Gamma'\rangle \xrightarrow{e} \langle n, \sigma'', \Gamma''\rangle}{\Delta \vdash \langle\ S_1, e_1\ , \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma'', \Gamma''\rangle}\ (\text{sExprConcat})$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma', \Gamma'\rangle}{\Delta \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma', \Gamma'\rangle}\ (\text{sExprAsStmt}) \qquad \frac{}{\Delta \vdash \langle \text{skip}, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma, \Gamma\rangle}\ (\text{sSkip})$$

$$\frac{\Delta \vdash \langle S_1, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma', \Gamma'\rangle \qquad \Delta \vdash \langle S_2, \sigma', \Gamma'\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}{\Delta \vdash \langle S_1; S_2, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}\ (\text{sStmtConcat})$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle 0, \sigma', \Gamma'\rangle \qquad \Delta \vdash \langle S_2, \sigma', \Gamma'\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}{\Delta \vdash \langle \text{if } e_1 \text{ then } S_1 \text{ else } S_2 \text{ end}, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}\ (\text{sIf-False})$$

$$\frac{n \neq 0 \qquad \Delta \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma', \Gamma'\rangle \qquad \Delta \vdash \langle S_1, \sigma', \Gamma'\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}{\Delta \vdash \langle \text{if } e_1 \text{ then } S_1 \text{ else } S_2 \text{ end}, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}\ (\text{sIf-True})$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle 0, \sigma', \Gamma'\rangle}{\Delta \vdash \langle \text{while}_{[b]}\ e_1 \text{ do } S_1 \text{ end}, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma', \Gamma'\rangle}\ (\text{sWhile-False})$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma', \Gamma'\rangle \qquad n \neq 0 \qquad \Delta \vdash \langle S_1; \text{while}_{[b]}\ e_1 \text{ do } S_1 \text{ end}, \sigma', \Gamma'\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}{\Delta \vdash \langle \text{while}_{[b]}\ e_1 \text{ do } S_1 \text{ end}, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}\ (\text{sWhile-True})$$

$$\frac{\Delta, f = (e_1, x) \vdash \langle S_1, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma', \Gamma'\rangle}{\Delta \vdash \langle \text{function } f(x) \text{ begin } e_1 \text{ end } S_1, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma', \Gamma'\rangle}\ (\text{sFuncDef})$$

figure 5.2 Regular semantics for the ECA language, without energy consumption semantics. Note that c stands for a ⟨*const*⟩ term, x for a ⟨*var*⟩ term, e for a ⟨*expr*⟩ term, S for a ⟨*stmt*⟩ term, $\boxdot$ for a ⟨*bin-op*⟩, f for a ⟨*function-name*⟩, and C for a ⟨*component*⟩.

The **sCmpF** rule shows the calculation of the new component state Γ. With $\mathbf{s}(\Gamma(C))$ we denote the state of component C in Γ. The reduction symbol $\xrightarrow{e}$ is used for expressions, which evaluate to a value, a new state function and a new component state environment. We use $\xrightarrow{s}$ for statements, which only evaluate to a new state function and component state environment.

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma \rangle \xrightarrow{e} \langle a, \sigma', \Gamma' \rangle}{\Delta, C.f = (\delta_f, \mathrm{rv}_f) \vdash \langle C.f(e_1), \sigma, \Gamma \rangle \xrightarrow{e} \langle \mathrm{rv}_f(\Gamma'(C), a), \sigma', \Gamma'[C \leftarrow \delta_f(\Gamma'(C), a)] \rangle} \text{(sCmpF)}$$

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).

5.2 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**.

5.2.1 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 $\mathbf{a}$ 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 s, we add τ to the component model, signifying the timestamp at which the component entered the current state. In the energy-aware component model of component C, the above

$$\frac{}{\Delta \vdash \langle c, \sigma, \Gamma, t\rangle \xrightarrow{e} \langle c, \sigma, \Gamma, t\rangle}\ \text{(eConst)} \qquad \frac{}{\Delta \vdash \langle x, \sigma, \Gamma, t\rangle \xrightarrow{e} \langle \sigma(x), \sigma, \Gamma, t\rangle}\ \text{(eVar)}$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma, t\rangle \xrightarrow{e} \langle n, \sigma', \Gamma', t'\rangle \qquad \Delta \vdash \langle e_2, \sigma', \Gamma', t'\rangle \xrightarrow{e} \langle m, \sigma'', \Gamma'', t''\rangle}{\Delta \vdash \langle e_1 \boxdot e_2, \sigma, \Gamma, t\rangle \xrightarrow{e} \langle n \,\overline{\boxdot}\, m, \sigma'', \Gamma'', t'' + t_{\boxdot}\rangle}\ \text{(eBinOp)}$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma, t\rangle \xrightarrow{e} \langle n, \sigma', \Gamma', t'\rangle}{\Delta \vdash \langle x := e_1, \sigma, \Gamma, t\rangle \xrightarrow{e} \langle n, \sigma'[x \leftarrow n], \Gamma', t' + t_{\text{assign}}\rangle}\ \text{(eAssign)}$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma, t\rangle \xrightarrow{e} \langle a, \sigma', \Gamma', t'\rangle \qquad \Gamma'[C \leftarrow \langle \delta_f(\mathrm{s}(\Gamma'(C)), a), \mathrm{a}(C) + E_{C.f} + \mathrm{td}(\Gamma'(C), t), t'\rangle] = \Gamma''}{\Delta, C.f = (\delta_f, \mathrm{rv}_{C.f}, t_{C.f}, E_{C.f}) \vdash \langle C.f(e_1), \sigma, \Gamma, t\rangle \xrightarrow{e} \langle \mathrm{rv}_{C.f}(\mathrm{s}(\Gamma'(C)), a), \sigma', \Gamma'', t' + t_{C.f}\rangle}\ \text{(eCmpF)}$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma, t\rangle \xrightarrow{e} \langle a, \sigma', \Gamma', t'\rangle \qquad \Delta \vdash \langle e_f, [x_f \leftarrow a], \Gamma', t'\rangle \xrightarrow{e} \langle n, \sigma'', \Gamma'', t''\rangle}{\Delta, f = (e_f, x_f) \vdash \langle f(e_1), \sigma, \Gamma, t\rangle \xrightarrow{e} \langle n, \sigma', \Gamma'', t''\rangle}\ \text{(eCallF)}$$

$$\frac{\Delta \vdash \langle S_1, \sigma, \Gamma, t\rangle \xrightarrow{s} \langle \sigma', \Gamma', t'\rangle \qquad \Delta \vdash \langle e_1, \sigma', \Gamma', t'\rangle \xrightarrow{e} \langle n, \sigma'', \Gamma'', t''\rangle}{\Delta \vdash \langle\ S_1 e_1\ , \sigma, \Gamma, t\rangle \xrightarrow{e} \langle n, \sigma'', \Gamma'', t''\rangle}\ \text{(eExprConcat)}$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma, t\rangle \xrightarrow{e} \langle n, \sigma', \Gamma', t'\rangle}{\Delta \vdash \langle e_1, \sigma, \Gamma, t\rangle \xrightarrow{s} \langle \sigma', \Gamma', t'\rangle}\ \text{(eExprAsStmt)} \qquad \frac{}{\Delta \vdash \langle \mathsf{skip}, \sigma, \Gamma, t\rangle \xrightarrow{s} \langle \sigma, \Gamma, t\rangle}\ \text{(eSkip)}$$

$$\frac{\Delta \vdash \langle S_1, \sigma, \Gamma, t\rangle \xrightarrow{s} \langle \sigma', \Gamma', t'\rangle \qquad \Delta \vdash \langle S_2, \sigma', \Gamma', t'\rangle \xrightarrow{s} \langle \sigma'', \Gamma'', t''\rangle}{\Delta \vdash \langle S_1; S_2, \sigma, \Gamma, t\rangle \xrightarrow{s} \langle \sigma'', \Gamma'', t''\rangle}\ \text{(eStmtConcat)}$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma, t\rangle \xrightarrow{e} \langle 0, \sigma', \Gamma', t'\rangle \qquad \Delta \vdash \langle S_2, \sigma', \Gamma', t'\rangle \xrightarrow{s} \langle \sigma'', \Gamma'', t''\rangle}{\Delta \vdash \langle \mathsf{if}\ e_1\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{end}, \sigma, \Gamma, t\rangle \xrightarrow{s} \langle \sigma'', \Gamma'', t'' + t_{\mathsf{if}}\rangle}\ \text{(eIf-False)}$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma, t\rangle \xrightarrow{e} \langle n, \sigma', \Gamma', t'\rangle \qquad n \neq 0 \qquad \Delta \vdash \langle S_1, \sigma', \Gamma', t'\rangle \xrightarrow{s} \langle \sigma'', \Gamma'', t''\rangle}{\Delta \vdash \langle \mathsf{if}\ e_1\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{end}, \sigma, \Gamma, t\rangle \xrightarrow{s} \langle \sigma'', \Gamma'', t'' + t_{\mathsf{if}}\rangle}\ \text{(eIf-True)}$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma, t\rangle \xrightarrow{e} \langle 0, \sigma', \Gamma', t'\rangle}{\Delta \vdash \langle \mathsf{while}_{[b]}\ e_1\ \mathsf{do}\ S_1\ \mathsf{end}, \sigma, \Gamma, t\rangle \xrightarrow{s} \langle \sigma', \Gamma', t' + t_{\mathsf{while}}\rangle}\ \text{(eWhile-False)}$$

$$\frac{n \neq 0 \qquad \Delta \vdash \langle e_1, \sigma, \Gamma, t\rangle \xrightarrow{e} \langle n, \sigma', \Gamma', t'\rangle \qquad \Delta \vdash \langle S_1; \mathsf{while}_{[b]}\ e_1\ \mathsf{do}\ S_1\ \mathsf{end}, \sigma', \Gamma', t' + t_{\mathsf{while}}\rangle \xrightarrow{s} \langle \sigma'', \Gamma'', t''\rangle}{\Delta \vdash \langle \mathsf{while}_{[b]}\ e_1\ \mathsf{do}\ S_1\ \mathsf{end}, \sigma, \Gamma, t\rangle \xrightarrow{s} \langle \sigma'', \Gamma'', t''\rangle}\ \text{(eWhile-True)}$$

$$\frac{\Delta, f = (e_1, x) \vdash \langle S_1, \sigma, \Gamma, t\rangle \xrightarrow{s} \langle \sigma', \Gamma', t'\rangle}{\Delta \vdash \langle \mathsf{function}\ f(x)\ \mathsf{begin}\ e_1\ \mathsf{end}\ S_1, \sigma, \Gamma, t\rangle \xrightarrow{s} \langle \sigma', \Gamma', t'\rangle}\ \text{(eFuncDef)}$$

figure 5.3 Energy-aware semantics. Note that c stands for a ⟨*const*⟩ term, x for a ⟨*var*⟩ term, e for a ⟨*expr*⟩ term, S for a ⟨*stmt*⟩ term, $\boxdot$ for a ⟨*bin-op*⟩, f for a ⟨*function-name*⟩, and C for a ⟨*component*⟩.

mentioned variables are stored as a tuple t of $\langle \mathbf{s}, \mathbf{a}, \tau \rangle$. We use $\mathbf{s}(t)$ to retrieve the state, $\mathbf{a}(t)$ to retrieve the energy consumed until the last state change, and $\tau(t)$ to retrieve the timestamp of the previous state change.

We assume that each component has a constant **power draw** while in a state. Therefore, the component model function ϕ_C maps component states onto the corresponding power draw, independent of time. To calculate the power consumed while in a certain state we define the $\mathbf{td}$ function, with as arguments the component and the current timestamp:

$$\mathrm{td}(C, t) = \phi_C(\mathbf{s}(C)) \cdot (t - \tau(C))$$

We model **incidental energy usage** associated with a component function $C.f$ with the constant $E_{C.f}$. 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 the semantic rule **eCmpF** for component functions as defined below, with $\delta_{C.f}, \mathrm{rv}_{C.f}, t_{C.f}$ representing the state update function, the return value function, and the time it costs to execute this component function. These variables are expected to be part of the environment Δ.

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma, t \rangle \xrightarrow{e} \langle a, \sigma', \Gamma', t' \rangle \qquad \Gamma'[C \leftarrow \langle \delta_f(\mathbf{s}(\Gamma'(C)), a), \mathbf{a}(C) + E_{C.f} + \mathrm{td}(\Gamma'(C), t), t' \rangle] = \Gamma''}{\Delta, C.f = (\delta_f, \mathrm{rv}_{C.f}, t_{C.f}, E_{C.f}) \vdash \langle C.f(e_1), \sigma, \Gamma, t \rangle \xrightarrow{e} \langle \mathrm{rv}_{C.f}(\mathbf{s}(\Gamma'(C)), a), \sigma', \Gamma'', t' + t_{C.f} \rangle} \text{ (eCmpF)}$$

Note the addition of the incidental energy consumption $E_{C.f}$ and the time dependent energy consumption $\mathrm{td}(\Gamma'(C), t)$ to the energy accumulator $\mathbf{a}(C)$, the increment of the global time with t_f and the update of the component timestamp $\tau(C)$. Evaluation by $\rightarrow$ in the energy-aware semantics extends the regular semantics with a timestamp and an energy accumulator, which are used to calculate the total energy consumption of the evaluation ($\mathbf{a}_{system}$ as defined below). The full energy-aware semantics are given in figure 5.3.

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 **eCmpF** 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 $\mathbf{td}$ function for each component. The total energy consumption of the system can be calculated at any time as follows:

$$\mathrm{a}_{\mathrm{system}}(\Gamma, t) = \sum_{C} \mathrm{a}(\Gamma(C)) + \mathrm{td}(\Gamma(C), t)$$

To be able to calculate time-dependent consumption, we need a timing analysis. Depending on your application and desired precision, a Worst-Case Execution Time (WCET) can be integrated, possible in an external fashion. However, we keep it simple for now. Each construct in the language therefore has an associated execution time, for instance $t_{\mathsf{if}}, t_{\mathrm{assign}}, t_{\mathsf{while}}$ and $t_{\boxdot}$ with $\boxdot$ signifying an operation in $\langle \textit{bin-op} \rangle$.

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 t_{assign} for the time it takes to perform an assignment.

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma, t \rangle \xrightarrow{e} \langle n, \sigma', \Gamma', t' \rangle}{\Delta \vdash \langle x := e_1, \sigma, \Gamma, t \rangle \xrightarrow{e} \langle n, \sigma'[x \leftarrow n], \Gamma', t' + t_{\mathrm{assign}} \rangle} \text{(eAssign)}$$

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 t, a, 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.

5.2.2 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 fix point 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 [BvG-20] 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. The earliest timestamp therefore 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 greater 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 $s_1 > s_2 \rightarrow \delta(s_1) > \delta(s_2)$. For this reason, δ_f cannot depend on the arguments of f. To signify this, we will use $\delta(s)$ in the logic, instead of $\delta(s, a)$. As a result, component models cannot influence each other. Our soundness proof (theorem 2 in section 5.4) requires this assumption.

5

5.2.3 Severeness of model restrictions

There are several restrictions to the modelling that may seem far from reality:

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.

components are modelled as 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.

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.

limited dependencies of the component state functions The arguments of a function in the ECA language can not influence the outcome of applying the corresponding state function δ. In other words, the effect of component state functions cannot depend on the arguments of the function in models. 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.

An alternative approach is presented in chapter 6, which relieves most of these restrictions. This approach is precise, as opposed to the overapproximation as proposed in this chapter.

5.2.4 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 5.4. These rules are deterministic; at each moment only one rule can be applied.

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: $\textsf{while}_{[b]}$. 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 b 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 5.4 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: $\{\Gamma; t; \rho\}\ S\ \{\Gamma'; t'; \rho'\}$, where Γ is the set of all energy aware component states, t is the global time and ρ represents the symbolic state environment retrieved from the earlier standard analysis. 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 **aCmpF** rule uses the $\mathbf{td}(\Gamma(C), t)$ 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 $\mathbf{process\text{-}td}$. An overestimation $\mathbf{oe}$ 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 $\mathbf{wci}$. 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 construc-

$$\frac{}{\{\Gamma;t;\rho\}\ n\ \{\Gamma;t;\rho\}}\ \text{(aConst)} \qquad \frac{}{\{\Gamma;t;\rho\}\ x\ \{\Gamma;t;\rho\}}\ \text{(aVar)}$$

$$\frac{\{\Gamma;t;\rho\}\ e_1\ \{\Gamma_1;t_1;\rho_1\} \qquad \{\Gamma_1;t_1;\rho_1\}\ e_2\ \{\Gamma_2;t_2;\rho_2\}}{\{\Gamma;t;\rho\}\ e_1 \boxdot e_2\ \{\Gamma_2;t_2 + t_{\boxdot};\rho_2\}}\ \text{(aBinOp)}$$

$$\frac{\{\Gamma;t;\rho\}\ e\ \{\Gamma_1;t_1;\rho_1\}}{\{\Gamma;t;\rho\}\ x := e\ \{\Gamma_1;t_1 + t_{\text{assign}};\rho_2\}}\ \text{(aAssign)}$$

$$\frac{\Gamma[C \leftarrow \langle \delta_{C.f}(\mathrm{s}(\Gamma(C))), \mathrm{a}(\Gamma(C)) + E_{C.f} + \mathrm{td}(\Gamma(C), t), t\rangle] = \Gamma_1}{\{\Gamma;t;\rho\}\ C.f(args)\ \{\Gamma_1;t + t_{C.f};\rho\}}\ \text{(aCmpF)}$$

$$\frac{\begin{array}{c}\{\Gamma;t;\rho\}\ e\ \{\Gamma_1;t_1;\rho_1\} \\ e = a \in \rho \qquad \{\Gamma_1;t_1;\rho_1[x' \leftarrow a]\}\ e_1[x_f \leftarrow x']\ \{\Gamma_2;t_2;\rho_2\} \qquad x' \text{ fresh in } e_1\end{array}}{\Delta, f = (e_1, x_f) \vdash \{\Gamma;t;\rho\}\ f(e)\ \{\Gamma_2;t_2;\rho_2\}}\ \text{(aCallF)}$$

$$\frac{}{\{\Gamma;t;\rho\}\ \mathsf{skip}\ \{\Gamma;t;\rho\}}\ \text{(aSkip)} \qquad \frac{\{\Gamma\}\ S\ \{\Gamma_1\} \qquad \{\Gamma_1\}\ e\ \{\Gamma_2\}}{\{\Gamma\}\ S, e\ \{\Gamma_2\}}\ \text{(aExprConcat)}$$

$$\frac{\{\Gamma;t;\rho\}\ S_1\ \{\Gamma_1;t_1;\rho_1\} \qquad \{\Gamma_1;t_1;\rho_1\}\ S_2\ \{\Gamma_2;t_2;\rho_2\}}{\{\Gamma;t;\rho\}\ S_1; S_2\ \{\Gamma_2;t_2;\rho_2\}}\ \text{(aConcat)}$$

$$\frac{\begin{array}{cc}\{\Gamma;t;\rho\}\ e\ \{\Gamma_1;t_1;\rho_1\} & \begin{array}{c}\{\Gamma_1;t_1 + t_{\text{if}};\rho_1\}\ S_1\ \{\Gamma_2;t_2;\rho_2\} \\ \{\Gamma_1;t_1 + t_{\text{if}};\rho_1\}\ S_2\ \{\Gamma_3;t_3;\rho_3\}\end{array}\end{array}}{\{\Gamma;t;\rho\}\ \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{end}\ \{\mathrm{lub}(\Gamma_2,\Gamma_3); \max(t_2,t_3); \rho_4\}}\ \text{(aIf)}$$

$$\frac{\Gamma_1 = \text{process-td}(\Gamma, t) \qquad \{\mathrm{wci}(\Gamma_1, e, S); t; \rho\}\ e\ \{\Gamma_2; t_1; \rho_1\} \qquad \{\Gamma_2; t_1 + t_{\text{while}}; \rho_1\}\ S\ \{\Gamma_3; t_2; \rho_2\}}{\{\Gamma;t;\rho\}\ \mathsf{while}_{[b]}\ e\ \mathsf{do}\ S\ \mathsf{end}\ \{\mathrm{oe}(\Gamma_1, t, \Gamma_3, t_2, b); t + b \cdot (t_2 - t); \rho_3\}}\ \text{(aWhile)}$$

figure 5.4 Energy analysis rules.

tion **fix**. The fix point 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 5.4, 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. It performs the analysis on S, then considers only the effects on C. If there are nested loops or conditionals, the effects on the state of C 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 Fix point function $\mathbf{fix}$. 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 fix point 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 $\mathbf{wci}$. 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 fix point:

$$\mathbf{wci}_i(S) = \mathbf{lub}(\mathbf{ci}_i^0(S), \mathbf{ci}_i^1(S), \ldots, \mathbf{ci}_i^{\mathbf{fix}_i(S)}(S))$$

The global version $\mathbf{wci}(\Gamma, S)$ is defined by iteratively applying the $\mathbf{wci}_i(S)$ function to each component C in Γ.

4 Overestimation function $\mathbf{oe}$. This function overestimates the energy-aware output states of the while 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 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 t' and the iteration bound b. 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 calculate the consumption of the loop, calculating per component: $\mathbf{a}(\Gamma_{base}(C)) = \mathbf{a}(\Gamma_{in}(C)) + b \cdot (\mathbf{a}(\Gamma_{out}(C)) - \mathbf{a}(\Gamma_{in}(C)))$. We perform something similar for the execution time: $t_{ret} = t + b \cdot (t' - t)$.

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 number of iterations. 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 consumption before the loop would also be included in the analysis of the iteration. We introduce $\mathbf{process\text{-}td}(\Gamma, t)$, which adds $\mathbf{td}(C, t)$ to $\mathbf{a}(C)$ and sets $\tau(C)$ to t, for each component C in Γ.

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 $\mathbf{a}_{system}(\Gamma_{end}, t_{end})$.

5.3 Wireless sensor node example

To illustrate our analysis, we model a wireless sensor node, which has a sensor S and a radio R. We analyse the energy usage of a program that repeatedly measures the sensor for ten seconds, then sends the measurement over the radio, shown in figure 5.5. The example illustrates both **time-dependent** (sensor) and **incidental** (radio) energy usage. We choose a highly abstract modelling to keep the example brief. A more elaborate example can be found in [Sch+14], where two algorithms are compared using an **implementation** of this analysis. A similar analysis of two algorithms is performed in [BvG-20] by hand.

```
1 while[n] n > 0 do
2   S.on();
3   ... some code taking 10 seconds ...
4   x := S.off();
5   R.send(x);
6   n := n - 1;
7 end
```

listing 5.5 Example program.

Modelling The sensor component S has two states: on and 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 a_{on}. There are two component functions, namely `on()` and `off()`, which switch between the two component states. The radio component R only has incidental energy consumption. It does not have a state. Its single component function is `send()`, which uses $t_{\mathrm{R.send}}$ time and $E_{\mathrm{R.send}}$ energy. We also need a timing model. For the sake of presentation, a very simply model is used in which only assignment takes time t_{assign}. The other language constructs are assumed to execute instantly.

Application of the semantics from figure 5.3 on the loop body results in a time t_{body} of $10 + t_{\mathrm{R.send}} + 2 \cdot t_{\mathrm{assign}}$ and an energy consumption $\mathrm{a}_{\mathrm{body}}$ of $10 \cdot \mathrm{a}_{\mathrm{on}} + E_{\mathrm{R.send}}$. Intuitively, the time and energy consumption of the whole loop are $n \cdot t_{\mathrm{body}}$ and $n \cdot \mathrm{a}_{\mathrm{body}}$.

Energy consumption analysis The analysis (figure 5.4) always starts with a symbolic state. Note that only the sensor component S has a state. We introduce the symbol $\mathrm{on}_0^{\mathrm{s}}$ for the symbolic start-state (on or off) of the sensor. We start the analysis with the while loop. To apply the **aWhile** rule, we first have to resolve the process-td and wci functions, then analyse the loop guard and body, then determine the final results with the oe function. The instantiated **aWhile** rule, with the t_{while} constant omitted because it is 0:

$$\frac{\begin{array}{c}\Gamma_1 = \text{process-td}(\Gamma_0, t_0) \\ \{\mathrm{wci}(\Gamma_1, n > 0; S_{\mathrm{body}}); t_0; \rho_0\}\ n > 0\ \{\Gamma_2; t_1; \rho_1\} \qquad \{\Gamma_2; t_1; \rho_1\}\ S_{\mathrm{body}}\ \{\Gamma_{it}; t_{it}; \rho_{it}\}\end{array}}{\{\Gamma_0; t_0; \rho_0\}\ \mathrm{while}_{[n]}\ n > 0\ \mathrm{do}\ S_{it}\ \mathrm{end}\ \{\mathrm{oe}(\Gamma_1, t_0, \Gamma_{it}, t_{it}, \rho(n)); t_0 + n \cdot (t_{it} - t_0); \rho_{end}\}}\ \text{(aWhile)}$$

We first add time-dependent energy consumption and set timestamps to 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. R does not have a state, so we only need to add the time-dependent consumption of S: $\mathrm{td}(\mathsf{S}, t_0) = \phi_S(\mathrm{on}_0^{\mathrm{s}}) \cdot (t_0 - \tau_0(\mathsf{S}))$, where $\tau_0(\mathsf{S})$ is the symbolic timestamp when starting the analysis.

We must now find the worst-case iteration, using the wci function. For the S component we need the $\mathrm{ci}_{\mathrm{s}}(n > 0; S_{\mathrm{body}})$ function. As the other components do not have a state, $\mathrm{ci}_{imp}(n > 0; S_{\mathrm{body}})$ and $\mathrm{ci}_r(n > 0; S_{\mathrm{body}})$ are the identity function. The loop body sets the sensor state to s_{off}, independent of the start state. So, $\mathrm{ci}_{\mathrm{s}}(n > 0; S_{\mathrm{body}})$ always results in s_{off}. Now we can find the fix point. In the first iteration, we enter the loop with symbolic state $\mathrm{on}_0^{\mathrm{s}}$. In the second iteration, the loop is entered with s_{off}. In the third iteration, the loop is again entered with s_{off}, which is therefore the fix point. 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. The worst-case iteration is calculated by:

$$\mathrm{wci}_{\mathrm{s}}(n > 0; S_{\mathrm{body}}) = \mathrm{lub}(\mathrm{ci}_{\mathrm{s}}^0(n > 0; S_{\mathrm{body}}), \mathrm{ci}_{\mathrm{s}}^1(n > 0; S_{\mathrm{body}})) = \mathrm{on}_0^{\mathrm{s}}$$

We continue with the loop body, starting with a component function call to on():

$$\frac{\Gamma_3 = \Gamma_2[\mathsf{S} \leftarrow \langle \delta_{S.on}(\mathrm{s}(\Gamma_2(\mathsf{S}))), \mathrm{a}(\Gamma_2(\mathsf{S})) + \mathrm{td}(\Gamma(C), t_1), t_1 \rangle]}{\{\Gamma_2; t_1; \rho_1\}\ \mathsf{S.on()}\ \{\Gamma_3; t_1; \rho_2\}}\ (\mathrm{aCmpF})$$

There is no incidental energy consumption or time consumption associated with the call. However, we add the time-dependent energy consumption to the energy accumulator, by adding $\mathrm{td}(\mathsf{S}, t)$. Since we have just set $\tau(\mathsf{S})$ to t_0 and the evaluation of $n > 0$ costs no time, hence $t_1 = t_0$, $\mathrm{td}(\mathsf{S}, t_1)$ results in 0. The function $\delta_{\mathsf{R.on}}$ produces component state s_{on}. It also saves the current timestamp to the component state, in order to know when the last state transition occurred. For simplicity, we omit the application of **aConcat**.

After ten seconds of executing other statements (which we assume only cost time, no energy), the sensor is turned off. The call to the function `off()` returns the measurement, which is assigned to x. We therefore first apply the **aAssign** rule, which adds t_{assign} to the global time. We now apply the **aCmpF** rule to the right-hand side of the assignment, `S.off()`. This updates the state of the component to s_{off}. In order to determine the energy cost of the component being on for ten seconds, the rule calculates $\mathrm{td}(\mathsf{S}, t_2)$. Because $t_2 = \tau(\mathsf{S}) + 10$ and our model specifies a power draw of a_{on} for s_{on}, this results in $10 \cdot \mathrm{a}_{\mathrm{on}}$.

We apply the **aCmpF** rule again, this time for the send() function of the radio. As the transmission costs a fixed amount of energy, all time-dependent constants associated with transmitting are set to zero. The application of **aCmpF** only adds the incidental energy usage specified by $E_{\text{R.send}}$ and the constant time usage $t_{\text{R.send}}$. Finally, we apply **aBinOp**, which adds no costs, and the **aAssign** rule, which again adds t_{assign}.

Analysis of the worst-case iteration results in global time t_{it} and energy-aware component state environment Γ_{it}. Next we apply the overestimation function $\mathbf{oe}(\Gamma_1, t_0, \Gamma_{it}, t_{it}, \rho(n))$. 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 $\mathbf{on}^{\mathbf{s}}_0$). 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 $t_0 + 10 + t_{\text{R.send}} + 2 \cdot t_{\text{assign}}$. So, $\mathbf{oe}$ results in a global time t_{end} of $t_0 + n \cdot (10 + t_{\text{R.send}} + 2 \cdot t_{\text{assign}})$. Note that this is equal to the time consumption yielded by the energy-aware semantics.

A similar calculation is made for energy consumption, for each component, before we can calculate $\mathbf{a}_{system}$. In total, the $\mathbf{oe}$ function results in an energy usage of $\mathbf{a}_0 + n \cdot (10 \cdot \mathbf{a}_{\text{on}} + E_{\text{R.send}})$. However, we still need to add the time-dependent energy consumption for each component. This is where potential overestimation occurs in this example. Since R does not have a state, we only need to add the time-dependent consumption of S. After the analysis of the loop, the state of the sensor is overestimated as $\mathbf{on}^{\mathbf{s}}_0$. This leads to an overestimation when $\mathbf{on}^{\mathbf{s}}_0 = s_{\text{on}} \wedge n > 0$, and otherwise the result is equal to that of the energy-aware semantics. The added consumption is:

$$\text{td}(\text{S}, t_{\text{end}}) = \phi_S(\mathbf{on}^{\mathbf{s}}_0) \cdot (t_{\text{end}} - t_0) = \phi_S(\mathbf{on}^{\mathbf{s}}_0) \cdot n \cdot (10 + t_{\text{R.send}} + 2 \cdot t_{\text{assign}})$$

5.4 Soundness

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 overestimates the actual energy consumption. We present only the fundamental theorems, the full proof is given in [BvG-20]. To guarantee soundness of the final result, soundness of the annotations, loop bounds, and symbolic states is assumed. We first show that the logic overestimates time consumption.

THEOREM 5.1 ***Timing overestimation.***
If $\langle S, \sigma, \Gamma, t\rangle \xrightarrow{S} \langle \sigma', \Gamma', t'\rangle$, *then for any* $\{\Gamma; t; \rho\}\ S\ \{\Gamma_1; t_1; \rho_1\}$ *holds that* $t_1 \geq t'$.

PROOF 5.1 *Theorem 1 derives from the property that the analysis does not depend on the timestamp in the precondition. For* $\{\Gamma_1; t_1; \rho_1\}\ S\ \{\Gamma_2; t_2; \rho_2\}$, *the duration* $t_2 - t_1$ *always overestimates 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 overestimation are the rules* ***aIf*** *and* ***aWhile****. The* ***aIf*** *rule computes a maximum between the final timestamps of its branches. In the* ***aWhile*** *rule, the execution time of one iteration of the loop is overestimated and multiplied by the loop bound, itself an overestimation of the number of iterations of the loop.* □

Overestimating the component state is fundamental for overestimating the total energy consumption. A larger component state requires more power and hence consumes more energy.

THEOREM 5.2 ***Component state overestimation.***
If $\{\Gamma; t; \rho\}\ S\ \{\Gamma_1; t_1; \rho_1\}$ *and* $\langle S, \sigma, \Gamma, t\rangle \xrightarrow{S} \langle \sigma', \Gamma', t'\rangle$ *then* $\Gamma_1 \geq \Gamma'$.

PROOF 5.2 *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 5.2.2).* □

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

THEOREM 5.3 ***Soundness.***
If $\{t; \Gamma; \rho\}\ S\ \{t_1; \Gamma_1; \rho_1\}$ *and* $\langle S, \sigma, \Gamma, t\rangle \xrightarrow{S} \langle \sigma', \Gamma', t'\rangle$ *then* $\mathrm{a}_{system}(\Gamma_1, t_1) \geq \mathrm{a}_{system}(\Gamma', t')$.

PROOF 5.3 *By induction on the energy-aware semantics and the energy analysis rules. As theorem 1 ensures that the final timestamp is an overestimation of the actual time-consumption, hence the calculation of energy usage is based on an overestimated period of time. Next, 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* a_{system} *that* $\mathrm{a}_{system}(\Gamma_1, t_1) \geq \mathrm{a}_{system}(\Gamma', t')$. *The total energy consumption resulting from the analysis is larger than that of every possible execution of the analysed program.* □

5.5 Discussion

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. In the Software Analysis course at Radboud University Nijmegen students have used this approach (supported by an implementation of this analysis as described in [Sch+14]) for exercises, successfully modelling various algorithms and hardware components.

Validity There are several validity constraints to the technique discussed in this chapter. Although the derivation of bounds is sound, the quality of the bounds derived depends directly on the quality of the component models used. The restrictions on component models are severe, e.g. the energy consumption is assumed to be constant in every state of the component. This is in practice not true for most devices, e.g. the energy consumption can be a function over time. To correctly derive a bound, component models should be bound on their maximum energy usage by other means. To this end, alternative techniques can potentially be applied, such as model transformations and timed model checking, however this is left for future research.

The construction and validation of component models is hard, as there are several real world practical issues. If basing the component model on specifications from hardware vendors, all kinds of errors in the specification are transferred to the component model. Production errors in the hardware, and eventually the degradation of hardware, can induce erratic energy consumption behaviour that does not conform to the specification/-component model. Validating a component model with a test setup is hard, as energy is hard to measure. Small differences in energy consumption are hard to measure correctly, and outside condition like temperature can influence the results greatly. It differs significantly from the other resources discussed in the other chapters, which are measurable with great precision within a computer by the computer itself. Validating if the number of states a component model can be in, is the same number of states a hardware model can be in is a hard problem by itself. With powerful models, this validation process with real hardware can potentially last a life time, as least of the hardware in question.

There is another potential source of derived bounds not matching the actual bounds of a realistic situation. Compiler errors and optimisations can impact the (energy) behaviour of a source program greatly. The compiler has influence on the timing of high level language constructs. The timing constants used for these language constructs should match the upper bound it takes to execute those language constructs. Complicating matters even more is the complex design of modern processors executing the software. Even relatively small embedded microprocessors have features, like a register bypass, which impact the execution timing of statements significantly.

These constraints on modelling hardware components and validity implications should be lifted and investigated in future work to make the technique discusses applicable to general real-world problems. However, depending on the context and the precision needed, the current technique can already be applicable. If the hardware component is relatively simple, a conforming component model can be constructed. Another relevant area for the techniques discussed is to give feedback to a prospective programmer, so during construction of software he can optimise the energy consumption.

Future work Many future research directions can be envisioned: 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.

Another research direction is the use of timed automata for modelling non linear energy consumption. Such timed automata can induce a state change after a certain time autonomous, without interaction from software. This can be used to model energy consumption behaviour where the consumption is depending on the time passed. This way, one can approximate models where each state has an energy consumption function ϕ over time.

USING DEPENDENT TYPES TO DEFINE ENERGY AUGMENTED SEMANTICS OF PROGRAMS

6

ABSTRACT *A hardware parametric overapproximating approach to analyse energy consumption statically has been proposed in chapter 5. The approach involves creating energy models for hardware components and passing the hardware models as a parameter to the analysis of the software that controls the system. The foundation of that analysis is an energy aware Hoare logic. However, the Hoare logic approach suffers both from lack of compositionality and from significant overshoot in the derived bounds. For large systems compositionality is a key property for an analysis to be applicable in practice.*

This chapter presents a hardware-parametric, compositional and precise type system to derive energy consumption functions. These energy functions describe the energy consumption behaviour of hardware controlled by the software. This type system has the potential to predict energy consumptions of algorithms and hardware configurations, which can be used on design level or for optimisation.

We instantiate the derived consumption functions for each function call, making the system composable. This way a library can be pre-analysed and annotated. A feature missing from the original ECA language was recursion, but using these same signatures, recursion is supported in this alternative approach.

based on [BvG-14]

energy analysis **program transformation** **dependent types** **precise**

Green computing is an important emerging field within computer science. Much attention is devoted to developing energy-efficient systems, with a traditional focus on hardware. However, this hardware is controlled by software, which therefore has an energy-footprint as well.

A few options are available to a programmer who wants to write energy-efficient code. The programmer can look for programming guidelines and design patterns, which generally produce more energy-efficient programs, e.g. [Sax10]. Then, he/she might make use of a compiler that optimises for energy-efficiency, e.g. [Zhu+09]. If the programmer is lucky, there is an energy analysis available for his specific platform, such as [JML06] for a processor that is modeled in SimpleScalar (this only analyses the energy consumption of the processor).

However, for most platforms this is not a viable option. In that case, the programmer might use dynamic analysis with a measurement set-up. This, however, is not a trivial task and requires a complex set-up [Jag+16; Fer+13]. Moreover, it only yields information for a specific benchmark [Mog+15]. Nevertheless, these approaches are always applicable. However we envision an approach that yields additional insight to the programmer in a predictive manner.

Our approach We propose a dependent type system, which can be used to analyse energy consumption of software, with respect to a set of hardware component models. Such models can be constructed once, using a measurement set-up. Alternatively, they might be derived from hardware specifications. This type system is precise, in the sense that no overapproximation is used. By expressing energy analysis as a dependent type system, one can easily reuse **energy signatures** for functions which were derived earlier. This makes this new dependent type system modular and composable. Furthermore, the use of energy signatures that form a precise description of energy consumption can be a flexible, modular basis for various kinds of analyses and estimations using different techniques (e.g. lower or upper bound static analysis using approximations or average case analysis using dynamic profiling information).

The presented work is related to our results in chapter 5, where we present an overapproximating energy analysis for software, parametric with respect to a hardware model. That analysis is based on an energy aware Hoare logic and operates on a simple while-language. This previous work poses many limitations on hardware models in order to overapproximate and requires **re-analysis** of the body of a function **at each function call**.

The most important contributions of this chapter are:

- a **dependent type system** that, for the analysed code, captures both energy consumption and the effect on the state of hardware components.
- through the use of energy type signatures the system is **compositional**, making it more suitable for larger systems. By using this compositionality we can support **recursion** in the ECA language.
- the dependent type system derives **precise information**. This in contrast to the energy aware Hoare logic in chapter 5, which uses overapproximations for conditionals and loops.

- compared with chapter 5 many restrictions on component models and component functions are now removed. Effectively all that is required now, is that the power consumption of a device can be modelled by a finite state machine in which states correspond to level of power draw and state transitions correspond to changes of power draw level. State transitions occur due to (external) calls of component functions. The state transition itself may consume a certain amount of energy. This makes the system very suited for control software for various devices where the actuators are performed directly without the need for synchronisation.

- the dependently typed energy signatures can form a solid, modular basis for various approximations with various different static or dynamic techniques.

- the artificial environment ρ that was introduced in chapter 5 to incorporate user verified properties, is not needed in this dependent type setting. Using dependent types, such properties can be incorporated in a more **elegant** way.

We start with a discussion of the applications in section 6.1 to define the scope. Using that scope, the considered language and its (energy) semantics in described in section 6.2. Next, an basic dependent type system is introduced in section 6.3 to build the energy type system on. Continuing in section 6.4, we extend the basic type system into an energy type system. This extended type system derives from each statement and expression both an energy bound and a component state effect. To demonstrate the type system, we analyse and compare two example programs in section 6.5. We conclude this chapter with future work, and a discussion.

6.1 Applications

The foreseen application area of the proposed analysis is predicting the energy consumption of control systems, in which software controls a number of peripherals. This includes control systems in factories, cars, airplanes, smart-home applications, etc. Examples of hardware components range from a disk drive or sound card, to heaters, engines, motors and urban lighting. The proposed analysis can predict the energy consumption of multiple algorithms and/or different hardware configurations. The choice of algorithm or configuration may depend on the expected workload. This makes the proposed technique useful for both programmers and operators.

The used hardware models can be abstracted, making clear the relative differences between component-methods and component-states. This makes the proposed approach still applicable even when no final hardware component is available for basing the hardware model on, or this model is not yet created. We observe that many decisions are based on relative properties between systems.

The type system derives precise types, in a syntax-directed manner. Soundness and completeness can be proven by induction on the syntax structure of the program. This proof is similar to the proof in [BvG-20], but more straightforward due to the absence of approximations.

There are certain properties hardware models must satisfy. Foremost, the models are discrete. Implicit state changes by hardware components cannot be expressed. Energy consumption that gradually increases or decreases over time can therefore not be modelled directly. However, discrete approximations may be used. Compared to the Hoare logic in 5, many restrictions are not present in the proposed type system. Foremost, this type system does not have the limitation that state change cannot depend on the argument of a component function nor that the return value of a component function cannot depend on the state of the component. More realistic models can therefore be used.

The quality of the derived energy expressions is directly related to the quality of the used hardware models. We envision that, in many cases, relative consumption information is sufficient to support design decisions. Depending on the goal, it is possible to use multiple models for one and the same hardware component. For instance, if the hardware model is constructed as a worst-case model, the type system will produce worst-case information. Similarly one can use average-case models to derive average case information.

6.2 Hybrid modelling: language and semantics

Modern electronic systems typically consist of both hardware and software. As we aim to analyse energy consumption of such **hybrid** systems, we consider hardware and software in a single modelling framework. Software plays a central role, controlling various hardware components. Our analysis works on a simple, special purpose language, called ECA. This language has a special construct for calling functions on hardware components. It

```
⟨fun-def⟩      ::= 'function' ⟨function-name⟩ '(' ⟨var⟩ ')' 'begin' ⟨expr⟩ 'end'

⟨bin-op⟩       ::= '+' | '-' | '*' | '>' | '>=' | '==' | '!=' | '<=' | '<' | 'and' | 'or'

⟨expr⟩         ::= ⟨const⟩ | ⟨var⟩
                 | ⟨var⟩ ':=' ⟨expr⟩ | ⟨expr⟩ ⟨bin-op⟩ ⟨expr⟩
                 | ⟨component⟩ '.' ⟨function-name⟩ '(' ⟨expr⟩ ')'
                 | ⟨function-name⟩ '(' ⟨expr⟩ ')'
                 | ⟨stmt⟩ ';' ⟨expr⟩

⟨stmt⟩         ::= 'skip' | ⟨stmt⟩ ';' ⟨stmt⟩ | ⟨expr⟩
                 | 'if' ⟨expr⟩ 'then' ⟨stmt⟩ 'else' ⟨stmt⟩ 'end'
                 | 'repeat' ⟨expr⟩ 'begin' ⟨stmt⟩ 'end'
                 | ⟨fun-def⟩ ⟨stmt⟩
```

listing 6.1 Grammar for the ECA language, with the **repeat** construct.

is assumed that models exist for all the used hardware components, which model the energy consumption characteristics of these components, as well as the state changes induced by and return values of component function calls.

The ECA language is described in section 6.2.1. Modelling of hardware components is discussed in section 6.2.2. Energy-aware semantics for ECA are discussed in section 6.2.4.

6.2.1 ECA language

The grammar for the ECA language is defined in listing 6.1. We presume there is a way to differentiate between identifiers that represents variables ⟨*var*⟩, function names ⟨*function-name*⟩, components ⟨*component*⟩, and constants ⟨*const*⟩.

The only supported type in the ECA language is a signed integer. There are no explicit booleans. The value `0` is handled as **false**, any other value is handled as **true**. The absence of global variables and the by-value passing of variables to functions imply that functions do not have side-effects on the program state. Functions are statically scoped. Unlike the previous chapter, **recursion is supported**.

The language has an explicit construct for operations on hardware components (e.g. memory, storage or network devices). This allows us to reason about components in a straightforward manner. Functions on components have a single parameter and always return a value. The notation $C.f$ refers to a function `f()` operating on a component `C`.

The language supports a **repeat** construct, which makes the bound an obvious part of the loop and removes the need for evaluating the loop guard with every iteration (as with a **while**). The dlangrepeat is used for presentation purposes, without loss of generality, since the more commonly used **while** has the same expressive power.

A typical (predictive recursive descent) parser of this language is in the $\mathrm{LL}(2)$ class of parsers, with a small second pass. This second pass is needed, to avoid a possible infinite lookahead that is needed to differentiate between expressions and statements. During the first phase expressions and statements are combined as the same construct. The small post processing step differentiates between the two. This way the language can still be efficiently parsed in a simple manner.

6.2.2 Hardware component modelling

In order to reason about hybrid systems, we need to model hardware components. A component model consists of a **component state** and a set of **component functions** which operate on the component state. A component model must capture the behaviour of the hardware component with respect to energy consumption. Component models are used to derive dependent types signifying the energy consumption of the modelled hybrid system. The model can be based on measurements or detailed hardware specifications. Alternatively, a generic component model might be used (e.g. for a generic hard disk drive).

A component state 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 two functions: rv_f which produces the return value, and δ_f which updates the (component) state. A component can have multiple component functions. Any state change in components can only occur during a component function call and therefore is made explicit in the source code.

To model energy consumption, each component has a power draw, which depends on the component state. The function ϕ in the component model maps the component state to a power draw. The result of this function is used to calculate **time-dependent energy consumption**.

$$\frac{}{\Delta^s \vdash \langle c, \sigma, \Gamma\rangle \xrightarrow{e} \langle \mathcal{Z}(c), \sigma, \Gamma\rangle}\ (\text{sConst}) \qquad \frac{}{\Delta^s \vdash \langle x, \sigma, \Gamma\rangle \xrightarrow{e} \langle \sigma(x), \sigma, \Gamma\rangle}\ (\text{sVar})$$

$$\frac{\Delta^s \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma', \Gamma'\rangle \qquad \Delta^s \vdash \langle e_2, \sigma', \Gamma'\rangle \xrightarrow{e} \langle m, \sigma'', \Gamma''\rangle}{\Delta^s \vdash \langle e_1 \boxdot e_2, \sigma, \Gamma\rangle \xrightarrow{e} \langle n \,\overline{\boxdot}\, m, \sigma'', \Gamma''\rangle}\ (\text{sBinOp})$$

$$\frac{\Delta^s \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma', \Gamma'\rangle}{\Delta^s \vdash \langle x := e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma'[x \leftarrow n], \Gamma'\rangle}\ (\text{sAssign})$$

$$\frac{\Delta^s \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle a, \sigma', \Gamma'\rangle}{\Delta^s, C.f = (\delta_f, \mathrm{rv}_f) \vdash \langle C.f(e_1), \sigma, \Gamma\rangle \xrightarrow{e} \langle \mathrm{rv}_f(\Gamma'(C), a), \sigma', \Gamma'[C \leftarrow \delta_f(\Gamma'(C), a)]\rangle}\ (\text{sCmp})$$

$$\frac{\Delta^s, f = (e_f, x) \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle a, \sigma', \Gamma'\rangle \qquad \Delta^s, f = (e_f, x) \vdash \langle e_f, [x \mapsto a], \Gamma'\rangle \xrightarrow{e} \langle n, \sigma'', \Gamma''\rangle}{\Delta^s, f = (e_f, x) \vdash \langle f(e_1), \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma', \Gamma''\rangle}\ (\text{sCall})$$

$$\frac{\Delta^s \vdash \langle S_1, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma', \Gamma'\rangle \qquad \Delta^s \vdash \langle e_1, \sigma', \Gamma'\rangle \xrightarrow{e} \langle n, \sigma'', \Gamma''\rangle}{\Delta^s \vdash \langle S_1, e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma'', \Gamma''\rangle}\ (\text{sExprConcat})$$

$$\frac{\Delta^s \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma', \Gamma'\rangle}{\Delta^s \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma', \Gamma'\rangle}\ (\text{sExprAsStmt}) \qquad \frac{}{\Delta^s \vdash \langle \mathsf{skip}, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma, \Gamma\rangle}\ (\text{sSkip})$$

$$\frac{\Delta^s \vdash \langle S_1, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma', \Gamma'\rangle \qquad \Delta^s \vdash \langle S_2, \sigma', \Gamma'\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}{\Delta^s \vdash \langle S_1; S_2, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}\ (\text{sStmtConcat})$$

$$\frac{\Delta^s \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle 0, \sigma', \Gamma'\rangle \qquad \Delta^s \vdash \langle S_2, \sigma', \Gamma'\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}{\Delta^s \vdash \langle \mathsf{if}\ e_1\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{end}, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}\ (\text{sIf-False})$$

$$\frac{\Delta^s \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma', \Gamma'\rangle \qquad n \neq 0 \qquad \Delta^s \vdash \langle S_1, \sigma', \Gamma'\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}{\Delta^s \vdash \langle \mathsf{if}\ e_1\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{end}, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}\ (\text{sIf-True})$$

$$\frac{n \leq 0}{\Delta^s \vdash \langle (S_1, n), \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma, \Gamma\rangle}\ (\text{sRepeatBase})$$

$$\frac{n > 0 \qquad \Delta^s \vdash \langle S_1, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma', \Gamma'\rangle \qquad \Delta^s \vdash \langle (S_1, n-1), \sigma', \Gamma'\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}{\Delta^s \vdash \langle (S_1, n), \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}\ (\text{sRepeatLoop})$$

$$\frac{\Delta^s \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle n, \sigma', \Gamma'\rangle \qquad \Delta^s \vdash \langle (S_1, n), \sigma', \Gamma'\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}{\Delta^s \vdash \langle \mathsf{repeat}\ e_1\ \mathsf{begin}\ S_1\ \mathsf{end}, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma'', \Gamma''\rangle}\ (\text{sRepeat})$$

$$\frac{\Delta^s, f = (e_1, x) \vdash \langle S_1, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma', \Gamma'\rangle}{\Delta^s \vdash \langle \mathsf{function}\ f(x)\ \mathsf{begin}\ e_1\ \mathsf{end}\ S_1, \sigma, \Gamma\rangle \xrightarrow{s} \langle \sigma', \Gamma'\rangle}\ (\text{sFuncDef})$$

figure 6.2 Regular semantics. Note that c stands for a ⟨*const*⟩ term, x for a ⟨*var*⟩ term, e for a ⟨*expr*⟩ term, S for a ⟨*stmt*⟩ term, $\boxdot$ for a ⟨*bin-op*⟩, f for a ⟨*function-name*⟩, and C for a ⟨*component*⟩.

6.2.3 Regular semantics

We use a fairly standard semantics for our ECA language, to which we add energy consumption semantics later on. The semantics are listed in figure 6.2. We explain the used notation below.

The function environment Δ^s (**s** for semantics) contains both the aforementioned component function signatures (for now a pair of a state transition function δ and a return value function $\mathbf{rv}$) and function definitions (pairs of function body e and parameter x). Component states are collected in the component environment Γ. We use the following notation for substitution: $\sigma[x \leftarrow a]$. With $[x \mapsto a]$, we construct a new environment in which x has the value a. We differentiate two kinds of reductions. Expressions reduce from a triple of the expression, program state σ and component state environment Γ, to a triple of a value, a new program state and a new component state environment, with the $\xrightarrow{e}$ operator. As statements do not result in a value, they reduce from a triple of the statement, σ, and Γ to a pair of a new program state and a new component state environment, with the $\xrightarrow{s}$ operator.

There are three rules for the **repeat** loop. The one labelled **esRepeat** calculates the value of expression e, i.e. the number of iterations for the loop. The **esRepeatLoop** rule then handles each iteration, until the number of remaining iterations is 0. At that point, the evaluation of the loop is ended with the **esRepeatBase** rule. To differentiate it from the normal evaluation rules we use a tuple of the statement to be evaluated and the number of times the statement should be evaluated.

6.2.4 Energy-aware semantics

The regular semantics are extended to include energy consumed by the components while running the program. To this end both the reduction functions $\xrightarrow{e}$ and $\xrightarrow{s}$ are extended so they also return the energy consumed.

Components consume energy in two distinct ways. A component function might directly induce a **time-independent** energy consumption. Apart from that, it can change the state of the component, affecting its **time-dependent** energy consumption. To be able to calculate time-dependent consumption, we need a basic timing analysis. Each construct in the language therefore has an associated execution time, for instance t_{if}. Every component

$$\frac{}{\Delta^{es};\Phi \vdash \langle c,\sigma,\Gamma\rangle \xrightarrow{e} \langle \mathcal{Z}(c),\sigma,\Gamma,\Phi(\Gamma)\cdot t_{\text{const}}\rangle}\ (\text{esConst}) \qquad \frac{}{\Delta^{es};\Phi \vdash \langle x,\sigma,\Gamma\rangle \xrightarrow{e} \langle \sigma(x),\sigma,\Gamma,\Phi(\Gamma)\cdot t_{\text{var}}\rangle}\ (\text{esVar})$$

$$\frac{\Delta^{es};\Phi \vdash \langle e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle n,\sigma',\Gamma',E'\rangle \qquad \Delta^{es};\Phi \vdash \langle e_2,\sigma',\Gamma'\rangle \xrightarrow{e} \langle m,\sigma'',\Gamma'',E''\rangle}{\Delta^{es};\Phi \vdash \langle e_1 \boxdot e_2,\sigma,\Gamma\rangle \xrightarrow{e} \langle n \boxdot m,\sigma'',\Gamma'',E'+E''+\Phi(\Gamma'')\cdot t_{\boxdot}\rangle}\ (\text{esBinOp})$$

$$\frac{\Delta^{es};\Phi \vdash \langle e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle n,\sigma',\Gamma',E'\rangle}{\Delta^{es};\Phi \vdash \langle x := e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle n,\sigma'[x \leftarrow n],\Gamma',E'+\Phi(\Gamma')\cdot t_{\text{assign}}\rangle}\ (\text{esAssign})$$

$$\frac{\Delta^{es};\Phi \vdash \langle e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle a,\sigma',\Gamma',E'\rangle}{\Delta^{es},C.f=(\delta_f,\text{rv}_f,t_f,E_f);\Phi \vdash \langle C.f(e_1),\sigma,\Gamma\rangle \xrightarrow{e} \langle \text{rv}_f(\Gamma'(C),a),\sigma',\Gamma'[C \leftarrow \delta_f(\Gamma'(C),a)],E'+\Phi(\Gamma')\cdot t_f+E_f\rangle}\ (\text{esCmp})$$

$$\frac{\Delta^{es},f=(e_f,x);\Phi \vdash \langle e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle a,\sigma',\Gamma',E'\rangle \qquad \Delta^{es},f=(e_f,x);\Phi \vdash \langle e_f,[x \mapsto a],\Gamma'\rangle \xrightarrow{e} \langle n,\sigma'',\Gamma'',E''\rangle}{\Delta^{es},f=(e_f,x);\Phi \vdash \langle f(e_1),\sigma,\Gamma\rangle \xrightarrow{e} \langle n,\sigma',\Gamma'',E'+E''\rangle}\ (\text{esCall})$$

$$\frac{\Delta^{es};\Phi \vdash \langle S_1,\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma',\Gamma',E'\rangle \qquad \Delta^{es};\Phi \vdash \langle e_1,\sigma',\Gamma'\rangle \xrightarrow{e} \langle n,\sigma'',\Gamma'',E''\rangle}{\Delta^{es};\Phi \vdash \langle S_1,e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle n,\sigma'',\Gamma'',E'+E''\rangle}\ (\text{esExprConcat})$$

$$\frac{\Delta^{es};\Phi \vdash \langle e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle n,\sigma',\Gamma',E'\rangle}{\Delta^{es};\Phi \vdash \langle e_1,\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma',\Gamma',E'\rangle}\ (\text{esExprAsStmt}) \qquad \frac{}{\Delta^{es};\Phi \vdash \langle \text{skip},\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma,\Gamma,0\rangle}\ (\text{esSkip})$$

$$\frac{\Delta^{es};\Phi \vdash \langle S_1,\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma',\Gamma',E'\rangle \qquad \Delta^{es};\Phi \vdash \langle S_2,\sigma',\Gamma'\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E''\rangle}{\Delta^{es};\Phi \vdash \langle S_1;S_2,\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E'+E''\rangle}\ (\text{esStmtConcat})$$

$$\frac{\Delta^{es};\Phi \vdash \langle e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle 0,\sigma',\Gamma',E'\rangle \qquad \Delta^{es};\Phi \vdash \langle S_2,\sigma',\Gamma'\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E''\rangle}{\Delta^{es};\Phi \vdash \langle \text{if } e_1 \text{ then } S_1 \text{ else } S_2 \text{ end},\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E'+\Phi(\Gamma')\cdot t_{\text{if}}+E''\rangle}\ (\text{esIf-False})$$

$$\frac{\Delta^{es};\Phi \vdash \langle e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle n,\sigma',\Gamma',E'\rangle \qquad n \neq 0 \qquad \Delta^{es};\Phi \vdash \langle S_1,\sigma',\Gamma'\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E''\rangle}{\Delta^{es};\Phi \vdash \langle \text{if } e_1 \text{ then } S_1 \text{ else } S_2 \text{ end},\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E'+\Phi(\Gamma')\cdot t_{\text{if}}+E''\rangle}\ (\text{esIf-True})$$

$$\frac{n \leq 0}{\Delta^{es};\Phi \vdash \langle (S_1,n),\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma,\Gamma,0\rangle}\ (\text{esRepeatBase})$$

$$\frac{n>0 \qquad \Delta^{es};\Phi \vdash \langle S_1,\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma',\Gamma',E'\rangle \qquad \Delta^{es};\Phi \vdash \langle (S_1,n-1),\sigma',\Gamma'\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E''\rangle}{\Delta^{es};\Phi \vdash \langle (S_1,n),\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E'+E''\rangle}\ (\text{esRepeatLoop})$$

$$\frac{\Delta^{es};\Phi \vdash \langle e_1,\sigma,\Gamma\rangle \xrightarrow{e} \langle n,\sigma',\Gamma',E'\rangle \qquad \Delta^{es};\Phi \vdash \langle (S_1,n),\sigma',\Gamma'\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E''\rangle}{\Delta^{es};\Phi \vdash \langle \text{repeat } e_1 \text{ begin } S_1 \text{ end},\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma'',\Gamma'',E'+E''\rangle}\ (\text{esRepeat})$$

$$\frac{\Delta^{es},f=(e_1,x) \vdash \langle S_1,\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma',\Gamma',E'\rangle}{\Delta^{es};\Phi \vdash \langle \text{function } f(x) \text{ begin } e_1 \text{ end } S_1,\sigma,\Gamma\rangle \xrightarrow{s} \langle \sigma',\Gamma',E'\rangle}\ (\text{esFuncDef})$$

figure 6.3 Energy-aware semantics. Note that c stands for a $\langle const\rangle$ term, x for a $\langle var\rangle$ term, e for a $\langle expr\rangle$ term, S for a $\langle stmt\rangle$ term, $\boxdot$ for a $\langle bin\text{-}op\rangle$, f for a $\langle function\text{-}name\rangle$, and C for a $\langle component\rangle$.

function f has a constant time consumption t_f that is part of its function signature, along with δ_f and $\mathbf{rv}_f$, as described in section 6.2.2. The pair stored in Δ^{es} (**es** for energy semantics) for each component function f is extended to also contain the execution time t_f, and an incidental energy consumption E_f. The full energy-aware semantics are given in figure 6.3.

A significant difference from the semantics as used in chapter 5 is the absence of global time. As at each step the power consumption is calculated (by calculating the power draw of each component times the duration), global timestamps are not needed.

As our system allows for a fixed number of components, known before the analysis is performed, we assume a global function Φ is present. This Φ sums the power draw for each known component, by applying the component power draw function ϕ to the component state as contained in Γ.

6.3 Basic dependent type system

Before we can introduce a dependent type system that can be used to derive energy consumption expressions of an ECA program, we need to define a standard dependent type system to reason about variable values. We separately introduce this for presentation purposes.

Note that, instead of direct expressions over input variables, we derive functions that calculate a value based on the concrete values of the input variables. This allows us to combine these functions using function composition and to reuse them as signatures for methods and parts of the code.

We will start with a series of definitions. A program state environment called $\mathbf{PState}$ is a function from a variable identifier to a value, i.e. $\mathbf{PState} : \mathbf{Var} \mapsto \mathbf{Value}$. Values are of type $\mathbb{Z}$, like the data type in ECA. A component state $\mathbf{CState}$ is collection of states of components that the component function can work on. A **value function** is a function that, given a (program and component) state, will yield a concrete value, i.e. its signature is $\mathbf{PState} \times \mathbf{CState} \rightarrow \mathbf{Value}$. A **state update function** is a function from $\mathbf{PState} \times \mathbf{CState}$ to $\mathbf{PState} \times \mathbf{CState}$. Such a function expresses changes to the state, caused by the execution of statements or expressions.

$$\frac{}{\Delta^v \vdash c : \langle\, \mathrm{const}_{\mathcal{N}(c)}\,,\ \mathrm{id}\,\rangle}\ (\text{btConst}) \qquad \frac{}{\Delta^v \vdash x : \langle\, \mathrm{lookup}_x\,,\ \mathrm{id}\,\rangle}\ (\text{btVar})$$

$$\frac{\Delta^v \vdash e_1 : \langle V_1, \Sigma_1\rangle \qquad \Delta^v \vdash e_2 : \langle V_2, \Sigma_2\rangle}{\Delta^v \vdash e_1 \boxdot e_2 : \langle V_1 \overline{\boxdot}\, (\Sigma_1 \ggg V_2), \Sigma_1 \ggg \Sigma_2\rangle}\ (\text{btBinOp})$$

$$\frac{\Delta^v \vdash e : \langle V, \Sigma\rangle}{\Delta^v \vdash x := e : \langle V, \Sigma \ggg \mathrm{assign}_x(V)\rangle}\ (\text{btAssign})$$

$$\frac{\Delta^v \vdash e : \langle V_{ex}, \Sigma_{ex}\rangle}{\begin{array}{l}\Delta^v,\ C.f = (x_f, V_f, \Sigma_f) \vdash \\ \quad C.f(e) : \langle \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg V_f,\ \mathrm{split}(\Sigma_{ex}, \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \Sigma_f)\rangle\end{array}}\ (\text{btCmp})$$

$$\frac{\Delta^v, f = (x_f, V_f, \Sigma_f) \vdash e : \langle V_{ex}, \Sigma_{ex}\rangle}{\begin{array}{l}\Delta^v,\ f = (x_f, V_f, \Sigma_f) \vdash \\ \quad f(e) : \langle \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathrm{subst}(V_f, \mathrm{rec}_V(f)),\ \mathrm{split}(\Sigma_{ex}, \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathrm{subst}(\Sigma_f, \mathrm{rec}_\Sigma(f)))\rangle\end{array}}\ (\text{btCall})$$

$$\frac{\Delta^v, f = (x_f) \vdash e : \langle V_{ex}, \Sigma_{ex}\rangle}{\begin{array}{l}\Delta^v,\ f = (x_f) \vdash \\ \quad f(e) : \langle \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathrm{rec}_V(f),\ \mathrm{split}(\Sigma_{ex}, \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathrm{rec}_\Sigma(f))\rangle\end{array}}\ (\text{btRec})$$

$$\frac{\Delta^v \vdash S : \Sigma_{st} \qquad \Delta^v \vdash e : \langle V_{ex}, \Sigma_{ex}\rangle}{\Delta^v \vdash S; e : \langle \Sigma_{st} \ggg V_{ex}, \Sigma_{st} \ggg \Sigma_{ex}\rangle}\ (\text{btExprConcat})$$

$$\frac{\Delta^v \vdash e : \langle V, \Sigma\rangle}{\Delta^v \vdash e : \Sigma}\ (\text{btExprAsStmt}) \qquad \frac{}{\Delta^v \vdash \mathsf{skip} : \mathrm{id}}\ (\text{btSkip})$$

$$\frac{\Delta^v \vdash S_1 : \Sigma_1 \qquad \Delta^v \vdash S_2 : \Sigma_2}{\Delta^v \vdash S_1; S_2 : \Sigma_1 \ggg \Sigma_2}\ (\text{btStmtConcat})$$

$$\frac{\Delta^v \vdash e : \langle V_{ex}, \Sigma_{ex}\rangle \qquad \Delta^v \vdash S_t : \Sigma_t \qquad \Delta^v \vdash S_f : \Sigma_f}{\Delta^v \vdash \mathsf{if}\ e\ \mathsf{then}\ S_t\ \mathsf{else}\ S_f\ \mathsf{end} : \mathrm{if}(V_{ex}, \Sigma_{ex} \ggg \Sigma_t, \Sigma_{ex} \ggg \Sigma_f)}\ (\text{btIf})$$

$$\frac{\Delta^v \vdash e : \langle V_{ex}, \Sigma_{ex}\rangle \qquad \Delta^v \vdash S : \Sigma_{st}}{\Delta^v \vdash \mathsf{repeat}\ e\ \mathsf{begin}\ S\ \mathsf{end} : \mathrm{repeat}^v(V_{ex}, \Sigma_{ex}, \Sigma_{st})}\ (\text{btRepeat})$$

$$\frac{\Delta^v, f = (x) \vdash e : \langle V_{ex}, \Sigma_{ex}\rangle \qquad \Delta^v, f = (x, V_{ex}, \Sigma_{ex}) \vdash S : \Sigma_{st}}{\Delta^v \vdash \mathsf{function}\ f(x)\ \mathsf{begin}\ e\ \mathsf{end}\ S : \Sigma_{st}}\ (\text{btFuncDef})$$

figure 6.4 Basic dependent type system. For expressions it yields a tuple of a **value function** and a **state update function**. For statements the type system only derives a **state update function**. Note that c stands for a $\langle const \rangle$ term, x for a $\langle var \rangle$ term, e for a $\langle expr \rangle$ term, S for a $\langle stmt \rangle$ term, $\boxdot$ for a $\langle bin\text{-}op \rangle$, f for a $\langle function\text{-}name \rangle$, and C for a $\langle component \rangle$.

To make the typing rules more clear, we explain a number of rules in more detail. The full typing rules can be found in figure 6.4. We will start with the variable access rule:

$$\frac{}{\Delta^v \vdash x : \langle\, \mathrm{lookup}_x\,,\, \mathrm{id}\,\rangle}\ (\mathrm{btVar})$$

All rules for evaluation of an expression return a tuple of a function returning the value of the expression when evaluated and a function that modifies the program state and component state. The former one captures the value, i.e. it is a state value function. The latter one captures the effect, i.e. it is a state update function. To access a variable, we return the **lookup** function (defined below), which is parametrised by the variable that is returned. Variable access does not affect the state, hence for that part the identity function **id** is returned.

The **lookup** function that the **btVar** rule depends on is defined as follows:

$$\begin{aligned}&\mathrm{lookup}_x : \mathrm{PState} \times \mathrm{CState} \to \mathrm{Value}\\&\mathrm{lookup}_x(ps, cs) = ps(x)\end{aligned}$$

Likewise we need to define a function for the **btConst** rule, dealing with constants:

$$\begin{aligned}&\mathrm{const}_x : \mathrm{PState} \times \mathrm{CState} \to \mathrm{Value}\\&\mathrm{const}_x(ps, cs) = x\end{aligned}$$

Before we can continue we need to introduce the $\ggg$ operator. We can compose state update functions with the $\ggg$ operator. Note that the composition is reversed with respect to standard mathematical function composition, in order to maintain the order of program execution. For now, T is $\mathrm{PState} \times \mathrm{CState}$. The $\ggg$ operator can be interpreted as: first apply the effect of the left operand, then execute the right operand on the resulting state. The $\ggg$ operator is defined as:

$$\begin{aligned}&\ggg : (\mathrm{PState} \times \mathrm{CState} \to \mathrm{PState} \times \mathrm{CState}) \times (\mathrm{PState} \times \mathrm{CState} \to T)\\&\qquad \to (\mathrm{PState} \times \mathrm{CState} \to T)\\&(A \ggg B)(ps, cs) = B(ps', cs') \text{ where } (ps', cs') = A(ps, cs)\end{aligned}$$

Moving on, we can explain the assignment rule below, which assigns a value expressed by expression e to variable x. As an assignment does not modify the value of the expression, the part capturing the value is propagated from the rule deriving the type of x. More interesting is the effect that captures the state change when evaluating the rule. This consists of first applying the state change Σ^v of evaluating the expression e and thereafter replacing the variable x with the result of e (as done by the **assign** function, defined below). This effect is reached with the $\ggg$ operator:

$$\frac{\Delta^v \vdash e : \langle V, \Sigma^v \rangle}{\Delta^v \vdash x := e : \langle V, \Sigma^v \ggg \mathbf{assign}_x(V) \rangle} \text{ (btAssign)}$$

The operator for assigning a new value to a program variable in the type environment:

$$\mathbf{assign}_x : (\text{PState} \times \text{CState} \to \text{Value}) \to (\text{PState} \times \text{CState} \to \text{PState} \times \text{CState})$$
$$\mathbf{assign}_x(V)(ps, cs) = (ps[x \mapsto V(ps, cs)], cs)$$

In order to support binary operations we need to define a higher order operator $\overline{\boxdot}$ where $\boxdot \in \{+, -, \times, \div, \ldots\}$. It evaluates the two arguments and combines the results using $\boxdot$. For now, T is **Value**.

$$\overline{\boxdot} : (\text{PState} \times \text{CState} \to T) \times (\text{PState} \times \text{CState} \to T) \to (\text{PState} \times \text{CState} \to T)$$
$$(A \,\overline{\boxdot}\, B)(ps, cs) = A(ps, cs) \boxdot B(ps, cs)$$

The binary operation rule can now be introduced. The pattern used in the rule reoccurs multiple times if two expressions (or statements) need to be combined. First the value function representing the first expression, V_1, is evaluated and is combined with the result representing the second expression, V_2, e.g. using the $\overline{\boxdot}$ operator. But this second value function needs to be evaluated in the correct context (state): evaluating the first expression can have side-effects and therefore change the program and component state, and influence the outcome of a value function (as the expression can include function calls, assignments, component function calls, etc). To evaluate V_2 in the correct context we first must apply the side effects of the first expressions using a state update function Σ_1, expressed as $\Sigma_1 \ggg V_2$. We can express the value function that represents the combination of two expressions as $V_1 \,\overline{\boxdot}\, (\Sigma_1 \ggg V_2)$. The complete definition:

$$\frac{\Delta^v \vdash e_1 : \langle V_1, \Sigma_1 \rangle \qquad \Delta^v \vdash e_2 : \langle V_2, \Sigma_2 \rangle}{\Delta^v \vdash e_1 \boxdot e_2 : \langle V_1 \overline{\boxdot} (\Sigma_1 \ggg V_2), \Sigma_1 \ggg \Sigma_2 \rangle} \text{ (btBinOp)}$$

Without explaining the rule in detail, we introduce the conditional operator. The **if** operator captures the behaviour of a conditional inside the dependent type. The first argument denotes a function expressing the value of the conditional. For now, T stands for a tuple $\text{PState} \times \text{CState}$.

$$\begin{aligned} \text{if} : &(\text{PState} \times \text{CState} \to \text{Value}) \times (\text{PState} \times \text{CState} \to T) \\ &\times (\text{PState} \times \text{CState} \to T) \\ &\to (\text{PState} \times \text{CState} \to T) \end{aligned}$$

$$\text{if}(c, then, else)(ps, cs) = \begin{cases} then(ps, cs) & \text{if } c(ps, cs) \neq 0 \\ else(ps, cs) & \text{if } c(ps, cs) = 0 \end{cases}$$

The **btRepeat** rule can be used to illustrate a more complex rule. The effect of repeat is the composition of evaluating the bound and evaluating the body a number of times. The latter is captured in a **repeat** function that is defined below. The resulting effect can be defined as follows:

$$\frac{\Delta^v \vdash e : \langle V_{ex}, \Sigma^v_{ex} \rangle \qquad \Delta^v \vdash S : \Sigma^v_{st}}{\Delta^v \vdash \text{repeat } e \text{ begin } S \text{ end} : \text{repeat}^v(V_{ex}, \Sigma^v_{ex}, \Sigma^v_{st})} \text{ (btRepeat)}$$

The **repeat** operator needed in the **btRepeat** rule captures the behaviour of a loop inside the dependent type. It gets a function that calculates the number of iterations from a type environment, as well as an environment update function for the loop body, and results in an environment update function that represents the effects of the entire loop. Because the value of the bound must be evaluated in the context before the effect is evaluated, we need an extra function:

$$\begin{aligned}
&\text{repeat}^v : (\text{PState} \times \text{CState} \to \text{Value}) \times (\text{PState} \times \text{CState} \to \text{PState} \times \text{CState}) \\
&\quad \times (\text{PState} \times \text{CState} \to \text{PState} \times \text{CState}) \\
&\quad \to (\text{PState} \times \text{CState} \to \text{PState} \times \text{CState}) \\
&\text{repeat}^v(c, start, body)(ps, cs) = \text{repeat}'^v(c(ps, cs), body, ps', cs') \\
&\quad \text{where } (ps', cs') = start(ps, cs)
\end{aligned}$$

The actual recursion is in the repeat'^v function, defined as:

$$\begin{aligned}
&\text{repeat}'^v : \mathbb{Z} \times (\text{PState} \times \text{CState} \to \text{PState} \times \text{CState}) \times \text{PState} \times \text{CState} \\
&\quad \to \text{PState} \times \text{CState} \\
&\text{repeat}'^v(n, body, ps, cs) = \begin{cases} (ps, cs) & \text{if } n \leq 0 \\ \text{repeat}'^v(n-1, body, ps', cs') & \text{if } n > 0 \\ \quad \text{where } (ps', cs') = body(ps, cs) & \end{cases}
\end{aligned}$$

In order to explain the component call rule **btCmp** and function call rule **btCall**, we need an operator for higher order scoping. This operator creates a new program environment, but retains the component state. It can even update the component state given a Σ function, which is needed because this Σ needs to be evaluated using the original program state. The definition is as follows:

$$\begin{aligned}
&\overline{[x \mapsto V, \Sigma]} : \text{Var} \times (\text{PState} \times \text{CState} \to \text{Value}) \\
&\quad \times (\text{PState} \times \text{CState} \to \text{PState} \times \text{CState}) \\
&\quad \to (\text{PState} \times \text{CState} \to \text{PState} \times \text{CState}) \\
&\overline{[x \mapsto V, \Sigma]}(ps, cs) = ([x \mapsto V(ps, cs)], cs') \text{ where } (_, cs') = \Sigma(ps, cs)
\end{aligned}$$

We also need an additional operator **split**, because the program state is isolated but the component state is not. The **split** function forks the evaluation into two state update functions and joins the results together. The first argument defines the resulting program state, the second defines the resulting component state.

$$
\begin{aligned}
&\text{split} : (\text{PState} \times \text{CState} \to \text{PState} \times \text{CState}) \\
&\qquad \times (\text{PState} \times \text{CState} \to \text{PState} \times \text{CState}) \\
&\qquad \to (\text{PState} \times \text{CState} \to \text{PState} \times \text{CState}) \\
&\text{split}(A, B)(ps, cs) = (ps', cs') \\
&\qquad \text{where } (ps', \_) = A(ps, cs) \text{ and } (\_, cs') = B(ps, cs)
\end{aligned}
$$

The component functions and normal functions are stored in the environment Δ^v. For each function, both component and language ones, a triple is stored, which states the variable name of the argument, a value function that represents the return value, and a state update function that represents the effect on the state of executing the called function. We assume the component function calls are already in the environment. The component function call, listed below, can now be easily expressed:

$$
\frac{\Delta^v \vdash e : \langle V_{ex}, \Sigma_{ex} \rangle}{\begin{array}{l}\Delta^v, C.f = (x_f, V_f, \Sigma_f) \vdash \\ \quad C.f(e) : \langle \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg V_f, \ \text{split}(\Sigma_{ex}, \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \Sigma_f) \rangle\end{array}} \text{(btCmp)}
$$

Like the component call, we can define the function call. However, to support recursion, a special **subst** higher order function is introduced to unfold the function definition once, just before it is executed on a concrete environment. The **subst** is defined as:

$$
\begin{aligned}
&\text{subst} : (\text{PState} \times \text{CState} \to T) \\
&\qquad \times (\text{PState} \times \text{CState} \to T) \\
&\qquad \to (\text{PState} \times \text{CState} \to T) \\
&\text{subst}(T, R)(ps, cs) = (T[R \leftarrow \text{subst}(T, R)])(ps, cs)
\end{aligned}
$$

A recursive call is represented by the abstract higher order function **rec**, a sort of placeholder for applying substitution on. There are multiple variants, depending on the resulting type, with the rec_V one for a resulting value function, and the rec_Σ one for a resulting state update function.

$$
\begin{aligned}
&\text{rec}_V : \text{PState} \times \text{CState} \to \text{Value} \\
&\text{rec}_\Sigma : \text{PState} \times \text{CState} \to \text{PState} \times \text{CState}
\end{aligned}
$$

If there is a function body B computing a **Value** with for example $\mathbf{rec}_V$ in it, the value of the recursive function can be computed by executing $\mathbf{subst}(B, \mathbf{rec}_V)$. As long as the original function terminates on the given input environment, this analysis will terminate on the same input. This is the essential difference from the **btCmp** rule, as can be seen in the definition of **btCall** below:

$$\frac{\Delta^v, f = (x_f, V_f, \Sigma_f) \vdash e : \langle V_{ex}, \Sigma_{ex} \rangle}{\begin{array}{l}\Delta^v, f = (x_f, V_f, \Sigma_f) \vdash f(e) : \langle\, \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathbf{subst}(V_f, \mathbf{rec}_V(f)), \\ \qquad\qquad \mathbf{split}(\Sigma_{ex}, \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathbf{subst}(\Sigma_f, \mathbf{rec}_\Sigma(f)))\, \rangle\end{array}} \text{(btCall)}$$

For each language function, a definition is placed in Δ^v by means of the **btFuncDef** rule. The body of the function is analysed, and recursive calls to the function are replaced with **rec** placeholders using the **btRec** rule. To support this, the function definition rule inserts a special definition in the function environment Δ^v, on which the **btRec** rule works. This leads to the following definition of **btFuncDef**:

$$\frac{\Delta^v, f = (x) \vdash e : \langle V_{ex}, \Sigma_{ex} \rangle \qquad \Delta^v, f = (x, V_{ex}, \Sigma_{ex}) \vdash S : \Sigma_{st}}{\Delta^v \vdash \text{function } f(x) \text{ begin } e \text{ end } S : \Sigma_{st}} \text{(btFuncDef)}$$

The placeholders are inserted using the **btRec** rule. This rule analyse the expression used as argument, like the component and function call rules do. The definition is in fact very similar to those definitions:

$$\frac{\Delta^v, f = (x_f) \vdash e : \langle V_{ex}, \Sigma_{ex} \rangle}{\begin{array}{l}\Delta^v, f = (x_f) \vdash \\ \quad f(e) : \langle \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathbf{rec}_V(f), \mathbf{split}(\Sigma_{ex}, \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathbf{rec}_\Sigma(f)) \rangle\end{array}} \text{(btRec)}$$

6.4 Energy-aware dependent type system

In this section, we present the complete type system, which can be used to determine the energy consumption of ECA programs. The type system presented here should be viewed as an extension of the basic type system in section 6.3. Each time the three dot symbol ... is used, that part of the rule is unchanged with respect to the previous section. An element is added to each tuple signifying the energy cost. Expressions now yield a triple, statements yield a pair. The added element is a function that, when evaluated on a concrete environment and component state, results in a concrete energy consumption.

An important energy judgment is the $\mathbf{td}^{ec}$ function. To account for time-dependent energy usage, we need this function which calculates the energy usage of all the components over a certain time period. The $\mathbf{td}^{ec}$ function is a higher order function that takes the time it accounts for as argument. It results in a function that is just like the other function calculating energy bounds: taking two arguments, a $\mathbf{PState}$ and a $\mathbf{CState}$. The definition is given below, which depends on the power draw function ϕ that maps a component state to an energy consumption (as explained in section 6.2.2):

$$\mathbf{td}^{ec} : \mathbf{Time} \to (\mathbf{PState} \times \mathbf{CState} \to \mathbf{EnergyCost})$$
$$\mathbf{td}^{ec}(t)(ps, cs) = t \cdot \sum_{e \in cs} \phi(e)$$

We can use this $\mathbf{td}^{ec}$ function to define a rule for variable lookup, as the only energy cost that is induced by variable access is the energy used by all components for the duration of the variable access. This is expressed in the **etVar** rule, in which the dots correspond to figure 6.3 (the first position is $\mathbf{lookup}_x$, the second $\mathbf{id}$):

$$\frac{}{\Delta^{ec} \vdash x : \langle ..., ..., \mathbf{td}^{ec}(t_{\text{var}}) \rangle} \text{(etVar)}$$

Using the $\overline{+}$ operator (introduced as $\overline{\odot}$, with type T equal to $\mathbf{EnergyCost}$), we can define the energy costs of a binary operation. Although the rule looks complex, it uses pattern introduced in section 6.3 for the binary operation. Basically, the energy cost of a binary operation is the cost of evaluating the operands, plus the cost of the binary operation itself. The binary operation itself only has an associated run-time and by this means induces energy consumption of components. This is expressed by the $\mathbf{td}^{ec}(t_{\odot})$ function. For the binary operation rule we need to apply this pattern twice, in a nested manner, as the time-dependent function must be evaluated in the context after evaluating both arguments. The (three) energy consumptions are added by $\overline{+}$. One can express the binary operation rule as:

$$\frac{\Delta^{ec} \vdash e_1 : \langle ..., \Sigma_1, E_1 \rangle \qquad \Delta^{ec} \vdash e_2 : \langle ..., \Sigma_2, E_2 \rangle}{\Delta^{ec} \vdash e_1 \odot e_2 : \langle ..., ..., E_1 \overline{+} (\Sigma_1 \ggg (E_2 \overline{+} (\Sigma_2 \ggg td^{ec}(t_{\odot})))) \rangle} \text{(etBinOp)}$$

Next is the component function call. The energy cost of a component function call consists of the time taken to execute this function and of explicit energy cost attributed to this call. The environment Δ^{ec} is extended for each component function $C.f$ with two elements: an energy judgment $E_{f'}$ and a run-time $t_{f'}$. Time independent energy usage

$$\frac{}{\Delta^{ec} \vdash c : \langle ..., ..., \mathrm{td}^{ec}(t_{\mathrm{const}}) \rangle}\ (\mathrm{etConst}) \qquad \frac{}{\Delta^{ec} \vdash x : \langle ..., ..., \mathrm{td}^{ec}(t_{\mathrm{var}}) \rangle}\ (\mathrm{etVar})$$

$$\frac{\Delta^{ec} \vdash e_1 : \langle ..., \Sigma_1, E_1 \rangle \qquad \Delta^{ec} \vdash e_2 : \langle ..., \Sigma_2, E_2 \rangle}{\Delta^{ec} \vdash e_1 \boxdot e_2 : \langle ..., ..., E_1 \mp (\Sigma_1 \ggg (E_2 \mp (\Sigma_2 \ggg \mathrm{td}^{ec}(t_{\boxdot})))) \rangle}\ (\mathrm{etBinOp})$$

$$\frac{\Delta^{ec} \vdash e : \langle ..., \Sigma, E \rangle}{\Delta^{ec} \vdash x := e : \langle ..., ..., E \mp (\Sigma \ggg \mathrm{td}^{ec}(t_{\mathrm{assign}})) \rangle}\ (\mathrm{etAssign})$$

$$\frac{\Delta^{ec} \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle}{\begin{array}{l}\Delta^{ec},\ C.f = (x_f, \ldots, \ldots, E_f, t_f) \vdash \\ \quad C.f(e) : \langle ..., ..., E_{ex} \mp (\overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg (\mathrm{td}^{ec}(t_f) \mp E_f)) \rangle\end{array}}\ (\mathrm{etCmp})$$

$$\frac{\Delta^{ec}, f = (x_f, \ldots, \ldots, E_f) \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle}{\Delta^{ec}, f = (x_f, \ldots, \ldots, E_f) \vdash f(e) : \langle ..., ..., E_{ex} \mp (\overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathrm{subst}(E_f, \mathrm{rec}_E(f))) \rangle}\ (\mathrm{etCall})$$

$$\frac{\Delta^{ec}, f = (x_f) \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle}{\Delta^{ec}, f = (x_f) \vdash f(e) : \langle ..., ..., \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathrm{rec}_E(f) \rangle}\ (\mathrm{etRec})$$

$$\frac{\Delta^{ec} \vdash S : \langle \Sigma_{st}, E_{st} \rangle \qquad \Delta^{ec} \vdash e : \langle ..., ..., E_{ex} \rangle}{\Delta^{ec} \vdash S, e : \langle ..., ..., E_{st} \mp (\Sigma_{st} \ggg E_{ex}) \rangle}\ (\mathrm{etExprConcat})$$

$$\frac{\Delta^{ec} \vdash e : \langle ..., \Sigma, E \rangle}{\Delta^{ec} \vdash e : \langle \Sigma, E \rangle}\ (\mathrm{etExprAsStmt}) \qquad \frac{}{\Delta^{ec} \vdash \mathsf{skip} : \langle \ldots, \mathrm{zero} \rangle}\ (\mathrm{etSkip})$$

$$\frac{\Delta^{ec} \vdash S_1 : \langle \Sigma_1, E_1 \rangle \qquad \Delta^{ec} \vdash S_2 : \langle \ldots, E_2 \rangle}{\Delta^{ec} \vdash S_1; S_2 : \langle \ldots, E_1 \mp (\Sigma_1 \ggg E_2) \rangle}\ (\mathrm{etStmtConcat})$$

$$\frac{\Delta^{ec} \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle \qquad \Delta^{ec} \vdash S_t : \langle \ldots, E_t \rangle \qquad \Delta^{ec} \vdash S_f : \langle \ldots, E_f \rangle}{\begin{array}{ll}\Delta^{ec} & \vdash \mathsf{if}\ e\ \mathsf{then}\ S_t\ \mathsf{else}\ S_f\ \mathsf{end} : \\ & \langle \ldots, E_{ex} \mp (\Sigma_{ex} \ggg \mathrm{td}^{ec}(t_{\mathrm{if}})) \mp \mathrm{if}(V_{ex}, \Sigma_{ex} \ggg E_t, \Sigma_{ex} \ggg E_f) \rangle\end{array}}\ (\mathrm{etIf})$$

$$\frac{\Delta^{ec} \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle \qquad \Delta^{ec} \vdash S : \langle \Sigma_{st}, E_{st} \rangle}{\Delta^{ec} \vdash \mathsf{repeat}\ e\ \mathsf{begin}\ S\ \mathsf{end} : \langle \ldots, E_{ex} \mp \mathrm{repeat}^{ec}(V_{ex}, \Sigma_{ex}, E_{st}, \Sigma_{st}) \rangle}\ (\mathrm{etRepeat})$$

$$\frac{\Delta^{ec}, f = (x) \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle \qquad \Delta^{ec}, f = (x, V_{ex}, \Sigma_{ex}, E_{ex}) \vdash S : \langle \Sigma_{st}, E_{st} \rangle}{\Delta^{ec} \vdash \mathsf{function}\ f(x)\ \mathsf{begin}\ e\ \mathsf{end}\ S : \langle E_{st}, \Sigma_{st} \rangle}\ (\mathrm{etFuncDef})$$

figure 6.5 Energy-aware dependent type system. It adds an element to the resulting tuples of figure 6.4 signifying the energy consumption of executing the statement or expression. Note that c stands for a ⟨*const*⟩ term, x for a ⟨*var*⟩ term, e for a ⟨*expr*⟩ term, S for a ⟨*stmt*⟩ term, $\boxdot$ for a ⟨*bin-op*⟩, f for a ⟨*function-name*⟩, and C for a ⟨*component*⟩.

can be encoded into this $E_{f'}$ function. For functions defined in the language the derived energy judgement is inserted into the environment. There is no need for these functions for an explicit run-time as this is part of the derived energy judgement. Using the patterns described above the component function call is expressed as:

$$\frac{\Delta^{ec} \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle}{\begin{array}{l}\Delta^{ec},\ C.f = (x_f, \ldots, \ldots, E_f, t_f) \vdash \\ \quad C.f(e) : \langle \ldots, \ldots, E_{ex} \mathbin{\overline{+}} ([x_f \mapsto V_{ex}, \Sigma_{ex}] \ggg (\mathrm{td}^{ec}(t_f) \mathbin{\overline{+}} E_f)) \rangle\end{array}} \text{(etCmp)}$$

Calculating the energy cost of the repeat loop can be calculated by evaluating an energy cost function for each loop iteration (in the right context). We therefore have to modify the $\mathbf{repeat}^{v}$ function to yield an energy cost, resulting in a new function definition:

$$\begin{array}{l}\mathbf{repeat}^{ec} : (\mathrm{PState} \times \mathrm{CState} \to \mathrm{Value}) \times (\mathrm{PState} \times \mathrm{CState} \to \mathrm{PState} \times \mathrm{CState}) \\ \quad \times (\mathrm{PState} \times \mathrm{CState} \to \mathrm{EnergyCost}) \times (\mathrm{PState} \times \mathrm{CState} \to \mathrm{PState} \times \mathrm{CState}) \\ \quad \to (\mathrm{PState} \times \mathrm{CState} \to \mathrm{EnergyCost}) \\ \mathbf{repeat}^{ec}(c, start, cost, body)(ps, cs) = \\ \quad \mathbf{repeat}'^{ec}(c(ps, cs), cost, body, ps', cs') \text{ where } (ps', cs') = \mathrm{start}(ps, cs)\end{array}$$

$$\begin{array}{l}\mathbf{repeat}'^{ec} : \mathbb{Z} \times (\mathrm{PState} \times \mathrm{CState} \to \mathrm{EnergyCost}) \\ \quad \times (\mathrm{PState} \times \mathrm{CState} \to \mathrm{PState} \times \mathrm{CState}) \\ \quad \times \mathrm{PState} \times \mathrm{CState} \to \mathrm{EnergyCost} \\ \mathbf{repeat}'^{ec}(n, cost, body, ps, cs) = \begin{cases} 0 & \text{if } n \leq 0 \\ \mathbf{repeat}'^{ec}(n-1, cost, body, ps', cs') & \text{if } n > 0 \\ \quad + cost(ps, cs) & \\ \quad\quad \text{where } (ps', cs') = body(ps, cs) & \end{cases}\end{array}$$

Using this $\mathbf{repeat}^{ec}$ function, the definition of the rule for the repeat is analogous to the previous definition.

$$\frac{\Delta^{ec} \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle \qquad \Delta^{ec} \vdash S : \langle \Sigma_{st}, E_{st} \rangle}{\Delta^{ec} \vdash \text{repeat } e \text{ begin } S \text{ end} : \langle \ldots, E_{ex} \mathbin{\overline{+}} \mathbf{repeat}^{ec}(V_{ex}, \Sigma_{ex}, E_{st}, \Sigma_{st}) \rangle} \text{(etRepeat)}$$

6.5 Example

In this section, we demonstrate our analysis on two example programs, comparing their energy usage. Each ECA language construct has an associated execution time bound. These execution times are used in calculating the energy consumption that is time-dependent. Another source of energy consumption in our modelling is time-independent energy usage, which can also be associated with any ECA construct. Adding these two sources together yields the energy consumption.

Consider the example programs in figures 6.6 and 6.7. Both programs play #n beeps at #hz Hz (#n and #hz are input variables). The effect of the first statement is starting a component named SoundSystem, which models a sound card (actually, functions have a single parameter; this parameter is omitted here as it is not used). After enabling component SoundSystem, beeps are played by calling the function `playBeepAtHz()` on SoundSystem. Eventually the device is switched off. The sound system has two states, off $<$ on.

For this example, we will assume a start state in which the component is in the off state and an input $n \in \mathbb{Z}$. Furthermore, the `SoundSystem.playBeepAtHz()` function has a time-independent energy cost of i, and a call to this function has an execution time of t seconds. The `System.sleep()` has no associated (time-independent) energy usage, and takes s seconds to complete. The other component function calls and the loop construct are assumed to have zero execution time and zero (time-independent) energy consumption. While switched on, the component has a power draw of u J/s (or W).

We will start with an intuitive explanation of the energy consumption of the program in figure 6.6, then continue by applying the analysis presented in this paper and comparing these results. Quickly calculating the execution time of the program yields a result of $n \cdot (t + s)$ seconds. As the component is switched on at the start and switched off at the

```
1 SoundSystem.on();
2 repeat #n begin
3   SoundSystem.playBeepAtHz(#hz);
4   System.sleep()
5 end;
6 SoundSystem.off()
```

listing 6.6 Example program.

```
1 repeat #n begin
2   SoundSystem.on();
3   SoundSystem.playBeepAtHz(#hz);
4   SoundSystem.off();
5   System.sleep()
6 end
```

listing 6.7 Alternative program.

end of the program, the time-dependent energy consumption is $n \cdot (t + s) \cdot u$. The time-independent energy usage is equal to the number of calls to transmit, thus $n \cdot i$, resulting in an energy consumption of $n \cdot (i + (t + s) \cdot u)$ J.

Now, we will show the results from our analysis. Applying the energy-aware dependent typing rules ultimately yields an energy consumption function and a state update function. Both take a typing and a component state environment as input. The first rule to apply is **etStmtConcat**, with the `SoundSystem.on()` call as S_1 and the remainder of the program as S_2. The effect of the component function call, calculated with **etCmp**, is that the component state is increased (as this signifies a higher time-dependent energy usage). This effect is represented in the component state update function $\delta_{\text{SoundSystem.on}}$. Since we have assumed that the call costs zero time and has no time-independent energy cost, the resulting energy consumption function is the identity function $\mathbf{id}$.

We can now analyse the loop with the **etRepeat** rule. We first derive the type of expression e, which determines the number of iterations of the loop. As the expression is a simple variable access, which we assumed not to have any associated costs, the program state and the component states are not touched. For the result of the expression the type system derives $\mathbf{lookup}_{\#n}$ as V_{ex} in the rule, which is (a look-up of) n. This is a function that, when given an input environment, calculates a number in $\mathbb{Z}$ which signifies the number of iterations.

Moving on, we analyse the body of the loop. This means we again apply the **etCmp** rule to determine the resource consumption of the call to `SoundSystem.playBeepAtHz()`. The call takes time t. Its energy consumption is u if the component is switched `on`. For ease of presentation, the component states $\{\mathsf{off},\mathsf{on}\}$ are represented as a variable $e \in \{0,1\}$. The time-independent energy usage of the loop body is i. The energy consumption function E_{st} of the body is $i + e \cdot (t + s) \cdot u$ J.

We can now combine the results of the analysis of the number of iterations and the resource consumption of the loop body (E_{st}) to calculate the consumption of the entire loop. Basically, the resource consumption of the loop E_{st} is multiplied by the number of iterations V_{ex}. This is done in the function $\mathbf{repeat}^{ec}(V_{ex}, \Sigma_{ex}, E_{st}, \Sigma_{st})$, in this case $\mathbf{repeat}^{ec}(\mathbf{lookup}_{\#n}, \mathbf{id}, E_{st}, \mathbf{id})$. Evaluation after the state update function (corresponding to the `SoundSystem.on()` call), which will update the state of the sound system to `on`. Evaluating the sequence of both expressions on the start state results in $n \cdot (i + (t + s) \cdot u)$ J, signifying the energy consumption of the code fragment.

Analysing the code fragment listed in figure 6.7 will result in a type equivalent to $n \cdot (i + t \cdot u)$ J, as the cost of switching the sound system on or off is zero. Given the energy characteristics of the sound system, the second code fragment will have a lower energy consumption. Depending on the realistic characteristics (switching a device on or off normally takes time and energy), a realistic trade-off can be made.

6.6 Related work

Much of the research that has been devoted to producing green software is focussed on a very abstract level, defining programming and design patterns for writing energy-efficient code [PBL13; Alb10; Sax10; Ran10; NW01; Siv+02]. In [Bri+13], a modular design for energy-aware software is presented that is based on a series of rules on UML schemes. In [Coh+12] and [Sam+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.

Contrary to the abstract levels of the aforementioned papers, there is also research on producing green software on a very low, hardware-specific level [JML06; Ker+13a]. Such analysis methods work on specific architectures ([JML06] on SimpleScalar, [Ker+13a] on XMOS ISA-level models), while our approach is hardware-parametric.

Furthermore, there is research on building compilers that optimize code for energy-efficiency. In [Zhu+09], the implementation of several energy optimisation methods, including dynamic voltage scaling and bit-switching minimisation, into the GCC compiler is described and evaluated. Iterative compilation is applied to energy consumption in [GCB05]. In [Meh+97], a technique is proposed in which register labels are encoded alternatively to minimise switching costs. This saves an average of 4.25% of energy, without affecting performance. The human-written assembly code for an MP3 decoder is optimised by hand in [Šim+00], guided by an energy profiling tool based on a proprietary ARM simulator. In [Zha+03], functional units are disabled to reduce (leakage) energy consumption of a VLIW processor. The same is done for an adaptation of a DEC Alpha 21264 in [YLL06]. Reduced bit-width instruction set architectures are exploited to save energy in [SD04]. Energy is saved on memory optimisations in [Joo+02; Ver+06; Jon+09; LC03], while [OYI01; Sap+02] focus on variable-voltage processors.

Sustainability can also be seen as a quality property, as explained in [Lag+15]. A framework called Green is presented in [BC10], allowing programmers to approximate expensive functions and calculate Quality of Service (QoS) statistics. It can thus help leverage a trade-off between performance and energy consumption on the one hand, and QoS on the other.

In [Bri+14], Resource-Utilization Models (RUMs) are presented, which are an abstraction of the resource behaviour of hardware components. The component models in this paper can be viewed as an instance of a RUM. RUMs can be analysed, e.g. with the model checker Uppaal, while we use a dedicated dependent type system. A possible future research direction is to incorporate RUMs into our analysis as component models.

Analyses for consumption of generic resources are built using recurrence relation solving [Alb+11a], amortised analysis [HAH11], amortisation and separation logic [Atk10] and a Vienna Development Method style program logic [Asp+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 power-draw that depends on the state of components..

Several dependently typed programming languages exist, such as Epigram [McB05] and Agda [BDN09]. The Deputy system, which adds dependent typing to the C language, is described in [Con+07]. Dependent types are applied to security in [ML10]. Security types there enforce access control and information flow policies.

6.7 Future work

In future work, we aim to implement this analysis in order to evaluate its suitability for larger systems and validate practical applicability. We intend to experiment with implementations of various derived approximating analyses in order evaluate which techniques / approximations work best in which context.

A limitation is currently that only a system controlled by **one central process** (processor) can be analysed. Modern systems often consists of a network of interacting systems. Therefore, incorporating interacting systems would increase applicability of the approach.

To make the presentation more concise, the subject language can be a first-order strict-evaluation functional program. This alleviates the need to have a separate basic dependent type system which transform all variables into expressions over input variables. However, expressions still need to be expressed in terms over input variables.

On the language level the type system is precise, however it does not take into account optimisations and transformations below the language level. This can be achieved by analysing the software on a lower level, for example the intermediate representation of a modern compiler. Another motivation to use such an intermediate representation as the language that is analysed, is the ability to support (combinations of) many higher level languages. In this way, programs written in and consisting of multiple languages can be analysed. It can also account for optimisations (such as common subexpression elimination, inlining, statically evaluating expressions), which in general reduce the execution time of the program and therefore impact the time-dependent energy usage (calls with side effects like component function calls are generally not optimised).

Another future research direction is to expand the supported language that is analysed. In order to support data types, a size analysis of data types is needed to enable iteration over data structures, e.g. using techniques similar to [SET13; TSv09].

Compared with the previous chapter, the derived energy consumption is precise instead of overapproximated. Future research can show if the expressions derived by this type system can be transformed in such a way that upper bound expressions are derived. To support this, recursion and loops should be transformed in a Cost Relation System (CRS), a special case of recurrence relations. Solving this CRS, one gets a direct formula expressing the energy consumption of the recursive function or loop. The difficult part is the manner in which the state update functions are overapproximated. To overapproximate these, heavy restrictions on component states and component functions were applied in the previous chapter. However, to work with realistic models of components, less restrictions should be imposed.

A systematic approach to constructing component models should be looked into. One can create a model from the specifications given by the vendor. Another way is using model learning techniques, that creates a finite state model from blackbox testing and measuring. All the states should have a time dependent energy consumption assigned to them, and all the transitions should be assigned incidental energy consumption. This model can be used as a component model.

6.8 Discussion

The presented type system captures energy bounds for software that is executed on hardware components of which component models are available. It is **precise, modular and elegant**, yet retains the hardware-parametric aspect.

The new type system is in itself **precise**, as there is no overestimation, as opposed to the Hoare logic in chapter 5, in which the conditional and loop are overestimated. Also the class of programs that can be studied is larger, as many of the restrictions needed for the overapproximation are now lifted. However, the same considerations on the validity of component models apply as mentioned in the discussion of the previous chapter.

The presented hardware-parametric dependent type system with function signatures enables **modularity**. While analysing the energy consumption of an electronic system, instead of re-analysing the body of functions each time a function call is encountered, the function signature is reused. This same signature is used to support recursion in the input language.

By using a dependent type system, the resulting approach is **elegant and concise**, for a couple of reasons. First, no externally verified properties are needed, like the ρ in the previous chapter. The basics of the type system consists of only a small number of higher order operators. The type system is gradually expanded from a basic type system to an energy type system, which is easier in the presentation.

7 DISCUSSION

SUMMARY *The approaches proposed in this thesis are a first step in making the software industry more aware of the resources they consume, and thereby making them more sustainable. The presented methods enable programmers to asses the footprint of their software, in terms of memory and energy.*

These approaches show that resource analysis is feasible for different resources, being memory or energy, in a precise or overapproximated fashion. Most of these approaches are implemented in prototypes, which demonstrate that these methods can be applied in a practical setting: the analysis methods require neither a supercomputer nor huge amounts of resources to use. Implementation details of the software analysed impact the resource consumption significantly, and this is reflected by the bounds (or types) yielded by applying these methods.

Results from these analysis methods can be used in different ways, either directly or indirectly. An indirect way to reduce energy usage is to reduce memory usage, as energy consumption of current embedded system can for a significant part be attributed to the energy consumption of the memory system: keeping state, moving data and invalidating caches. A direct way to use the results is to calculate the conditions that are required for a piece of software to function, e.g. the amount of memory needed in a system or the the remaining battery charge required to maintain a working state for the required amount of time for an operation. Another direct way to use the results is to compare two algorithms. Depending on the envisioned use, the most energy-efficient algorithm can be selected. Because we define sustainable software as correct software with a relatively low energy consumption, this can be used to select the most sustainable algorithm.

Abstraction and encapsulation are often propagated in computer science as a design philosophy. However, they can hinder the insight of programmers into the behaviour of their programs. The proposed analysis methods, and more specifically the accompanying case studies, show that the implementation details have a significant influence on the derived resource consumption bounds. More layers of abstraction and encapsulation means less control, and potentially higher resource consumption.

Correctness plays an important role in resource consumption. Erroneously behaving software and devices running the software can be expensive, in terms of maintenance, keeping the device active or replacing the device altogether. If a system is faulty, resources can be consumed without producing meaningful results. For one of the publications by

the author, [BvG-3], a correctness bug, hampering processing data, resulted in loss of 3.5 days of work by 800 cores in about 80 cluster nodes. This is equivalent to the energy consumption of about four households in the Netherlands for the period of one year. Only after it was too late, the author realised that proper testing should be done before large deployments. This is one personal example, but there are numerous publicly available similar mishaps, yet education in these matters is limited. Computer science curricula could devote more time to increase the awareness of the resources used by the software industry.

In this thesis, a symbolic execution method that has seen wide application on software, has been applied to hardware designs to deduce channel types. This shows there are similarities between hardware and software, although they differ greatly. Software has often a reduced concurrency level, however, hardware features highly parallel designs. Nevertheless, the derived channel types can be used to check absence of misrouting, and is therefore a correctness property.

Scientifically speaking, there is still much work to be done. This includes, but is not limited to, the following list.

analysing parallel programs for resource usage Parallel programs have many interesting interactions, for a large part depending on the exact timing. This makes deriving upper bounds on resource analysis difficult, as the precision suffers. As a result, the derived bound is often overestimated, making it less useful in practice.

integrate timing in the resource bounds Resource usage can be viewed as a time-dependent function over time. This particular view can give new insight into software's resource usage. Together with timing information on process or thread synchronisation, and scheduling policies, this opens up the possibility to derive more precise resource bounds by adding bounds of processes in the periods between synchronisation moments.

making the tools applicable in practice The methods are demonstrated using prototypes in limited case studies. These methods are yet to be tested on a large scale in a practical setting, by programmers. Ideally, these are integrated in an IDE, enabling their use as tools to be used in a daily workflow of a programmer.

support for programming languages used in practice Not all language constructs used in the latest iterations of programming languages are supported by the analysis methods. There will always be a gap between the latest versions of programming languages and analysis methods, as certain programming constructs hamper the analysis of programs. Just as the use of goto is considered harmful, certain constructs should be avoided if a programmer wants to keep their code analysable by these approaches. Other constructs need to be implemented and/or the analysis methods adjusted to cover these constructs.

symbolic execution for energy Another approach is possible besides the program transformation deriving an exact bound for concrete input and the overapproximation resulting in symbolic bounds. By using symbolic execution, a result can be obtained that is both symbolic and precise. Future research should be focussed on using a different interpretation of the types derived in chapter 6, so that these types can become a common foundation to base multiple kinds of analyses on.

validation study A validation study, validating the obtained bounds with measurements from a real setup, should be performed. This is useful to verify the reliability of the approaches presented in this thesis.

Returning to the societal impact of software as discussed in the introduction, the methods presented in this thesis can help to verify that software does not negatively impact society, not even when large quantities of devices are running the software. When development teams are able to apply these approaches, they may gain additional insight in the behaviour of their software and may better assess its energy footprint. Problems may be spotted early in the design and development process, before mass production and distribution of the devices. The ability to analyse programs in this way is one essential step towards making the software industry more sustainable. However, much remains to be done. The approaches presented in this thesis have, together with contributions of others in this line of research, potentially a high societal relevance.

APPENDICES &

A SUMMARY

Motivation Many aspects of modern life depend on, and are controlled by, software. Almost every modern device contains software. Because we heavily depend on these devices, it is important to asses the sustainability of software running on these devices. We define sustainable software as correct software with a relatively low energy consumption.

If software demonstrates erroneous behaviour, undesirable effects can occur. This can impact users and the environment heavily. Limited memory usage is important for the correct working of software, especially in devices one uses daily. Erroneously behaving software and devices running the software can be expensive, in terms of maintenance, keeping the device active or replacing the device altogether. To avoid wasting resources, we consider correctness a prerequisite of sustainable software.

The second aspect of sustainable software is energy usage. As traditionally many energy savings did occur in the hardware side of a computer, energy consumption is a blind spot when developing software. However, recently the advancement in these hardware savings has lost its pace. At the same time, it becomes more and more clear that software has a huge impact on the behaviour and the properties of devices it runs on. Software is in control of the device and its energy consumption.

Small effects are aggravated at large scale. The combination of many individual negative effects can affect our society at large. If devices that are present in large quantities in our society all exhibit the same negative behaviour, it can impact public utilities and our economy.

In this thesis, methods are proposed to analyse energy and memory consumption, and correctness of software. These methods enable a programmer to improve the sustainability of the software they are developing.

Results The analysis methods as proposed in this thesis are a first step in making the software industry more aware of the resources they consume. These approaches show that resource analysis is feasible for different resources, being memory or energy, in a precise or overapproximated fashion. Most of these approaches are implemented in prototypes, which demonstrate that these methods can be applied automatically in a practical setting, since these analysis methods neither require a super computer nor huge amounts of resources to use. Implementation details of the software that is analysed impact the energy and memory consumption of software significantly, and this is reflected by the bounds (and types) yielded by applying these methods.

Results from these analysis methods can be used in different ways, either directly or indirectly. An indirect way to reduce energy usage is to reduce memory usage. A direct way to use the results is to compare two algorithms. Depending on the envisioned use, the most sustainable algorithm can be selected. Correctness plays an important role in resource consumption: if a system is faulty, resources can be consumed without producing meaningful results.

Abstraction and encapsulation are often propagated, however, they can hinder the insight of programmers into the behaviour of their programs. The proposed analysis methods, and more specifically the accompanying case studies, show that the implementation details hidden by abstraction and encapsulation have a significant influence on the derived resource consumption bounds. More layers of abstraction and encapsulation means less control, and potentially higher resource consumption.

Impact The methods presented in this thesis can help to asses whether software is sustainable. This can prevent that software impacts society negatively, even when large quantities of devices are running the software. The presented methods enable programmers to asses the footprint of their software, in terms of memory and energy. When development teams are able to apply these approaches, they may gain additional insight in the behaviour of their software. Due to the shift in energy consumption globally, it is essential that the software industry becomes sustainable. The ability to analyse programs in this way is one necessary step towards making the software industry more sustainable. The approaches presented in this thesis have, together with contributions of others in this line of research, potentially a high societal relevance.

B SAMENVATTING

Motivatie Grote delen van ons moderne leven hangen af van, of worden geleid door, software. Bijna alle moderne apparaten bevatten software. Omdat we afhankelijk zijn van deze apparaten, is het belangrijk om te weten of de software die erop draait duurzaam is. Onder duurzaamheid van software verstaan we het correct functioneren ervan met een relatief laag energieverbruik.

Als software niet correct functioneert, kunnen ongewenste effecten optreden. Dit kan weer grote gevolgen hebben voor de gebruikers of voor ons milieu. Tevens is beperkt geheugengebruik belangrijk voor het correct functioneren van software, met name voor apparaten die dagelijks gebruikt worden. Slecht functionerende software en apparaten waarop het draait kunnen duur zijn in onderhoud, duur in exploitatie en duur om ze eventueel te vervangen. Om verspilling te voorkomen, beschouwen we correctheid van software als een noodzakelijke voorwaarde voor duurzame software.

Het tweede aspect van duurzame software is energieverbruik. Van oudsher werd energie bespaard binnen de hardware kant van een computer, waardoor energieverbruik een blinde vlek was tijdens het ontwikkelen van software. Echter, de laatste tijd gaan de ontwikkelingen op het gebied van deze energiebesparingen bij hardware langzamer. Tegelijk wordt het duidelijker dat software een grote rol speelt bij het gedrag en de eigenschappen van de apparaten waarop de software draait. Met andere woorden, software is de baas over het apparaat, en daarmee ook over het energieverbruik ervan.

Kleine effecten kunnen op grote schaal gevolgen hebben. De optelsom van veel kleine individuele effecten kan invloed hebben op onze samenleving. Als apparaten die veel voorkomen in onze samenleving allemaal hetzelfde negatieve gedrag vertonen, kan dat leiden tot schade aan onze publieke voorzieningen en onze economie.

In dit proefschrift worden methoden ontwikkeld om correctheid, geheugengebruik, en energieverbruik van software te analyseren. Deze methoden stellen een programmeur in staat om de software die hij/zij ontwikkelt duurzamer te maken.

Resultaten De analysemethoden zoals voorgesteld in dit proefschrift zijn een eerste stap om de software industrie (nog) meer bewust te maken van hun invloed op de samenleving en het milieu. Deze methoden tonen aan dat het mogelijk is om software te analyseren voor verschillende aspecten zoals geheugengebruik en energieverbruik. Van de meeste methoden zijn automatische prototypes gemaakt. Deze prototypes demonstreren dat de methoden ook in de praktijk automatisch toegepast kunnen worden, aangezien de prototypes noch een snelle computer nodig hebben noch grote hoeveelheden geheugen of energie om te gebruiken. Uit de resultaten van dit proefschrift komt naar voren dat de manier waarop software opgebouwd is grote invloed heeft op het gedrag van software in termen van correctheid, geheugengebruik, en energieverbruik.

Resultaten van deze methoden kunnen direct of indirect gebruikt worden. Een indirecte manier om energieverbruik te beperken is om het geheugenverbruik te beperken. Een meer directe manier is het vergelijken van de verschillende manieren waarop software is opgebouwd. Afhankelijk van het verwachte verbruik van de software, kan de keuze gemaakt worden welke opbouwmanier het meest duurzaam is. Correctheid is noodzakelijk: foutieve software kan energie verbruiken zonder nuttige resultaten op te leveren.

Abstractie wordt vaak ingezet binnen informatica, maar kan het inzicht van programmeurs in het gedrag van software hinderen. De voorgestelde analysemethoden en de bijbehorende praktijkstudies laten zien dat juist de opbouwdetails die verborgen worden door abstractie grote invloed kunnen hebben op het verbruik van software. Meer abstractielagen betekent minder controle, en mogelijk hoger verbruik.

Impact De methoden die voorgesteld worden in dit proefschrift kunnen gebruikt worden om aan te tonen of software duurzaam is. Hiermee kan worden voorkomen dat software ontwikkeld wordt die de samenleving negatief beïnvloedt, zelfs als het apparaat waarop de software draait in groten getale aanwezig is. De voorgestelde methoden stellen programmeurs in staat om de voetafdruk van hun software vast te stellen, in termen van correctheid, geheugengebruik, en energieverbruik.

Voorspellingen geven aan dat het wereldwijde energieverbruik nog gaat stijgen. Dit maakt het noodzakelijk dat ook de software industrie duurzamer wordt. De mogelijkheid om programma's te analyseren is een eerste stap om de software industrie duurzamer te maken. De voorgestelde methoden in dit proefschrift hebben daardoor, samen met bijdragen van andere in dit wetenschapsgebied, potentieel een grote maatschappelijke relevantie.

C CONTRIBUTIONS

C.1 Peer reviewed publications

[BvG-1] Bernard van Gastel, Leonard Lensink, Sjaak Smetsers, and Marko van Eekelen. "**Reentrant readers-writers: a case study combining model checking with theorem proving**". In: *Formal methods for industrial critical systems: 13th international workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, revised selected papers*. Edited by Darren Cofer and Alessandro Fantechi. Volume 5596. Lecture Notes Computer Science. **The authors received the EASST Award for Best Paper, recognising the paper as the 'Best Software Science Paper' of FMICS2008**. Berlin, Heidelberg: Springer, 2008, pages 85–102. DOI: 10.1007/978-3-642-03240-0_10.

[BvG-2] Bernard van Gastel, Leonard Lensink, Sjaak Smetsers, and Marko van Eekelen. "**Deadlock and starvation free reentrant readers-writers: a case study combining model checking with theorem proving**". In: *Science of computer programming* 76.2 (2011), pages 82–99. DOI: 10.1016/j.scico.2010.03.004.

[BvG-3] Abdulaziz Alhussien, Nader Bagherzadeh, Freek Verbeek, Bernard van Gastel, and Julien Schmaltz. "**A formally verified deadlock-free routing function in a fault-tolerant NoC architecture**". In: *25th symposium on integrated circuits and systems design (SBCCI), 2012*. Aug. 2012, pages 1–6. DOI: 10.1109/SBCCI.2012.6344433.

[BvG-4] 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)*. Edited by Martin Schoeberl and Andy J. Wellings. JTRES'12. New York: ACM, 2012, pages 135–144. DOI: 10.1145/2388936.2388959.

[BvG-5] Bernard van Gastel and Julien Schmaltz. "**A formalisation of xMAS**". In: *Proceedings international workshop on the ACL2 theorem prover and its applications, ACL2 2013, Laramie, Wyoming, USA, May 30-31, 2013*. Edited by Ruben Gamboa and Jared Davis. Volume 114. EPTCS. 2013, pages 111–126. DOI: 10.4204/EPTCS.114.9.

[BvG-6] Sebastiaan Joosten, Bernard van Gastel, and Julien Schmaltz. "**A macro for reusing abstract functions and theorems**". In: *Proceedings international workshop on the ACL2 theorem prover and its applications, ACL2 2013, Laramie, Wyoming, USA, May 30-31, 2013*. Edited by Ruben Gamboa and Jared Davis. Volume 114. EPTCS. 2013, pages 29–41. DOI: 10.4204/EPTCS.114.3.

[BvG-7] 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: *NASA formal methods*. Edited by Guillaume Brat, Neha Rungta, and Arnaud Venet. Volume 7871. Lecture Notes in Computer Science. Springer, 2013, pages 63–77. DOI: 10.1007/978-3-642-38088-4_5.

[BvG-8] Abdulaziz Alhussien, Freek Verbeek, Bernard van Gastel, Nader Bagherzadeh, and Julien Schmaltz. "**Fully reliable dynamic routing logic for a fault-tolerant NoC architecture**". In: *Journal of integrated circuits and systems* 8.1 (2013), pages 43–53.

[BvG-9] 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)*. Volume 8552. LNCS. Springer, 2014, pages 93–109. DOI: 10.1007/978-3-319-12466-7_6.

[BvG-10] Rody Kersten, Bernard van Gastel, Olha Shkaravska, Manuel Montenegro, and Marko van Eekelen. "**ResAna: a resource analysis toolset for (real-time) Java**". In: *Concurrency and computation: practice and experience* 26.14 (2014), pages 2432–2455. DOI: 10.1002/cpe.3154.

[BvG-11] Bernard van Gastel, Freek Verbeek, and Julien Schmaltz. "**Inference of channel types in micro-architectural models of on-chip communication networks**". In: *22nd international conference on very large scale integration, VLSI-SoC, Playa del Carmen, Mexico, October 6-8, 2014*. Edited by Lorena Garcia. IEEE, 2014, pages 1–6. DOI: 10.1109/VLSI-SoC.2014.7004168.

[BvG-12] Daniel Bernstein, Bernard van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. "**TweetNaCl: a crypto library in 100 tweets**". In: *Progress in cryptology - LATINCRYPT 2014 - third international conference on cryptology and information security in Latin America, Florianópolis, Brazil, September 17-19, 2014, revised selected papers*. Edited by Diego F. Aranha and Alfred Menezes. Volume 8895. Lecture Notes in Computer Science. Springer, 2014, pages 64–83. DOI: 10.1007/978-3-319-16295-9_4.

[BvG-13] Sylvia Stuurman, Bernard van Gastel, and Harrie Passier. "**The design of mobile apps: what and how to teach?**" In: *Proceedings of the computer science education research conference, CSERC 2014, Berlin, Germany, November 5-6, 2014*. Edited by Erik Barendsen and Valentina Dagiene. ACM, 2014, pages 93–100. DOI: 10.1145/2691352.2691360.

[BvG-14] Bernard van Gastel, Rody Kersten, and Marko van Eekelen. "**Using dependent types to define energy augmented semantics of programs**". In: *Proceedings of the fourth international workshop on foundational and practical aspects of resource analysis (FOPARA'15)*. Volume 9964. LNCS. Springer, 2016, pages 1–20. DOI: 10.1007/978-3-319-46559-3_2.

C.2 Media

[BvG-15] Roel van der Heijden. **Hoeveel Moore nog?** Kennislink. Interview about **the future of processors** in a science magazine for the general public. April 17th, 2013. URL: http://www.kennislink.nl/publicaties/hoeveel-moore-nog.

[BvG-16] Hoe?Zo! Radio. Live radio interview Dutch national radio about **when will Moore's Law end?** November 28th, 2013. URL: http://www.dekennisvannu.nl/site/media/Wanneer-sneuvelt-de-wet-van-Moore/3141.

[BvG-17] George van Hal. **Het boeket van bier**. New Scientist. Interview about **recommending beers based on the taste of the user**, using the Bayesian Sets algorithm, in a science magazine for the general public. May 21th, 2015. URL: https://shop.newscientist.nl/shop/new-scientist-nummer-23-2015/141570163809/kopen.html.

C.3 Other publications

[BvG-18] Rody Kersten, Bernard van Gastel, and Marko van Eekelen. "**Software en energiegebruik**". In: *Duurzame ICT, grondstof en energiebron voor een duurzame wereld*. Edited by Roel Croes. Academic Service, 2010, pages 23–33.

[BvG-19] Bernard van Gastel, Rody Kersten, and Marko van Eekelen. **Course material on memory analysis**. Institute for Programming research and Algorithmics (IPA). This course was part of the IPA winterschool on software technology. Jan. 2013. URL: `http://www.win.tue.nl/ipa/?event=software-engineering-and-technology-course-3`.

[BvG-20] Paolo Parisen Toldin, Rody Kersten, Bernard van Gastel, and Marko van Eekelen. **Soundness proof for a Hoare logic for energy consumption analysis**. Technical report ICIS–R13009. Radboud University Nijmegen, Oct. 2013.

[BvG-21] Bernard van Gastel and Marko van Eekelen. **Lecture notes on 'Analysing energy consumption by software', for the TACLe Summerschool 2016 in Yspertal, Austria**. Technical report. Radboud University Nijmegen, Sept. 2016.

D BIBLIOGRAPHY

[Alb10] Susanne Albers. "**Energy-efficient algorithms**". In: *Communications of the ACM* 53.5 (2010), pages 86–96 (cited on pages 107, 152).

[Alb+11a] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. "**Closed-form upper bounds in static cost analysis**". In: *Journal of automated reasoning* 46.2 (Feb. 2011), pages 161–203. DOI: 10.1007/s10817-010-9174-1 (cited on pages 72, 86, 153).

[Alb+08] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. "**COSTA: design and implementation of a cost and termination analyzer for Java bytecode**". In: *Formal methods for components and objects*. Edited by Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever. Volume 5382. Lecture Notes in Computer Science. Springer, 2008, pages 113–132 (cited on pages 73, 85, 101, 107).

[Alb+11b] Elvira Albert, Richard Bubel, Samir Genaim, Reiner Hähnle, Germán Puebla, and Guillermo Román-Dı´ez. "**Verified resource guarantees using COSTA and KeY**". In: *Proceedings of the 20th ACM SIGPLAN workshop on partial evaluation and program manipulation*. PEPM '11. Austin, Texas, USA: ACM, 2011, pages 73–76 (cited on page 100).

[AGG10] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. "**Parametric inference of memory requirements for garbage collected languages**". In: *ISMM'10*. Edited by Jan Vitek and Doug Lea. ACM, 2010, pages 121–130. DOI: 10.1145/1806651.1806671 (cited on pages 85, 101).

[AGM11] Elvira Albert, Samir Genaim, and Abu Naser Masud. "**More precise yet widely applicable cost analysis**". In: *VMCAI'11*. Edited by Ranjit Jhala and David A. Schmidt. Volume 6538. Lecture Notes in Computer Science. Springer, 2011, pages 38–53. DOI: 10.1007/978-3-642-18275-4_5 (cited on page 101).

[Alh+12] Abdulaziz Alhussien, Nader Bagherzadeh, Freek Verbeek, Bernard E. van Gastel, and Julien Schmaltz. "**A formally verified deadlock-free routing function in a fault-tolerant NoC architecture**". In: *25th symposium on integrated circuits and systems design (SBCCI), 2012*. Referenced in the thesis as [BvG-3], see appendix C. Aug. 2012, pages 1–6. DOI: 10.1109/SBCCI.2012.6344433.

[Alh+13] Abdulaziz Alhussien, Freek Verbeek, Bernard E. van Gastel, Nader Bagherzadeh, and Julien Schmaltz. "**Fully reliable dynamic routing logic for a fault-tolerant NoC architecture**". In: *Journal of integrated circuits and systems* 8.1 (2013). Referenced in the thesis as [BvG-8], see appendix C, pages 43–53.

[Ama05] Roberto M. Amadio. "**Synthesis of max-plus quasi-interpretations**". In: *Fundamenta informaticae* 65.1-2 (Aug. 2005), pages 29–60. URL: http://portal.acm.org/citation.cfm?id=1227143.1227146 (cited on page 72).

[APV07] Saswat Anand, Corina S. Pasareanu, and Willem Visser. "**JPF-SE: a symbolic execution extension to Java PathFinder**". In: *Tacas*. Edited by Orna Grumberg and Michael Huth. Volume 4424. Lecture Notes in Computer Science. Springer, 2007, pages 134–138. DOI: 10.1007/978-3-540-71209-1_12 (cited on page 44).

[And+04] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. "**Zing: a model checker for concurrent software**". In: *CAV*. Edited by Rajeev Alur and Doron Peled. Volume 3114. Lecture Notes in Computer Science. Springer, 2004, pages 484–487 (cited on page 44).

[AHS98] Myla Archer, Constance Heitmeyer, and Steve Sims. "**TAME: A PVS interface to simplify proofs for automata models**". In: *User interfaces for theorem provers*. Eindhoven, The Netherlands, 1998. URL: citeseer.ist.psu.edu/archer98tame.html (cited on pages 43, 45).

[Asp+07] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. "**A program logic for resources**". In: *Theoretical computer science.* 389.3 (Dec. 2007), pages 411–445. DOI: 10.1016/j.tcs.2007.09.003 (cited on pages 107, 153).

[Atk10] Robert Atkey. "**Amortised resource analysis with separation logic**". In: *Programming languages and systems*. Edited by Andrew D. Gordon. Volume 6012. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pages 85–103. DOI: 10.1007/978-3-642-11957-6_6 (cited on pages 107, 153).

[BC10] Woongki Baek and Trishul M. Chilimbi. "**Green: a framework for supporting energy-conscious programming using controlled approximation**". In: *SIGPLAN notices* 45.6 (June 2010), pages 198–209. DOI: 10.1145/1809028.1806620 (cited on page 153).

[Bar05] Marco Antonio Barbosa. "**A refinement calculus for software components and architectures**". In: *SIGSOFT software engineering notes* 30.5 (2005), pages 377–380. DOI: 10.1145/1095430.1081767 (cited on page 45).

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. "**The SMT-LIB Standard: Version 2.0**". In: *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*. Edited by A. Gupta and D. Kroening. 2010 (cited on page 83).

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. **Verification of object-oriented software: the KeY approach**. LNCS 4334. Springer, 2007 (cited on pages 73, 77).

[Ben08] Mordechai Ben-Ari. **Principles of the SPIN model checker**. Springer, 2008 (cited on page 21).

[BM02] Luca Benini and Giovanni De Micheli. "**Networks on Chips: a new SoC paradigm**". In: *IEEE computer* 35.1 (Jan. 2002), pages 70–78. DOI: 10.1109/2.976921 (cited on page 48).

[Ber+14] Daniel Bernstein, Bernard E. van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. "**TweetNaCl: a crypto library in 100 tweets**". In: *Progress in cryptology - LATINCRYPT 2014 - third international conference on cryptology and information security in Latin America, Florianópolis, Brazil, September 17-19, 2014, revised selected papers*. Edited by Diego F. Aranha and Alfred Menezes. Volume 8895. Lecture Notes in Computer Science. Referenced in the thesis as [BvG-12], see appendix C. Springer, 2014, pages 64–83. DOI: 10.1007/978-3-319-16295-9_4.

[BC04] Yves Bertot and Pierre Castéran. **Interactive theorem proving and program development. Coq'Art: the calculus of inductive constructions**. Texts in Theoretical Computer Science. Springer Verlag, 2004. URL: http://www.labri.fr/publications/l3a/2004/BC04 (cited on page 34).

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. "**A brief overview of Agda – a functional language with dependent types**". In: *Theorem proving in higher order logics*. Edited by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Volume 5674. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pages 73–78. DOI: 10.1007/978-3-642-03359-9_6 (cited on page 153).

[Bri+13] Steven te Brinke, Somayeh Malakuti, Christoph Bockisch, Lodewijk Bergmans, and Mehmet Akşit. "**A design method for modular energy-aware software**". In: *Proceedings of the 28th annual ACM symposium on applied computing*. Coimbra, Portugal: ACM, 2013, pages 1180–1182. DOI: 10.1145/2480362.2480584 (cited on pages 107, 152).

[Bri+14] Steven te Brinke, Somayeh Malakuti, Christoph Bockisch, Lodewijk Bergmans, Mehmet Akşit, and Shmuel Katz. "**A tool-supported approach for modular design of energy-aware software**". In: *Proceedings of the 29th annual ACM symposium on applied computing, Gyeongju, Korea*. SAC'14. ACM, Mar. 2014 (cited on pages 107, 153).

[BTM00] David Brooks, Vivek Tiwari, and Margaret Martonosi. "**Wattch: a framework for architectural-level power analysis and optimizations**". In: *SIGARCH computer architecture news* 28.2 (May 2000), pages 83–94 (cited on page 107).

[Bro03] Christopher W. Brown. "**QEPCAD B: a program for computing with semi-algebraic sets using CADs**". In: *SIGSAM bulletin* 37.4 (Dec. 2003), pages 97–108. DOI: 10.1145/968708.968710 (cited on pages 73, 89).

[BvG-0] Bernard E. van Gastel. "**Verifying reentrant readers-writers**". Master's thesis. Nijmegen, Netherlands: Radboud Universiteit, Mar. 2010 (cited on page 33).

[Cha+06] Patrice Chalin, Joseph R Kiniry, Gary T Leavens, and Erik Poll. "**Beyond assertions: advanced specification and verification with JML and ESC/Java2**". In: *Formal methods for components and objects 2005, revised lectures*. Volume 4111. Lecture Notes in Computer Science. Springer, 2006, pages 342–363 (cited on page 77).

[CK10] Satrajit Chatterjee and Michael Kishinevsky. "**Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics**". In: *Proceedings of the 22nd international conference on computer aided verification (CAV'10)*. Edited by Tayssir Touili, Byron Cook, and Paul Jackson. Volume 6174. Lecture Notes in Computer Science. Springer, July 2010 (cited on page 48).

[CK12] Satrajit Chatterjee and Michael Kishinevsky. "**Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics**". In: *Formal methods in system design* 40.2 (Apr. 2012), pages 147–169. DOI: 10.1007/s10703-011-0134-0 (cited on page 48).

[CKO10] Satrajit Chatterjee, Michael Kishinevsky, and Ümit Y. Ogras. "**Quick formal modeling of communication fabrics to enable verification**". In: *Proceedings of the IEEE international high level design validation and test workshop (HLDVT'10)*. 2010, pages 42–49. DOI: 10.1109/HLDVT.2010.5496662 (cited on page 48).

[CKO12] Satrajit Chatterjee, Michael Kishinevsky, and Ümit Y. Ogras. "**xMAS: quick formal modeling of communication fabrics to enable verification**". In: *IEEE design & test of computers* 29.3 (2012), pages 80–88. DOI: 10.1109/MDT.2012.2205998 (cited on pages 48, 58, 63).

[CL87] Charles K. Chui and Ming-Jun Lai. "**Vandermonde determinants and Lagrange interpolation in R^s**". In: *Nonlinear and convex analysis* (1987), pages 23–35 (cited on page 75).

[CES83] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. "**Automatic verification of finite state concurrent systems using temporal logic specifications: a practical approach**". In: *POPL*. 1983, pages 117–126 (cited on page 17).

[CF09] Darren D. Cofer and Alessandro Fantechi, editors. **Formal methods for industrial critical systems, 13th international workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, revised selected papers**. Volume 5596. Lecture Notes in Computer Science. Springer, 2009. DOI: 10.1007/978-3-642-03240-0.

[Coh+12] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. "**Energy types**". In: *SIGPLAN notices* 47.10 (Oct. 2012), pages 831–850. DOI: 10.1145/2398857.2384676 (cited on pages 107, 152).

[Con+07] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and GeorgeC. Necula. "**Dependent types for low-level programming**". In: *Programming languages and systems*. Edited by Rocco De Nicola. Volume 4421. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pages 520–535. DOI: 10.1007/978-3-540-71316-6_35 (cited on page 153).

[Cop+09] Marcello Coppola, Miltos D. Grammatikakis, Riccardo Locatelli, Giuseppe Mariuccia, and Lorenzo Pieralisi. **Design of interconnect processing units Spidergon STNoC**. CRC Press, 2009 (cited on page 50).

[Cor+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. "**Bandera: extracting finite-state models from Java source code**". In: *Proceedings of the 2000 international conference on software engineering*. 2000, pages 439–448. DOI: 10.1109/ICSE.2000.870434 (cited on page 44).

[Cor+01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. **Introduction to algorithms**. 2nd. McGraw-Hill Higher Education, 2001 (cited on page 63).

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. "**Concurrent control with 'readers' and 'writers'**". In: *Communications of the ACM* 14.10 (1971), pages 667–668. DOI: 10.1145/362759.362813 (cited on page 17).

[DMP10] 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. Springer, 2010, pages 305–319 (cited on page 72).

[DP11] Javier de Dios and Ricardo Peña. "**Fm 2011: formal methods: 17th international symposium on formal methods, limerick, ireland, june 20-24, 2011. proceedings**". In: edited by Michael Butler and Wolfram Schulte. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. Chapter Certification of Safe Polynomial Memory Bounds, pages 184–199. DOI: 10.1007/978-3-642-21437-0_16 (cited on page 102).

[Eek+08] Marko C.J.D. van Eekelen, Stefan ten Hoedt, René Schreurs, and Yaroslav S. Usenko. "**Analysis of a session-layer protocol in mCRL2. Verification of a real-life industrial implementation**". In: *Proc. 12th international workshop on formal methods for industrial critical systems (FMICS 2007)*. Edited by P. Merino and S. Leue. Volume 4916. Lecture Notes Computer Science. Springer, 2008, pages 182–199 (cited on page 43).

[Eek+07] Marko C.J.D. van Eekelen, Olha Shkaravska, Ron van Kesteren, Bart Jacobs, Erik Poll, and Sjaak Smetsers. "**AHA: amortized heap space usage analysis**". In: *Selected papers of the 8th international symposium on trends in functional programming (TFP'07), New York, USA*. Edited by Marco Morazán. Intellect Publishers, UK, 2007, pages 36–53 (cited on page 72).

[Erm+07] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. "**Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis**". In: *7th international workshop on worst-case execution time (WCET) analysis*. Edited by Christine Rochange. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007 (cited on page 100).

[EU-1] European Union. **Energy efficient products**. Published online. Retrieved from `http://ec.europa.eu/energy/en/topics/energy-efficiency/energy-efficient-products`. May 26th, 2016 (cited on page 5).

[EU-2] European Union. **642/2009 and 1062/2010: televisions**. Published online. Retrieved from `http://www.eceee.org/ecodesign/products/televisions`. May 26th, 2016 (cited on page 5).

[EU-3] European Union. **Regulation 801/2013: non-tertiary coffee machines (lot 25)**. Published online. Retrieved from `http://www.eceee.org/ecodesign/products/Lot25_non_tertiary_coffee_machines`. May 26th, 2016 (cited on page 5).

[EU-4] European Union. **Vacuum cleaners**. Published online. Retrieved from `https://ec.europa.eu/energy/en/topics/energy-efficiency/energy-efficient-products/vacuum-cleaners`. May 26th, 2016 (cited on page 5).

[EU-5] European Union. **Electricity production, consumption and market overview**. Published online. Retrieved from `http://ec.europa.eu/eurostat/statistics-explained/index.php/Electricity_production,_consumption_and_market_overview`. May 26th, 2016 (cited on page 5).

[Fer+13] Miguel A Ferreira, Eric Hoekstra, Bo Merkus, Bram Visser, and Joost Visser. "**SEFLab: a lab for measuring software energy footprints**". In: *2nd international workshop on green and sustainable software (GREENS), 2013*. IEEE. 2013, pages 30–37 (cited on page 131).

[FJ10] Jędrzej Fulara and Krzysztof Jakubczyk. "**Practically applicable formal methods**". In: *SOFSEM'10: proceedings of the 36th conference on current trends in theory and practice of computer science*. 'pindlerov Mlýn, Czech Republic: Springer, 2010, pages 407–418 (cited on page 100).

[GD13] Ruben Gamboa and Jared Davis, editors. **Proceedings international workshop on the ACL2 theorem prover and its applications**. Volume 114. EPTCS. 2013. DOI: `10.4204/EPTCS.114`.

[Gar14] Lorena Garcia, editor. **22nd international conference on very large scale integration, vlsi-soc, playa del carmen, mexico, october 6-8, 2014**. IEEE, 2014.

[GKv16] Bernard E. van Gastel, Rody W.J. Kersten, and Marko C.J.D. van Eekelen. "**Using dependent types to define energy augmented semantics of programs**". In: *Proceedings of the fourth international workshop on foundational and practical aspects of resource analysis (FOPARA'15)*. Volume 9964. LNCS. Referenced in the thesis as [BvG-14], see appendix C. Springer, 2016, pages 1–20. DOI: `10.1007/978-3-319-46559-3_2`.

[Gas+08] Bernard E. van Gastel, Leonard Lensink, Sjaak Smetsers, and Marko C.J.D. van Eekelen. "**Reentrant readers-writers: a case study combining model checking with theorem proving**". In: *Formal methods for industrial critical systems: 13th international workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, revised selected papers*. Edited by Darren Cofer and Alessandro Fantechi. Volume 5596. Lecture Notes Computer Science. **The authors received the EASST Award for Best Paper, recognising the paper as the 'Best Software Science Paper' of FMICS2008**. Referenced in the thesis as [BvG-1], see appendix C. Berlin, Heidelberg: Springer, 2008, pages 85–102. DOI: `10.1007/978-3-642-03240-0_10`.

[Gas+11] Bernard E. van Gastel, Leonard Lensink, Sjaak Smetsers, and Marko C.J.D. van Eekelen. "**Deadlock and starvation free reentrant readers-writers: a case study combining model checking with theorem proving**". In: *Science of computer programming* 76.2 (2011). Referenced in the thesis as [BvG-2], see appendix C, pages 82–99. DOI: `10.1016/j.scico.2010.03.004`.

[GS13] Bernard E. van Gastel and Julien Schmaltz. "**A formalisation of xMAS**". In: *Proceedings international workshop on the ACL2 theorem prover and its applications, ACL2 2013, Laramie, Wyoming, USA, May 30-31, 2013*. Edited by Ruben Gamboa and Jared Davis. Volume 114. EPTCS. Referenced in the thesis as [BvG-5], see appendix C. 2013, pages 111–126. DOI: `10.4204/EPTCS.114.9`.

[GVS14] Bernard E. van Gastel, Freek Verbeek, and Julien Schmaltz. "**Inference of channel types in micro-architectural models of on-chip communication networks**". In: *22nd international conference on very large scale integration, VLSI-SoC, Playa del Carmen, Mexico, October 6-8, 2014*. Edited by Lorena Garcia. Referenced in the thesis as [BvG-11], see appendix C. IEEE, 2014, pages 1–6. DOI: `10.1109/VLSI-SoC.2014.7004168`.

[GH06] Zoubin Ghahramani and Katherine A Heller. "**Bayesian sets**". In: *Advances in neural information processing systems 18*. Edited by Y. Weiss, B. Schölkopf, and J. C. Platt. MIT Press, 2006, pages 435–442. URL: `http://papers.nips.cc/paper/2817-bayesian-sets.pdf` (cited on page 9).

[GCB05] Stefan Valentin Gheorghita, Henk Corporaal, and Twan Basten. "**Iterative compilation for energy reduction**". In: *Journal of embedded computing* 1.4 (2005), pages 509–520 (cited on pages 107, 152).

[GSE13] Attila Gobi, Olha Shkaravska, and Marko C.J.D. van Eekelen. "**Higher-order size checking without subtyping**". In: *Proceedings of the 13th international symposium on trends in functional programming (TFP2012)*. Edited by Hans-Wolfgang Loidl and Kevin Hammond. Volume 7829. Lecture Notes in Computer Science. Springer, 2013, pages 53–68 (cited on page 100).

[Goe+06] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. **Java concurrency in practice**. Addison Wesley Professional, 2006 (cited on page 19).

[GCK11] Alexander Gotmanov, Satrajit Chatterjee, and Michael Kishinevsky. "**Verifying deadlock-freedom of communication fabrics**". In: *Verification, model checking, and abstract interpretation (VMCAI'11)*. Volume 6538. 2011, pages 214–231. DOI: `10.1007/978-3-642-18275-4\_16` (cited on page 48).

[Gro08] Adriaan de Groot. "**Practical automaton proofs in PVS**". PhD thesis. Radboud University Nijmegen, 2008 (cited on page 43).

[Gul09] Sumit Gulwani. "**SPEED: symbolic complexity bound analysis**". In: *CAV'09: proceedings of the 21st international conference on computer aided verification*. Grenoble, France: Springer-Verlag, 2009, pages 51–62 (cited on page 100).

[GJK09] Sumit Gulwani, Sagar Jain, and Eric 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*. Dublin, Ireland: ACM, 2009, pages 375–385 (cited on page 100).

[GZ10] Sumit Gulwani and Florian Zuleger. "**The reachability-bound problem**". In: *Proceedings of the 2010 ACM SIGPLAN conference on programming language design and implementation*. PLDI'10. Toronto, Ontario, Canada: ACM, 2010, pages 292–304 (cited on pages 72, 100).

[Ha+04] Vu Ha, Murali Rangarajan, Darren Cofer, Harald Rues, and Bruno Dutertre. "**Feature-based decomposition of inductive proofs applied to real-time avionics software: an experience report**". In: *ICSE'04: proceedings of the 26th international conference on software engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pages 304–313 (cited on page 43).

[HGR07] Andreas Hansson, Kees Goossens, and Andrei Rdulescu. "**Avoiding message-dependent deadlock in network-based systems on chip**". In: *VLSI design* (2007) (cited on page 52).

[HP00] Klaus Havelund and Thomas Pressburger. "**Model checking Java programs using Java PathFinder**". In: *STTT* 2.4 (2000), pages 366–381 (cited on page 44).

[HS96] Klaus Havelund and Natarajan Shankar. "**Experiments in Theorem Proving and Model Checking for Protocol Verification**". In: *FME'96: industrial benefit and advances in formal methods*. Edited by Marie-Claude Gaudel and Jim Woodcock. Springer-Verlag, 1996, pages 662–681. URL: `citeseer.ist.psu.edu/248384.html` (cited on page 43).

[Her+11] Jörg Herter, Peter Backes, Florian Haupenthal, and Jan 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. URL: `http://rw4.cs.uni-sb.de/~jherter/papers/CAMAecrts11.pdf` (cited on page 101).

[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. Lecture Notes in Computer Science. Springer, 2010, pages 287–306 (cited on page 102).

[HAH11] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. "**Multivariate amortized resource analysis**". In: *POPL'11*. Edited by Thomas Ball and Mooly Sagiv. ACM, 2011, pages 357–370 (cited on pages 72, 102, 107, 153).

[HJ03] M. Hofmann and S. Jost. "**Static prediction of heap space usage for first-order functional programs**". In: *POPL'03*. New Orleans, Louisiana, USA: ACM Press, 2003, pages 185–197. DOI: `10.1145/604131.604148` (cited on page 102).

[Hol04] G.J. Holzmann. **The SPIN model checker: primer and reference manual**. Addison-Wesley, Boston, USA, 2004 (cited on page 21).

[Hun+06] James J. Hunt, Fridtjof B. Siebert, Peter H. Schmitt, and Isabel 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*. Paris, France: ACM, 2006, pages 162–169. DOI: `10.1145/1167999.1168026` (cited on pages 99, 100).

[HTS08] James J. Hunt, Isabel Tonin, and Fridtjof 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*. Santa Clara, California: ACM, 2008, pages 97–105. DOI: `10.1145/1434790.1434806` (cited on pages 73, 93, 108).

[IEA15a] International Energy Agency. **Key world energy statistics 2015**. 2015. URL: http://www.iea.org/publications/freepublications/publication/KeyWorld_Statistics_2015.pdf (cited on page 5).

[IEA15b] International Energy Agency. **Energy and climate change**. 2015. URL: http://www.worldenergyoutlook.org/media/news/WEO2015_COP21Briefing.pdf (cited on page 6).

[JSW07] Bart Jacobs, Sjaak Smetsers, and Ronny Wichers Schreur. "**Code-carrying theories**". In: *Formal aspects of computing* 19.2 (2007), pages 191–203. DOI: 10.1007/s00165-006-0013-4 (cited on page 45).

[Jag+16] Erik A. Jagroep, Jan Martijn van der Werf, Sjaak Brinkkemper, Giuseppe Procaccianti, Patricia Lago, Leen Blom, and Rob van Vliet. "**Software energy profiling: comparing releases of a software product**". In: *Proceedings of the 38th international conference on software engineering companion*. ICSE '16. Austin, Texas: ACM, 2016, pages 523–532. DOI: 10.1145/2889160.2889216 (cited on page 131).

[JML06] Ramkumar Jayaseelan, Tulika Mitra, and Xianfeng Li. "**Estimating the worst-case energy consumption of embedded software**". In: *Proceedings of the 12th IEEE real-time and embedded technology and applications symposium*. IEEE, 2006, pages 81–90 (cited on pages 107, 130, 152).

[Jon+09] Timothy M. Jones, Michael F. P. O'Boyle, Jaume Abella, Antonio González, and Oğuz Ergin. "**Energy-efficient register caching with compiler assistance**". In: *ACM transactions on architecture and code optimization* 6.4 (2009), pages 1–23. DOI: 10.1145/1596510.1596511 (cited on page 152).

[Joo+02] Yongsoo Joo, Yongseok Choi, Hojun Shim, Hyung Gyu Lee, Kwanho Kim, and Naehyuck Chang. "**Energy exploration and reduction of SDRAM memory systems**". In: *DAC'02: proceedings of the 39th annual design automation conference*. New Orleans, Louisiana, USA: ACM, 2002, pages 892–897. DOI: 10.1145/513918.514138 (cited on page 152).

[JGS13] Sebastiaan J. C. Joosten, Bernard E. van Gastel, and Julien Schmaltz. "**A macro for reusing abstract functions and theorems**". In: *Proceedings international workshop on the ACL2 theorem prover and its applications, ACL2 2013, Laramie, Wyoming, USA, May 30-31, 2013*. Edited by Ruben Gamboa and Jared Davis. Volume 114. EPTCS. Referenced in the thesis as [BvG-6], see appendix C. 2013, pages 29–41. DOI: 10.4204/EPTCS.114.3.

[Jun+06] Meuse N. O. Junior, Silvino Neto, Paulo Romero Martins Maciel, Ricardo Massa Ferreira Lima, Angelo Ribeiro, Raimundo S. Barreto, Eduardo Tavares, and Frederico Braga. "**Analyzing software performance and energy consumption of embedded systems by probabilistic modeling: an approach based on coloured Petri nets.**" In: *ICATPN'06*. 2006, pages 261–281 (cited on page 107).

[Kal+09] Tomás Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, Ben Titzer, and Jan 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*. ACM, 2009, pages 41–50 (cited on page 99).

[KZ08] Aman Kansal and Feng Zhao. "**Fine-grained energy profiling for power-aware application design**". In: *SIGMETRICS performance evaluation review* 36.2 (Aug. 2008), pages 26–31 (cited on page 107).

[Kat01] Shmuel Katz. "**Faithful translations among models and specifications**". In: *FME'01: proceedings of the international symposium of formal methods Europe on formal methods for increasing software productivity*. London, UK: Springer-Verlag, 2001, pages 419–434 (cited on page 43).

[Ker+13a] Steven Kerrison, Umer Liqat, Kyriakos Georgiou, Alejandro Serrano Mena, Neville Grech, Pedro Lopez-Garcia, Kerstin Eder, and Manuel V. Hermenegildo. "**Energy consumption analysis of programs based on XMOS ISA-level models**". In: *LOPSTR'13*. Springer, Sept. 2013 (cited on pages 107, 152).

[Ker+13b] Rody W.J. Kersten, Bernard E. van Gastel, Manu Drijvers, Sjaak Smetsers, and Marko C.J.D. van Eekelen. "**Using model-checking to reveal a vulnerability of tamper-evident pairing**". In: *NASA formal methods*. Edited by Guillaume Brat, Neha Rungta, and Arnaud Venet. Volume 7871. Lecture Notes in Computer Science. Referenced in the thesis as [BvG-7], see appendix C. Springer, 2013, pages 63–77. DOI: `10.1007/978-3-642-38088-4_5`.

[Ker+14a] 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**". In: *Concurrency and computation: practice and experience* 26.14 (2014). Referenced in the thesis as [BvG-10], see appendix C, pages 2432–2455. DOI: `10.1002/cpe.3154`.

[Ker+14b] Rody W.J. Kersten, Paolo Parisen Toldin, Bernard E. van Gastel, and Marko C.J.D. 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. LNCS. Referenced in the thesis as [BvG-9], see appendix C. Springer, 2014, pages 93–109. DOI: `10.1007/978-3-319-12466-7_6`.

[Ker+12] Rody W.J. Kersten, Olha Shkaravska, Bernard E. van Gastel, Manuel Montenegro, and Marko C.J.D. 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)*. Edited by Martin Schoeberl and Andy J. Wellings. JTRES'12. Referenced in the thesis as [BvG-4], see appendix C. New York: ACM, 2012, pages 135–144. DOI: `10.1145/2388936.2388959`.

[KSE08] Ron van Kesteren, Olha Shkaravska, and Marko C.J.D. 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. Electronic Notes in Theoretical Computer Science. 2008, pages 45–63 (cited on pages 72, 100).

[Kin76] James C. King. "**Symbolic execution and program testing**". In: *Communications of the ACM* 19 (7 July 1976), pages 385–394 (cited on page 82).

[Lac+98] Ph Lacan, Jean Noel Monfort, LVQ Ribal, A Deutsch, and G Gonthier. "**Ariane 5 – the software reliability verification process**". In: *DASIA'98 – data systems in aerospace*. Volume 422. 1998, page 201 (cited on page 3).

[Lag+15] Patricia Lago, Sedef Akinli Koçak, Ivica Crnkovic, and Birgit Penzenstadler. "**Framing sustainability as a property of software quality**". In: *Commun. ACM* 58.10 (2015), pages 70–78. DOI: `10.1145/2714560` (cited on page 153).

[LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. "**UPPAAL in a nutshell**". In: *International journal on software tools for technology transfer* 1.1-2 (1997), pages 134–152. URL: `citeseer.ist.psu.edu/larsen97uppaal.html` (cited on page 43).

[LKP07] Gary T. Leavens, Joseph R. Kiniry, and Erik Poll. "**A JML tutorial: modular specification and verification of functional behavior for Java**". In: *CAV*. Edited by Werner Damm and Holger Hermanns. Volume 4590. Lecture Notes in Computer Science. Springer, 2007, page 37. DOI: 10.1007/978-3-540-73368-3_6 (cited on page 44).

[Lea+13] Gary T. Leavens et al. **JML reference manual. draft revision 2344**. May 2013 (cited on pages 72, 77).

[LC03] Hyung Gyu Lee and Naehyuck Chang. "**Energy-aware memory allocation in heterogeneous non-volatile memory systems**". In: *ISLPED'03: proceedings of the 2003 international symposium on low power electronics and design*. Seoul, Korea: ACM, 2003, pages 420–423. DOI: 10.1145/871506.871609 (cited on page 152).

[Lok+09] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. "**A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models**". In: *CGO'09: proceedings of the 7th annual IEEE/ACM international symposium on code generation and optimization*. Washington, DC, USA: IEEE Computer Society, 2009, pages 136–146 (cited on page 100).

[McB05] Conor McBride. "**Epigram: practical programming with dependent types**". In: *Advanced functional programming*. Edited by Varmo Vene and Tarmo Uustalu. Volume 3622. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pages 130–170. DOI: 10.1007/11546382_3 (cited on page 153).

[Meh+97] Huzefa Mehta, Robert Michael Owens, Mary Jane Irwin, Rita Chen, and Debashree Ghosh. "**Techniques for low energy software**". In: *ISLPED'97: proceedings of the 1997 international symposium on low power electronics and design*. Monterey, California, United States: ACM, 1997, pages 72–75. DOI: 10.1145/263272.263286 (cited on page 152).

[Mic+08] Marianne de Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. "**Static loop bound analysis of C programs based on flow analysis and abstract interpretation**". In: *Rtcsa '08: proceedings of the 2008 14th IEEE international conference on embedded and real-time computing systems and applications*. Washington, DC, USA: IEEE Computer Society, 2008, pages 161–166 (cited on page 100).

[Mog+15] F. A. Moghaddam, T. Geenen, P. Lago, and P. Grosso. "**A user perspective on energy profiling tools in large scale computing environments**". In: *Sustainable internet and ict for sustainability (sustainit), 2015*. Apr. 2015, pages 1–5. DOI: 10.1109/SustainIT.2015.7101364 (cited on page 131).

[MRH12] 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. Lecture Notes in Computer Science. Talinn, Estonia: Springer Verlag, Mar. 2012, pages 209–223 (cited on page 71).

[MPS15] Manuel Montenegro, Ricardo Peña, and Clara Segura. "**Space consumption analysis by abstract interpretation: reductivity properties**". In: *Science of computer programming* 111, Part 3 (2015). Special Issue on Foundational and Practical Aspects of Resource Analysis (FOPARA) 2009 and 2011, pages 458–482. DOI: 10.1016/j.scico.2014.04.014 (cited on page 102).

[Mon+12] Manuel Montenegro, Olha Shkaravska, Marko van Eekelen, and Ricardo 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 2011*. LNCS 7177. Springer, 2012, pages 36–53 (cited on pages 73, 88).

[ML10] Jamie Morgenstern and Daniel R. Licata. "**Security-typed programming within dependently typed programming**". In: *Proceedings of the 15th ACM SIGPLAN international conference on functional programming*. ICFP '10. Baltimore, Maryland, USA: ACM, 2010, pages 169–180. DOI: 10.1145/1863543.1863569 (cited on page 153).

[NW01] Kshirasagar Naik and David S. L. Wei. "**Software implementation strategies for power-conscious systems**". In: *Mobile networks and applications* 6.3 (2001), pages 291–305. DOI: 10.1023/A:1011487018981 (cited on page 152).

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. **Isabelle/HOL – a proof assistant for higher-order logic**. Volume 2283. LNCS. Springer, 2002 (cited on page 34).

[Nog+11] Bruno Nogueira, Paulo Maciel, Eduardo Tavares, Ermeson Andrade, Ricardo Massa, Gustavo Callou, and Rodolfo Ferraz. "**A formal model for performance and energy evaluation of embedded systems**". In: *EURASIP journal on embedded systems* (Jan. 2011), 2:1–2:12. DOI: 10.1155/2011/316510 (cited on page 107).

[OYI01] Takanori Okuma, Hiroto Yasuura, and Tohru Ishihara. "**Software energy reduction techniques for variable-voltage processors**". In: *IEEE design and test of computers* 18.2 (2001), pages 31–41. DOI: 10.1109/54.914613 (cited on page 152).

[OZH16] Rik Oldenkamp, Rosalie van Zelm, and Mark A.J. Huijbregts. "**Valuing the human health damage caused by the fraud of volkswagen**". In: *Environmental pollution* 212 (2016), pages 121–127. DOI: 10.1016/j.envpol.2016.01.053 (cited on page 4).

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. "**PVS: a prototype verification system**". In: *11th international conference on automated deduction (CADE)*. Edited by Deepak Kapur. Volume 607. Lecture Notes in Artificial Intelligence. Saratoga, NY: Springer-Verlag, June 1992, pages 748–752. URL: http://www.csl.sri.com/papers/cade92-pvs/ (cited on page 34).

[PJ98] Jens Palsberg and C Barry Jay. "**The essence of the visitor pattern**". In: *The 22th annual international computer software and applications conference, 1998. COMPSAC'98. proceedings*. IEEE. 1998, pages 9–15 (cited on page 62).

[Pan+06] Vera Pantelic, Xiao-Hui Jin, Mark Lawford, and David Lorge Parnas. "**Inspection of concurrent systems: combining tables, theorem proving and model checking**". In: *Software engineering research and practice*. Edited by Hamid R. Arabnia and Hassan Reza. CSREA Press, 2006, pages 629–635 (cited on pages 17, 43).

[PR04] Andreas Podelski and Andrey Rybalchenko. "**A complete method for the synthesis of linear ranking functions**". In: *Verification, model checking, and abstract interpretation*. Edited by Bernhard Steffen and Giorgio Levi. Volume 2937. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, pages 465–486 (cited on pages 87, 100, 108).

[PBL13] Giuseppe Procaccianti, Stefano Bevini, and Patricia Lago. "**Energy efficiency in cloud software architectures**". In: *27th international conference on environmental informatics for environmental protection, sustainable development and risk management, environfo 2013, hamburg, germany, september 2-4, 2013. proceedings*. 2013, pages 291–299 (cited on page 152).

[QS82] Jean-Pierre Queille and Joseph Sifakis. "**Specification and verification of concurrent systems in CESAR**". In: *Symposium on programming*. Edited by Mariangiola Dezani-Ciancaglini and Ugo Montanari. Volume 137. Lecture Notes in Computer Science. Springer, 1982, pages 337–351 (cited on page 17).

[Ran10] Parthasarathy Ranganathan. "**Recipe for efficiency: principles of power-aware computing**". In: *Communications of the ACM* 53.4 (2010), pages 60–67 (cited on pages 107, 152).

[Rei+07] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. "**Timing predictability of cache replacement policies**". In: *Real-time systems* 37.2 (Nov. 2007), pages 99–122. DOI: 10.1007/s11241-007-9032-3 (cited on page 101).

[RJE03] K. Richter, M. Jersak, and R. Ernst. "**A formal approach to MpSoC performance verification**". In: *Computer* 36.4 (Apr. 2003), pages 60–67 (cited on page 48).

[Rob+06] Robby, Edwin Rodrı´guez, Matthew B. Dwyer, and John Hatcliff. "**Checking JML specifications using an extensible software model checking framework**". In: *STTT* 8.3 (2006), pages 280–299. DOI: 10.1007/s10009-005-0218-5 (cited on page 44).

[Sam+11] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. "**EnerJ: approximate data types for safe and general low-power computation**". In: *SIGPLAN notices* 46.6 (June 2011), pages 164–174. DOI: 10.1145/1993316.1993518 (cited on pages 107, 152).

[Sap+02] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C-H. Hsu, and U. Kremer. "**Energy-conscious compilation based on voltage scaling**". In: *LCTES/SCOPES'02: proceedings of the joint conference on languages, compilers and tools for embedded systems*. Berlin, Germany: ACM, 2002, pages 2–11. DOI: 10.1145/513829.513832 (cited on page 152).

[Sax10] Eric Saxe. "**Power-efficient software**". In: *Communications of the ACM* 53.2 (2010), pages 44–48. DOI: 10.1145/1646353.1646370 (cited on pages 107, 130, 152).

[SW12] Martin Schoeberl and Andy J. Wellings, editors. **The 10th international workshop on java technologies for real-time and embedded systems, JTRES'12, copenhagen, denmark, october 24-26, 2012**. ACM, 2012.

[Sch+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*. ACM. 2009, pages 169–178 (cited on page 99).

[Sch+14] Marc Schoolderman, Jascha Neutelings, Rody W.J. Kersten, and Marko C.J.D. van Eekelen. "**ECA-logic: hardware-parametric energy-consumption analysis of algorithms**". In: *Proceedings of the 13th workshop on foundations of aspect-oriented languages*. FOAL'14. Lugano, Switzerland: ACM, 2014, pages 19–22. DOI: 10.1145/2588548.2588553 (cited on pages 121, 126).

[Sha00] Natarajan Shankar. "**Combining theorem proving and model checking through symbolic analysis**". In: *Concur*. Edited by Catuscia Palamidessi. Volume 1877. Lecture Notes in Computer Science. Springer, 2000, pages 1–16 (cited on page 43).

[Sha+13] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and AdityaV. Nori. "**A data driven approach for algebraic loop invariants**". In: *Programming languages and systems*. Edited by Matthias Felleisen and Philippa Gardner. Volume 7792. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pages 574–592. DOI: 10.1007/978-3-642-37036-6_31 (cited on page 100).

[SET13] Olha Shkaravska, Marko C. J. D. van Eekelen, and Alejandro Tamalet. "**Collected size semantics for strict functional programs over general polymorphic lists**". In: *Foundational and practical aspects of resource analysis - third international workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, revised selected papers*. Edited by Ugo Dal Lago and Ricardo Peña. Volume 8552. Lecture Notes in Computer Science. Springer, 2013, pages 143–159. DOI: 10.1007/978-3-319-12466-7_9 (cited on page 154).

[SET11] Olha Shkaravska, Marko C.J.D. van Eekelen, and Alejandro 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. Hatfield, UK: Springer-Verlag, 2011, pages 118–137. URL: http://dl.acm.org/citation.cfm?id=2044476.2044483 (cited on page 100).

[SEK09] Olha Shkaravska, Marko C.J.D. van Eekelen, and Ron van Kesteren. "**Polynomial size analysis of first-order shapely functions**". In: *Logic in computer science* 2:10.5 (2009). URL: http://arxiv.org/abs/arxiv:0902.2073 (cited on pages 100, 108).

[SKv10] Olha Shkaravska, Rody W.J. Kersten, and Marko C.J.D. 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*. Edited by Andreas Krall and Hanspeter Mössenböck. ACM Digital Proceedings Series. Vienna, Austria, 2010, pages 99–108 (cited on pages 74, 75).

[SKE07] Olha Shkaravska, Ron van Kesteren, and Marko C.J.D. van Eekelen. "**Polynomial Size Analysis for First-Order Functions**". In: *Typed lambda calculi and applications (TLCA'2007), Paris, France*. Edited by S. Ronchi Della Rocca. Volume 4583. LNCS. Springer, 2007, pages 351–366 (cited on page 100).

[SD04] Aviral Shrivastava and Nikil Dutt. "**Energy efficient code generation exploiting reduced bit-width instruction set architectures (rISA)**". In: *ASP-DAC'04: proceedings of the 2004 Asia and South Pacific design automation conference*. Yokohama, Japan: IEEE Press, 2004, pages 475–477 (cited on page 152).

[Sie02] Fridtjof Siebert. "**Hard realtime garbage collection in modern object oriented programming languages**". PhD thesis. University of Karlsruhe, 2002 (cited on pages 73, 90).

[Šim+00] Tajana Šimunić, Luca Benini, Giovanni De Micheli, and Mat Hans. "**Source code optimization and profiling of energy consumption in embedded systems**". In: *ISSS'00: proceedings of the 13th international symposium on system synthesis*. Madrid: IEEE Computer Society, 2000, pages 193–198. DOI: 10.1145/501790.501831 (cited on page 152).

[SC01] Amit Sinha and Anantha P. Chandrakasan. "**JouleTrack: a web based tool for software energy profiling**". In: *Proceedings of the 38th annual design automation conference*. DAC '01. ACM, 2001, pages 220–225 (cited on page 107).

[Siv+02] Anand Sivasubramaniam, Mahmut Kandemir, N. Vijaykrishnan, and Mary Jane Irwin. "**Designing energy-efficient software**". In: *International parallel and distributed processing symposium*. Los Alamitos, CA, USA: IEEE Computer Society, 2002. DOI: 10.1109/IPDPS.2002.1016580 (cited on page 152).

[SE08] Sjaak Smetsers and Marko C.J.D. van Eekelen. "**LaQuSo: using formal methods for analysis, verification and improvement of safety critical software**". In: *ERCIM news, special theme safety-critical software* 75 (Oct. 2008). Edited by Pedro Merino and Erwin Schoitsch, pages 38–39 (cited on page 44).

[SGP14] Sylvia Stuurman, Bernard E. van Gastel, and Harrie Passier. "**The design of mobile apps: what and how to teach?**" In: *Proceedings of the computer science education research conference, CSERC 2014, Berlin, Germany, November 5-6, 2014*. Edited by Erik Barendsen and Valentina Dagiene. Referenced in the thesis as [BvG-13], see appendix C. ACM, 2014, pages 93–100. DOI: 10.1145/2691352.2691360.

[Sut05] Herb Sutter. "**The free lunch is over: a fundamental turn toward concurrency in software**". In: *Dr. dobb's journal* 30.3 (Mar. 2005) (cited on page 16).

[TSv09] Alejandro Tamalet, Olha Shkaravska, and Marko C.J.D. van Eekelen. "**Size analysis of algebraic data types**". In: *Trends in functional programming*. Edited by Peter Achten, Pieter Koopman, and Marco T. Morazán. Volume 9. Trends in Functional Programming. Intellect, 2009, pages 33–48 (cited on pages 100, 154).

[Tew+08] Hendrik Tews, Tjark Weber, Marcus Völp, Erik Poll, Marko C.J.D. van Eekelen, and Peter van Rossum. **Nova micro–hypervisor verification**. Technical report ICIS–R08012. Robin deliverable D13. Radboud University Nijmegen, May 2008 (cited on page 45).

[Tim16] Jennifer Timian. **Frontal air bags may not deploy**. National highway traffic safety administration. Retrieved from http://www-odi.nhtsa.dot.gov/acms/cs/jaxrs/download/doc/UCM539178/RCAK-16V651-9380.pdf. September 9th, 2016 (cited on page 3).

[Tro15] Jeroen Trommelen. **Ook 'groene' koelkasten voorzien van sjoemelsoftware**. Volkskrant. Retrieved from http://s.vk.nl/s-a4165935/. October 18th, 2015 (cited on page 2).

[TC12] Shiao-Li Tsao and Jian Jhen Chen. "**SEProf: a high-level software energy profiling tool for an embedded processor enabling power management functions**". In: *Journal of systems and software* 85.8 (2012), pages 1757–1769 (cited on page 107).

[UN16] Department of Economic United Nations and Population Division Social Affairs. **World population prospects: the 2015 revision**. 2016. URL: http://esa.un.org/unpd/wpp/Download/Standard/Population/ (cited on page 5).

[VS11] Freek Verbeek and Julien Schmaltz. "**Hunting deadlocks efficiently in microarchitectural models of communication fabrics**". In: *Proceedings of the international conference on formal methods in computer-aided design*. FMCAD '11. Austin, Texas, 2011, pages 223–231 (cited on pages 48, 67).

[VS12] Freek Verbeek and Julien Schmaltz. "**Automatic generation of deadlock detection algorithms for a family of micro architectural description languages**". In: *IEEE international high level design validation and test workshop (HLDVT'12)* (Nov. 2012) (cited on page 48).

[Ver+06] Manish Verma, Lars Wehmeyer, Robert Pyka, Peter Marwedel, and Luca Benini. "**Compilation and simulation tool chain for memory aware energy optimizations**". In: *SAMOS*. 2006, pages 279–288 (cited on page 152).

[Vis+03] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. "**Model checking programs**". In: *ACM transactions on design automation of electronic systems* 10.2 (2003), pages 203–232. DOI: 10.1023/A:1022920129859 (cited on page 44).

[Wan+10] Shengyi Wang, Zongyan Qiu, Shengchao Qin, and Wei-Ngan Chin. "**Stack bound inference for abstract Java bytecode**". In: *Proceedings of the 2010 4th IEEE international symposium on theoretical aspects of software engineering*. 2010, pages 57–66. DOI: 10.1109/TASE.2010.24 (cited on page 102).

[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. Copenhagen, Denmark: ACM, 2012, pages 125–134. DOI: **10.1145/2388936.2388958** (cited on pages 71, 98).

[Weg75] Ben Wegbreit. "**Mechanical program analysis**". In: *Communications of the ACM* 18.9 (1975), pages 528–539 (cited on page 86).

[Wil+08] Reinhard Wilhelm et al. "**The worst-case execution-time problem–overview of methods and survey of tools**". In: *ACM transactions on embedded computing systems* 7.3 (May 2008), 36:1–36:53. DOI: **10.1145/1347375.1347389** (cited on pages 101, 108).

[YLL06] Yi-Ping You, Chingren Lee, and Jenq Kuen Lee. "**Compilers for leakage power reduction**". In: *ACM transactions on design automation of electronic systems* 11.1 (2006), pages 147–164. DOI: **10.1145/1124713.1124723** (cited on page 152).

[Zha+03] W. Zhang, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and V. De. "**Compiler support for reducing leakage energy consumption**". In: *DATE'03: proceedings of the conference on design, automation and test in Europe*. Washington, DC, USA: IEEE Computer Society, 2003 (cited on page 152).

[Zhu+09] Dmitry Zhurikhin, Andrey Belevantsev, Arutyun Avetisyan, Kirill Batuzov, and Semun Lee. "**Evaluating power aware optimizations within GCC compiler**". In: *Grow-2009: international workshop on GCC research opportunities*. 2009 (cited on pages 107, 130, 152).