RWTH Aachen

Department of Computer Science
Technical Report

Synthesis of State Space Generators for Model Checking Microcontroller Code

Dominique Marcel Gückel

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2014-15

RWTH Aachen · Department of Computer Science · November 2014

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

Synthesis of State Space Generators for Model Checking Microcontroller Code

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Dominique Marcel Gückel
aus
Greven / Nordrhein-Westfalen

Berichter: Universitätsprofessor Dr.-Ing. Stefan Kowalewski
Universitätsprofessor Dr. rer. nat. Rainer Leupers

Tag der mündlichen Prüfung: 15. Oktober 2014

Dominique Marcel Gückel
Lehrstuhl Informatik 11
gueckel@embedded.rwth-aachen.de

Aachener Informatik Bericht AIB-2014-15

Herausgeber: Fachgruppe Informatik
RWTH Aachen University
Ahornstr. 55
52074 Aachen
GERMANY

ISSN 0935-3232

Abstract

Creating software for embedded systems requires rigid quality measures. The reason for this is that errors in the software may have very expensive or even disastrous consequences. This gives rise to the use of formal methods for software verification, such as model checking, theorem proving, and static analysis.

Many embedded systems rely on either application-specific circuits, reconfigurable logics, or microcontrollers. Manufacturers of microcontrollers typically offer a wide variety of devices based on the same core architecture, which are equipped differently and thus offer different functionality. Furthermore, some tool chains exist that allow developers not only to choose from such off-the-shelf devices, but to customize them for specific kinds of tasks. In some cases, this may go to the extent of actually designing new architectures.

It is precisely this wide variety of available devices that complicates the use of automated verification. Tools need to be adapted to a new platform, or even recreated in case they should be implemented in a too hardware-dependent way.

The topic this thesis deals with is the reduction of the necessary effort for adapting a verification tool to new microcontrollers. To this end, we designed a language for describing microcontrollers, SGDL, and a compiler for translating such descriptions into operative simulators and static analyzers. We based our work on [MC]SQUARE, which is a platform for model checking and static analysis of assembly code software.

In order to counter the state explosion problem, it is also necessary to include abstractions in generated simulators. We illustrate, on a number of abstraction techniques, how they can be integrated into the approach and whether they can be generated either partly or entirely.

A number of case studies concerning the implementation of simulators with our new language is presented. Additionally, we examine the effectiveness of the aforementioned abstractions that are integrated into the generated simulators, and compare the results to those obtained when using manually implemented simulators.

Zusammenfassung

Die Implementierung von Software für eingebettete Systeme erfordert strenge Maßnahmen zur Qualitätssicherung. Der Grund hierfür ist, daß Fehler in dieser Art von Software sehr teuer werden oder gar katastrophale Auswirkungen haben können. Hieraus motiviert sich der Einsatz formaler Methoden zur Verifikation von Software, wie etwa Model-Checking, Theorem Proving, oder statischer Analysen.

Viele eingebettete Systeme basieren entweder auf applikationsspezifischen Schaltungen, rekonfigurierbarer Logik, oder Mikrocontrollern. Typischerweise bieten die Hersteller von Mikrocontrollern eine Vielzahl verschiedener Geräte mit der gleichen Kernarchitektur an, die sich im Hinblick auf die verfügbare Ausstattung unterscheiden. Darüber hinaus gibt es Software-Werkzeuge, mit denen Entwickler bestehende Mikrocontroller an ihre eigenen Anforderungen anpassen oder sogar bei Bedarf neue Architekturen entwickeln können.

Aus der Vielfalt an verfügbaren Geräten ergeben sich allerdings auch Nachteile. Die Möglichkeit zur automatisierten formalen Verifikation wird eingeschränkt, weil die dafür benötigten Werkzeuge zunächst einmal an jede neue Plattform angepaßt werden müssen. Je nach Werkzeug kann dies auch bedeuten, daß eine komplette Neuimplementierung nötig wird.

Das Thema dieser Dissertation ist die Reduktion des Aufwandes, der nötig ist, um ein Verifikationswerkzeug an neue Mikrocontroller anzupassen. Zu diesem Zweck haben wir eine Sprache zur Beschreibung von Mikrocontrollern entworfen, SGDL, und einen Compiler implementiert, der Beschreibungen in dieser Sprache übersetzen kann in funktionsfähige Simulatoren und statische Analyse-Werkzeuge. Unsere Arbeit basiert auf [MC]SQUARE, einer Plattform für das Model-Checking und die statische Analyse von Assembler-Code.

Um das Problem der Zustandsexplosion zu begrenzen, ist es erforderlich, daß die generierten Simulatoren auch Abstraktionstechniken unterstützen. Anhand einer Reihe von Abstraktionstechniken zeigen wir, wie diese in den vorgestellten Ansatz integriert werden können, und inwieweit es möglich ist, sie ganz oder teilweise automatisch zu erzeugen.

In unseren Fallstudien demonstrieren wir die Implementierung von Simulatoren mit unserer neuen Sprache. Darüber hinaus untersuchen wir die Wirksamkeit der bereits genannten Abstraktionstechniken in den generierten Simulatoren, und vergleichen die Ergebnisse mit denen von handgeschriebenen Simulatoren.

Acknowledgments

First of all, I would like to thank Prof. Dr.-Ing. Stefan Kowalewski for supervising my thesis. The time I spent at his institute, the Embedded Software Laboratory of RWTH Aachen University, has been extremely helpful, and provided me with a lot of insights into several topics. I would also like to thank Prof. Dr. rer. nat. Rainer Leupers for being the second supervisor of my thesis. Furthermore, my thanks go to Prof. Dr. Ir. Joost-Pieter Katoen and Prof. Dr. Horst Lichter for being part of the examination board.

Thanks to Dr. Bastian Schlich, who introduced me into research, and encouraged me to start out as a doctoral student in the first place. Also thanks to the other researchers in the [MC]SQUARE project, namely Dr. Jörg Brauer, Sebastian Biallas, and Volker Kamin.

Thanks also to my former colleagues at the Embedded Software Lab, who have made my time there both interesting and delightful. Special thanks go to Dr. Andre Stollenwerk.

I am especially grateful for the assistance provided by Ben Titzer and Jens Palsberg in the early days of my research project.

Originally, I started my research as a scholarship holder in the DFG Research Training Group 1298 *AlgoSyn*, which funded me for the first three years at the Embedded Software Laboratory. The environment provided by AlgoSyn, i.e., the tutorials, mini lectures, and regular presentations, was really beneficial for my research. Most of all, I am extraordinarily grateful that AlgoSyn funded most of the student researchers who assisted me in my work.

This research would not have been possible without the contribution of several students. First and foremost, my thanks go to Ivica Bogosavljevic. Furthermore, I would like to thank Florian Caron, Christian Dehnert, Richard Musiol, Irfan Simsek, Norbert Wiechowski, and all other team members who have contributed directly or indirectly.

Thanks also to my superiors at IVU Traffic Technologies for granting me the vacation required for preparing the oral exam. I really appreciate this, as the request was granted on very short notice.

Finally, I would like to thank my family and my friends for their support of all kind during the long years of research!

Aachen, November 2014

Contents

1 Introduction

Nowadays, we are surrounded by countless computer systems. These systems are not necessarily immediately recognizable as such because they are integrated into other systems, and therefore, are called *embedded systems*. People normally do not interact directly with the computer system, but with the *embedding system*. In many cases, the functionality of the embedding system would not be possible without the embedded system. Typical tasks for these devices are to measure data from sensors interfacing to their environment, and respond to such input by controlling actuators that are part of the embedding system.

A very common field of application for embedded systems are cars. Several assistance systems, like anti-lock braking systems, electronic stability programs, or cruise control, are realized by means of embedded systems. In this setting, the car is the overall embedding system, which has several embedded systems integrated into it. For such tasks, it is also necessary for embedded systems to be networked, as for instance the data acquired by the systems that measure the speed of individual wheels need to be joined. Hence, even though embedded systems are typically integrated into components, they are not necessarily isolated from each other.

Another scenario for applying embedded systems are industrial plants. In such settings, the embedded systems are tasked, for instance, with controlling conveyor belts, robotic arms, and other devices necessary for production. It is also possible to control entire production processes at the field level.

There are several categories for distinguishing embedded systems. The aforementioned use of embedded systems in cars is typically referred to as an example of *product automation*, whereas their usage in plants is called *production automation*. Product automation implies that the embedded system, or systems, must not exceed given cost thresholds, as they have to be produced in quantities. Opposed to this, production plants are not built nearly as many times as products, therefore the cost for individual embedded systems is not as important. Due to this difference, the type of systems used in these settings differ. For products, frequently used devices include microcontrollers, and several types of programmable logic, e.g. Field Programmable Gate Arrays (FPGAs). Plants, on the contrary, typically rely on more elaborate and expensive devices, such as Programmable Logic Controllers (PLCs), which already provide some critical functionality required in their respective fields of application. For instance, PLCs are able to guarantee the completion of operations within a certain time, i.e., are capable of real time computations.

Creating similar functionality on microcontrollers is typically also possible, but requires additional effort, including the effort for establishing the correctness of such capabilities.

The correct operation of products containing embedded systems depends heavily on the correctness of the hardware and software of which the embedded system consists. Furthermore, errors in the software of such systems are particularly acute because it is virtually impossible for the vendor to apply a patch. Unlike in general-purpose computing, the systems are typically not accessible to the end user, and in safety-critical systems like cars, it is not even desirable to expose an interface to the user. Hence, to fix mistakes, it is necessary to recall such devices, or even to replace them. Thus, it is vital for a software developer to detect errors before the system is shipped to the customer, and, to this end, apply rigid verification techniques.

1.1 Objectives

Embedded systems used in products are often based on processor-based architectures. In many cases, these processors are stand-alone computers on a single chip, so called systems-on-chip (SoC). This means that they are equipped with some on-chip memory, peripherals, and I/O ports for interfacing to the outside world. Microcontrollers fall into this category of devices.

Developers wanting to create a system based on microcontrollers have a wide range of devices at their disposal. Many device manufacturers sell microcontrollers, e.g. Intel, Renesas, Infineon, PIC, and Atmel. Each of these provides one or more families of controllers, and within these families, very different devices, ranging from cheap, sparsely equipped chips to complex devices with many on-chip peripherals. The latter sometimes eliminate the need for integrating further chips into a design, for instance when a microcontroller provides communication controllers for busses such as CAN. In some cases, developers can even rely on tools to create customized microcontrollers, which allows them to remove unnecessary parts and add needed ones, thus reducing the overall cost for a design.

While this wide range of devices certainly has advantages, it complicates the development of reliable software. As long as developers build their system from off-the-shelf devices from the existing manufacturers, they can typically use the development tools provided by the respective device manufacturer. In most cases, this includes assemblers, linkers, and C compilers. Developing software in C provides at least a certain degree of hardware abstraction, thus easing the porting of software in case a chosen platform should prove to be no longer suitable. However, when creating customized hardware, there is no compiler support yet. The compiler needs to be created. Software quality ensurance faces a similar problem, as existing tools also need to be rewritten or retargeted to the new platform.

Some tool chains exist that facilitate the creation of compilers, assemblers, and similar tools, for new platforms. To the best of our knowledge, however, this has so far not been the case with regard to model checkers, which are a special kind of tool for detecting errors. It is therefore our goal to create a means for doing so, such that it becomes possible to retarget a model checking tool to new platforms with time efforts that would make the procedure feasible for industrial applications.

1.2 Contribution

In this thesis, we make the following contributions:

- We describe a system for generating state space generators from a hardware description. The description can be created by developers in a language, SGDL, which we designed with the goal of accelerating the development of such generators.

- To illustrate the advantage of creating state space generators with our tool, instead of implementing them manually in Java, we present three implementation-related case studies. In these case studies, we used our tool chain to create simulators for the Atmel ATmega16 and ATmega644 microcontrollers and for the 8051, a member of the Intel MCS-51 family of microcontrollers.

- We show that even when synthesizing simulators instead of manually fine-tuning them, it is possible to integrate abstraction techniques. In this respect, we also prove that certain abstractions can be added automatically, without the developer having to be familiar at all with the concept.

- In a series of case studies consisting of executable programs for all three platforms, we compare several qualities of synthetic simulators with those of existing handcrafted simulators, some of which have been optimized over years. Our main result from these case studies is that synthetic simulators can compete with handcrafted code in most aspects.

- Finally, we develop ideas on how to further improve the generation of automatic abstractions. With this purpose in mind, we have created a tool for static analysis of hardware descriptions, which can derive certain information about an architecture automatically. The derived information can then be used for lifting the given concrete syntax of, for instance, instructions, to an abstract one.

1.3 Outline

The rest of this thesis is structured as follows. Chapter 2 presents preliminary work relevant for the topic of this thesis. Next, Chapter 3 discusses related work and certain requirements, from which we deduced the necessity of our research. A brief overview of the synthesis system that we created during that research is given in Chapter 4. Following, Chapters 5 to 7 detail the structure of the synthesis system, starting with the description of the input in a domain-specific language, and ending with a discussion of the generated code. Chapter 8 explains the abstraction techniques which we integrated into the synthesis system as an improvement. Next, two case study chapters first present the implementation of simulators using our new language (Chapter 9), and then illustrate how the generated simulators can be applied to microcontroller programs for the respective platforms (Chapter 10). Finally, Chapter 11 concludes this thesis.

1.4 Bibliographic Notes

Parts of the research presented in this thesis has been previously published in some of our earlier publications. The idea of retargeting a model checker by means of a hardware description was first described in our contribution to a doctoral symposium [26], and in a journal article [79]. A first operative version was presented in [29]. Improvements over the initial version, which introduced early versions of some abstraction techniques, were the topic of another publication [28]. This publication also included a first version of our implementation of an abstraction called *path reduction* (originally created by other authors). We contributed to enhancing this technique in another publication [10], the results of which were first obtained using one of our generated simulators, before backporting them to the simulator used in the publication.

Related work by other authors is presented in the individual chapters, either within the presentation where appropriate, or in a concluding section dedicated to related work.

2 Preliminaries

Within this chapter, we introduce the background that this thesis relies upon. The main tiers of our work are model checking, static analysis, and compiler construction. Model checking is a technology for verifying systems, and is used, in our setting, for verifying properties of microcontroller programs. Static analysis is a technique that can serve for various purposes. We use it in both for supporting the model checker and in a more classical setting, that is, in a compiler for processing the input.

The rest of this chapter is structured as follows. First of all, in Sect. 2.1, we define notations that are frequently used throughout the thesis. In Sect. 2.2, we explain Kripke structures, which we use to model systems. The next section then introduces temporal logics, with a focus on Computation Tree Logic (CTL). In Sect. 2.4, we introduce the concept of model checking, and illustrate how model checking combines the previously defined temporal logics and Kripke structures. An overview of static analysis is given in Sect. 2.5. Finally, an introduction into the [MC]SQUARE model checking tool concludes this chapter.

Our descriptions of Kripke structures, temporal logics, and model checking, are based on Clarke et al. [16], and on Baier and Katoen [9]. For static analysis, we refer to Nielson et al. [49]. The description of the [MC]SQUARE model checker is based on Schlich [68] and on our own contributions to this project.

2.1 Notations

2.1.1 Number Representations

For representing numbers by different bases, we abide by the notation used by Oberschelp and Vossen [51]. Hence, we define the following:

1. A number $z \in \mathbb{N}$ is a short-hand representation of a sum:

$$z = (z_{n-1} z_{n-2} \dots z_0) = \sum_{i=0}^{n-1} z_i b^i$$

   where $n, z_i, b \in \mathbb{N}$. n is the length of the representation and b is a base, also called a *radix* [78]. Unless specified otherwise, we use $b = 10$.

2. For representing a number by a base other than 10, we enclose the number by parentheses and append the value for b at the lower right parenthesis:

$$(z)_b$$

Hence, for instance $10 = (10)_{10} = (1010)_2$.

3. Hexadecimal values are prefixed with *0x*, for example *0xff*.

4. Binary values are prefixed with *0b*, for example *0b1011*.

2.1.2 Fonts

Tool names are printed in uppercase, e.g. SGDL-STA. The names of temporal logics are printed in the same style, e.g. CTL. Java class names are printed in a slanted font, such as *Analyzer*. Finally, for code snippets we use a typewriter style, e.g. `execute`, and for longer code fragments we use a syntax highlighting appropriate for the respective language.

2.2 Kripke Structures

Kripke structures, or labeled transition systems, are a means for modeling concurrent systems.

Definition 2.1. Kripke structure
Let AP be a set of atomic propositions. Let $Pot(M) := 2^M$, i.e., the powerset of M. A Kripke structure K over AP is defined as $K = (S, S_0, R, L)$, where

- S is a finite set of states
- $S_0 \subseteq S$ is the set of *initial* states
- Act is a set of actions
- $R \subseteq S \times Act \times S$ is a transition relation
- $L : S \to Pot(AP)$ is a function that maps each state to the set of atomic propositions valid in that state. L is also called *labeling function*.

This definition, which defines a set of actions that can be associated with transitions, is taken from Baier and Katoen. Actions are labels for transitions. The definition used by Clarke et al. does not require such a set, even though they sometimes also use labelled transitions.

Kripke structures extend the concept of propositional logics by states, which are also called universes. An *atomic proposition* no longer is either true or false,

```
r := initial node of the Kripke structure
visit(r)

visit(v)
  for all successors v' of v
    add v' to the tree
    add edge (v,v')
    visit(v')
```

Listing 2.1: Algorithm for unwinding a Kripke structure

but may have different values depending on the current state. Hence, using Kripke structures, it is possible to describe systems that evolve over time, such as automata, or programs modifying the memory of a computer device.

For reasoning about Kripke structures, we usually transform them into a tree representation, which is called a *computation tree* [16]. Computation trees are constructed by *unwinding* [16] the Kripke structure. We sketch the required algorithm in pseudo code in Listing 2.1. Note that the resulting tree may be infinite, as the Kripke structure may contain loops. Thus, the tree construction algorithm may not terminate. Each of the paths in the tree corresponds to a possible sequence of states in the Kripke structure. Such sequences are called *executions* by both Clarke et al. and Baier and Katoen.

2.3 Temporal Logics

Formulas in traditional propositional logics consist of Boolean statements called *atomic propositions*, and of conjunctions of such statements by means of operators such as *AND*, *OR*, and negation. Atomic propositions can be either true or false, but is is possible to map more general expressions to these two truth values. For instance, it is legal to express that a variable v has a given value of 12 by an atomic proposition $v = 12$. The satisfiability problem for propositional logics (SAT) is decidable, though it is known to be NP-complete (Cook's theorem, as described in [3, 35]; a proof is given by Hopcroft et al. [34]).

A general assumption in propositional logics is that there is just a single universe, in which formulas can be evaluated to either true or false. This restriction complicates the modeling of time-related properties, e.g. specifying that a variable sequentially takes several given values. Kripke structures allow the modeling of such sequences, as each of the states can be seen as a certain point in time, and may have its own set of true and false atomic propositions. Furthermore, states in Kripke structures may also have more than one successor, corresponding to multiple, possibly different, evolutions of a system. Properties that are satisfied on one set of paths need not necessarily be true on other sets. Hence, to benefit from these modeling possibilities, a logic should provide

- path quantifiers: a set of paths (e.g. *all* or *at least one*) satisfy a property
- recursion: conventional propositional and also predicate logic can only specify properties proportional in the length of the formula. In order to describe properties such as *eventually (after finitely many steps)* or *always*, it is necessary for the logic to provide a recursional description mechanism.

These restrictions are remedied by temporal logics. Two of these are LTL and CTL, which we describe in more detail in the following sections. Both are special cases of a more generalized logic called CTL*, from which they can be derived by imposing certain restrictions on the usage of path quantors and temporal operators.

An important aspect of temporal logics is that the term *temporal* does not relate to a precise timing. These logics typically do not provide the means to describe properties such as *whenever p holds, after two seconds q holds.* Instead, the term temporal rather refers to a relative order of events (cf. [9]), meaning that is it possible to describe sequences of events while omitting the exact period of time elapsing between them.

Definition 2.2. Temporal operators
A temporal operator is one of $\{X, F, G, U\}$ with the following semantics:

- $X\Phi$ means that the formula Φ holds starting from the next state, seen from a current state s in a Kripke structure.
- $F\Phi$ means that the formula Φ *eventually* holds.
- $G\Phi$ means that the formula Φ holds globally, i.e., in all states.
- $\Phi U \Psi$ means that the formula Φ holds in all states along the path until a state is reached in which Ψ holds, and there has to be such a state.

Definition 2.3. Path quantifiers
A path quantifier is one of $\{A, E\}$ with the following semantics:

- $A\Phi$ means that the formula Φ has to hold on all paths starting from the current state.
- $E\Phi$ means that there has to be at least one path, starting from the current state, on which the formula Φ holds.

2.3.1 LTL

Linear time temporal logics (LTL) allow formulas to express properties along paths of arbitrary length. The general assumption is that given a state s in a Kripke structure $\mathcal{K}$ and a formula Φ, there can be only one future of s, i.e. a sequence of successor states $(s', s'', \ldots)$. In case the unwinding of $\mathcal{K}$ yields a tree containing nodes with multiple successors, the resulting paths are considered separately. All LTL formulas are implicitly preceded by an A path quantifier, hence

$$\mathcal{K}, s \models \Phi \Leftrightarrow \Phi \text{ evaluates to true for each of the paths starting at } s.$$

As the model checking tool [MC]SQUARE, which we used for our research, started as a CTL model checker without support for LTL, we conducted our research on the basis of the CTL model checking algorithm. At the time of this writing, the tool had recently been extended to additionally support LTL. Therefore, we do not provide further details on LTL, but only use this short reference to point out some of the characteristics of CTL in the next section.

2.3.2 CTL

Computation Tree Logic (CTL) is a branching time temporal logic. CTL can be derived from CTL* by requiring that each path quantor must be combined with exactly one temporal operator and vice versa.

Definition 2.4. Syntax of CTL
Let AP be a set of atomic propositions. The set of CTL formulas over AP is defined as follows:

- p is a CTL formula $\forall p \in AP$
- If Φ is a CTL formula, then $\neg\Phi$ is a formula
- If Φ is a CTL formula, then
  - $AG\Phi$, $AF\Phi$, $AX\Phi$
  - $EG\Phi$, $EF\Phi$, $EX\Phi$

  are CTL formulas, where A and E are path quantors, and G, F, X are temporal operators, as declared above
- If Φ, Ψ are CTL formulas, then
  - $A(\Phi U \Psi)$
  - $E(\Phi U \Psi)$

  are CTL formulas, where A, E are path quantors and U is a temporal operator

- If Φ, Ψ are CTL formulas, then
  - $\Phi \vee \Psi$
  - $\Phi \wedge \Psi$
  - $\Phi \rightarrow \Psi$

  are CTL formulas

Additionally, there is a *release* operator R, but that operator can be expressed by means of the other operators (cf. Clarke). Clarke also describes how to represent any CTL formula using only an existential subset, that is, using formulas containing only the operator and quantifier types EX, EG, and EU.

Unlike LTL, which specifies properties over single paths of the unwinding tree and implicitly requires these to be true for all such paths, CTL specifies properties over different paths starting from specific nodes in the tree. There is no implicit A quantifier.

Example 2.1. ***Examples for CTL formulas***

- $AG\neg(\Phi \wedge \Psi)$ *: on all paths,* Φ *and* Ψ *are mutually exclusive*
- $EF\Phi$ *: there is a path on which* Φ *eventually holds*
- $A(r_1 = 10\, U\, r_{17} \geq 20)$ *: on all paths, the atomic* $r_1 = 10$ *must evaluate to true (i.e.,* r_1 *has a certain value) until* r_{17} *is larger or equal to* 20

2.4 Model Checking

Model checking is a formal method for verifying systems. Starting from a model of a system and a specification, a model checking algorithm can automatically verify whether the system satisfies its specification. In case the system is found to violate the specification, it is usually possible to create a counterexample, that is, a trace of events leading to the state of the system in which the specification is no longer satisfied. The procedure as such requires large amounts of memory, and it is possible that the model checking algorithm has to abort verification. In that case, it is possible that the result is only valid for the subset of system states visited so far, or that there is no result at all.

The underlying idea of model checking is that of exhaustive exploration of all reachable system states. It is possible to separate two phases, *exploration*, also called state space building, and *verification*, though some model checkers entwine these with each other, for instance in an alternating mode of operation that allows for early pruning of certain paths. In the state space building phase, the algorithm uses a given a set of initial states of the system, and checks this set for possible evolutions. All steps the system may take are examined, leading to a new set of states. These are then the basis for further exploration, and are again checked for possible evolutions. Eventually, when no more new states are created, state space building terminates, and the actual verification phase starts. In this phase, depending on the (means of) specification, the model checking algorithm typically searches the state space for patterns of states, such as chains on which all states satisfy some property. Finally, depending on the result of the verification phase, the aforementioned counterexample may be generated.

The following parts of this section focus on the use of Kripke structures and temporal logics in the context of model checking.

2.4.1 Specification

Prior to any formal verification, it is necessary to capture the requirements for a given system in a sound manner that is amenable to algorithmic uses. Natural language is not suitable for this purpose, as it is usually too vague and ambiguous. Even the result of a proper requirements analysis (e.g. [74]) does not provide the mathematically rigorous description of system properties that is needed. Hence, the first step towards formal verification by means of model checking is to translate the natural-language specification into a formula in some temporal logic, and to represent that formula in the input format of the model checking tool of choice.

The temporal logic introduced in detail the last section, CTL, allows the description of evolutions of a system. In our setting, which is formal verification of embedded software, the atomics are statements about the memory contents of a microcontroller. Paths relate to the sequence of steps the microcontroller takes while executing a program, that is, a statement involving temporal operators specifies possible computations, while path quantors impose requirements such as *all possible computations starting from here* or *at least one computation starting from here* must satisfy a given requirement.

2.4.2 Modeling

Using Kripke structures, it is possible to describe a system that may take several possible states. For software verification, the set of initial states is equivalent to the state the machine has before execution of the program starts. Each instruction then correlates to at least one transition from one state to another. That is, an instruction *modifies* the system state. In the terms of a Kripke structure, the state reached after execution of an instruction is typically labelled differently: a different program or line counter, and a different set of atomic propositions. Hence, statements on the value of such atomics may also evaluate differently for different states.

Modeling systems can be handled either manually or automatically. For general purpose model checkers such as SMV [16], NuSMV [14], and UPPAAL [39], the input describes a set of automata, represented in the idiom understood by the tool. Such an approach has various advantages, but also a number of disadvantages. For one, it is possible to deliberately add or omit information about the behavior of the actual system, which helps counter the state explosion problem (cf. the next section and Chapt. 8). On the downside, this also means that the modeling has to be done manually, which may become difficult for very large or complex systems, or when the modeling step has to be done anew. The latter typically becomes necessary when model checking is successful, i.e., when an error is found, and the problem is fixed in the system under test. Then, the new system candidate has to be checked again, which requires a new or modified system model.

In the case of the already mentioned software model checking, the likelihood of such changes is very high, considering that modifying a program is simple. However, this is not necessarily a problem for model checking, as the changes of machine state associated with instructions in any programming or machine-level language are predictable, either from the language definition or the processor data sheet. Thus, it is possible to automate the modeling step for software.

Examples for tools that provide automatic modeling are Java PathFinder [33, 83] (Java source code and Java bytecode), MoonWalker [1] (CIL bytecode of Microsoft .NET programs), CBMC [17] (ANSI C), and [mc]square [68] (microcontroller binary code). The internal approaches in these tools vary profoundly, as for instance later versions of Java PathFinder and [mc]square interpret machine code for building state spaces, whereas CBMC translates the source code of the program into a boolean formula and delegates the actual checking to a SAT solver.

2.4.3 Verification

In the verification phase, the formalized specification and the system model serve as input for the model checking algorithm. Fig. 2.1 illustrates the procedure. Depending on the algorithm used, either a subset or the entire reachable state space is created and checked for property violations. Eventually, there may be three possible results. One of these is that the system model satisfies the formula, which means that no property violation was found. Another possible result represents the opposite case: some system state or a set of states violates the formula. In that case the model checker may attempt to create a counterexample (see below). Finally, the third possible situation is that the model checking algorithm was still active but ran out of memory, which resulted in a premature end of the verification. Such a situation may occur due to the state explosion problem, which refers to a rapid growth of the number of sytem states. This growth may be exponential in

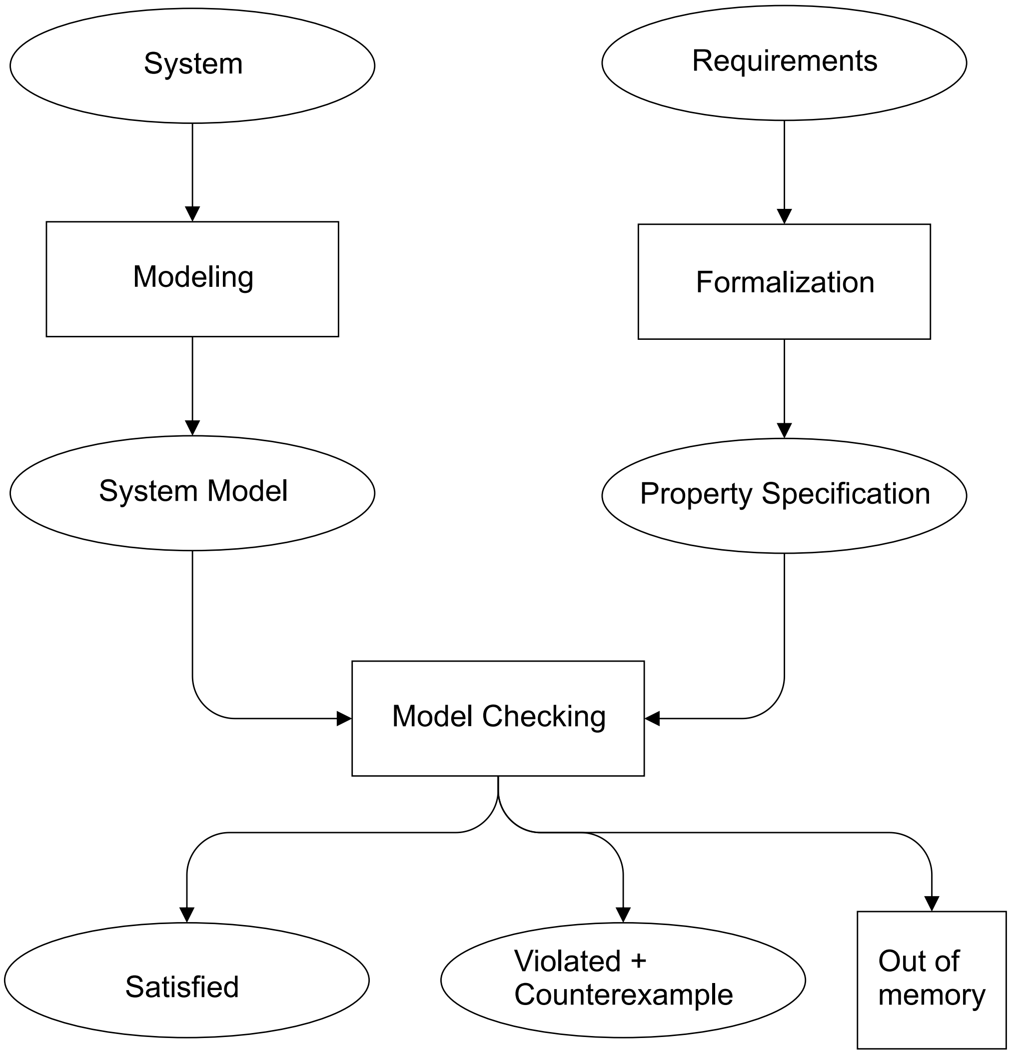

Figure 2.1: Model checking procedure as described by Baier and Katoen [9] and Schlich [68]

the number of involved concurrent components in the system model, for instance the number of states of individual automata in an automaton-based system model.

Counterexamples illustrate the causes of errors, that is, property violations. They consist of a sequence of steps starting from an initial state, up to a state where a property from the specification no longer holds true. Hence, they provide an invaluable insight into problematic system behavior, which helps the developer reduce the number of remaining errors. This is fundamentally different from the outcome of testing, where only the effects of errors become visible, but not their origins.

Creating a counterexample is possible under certain circumstances, and its complexity depends highly on the logics used for specification. For automata-based LTL model checking [9], creating a counterexample imposes virtually no additional cost, as the necessary steps are part of the verification already. However, LTL verification is PSPACE-complete [9, 16], and therefore already very complex. On the other hand, CTL model checking does not automatically create a counterexample, which is why creating one requires additional effort. This is alleviated by the possibility to model check formulas with less time and space complexity than for LTL. Still, it is possible that some abstractions used during the verification process require special treatment of the state space created by the actual model checking algorithm in order to extract a counterexample from it, or that such a creation is altogether infeasible. One example for such an abstraction is path reduction (cf. Chapt. 8).

2.5 Static Analysis

Static analyses gather information about a program without actually executing it. Hence, the object of the analysis is not the binary program, as would be the case for dynamic approaches like testing. Instead, static analysis focuses on the source code. A variety of techniques exist, and some authors also include manual approaches such as code inspections, reviews, or code metrics (e.g. Liggesmeyer [40]). In this section, however, we do not consider these. Instead, we focus on automated techniques as described by Nielson et al. [49] and Aho et al. [2]. This section is based primarily on these two sources.

2.5.1 Motivation

Static analyses are extensively used in compilers, with the intention of optimizing a program at compile time. It is possible, for instance, to detect unnecessary and repeated computations, unreachable code, or loop-invariant code, i.e., code that could be moved out of a loop without altering the program semantics. Some examples for such uses are described below in this section.

Static analysis can also be used in stand-alone tools. Possible fields of application are vulnerability scanning [2] and the examination of timing behavior. For the latter and some additional examples, we refer to the section on related work in Chapter 6.

2.5.2 Control Flow Analysis

The goal of control flow analysis is to create a control flow graph (CFG) from a program. This graph illustrates the relation between instructions in the program, that is, possible sucessors, predecessors, loops and branches. Data flow analyses, which we describe below, typically operate on control flow graphs.

Creating the control flow graph can itself already be a challenging analysis. This is especially acute when analysing binary code. Under such circumstances, the successor relation is not always clear, for example due to the existence of indirect jumps. In some cases, like the analysis of potentially malicious code, the program may also be designed such that the control flow is not obvious.

2.5.3 Data Flow Analysis

As described by Nielson et al., data flow analyses can be conducted given

- a CFG, or flow relation
- a suitable lattice for representing the desired program properties
- a set of rules describing how program locations modify the analysis information

Then, it is possible to derive an equation system from the graph. The solution of the equation system can be obtained by a fixed point iteration.

Some examples of data flow analyses, and how they relate to specific optimizations in compiler construction, are listed below. These are taken from [2]:

- *Live Variable Analysis*: computes, for all program locations, the set of variables whose values are going to be read. Any variable not contained therein will either be overwritten before it is read the next time, or it will never be read again before the end of the program. Such variables are called dead. The corresponding code optimization employed by a compiler is called *Dead Code Elimination*: assignments to a dead variable can be removed from the program without altering the program semantics.

- *Reaching Definitions Analysis*: computes, for all program locations and all variables within scope, the locations at which these variables were last assigned a value (i.e., were defined). This analysis allows to determine potential values of variables at given program locations. In case a variable is found to have only a single possible value, it can be used by an optimization called *constant propagation*, which replaces variables by their known value.

- *Available Expressions Analysis*: computes the set of expressions that are certain to have been computed on the way to any program location. For an expression to remain in the set for a location, it must still evaluate to the same value, that is, no subexpression or variable contained therein may have changed. Using this analysis, it is possible to reuse previously computed expressions without computing their value again. The corresponding optimization is called *Common Subexpression Elimination*.

2.6 [mc]square

[MC]SQUARE is a model checker for microcontroller binary code, which was originally developed by Schlich [68] at the Embedded Software Laboratory of RWTH Aachen University in Germany. Over the years, several students of RWTH Aachen University as well as external partners contributed to it.

[MC]SQUARE operates on disassembled binary programs and follows a hardware-dependent approach: instead of providing a general purpose input language as an interface to the user, [MC]SQUARE can directly load the files intended to be deployed to the target microcontroller. Currently, it supports the Atmel ATmega16 and ATmega644, Intel MCS-51, and the Renesas R8C \23. Furthermore, it supports programs for Programable Logic Controllers (PLCs) written in Instruction List (IL) [84]. Previously supported platforms, for which support has been discontinued in [MC]SQUARE, comprise the Atmel ATmega128 and Infineon XC167. Concerning temporal logics used for specifying properties, [MC]SQUARE supports CTL. Formulas may contain atomic propositions about virtually any memory location occurring in a microcontroller, including registers and main memory.

An overview of the model checking process implemented in [MC]SQUARE is shown in Fig. 2.2. Programs to be checked for property violations serve as the input to the tool, along with a formula. In a first processing step, both of these are parsed by suitable parsers and then passed to the core component of [MC]SQUARE, which consists of the actual model checking algorithm steering the verification, a simulator, and the state space. After termination of the model checking algorithm, and depending on the result, the counterexample generator may be triggered to extract a counterexample or a witness from the statespace created so far.

Programs are given as binary files in container formats such as ELF [80], Intel HEX, or Motorola S-Record. It is possible though not necessary to also provide the corresponding C source files, which, if present, can be used by [MC]SQUARE to create counterexamples that directly relate to source code lines. Otherwise, [MC]SQUARE is still able to point out problems with the input program, but will have to display errors in assembly only instead. In Fig. 2.2, the source code files are depicted as optional files.

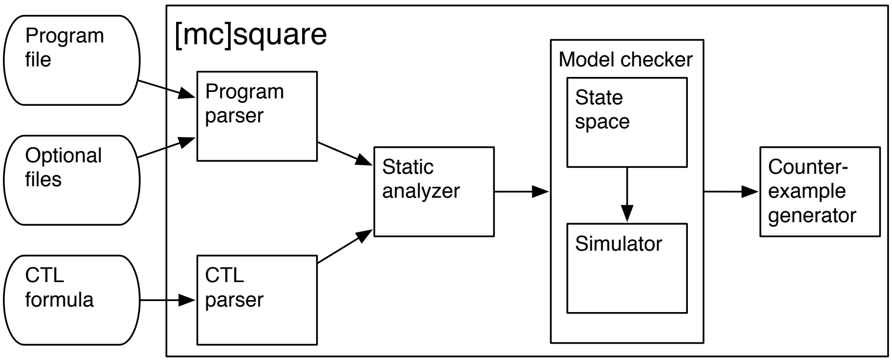

Figure 2.2: Core components of the model checking process in [mc]square. The figure was originally designed by Schlich and since then used in most publications on [mc]square, including our own.

Both program and temporal formula may be preprocessed by a static analyzer. The task of this component is to discover useful pieces of information about the program and to annotate it accordingly. *Useful* in this context relates to possible reductions of the state space size during the actual model checking, for instance by determining when values are no longer required, or which parts of the program are entirely irrelevant for the result and can safely be omitted. We give an overview of the effects of these techniques and of related work in Chapter 8, Abstraction.

Simulators are used to create the state space in [mc]square. The tool provides a simulator for each of the supported platforms. At a glance, the mode of operation of these simulators resembles that of regular simulators, such as those provided by hardware vendors: they contain a model of the hardware, execute programs by applying the semantics of instructions to the aforementioned model, and simulate the effects of interrupts and on-chip peripheral devices. However, there are also several differences due to the requirements in state space building:

- Simulators in [mc]square need to support nondeterminism. Nondeterminism arises from the necessity of creating an over-approximation of the behavior of the real hardware, and is introduced into the system primarily from two sources: first, the assumption that the environment of the microcontroller may exhibit any behavior, and second, from interrupts, that may occur but may also not occur.

- Abstraction from time. The model checking algorithm used in [mc]square is based on the assumption that there is no time except for a partial ordering of events. This means that simulators do not have to keep track of the amount of time elapsed since a certain event, such as the start of the simulation, and states accordingly do not have to store it. Hence, for states to be identical, it suffices that all memory locations are identical. This design decision in [MC]SQUARE eliminates the need for real time model checking algorithms, and also has the beneficial effect that states reachable in a loop may have to be stored only once. Therefore, the absence of time facilitates the verification process. A disadvantage, though, is that all timing-related on-chip peripherals also have to be considered to be able to show nondeterministic behavior. Therefore, timer and counter registers are also possible sources for nondeterminism.

- Third, the simulation should omit information irrelevant for the verification process. A very accurate simulation of internal processes of a device, such as the states of an analog to digital converter (A/D converter) may be helpful when developing circuitry containing the actual device, but in the realm of model checking, this additional information may increase the number of different states that have be stored. Therefore, simulators in [MC]SQUARE do usually not simulate devices as accurately as the simulators provided by hardware manufacturers.

- Finally, the state of the hardware is frequently stored to and restored from the state space, requiring simulators in [MC]SQUARE to perform these two steps very quickly. This requirement excludes a complex object oriented internal data structure for state representations, as traversing such structures is very time-consuming. General purpose simulators from hardware vendors usually never have to perform save and restore steps, which is why they are not optimized for it, and consequently, perform poorly in this regard.

If desired, it is possible to provide a more accurate model of the environment in order to reduce the over-approximation of the behavior of external devices. Such a model is called a *user defined environment* (UDE), which we introduced in an earlier publication [70]. The UDE provides details about possible values of input ports and also internal devices such as timers, which are usually modeled using nondeterminism. UDEs are attached to the simulator component and extend it, without being visible to other parts of [MC]SQUARE. Without a UDE, model checking is still possible, as providing one is strictly optional.

The state creation process in [MC]SQUARE works as follows:

- Load a state from the state space into the simulator.

- Determine whether any nondeterministic bits need to be assigned a deterministic value, that is, either 0 or 1.

- For each assignment, decide whether it indicates the occurrence of an interrupt. If this is the case, then simulate the effect of that interrupt. Otherwise, proceed with the current instruction instead.
- Check atomic propositions for the state reached.
- Store resulting states in the state space. Add successor edges from the original state to the newly created ones.

Nondeterministic bits in the simulation represent either a 0 or a 1. Under certain circumstances, it is necessary to determine the actual value of such a location before simulation may proceed. A very frequent example of this is the decision whether a flag bit used by an active interrupt is set or not, that is, whether simulation has to continue in the way expected for *an interrupt occurred* or in the regular way. We call the process of determining the deterministic value, in compliance with Schlich [68], *instantiation* or *determinization*.

Instantiation of n nondeterministic bits results in 2^n possible assignments, and consequently, states. Thus, the number of states increases exponentially in the number of instantiated bits, and consecutive instantiations typically result in state explosion. Therefore, several abstractions are implemented in [MC]SQUARE. Details on these are given in Chapter 8.

As pointed out before, [MC]SQUARE can create counterexamples and witnesses if necessary. These can be inspected on the GUI in different views: first of all, in either the assembly code or in the C code. In the presence of the source code and suitable debug information, these two panels are also linked, such that stepping through the states of the counterexample in one of the panels also updates the position in the other. Next, the counterexample can also be displayed graphically, as a sequence of states in the state space. Finally, it can be shown in the control flow graph on the static analyzer panel.

Apart from model checking, [MC]SQUARE also provides means for manual simulation. The developer may use the aforementioned C and assembly code panels to step through the program and inspect the reachable states. When interrupts are enabled, or nondeterministic choices on data occur, the simulator on the GUI allows to select a specific trace through the program. It is noteworthy that due to the state space being used for storing all seen states, this debugger also allows stepping backwards, which is a feature not normally found in software debuggers. In particular, this feature enables the developer to step backwards to points where choices were required (i.e., nondeterminism had to be resolved), and evaluate the effects of different choices.

3 Hardware Descriptions

The intention of this chapter is to point out the reasons for describing hardware, give an overview of available approaches, and establish the link to model checking of software for embedded systems. The latter step directly leads to a motivation of our research.

Several means for describing hardware were developed over the past decades, with different intentions. The most frequent reasons were either retargeting of existing tools, exploration of possible hardware designs, actually designing hardware, or simulating devices with the intention of developing software for these platforms. We provide an overview of these different goals in the next section. Section 3.2 contains a survey of existing approaches, which were designed to meet these goals.

In the third section, we point out why it is desirable to integrate a means for describing hardware into a model checker. Briefly, the issue encountered in the model checker [MC]SQUARE is its hardware-dependent approach. As [MC]SQUARE relies on simulators to build state spaces, it benefits from a very accurate model, and can provide exact and easy to understand counterxamples. On the downside, this approach renders it inherently hardware-dependent, and adapting it to new platforms is rather time-consuming. The requirements analysis in Section 3.3 therefore points out the features we deem necessary to remedy this hardware dependency.

3.1 Reasons for Describing Hardware

Within the scope of this section, we focus solely on the *reasons* for describing hardware designs by means of software. Examples for actual systems are given in the next section because many of these systems may be used for more than one area of application.

Hardware description languages can be classified in various ways. Some of them provide means for a low-level description of individual circuits and can be synthesized to actual hardware. The most prominent members of this class are VHDL and Verilog [42, 48], which can be used for designing devices and simulating their behavior on the level of signals. Other languages aim at the description of more specific classes of devices, such as processors. Unlike VHDL, these languages usually assume, for instance, that there are instructions to be processed by a device, with the consequence that there must be some means for loading, decoding, and executing them. Other typical assumptions may include the existence of special-purpose memory locations (registers), and among these, registers like a program counter and a stack pointer. For reasons pointed out below, we focus on processor-oriented languages.

Beyond this very coarse categorization, with VHDL / Verilog on one side, and special purpose languages for processors on the other, it is possible to create finer categories for the latter. A categorization which we consider helpful is presented by Halambi et. al. [32]. In their publication, they focus on systems on chip, and use the term *architecture description language (ADL)* for what we call processor-oriented. Their classification scheme is based on whether a language is designed primarily to describe the behavior of a device, or whether it rather focuses on describing its structure. Languages that allow, to some extent, to describe both, they call mixed level architecture description languages.

Another method of distinguishing languages is the level of abstraction they provide. VHDL provides a very low-level means of describing, based on state machines, clock cycles, and assignments to signals. Another level would be the *register transfer level*. On this level, actions performed by the hardware are described as assignments of values to individual registers. Machine instructions thus become a sequence of such assignments. Languages on this level are typically difficult to read and write. CSDL [59] is a language developed at the university of Virginia within the Zephyr compiler project. It consists of four languages, one of which is $\lambda - RTL$. This language is a register-transfer level language, but it already contains some improvements over plain RTL approaches for the convenience of a human developer. Contrasting to these low-level approaches, languages such as SystemC [42] resemble typical general-purpose programming languages.

3.1.1 Tool Retargeting

The idea of avoiding reimplementation effort by means of high-level languages can be traced back at least to the advent of Unix and the C programming language. According to Tanenbaum [77], from which this description of the history of Unix and C is taken, the motivation for designing C was to be able to reimplement Unix in a language that would ease porting the operating system to a new architecture. Instead of reimplementing the entire system, the necessary steps would be reduced to creating a C compiler that could generate code for the new architecture, and adapt some hardware-dependent parts written in assembler. Johnson eventually succeeded in creating such a portable C compiler. For details and references, we refer to the overview by Tanenbaum.

As C is still a relevant language in the operating systems and embedded systems domain, C compilers are still relevant as well. Therefore, it is necessary to create C compilers for new platforms. There are two approaches to this:

- retargetable C compilers

- generation of C compilers

An example for a retargetable C compiler is the GNU Compiler Collection, commonly referred to as GCC [24]. GCC has a backend that uses so-called machine description files, which are necessary for translating the intermediate code into binary code for the target machine. Supplying a new set of these files is therefore sufficient for adding support for a new platform to GCC. The other approach would be the generation of C compilers. Certain hardware description languages and associated toolkits facilitate the retargeting of C compilers, or even create entire tool chains consisting of compilers, assemblers, linkers, and profilers. Some of these are listed in the next section.

3.1.2 Synthesis and Simulation of Actual Hardware

As pointed out in the introduction of this chapter, languages like VHDL and Verilog provide means for describing hardware, and also synthesizing descriptions to actual hardware. For instance, it is possible to create a VHDL description that can be synthesized into an FPGA or CPLD. The advantage of such techniques is that changing the hardware is simple, compared to creating application-specific integrated circuits (ASICs), which are hard-wired for a specific application. Beneficial effects are on the one hand, that designing devices is easier, as no time-consuming manufacturing step is necessary, and on the other hand, it may be possible to modify a device that has already been shipped to customers.

With regard to simulation, however, there are also severe disadvantages when using VHDL or Verilog. As pointed out for instance by Pees et al. [55], these descriptions contain far too many details, which are not needed for evaluating new designs. Apart from the consequence that developers thus need to implement aspects of the hardware they are not interested in at that stage of design, this also reduces simulation speed. Finally, another important aspect pointed out by Pees et al. is that it rather difficult to extract the instruction set from such descriptions. VHDL descriptions can describe the components of the machine that fetch, decode and execute instructions, but there is no need nor means for describing the instructions themselves. With regard to the machine, they are just input, i.e., data, but not entities of the hardware description.

3.1.3 Design Space Exploration

Design space exploration refers to the development of a device for a specific scenario. A new device is tailored such that it meets its requirements. Often, this term is used together with the term *hardware-software co-design*. The latter refers to a division of tasks that need to be performed by the system into tasks that need to be executed by specialized hardware, and tasks that are better to be realized in software. For instance, designing an ASIC with specialized hardware devices may reduce power consumption, or enable it to perform time-critical tasks such as language processing without having to use a processor core that is too powerful and therefore, expensive. Other functionality that is prone to modification, or in general difficult to implement in hardware, should instead be handled by software.

Consequently, design space exploration involves the modification of the target processor, which in turn implies the modification of software tools designed for it. The latter typically consist of an assembler, a compiler (in most cases, C), and of tools for evaluating the performance of the overall system, e.g. profilers. While the modification of the processor design could be handled in general-purpose hardware description languages, the subsequent re-implementation of the software tool chain cannot be handled manually because the manual overhead would seriously restrict the modifications to the hardware. Hence, some languages and toolchains were specifically designed to allow for describing the hardware, and generating the necessary tool chain from the same description.

Tools like the CoWare Processor Designer [18], or the MESCAL project [37] were developed to support this design process.

3.1.4 Verification of Hardware

Faults in hardware may have serious and costly consequences, e.g. the Pentium FDIV bug.The latter has actually become a textbook example for motivating the use of formal methods, such as model checking and static analysis (e.g. [9]), on hardware designs [9, 16]. Unlike testing, these techniques typically do not operate on the physical device, which is why they need to be provided with some form of description of the hardware.

Apart from faults, there are other reasons for examining hardware descriptions. A very frequently found example is the analysis of worst case execution time (WCET) properties. Determining the maximum amount of time it takes for a device to execute certain instructions is important when deciding whether a system can be guaranteed to respond to an input within a given amount of time. Thus, WCET properties directly relate to the field of real-time computing.

3.2 Survey of Existing Approaches

LISA [55] is a language that started as a pipeline description language. In the classification scheme of Halambi et al., LISA is a mixed-level architecture description language, meaning that it focuses both on the structure of a device and on the behavior, i.e., the instruction set. The language LISA is part of a commercial product by CoWare, now part of Synopsys [18]. Given a LISA description, the tool chain allows the developer to automatically generate tools such as compilers, assemblers, profilers, and simulators. Simulators can be both cycle-accurate and instruction-accurate.

EXPRESSION is an architecture description language developed at the University of California, Irvine [25, 31, 32]. It was designed for design space exploration of embedded systems on chip (SoC) and concentrates on both the structure and the behavior of processor-based designs. EXPRESSION can be used for generating retargetable compilers as well as functional (i.e., instruction-accurate) and cycle-accurate simulators.

ISDL is a language created at MIT, the main scope of which is the description of Very Long Instruction Word (VLIW) architectures [30]. The non-abbreviated name is Instruction Set Description Language. It was explicitly designed to support design space exploration, and can be used to generate a variety of tools. Tools named in publications on ISDL include an assembler, a compiler, and a so-called instruction-level simulator. The latter is another synonym for instruction-accurate simulation, also called functional simulation.

Another language abbreviated ISDL was developed at the University of California, LA, by Titzer and Palsberg. In a technical report [82] on the language and system, the language was also called Isildur. Despite the same naming, there is no connection between the two ISDL languages. Isildur was used in the AVRora project, which aims at simulating sensor nodes, where it was used to describe the instruction set of the Atmel AVR family of microcontrollers. Apparently, the language emphasizes the instruction set, but provides almost no means for describing the structural aspects of a device.

MADL stands for Mescal Architecture Description Language, and was developed at Princeton [37, 58]. It is based on a two layer approach, which are called the core layer and the annotation layer. The core layer contains a description of machine behavior in the form of finite automata, which are used to represent the state of the available instructions. Instructions can be bound to different pipeline states of the target machine, and there are no assumptions about the structure of the pipeline. Hence, MADL appears to be suitable for describing any kind of pipelined architecture. The second layer, called annotation layer, relates to different software tools that can be generated. Hence, this separation of concerns makes the approach very flexible, as it avoids mixing tool-specific and device-specific information. Tools generated from MADL descriptions include cycle-accurate and instruction-accurate simulators, a register allocator, and a reservation table scheduler. In this respect, MADL appears to be about as flexible as EXPRESSION and LISA.

3.3 Requirements Analysis

[MC]SQUARE is an assembly code model checker, which builds state spaces using simulators, which are used to simulate microcontrollers. Based on experience from adding support for new platforms, it is possible to estimate the effort for manually implementing such a simulator. The Atmel ATmega128 and XC167 simulators were implemented by diploma thesis students [47, 67], and the 8051 simulator was implemented by Reinbacher partly before and partly within a master's thesis [61–63]. At the end of these theses, the simulators were operative, but not all of the peripherals had been implemented. Depending on whether there was interest in further uses of the simulator, it was then necessary to continue development, ending up in a total effort of about one year for a single full-time developer.

Thus, implementing these simulators manually is too time-consuming, especially for using [MC]SQUARE in industrial settings. Our motivation was therefore to reduce this effort. We intended to use a hardware description language, and to *generate* simulators instead of manually implementing them. As pointed out in Section 2.6, simulators in [MC]SQUARE differ from simulators used in other scenarios. Based on the list of characteristics, we deduced a set of functional requirements for any generated simulators:

- It must be possible to generate instruction-accurate simulators.
- The language must provide means for describing abstractions, such that generated simulators support these out of the box.
- It must be possible to integrate simulators into [MC]SQUARE.
- The range of devices that can be described must cover at least all the existing platforms in [MC]SQUARE.
- Both language and simulators must natively handle nondeterminism.
- It must be possible to also generate a static analyzer from the description.
- There must be no licensing issues restricting the manipulation of the language, the processing chain, or the distribution of simulators.

Furthermore, we deem the following non-functional requirements necessary:

- Generated simulators must be fast, i.e. support load and store operations for their entire internal state.
- The effort for creating a simulator description must be considerably lower than six months.

- The developer must be able to decide which and how many details he wants to integrate into a description.

Starting from these requirements, we then conducted a survey of the available languages and generation systems. As we intended to eventually create both simulator and static analyzer for a platform, we concluded that a layered architecture like the one used in MADL would suit our purposes. However, the language emphasizes the pipeline of a processor, whereas microcontrollers seldom feature any pipelining, being focussed on low cost rather than high performance. A similar argument applies to LISA, which is also proprietary, and therefore not very easy to modify in case this should have become necessary. Furthermore, work on MADL appeared to have ceased, and the fragments available to the public were not operative.

Considering that we would likely have to modify the language during our research, we decided for the language Isildur used in the AVRora project. This language had been developed up to a point where describing microcontroller instruction sets had become feasible, and its developer, Ben Titzer, granted us the right to modify it at will. Furthermore, there were previous experiences with AVRora in [MC]SQUARE, as the original simulator used in [MC]SQUARE was the AVRora simulator. There was no complete processing chain, except for a parser and a few fragments of the abstract syntax tree. Thus, this language appeared to be suitable.

4 A System for Synthesizing Simulators

In this chapter, we summarize the most important aspects of the synthesis system. First, we briefly point out its features and its architecture. Next, we focus on the concepts used for state space building, or simulation, and those relevant for static analysis. The chapter concludes with an overview of the concept of abstraction. These illustrations are relevant for the subsequent chapters, in which we detail how we realized them.

4.1 Features

The main features of our simulator synthesis system are:

- Generation of simulators
  - New language for the description of microcontrollers: SGDL
  - Description of simulators in SGDL considerably shorter than full implementation, ratio approximately 1 line of code (LOC) in SGDL for 10 lines of generated code
  - Considerably faster development: 1 week for basic MCS-51 implementation
- Input assist
  - Static analyzer for SGDL integrated into the synthesis system, which can automatically deduce information about a platform; reduces need for explicit descriptions
  - Code templates for the SGDL preprocessor to accelerate development
  - Syntax highlighting for SGDL in the open source editor Programmer's Notepad [75]
- Supported platforms
  - Atmel ATmega16
  - Atmel ATmega644
  - Intel MCS-51

- Library of supported instructions for all Atmel AVR microcontrollers, automatically extracted from data sheets

- Support for binary file formats
  - Executable and Linkable Format (ELF)
  - Intel HEX file format

- Automatic and assisted semi-automatic generation of abstraction techniques
  - Lazy Stack Evaluation
  - On-the-Fly Path Reduction
  - Dead Variable Reduction
  - Delayed Nondeterminism
  - Generated simulators support static analysis of programs to allow further abstractions

4.2 Architecture

In this section, we describe the architecture of the synthesis system as a whole. On this level, elements of the architecture include a language for describing the input, a processing tool chain, and eventually, generated components. These individual elements of the architecture refer to complex systems, each of which has an architecture of its own, which we detail in subsequent chapters.

Fig. 4.1 illustrates the top-level architecture. Generating state space generators for microcontrollers, that is, simulators combined with an optional static analyzer, requires first of all a means for describing the relevant aspects of microcontroller devices. As our requirements were not met by any of the ADLs presented so far, we have decided to develop a new ADL for this purpose, which is called SGDL. An SGDL description is processed by the SGDL compiler contained in the synthesis framework. The compiler generates an executable simulator and a static analyzer for the target microcontroller. Additionally, it creates glue code interfacing these two components to the other parts of [MC]SQUARE, especially the core static analyzer, the model checker, and the graphical user interface.

In the following, we provide a brief overview of these major components of the system. Details are given in the respective chapters.

4.2.1 Description of Input

For a given device, we need to capture at least the instruction set, resources (i.e., memories), the relationship between memories, and the interrupt system. Moreover,

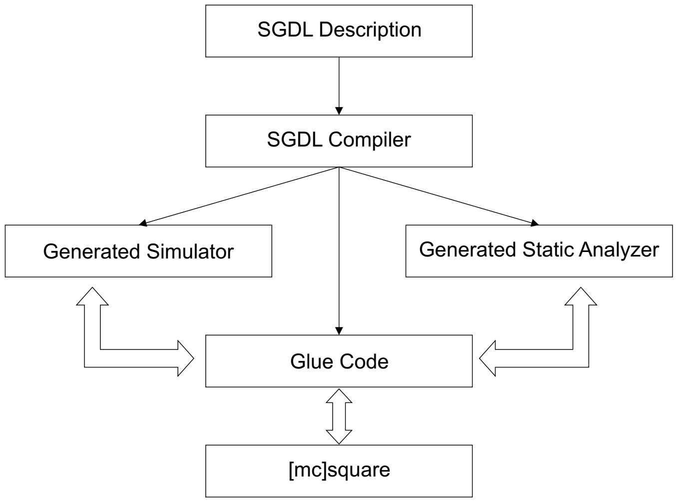

Figure 4.1: Architecture of the synthesis system. Thin arrows illustrate the processing direction at compile time, whereas thick arrows show data flow in the generated system, that is, at runtime.

we need a description of possibly nondeterministic behavior, that is, situations in which we cannot predict the exact behavior, but need to approximate. This involves the following aspects:

- All cases in which the simulated microcontroller reads values from its environment. Behavior of external devices cannot be predicted, and therefore, for the sake of preserving an over-approximation, we need to consider it as possible that *any* value is sent from the environment to the microcontroller. Under certain circumstances, it is possible to reduce the over-approximation, as was shown by us in a previous publication [70].

- Occurrence of interrupts. In our discrete timing model, whenever the simulated device can execute an instruction, we also need to consider the possibility that an event associated with an active interrupt occurs. Such an event may result in a branch of the computation to the interrupt handler. Since such an event *may*, but not necessarily *must* occur, there are two possible continuations of the current computation path, i.e., there is a nondeterministic choice. The description of the hardware has to provide ample means for describing the event sources of interrupts, the condition leading to interrupts, and the actions to perform in case the system decides that an interrupt has occurred.

- Timing. [MC]SQUARE conducts discrete model checking. Hence, transitions in the Kripke structure are either related to the execution of instructions, the occurrence of interrupts, or some step performed by a concurrent on-chip peripheral changing its state. Opposed to this, in continuous-time model checkers such as UPPAAL [39], the passing of time already leads to new states. While the approach in [MC]SQUARE is therefore less likely to result in state explosion, it has the obvious disadvantage that all timing-related information is lost. Thus, *timers* on the microcontroller cannot be modeled accurately, and the value of their timer/counter registers must be considered nondeterministic whenever the timer/counter is active.

- Parts of the system intentionally modeled using nondeterminism. An accurate description of machine states can have disadvantageous effects. For instance, an analog to digital converter (ADC) has several internal states which it assumes during a conversion. Modeling all of these results in several distinguished states, each of which needs to be stored in the state space. However, from the stance of instruction-accurate simulation, the internal states of the ADC may be irrelevant. Hence, a less accurate modeling, in which starting the ADC sets its result register and *conversion finished* flag to nondeterministic, can cause many of the intermediate states to vanish, and therefore counter the state explosion problem. Consequently, our description mechanism has to provide means for marking *any* memory location as nondeterministic to allow for such simplified modeling.

In Sections 4.3 and 4.4, we detail several further requirements relevant for simulation and static analysis. Chapter 5 describes the language we developed to meet these requirements.

4.2.2 Processing the Input

The input is processed by the SGDL compiler, which is part of the synthesis framework. Besides processing the hardware description and generating the state space generator, the compiler also contains a subcompiler for a specialized subset of SGDL. The subset contains language elements for describing certain properties of the generated code, such as its location in the [MC]SQUARE package hierarchy, and how to derive the names of the main classes. Using a description in this sublanguage, the SGDL compiler can automatically integrate a newly created simulator into [MC]SQUARE. Details on the SGDL compiler are provided in Chapt. 6.

4.2.3 Generated State Space Generators

Generated simulators consist of generated code and of code in libraries. The generated code is specific to the description in the input, whereas the library code represents parts of simulators that are comparatively independent of an actual hardware, and can therefore be reused. Examples for the latter include storage classes for memories, generalized data type classes, and classes for creating bit patterns, which are used in resolving nondeterminism.

4.3 State Space Building

Our generated simulators create the state space in the same way as the existing simulators that were implemented manually. That is, they are initialized to a state that should be equivalent to the reset state of the physical device. Machine states can be changed by instructions, interrupts, and internal devices. SGDL provides means for describing all of these. A detailed description of the mode of operation of the generated simulators is given in Chapter 7.

4.4 Static Analyzers

There are two static analyzers involved with the synthesis system:

- In order to support the model checking process, [MC]SQUARE provides a framework for static analysis of microcontroller code. This framework can be used to perform a preprocessing and annotation of a microcontroller program, before the model checking is started. Information obtained during the static analysis can then be used to reduce the size of the state space.

- SGDL itself can be analyzed by a static analyzer that is integrated into the SGDL compiler. By means of this analyzer, it is often possible to omit some explicit descriptions of hardware behavior, as it can be derived from the information that is already present.

Analyzers for microcontroller code, which are based on the framework in [MC]SQUARE, are too different from analyzers for high-level languages such as SGDL. Therefore, the two analyzers do not share any code.

The framework in [MC]SQUARE facilitates the creation of a static analyzer for a new platform. Still, there is some effort involved in creating such an operative analyzer. Therefore, the SGDL compiler can generate the necessary code from an SGDL description, provided the necessary information is available in the description. In case it is not available in an explicit form (i.e., implemented by the simulator developer), the analyzer in the SGDL compiler, SGDL-STA, will try to obtain it. If that should not yield the required result, the generated analyzer may be inoperative, either as a whole or in parts.

The generation of static analyzers is covered in Sect. 8.5. SGDL-STA is illustrated in Sect. 6.5.

4.5 Abstraction

As model checking in general suffers from the state explosion problem, we need to integrate abstractions into our generated simulators. Abstractions help reduce the number of states that have to be stored during model checking, and in some cases they even reduce the number of states that have to be created. In the ideal case, such abstractions should be generated automatically from the description of the platform, that is, without requiring additional information provided by the simulator developer. The intention behind this is that ideally, a developer capable of understanding microcontroller data sheets, should also be able to implement simulators for model checking. The alternative would be to require him to be an expert both in microcontrollers and in formal verification. Besides the resulting higher cost of simulator implementation, this would also result in reimplementation of existing abstractions, which should be avoided wherever applicable.

The SGDL compiler can actually create some abstractions completely automatically, e.g. Path Reduction. Others, like Delayed Nondeterminism and Dead Variable Reduction, require some additional information that has to be provided by the developer, but the synthesis system at least assists in generating the actual abstraction from the high-level information. Finally, some abstractions, like the ADC example in Sect. 4.2, where irrelevant intermediate states of an internal device are abstracted away, depend on the level of precision that is desired for verification. For such abstractions, it is still necessary for the developer to carefully design the simulator to suit his needs, as the SGDL system cannot possibly guess which precision is adequate.

A thorough introduction into the available abstractions and the concept of abstraction in general is provided in Chapter 8.

5 State Space Generator Description Language

The language that serves as input for our synthesis system is called *State Space Generator Description Language* (SGDL). SGDL was originally based on the *Isildur* language (ISDL) from the AVRora project [81, 82], which served as a starting point for our research. AVRora is a project aiming at a cycle-accurate simulation of sensor networks, initiated by Titzer. The original intention for creating ISDL was to reduce the effort for adding support for new types of sensor nodes. However, the project restricted itself to supporting certain specific types of nodes, one of which is based on the Atmel AVR family of microcontrollers. Thus, there was no need to foster ISDL, which was therefore dropped in a relatively early stage of development. Up to that point, however, the project had already produced a complete ISDL description of the AVR instruction set. The description was not complete in that resources were not defined, interrupts were missing, and there was no means of restricting the instruction set to a subset supported by a specific device.

State space building in [MC]SQUARE was originally based on the AVRora ATmega16 simulator, which was later replaced by a reimplementation that better suits the needs of a model checker. For this reason, we decided to design our new hardware description language starting from ISDL. Bogosavljevic [11] extended ISDL in his master's thesis and created a first version of the synthesis system that is now the SGDL compiler. The resulting language was then called DICES, and subsequently renamed to SGDL.

Even though there are still similarities between the two languages, ISDL and SGDL are not compatible. Many features have been invented specifically for SGDL, such as the mechanisms for describing resources (i.e., memories, interrupts and peripherals), guards for enabling instructions, or all language elements related to abstraction. Features inherited from ISDL relate mostly to the description of instructions and subroutines, though most of these have been extended or modified as well. This chapter describes syntax and semantics of SGDL in detail.

5.1 Overview

An SGDL description of a microcontroller consists of these major components, which are detailed in the following sections:

- header
- memory description
- instruction set description
- interrupt system description
- declaration of atomics
- specification of loaders

The *header* contains the name of the architecture, and is the only element outside any block. All other elements are contained within the brackets of the main block. The *memory description* declares all memories present in the described microcontroller. From this, we generate the resource model used in simulation.

The *instruction set description* describes all instructions supported by the microcontroller, and may also include additional helper methods. Instructions operate on the resource model and modify it.

All interrupts are described in the *interrupt description*. Interrupts can also modify the resource model.

Atomics assign names to memory locations and can be used in CTL formulas. Furthermore, they are also used for display in the [MC]SQUARE GUI. They appear in the memory monitor on the simulation panel, where they are shown in tooltips when the mouse hovers over a named location.

Finally, a *loader* specification indicates that the microcontroller simulator should be able to read programs from a specific file format. The information in the loader section describes how to generate the necessary binary file loader, which can extract the instruction stream from a container format such as ELF [80], and write it to the correct location in the program memory.

5.2 Memory Description

Memory descriptions in SGDL consist of two mandatory components: *memory declaration* and *memory access*. A memory declaration is in principle equivalent to an array declaration in any imperative programming language. However, the declared array can never be accessed directly. For accessing the memory, for instance from within an instruction, a *memory alias* is required. The alias contains all necessary information for accessing the memory cell, e.g. whether the memory is accessed bytewise or wordwise, the size of a word, the endianness, and how to interpret the content of the cell. This abstraction layer allows for different ways of accessing memory cells, which is required in some architectures. For instance, on the Atmel ATmega16, the register bank is mapped into the SRAM address space [5, 6].

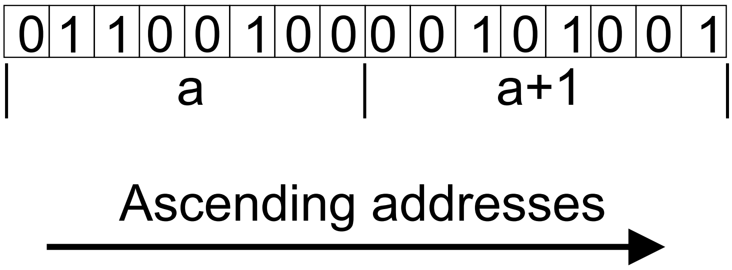

Figure 5.1: Interpretation of memory contents. The contents of the 8 bit cells a and $a+1$ can be read as either two separate 8 bit values or a single 16 bit value. Furthermore, the ordering of the bytes may be swapped, with either a or $a+1$ as the lowest byte in a 16 bit value. Other interpretations are also possible.

Hence, accessing the register bank (seemingly) modifies the SRAM, and vice versa. Using aliases, these different addressing modes can be resolved without the need for conditional access code (i.e., if-blocks). Memory aliases are semantically equivalent to typed pointers in languages such as C and Pascal.

Example 5.1. ***Interpretation of memory contents***
Let a be an address in a memory space, $a \in \mathbb{N}$. Let $val(a)$ denote the value of the memory cell at address a. All cells have a width of eight bits.

Valuation. Let $val(a) := 100 = (01100100)_2, val(a+1) := 41 = (00101001)_2$.

The memory layout and representation are depicted in Fig. 5.1. Read in ascending order, the memory cells contain the concatenated bit patterns of the two values, i.e., 0110010000101001. *Possible interpretations of this pattern include*

- *Word size 8 bits: values 100, 41*
- *Word size 16 bits, big endian mode: the value is interpreted as*

$$(val(a)\, val(a+1))_2 = (25641)_{10}$$

- *Word size 16 bits, little endian mode: the value is interpreted as*

$$(val(a+1)\, val(a))_2 = (10596)_{10}$$

5.2.1 Memory Declaration

Syntax:

```
memory array <name> = {<attribute list>};
```

- <name>: an identifier
- <attribute list>: a comma-separated list of attribute = value and attribute = "some string" pairs. The ordering of the elements in the list is irrelevant.

The following values are allowed or required in the attribute list:

- size: Integer attribute. Mandatory. Denotes the size of the array, counted in bytes.
- always_deterministic: String attribute. Optional. Defines which memory cells must always remain deterministic. The value of this attribute is a comma-separated list of integers, which can be given either as decimals or hexadecimals.
- memory_type: String attribute. Optional. Selects a different array implementation. The default memory array of size n in SGDL corresponds to a byte array in the target language of size n. Alternatively, this attribute can be set to differentialArray, which is a kind of array that stores only those values that are unequal to zero. This feature is only useful if most of the memory is known beforehand to be zero, or if the content of the memory changes only marginally during simulation (which often holds true for the program memory, for instance). In such cases, the reduced memory footprint of individual microcontroller states requires less time for transferring data to and from the state space, thus accelerating state space building.
- never_reset: String attribute. Optional. Defines which memory cells must never be reset by an abstraction technique such as Dead Variable Reduction (cf. Sect. 8.5).
- bitwise: String attribute. Optional. Defines which memory cells are to be modeled bitwise in static analysis. The value of this attribute is a comma-separated list of integers, which can be given either as decimals or hexadecimals.

5.2.2 Memory Alias Declaration

Syntax:

```
memory alias <name> : <type> = {<attribute list>};
```

- `<name>`: an identifier
- `<type>`: one of the following values
  - `memory`: used for multi-byte ranges of memory
  - `bit`: used for single bits
  - `reg`: similar to `memory`, but limited to a maximum length of 32 bits
  - `<attribute list>`: a comma-separated list of `attribute = value` and `attribute = "some string"` pairs. The ordering of the elements in the list is irrelevant.

The allowed or required values in the attribute list depend on the type of alias:

- `memory` requires / allows
  - `block_size`: Integer attribute. Defines the size in bytes of the element at each address of the alias.
  - `block_count`: Integer attribute. Defines the number of elements.
  - `endianness`: String attribute. Possible values: "little", "big". This attribute is relevant for block sizes larger than 1, where it defines whether the least significant bit (LSB) or most significant bit (MSB) comes first.
  - `underlying`: String attribute. Defines the name of the memory array this alias refers to.
  - `u_from`: Integer attribute. Defines the index in the memory array where to start the mapping (i.e., where address 0 of the alias is located).
  - `u_to`: Integer attribute. Defines the last index in the memory array covered by this alias.
  - `op`: String attribute. Defines how to interpret the content of the memory cells referenced by the alias. For instance, it can be interpreted as a signed byte or an unsigned byte value.
  - `uniquetype`: String attribute. Optional. If the alias is supposed to fulfill a special function, which no other alias may have, then this can be indicated by setting this attribute. Possible values: "program" (the only program memory), "data" (the main address space).
  - `description`: String attribute. Optional. If set, this alias is marked for display on the graphical user interface. The compiler will generate code to create a new tab in the memory monitor on the GUI, and the tab will be labeled with the description text.
  - `writable`: String attribute. Optional. Possible values: "true", "false". Defaults to "true".

  - `addressOutOfBoundsBehavior`: String attribute. Optional. Defines what should happen if the alias is accessed with an invalid address. Possible values: "error" (default, causes a simulator error), "zero" (read accesses return 0, write accesses are ignored), "wrap" (wrap around: access address modulo `block_count`)

- `bit` requires / allows
  - `underlying`: String attribute. Same semantics as for `memory` aliases.
  - `u_byte`: Integer attribute. Defines an index in the memory array, counting in bytes from the start of the array. The bit addressed by the alias has to be one of the eight bits in this byte.
  - `u_bit`: Integer value $\in \{0, \dots 7\}$. Defines the index of the bit inside the byte referenced by `u_byte`.

- `reg` requires / allows
  - `underlying`: String attribute. Same semantics as for `memory` aliases.
  - `u_byte`: Integer attribute. Similar to the attribute `u_byte` in the `bit` alias type. This value describes the first byte index of where the register is located. Unlike `bit`, however, the `reg` alias can also access adjacent bytes with a higher index.
  - `bit_width`: Integer attribute. Defines the number of bits that form this register, starting with the first bit of the referenced byte, i.e., $u\_byte[0]$.
  - `op`: String attribute. Same semantics as for `memory` aliases.
  - `endianness`: String attribute. Same semantics as for `memory` aliases.
  - `uniquetype`: String attribute. If the alias is supposed to fulfill a special function, which no other alias may have, this can be indicated by setting this optional attribute. Possible values: "pc" (indicating this alias is the unique program counter).

5.2.3 Memory Initialization

Syntax:

```
init memory = {
  <statements>
};
```

- `<statements>`: Arbitrary code. See the section on code blocks (Sect. 5.9) for details.

5.2.4 Mandatory Memories

Each platform has certain special function memories: a program memory, a program counter, one or multiple stack pointers, and a main address space. Each of these has to be defined in the SGDL description. Those memories that are unique for a platform are declared by setting the optional `uniquetype` attribute in an alias declaration. If set for an alias, then no other alias may have its `uniquetype` attribute set to the same value. For example, to declare a program counter (PC), we first of all have to declare a memory array containing the PC. Next, we define an alias to access the array, and set its `uniquetype` attribute to the value `pc`.

The following values for `uniquetype` attributes exist:

- `pc`: program counter
- `program`: the program memory where the program is stored
- `data`: the main address space

Stacks are another type of mandatory memory. An architecture may have more than one stack, hence `uniquetype` is not applicable for the declaration of stacks. Instead, there is a dedicated language element for declaring stacks:

```
stack <name> = {<attribute list>};
```

where

- `<name>`: an identifier. Mandatory.
- `<attribute list>`: a list of entries of the form *key=value*
  - `stackpointer`: String attribute. Mandatory. The name of the alias to be used as a stack pointer.
  - `memory`: String attribute. Mandatory. The name of the alias where the stack is actually stored.
  - `bottom`: Integer attribute. Mandatory. The address to which the first element in the stack will be pushed.
  - `direction`: Mandatory. The direction in which the stack grows. Possible values: `left`, `right`. The values correspond to a graphical representation of intervals, where left means smaller addresses and right larger ones. Hence, for stacks growing from high to low addresses, `left` is the appropriate setting.
  - `stackpointerposition`: String attribute. Mandatory. Possible values: `ahead`, `exact`. This attribute describes whether the stack pointer points to the next free position on the stack or to the last occupied one.

  - `maxtop`: Integer attribute. Mandatory. Indicates the largest (or smallest, depending on growth direction) address that the stack may occupy. Can be used to identify faulty program behavior and abort simulation in case the stack grows into reserved memory areas.

5.2.5 Memory Cell Dependencies

Memory cells can depend on other memory cells. For instance, the content of an I/O register, which represents the value of an I/O port, usually depends on the port configuration. That configuration is stored in one or multiple memory locations, which are often registers themselves. Reading from or writing to the I/O register requires at least a read access to the configuration registers, i.e., there is a dependency between registers.

Besides dependencies between registers, there are also dependencies between registers and interrupts. An interrupt may be enabled depending on the value of one or several configuration bits, which may be distributed among multiple registers.

Certain abstractions, such as Dead Variable Reduction, might modify memory locations. These abstractions have to be aware of such dependencies, lest they result in spurious behavior. The syntax for describing dependencies is depicted below.

Syntax:

```
dependencies {
  (<dependency declaration>)+
}
```

`dependency declaration` can be one of

- `on <access type> <register> <access type> <register>`
  `using mask <mask>;`
- `<register> influences interrupt <interrupt> using mask <mask>;`

where

- `access type`: can be either `read` or `write`.
- `register`: Indicates the name of a register alias.
- `mask`: an Integer value indicating the relevant bits in the register.
- `interrupt`: Indicates the name of an interrupt.

5.3 Instruction Set Description

The instruction set description consists of the following language elements:

- encoding format declarations (optional)
- operand type declarations (optional)
- the actual instruction elements (mandatory)
- a value for the global attribute `instruction word size` (optional)
- subroutine declarations

5.3.1 Encoding Format Declarations

Encoding formats can be used to facilitate the declaration of instruction elements. They describe the structure of binary encodings of instructions. Declaring and using an encoding format is useful whenever several instructions share a similar encoding, such as different instructions using the same addressing mode. For instructions with unique patterns, it is also possible to declare the encoding within the scope of the instruction element, without declaring a name for it.

Syntax:

```
format <identifier> = {<attribute list>};
```

- `<identifier>`: a name by which the format can be referenced
- `<attribute list>`: an ordered list of entity declarations. An entity is declared implicitly by an identifier and an indicator. The latter defines the bits of which the entity is composed. The list of entities is read from left to right.

Example:

```
format ADDRESS_11_FIRST = {target[10:8], opcode[4:0],
   target[7:0]
```

In this example, the first three bits of the binary input stream are the bits 10 to 8 of an entity called `target`. The next five bits of the input stream are all five bits of an entity called `opcode`. After that, the next eight bits of the input stream are bits 7 to 0 of `target`. Inside the instruction element, we can reference the entities by their names, and we do not have to care about their representation. In this example, this especially means that `target` can be referenced as a whole by its name, even though its binary representation in the stream is split (a so-called *split field*).

Note that even though the structure of the encoding is defined by a format, the format does not contain any statement regarding the value of bits. Thus, any bit of any entity in the encoding could be 0 or 1. This distinction is only made when defining instructions *using* such a pattern. Thus, in the above example, there are 2^5 possible values for opcode.

5.3.2 Operand Type Declarations

Operand types are similar to type declarations in C/C++. Any sequence of bits can be assigned a domain by an operand type declaration.

Syntax:

```
operand <identifier>[<length>] :
  <operand class> = <domain declaration>;
```

- <identifier>: a name for the operand type
- <length>: the number of bits needed for representing an operand of this type
- <operand class>: the type of the operand. Possible values:
  - unsigned: an unsigned integer domain
  - signed: a signed integer domain
  - symbol: a custom name for each of the bit patterns. Bit patterns are represented by their decimal values.
- <domain declaration>: depends on the value of operand class:
  - for signed and unsigned, possible values are: [<x>, <y>], where <x>, <y> are signed integers.
  - for symbol, possible values are: $\{< \mathit{identifier}_0 >=< \mathit{value}_0 >, \ldots, < \mathit{identifier}_n >=< \mathit{value}_n >\}$, where the identifiers can be selected as desired and $n = 2^{<length>} - 1$

Example:

```
operand SIGNED_IMM8[8] : signed = {-128, 127];
operand ADDRESS_REGISTER[1] : symbol = {
   R0 = 0, R1 = 1
};
```

The declaration of operand types already existed in ISDL and remained almost unchanged in SGDL. The only exception is that we also allowed the use of named constants, e.g. for the bit width.

5.3.3 Instruction Element

The instruction element of SGDL is used for describing the instruction set of a device. Each instruction in the device's instruction set is mapped to one occurrence of the instruction element, unless peculiarities of the device require more than one such occurrence.

An instruction declaration consists of several subsections. The ordering of these subsections is irrelevant. The following sections are allowed:

- `encoding` (mandatory) (from AVRora)
- `operandtypes` (optional) (from AVRora)
- `no auto increment of pc` (optional)
- `syntax` (optional) (from AVRora)
- `enabled` (optional)
- `cycles` (optional) (from AVRora)
- `instantiate` (optional)
- `dnd instantiate` (optional)
- `static behavior` (optional)
- `execute` (mandatory) (from AVRora)
- `execute "DND"` (optional)

The `instruction` element and each of its subsections is described in detail in the following sections.

Syntax of `instruction`

```
instruction <identifier> {
  encoding = <indirect encoding> | <direct encoding>;
  operandtypes = {<list>};
  no auto increment of pc;
  syntax = <format string>;
  enabled = <boolean constant>;
  cycles = <number>;
  instantiate = <list>;
  dnd instantiate = <list>;
```

```
  static behavior = {
    type = <type>;
    reads = {<reads list>};
    writes = {<writes list>};
    target = <target expression>;
  }
  execute = {<statements>};
  execute "DND" = {<statements>};
};
```

Identifier

The `identifier` is a mandatory attribute of an instruction element. It assigns a unique name to the element, which is used in code generation. The name is also used for display on the GUI, where an instruction is represented by a string concatenation of name and instruction operands. It is possible to change the latter by assigning a value to the `syntax` attribute of the instruction element.

Encoding

The `encoding` is a mandatory attribute. It describes a binary bit pattern by which the instruction can be identified within a bit stream. There are two possible ways of describing the encoding:

- direct encoding: the encoding is described entirely in situ.
- indirect encoding: the encoding follows some pattern, i.e., a *format* from the `format` section (cf. Sect. 5.3.1).

Direct encodings are useful when describing single instructions that are not part of any identifiable group. However, most instructions on many devices can be grouped, for instance by the addressing mode they use. A typical structure would be "n bits of opcode, followed by k bits identifying a register x, followed by l bits identifying another register y". The practical implication of the existence of such patterns is that SGDL developers should use formats whenever applicable to avoid redundancies in the description.

- Syntax of direct encoding:

  ```
  encoding = {<binary pattern>};
  ```

  Example:

  ```
  encoding = { 0b01000100 };
  ```

- Syntax of indirect encoding:

  ```
  encoding = <format name> where {<attribute list>};
  ```

  In the attribute list, the identifiers in the declaration of the encoding format have to be mapped to 0 or 1, or remain undetermined. Determined bits are used for matching the instruction in the binary input stream. Bits which are not determined are free, and are assigned a value from the actual bit pattern in the stream when the instruction is matched. Example:

  ```
  encoding = ACC_DIRECT where { opcode = 0b01010101 };
  ```

Operand Types

Operand types describe how to interpret the bit pattern of variables, that is, the entities in encodings related to free bits. Those can be accessed by their name in any code block within the scope of the instruction. Given an encoding and no operand type, the only information about a parameter is its length in bits. Hence, additional information is required in order to decide whether the parameter is, for instance, a `signed` or `unsigned int`, or a symbol.

Another reason for using an operand type is to clarify the semantics of parameters. For instance, an eight bit value can be considered an address or an immediate. In both cases, it has to be interpreted as an unsigned integer value (`unsigned byte`, `ubyte`). Therefore, any computation involving the value will yield the same result. However, to clarify the semantics, the developer can create two types ADDRESS8 and IMM8.

Example:

```
operandtypes = {imm : IMM8};
operandtypes = {reg : GPR};
operandtypes = {address : ADDRESS8, imm : IMM8};
```

Pre-defined operand types are

- `ubyte`
- `uint`
- `boolean`
- `void` (not available for instruction parameters)

Automatic Incrementation of PC

Syntax:

```
no auto increment of pc;
```

By default, in the generated simulator, the program counter (PC) is automatically incremented by the size of the current instruction *before* the code in the `execute` section is executed. While this facilitates development for most instructions, it can complicate the implementation of instructions that directly modify PC. Especially relative and conditional jumps can be easier to implement (that is, closer to the data sheet) without the automatic incrementation. Adding this directive to an instruction deactivates automatic incrementation for that instruction only.

Syntax

For pretty-printing instructions on the GUI, developers may add a formatting string. This is one of the features inherited from ISDL that is barely used in SGDL, as it has no effect on actual instruction semantics.

Enabling of Instructions

Allows to enable or disable instructions when compiling a simulator. This feature allows the developer to create a description of instructions for an entire family of devices, and to only activate those instructions supported by a specific device. Instructions that are not enabled are skipped at compile time by the SGDL compiler. Hence, they are not present in the generated simulator, and also not in the generated static analyzer.

Symbolic constants are allowed for this property. Hence, it is possible to create a list of constants for numerous devices, and to use the include capability of SGDL to import the correct one for a given device. We call this step *parameterization of the instruction set*. Currently, we provide a parameterized instruction set and a library of enabling constants for most members of Atmel's AVR family of 8 bit microcontrollers.

The `enabled` attribute is optional. If absent, it defaults to true.

Cycles

The integer value for the attribute `cycles` indicates the number of cycles required for executing the instruction. This attribute is inherited from ISDL and its value is currently ignored by the SGDL compiler. We have decided not to remove it, though, because certain operations on some microcontrollers may fail if not completed within a certain number of cycles. An example are flash memory write operations on the Atmel AVR microcontrollers. Adding the ability to check for such problems would require such an attribute.

Instantiate and Dnd Instantiate

Some instructions may have to operate on nondeterministic data. For deterministic simulation, such data has to be instantiated first, that is, the value of nondeterministic bits is determined to be either 0 or 1. After instantiation, the code in the `execute` section of the instruction can be executed just as for any other instruction. Hence, the code in the `execute` section will only operate on two-valued boolean logic, instead of ternary logics (i.e., values in $\{0, 1, n\}^*$).

The `instantiate` directive tells the SGDL compiler to generate the necessary code for instantiating any location named within the directive. `instantiate` is used for deterministic simulation, whereas `dnd instantiate` is used when Delayed Nondeterminism (cf. Sect. 8.6) is active.

Example:

```
instruction "in" {
   ...
   instantiate = { $ioregs(imm) };
   dnd instantiate = {};
   ...
   execute = {
      $regs(rd) = $ioregs(imm);
   };
  execute "DND" = {
     $regs(rd) = $ioregs(imm);
     #regs(rd) = #ioregs(imm);
  }
};
```

In this example, `$ioregs(imm)` is instantiated before execute. In case Delayed Nondeterminism is used, the `dnd instantiate` directive is used before executing `execute "DND"`, which instantiates nothing (empty set). Note that the content of the two sections is independent and always defaults to the empty set, hence the `dnd instantiate` section is not required in this case.

Static Behavior Section

Syntax:

```
static behavior = {
  type = <type>;
```

```
  reads = {<reads list>};
  writes = {<writes list>};
  target = <target expression>;
}
```

- `<type>`: Optional attribute. Specifies the relationship between this instruction and succeeding instructions. The value of the attribute is needed for constructing a control flow graph for a program containing an instance of the instruction. It can be one of the following:
  - `add`: the successor of the current instruction is the next instruction in the stream. This is the default case, implying that the instruction does not explicitly modify the program counter.
  - `jump`: the successor of the current instruction is the instruction indicated by the value of the `target` attribute, which has to be provided when using this type.
  - `branch`: similar to `jump`, this type indicates that the instruction is a conditional jump. As such, it has two possible successors, both of which have to be indicated in the value of the `target` attribute.
  - `call`: similar to `jump`, the instruction causes execution to continue at the location indicated by the value of the `target` attribute. The difference is that in the CFG, the transition from the current to the target instruction has to be represented by a call edge. Additionally, the target is considered the start of a function instead of a regular node. Therefore, it becomes the first node of a new CFG fragment.
  - `end`: `end` is the counterpart to `call`, which is used at the end of a CFG fragment. It indicates that the current instruction does not have any successors.

  Note that for most instructions, the type can be derived automatically from the `execute` section by means of a static analysis. The static analyzer for SGDL, SGDL-STA, conducts such an analysis. Therefore, it is not necessary to specify a type, unless the automatically derived type deviates from the required type. In the latter case, an explicitly provided type overrides the result from the analysis. See Sect. 6.5 for details on the analyses.

- `<reads list>, <writes list>`: Mandatory attributes. Each of these lists is a comma-separated list of alias addresses accessed by the instruction in reading or writing, respectively. The addresses are specified in the same manner as in code blocks, that is, an alias name followed by an index in parentheses, and an optional bit index in brackets. Arithmetic expressions are also allowed in the indexes, for instance to compute an address dependent on an operand. When creating these lists, developers also have to add addresses accessed in subroutines called by this instruction (transitive hull of function calls).

- `target expression`: Mandatory attribute in case the instruction type is in $\{jump, branch, call\}$. The value of this attribute is an expression (therefore allowing arithmetic operations) describing the target of the jump. Subroutine calls in the expression are also allowed for computing the address.

  As static analysis operates not on the concrete, but on an abstract machine model, it may be necessary to add an offset to the target.

Example:

```
instruction "brcs" {
  encoding = BRSET where { bit = 0b000 };
  operandtypes = {target: SREL};
  cycles = 1;
  static behavior = {
    type = "branch";
    reads = {$C};
    target = relative(target, $pc) + 1;
  };
  execute = {
    if ( $C ) relativeBranch(target);
  };
};
```

The above example illustrates the similarity of static behavior section and execute section. This redundancy may be problematic, just as in any other program. Therefore, it is beneficial to have the information in the static behavior section derived by SGDL-STA whenever possible, and only provide an explicit description for those instructions for which the analysis fails to derive an accurate result.

Execute Sections

Syntax:

```
execute = {<code block>};
execute "DND" = {<code block>};
```

- `code block`: a list of statements, as described in Sect. 5.9

When the simulator has to execute an instruction, that is, apply the effects of that class of instruction to the resource model, it uses the code in either the `execute` or the `execute "DND"` section. Which one of these is selected depends on the active type of simulation. For details, refer to the chapter on abstractions (Chapt. 8).

5.3.4 Global Attribute "instruction word size"

Syntax:

```
global instruction word size = <integer value>;
```

The optional global attribute `instruction word size` specifies the number of bytes occupied by an instruction. This is used only in the disassembler: whenever the disassembler encounters a bit pattern that cannot be matched by an instruction pattern, it can either stop disassembling immediately (not knowing where the next instruction would start), or use this value to seek the beginning of the next valid instruction. If unspecified, the disassembler will create *InvalidInstruction* objects for all bytes following an unmatched pattern.

5.3.5 Subroutine Declarations

There are three types of subroutines: internally-defined subroutines, user-defined subroutines, and subroutines declared as external.

Internally Defined Subroutines

Internally defined subroutines are provided by the synthesis system. Technically, they are implemented in the synthesizer's code generation templates, resulting in generated code that can be adapted for each simulator. The implemented functions are shown below. The name before the colon is the name of the function, followed by the return type after the colon:

- `instructionSize(): ubyte`. Returns the size of the current instruction. Note that inside execute sections, the PC usually already points to the next instruction. This is the case because PC is incremented implicitly by the size of the current instruction before the code in the execute section is executed (unless the developer turns it off explicitly by the "`no auto increment of pc`" directive). Hence, the value returned by `instructionSize()` is actually the size of the next instruction. This function allows for the implementation of skip instructions.

- `notImplemented(): void`. Throws an error if called. Can be used to signal the use of code blocks that have not been fully implemented yet.

- `sleep()`: `void`. Sets an internal flag indicating that the device is asleep.
- `wakeUp()`: `void`. Sets an internal flag indicating that the device is not asleep.

User-Defined Subroutines

User-defined subroutines are declared and defined in the SGDL file, or files linked into the main file. The syntax is

```
subroutine <name>(<parameter list>) : <return type> { <code block> };
```

The parameter list can be empty or a comma-separated list of name and operand type pairs. Its syntax is shown below:

```
<identifier> : <operand class>
```

Hence, an identifier is followed by a colon and its data type.

The return type of the subroutine can be any operand class including the built-in types `void`, `ubyte` and `uint`. In case of a non-`void` return type, there must be a **`return`** statement at the end of each of the possible execution paths through the subroutine.

The code block consists of any number of statements. Sect. 5.9 provides details on code blocks.

Example:

```
subroutine getIndirect(addressReg:ADDRESS_REGISTER) : ubyte {
   local memoryContent: ubyte = 0;
   if (addressReg == 0){
      memoryContent = $sram( getRegInActiveBank(0) );
   }
   else{
      memoryContent = $sram( getRegInActiveBank(1) );
   }
   return memoryContent;
};
```

External Subroutines

Subroutines declared as external are declared inside the SGDL file, but their implementation is located in the file *<platform name>External.java*. The platform name is the name from the header of the SGDL file.

Syntax:

```
external <name>(<parameter list>) : <return type>;
```

Thus, syntax and semantics of the elements are the same as for user-defined subroutines, except that the code block is substituted by a semicolon.

An important difference between internal and external subroutines is that the latter are Java code instead of SGDL code. As such, they are not amenable to the analyses conducted by the SGDL static analyzer, SGDL-STA (cf. Sect. 6.5). Therefore, it is not possible to automatically derive the effects these functions have on the resource model of the simulated microcontroller. It is safe though inaccurate to assume that calling such a function may change anything, and that it might read any memory location. This effect propagates upwards to all callers of such a function. Hence, any `instruction` depending directly or indirectly on an external subroutine cannot be analyzed, and requires a static behavior section.

5.4 Atomics

Atomics assign names to memory locations. They serve two purposes: first, a memory location with a name can be referenced by that name in CTL formulas. Second, the name of a location is displayed on the GUI in the memory monitor on the simulation panel whenever the mouse cursor hovers over that location.

Syntax:

```
atomics = {<name list>};
```

- `<name list>`: a comma-separated list of entries of the form `<identifier> = $<alias name>(<integer constant>)`
- `<integer constant>`: a decimal or hexadecimal value denoting the address in the alias where `<identifier>` is located

Example:

```
atomics = {
   PCON = $sfrs(0x7),
   SBUF = $sfrs(0x19),
   SCON= $sfrs(0x18)
};
```

5.5 Loader Description

Loaders load the program from the compiler's output and write it into the simulated device's memory. The output from the compiler can be a container format like ELF.

In the loader description, there always is one section providing general information on the device and several optional sections, one for each class of loader.

Syntax:

```
loader general = {<general attribute list>};
loader elf = {<elf attribute list>};
loader hex = {<hex attribute list>};
```

Syntax of the attribute lists:

- <general attribute list>: contains attributes of the form <identifier> = <value>. Possible attributes:
  - PLATFORM_NAME: String attribute. Arbitrary value is possible.
  - PROGRAM_MEMORY_ADDRESSING: String attribute. Possible values BYTEWISE and WORDWISE
  - PROGRAM_SECTION_MAX_PHYSICAL_ADDRESS: Integer attribute.
  - GLOBAL_VARIABLE_OFFSET: Integer attribute.
- <elf attribute list>: contains attributes of the form <identifier> = <value>. Possible attributes:
  - MACHINE_CODE: Integer attribute. Denotes the machine code used in the elf format. The machine codes prevents tools for specific platforms from loading code intended to be used on other platforms (if they evaluate it). This is a multi-valued attribute, i.e., it can have more than one value if necessary. To assign more than one value, the developer may use multiple assignments to MACHINE_CODE within the same <elf attribute list>.
- <hex attribute list>: contains attributes of the form <identifier> = <value>. Possible attributes:
  - DEBUG_FORMAT: String attribute. Defines which debug file parser should be used for obtaining additional information about the program, such as where the compiler has placed variables, which code line in the assembly relates to which C code line etc. Possible values: Default, Avr.

5.6 Modeling of Peripherals

State space building in [MC]SQUARE is based on instruction-accurate simulation of the microcontroller. In each step, an instruction or an interrupt modifies the resource model, thus resulting in a successor state. From this stance, actions of peripheral components correspond to concurrent modifications of the resource model, which do not relate to the current instruction.

This view allows developers to model the on-chip peripherals by means of function calls that have to occur at certain events during successor state creation. For this purpose, SGDL provides so-called *triggers*, or hooks, which are called automatically. Currently, SGDL provides three types of triggers: `beforeInstruction`, `afterInstruction`, and `beforeAndAfterInstruction`.

A `beforeInstruction` trigger is executed at the beginning of each step, before the simulator starts dealing with instructions or interrupts. This can be used, for example, for deriving the nondeterminism status of I/O registers from the value of the according configuration registers. An `afterInstruction` trigger, on the other side, is executed after the step, but before internal cleanups (such as correcting erroneous nd mask/value combinations). Hence, the actions in such a trigger will still be subject to automatic cleanups. They can be used to conduct customized post-processing steps, which would otherwise have to be inserted into each instruction. If the same action has to be executed both before and after an instruction, the `beforeAndAfterInstruction` is the appropriate choice. During our implementation case studies, we found the `beforeAndAfterInstruction` trigger to be the most commonly used form.

Example:

```
trigger "DDRinputChange" event = "beforeInstruction" {
   // if bit in DDR is 0, then the same bit in PIN is
   // nondeterministic
   #PINA = ~$DDRA;
   #PINB = ~$DDRB;
   #PINC = ~$DDRC;
   #PIND = ~$DDRD;
};

trigger "CleanupExample" event = "afterInstruction" {
   $sram(1) = 0;
};
```

5.7 Modeling of Interrupts

The interrupt system can differ considerably between microcontrollers. Moreover, for our purposes, an accurate modeling of interrupts is crucial. Hence, the interrupt subsystem in generated simulators is very complex. Consequently, the associated language elements are also complex. Besides the `instruction` element, the subset of the language associated with interrupts is the most complex part of SGDL. For details on the mode of operation, refer to Sect. 7.3. This section first details how to capture the static properties of interrupts, that is, the interrupt vector table. Afterwards, we focus on the description of the operational behavior of interrupts. Operational aspects involve the checking whether an interrupt may occur, and how the resource model is modified in that case.

5.7.1 Interrupt Vector Table

Upon occurrence of an interrupt, microcontrollers usually first save the current value of the program counter in some location such as the stack, before loading a special fixed value into the program counter. These values, which are addresses in the program memory, are called *interrupt vectors*, and the set of all such values is called *interrupt vector table*. We require a description of the interrupt vector table for two distinct purposes: first, when conducting static analysis of a program for the microcontroller, we need to know the entry points of subroutines. Any interrupt vector represents an entry point to a subroutine, namely the interrupt service routine (ISR). Second, simulation on the GUI and counterexample representation benefit greatly from labelled transitions, as the labels ease understanding of the steps. For transitions related to interrupts, it is desirable to obtain and display the name of the interrupt that has occured, instead of being able to provide just a generic label such as *some interrupt*.

Technically, the desired properties can be implemented by creating a list of tuples of the form $(\textit{interrupt name}, \textit{program counter})$. In case a platform provides more than one vector for an interrupt (for instance, multiple vector tables that can be switched between), the developer has to provide multiple program counter values. We have decided to cover the latter case by allowing a comma-separated list of integers.

Syntax:

```
interruptvector {
    (interrupt <name> at pc <value>(, <value>)* ;)*

};
```

Example:

```
interruptvector {
   interrupt "RESET" at pc 0x0;
   interrupt "INT0" at pc 0x2;
   interrupt "INT1" at pc 0x4;
   interrupt "TIMER2_COMP" at pc 0x6;
};
```

5.7.2 Operational Behavior of Interrupts

Syntax:

```
interrupt <name> {
  id { <id> };
  sourceEnabled { <source condition> };
  activated { <activated condition> };
  post { <post code> };
  posted { <posted condition> };
  priority { <priority> };
  interruptvector { <vector computation code> };
  setNondeterminism { <set nondeterminism code> };
  getNondeterminism { <nondeterminism condition> };
  execute { <code block> };
};
```

- `<name>`: a unique name for the interrupt
- `<id>`: a unique identifier for the interrupt
- `<source condition>`: code for checking whether the event source for this interrupt is active
- `<activated condition>`: code for checking whether the interrupt is configured as active
- `<post code>`: code for posting any flags associated with the occurrence of the interrupt. When the interrupt system decides that the interrupt occurred, this code will be executed.
- `<posted condition>`: code for checking whether the flags for this interrupt have been posted
- `<priority>`: code returning an integer value indicating the priority of the interrupt. It is possible to consider the current machine state in this computation to model architectures with configurable interrupt priorities.
- `<vector computation code>`: code returning an integer value. The result is the current value of the interrupt vector for this interrupt, which may, on some architectures, depend on the machine state. The set union over all possible return values should be the same set as in the interrupt vector table (cf. Sect. 5.7.1).

- `<set nondeterminism code>`: code used for marking flags as nondeterministic if the event source of the interrupt is active.
- `<nondeterminism condition`: code for checking whether the flags are nondeterministic.
- `<code block>`: code to be executed whenever the interrupt handler is to be entered. Usually this consists of saving the program counter on one of the hardware's stacks and branching to the program location in the interrupt vector table.

5.8 Data Types and Type System

SGDL provides several pre-defined data types. These are described in Sect. 5.3.3. Additionally, the developer may create customized operand types, as detailed in Sect. 5.3.2.

Currently, there are neither String data types nor floating point types. All available types except boolean are integers, which can be either signed or unsigned. The `symbol` operand type allows a mapping of symbolic names to arbitrary integer values, that is, it is an ordinal type that can be used to realize enumerations.

The type system does not preserve any information about types when performing computations on data. This is realized by implicitly converting, or widening, the type to a new type *unknown* (cf. Aho et al. on type systems in compilers [2]). In assignments, in which the left-hand side is typed, this requires a following implicit conversion to the target type. A possible direction for future work would be to replace this behavior by a strongly typed approach based on type synthesis (also described by Aho et al.).

5.9 Code Blocks

Code blocks can contain statements and declarations of local variables. The syntax of statements resembles C and Java, whereas the syntax for declaring local variables is similar to Pascal. Statements can be assignments, control structures, and function calls. Each statement has to be terminated by a semicolon. This part of the SGDL language is for the most part inherited from ISDL. Our main contributions for code blocks are assignments to the nondeterminism mask, loops, scopes for variables, and named constants.

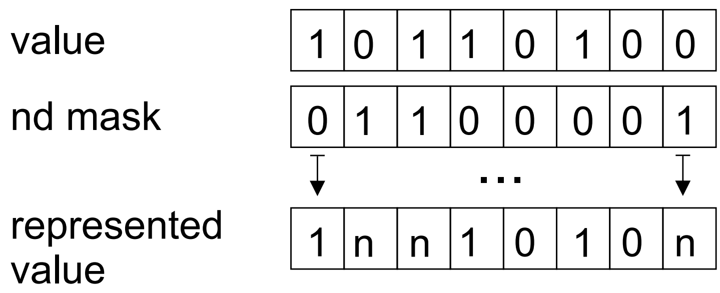

Figure 5.2: Representation of ternary values by means of parallel array structures. The bottommost line is not stored but illustrates the logical value.

5.9.1 Accessing Global Variables

We use the term *global variable* as a synonym for any accessible memory location in the resource model, that is, for aliases. Global variables, as opposed to local ones, always consist of two parallel and separate memory structures, as is shown in Fig. 5.2. The so-called *value* of a location represents the content of the location as it is present in the actual device, while the *nondeterminism mask (nd mask)* does *not* correspond to a physical memory. Both locations together are used to represent a value in ternary logic, which extends regular two-valued boolean logic by a third symbol meaning *unknown value*. Whenever a bit is set to 1 in the nd mask, we say the corresponding bit in the composed value is nondeterministic. Thus, it could be 0 or 1, meaning the actual content of the value storage is irrelevant.

Whether an assignment or a fetch accesses the value or the nd mask depends on the prefix of the alias. Using a $, that is, a dollar access, indicates an access of the value. Opposed to this, prefixing an alias with a # indicates an access of the nd mask. An example is shown in the next section.

5.9.2 Assign Statements

An assign statement assigns a value to a global or local variable. The left-hand side value (lhs) must be an identifier that can be resolved to an address. It denotes the target location of the assignment. The right-hand side value (rhs) value, which is the source of the assignment, can be a literal, an expression, or a function call.

Syntax

```
<target> = (<literal> | <expression> | <function call>);
```

Example:

```
$sram(10) = 12;
$sram(11) = 0xff;
$registers(0x10) = calcSomeValue();
someLocalVariable = calcSomeValue() + 1;
$someBitAlias = true;
```

The example illustrates that the allowed values differ depending on the type of the target. Assignments to bit aliases require that the source either is a boolean value or contains such a value. Other alias types allow for integer values instead. Moreover, local variables, in this example `someLocalVariable`, can and must be accessed without providing an address. In contrast to this, memory aliases require an address. Register aliases may be used with a bit index instead, and bit aliases need neither.

Value Versus Nondeterminism Mask Assignments

Syntax

```
$<target> = (<literal> | <expression> | <function call>);
#<target> = (<literal> | <expression> | <function call>);
```

As described earlier in this section, there are two parallel memory structures for each global variable. A bit that is set to one in the nd mask (i.e., the structure accessed with a hash) is considered to be nondeterministic. For instance, consider `sram` to be an alias with a block size of 1 byte. Then, to mark all 8 bits in address 0 as nondeterministic, we can write

$$\#sram(0) = \mathit{0xff}$$

In this case, the value of $\$sram(0)$ becomes irrelevant, as the symbolic value of the location is $(nnnn\,nnnn)$. Thus, if any computation requires the actual value of this memory cell, each of the n bits has to be instantiated, resulting in all possible bit patterns including the one actually stored in the \$ location.

Bit Assignments and Bit Reads

Bits are accessed by an integer enclosed by brackets (i.e., []). Example:

```
$carryFlag = result[8];
```

This assignment fetches the value of the bit at index 8 in `result`, implicitly converts it to boolean by checking whether it is inequal to 0, and assigns the resulting boolean object to the alias named `carryFlag`.

Ranges of bits can be read and written as well. For this, developers have to provide multiple integers, separated by a colon, that is, `[n:k]`.

Example:

```
local temp1: uint = a1[3:0];
temp[7:0] = target[7:0];

subroutine lowByte(v: uint): ubyte {
   return v[7:0];
};
```

When accessing bits inside aliases of the `memory` type, it is also necessary to specify the index in the alias as well. Example:

```
$sram[7] = 1; //wrong, no index specified
$sram(0xa)[7] = 1; //correct: write to address 0xa at bit 7
```

This addressing using both an address and a bit is neither necessary nor allowed for aliases of the types `register` and `bit`. The reason why `memory` aliases require them is that this type of alias offers the most versatile (i.e., direct) access to the underlying memory array. Register and bit aliases provide more abstraction because the array structure of the underlying array is not visible.

5.9.3 Control Structures

The available control structures in SGDL are `if` and `for` statements.

Syntax of `if`

```
if (<condition>) <statement>
if (<condition>) { <statement block> }
```

Optional `else`:

```
else <statement>
else { <statement block> }
```

Example:

```
if ( dist == 4 ) cyclesConsumed = cyclesConsumed + 2;

if(($TCCR0 << 5) == 0){
   #TCNT0 = 0x00;
   #TIFR = #TIFR & ~((1 << 0) | (1 << 1));
}else{
   #TCNT0 = 0xFF;
}
```

Syntax of for

```
for (local <variable> : <data type> = <value>; <condition>; <action>)
  <statement>
for (local <variable> : <data type> = <value>; <condition>; <action>)
  {<statement block>}
```

- <variable>: a variable identifier
- <data type>: an integer data type
- <value>: an integer value
- <condition>: a condition that must hold while in the loop
- <action>: an action to be executed in each iteration of the loop, typically an incrementation of <variable>
- <statement>: a single statement as the loop body
- <statement block>: the loop body, consisting of multiple statements

Example:

```
for (local i: uint = 96; i < 1120; i = i + 1 ){
   $sram(i) = 0;
   #sram(i) = 0xff;
}
```

5.9.4 Local Variables

Local variables can be declared inside any code block. Their visibility and validity is restricted to that specific code block.

Syntax

```
local <identifier> : <data type> = <expression>
```

- <identifier>: a name for the variable
- <data type>: a data type for the variable
- <expression>: the initial value for the variable, given as a literal, an expression or a function call.

Example:

```
local byteAddress : uint = bitAddress / 8;
```

5.9.5 Function Calls

Functions, or subroutines, can be called in any expression or as a statement (procedures), that is, they can be called in any code block. Functions are allowed to have side effects, meaning they can modify the resource model. While side effects are not desirable in regular programs, and should be avoided for most functions in SGDL as well, there are a few use cases for them when modeling processors. For instance, a function can be used to compute the result of the addition of two register contents. The same function can also set the device's flag register accordingly. Thus, multiple instructions can share the same function, which is important because processors usually offer variants of the same instruction, e.g. *add register to register*, *add immediate to register*, or *addition with* versus *without carry*.

Examples:

```
callSomeFunction();
callSomeFunction(1,2,3);
variable = someFunction( $sram(1000) );
```

5.9.6 Accessing the [mc]square options

The generated code can react to the [MC]SQUARE options, which allows users of [MC]SQUARE to change behavior at runtime. In SGDL, the global options of [mc]square can be accessed by means of the symbol OPTIONS.

Syntax:

```
OPTIONS.<keyword>
```

where `<keyword>` is one of the following:

- `USE_LSE`: evaluates to true at runtime in case the user has enabled Lazy Stack Evaluation.
- `DET_SIMULATION`: evaluates to true at runtime in case the user has deactivated Delayed Nondeterminism.
- `DND_SIMULATION`: evaluates to true at runtime in case the user has activated Delayed Nondeterminism.

Example:

```
if (OPTIONS.USE_LSE){
   $sram(0) = $sram(0);
}
```

5.10 Comments

Comments in SGDL are equivalent to comments in C and Java. Hence, there is a type of comment marking the rest of the current line as a comment, and a multi-line comment.

Example:

```
//this is a single line comment
<some language expression>; //the rest of this line is a comment

/*
 * This is a multi-line comment.
 */
```

5.11 Constants

SGDL code can contain `#define` preprocessor directives just like C (cf. Sect. 5.12). These are replaced by a preprocessor which is run in case at least one such directive occurs in the input. However, just as in C, there is no type checking for this type of constants. Therefore, SGDL also contains typed constants, which are to be preferred.

The syntax for declaring typed constants resembles the syntax for declaring and immediately defining local variables:

```
const <identifier> : <data type> = <expression>
```

Example:

```
//Example: declaration of constants
const intValue : ubyte = 1;
const someFlag : boolean = true;
const SREG_ADDRESS : uint = 0x95;
const I_BIT : ubyte = 7;

//Example: assignment using constants
$sram(SREG_ADDRESS) = $sram(SREG_ADDRESS) | (1 << I_BIT);
```

5.12 Preprocessor Directives

SGDL files may contain preprocessor directives in the style of the C programming language, which are processed by a preprocessor run before the actual compiler. The preprocessor can perform various actions related to string pattern matching and string replacement. Similar to C, directives in SGDL consist of a hash followed by a keyword, e.g. `#define`. Some of the directives serve as a *kick-starter* for simulator development, i.e., they are translated into SGDL template code by the preprocessor. Thus, it is possible to accelerate recurring tasks such as implementing instructions the addressing modes of which are already known. The output is placed into a separate file, from which the developer may fetch it and insert it into the original SGDL file. This provides text editor-independent support during development. In the following paragraphs, we detail the available directives.

5.12.1 Defines

`#define` works just like its C counterpart:

```
#define key value
```

where `key` is an identifier and `value` can be any string. Every occurrence of `key` in the input file will be replaced by `value`. The positioning of the directive within the code is irrelevant, as the SGDL preprocessor passes twice over the input. On the first pass, it collects all the key-value pairs, and on the second pass, it replaces occurrences of keys by their values.

5.12.2 Generation of Empty Source Files

`#create_template` is the first of the kick-starter templates. When used, it will be replaced by a skeleton SGDL file containing the basic structure of a simulator.

5.12.3 Instruction Templates

`#instruction` generates empty instruction elements. Three variants are available:

- `#instruction name`: creates an empty instruction called `name`
- `#instruction name` $\texttt{encoding}_1$ $\texttt{encoding}_2$`, ..., `$\texttt{encoding}_n$:
  creates n empty instructions, each called `name`, one for each encoding.
- `#instruction name n`: creates n empty instructions, each called `name`. Each of these has its own unique encoding, which is created by appending increasing numbers to a fixed prefix.

5.12.4 Alias Templates

`#alias` creates empty alias declarations. As there are three alias types, each of which requires a different set of attributes to be defined, there also exist three variants of this preprocessor directive:

- `#alias name memory`: creates a new alias of the `memory` type, which is used for ranges consisting of multiple addresses.
- `#alias name register`: creates a new alias of the `reg` type, which is used for single addresses consisting of up to 32 bits.
- `#alias name bit`: creates a new alias of the `bit` type, which is used for single bits.

5.12.5 Interrupt Template

`#interrupt name` creates an empty interrupt called `name`.

5.13 Compilation Units

SGDL descriptions of microcontrollers may consist of multiple compilation units. The main compilation unit must have the file extension ".sgdl". All units that can be integrated into another unit need to have their file extensions set to ".fragment.sgdl". It is possible to integrate a fragment into another fragment. The `include` statement directs the SGDL compiler to process another file before continuing with the current file.

Syntax

```
include <relative path>;
```

- `<relative path>`: a path to a fragment file. The path is relative to the directory containing the current file (i.e., the file containing the `include` statement).

6 SGDL Compiler

In this chapter, we detail the structure of the SGDL compiler. The core compiler is a stand-alone command line tool that is independent of [MC]SQUARE and also does not depend on a specific development environment. In order to facilitate development, though, we have added an Eclipse plugin, which automatically recompiles simulators in case any of their source files is modified, and which also features error highlighting in the source code. Additionally, we have added syntax highlighting for SGDL in Programmer's Notepad, which is an editor that ships with the WinAVR compiler package.

6.1 Outline

The task of the SGDL compiler is to transform the SGDL description of a microcontroller into an executable form. The executable form, from our point of view, is a state space generator (simulator) in the form of Java source code. Even though this is not immediately executable, it already consists of concrete instructions (as opposed to the declarative character of SGDL), and can be translated into executable Java bytecode by the Java compiler.

Technically, the SGDL compiler is a special-purpose compiler with a structure similar to standard compilers, as described for instance by Aho et al [2]. Its runtime structure is shown in Fig. 6.1, whereas Fig. 6.2 shows its package structure in the form of an UML package diagram [19, 21, 42]. The following sections focus on components of this structure. On the side of the input, apart from the actual compiler, are the data sheet of the microcontroller or microcontroller family, the SGDL input file, and the simulator information file. On the side of the output, there is the generated simulator.

A data sheet is the usual starting point for any developer wanting to implement a simulator. This holds true regardless of whether a tool is used, such as our synthesis tool chain, or whether the developer implements the simulator directly in an imperative programming language such as Java. SGDL was designed to be similar to the notations used in data sheets, thus to allow developers to translate the information they already have to a machine-usable representation with very little effort.

The SGDL description is the result of the manual translation. It has to contain all the platform-specific information that is to be used in the simulator. Fur-

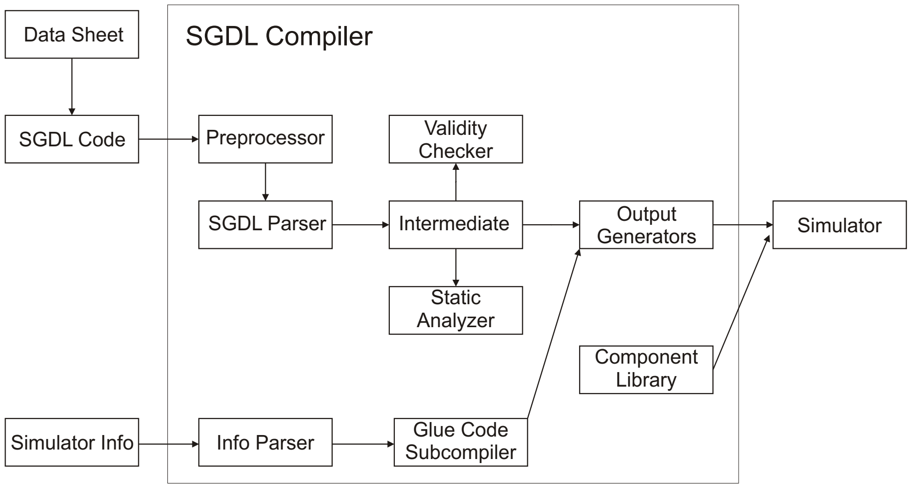

Figure 6.1: Structure of the SGDL compiler

thermore, the developer may augment the description by some meta information about the code to be generated, such as its location in the Java package structure of [MC]SQUARE. If present, such meta information allows the SGDL compiler to generate some glue code, which automatically integrates the new simulator into [MC]SQUARE.

6.2 Preprocessor

The preprocessor is the first component of the SGDL compiler that processes the input. Its main purpose is to provide a means for quickly generating code fragments, but it also provides means for defining constants. Technically, it performs simple string pattern matching and replacing. The resulting output is then passed on to the actual SGDL parser. A more detailed description of the preprocessor and the available directives is given in Sect. 5.12.

6.3 Parser

The SGDL parser reads the input provided by the developers and translates it into the internal representation. While doing so it additionally checks for syntax errors and also some semantic errors. Regarding the latter, the parser creates symbol tables (one new table per declaration block) in the usual tree-like manner. Based on these tables, the parser can decide whether identifiers have been declared before

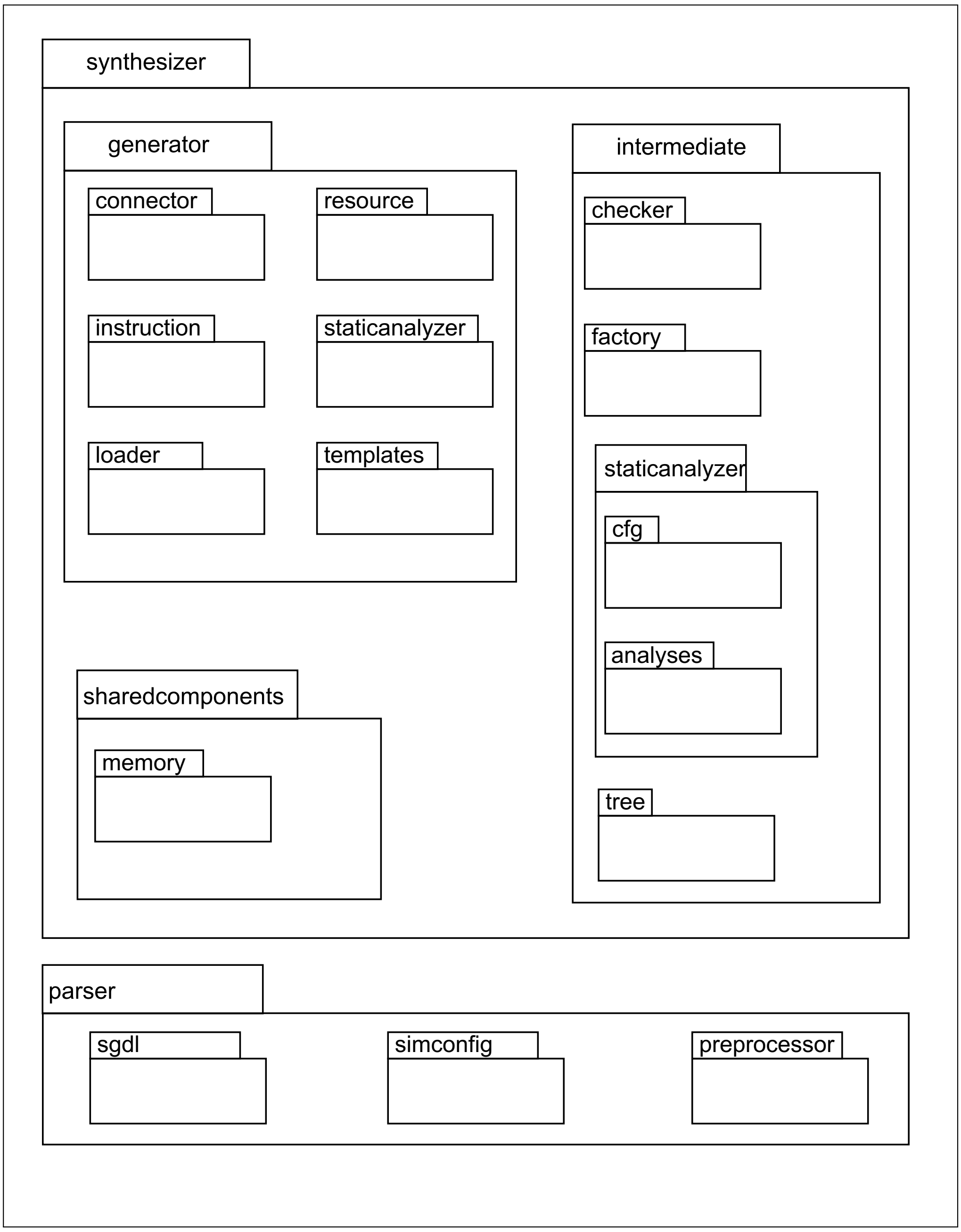

Figure 6.2: UML package diagram of the SGDL compiler

their use. The symbol tables are stored for later use during the static analysis and subsequently, the synthesis phases. For further details on the construction and uses of symbol tables, we refer to [2].

Originally, the parser was based on the ISDL parser from the AVRora project, though, as the new language evolved, we added more and more extensions and modifications of our own. It is still based on a grammar for the JavaCC [57] parser generator.

6.4 Abstract Syntax Tree and Intermediate Representation

Logically, the SGDL compiler contains two intermediate representations. The first of these is the Abstract Syntax Tree, created on-the-fly from the parse tree while parsing the input. The second is the actual intermediate representation, which is created after parsing, and is more amenable to synthesis purposes than the AST.

Temporally, however, the compiler contains *three* intermediate representations. Certain information is not present in the AST because it has to be derived from it by combining information that is split among several entities. Two steps require such a preprocessing: first, the checking for errors. For instance, no two instructions may have the same binary encoding. If we allowed this, then we could not decide which instruction to create when we encounter that bit pattern in the binary instruction stream. However, in the AST, there are only instruction elements, some of which may be referring to format elements, but there is no binary pattern associated with instructions. Such patterns only become visible when merging the information from instruction and format elements. The second major step relying on preprocessed information is the static analysis of the input by the SGDL static analyzer, SGDL-STA. Analysis results are required for the synthesis of code, which is why the analyzer has to operate on a not yet complete intermediate representation.

Concerning sources, only the first intermediate representation is based on AVRora. All subsequent components, especially those related to error-checking and static analysis, were added within the scope of our project.

6.5 Static Analyzer

The SGDL compiler features a static analyzer, SGDL-STA, which can automatically deduce certain information about an architecture. Such information may then be used for a variety of purposes, such as enhanced error detection while creating the SGDL description, or to eliminate the need for explicit description of already obvious features.

The foremost reason for us to analyze instructions by means of a static analyzer is the verbosity and redundancy in the SGDL Static Behavior Section (SSBS). This section, which is an optional part of the `instruction` element (cf. Sect. 5.3.3), describes certain properties of instructions:

- which locations are read and written when executing the instruction
- the type of instruction, e.g. whether it is a regular instruction, or manipulates the control flow (call, return, or a conditional / unconditional jump)
- for control-flow manipulating instructions, the number of succeeding instructions, and how to determine their addresses

If this information is available, the SGDL compiler can generate an operative static analyzer for the target platform (not to be confused with the analyzer for SGDL itself, SGDL-STA). The description of read and written locations is a prerequisite for generating the LVA and RDA builders, and the control flow type is necessary for the target analyzer because it has to be able to construct a control flow graph from machine code. Hence, without the SSBS, the target simulator is not equipped with an operative analyzer, which in turn deactivates all abstractions based on static analyses.

Comparing the `execute` section and the SSBS reveals that the former already contains all the required pieces of information, though in the form of SGDL code blocks. Locations read and written can be deduced from expressions and assignments in SGDL. Similarly, the control flow type can be determined. The idea for this is to consider the way an instruction manipulates the program counter, and, in case of calls and returns, the stack. We detail this procedure in one of the following subsections. In case the analysis conducted by SGDL-STA succeeds, it it still possible to provide an SSBS, but it is no longer required. Our experiments have shown that for the platforms we implemented, in fact very few instructions remain that require manual annotation in an SSBS, while for most instructions, the analyzer correctly derives the type.

6.5.1 Mode of Operation

A run of SGDL-STA for a given platform description in SGDL consists of multiple phases. First of all, the analyzer constructs control flow graphs including call edges. Next, it runs each of the individual analyses, which in turn can be decomposed into an intraprocedural analysis and an interprocedural analysis. Upon completion of the analyses, SGDL-STA enters the deduction phase, in which the information gained during the analysis phase are used to obtain useful facts about the platform under consideration. A more detailed description of each of these phases is given in the following paragraphs.

Control Flow Graph

All static analyses performed by SGDL-STA operate on control flow graphs (CFGs), each of which includes call edges, i.e., call graphs. A CFG is constructed for each code block in the input: `execute` sections in instructions, subroutines, and the code blocks in interrupts. For interrupts, the relevant blocks are the `execute` section (executed when the interrupt has occurred), the `post` section (executed to simulate the effect of setting the interrupt flag), and the `priority` section (used in determining the active interrupt with the highest priority, if any).

The procedure for each code block is the same, regardless of where it is used. First of all, a node is created for each of the instructions of which the code block is composed, and bidirectional edges are added to link each such node to its successors and predecessors. Nodes obtain a reference to the AST object representing the instruction. Next, the construction algorithm creates a virtual line numbering and assigns a number to each of the nodes. Line numbers are required for identifying nodes in later phases of the analysis, and also to be able to inspect the CFG in a textual representation for debugging purposes. Having enumerated all the nodes, the construction algorithm then adds synthetic entry and exit nodes that do not correspond to any instruction. These serve special purposes during the analyses. Entry nodes only have one successor, which is the first instruction in the code block. Exit nodes may have multiple predecessors, due to branches or multiple `return` statements in a code block. An example is shown in Listing 6.1 and the corresponding graph in Figure 6.3.

```
instruction "mul_a_b" {
...
   execute = {
      local result : uint = $ACC * $B_REG;
      $ACC = lowByte(result);
      $B_REG = highByte(result);
      if (result > 0xff) {
         $OV = true;
      }
      $CY = false;
   };
```

Listing 6.1: Execute block of the SGDL description of the instruction `MUL AB` from the MCS-51 instruction set, which multiplies the contents of the accumulator (`ACC`) and of register B (`B_REG`)

Upon completion of the construction of CFGs for all code blocks, the CFG construction algorithm iterates again over all CFGs and searches for function calls. A function call is represented by a call edge that is added to the node in which

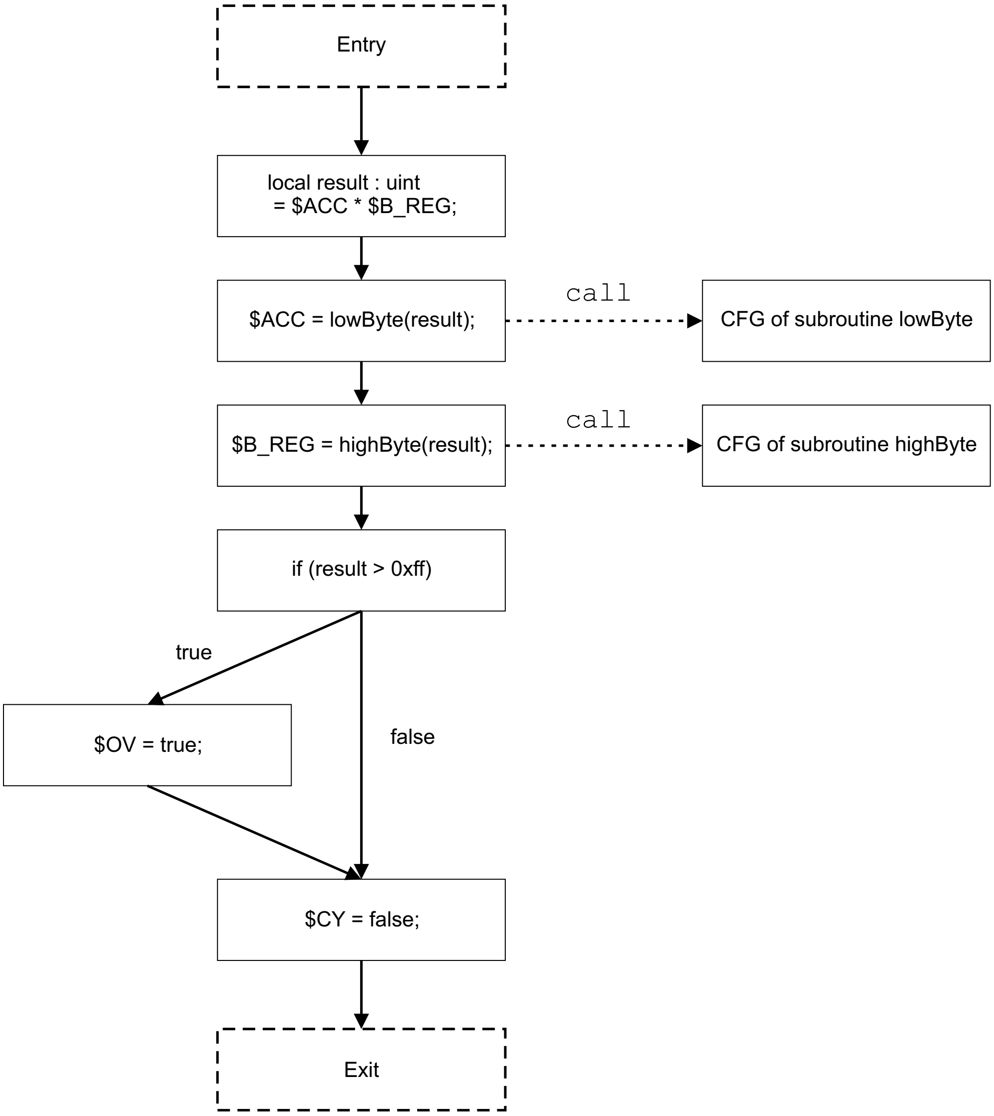

Figure 6.3: Control flow graph for the MCS-51 instruction `MUL AB`

the call occurs. The target of a call edge is always an entry node of another CFG fragment. In case the function call is not caused by a call statement but is part of an expression, it is also possible that a node is the source for multiple call edges.

Analyses

SGDL-STA is capable of both intraprocedural and interprocedural analysis. An intraprocedural analysis focusses on a single CFG and ignores any call edges contained therein, whereas interprocedural analysis takes the call edges into account. Interprocedural analysis is required because any code block may contain function calls, and subroutines may change the global resource model representing the microcontroller memories. We decided to implement interprocedural analyses using the call summary approach [49], which can be summarized as *analyze individual methods, then project the results into the source nodes of call edges*. In the form we implemented it, this is a context-insensitive analysis, though this could be improved, for instance by means of call strings (cf. [49]).

Using summaries for function calls allows us to also handle recursive functions, including indirect recursion, i.e., recursion that involves one or more intermediate functions before the recursion becomes obvious. However, we decided to exclude recursion to facilitate the structure of the analyzer. Therefore, all functions involved in recursive calls are not analyzed. This is achieved by computing the transitive hull of function calls and intersecting it with the list of functions known to be recursive. Consequently, instructions and interrupt code blocks calling any such functions are also excluded. A similar exclusion is necessary for external subroutines, which are implemented as Java code outside of the SGDL description. As these are not accessible for SGDL-STA, we would have to over-approximate their behavior by assuming that they could read and write anything. Given that this information would propagate along call edges to callers of such functions, the result for any method, instruction or interrupt code block involved in such a call hierarchy would be that no accurate analysis result could be obtained. Thus, excluding these from the analysis phase does not impact correctness but saves time because the result is already known.

Currently, SGDL-STA supports three analyses: Reaching Definitions Analysis (RDA), Read Variable Analysis (RVA), and Written Variable Analysis (WVA). All of these are forward analyses, that is, information is propagated along the CFG edges in the same direction as the program would be executed.

Reaching Definitions Analysis is a standard textbook analysis (cf. for instance Nielson et. al. [49]), which determines, for each variable v in a program and program location l, the set of preceding locations $\{k_1, \dots k_n\}$ where v was last assigned a value. Variations of RDA also collect the assigned value or expression along with the location, allowing to recognize the set of possible values v may have at l.

Read Variable Analysis is a variant of another textbook analysis, which is called Live Variable Analysis (LVA). Technically, it is what Nielson et al. [49] refer to as the *collecting semantics* of LVA, that is, an LVA with the usual *gen* function and an empty *kill* function. This ensures that information about read variables is only added to the analysis information sets, but never removed. Applied to a CFG, the RVA information in the exit node therefore provides an overview of all variables ever read within the scope of the associated code.

Written Variable Analysis is the counterpart of RVA with regard to written variables. Hence, the WVA result in exit nodes points out which variables could possibly be written by the code fragment. As with RVA, the fact that no information is ever removed allows this analysis to be executed either as a forward or a backward analysis without altering the result.

Fixed Point Computation

Static analysis for SGDL is conducted by performing a fixed point iteration over each of the equation systems induced by individual code fragments. That is, nodes in the CFG representing instructions contain an entry set AI_{entry} and an exit set AI_{exit}, $AI \in \{RDA, RVA, WVA\}$, representing the status of the analysis information before and after executing the instruction. An instruction defines a transfer function that describes how to modify the entry information in order to obtain the exit information. Consecutively applying the transfer functions eventually leads to a fixed point, which is reached as soon as re-applying the transfer functions to the associated AI sets does not change anything anymore.

In the first phase, SGDL-STA computes fixed points for individual CFGs, that is, intraprocedurally. In the second phase, an interprocedural fixed point computation uses the results from individual CFGs to compute fixed points for callers, based on function summaries. In case anything changes during a fixed point computation, the procedure is repeated. This corresponds to the global fixed point computation algorithm, which is the most basic fixed point computation strategy in the literature. For the rather small code fragments we encountered so far in microcontroller descriptions, this simple strategy did not cause any performance problems. If this should ever pose a problem, it could be remedied by implementing one of the more sophisticated algorithms, such as the worklist algorithm [49].

6.5.2 Structure of the Analyzer

Figure 6.2 shows the overall structure of the SGDL compiler, of which SGDL-STA is a part. The packages named therein directly relate to Java packages of the same name. As SGDL-STA operates on the intermediate representation, its package structure is located inside the `intermediate` package. The root package for SGDL-STA is called

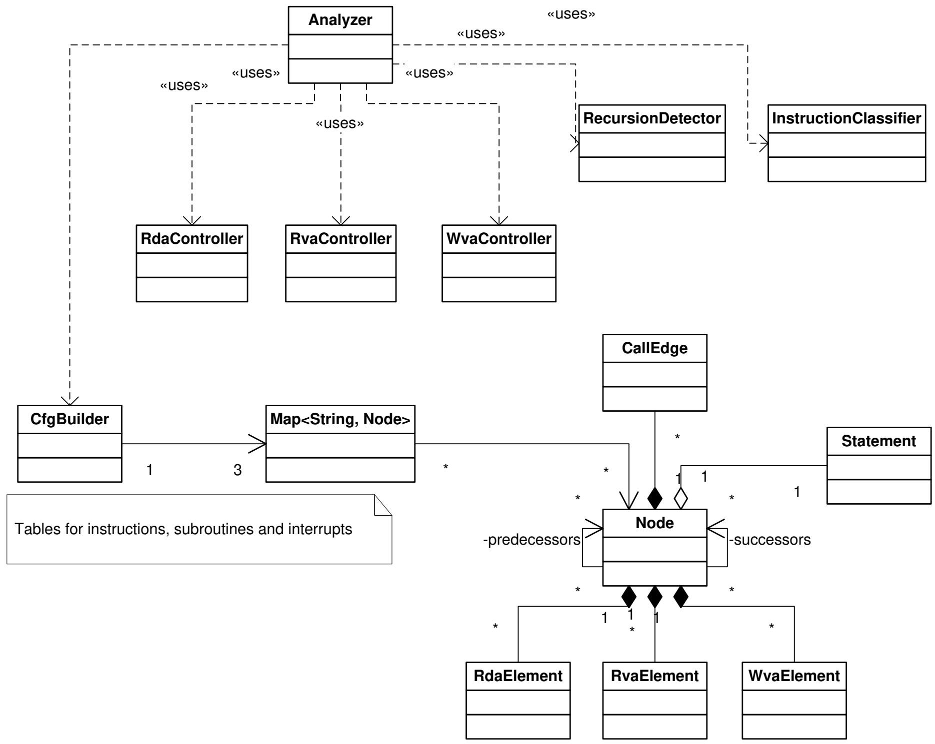

Figure 6.4: UML class diagram showing the most important classes of SGDL-STA

staticanalzyer.

Classes in the topmost package, `staticanalyzer`, either contain such functionality that is not part of any specific analysis, or provides access points for other parts of the compiler. Creation, manipulation and traversal of the CFG is handled by the classes inside the package `cfg`. Finally, `analyses` contains the actual analyses, the algorithm for fixed point computation, and the representation of the analysis information by which the CFG can be annotated.

An overview of the most important classes is shown in the UML class diagram [19, 21, 66] in Figure 6.4, and their sizes are listed in Table 6.1. For the sake of brevity, less relevant classes have been omitted, and are also not contained in the description below.

- Package `staticanalyzer`
  - *Analyzer*: the core class of SGDL-STA, which controls the analysis process and provides an interface to other components, both for starting an analysis and for retrieving the results.

  - *InstructionClassifier*: determines the type of an instruction on the basis of its behavior. This class is covered in detail in Sect. 6.5.3.
  - *RecursionDetector*: computes the set of recursive instructions and functions.

- Package `cfg`
  - *CallEdge*: represents a call edge in the call graph.
  - *CfgBuilder*: constructs control flow and call graphs for intra- and interprocedural analyses.
  - *CfgTools*: encapsulates frequently used graph algorithms, such as depth first search, getting a list of exit nodes, and getting a list of called subroutines.
  - *Node*: represents a node in a CFG. Nodes contain a pair of analysis information sets for each analysis, one representing an entry information and another the exit information. Nodes may also contain any number of references to call edges.

- Package `analyses`
  - *AbstractAnalysisController*: a common super class for all analysis controllers. An analysis controller is responsible for conducting a single analysis such as RDA. For the most part, the abstract analysis controller is a container for data structures and for references, for instance to a *CfgBuilder*.
  - *AbstractAnalysisInformation*: an interface defining methods for joining and cloning analysis information (AI) sets. By using this interface, we have decoupled the analysis algorithm from the data on which it operates.
  - *AnalysisAlgorithms*: generalized algorithms for static analysis. Currently contains an implementation of a forward traversal algorithm, based on a global fixed point iteration.
  - *BaseAnalysisInformation*: a lattice element containing entries.
  - *MemoryCellId*: this class and its subclasses provide a way of assigning unique identifiers to memory locations, which are required as elements of analysis information sets.
  - *RdaBuilder*: encapsulates the RDA-specific part for conducting an analysis. This is technically an implementation of the strategy pattern (cf. Gamma et al. [23]), which parameterizes the algorithm for computing a fixed point. The most important parts of this include a definition

of the *kill* and *gen* functions for all language elements that could be encountered, and statement visitors for both the intraprocedural and interprocedural modes of operation.

- *RdaController*: controls the RDA. This involves the following steps:
  * Initialization of all RD_{entry} and RD_{exit} sets in all CFGs
  * Creation of an appropriate intra- or interprocedural *RdaBuilder*
  * Conducting the RDA for instructions, subroutines, and interrupt code blocks
- *RdaElement*: represents an analysis information for RDA, such as RD_{entry} and RD_{exit}
- *RdaInformationHandler*: subclass of *AnalysisInformationHandler*, which determines which analysis information sets are to be manipulated. Allows the analysis algorithm to operate on the appropriate sets without having to know them, i.e., decoupling. In this case, accesses are directed to the RDA sets.
- *RvaBuilder*, *RvaController*, *RvaElement* and *RvaInformationHandler*: classes for RVA, analogously to the classes of the same name for RDA
- *WvaBuilder*, *WvaController*, *WvaElement* and *WvaInformationHandler*: classes for WVA, analogously to the classes of the same name for RDA

6.5.3 Instruction Set Classification

As pointed out in the motivation for this section, SGDL-STA should derive the control flow type of instructions to avoid requiring the developer to denote it explicitly in the SSBS. For an instruction to be classified, the classification algorithm requires the RDA, RVA and WVA to have been completed successfully. The idea of the algorithm is to examine read and write accesses to certain special memory addresses, that is, the program counter (PC) and the stack pointer (SP). An illustration of the steps is shown in the UML activity diagram [19, 21, 42] in Fig. 6.5.

Combined with the analysis phase, the classification algorithm works as follows (as described in one of our previous publications [27]):

- Construct the control flow graphs for the current instruction and all called functions, including transitive calls
- All instructions implicitly increment the program counter by their own size, so insert one reaching definition (RD) for the program counter (PC) into the entry node of the CFG for the instruction

Class	Lines of code
Analyzer	322
InstructionClassifier	266
RecursionDetector	93
Other classes in package `staticanalyzer`	619
AbstractAnalysisController	88
AbstractAnalysisInformation	22
AddressBasedMemoryCellID	59
AnalysisAlgorithms	101
AnalysisBuilder	46
AnalysisInformationHandler	25
BaseAnalysisInformation	235
MemoryCellAnalysisInformationEntry	171
MemoryCellID	24
RDA-related classes	855
RVA-related classes	465
WVA-related classes	443
Other classes in package `analyses`	134
CallEdge	30
CfgBuilder	571
CfgTools	140
Node	159
Other classes in package `cfg`	110
Total	4,978

Table 6.1: Code size overview for SGDL-STA

- Conduct all analyses
- Classify instructions based on the number and origin of RDs for PC reaching the exit node:
  - 1 RD from the entry node: regular instruction (PC is not modified in the code block)
  - 1 RD, but not from the entry node: program counter is inevitably overwritten with a single value, i.e. an unconditional jump instruction
  - 2 RDs: a conditional jump
- Refinement phase: jump instructions manipulating the stack could be call or return instructions, depending on whether they push the PC to or pop it from the stack. These operations can be distinguished by the associated read and write operations, which are represented in the results of RVA and WVA.

While this algorithm yields the type of an instruction, it does not provide the target for jumps, calls and returns. To this end, we developed a concept in [27], which would enable a subsequent analysis to obtain the jump targets as well. Summarizing it, it is based on locating assignments to PC by means of a Reaching Definitions Analysis, and then attempting to obtain the expression assigned to PC. The procedure involves a backwards search through the CFG in order to collect subexpressions used in such assignments, which appears feasible to us for the rather small and simple-structured code fragments used in `execute` sections. For instructions for which this succeeds, an explicit entry in the SSBS becomes obsolete. All other jump, call and return instructions still require it.

6.5.4 Further Uses

Automatic Derivation of Instruction Semantics

As pointed out in the last section, we investigated how to obtain jump targets for jumps and function-call related instructions. The same approach could possibly be generalized and used to automatically rewrite the semantics of given instructions. This would then not focus on jumps, but rather on arithmetic and logic operations as well as data transfer instructions. The overall idea is to avoid operating on concrete values, but to use symbols instead. A thorough description of the concept, which we have not implemented yet, is given in one of our publications [27]. The feasibility of the approach depends on whether SGDL-STA can be extended such that it collects all sub-expressions used in assignments, which would allow rewriting them based on the free variables of a code block (i.e. parameters or operands).

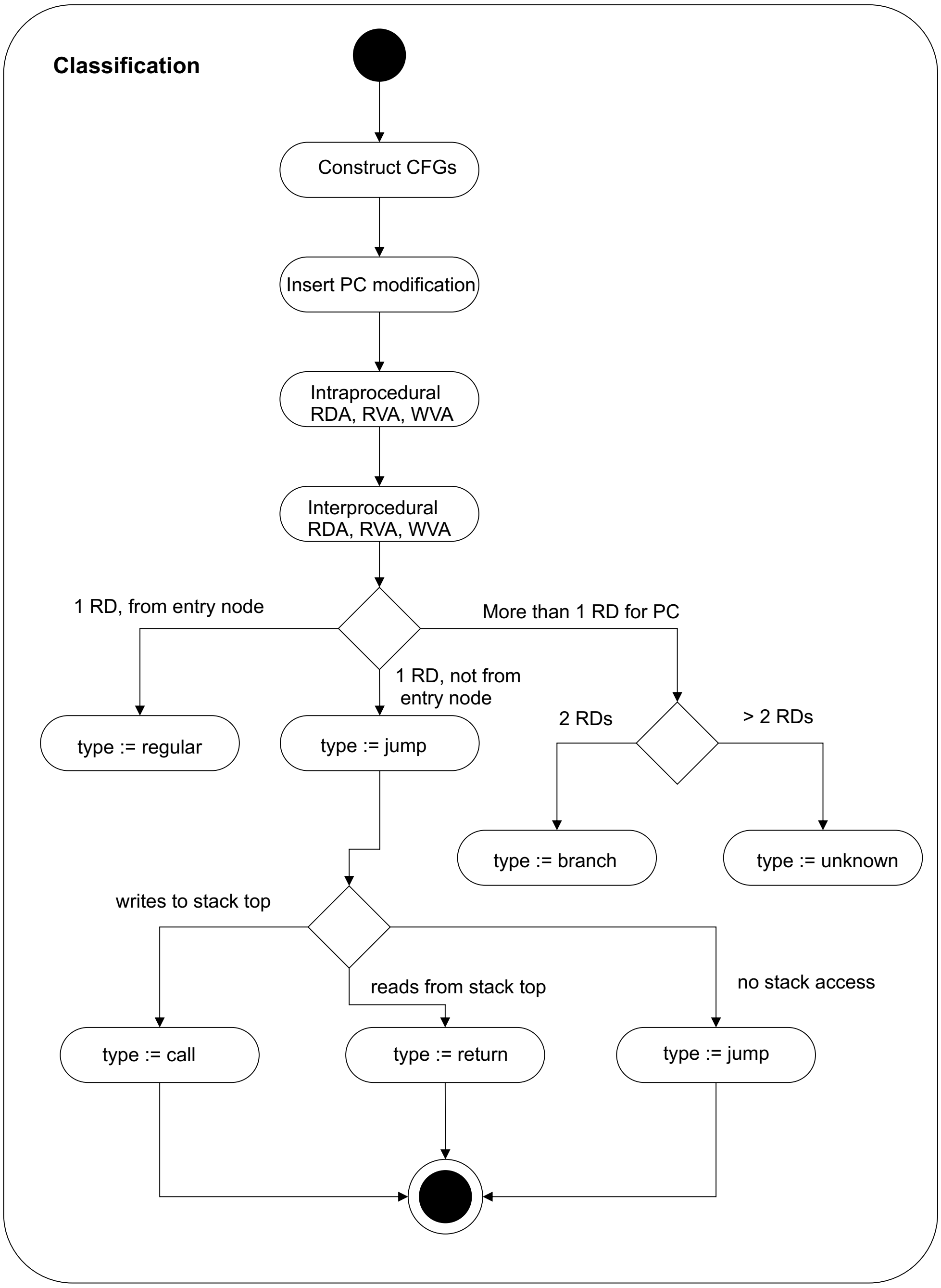

Figure 6.5: Instruction classification algorithm as an UML activity diagram. The outgoing edges from the final nodes for types *regular*, *branch* and *unknown* to the exit node have been omitted for clarity.

Instruction Disassembling and GUI Presentation

Instructions should be displayed on the [MC]SQUARE GUI in a human-readable way. As SGDL forces all instructions to be given a name, it is simple to display a name instead of a bit pattern. Unfortunately, it is not as simple with the operands, as their purpose is not statically known. For instance, the AVR instruction `OUT` *0x3e 0x1d* should rather be presented as `OUT SPH, R29`.

For enumerable locations such as registers, SGDL has inherited a means from ISDL for assigning names to them, but only if the reference is stored in an appropriate data type. Therefore, it is possible to create the output R29 instead of *0x1d*, but the approach is not applicable to larger memories and fails in the presence of instructions adding an offset to their operands. `OUT` is an example of such an instruction, which adds *0x20* to its first argument. In the above example, therefore, the first parameter *0x3e* actually points at address *0x5e* in the global address space, which is named `SPH`.

Hence, in the current implementation, operands are typically displayed as integer literals, with the above exception. In order to determine whether the name of an address should be displayed instead, it is necessary to determine which operands are used as an address or part of an address expression. Such an expression could then be rendered as a text and displayed, in case it does not contain function calls or is overly complex to display. From a technical stance, the steps required for such a resolution closely resemble those described in the previous section.

We consider this approach only useful as a possible side-result of the research proposed in the previous section. Implementing such analyses for the sole purpose of improving the GUI is not beneficial because there is an easier approach to this. The simulator developer could provide some sort of formatting string, which would be evaluated at compile time. However, this would necessarily prolong the SGDL description, in the worst case by one additional line per instruction for defining the string. Therefore, this should be avoided if possible.

Interrupt Analysis

Furthermore, the analyzer may help reduce the need for explicit descriptions with regard to the interrupt system. The behavior of an interrupt may depend on several configuration registers, which must never be altered by any abstraction during simulation. In particular, this applies to Dead Variable Reduction (DVR) (cf. Sect. 8.5), which resets unused memory locations to 0. To avoid an unintentional reset of locations that may have side effects, the generated simulator needs to be equipped with a list of excluded addresses. The current approach to this is to add an SGDL `dependencies` section and therein, create entries of the form `register influences interrupt interruptname using mask bitmask`, where interrupt-

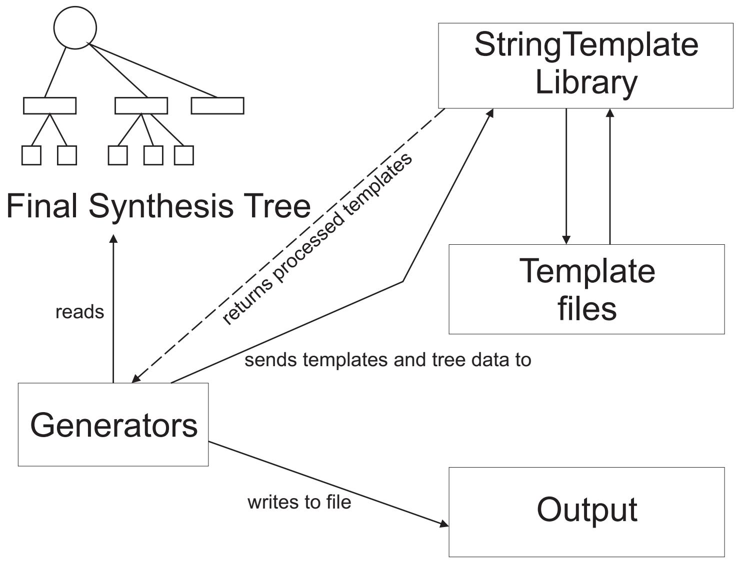

Figure 6.6: Structure of the backend for code generation in the SGDL compiler

name designates an already declared interrupt and bitmask indicates the relevant bits in the register.

For the ATmega16, the part of the dependencies list related to interrupts consists of 40 entries. Each of these corresponds to a read access of the named register in one of the interrupt code blocks. The most frequently occurring register is SREG, and the bit mask value is *0x80*, that is, the global interrupt enable flag. Obviously, to obtain the same information, SGDL-STA would need to conduct a Read Variable Analysis on each of the interrupt code blocks. At the time of this writing, it already creates the necessary CFG fragments, but no analysis is executed yet.

6.6 Compiler Backend For Code Synthesis

The backend of the SGDL compiler processes the last intermediate representation, which we refer to as *synthesis tree*, and generates Java code. The structure of the backend is depicted in Fig. 6.6. As displayed in the figure, there are four components involved: the synthesis tree, generators, the STRINGTEMPLATE library, and several template files.

Code generation is controlled by so-called *generators*. For each output file to be created, there is a specific generator. A generator requires the synthesis tree, from which it reads the information necessary for creating its output. For most generators, this is only a subset of the entire tree, e.g. the instruction-related leaves for the instruction set generators. The generator may perform one final processing step to rearrange the data, and then assigns the data to a template for the StringTemplate library.

StringTemplate [54] is a template engine for generating strings. It is part of the ANTLR [53] parser generator project. Technically, StringTemplate provides us with the equivalent of a parser generator with regard to generating output. To use it, we define templates (i.e., grammars), in which we declare terminal symbols (i.e., text that is to be generated exactly as defined in the template) and nonterminal symbols (i.e, text that can be replaced). Therefore, we can assign values to the nonterminal symbols, and eventually trigger StringTemplate to generate all possible output words. The library accepts Java objects as values, and can automatically process multi-valued attributes such as lists. StringTemplate accesses class members named in templates by means of Java Reflection [45], generating the names of accessor methods if necessary. As this also allows for navigating of data structures, ignoring the visibility concept of Java, we have decided to restrict the access of StringTemplate to the very last synthesis tree.

The templates used for StringTemplate are stored in text files. The main template in each of these files corresponds to a Java class, which may rely on several internal templates. For example, the template for creating a deterministic instruction visitor (cf. the structure of the generated simulator in Chapt. 7), is less than 100 lines of code, with the main template consisting of 75 LOC. In the case of the ATmega16 simulator, StringTemplate generates approximately 1,800 LOC of code from this template. Thus, the template approach allows for very compact representations of complex structures.

Finally, after StringTemplate has applied the attributes to the template, it returns control to the generator. The last step for the generator is then to write the output to a file, the name of which is specific for each generator. After that, the backend calls the next generator in a simple linear list of generators, where the same process takes place. Eventually, when all generators are finished, the synthesizer terminates.

6.7 Glue Code Subcompiler

After the main Sgdl compiler has finished processing an Sgdl description, the generated simulator is available within the package structure of [mc]square. In case of a newly added simulator, however, the model checker is not yet aware of its existence. Options and the graphical user interface need to be adapted accordingly.

In order to ease adding a new simulator, we have performed two steps. First, we have refactored the hardware dependent parts of [mc]square such that automati-

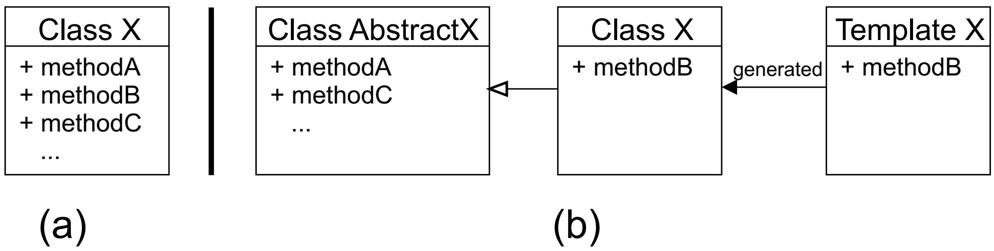

Figure 6.7: Replacement pattern for platform-dependent class X. (a) shows the starting point, with methodB being platform-dependent. (b) shows the situation after extracting the platform-independent code of X into AbstractX. The class X is now a generated class, created from a template file.

cally generated code can be used. Most of these locations involve checking variables of a few `enum` data types for the currently active microcontroller, and setting options or GUI elements depending on the value of the variable. For this purpose, we have followed a pattern: wherever we encountered platform dependency in a class X, we have extracted all non-hardware dependent code to a super class $AbstractX$. The remaining code in the original class X, which was usually very compact, became the basis for a StringTemplate template file. The procedure is illustrated in Fig. 6.7. By this procedure, we safeguarded the code against accidental manual modification, which happens frequently when using for instance the refactoring tools provided by the Eclipse IDE. At the same time, it allowed us to generate the hardware dependent parts by overwriting the now very compact subclasses (in the above example, X).

The second step we performed to assist in adding new simulators was the creation of a specific subcompiler. The input for the subcompiler is a very simple file, which provides information on which platforms are to be supported in [mc]square, in which package they are located, and whether they are handcrafted or synthetic. In the latter case, the file also contains the common prefix of all the simulator files, and also which loaders (ELF, HEX) the platform supports. Name prefixes combined with the package name allow for fully qualified referencing of the simulator classes in the generated glue code. An excerpt from such a description is shown in Listing 6.2.

Theoretically, it is not strictly necessary for the subcompiler to rely on explicitly provided information about the available simulators. The main Sgdl compiler could provide this information as well. Nevertheless, we decided to keep these two compilers separated for reasons related to software quality and to performance. The first argument is based on the fact that we intentionally designed the main compiler not to modify anything except in packages containing SGDL files. We did not desire accidental changes to code outside of the respective package, which may occur due to errors during development. Such changes, on the other hand, are obviously exactly the purpose of the subcompiler. Apart from that, our second argument, performance, relates to the fact that changes to an existing simulator are far more frequent than the addition of a new simulator. Hence, there is no need for regenerating options and GUI on each compile.

```
handwritten platform "ATmega16" {
   enum name = "ATMEGA16";
   description = "ATmega16";
}

handwritten platform "PLC" {
   enum name = "PLC";
   description = "PLC";
}

generated platform "ATmega16Synthetic" {
   enum name = "ATMEGASYNTH";
   description = "ATMega Synthesized";
   package name = "mc_square.simulator.avr.atmega16";
   class name prefix = "Atmega16";
   supported loaders = ".elf", ".hex";
}
```

Listing 6.2: Excerpt from the list of available simulators used by the subcompiler

6.8 Related Work

6.8.1 Compiler Technology

Bogosavljevic [11] created a language processor for a language called DICES in his thesis. That work was itself based on code from AVRora [81], and some of the code still remains as part of the SGDL compiler. The parts associated with AVRora are the parser and the abstract syntax tree, both of which were modified by us to accommodate for the new language elements in SGDL and requirements in the processing chain.

In most respects, the SGDL compiler abides by the general principles found in compilers. Its parser, which is generated by JavaCC [57], is based on an $LL(k)$ approach. There are multiple intermediate representations, a static analyzer, and finally, a code generation stage. Variables in the input languages are restricted to a certain scope, which we realized by means of symbol tables, also a general concept. A thorough overview of these concepts, and of possible implementations, can be found in the very comprehensive book on compilers by Aho et al. [2].

6.8.2 Static Analysis

An overview of static analysis in general is given by Nielson et al. [49], and, to a certain degree, also by Aho et al. [2].

The following overview of related work on static analysis, which is more specifically focussed on hardware descriptions, is taken partly from one of our earlier publications [27].

HOIST is a system by Regehr and Reid [60] that can derive static analyzers for embedded systems, specifically for the Atmel ATmega16 microcontroller. In this regard, their approach resembles ours, as we also create static analyzers for microcontrollers, and as we also support the Atmel ATmega16. The key difference is that they do not use a description of the hardware, but either a simulator or the actual device. For a given instruction that is executed on the microcontroller, HOIST conducts an exhaustive search over all the possible inputs, executes the instruction once for any given input, and protocols the effects on the hardware. These deduced transfer functions are then compacted into binary decision diagrams (BDDs), and eventually translated into C code. Benefits of this approach are that it can be automated almost entirely, and that it achieves a very high accuracy in instruction effects. On the downside, however, there is the exponential complexity in the number of bits in the parameters of instructions. In our approach, there is no such complexity issue, as SGDL-STA operates not on a simulator or the physical device, but on a description of it, and does not need to execute instructions to examine their effects. However, this also restricts the validity of our results, which depend on the correctness of the description, whereas the results obtained by HOIST are implicitly verified against the actual device.

Chen et al. [15] have created a retargetable static analyzer for embedded software within the scope of the MESCAL project [37]. Similar to our approach, they process a description of the architecture, which in their case is called a Mescal Architecture Description (MAD). Automatic classification of instructions for constructing the CFG is apparently also possible in their approach, and they hint at that this is possible due to some attributes present in the MAD that allow identification of, for instance, the program counter. However, no further details are provided on the ideas involved in classification. The generated analyzer is suitable for analyzing worst case execution time (WCET) issues of certain classes of programs intended to run on the hardware.

Schlickling and Pister [72] also analyze hardware descriptions, in their case VHDL code. Their system first translates the VHDL input into a sequential program, before applying well-known data flow analyses such as constant propagation analysis. These analyses are then used to prove or disprove WCET properties of the hardware. In contrast to this, we focus on the way the resource model is altered by instructions, and timing-related properties are of no relevance for our approach.

Might [46] focuses on the step from concrete to abstract semantics for a variant of lambda calculus. In their examples, they also relate their work to register machines, which, albeit a concept from theory, share some commonalities with microcontrollers. They point out the similiarities between concrete and abstract semantics, and contribute an almost algorithmic approach for lifting concrete semantics to abstract ones, which is to to be used by developers. Our work on SGDL-STA pursues a similar goal, in that we intended to derive some abstractions automatically. As this is future work for our system, however, we cannot state for certain whether such an entirely automatic approach is actually feasible in our setting.

7 Generated Simulators

Within the scope of this chapter, we detail the structure and mode of operation of generated simulators in general. Regarding the structure, we describe the underlying resource model and the instructions modifying it, as well as the organization of the interrupt subsystem. Concerning the mode of operation, we commence by explaining the process of creating successor states, followed by a discussion of necessary components such as the disassembler and binary file loaders. Finally, we describe the automatically created interfaces for monitoring simulator activities, and how to validate simulators against their respective data sheets.

7.1 Resource Model

The term *resource* in a generated simulator refers to globally available memory. Local variables declared in subroutines are not part of the resource model because they only exist in the *simulating* system. As such, they are not part of the states that have to be stored in the state space.

As pointed out in Chapter 5, we distinguish memory arrays and memory aliases. These are directly mapped to Java classes. Both *MemoryArray* and *MemoryAlias* are superclasses of a class hierarchy. Depending on specific options present in the SGDL description, it is possible that the generated code references a subclass. For instance, when the developer requests a memory array using differential storage, an instance of *DifferentialMemoryArray* is created instead of a *MemoryArray*.

7.2 Instructions

Instructions are mapped to a set of classes, as shown in the UML class diagram in Fig. 7.1. Technically, these classes can be distinguished by whether they are used for simulation or for static analysis. Those classes that are used for simulation of program execution modify the resource model, whereas the classes for static analysis operate on an abstract machine model, as described by Schlich [68]. Besides those, there exist a few shared classes.

For implementing instructions, we decided to abide by the same principle used by all the other simulators in [MC]SQUARE. Hence, instructions are implemented using the *visitor pattern*, as described by Gamma [23]. This pattern implies that

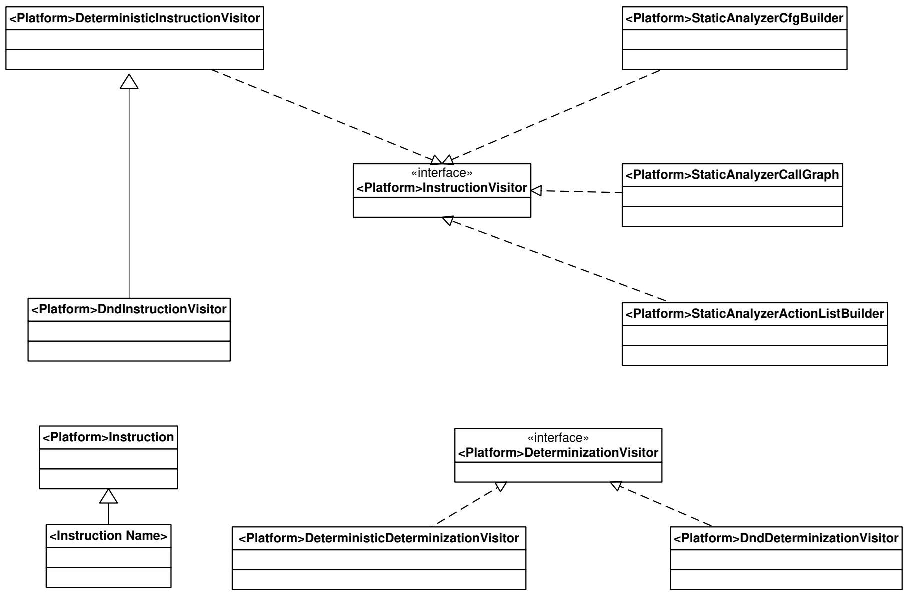

Figure 7.1: Generated classes that are related to the instruction set

each instruction in the SGDL file is represented by a class, and that behavioral aspects (i.e., the content of the `execute` and `execute DND`) sections are separated from these classes. Instead, behavior is placed into a `Visitor` class. Therefore, executing a behavior associated with an instance of an instruction consists of the following steps:

- create an instance `v` of a visitor, for instance a *DeterministicInstructionVisitor*, which implements the code for a completely deterministic simulation
- call the instruction's `accept` method and pass the visitor as an argument
- the instruction calls the overloaded `accept` method of the visitor, passing itself as a reference. Thus, the runtime environment selects the appropriate implementation of `accept` by means of the signature.
- the now called method of the visitor contains the actual behavior associated with the instruction

Using this principle allows us to store a single data structure for the instruction instead of one per simulation type. Additionally, the concept also covers the requirements of static analysis, in that the abstract semantics is just a different kind of instruction visitor operating on the same data structure as regular simulation.

In the following, we describe the classes relating to the instruction set. All of these are implicitly prefixed with the name of the architecture, hence, in an actual simulator, the name is <architecture> <name from below>:

- *Instruction*: contains the previously described instruction classes, which are implemented as inner subclasses of *Instruction*.
- Simulation-related:
  - *InstructionVisitor*: an interface used in the visitor pattern implementation for instruction execution
  - *DeterministicInstructionVisitor*: a visitor used for deterministic simulation, in which nondeterminism is resolved immediately whenever it is encountered
  - *DelayedNondeterminismInstructionVisitor*: a visitor used for simulation with the abstraction *Delayed Nondeterminism* (cf. Chapt. 8) active (meaning nondeterminism is propagated through memory)
- Instantiation-related:
  - *DeterminizationVisitor*: an interface used in the determinization (also called *instantiation*) of nondeterministic values. This is also an implementation of the visitor pattern used with the instruction class structure.
  - *DeterministicDeterminizationVisitor*: a visitor that implements *DeterminizationVisitor*, used together with *DeterministicInstructionVisitor* in deterministic simulation
  - *DndDeterminizationVisitor*: analogous to *DeterministicDeterminizationVisitor*, but for simulation using Delayed Nondeterminism
- Static analysis-related:
  - *StaticAnalyzerActionListBuilder*: abstract semantics for all instructions in the instruction set architecture. For a detailed description of the concept, refer to Sect. 8.5.2, where we describe the generation of the static analyzer in the context of an abstraction called Dead Variable Reduction. The existence of such a class is required by the static analyzer component of [MC]SQUARE.
  - *StaticAnalyzerCfgBuilder*: the static analysis framework contained in [MC]SQUARE requires a simulator to provide an implementation of a *CfgBuilder*. The latter is contained in the static analyzer package, and as that is *not* part of our contribution, we do not provide any details on it here. The builder is used to create a control flow graph (CFG) from

a program written for the platform. As the program consists of individual instructions, and each instruction becomes a node in the graph depending on the effect it has on control flow, this task is realized by an implementation of *InstructionVisitor*.

– *StaticAnalyzerCallGraph*: this class closely resembles the CFG builder, but its task is not to add regular control flow edges. Instead, it has to add call and return edges to an already existing CFG for instructions related to function calls. Thus, it is used during the transition from a control flow graph to a *call graph*. The existence of this class is also required by the static analyzer component of [MC]SQUARE, where the actual graph construction takes place.

7.3 Interrupts

The interrupt system in our generated simulators is a generalization of the systems implemented in the handcrafted simulators in [MC]SQUARE. The foremost difference is that we need to be able to handle *any* kind of interrupt priority system, which increases the complexity even for architectures with a fixed priority (e.g. Atmel AVR microcontrollers). The reason why this is relevant is detailed in Sect. 7.4, where we explain successor state creation.

The interrupt system principally consists of interrupt classes, one for each interrupt, an interrupt list, and a container class that becomes a so-called interrupt manager at runtime. Futhermore, there is a comparator for interrupt priorities. All classes of the interrupt system are contained in the manager class, which interfaces to the determinizer. The interrupt classes contain the Java code corresponding to the input from the SGDL file. Finally, the list and the comparator interact to provide a sorted list of interrupts, which allows retrieving the interrupt that currently has the highest priority and is also active.

7.4 Determinizer and Splitter

During successor state creation, the *determinizer* has to prepare the simulator state such that there is no nondeterministic choice regarding the next step. That is, it has to create a distinct state for each possible occurrence of an active interrupt. For those situations in which no interrupt occurs, the determinizer has to ensure that the next instruction can be processed. If necessary, this includes resolving nondeterminism in the input for the instruction, i.e., also creating multiple states. After the determinizer has finished, the next step the simulator has to take is determined unambiguously by the hardware state.

An important entity used during this process is a so-called *splitter*. Whenever a memory location containing nondeterministic bits has to be instantiated, the result is a set of possible valuations for this location. Each of these valuations corresponds to a distinct state of the microcontroller. This split of the formerly partially symbolic state into multiple states is the reason why these entities are called splitters. Splitters are used by the determinizer.

Determinizer and splitter are concepts that were already present in the hand-crafted simulators. In generated simulators, they are necessarily generalized. With regard to the determinizer, this means increased complexity, whereas the generalized splitters are comparatively simple. This is due to the fact that they cannot exploit platform peculiarities.

7.4.1 Mode of Operation

The determinizer is implemented as a state machine. It is created for a given microcontroller state, starts in its intial state, and, when requested to return the next possible valuation, it does so and may change its internal state. The following states exist:

- *Init Interrupts*: this is the initial state. The interrupt manager initializes all interrupts for the current machine state. This means that all interrupts are checked whether they are enabled, and in case they are, they are added to a list of currently nondeterministic interrupts. Finally, the list is sorted by the current interrupt priority, depending on the state of the hardware. The determinizer then transitions to the *Interrupts* state.

- *Interrupts*: in this state, the determinizer iterates over a list of all interrupts, which is sorted by priority.
  - If any active interrupt is found the flag of which is already posted, the determinizer can immediately proceed to the finished state. The reason is that due to the sorting, no other interrupt could possibly occur and have a higher priority. Also, as *any* occuring interrupt would alter control flow to the interrupt handler, it is also certain that no instruction could be executed instead.
  - Otherwise, if an active interrupt is found that is currently marked as nondeterministic, i.e., it could occur but does not have to, the determinizer creates two states: one in which the interrupt occurs, and one in which it does not. In the former the determinizer proceeds to the finished state, having discovered an occurring interrupt, and in the latter, it returns to Interrupts, where the search for an interrupt continues.

- *Nondeterministic Interrupts*: serves as a marker for the determinizer that the last creation of a microcontroller state yielded a situation where an interrupt marked as nondeterministic was instantiated to *has occurred*. The necessary step from here is to create the situation where it certainly does not occur, and to continue the search for the next lower-priority interrupt by returning to the *Interrupts* state.
- *Init Instructions*: in this state, it is certain that the successor states will be created by executing the current instruction. Hence, the state machine has to determine whether it is necessary to instantiate any of the arguments used by that instruction. To this end, it creates a splitter and stores it, before performing a transition to the state *Instructions*.
- *Instructions*: having reached this state, the determinizer uses the splitter created in the last state to create all possible valuations for the arguments of the instruction. Each of these is returned to the simulator as a complete microcontroller state. When the last state has been created, the determinizer enters the *Finished* state.
- *Finished*: the final state of the state machine, which is also a sink.

7.5 Loaders

Several steps are necessary for loading a binary program into [MC]SQUARE. The first of these is to read and parse the binary file created by the compiler, which may contain the initial content for multiple memories, such as Flash and EEPROM. Commonly found file formats are, for instance, the Executable and Linkable Format (ELF) [80], Intel HEX, and Motorola S-Record. Simulators in [MC]SQUARE can access these file formats by means of so-called *loaders*, which can be parameterized. A loader consists of a parser for the corresponding file content and a component that writes the discovered memory contents to their designated memories. Eventually, when the program memory has been written, the loader may also invoke the disassembler (see the next section).

We decided to reorganize the already existing ELF and HEX loaders such that they can be reused for multiple platforms. This step included introducing parameters, such as program memory word sizes, endianness, maximal addresses for data, and platform identifiers that are checked during loading to ensure a file is intended for the active platform. Such information is encoded in specialized classes, which are instantiated and passed to the loaders when loading a file. The concept has proven to be applicable to both handcrafted and synthetic simulators. In case of generated simulators, the necessary file is generated from the loader section in the

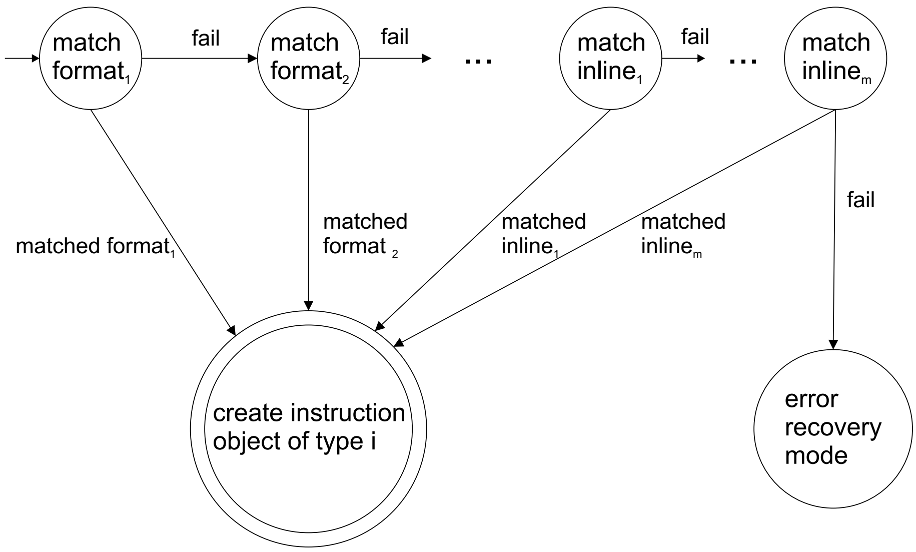

Figure 7.2: Mode of operation of the disassembler

SGDL file. Hence, we obtained a library of reusable loaders, from which developers can select as many as desired.

7.6 Disassembler

The task of the disassembler is to read the memory of the simulated device at a specified position and to create instruction objects from the memory contents. For this purpose, it contains patterns of the binary encodings of all instructions in the instruction set.

Fig. 7.2 illustrates the disassembler's mode of operation for a single instruction. First of all, the disassembler tries to match a prefix of the binary stream to one of the formats from the `format` section in the SGDL file. If this succeeds for any format, disassembling continues for that format, trying to match the prefix to a specific instruction. In case this succeeds, the position in the stream is increased by the size of the instruction, and the process starts anew for the next instruction. Otherwise, the disassembler attempts a match against the next format. In case none of the formats matches the prefix, the disassembler tries to find a match for instructions that do not use a format (i.e., they declare their encoding inline). Finally, if this also does not yield a valid instruction, the disassembler enters error recovery mode.

In error recovery mode, the disassembler can try two different strategies. If the developer has provided a maximum length for instructions in the SGDL file, then the disassembler can try all addresses following the current one up to current plus the maximum size for a valid instruction. In that case, the current address contains an invalid instruction, for which an instance of *InvalidInstruction* is inserted into the disassembled program. Otherwise, that is, without a value for the maximum instruction size, the disassembler cannot continue, as we cannot distinguish between opcode and data any more. In that case, we insert *InvalidInstruction* objects for all remaining addresses.

It is worth mentioning that the current implementation of the disassembler tries to match fixed bit patterns against the binary input stream. Thus, the prefix of the input is read several times. This does not necessarily pose a problem in case of short programs, especially if they are disassembled only once upon loading. For large programs, or when disassembling becomes necessary *during* simulation (on-the-fly disassembling), however, this can severely slow down simulation. Existing algorithms for string pattern matching, for instance Knuth-Morris-Pratt [2, 38], or techniques based on so-called *tries* (e.g. suffix trees) [2], could possibly be used for enhancing this.

7.7 Data Types

All data types are internally mapped to the Java type `int`, which is 32 bits wide. Compared to an entirely symbolic number representation, which would provide arbitrary bit widths, this approach has both advantages and disadvantages. The obvious disadvantage is that in the current form, SGDL cannot be used to model microcontrollers with more than 32 bits register width. Even modeling a 32 bit machine could be problematic because of the internal representation of integers in the Java Virtual Machine (JVM), which uses two's complement representation. An advantage of this approach over a symbolic one, however, is the improved performance. It is not necessary to perform computations on symbols (i.e., in software), but the simulating computer can do these natively, that is, in hardware. Furthermore, model-checking software with [MC]SQUARE for any architecture of more than 16 bits width seems currently infeasible due to the explicit-state approach. Hence, we conclude that using `int` is no real restriction at the moment, and that the performance advantage justifies this approach. Should there be need to change this, we assume that moving to 64 bit, by using the Java type `long` instead, would be straightforward, and a symbolic approach seems possible as well.

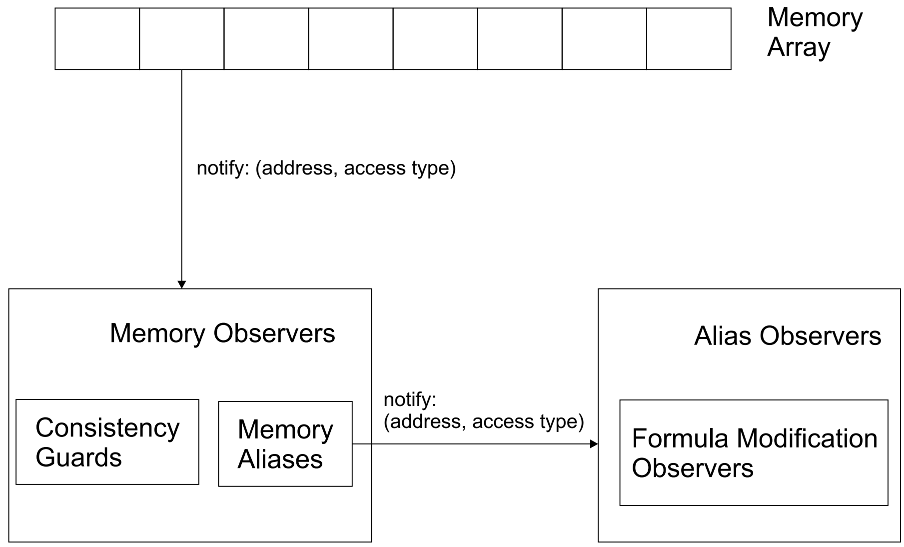

Figure 7.3: Double observer pattern for memories

7.8 Observer Interface

Resources in generated simulators, as described in Sect. 7.1, relate to global memories, represented by memory arrays and memory aliases. Monitoring accesses to such resources may be necessary for various reasons. We required such a possibility, for instance, for implementing an abstraction called *On-the-Fly Path Reduction*, for sanity and consistency checks of memory contents, and also for a variant of an abstraction called *Delayed Nondeterminism*. Details on these techniques, including a motivation for each, are provided in Chapt. 8.

An obvious solution to the problem of monitoring memories is the *observer pattern* (cf. [23]). However, in our design, the memory system consists of two different levels, namely the actual storage level realized by memory arrays, and the access level, represented by the memory aliases. Consequently, there are also two different starting points for monitoring. We decided for a double observer pattern implementation, as illustrated in Fig. 7.3. Both the memory aliases and the memory arrays can be observed, and all memory aliases are also observers for memory arrays. This design decision was motivated by several reasons:

1. Completeness: accessing a memory cell should alert *all* observers listening to it. It must not be possible that an access via one alias is properly registered by its observers, but another alias for the same cell is unaware of the access.

2. Flexibility: depending on the purpose behind monitoring, it should be possible to attach an observer either to the lowest level (i.e., memory arrays), or to a higher level (i.e., an alias).

3. Loose coupling: memory accesses in SGDL code blocks are always routed through aliases. Hence, in case an alias is given, no knowledge about the actual storage is required. Thus, it should be possible to attach an observer to the alias.

Property (1) is guaranteed in our design because any access to a memory cell conducted via an alias will result in the underlying array to broadcast an access message to its observers, including all aliases registered with it. The necessary registration is performed automatically on initialization of the simulator. Properties (2) and (3) are fulfilled because aliases themselves can be observed.

The two interfaces implemented by the observers, `MemoryAliasAccessObserver` and `MemoryArrayAccessObserver`, distinguish between read and write accesses, and the observables respect the difference as well. Hence, it is possible for observers to restrict their attention to read or to write events, if required.

7.9 Validation

Generated simulators may contain errors introduced by the developer, errors in the synthesis system, or both. If an instruction is not implemented correctly with regard to its specification in the data sheet, the generated simulator will create incorrect state spaces, which in turn may result in incorrect model checking results. Hence, it is necessary to increase the confidence in the correctness of the generated simulator. For this purpose, every instruction and every subroutine in the SGDL input should be tested adequately. Other components that are provided by the synthesis system, such as memory storage classes and operands, are independent of specific simulators, and can therefore be tested independently. Within this section, we take it for granted that these components are properly tested and operate as supposed to. Hence, the focus is on how and to what extent the synthesis system can assist in validating the generated code against the device specification.

In order to facilitate validation, the synthesizer not only generates the simulator, but also a class containing JUnit [22] tests for the instruction set. The test class covers the following:

- Creation of the simulator. Each test run is preceded by the initialization of a new hardware, options, and deterministic instruction visitor. These components are then available in the actual tests.

- Subroutines. The generator creates test stubs for each of the subroutines in the SGDL file. These stubs are already marked for test execution by means of the Junit `@Test` annotation. It is up to the developer to design test cases for the method under test, and to implement the actual test code.

- Instruction encodings. For each instruction present in the SGDL input, an empty encoding pattern in the form of a Java `byte` array is created. These encodings are then used in the test for that instruction. A generated test checks whether the length of the array matches the length of the instruction that is derived from the SGDL input.

- Instruction disassembling. The test for the generated disassembler invokes the latter with the aforementioned encoding in the array. In case the disassembler operates correctly, it should return a single non-null instruction of the respective class. Otherwise, the test fails.

- Instruction execution. Following successful disassembling, the generated test attempts to execute the instruction. As we assume that there might be errors anywhere in SGDL file and synthesis system, we do not generate any code for automatic checking of any postconditions. Hence, the developer has to add the necessary code for inspecting the machine state reached after the execution.

An important aspect of these tests is that the mentioned instruction patterns are empty, thus forcing the developer to inspect the device's data sheet or instruction set manual for finding valid opcode and parameter encodings. While it would be possible to derive valid patterns for the instructions from the descriptions in the SGDL file, we have deliberately abstained from doing so due to principles from software testing theory:

- Testing requires some means for determining the correct result for an input. Typically, this sort of reference is called a *test oracle* (e.g. [40]).

- In case of regression tests, for instance, an older version of the system serves as the oracle, and in model based testing, a model of the system [40, 74].

Consequently, generating both the simulator and the pattern from the *same* description could result in the test not discovering any errors even for a completely incorrect implementation. Thus, we decided to automate the test case generation as far as seemed reasonable. In order to allow for the necessary manual extension of the tests, the SGDL compiler will not overwrite an already existing test class.

Listing 7.1 shows an excerpt from the automatically generated JUnit test class for our SGDL implementation of the Intel MCS-51 microcontroller. Positions to be manually enhanced by the developer are marked by generated `TODO` markers.

```
//TODO: fill in the correct binary representation of the instructions here. Select variable values as you like!
      protected static final byte[] Mcs51_ACALL_ENCODING = {0x51, 0x2c};
      protected static final byte[] Mcs51_ADD_ACC_REG_ENCODING = {0x28};
      protected static final byte[] Mcs51_ADD_ACC_DIRECT_ENCODING = {0x25, 0};

      private Mcs51Hardware hardware;
    private OptionsValuesWritable options;
   private Mcs51DeterministicInstructionVisitor visitor;

    @Before
   public void setUp(){
      OptionsApi.initialize();
      options = OptionsValues.getWritable();
      hardware = new Mcs51Hardware(options);
      visitor = new Mcs51DeterministicInstructionVisitor(hardware);
   }

   @After
   public void tearDown(){
      options = null;
      hardware = null;
      visitor = null;
   }

   /**
     * Test code for the subroutine
     *     <code>Operand getRegInActiveBank(Operand reg)</code>
     */
     @Test
```

```
    public void testSubroutine_getRegInActiveBank() {
    }

    /**
    * Test code for the instruction
    *     <code>Mcs51Instruction.acall(IMM11 target))</code>
    */
    @SuppressWarnings("null")
    @Test
    public void testMcs51_ACALL(){
     //Initialize the program memory using the values
         provided in the constants section
     assertEquals("tstTestSetup_ACALL", 2,
        Mcs51_ACALL_ENCODING.length);
       byte[] program = new byte[2];
       for (int i = 0; i < Mcs51_ACALL_ENCODING.length; i
          ++){
        program[i] = Mcs51_ACALL_ENCODING[i];
       }

     //Test whether the disassembler properly creates an
         Mcs51Instruction.ACALL from the byte[]
     Mcs51Instruction[] instructions = Mcs51Disassembler.
         disassemble(program, 0, hardware);
     assertTrue("tstDisassACALL1", instructions != null);
     assertTrue("tstDisassACALL2", instructions.length >=
         1);
     assertTrue("tstDisassACALL3", instructions[0] != null)
         ;
     try{
         @SuppressWarnings("unused")
         Mcs51Instruction.ACALL castAttempt = (
             Mcs51Instruction.ACALL)instructions[0];
     }
     catch(ClassCastException e){
         fail("Wrong class created by disassembler! Expected
             : Mcs51Instruction.ACALL, but was: " +
             instructions[0].getClass());
     }

     //TODO: add code for testing the results of
```

```
        instruction execution here!
    instructions[0].accept(visitor);
}
```

Listing 7.1: Auto-generated test for the Intel MCS-51 instruction set

The only manual modification in Listing 7.1 involves the selection of the array contents at the beginning. All other code has been generated. Hence, these tests usually provide a basic coverage of instruction behavior, and require little effort. However, without exploiting knowledge about the instructions it is not possible to automatically cover corner cases, and therefore it is necessary to manually implement further tests. We detail possible approaches in this direction in Sect. 9.3.

8 Abstraction Techniques

In this chapter, we present the abstractions that are available in the synthetic simulators. Abstractions are used to counter the state explosion problem, which is a general problem in model checking. In the first section, we provide detailed information on this problem, and how this relates to the state space generator. All following sections then focus on specific approaches.

Most of the abstractions presented in this chapter were already described in some of our earlier publications, or were originally introduced by other authors. A thorough description of such previous contributions by others, and how they relate to our designs, is given in Sect. 8.8. Concerning our own previous work, Lazy Stack Evaluation for synthetic simulators was presented in [28], together with an early version of On-the-Fly Path Reduction, which was then still named *Dynamic Path Reduction*. A more advanced version of On-the-Fly Path Reduction was presented in [10], though the simulator used in that publication was not a generated one. Still, the advanced version was first implemented in one of our synthetic simulators, and later ported to the handcrafted simulator used in the publication. Regarding further abstractions, we contributed a publication to a workshop [27] in which we pointed out the necessary steps for automating the generation of a static analyzer and of a semantics for delayed nondeterminism.

8.1 Motivation

Model checking in general is a means for verification that requires large amounts of memory. The memory is needed for storing the state space, which is traversed in order to check for property violations. In case the system employed for the verification process runs out of memory, model checking may have to terminate prematurely. In that case, depending on the actual algorithm used, the usefulness of the results obtained may be very limited: they may be valid only for the part of the system checked so far, or there may be no result at all yet. This problem is particularly acute for software model checking, as the number of states to be stored correlates with the number of steps the system may take, and a step in a program is typically a single instruction. On the assembly level, this implies a large number of states to be stored even for small programs, as most instructions result in rather slight modifications of the machine state.

Microcontrollers usually have small memory sizes, ranging from less than hundred bytes to more than ten MBytes for very large devices. Typically, though, microcontrollers have memories much smaller than one MB. The devices we consider are the Atmel ATmega16 and the Intel MCS-51. Omitting the separate program memory and EEPROM, the ATmega16 features only one kByte of SRAM plus 96 bytes of registers, resulting in an overall random-access memory size of 1120 bytes. At a glance, this might seem so small as to render storage issues irrelevant. However, this is not the case. Each of the 1120 bytes consists of eight bits, hence there are 8960 bits, each of which may be either 0 or 1. Therefore, the number of states the microcontroller might be in is

$$2^{8960} \approx 1.69 \cdot 10^{2697}$$

Therefore, regardless of the memory technology available, it is simply infeasible to store all possible states, or even to only compute them. The corresponding value for the MCS-51 with its 128 bytes of RAM would be $1.8 \cdot 10^{308}$ states, which is why storing all states would be equally infeasible even for this device with its rather small amount of memory.

It is possible to remedy this problem by several approaches. Clarke et al. [16] for instance distinguish between symbolic algorithms, symmetry reduction, assume guarantee reasoning, , induction, and abstraction. We illustrate some of these in the section on related work (cf. Sect. 8.8). Within the scope of this chapter, we focus on reductions of the number of states that are based on abstraction. The general idea of abstraction (as defined by both Clarke and Baier and Katoen [9]) is that instead of verifying properties of the actual system, we first map the system to a smaller, abstract system. It is necessary that the mapping preserves the validity of properties, such that a formula proven to be valid for the abstract system must also hold true for the actual system. In that case, it is therefore not necessary to create the actual system at all.

The aforementioned preservation of properties is crucial for any abstraction. Similar to the problem of creating an appropriate model of a system, it is possible to omit important aspects of a system when abstracting. Hence, it may occur that checking a given specification against the abstracted system results in a false negative. That is, the concrete system is rejected because the property does not hold on the abstract system, even though it would hold true on the concrete one. Analogously, a false positive would be the acceptance of a faulty concrete system because the abstract system satisfies a property, for instance because it may exhibit more behavior than the concrete one due to simplification. Hence, designing abstractions requires special care, and in some cases it is necessary to re-check a system without an abstraction to ensure whether a discovered issue is actually present in the concrete system.

8.2 Language Elements Related to Abstraction

In this section, we provide a brief overview of language elements that were introduced into SGDL only for abstraction purposes. These are a distinguishing feature of the language when compared to other hardware description languages. Details on the syntax of these language elements are given in Chapter 5, where the language as such is described. Their semantics is covered in the context of specific abstraction techniques, which are described in later sections of this chapter.

- Static Behavior Section
  - Set of read locations
  - Set of written locations
  - Set of possible successors
- Dependency Section
  - Register to register dependencies
  - Register to interrupt dependencies
- Multiple `execute` sections per instruction (e.g. `execute "DND"`)
- Instantiation sets
  - For deterministic simulation: `instantiate`
  - For simulation with delayed nondeterminism: `dnd instantiate`
- Nondeterminism mask in all global memories
- Possibility for differential storage of memories to improve memory footprint of states in the state space
- Stack and stack pointer declaration

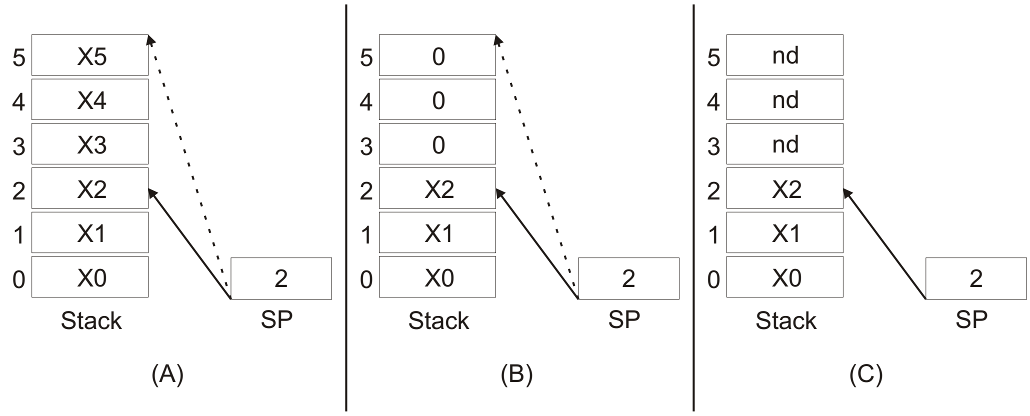

Figure 8.1: Principle of Lazy Stack Evaluation: (A) evolution of the stack without reset, (B) reset to 0, (C) reset to nondeterministic

8.3 Lazy Stack Evaluation

Lazy Stack Evaluation (LSE) [1] (see also [68]) is a very powerful abstraction that preserves the expressiveness of the abstracted system. In the handcrafted simulators, it has proven itself to be an invaluable abstraction, without which most programs would not be manageable for the model checker, and which is therefore active by default.

The idea behind lazy stack evaluation is shown in Fig. 8.1. It is based on two premises:

- A program generated by a compiler will only access the contents of the frame of a function while the frame of that function is still stored on the stack. That is, the values of local variables and the return address become irrelevant when exiting a function.

- The stack pointer indicates the actual size of the stack, and is only modified by means of `push` and `pop` operations.

Whenever an element is taken from the stack by a `pop` operation, reducing the size of the stack, it is copied to some target location. However, the originating memory cell still contains the value. Due to the assumptions, though, it is certain that it will never be read again before it is overwritten. Hence, the stack pointer register (or registers, in case the platform provides more than one) also points to the last element that actually needs to be stored. This observation is exploited by LSE.

[1]Bibliographic note: this section is an improved and extended version of our description of LSE in one of our previous publications [28]

By resetting the elements beyond the top to a common reset value, it allows states that differ only in those irrelevant memory locations to be merged. In the example in Fig. 8.1, this is depicted for a previous value of the stack pointer (SP) of 5. By three `pop` operations, the value of SP has been reduced to 2, allowing for the values X3, X4 and X5 to be discarded.

It is possible to guarantee a valid over-approximation even in the rare case the program should read a value beyond the address pointed to by SP. Such a read access may be caused for instance by a dangling pointer. To safeguard against this, locations are not only reset on `pop`, but also marked nondeterministic. Vice versa, pushing a (deterministic) value on the stack will render a location deterministic again. Thus, should the program ever read an address beyond SP, which used to be part of the stack, the content of that address will be instantiated, generating all possible values for that address. Hence, the set of values generated then will also include the original value that was deliberately dropped by the reset step of LSE.

Obviously, LSE can have little impact if the program scarcely modifies the stack pointer. For instance, among the programs introduced in the next chapter, there is a program called *Experimental Plant*, which was written for the ATmega16 microcontroller. *Experimental Plant* contains only two active interrupts and no function calls except an initialization method that is called only once. LSE is ineffective on this program. The opposite case is given when there are many function calls joined with a large number of simultaneously active interrupts. Then, it is possible for an interrupt to occur at any point during function execution, especially while entering or leaving the function. This, combined with the different orderings in which the interrupts may occur, can result in the stack containing many different combinations of remaining return values. Under such circumstances, LSE can reduce up to 98% of the state space. The effect becomes visible for the test program called *Window Lift*, which is also introduced in the next chapter. The differences between the *None* and the *LSE* settings in Tables 10.4 and 10.5 are due to the application of LSE.

When activated, LSE invariably marks the former top of the stack nondeterministic, resulting in a side effect: the memory cells marked nondeterministic describe the maximum size the stack has reached in the past. Even though the logics we use do not allow specifying statements about the content of the NDM, the information is nevertheless stored. Hence, states can differ in these locations, and it is desirable to drop this undesired information. This is possible by initializing the entire address space as nondeterministic when creating the simulator, which results in `pop` operations returning the NDM to its previous (nondeterministic) state instead of adding new information. Special care is needed, though, as such an initialization creates nondeterministic values in internal memory cells that would never become nondeterministic in certain types of simulation. Hence, even in deterministic simulation, in which instructions operate on deterministic data only, read accesses must instantiate, and write accesses must be accompanied by modifications of the NDM. The latter is especially important, as otherwise it is possible for memory cells to contain non-zero values while being nondeterministic at the same time, thus allowing for state explosion. However, adding the NDM modifications explicitly to the `execute` sections can interfere with certain abstractions, apart from the additional manual effort. In Sect. 8.7, we describe techniques that can automatically correct the NDM if necessary, preserving the flexibility of the code in the `execute` section.

Lazy stack evaluation therefore deals with a problem that is not present in theory, but occurs in real hardware. A stack on paper would never contain any values that have been removed by a `pop`. In contrast to this, a processor-based architecture only needs to return the value at the top of the stack and modify the stack pointer (either grow or shrink it), but there is no need for the processor to reset any values in memory. Hence, the value at the former top of the stack remains in memory. LSE can therefore only preserve a simulation relation between the abstracted and the concrete state space: the state space after the reset may exhibit more behavior than the concrete one.

In case the SGDL description of a platform routes all accesses to the stack over a `push` and a `pop` subroutine, LSE can be implemented with minimal effort. In the case of the ATmega16 simulator, the following modifications were necessary:

- in `pop`:
  - store the value of the stack, which is about to be returned, in a temporary variable
  - reset the value by writing a 0 to the SRAM location pointed to by SP
  - label all bits of the location as nondeterministic by writing *0xff* to its nondeterminism mask
  - return the value
- in `push`: reset the nd mask of the location where the new element is placed by writing *0x00* to it, thus marking it as deterministic

Thus, the overall effort amounts to three new lines of code (reset value, set nd mask, and reset nd mask). We have investigated the possibility of automating this, and concluded that it is possible by providing generic and already adapted versions of `push` and `pop`. Simulator developers would then have to use these methods for implementing stacks instead of implementing the associated semantics themselves. However, the generic methods need to consider platform peculiarities such as endianness, word size, the direction in which the stack grows, whether SP points to the last occupied or the next free position, and the possibility of multiple stacks. Hence, they would not necessarily relieve the developer due to their own complex handling.

8.4 On-the-Fly Path Reduction

Bibliographic note. The contents of this section are an improved and extended version of our description of *Dynamic Path Reduction* in one of our previous publications [28]. Dynamic Path Reduction itself was originally implemented for the handcrafted simulators in [MC]SQUARE by other members of the team. Our contribution is the elimination of remaining hardware dependencies in the technique, such that it became possible for us to generate the necessary code directly, that is, without the developer having to provide any additional information. Additionally, we experimented with the approach and improved it. Some of these improvements were later ported back into the handcrafted ATmega16 simulator and used for a publication in which we participated [10].

8.4.1 Concept of and Approaches to Path Reduction

The idea behind *path reduction* is to reduce long chains of states with only one successor each into a single step from the first state in the chain to the last. States in between are not stored. Apart from saving memory, this also has the advantage that the simulator state does not have to be stored after each step, which is very time-consuming. Previous examinations have shown that saving and restoring states consumes most of the time in model checking with [MC]SQUARE, i.e., most time is spent on moving arrays in memory, whereas the effort for model checking is negligible. A possible disadvantage of path reduction, however, which can occur depending on the program and the formula, is that states that have to be revisited during model checking may have to be recreated (when they are not stored). Hence, the typically much smaller state space size does not imply that the time required for model checking also shrinks. Instead, the time effort can even increase. Several conditions must be met for allowing states to be compacted by path reduction:

- no split-up due to instantiation of a nondeterministic value
- no active interrupts
- no change of any memory location relevant for the CTL formula
- no state occurs twice in the chain, i.e. no loop in the state space

Static path reduction [71, 86] uses a static analysis to derive the information about instantiation and active interrupts. The analysis is conducted prior to simulation. When in doubt about whether the condition will be fulfilled at a specific location, the information provided by the static analysis has to be that reduction is not possible. *On-the-Fly Path Reduction (OTF-PR)* [10] does not require a static analysis, but simply checks for each created state during simulation whether the above conditions are still satisfied. This results in an additional effort at runtime, but is more accurate, as there can be no doubt about whether it is safe to add a state to a chain. Thus, OTF-PR is more effective than static path reduction, and can be implemented with far less effort.

From a theoretical point of view, there is one situation in which static path reduction could perform better than OTF-PR. This would be the case for an extremely long chain. Due to the necessity to prevent loops in the state space, OTF-PR has to store some information about already seen states, which would be, in the most memory-efficient implementation, a hash value. Hence, it is possible that the associated data structure grows beyond memory limitations, causing model checking to abort. Opposed to this, the static approach would determine the chain ranges beforehand, and not allocate any memory while creating the chain. However, this scenario is not likely to occur. It corresponds to a program in which the microcontroller only computes and does not react to events from the environment, i.e., there must be no input operations, no interrupts, and deactivated internal peripherals. Moreover, it must not reach a previously reached internal state, as this would also terminate the chain.

Hence, we conclude that for realistic programs not tailored to break the principle, OTF-PR is always a better choice than static path reduction. The results from [10] underline this conclusion. Therefore, we have decided against implementing the static approach in synthetic simulators.

8.4.2 Condition Checking in On-the-Fly Path Reduction

As mentioned in the last subsection, there are four conditions that need to be checked by an implementation of OTF-PR. The checking for only a single successor is related to the first two conditions in the above list, that is, there must be no instantiation and no possible interrupt. Ensuring that this condition is met is trivial, as we only need to verify, after creating the successors of a state, whether there is only a single successor. For the third condition, which turned out to be more complicated, we investigated the impact of two different approaches, described in detail below. Loop detection in the state space (fourth condition), is again a simple task, as it is not hardware-dependent at all. In our implementation, we decided for a loop detection based on hash collision checking.

The only obstacle we encountered was related to the third condition, i.e., checking whether the step modifies any memory location relevant for the CTL formula. We decided to obtain this information by means of a dynamic analysis, instead of forcing the SGDL developer to provide a list of written locations. Fig. 8.2 illustrates our solution for this problem. The idea is to monitor write accesses to the memories during successor state creation. If an access modifies a formula-relevant location, then the successor state which was just created cannot be added to the current

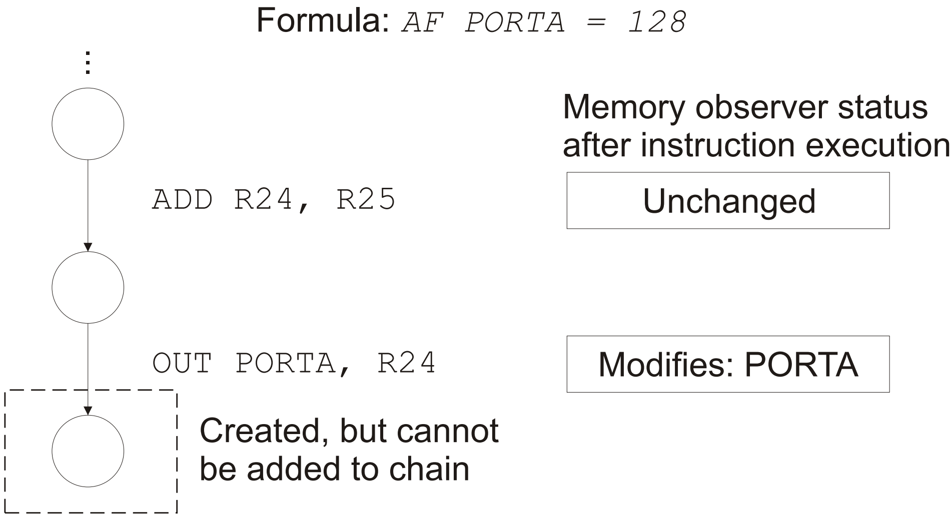

Figure 8.2: Monitoring of memory accesses in OTF Path Reduction

chain. Instead, its predecessor, to which we need to keep a reference, is the final state in the chain. This approach causes the synthetic simulators to always perform, for each chain, one step more than necessary to discover formula modification. However, the benefit of it is that it eliminates any need for knowing the effects of instruction execution in advance, thus facilitating implementation.

A refinement of the technique is based on the observation that not every modification of a formula-relevant location has an effect on the validity of the formula. For the validity to change, at least one of the atomic propostions has to toggle its truth value, e.g. from *true* to *false*. Atomics like $x \bowtie y$, where x, y are either names of registers, memory addresses, or literals, $\bowtie \in \{=, \leq, \leq, \geq, \geq\}$, are either true or false for *equivalence classes* of values. Hence, writing a new value to a location that is in the same equivalence class as the previous value will not toggle the truth value of the atomic. In case this holds true for all atomics of the formula, the truth value of the latter will also not change, implying the new state is equivalent to its predecessor with regard to formula validity. We can exploit this by requiring that a chain must be terminated not already when a formula-relevant location is written, but only when this has an impact on formula validity. In some of our case studies, this modification has allowed for an additional 30% reduction of the number of states stored in the state space.

8.5 Dead Variable Reduction

At some point during simulation, it may occur that values are no longer needed neither for the computation, nor for verifying the formula. We call the memory locations containing such values *dead variables*. A variable x becomes irrelevant for the computation at a program counter l iff, for all possible continuations starting at l, x is never read again.

Let D_l be the set of dead variables at l. By definition, modifying the value of any $x \in D_l$ does not impact the outcome of the simulation, and also does not affect the truth value of the formula. We can exploit this observation for reducing the number of states that have to be stored, by means of merging states. Let s_1, s_2 be states, representing the microcontroller at program counter l, and let $s_1 \neq s_2$. If s_1, s_2 differ only at memory locations $x_i \in D_l, 1 \leq i \leq n, n \in \mathbb{N}$, we can set $x_i := 0 \, \forall i \in \{1, \ldots n\}$, to obtain states x'_1, x'_2. Hence, $x'_1 = x'_2$, meaning only one of them has to be stored.

This abstraction is known in the literature under the term *Dead Variable Reduction*. Applying it to a program necessitates the presence of information about which variables are dead at a given program location. For implementing it, it suffices to know the location at which a variable *dies*, i.e., changes its status from *needed* to *no longer needed*.

8.5.1 Static Analysis

In order to obtain the sets of dying variables, we conduct a static analysis of the microcontroller program. The procedure is exactly the same as described by Schlich [68], with the only difference being that in our setting, the simulators are synthetic ones, and that the static analyzer as such is also generated instead of handcrafted. Hence, we just briefly summarize the intent for using a static analysis:

- Future behavior. For a given program location l and memory locations $x_i, \ldots x_n$, we need to decide whether $x_i \in D_l$. This problem is equal to deciding whether there will be a next read access to x_i, i.e., requires deciding the reachability problem for Turing machines. This problem is undecidable, as being able to decide whether a certain configuration is reachable could be used to decide the halting problem. The latter is undecidable, as shown for instance by Hromkovic [35]. We show a very simple proof in Fig. 8.3, which is similar to a proof used by Nielson et al. [49] to show that the constantness property for variables is undecidable.

  Therefore, it is not possible to decide this problem, but is is possible to obtain a safe under-approximation of reducible variables by means of a static analysis.

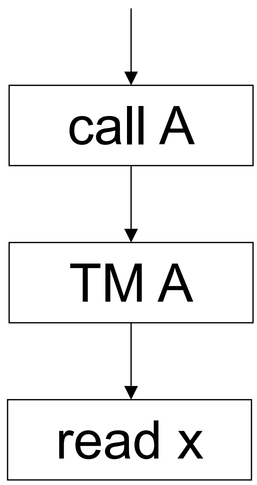

Figure 8.3: Proof that it is not possible to determine, at the call instruction, whether the variable x is still alive, due to undecidability. The problem is that reaching the read access for variable x depends on whether the Turing machine A terminates.

The under-approximation guarantees that no variable that is in fact alive is reset.

- Scalability. Static analysis scales well with the size of the program, and as such, poses only negligible overhead when run before the state space building starts. During the latter, there is therefore no overhead for checking any additional conditions on variable liveness.

- Termination. Static analyses obtain information about a program Π without actually executing it. Also, instead of aiming at exact information, which is in most cases not computable due to undecidability issues, static analysis usually only yields *approximations* of actual program behavior. While this requires a thorough design of analyses, depending on whether the analysis must return an over- or an under-approximation of the desired property, it also has advantages. The most important of these is *termination*, which can be guaranteed for the analysis algorithm.

The most often used form of static analysis consists of a translation of the input program Π into a control flow graph (CFG), either of the *Single Instruction* or of the *Basic Block* type. Depending on the type, either single instructions or non-branching blocks of instructions become nodes in the graph, which are interconnected by edges iff control flow can pass from one instruction to its successor in Π.

Analyses that are conducive to conducting DVR are *Live Variable Analysis (LVA)* and *Reaching Definitions Analysis (RDA)*. These are standard textbook analyses, described for instance in [49]. Primarily, DVR depends on the results of LVA, but its accuracy can be increased by an additional RDA. Other analyses may be helpful for increasing the accuracy, and thus, allow for a larger numbers of variables to be recognized as dead.

8.5.2 Generating the Static Analyzer

Static analysis for microcontroller code in [MC]SQUARE is provided to simulator developers in the form of a framework. This framework is not part of our contribution, but was created by other team members (cf. [41, 68, 71]). It helps to construct a platform-specific analyzer, which can conduct the actual analyses, if provided with some platform-specific information. The foremost of these describe the relation between instructions and their succeeding instructions, which is needed for constructing a control flow graph for a given microcontroller program. Furthermore, for conducting LVA and RDA, the framework requires a list of read and written locations, respectively.

The following list describes the generated classes related to the static analyzer:

- *StaticAnalyzer*: conducts the analyses and provides an interface to the results.
- *StaticAnalyzerFactory*: creates objects used in static analysis, such as call graphs, dependency maps, and builders for the various available analyses.
- *StaticAnalyzedProgram*: a container for microcontroller programs, storing information derived from the input program by means of static analysis.
- *CfgBuilder*: creates a control flow graph for a given microcontroller program.
- *CallGraph*: provides a call graph of the program to be analyzed. To this end, this class implements the visitor pattern for the platform's instruction set, with the semantic of all non-call instructions being empty, and call instructions adding *call edges*.
- *ActionListBuilder*: contains the abstract semantics of all instructions in the instruction set architecture. The static analyzer in [MC]SQUARE abstracts from concrete instruction effects in that only reads and writes need to be denoted, with a further distinction of entire addresses, bits, and interval accesses. We adopted this concept for the synthetic simulators, with most of the information for this class being derived from the SGDL static behavior sections (SSBS).

- *DependencyMap*: stores information about possible side effects of read and write accesses on other registers and on interrupts (cf. Sect. 5.2.5).
- *Constants*: contains constants used by the generated part of the static analyzer.
- *LvaBuilder*: this is the main class for performing *Live Variable Analysis* in generated simulators.
- *LvaLatticeElement*: a data structure used for storing elements of the lattice used in the LVA.
- *RdaBuilder*: analogous to *LvaBuilder*, but for *Reaching Definitions Analysis.*
- *RdaLatticeElement*: analogous to LvaLatticeElement.
- *Resetter*: computes and sets, for each instruction in the program, the set of dying variables, which is used *after* the analysis has terminated for actually performing the reset step of DVR.

The information for generating these classes originates from several different sources in the SGDL file, and some of it can also be derived automatically. First of all, the read and write effects of instructions are denoted explicitly in the SSBS of each instruction (cf. Sect. 5.3.3). This section is also supposed to provide a type for the instruction, and, in case that type deviates from *add* (i.e., the instruction jumps to some not immediately following succeeding instruction), provide a means for computing that successor. By this information, it is possible to create the control flow graph, hence it is used for generating the *CfgBuilder*. Finally, dependencies are listed in the SGDL file in the element of the same name.

As described in Sect. 6.5, some of these pieces of information need not be provided explicitly in the SGDL file. Instead, a static analysis of the SGDL file itself, focussing on instructions, subroutines, and code blocks of interrupts, can yield the necessary information. At the time of this writing, the SGDL static analyzer, SGDL-STA, can successfully determine the type of all instructions for the Intel MCS-51 and Atmel AVR instruction sets, thus eliminating the need for that part of the SSBS.

8.5.3 Application in State Space Building

If an operative static analyzer is available for a platform, [MC]SQUARE can use it to analyze the microcontroller program. The analysis is conducted before the actual model checking is started, i.e., before state space building begins. As described in the previous section, the resetter annotates all instructions in the program with the set of dying variables. Therefore, applying the actual reduction during simulation consists of iterating over the set of the current instruction, and resetting the locations listed therein to zero.

8.6 Delayed Nondeterminism

State space building for microcontroller software requires a means for handling nondeterministic data. Nondeterminism arises, for instance, from I/O ports, timers/counters, and interrupts. As described in Sect. 2.6, when reading input from an I/O port, we have to assume that every pin configured as an input pin could have a logical value of either 0 or 1, thus requiring the creation of successor states for both cases. For n input pins, this results in 2^n different successor states for a state in which a read command is executed. We also refer to this split-up as the *instantiation* of nondeterministic locations.

Hence, there is an exponential blowup of the number of successors (and therefore, paths) occurring immediately when accessing a memory location that is at least partially nondeterministic. We refer to this kind of simulation as *deterministic simulation* because the only instructions that could encounter a nondeterministic value are those able to read I/O and timer/counter registers. For all other instructions, it is guaranteed that they will never have to operate on nondeterministic data.

Noll and Schlich [50] investigated how to avoid the immediate split-up caused by the instantiation of nondeterministic bits. Their approach, which we generalized and incorporated into our generated simulators, is called *Delayed Nondeterminism*.

8.6.1 Concept of Delayed Nondeterminism

The general idea in delayed nondeterminism is to postpone the instantiation until an exact value is required. Figures 8.4 and 8.5 illustrate this. Fig. 8.4 shows the evolution of a computation path using deterministic simulation, whereas Fig. 8.5 shows the same computation path using delayed nondeterminism.

Deterministic simulation. For an 8 bit processor such as the Atmel AVRs, each of the read instructions in the code fragment would result in 256 successor states being created. Their distinguishing value is stored in register R6. Each of these states is the starting point of a new path on which the following instructions have to be executed. Therefore, there are actually 256 different microcontroller states when executing the second read instruction. Thus, the blowup amounts to 65,536 states being created by the two read instructions.

Delayed Nondeterminism. Instead of immediately instantiating the nondeterministic bits, delayed nondeterminism propagates nondeterministic values through memory. That is, internal memory cells may be marked as being nondeterministic, and therefore more instructions than before may have to deal with nondeterminism. As can be seen from the second figure, this approach eliminates most of the states in the example, which do not even have to be created. The reduction is possible because the instruction at program counter *0x2f* does not depend on the exact value copied into register R6, and the instruction at program counter *0x31* only

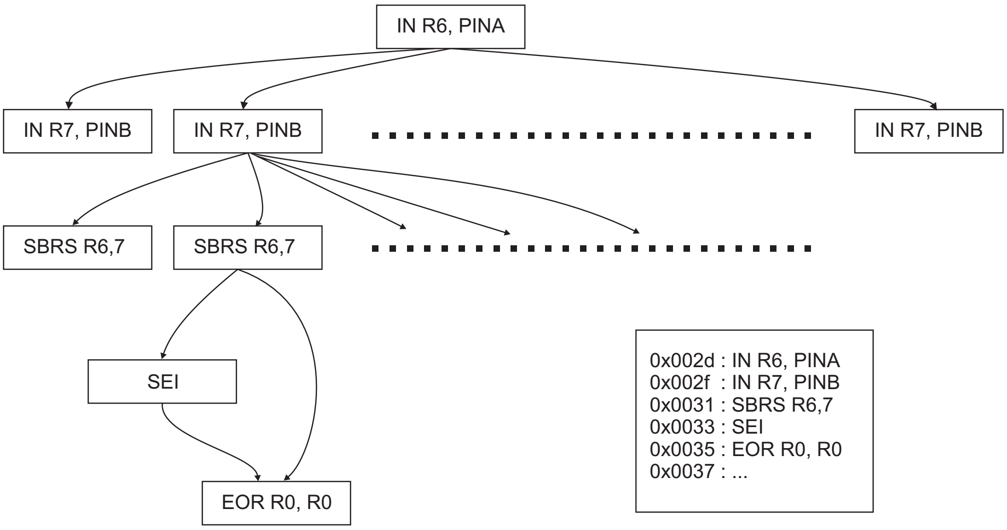

Figure 8.4: Computation path using deterministic simulation

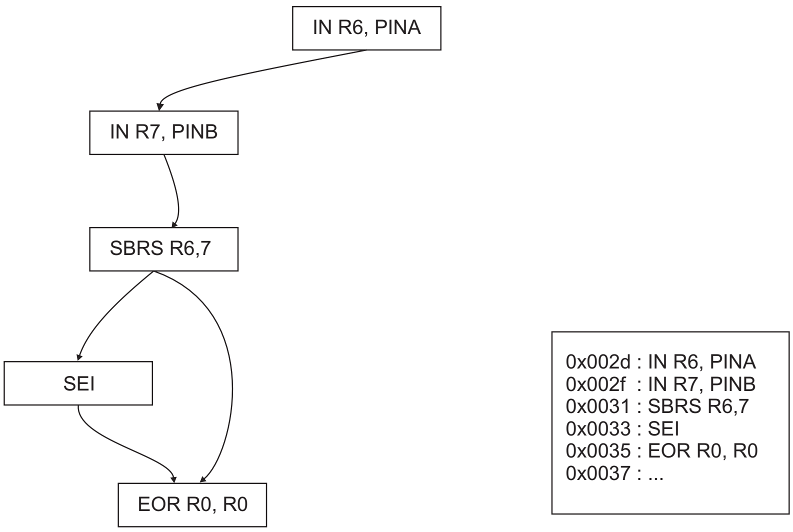

Figure 8.5: Computation path using Delayed Nondeterminism

requires a single bit of R6. Furthermore, the exact value of R7 (i.e., the second read) is not needed at all within the scope of this code snippet, hence instantiating it is not required yet.

Analysis. RISC-like architectures like the Atmel ATmega microcontrollers usually perform computations in a *load-store* manner, meaning values are loaded from some memory location into some internal registers, transformed, and written back to memory. This structure is exploited by delayed nondeterminism: the instruction encountering nondeterministic values in the first place, e.g. a read, does not require a deterministic value. Reads do neither modify nor use the value, they only *copy* it into an internal memory. Subsequent instructions may then also only propagate the marker, or they may depend on the actual value. In that case, it is possible that not all bits need to be instantiated but only a subset. This can be observed in Fig. 8.5, where the SBRS only needs bit 7 to be deterministic.

For delayed nondeterminism to have an impact on actual programs, several conditions must be met:

- First of all, the microcontroller must provide copy instructions that do not have side effects. For instance, the Infineon XC167 sets flags depending on the data contained in the source of a copy [50, 67]. On such a platform, it is either necessary to allow nondeterministic flags, resulting in nonderministic control flow, or to immediately instantiate the source. In the latter case, delayed nondeterminism is therefore not applicable. In the former case, i.e., permitting nondeterministic control flow, the expressiveness of the model checking results is severely reduced (cf. *Nondeterministic Status Register* in [36]). The reason for this is that in the deterministic case, data involved in operations results in flags being set. If that dependency is deliberately broken up, it is possible for the control flow to reach locations that would otherwise be inaccessible (e.g. if the flags indicate that the last operation yielded a result that was at the same time zero, negative, and odd).

- Second, programs must depend on data the value of which is unknown at compile time. A program that does not depend on values read from the environment would result in a state space that cannot be reduced by delayed nondeterminism. Such programs are not unrealistic, for instance as they may perform internal computations and interact with the environment by sending values to it. They may still result in state explosion, for instance if the results of such internal operations are stored in memory.

8.6.2 Delayed Nondeterminism in Generated Simulators

We investigated multiple approaches towards adapting the technique such that it becomes applicable to arbitrary platforms:

- Dynamic Delayed Nondeterminism. This approach consists of simply monitoring the memory locations accessed by an instruction in a deterministic simulation step. Then, the simulator state is reverted, and the source locations are checked for nondeterministic bits. If any such bit is found, then all written locations are marked as fully nondeterministic. The only exception are those registers relevant for control flow, which trigger an instantiation in case they are a write target.
- Explicit description of DND semantics. For this approach, we added an optional second `execute` section to the instruction element in SGDL. The new section contains a manually implemented second semantics to be used only when delayed nondeterminism is active.
- Automatic derivation of the semantics for delayed nondeterminism. A thorough analysis of the SGDL code is used to determine the effects of instructions, and to derive an abstract semantics. In case this is not possible, it relies on an additional manual description.

Our examinations showed that the principle of Dynamic Delayed Nondeterminism (DDND) as such works as intended, and requires no modification of the SGDL descriptions. Nondeterminism in the input is propagated through memory. However, there is a serious disadvantage: for a given instruction, a single nondeterministic bit in the input suffices to render all the output locations nondeterministic. This effect propagates as well, effectively resulting in many memory locations becoming nondeterministic. At some point in time, for most programs we examined, an instantiation is triggered, which then creates even more successor states than deterministic simulation would have. Therefore, we decided to drop this approach.

An explicit description of the DND semantics was the key idea for our second approach. This is technically very close to the way delayed nondeterminism is implemented in the the manually created simulators, where the developer is tasked with creating a second set of instruction semantics. This requires the developer to implement a new instruction visitor, and, for each instruction, code for instantiating all memory locations that have to be deterministic. Furthermore, the new abstraction has to be integrated into the simulator. For a synthetic simulator, the corresponding steps in the SGDL description are: add a new section called `execute "DND"` to each instruction for which the DND semantics deviates from the deterministic semantics, and define a set of locations to be instantiated in DND mode by means of the `dnd instantiate` directive. The integration of the new available abstraction can be handled automatically by the SGDL compiler.

While the explicit approach via `execute "DND"` and `dnd instantiate` works as supposed, and yields reductions similar to those reached in the manual simulators (cf. Chapt. 10), it is also the one in which the SGDL approach provides the least support. That is, it still requires the developer to be familiar with the concept of Delayed Nondeterminism, and to manually implement the necessary instruction semantics. It is desirable to automate these tasks. We explored approaches to this goal and developed a concept based on static analyses of the code sections in SGDL descriptions (i.e., execute sections, interrupt handler code sections, and subroutines), which we published in a paper [27]. As of this writing, however, the concept has not been implemented yet, hence we refer to the section on future work for details on the approach (cf. Chapt. 11).

8.7 Sanity Check-Based Approaches

Certain abstractions are specific to synthetic simulators, that is, there is no counterpart in existing handcrafted simulators. Their purpose is to deal with peculiarities in the use of resources, in which an inappropriate handling of value and nondeterminism mask may result in state explosion. While this class of problem may also occur during development of handcrafted simulators, it is easier to resolve there, as information on both the active instruction and the intended mode of simulation (e.g., deterministic, or delayed nondeterminism) is readily available.

The abstractions presented in this section are, from a technical stance, comparatively simple, which contrasts with their impact on state space sizes. When deactivated, several of our case study programs resulted either in state explosion, or would require very long verification times. For instance, the time effort for building the state space for the test program called *Light Switch Error* increased from minutes to hours. Actual values are given in Chapt. 10.

8.7.1 Implicit Determinization Guards

Memory cells may become nondeterministic during simulation, for instance by instructions copying a nondeterministic value into them, or by Lazy Stack Evaluation. The associated difficulty is that eventually, these cells should become deterministic again. For instance, the AVR instruction `LDI Rx, #imm` loads a constant value into a memory cell. Constants being always deterministic, this instruction should therefore write `imm` into the value and reset the nondeterminism mask of target register `Rx`. However, extending the instruction description such that the NDM is always modified even for fully deterministic simulation would render the `execute` section useless for an automatic derivation of techniques such as delayed nondeterminism.

The above situation becomes a serious problem when using Lazy Stack Evaluation with memory initialization. As described in Sect. 8.3, LSE benefits from a memory initialization, in which the entire memory is marked as nondeterministic on startup. However, this also marks the area in which global variables are located as nondeterministic. Without an adaptation of the NDM, this nondeterminism is not eliminated even if the program contains instructions to initialize these variables to some value (*program initialization*, as opposed to *memory initialization*). Hence, when the program accesses a global variable, that variable has to be instantiated, which usually results in state explosion. Moreover, this effectively eliminates program initialization, allowing undesired values that cannot occur on the real device, therefore reducing expressiveness.

A simple though effective solution to the requirement "keep explicit NDM modification out of execute sections" and to this problem is illustrated in the Message Sequence Chart [66] in Fig. 8.6. The idea is to monitor all addresses written during successor state creation, i.e., the effects of instructions, interrupt handling code, and triggers. To this end, we use the observer interface described in Sect. 7.8. The observer creates two sets: a set of addresses at which the value was modified, and another set of addresses at which the NDM was modified. At the end of a step, we use these sets to compute the set of addresses the value of which was modified, but the NDM remained unchanged. We can safely assume that these addresses were assigned a deterministic value by an instruction unaware of the necessity to modify the NDM. Hence, in a final step, we reset the NDM of these addresses to 0, thus marking the location as entirely deterministic.

If the developer wants to prevent this mechanism from being applied to a location, he can either deactivate it in the options for all memories, or explicitly modify the NDM when modifying the value to indicate that there are no "forgotten" NDM bits.

8.7.2 Restriction to Ternary-Valued Logics

Ternary, or three-valued, logics, are closely related to two-valued Boolean logics (cf. for instance [3, 51]). Besides the values for *true* and *false*, represented by 0 and 1, there also exists a third value in ternary logics, which represents an undetermined or *nondeterministic* value. In our setting, we call this value n. Considering n as a placeholder for a value that could be either 0 or 1 leads to a natural extension of the truth tables for arithmetic and logic operations. For example, $0 \wedge n = 0$, and $1 \wedge n = n$. It would also be safe though inaccurate to define the outcome of any operation involving n as being n again.

As pointed out before, memory in all generated simulators is represented by two parallel arrays. One of these arrays is called the *value*, whereas the other is called the *nondeterminism mask (nd mask).* The latter serves as a bit vector indicating all nondeterministic bits, i.e., if a bit is set in the nd mask, the logical value of the bit at that address is assumed to be any of $\{0, 1\}$. Hence, in such cases the content of the array for storing the *value* becomes irrelevant. Nevertheless, the array still exists for such addresses, therefore it has to have some value, which we decide, by convention, to be uniformly 0 for all nondeterministic bits. In this configuration,

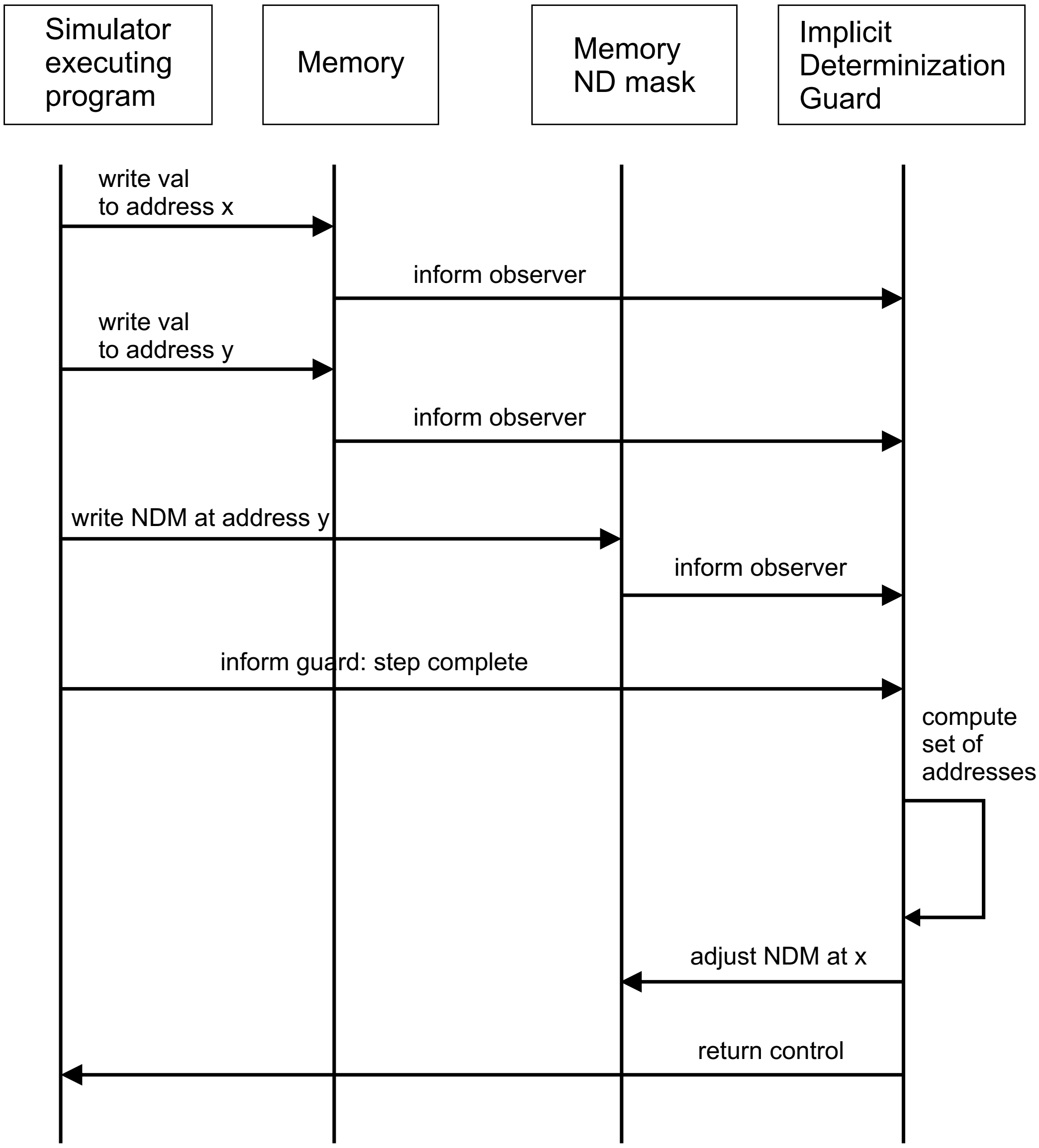

Figure 8.6: Message Sequence Chart illustrating the mode of operation of the Implicit Determinization Guard. During the step, the NDM of x is not explicitly modified, but the one for y is. Hence, the guard only performs a cleanup for x.

the memory contents represent the aforementioned value n. Therefore, memory cells containing a 1 in both the value *and* the nd mask array are considered to be in an inconsistent state, which does not correspond to any of the values in ternary logic (i.e., a fourth value).

The fourth value poses a problem in case the simulator developer has not properly covered the conditions under which a fourth value may be created. While an over-approximation is guaranteed even with the fourth value (the set bit in the nd mask would trigger an instantiation if necessary), this still allows for logically equivalent states being stored several times. To prevent this, we have added another sanity check, which enforces the restriction policy to allow only three values. Similar to the implicit determinization guards, a *ConsistencyGuard* monitors write accesses to the memory to which it is attached, which can be used at the end of a step to sanitize the content of modified cells.

8.8 Related Work

8.8.1 General Approaches Towards Countering the State Explosion Problem

A variety of solutions have been proposed to counter the state explosion problem. The survey by Clarke et al. [16] for instance mentions symbolic representations, partial order reduction, exploiting modularity, assume guarantee reasoning, inductive reasoning, abstraction and symmetry. In the next paragraph, we summarize this survey.

Symbolic representations and symbolic algorithms aim at replacing the explicit representation of values in states by symbols. Thus, entire sets of states may be represented by a symbol. According to Clarke, the concept dates back to McMillan, who discovered that using ordered binary decision diagrams (OBDDs) [12] would allow the verification of systems that were several orders of magnitude larger than those represented explicitly. The original approach scaled up to 10^{20} states, and further research on OBDDs increased this value up to 10^{120} states.

Considering [MC]SQUARE, there is also a symbolic representation, which is used in delayed nondeterminism. Technically, a state containing cells marked as being nondeterministic represents a set of states.

Except for partial order reduction and abstraction, none of the other approaches is used in [MC]SQUARE. Partial order reduction is used in a technique called Interrupt Handler Execution Reduction (IHER). However, the SGDL compiler does not explicitly generate support for it, hence it is not relevant for our contribution. Concerning abstraction, [MC]SQUARE features a variety of methods, some of which do not strictly comply with the definition found in the literature. We rather use the term for *any* means for reducing the size of the state space, compliant with previous publications on [MC]SQUARE. For instance, path reduction does not comply with the definition of an abstraction because the actual state space is not mapped to a smaller one. When path reduction is active, [MC]SQUARE still has to create and check the reachable part of the state space, though then it does not have to store it completely. Similarly, the sanity-check based approaches are labelled abstractions, though in the stricter sense, they are rather post-processing steps involved in state creation.

8.8.2 Specific Approaches

Lazy Stack Evaluation was presented by Schlich [68]. They integrated this technique into the handcrafted simulators in [MC]SQUARE that existed at the time, that is, the ATmega16 and ATmega128, and the MCS-51. Our work on LSE is based upon their results, and our contribution is an examination of the approach, with the intention of generalizing it. As pointed out in Sect. 8.3, we achieved an integration of LSE into the synthetic simulators, though not a fully automatic one.

Yorav and Grumberg [86] propose using static analyses for state space reductions. In their contribution, they present two reductions based on static analyses, one of which they call dead variable reduction. Self and Mercer [73] present a method for dynamic dead variable reduction that exploits runtime information in order to increase the accuracy of the analysis. Using runtime information allows them to rule out irrelevant computation paths, for instance code fragments after branches that cannot be reached for given parameter valuations. Variables used only in such unreachable code may be dead. Trying to detect this fact using a strictly static analysis must fail, however, as from the stance of the analysis, the branch might be taken. Thus, a static analysis must yield that the variables in question might still be alive. Self and Mercer implemented their suggested solution in the Estes model checker [44]. As they point out, there is a restriction: the state space of the program to be analyzed must not contain nondeterministic choices. The absence of nondeterminism implies that there may be only a single future for any given program location, which they require for the correctness of their method. If this cannot be guaranteed, then their dynamic DVR is not applicable. From our point of view, this is a restriction that may be acceptable for desktop and general purpose computing. In that setting, a program may be started with some parameters, perform computations, and terminate yielding a result. Opposed to this, embedded software often is used to control devices, and to do so continuously. Hence, embedded control software does not necessarily terminate, but reads input from an environment in an infinite loop. Due to these requirements, we consider it impractical to use the Self-Mercer DVR approach for verifying embedded software.

Path reduction is the other abstraction presented by Yorav and Grumberg in

[86]. Their method is based on a static analysis of the input program, of which they examine the control flow graph. In order to distinguish approaches based on static analyses from those that check reduction conditions on the fly, we call such approaches *static path reductions*. Opposed to these are *on the fly* algorithms. Originally, the on the fly algorithm in [MC]SQUARE was called *dynamic path reduction*, but this term conflicted with an already existing publication by Yang et al. [85]. Their contribution aims at checking for the feasibility of paths by means of an SMT solver, with the intention of pruning infeasible paths. Thus, their technique has a different objective than the path reduction in [MC]SQUARE. Consequently, in a contribution to a conference in which we participated [10], the technique was renamed. Throughout this thesis, we abide by the new naming. The key idea for our classification scheme is that an on the fly algorithm can detect reducible chains during state space creation.

A common denominator of both path reduction techniques is that they merge equivalent states into single states. The underlying theoretical foundation is reminiscent to that of a *stuttering bisimulation*, as described by Baier and Katoen [9]. Stuttering bisimulation is a more lenient variant of bisimulation, in which the simulating transition system may combine multiple steps of the simulated system into a single step (or represent a single step by many, respectively). It is necessary for the states involved to satisfy certain equivalence conditions, which is precisely the same requirement as in our reductions. Transition systems for which this holds true may be considered abstractions or refinements of the other, and the concept may be used to model systems at different levels of precision. Thus, it is possible to check a smaller transition system instead of a larger one, and only start verification of the larger system when no more errors are visible in the smaller (coarser) one. Concerning this aspect, however, there is a thorough deviation to the path reduction used in [MC]SQUARE: path reduction does not allow to check a smaller transition system instead of larger one. While the system that has to be stored may in fact be smaller, i.e., the state space consumes less memory, the verification process still has to check the entire reachable state space. Hence, the larger system is always built even if path reduction is active.

Delayed nondeterminism is described by Noll and Schlich [50] and Schlich [68]. Improvements over this were investigated by Kamin [36], whose contribution also contains new approaches such as the *Nondeterministic Status Register* technique.

A survey of several on the fly approaches to state space reduction is given by Pelanek [56]. The paper covers, among others, techniques based on discovering dead variables as well as techniques similar to path reduction. It also illustrates the theoretical background and examines their effectiveness.

Besides these approaches to reducing the number of states to be stored, it is also possible to reduce the size of the state space by means of a more efficient storage. We already mentioned the approach originally dating back to Clarke, namely using

OBDDs for representing sets of states. Another concept tackles the memory footprint by avoiding redundant representations and compressing states. [mc]square can compress individual states using Run Length Encoding (RLE) and ZIP compression. As already described by Schlich [68], RLE appears to be the superior choice for storing states representing microcontroller memories, as these memories are often initialized to 0 and only a comparatively small number of cells is changed during program execution. The latter holds true at least as long as programs do not depend on large blocks of data, which might occur, for instance, when attaching storage devices to an embedded system. Accessing data blocks on hard disk drives or Compact Flash cards would imply at least 512 bytes of memory consumption, as this is the smallest block size for these devices. Such a chunk of variable data would be infeasible to handle with RLE. Therefore, we reconsidered using a ZIP-like compression. The shortcoming of ZIP in [mc]square appears to be related to the way the compression is used: whenever the simulator creates a new state, it is first compressed, then added to the state space. The compression is performed independently of previous compressions. For RLE, this is not a disadvantage. For ZIP, which belongs to a class of compression algorithms called *dictionary-based compression algorithms*, however, this approach is a huge disadvantage. ZIP does not require storing the dictionary along with the compressed data, but creates it on the fly when compressing or uncompressing. Therefore, using ZIP on individual states is certain to waste most of the benefits of this algorithm, especially when the individual data chunk to be compressed is small. Possible improvements were investigated by Caron in a bachelor thesis [13], in which the dictionary was to be separated from the states. States would be represented by index entries, which were also to be separated for the different memories present in the microcontroller unit. The resulting algorithm was a *global state space compression*, as opposed to the existing *local* compressions. It resembles the approach used in the second generation of the Java Pathfinder model checker, where it is called *collapsing* of states [83]. Benefits of this approach over the RLE compression became already visible in the vicinity of one kB of memory usage, though the data used in the case studies was not, as intended, realistic data blocks, but generated test data.

9 Implementing Simulators With SGDL

In this chapter, we present three case studies concerning the implementation of state space generators and static analyzers using our synthesis tools. For the sake of brevity, we refer, within this chapter, to the implementation of *simulators*, whenever we want to refer to the implementation of an SGDL description of such a device.

Two of the case studies involved the implementation of a simulator for a device from Atmel's AVR family of microcontrollers. For these, we extracted as much code as seemed reasonable from the individual descriptions, and created a common code base for the AVR family. Furthermore, as specific devices support only a subset of the entire AVR instruction set, we created a library of supported instructions for the entire family. This library was created by one of our separate tools, which can parse the individual data sheets provided by Atmel (in PDF format), determine the subset of supported instructions, and create one file per device that contains boolean constants for all instructions. For using such constant lists, we extended SGDL such that instructions can be enabled or disabled (i.e., a guard for the availability of instructions during the synthesis process), and used a naming scheme for the constants that enable instructions. Hence, we can create simulators for specific AVR-family microcontrollers by adding an appropriate `include` statement that loads the associated constant list from the library, and by loading the parameterized instruction set.

Our third case study focussed on a microcontroller from Intel, the Intel 8051 from the MCS-51 family, which has a comparatively different architecture than the AVRs. While the foremost difference is the instruction set, there is also a significant difference in the interrupt system, and in the handling of I/O ports.

Bibliographic note: we published a shorter version of the descriptions of the ATmega16 and MCS-51 in a paper [28].

9.1 Atmel ATmega16

9.1.1 Background

The ATmega16 is a microcontroller from Atmel's AVR family [6]. All AVR microcontrollers are 8 bit devices and feature a reduced instruction set (RISC) architecture. Hence, the number of instructions is rather low at 131, while at the same time the number of general purpose registers (GPRs) has to be high, in this case 32.

The latter is due to the load-store concept, in which data is loaded from memory, modified inside registers, and rewritten to memory, as opposed to the availability of complex instructions performing operations (seeemingly) directly in memory (i.e., a CISC architecture) [51, 78].

Concerning memories, the ATmega16 provides 32 GPRs, 64 I/O registers, and 1 kByte of SRAM. The latter are mapped into the main address space, which consists of 1120 addresses. Additionally, it has 16 kBytes of flash memory and 512 bytes of EEPROM. The ATmega16 is structured as a Harvard architecture [5, 78], that is, program and data memory are separated, with the program being stored in the flash and the volatile data in the SRAM. Furthermore, there is a third address space for the EEPROM. Programs running on the device can modify themselves, which allows the implementation of boot loaders.

No memory management or protection unit is available, thus programs can basically read and write any memory location. The only exception is that the device can be configured at programming time by setting or unsetting so-called *fuses* and *lock bits*, which prevent access to some memory locations. However, while this can prevent programs from overwriting certain components, it mainly aims at securing the content of the device against unauthorised modification. Additionally, fuses can be used to configure the mode of operation, for instance by determining the location of the interrupt vector on startup, or the clock source in the physical device.

A total of 21 interrupts is available on the ATmega16. Interrupts are not prioritized except for their numbering, that is, when multiple interrupts occur at the same time, the one with the lower number will be dealt with first. The absence of hardware support for interrupt priorities implies that there is no means for the developer to configure an interrupt such that it can preempt another interrupt. When entering an interrupt handler, the hardware automatically disables the global interrupt enable flag. However, it is possible for the developer to immediately reactivate interrupts after such an event in order to allow for preemption.

The device communicates with its environment by means of four I/O ports, each of which is 8 bits wide. Developers can configure each pin individually as either input or output by means of special configuration registers, which are called *data direction registers*. Additionally, I/O pins can have so-called *alternate port functions* in case an on-chip peripheral component takes control of them, for instance an analog-to-digital converter, a timer generating PWM signals, or a USART. Further details on on-chip peripherals are available from the data sheet [5].

9.1.2 Implementation

Our implementation of the ATmega16 is based on an existing instruction set description for the AVR family taken from the AVRora project. As pointed out before, we originally decided to base our synthesis system on the ISDL language and associ-

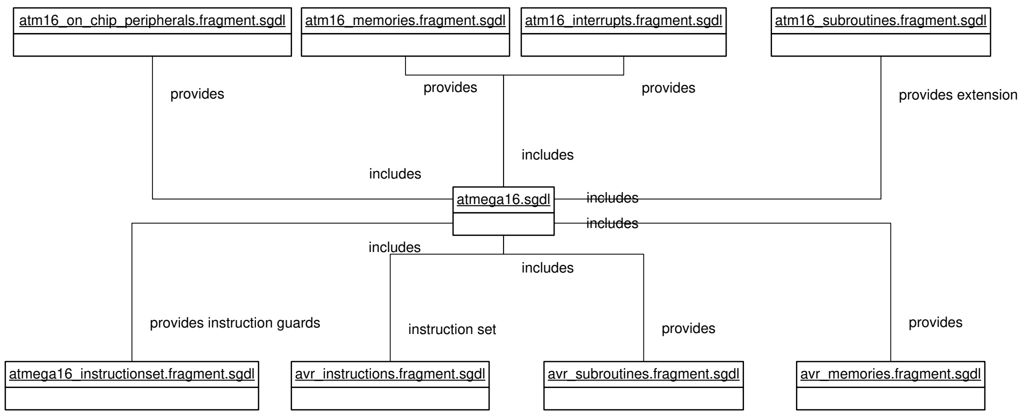

Figure 9.1: UML class diagram of the ATmega16 SGDL description

ated processing tool chain from AVRora, and to gradually extend and modify that system. The AVR description of the instruction set from AVRora was mostly complete. Due to the lack of a suitable description mechanism for resources (mainly, memories and interrupts), however, we had to develop means for describing these on our own, and implement the necessary code. The ATmega16 served as our test bed for these extensions, and was developed along with SGDL and the tool chain for processing SGDL code. Consequently, there is no valid data available on the exact time required for implementing this simulator.

Figure 9.1 illustrates the structure of the SGDL description of the ATmega16 simulator. The individual sizes of these files, measured in lines of code (LOC), is displayed in Table 9.1. An overview of their respective functions is shown below:

- ATmega16-specific files
  - *atmega16.sgdl*: this is the main file of the simulator description, which declares the name of the microcontroller. Its main purpose is to declare constants and to include more specific parts of the description. The aforementioned constants are needed for instance to locate such registers in memory whose location may differ between different members of the AVR family. The included instruction set description relies on the existence of these constants.
  - *atm16_interrupts.fragment.sgdl*: provides a description of the interrupt system of the ATmega16, consisting of the declarations of individual interrupts and a declaration of the interrupt vector.
  - *atm16_memories.fragment.sgdl*: contains declarations of memory aliases

File	Lines of code	Device-specific
atmega16.sgdl	241	yes
atm16_interrupts.fragment.sgdl	877	yes
atm16_memories.fragment.sgdl	289	yes
atm16_on_chip_peripherals.fragment.sgdl	86	yes
atm16_subroutines.fragment.sgdl	92	yes
atmega16_instructionset.fragment.sgdl	122	no
avr_instructions.fragment.sgdl	2,057	no
avr_memories.fragment.sgdl	109	no
avr_subroutines.fragment.sgdl	279	no
Total	4,152	

Table 9.1: Code size overview for the ATmega16 SGDL description

that cannot be supposed to be present in all AVR microcontrollers, but only in the ATmega16. Also contains the ATmega16-specific part of memory initialization, and a description of dependencies between memories that is needed for generating a static analyzer (e.g. which additional registers are read or written by the hardware when reading or writing a register).

  - *atm16_on_chip_peripherals.fragment.sgdl*: contains code to model on chip peripheral devices such as timers, the behavior of ports, the USART, or the A/D converter.
  - *atm16_subroutines.fragment.sgdl*: subsumes all of the functions and procedures used in the other files.

- Library files. *atmega16_instructionset.fragment.sgdl*: contains, for each of the instructions declared in *avr_instructions.fragment.sgdl*, a boolean constant that either enables or disables the instruction. Disabled instructions are supposed to be not present in the specific architecture, in this case the ATmega16.

- Shared files
  - *avr_instructions.fragment.sgdl*: this is the instruction set for the entire AVR family.
  - *avr_memories.fragment.sgdl*: declares memories common to all AVR devices. Constants are used wherever the size of a memory varies between devices, such that it is possible to define the value of that constant in one of the files for a specific device.

– *avr_subroutines.fragment.sgdl*: provides subroutines required for modelling AVR microcontrollers.

9.1.3 Validation

Besides the measures described in Sect. 7.9, which can be performed for any generated simulator, we also carried out the following steps to increase confidence in the generated ATmega16 simulator:

- Parallel runs with the handcrafted ATmega16
- Randomized HEX file input

As [MC]SQUARE already provides a handcrafted simulator for this platform, which has been developed and error-checked over several years, the obvious approach for establishing a certain degree of confidence in the new simulator was to run both simulators in parallel for the same program. This procedure is known in the literature as *back to back testing* (cf. for instance [40]). The first stage of this was to perform *trace comparisons*. For this, we instrumented both simulators such that they logged their program counter values to a suitable location. The resulting traces allowed us to detect deviations in the control flow. Such deviations typically indicated different handling of branch instructions, or of flags used by such branches that were incorrectly set by preceding instructions. While this kind of comparison proved useful during the early stage of development, however, it became less effective over time. The primary reason for this is that it can only detect serious errors in instruction descriptions. Moreover, it requires the use of programs that contain few or no nondeterministic choices because a different successor creation order might invalidate the trace, even if the created states were semantically equivalent when ordered appropriately.

Thus, the second stage in comparing the output from both simulators targeted the resulting state spaces during model checking. Such tests may be conducted more or less lenient. The strictest equivalence requirement would be to demand isomorphism, i.e., the resulting state spaces must be identical except for their numbering. A slightly less strict variant would be to require the number of states in each state space to be identical. However, both criteria are impractical. The reason for this is that neither the generated nor the handcrafted simulator provide an exact simulation of the ATmega16 as described in the data sheet. For instance, peripherals used to be simulated very accurately in earlier versions of the handcrafted simulator, even though the details were not accessible to the model checker. Therefore, an improved version deliberately abstracted from these details, resulting in fewer states without loss of expressiveness. As a consequence, however, the number of states varied. The *identical number of states* criterion would therefore not be satisfiable even for a single simulator, and certainly not for *different* simulators. Consequently, our correctness requirement is that for any given program, the generated simulator must create a number of states that is at least in the same order of magnitude *if and only if both simulators use the same set of abstractions.* Chapter 10 illustrates to what extent this goal has been met.

Randomly generated instructions were the third stage of testing. To this end, we generated HEX files containing random instructions, and had these executed by both simulators. A complete dump of memory contents was then used to check for deviations. This is an extended version of the trace comparison used in stage 1. The concept is analogous to the one described in detail in the section on the implementation of the MCS-51. Using these tests, one error in flag handling was detected.

9.2 Atmel ATmega644

9.2.1 Background

The ATmega644 is another member of the AVR family of microcontrollers from Atmel [7]. Hence, its internal structure is similar to that of the ATmega16. Our motivation for adding support for this device is twofold: first, the device is used in a lab course for undergraduate students at our institute [76], which provides us with a large number of possibly faulty programs to verify. Second, we wanted to prove the effectiveness of our library of AVR descriptions.

In the aforementioned lab course, students are expected to create an operating system for an embedded system, which involves preemptive multitasking using several scheduler strategies, memory management, and communication with external devices. From our experience in supervising sessions of this course, there are several frequent errors related to race conditions and wrong use of peripherals, which are hard to detect using only testing and debugging. Model checking these programs could be more efficient, though likely to be prone to state explosion. Our intention was therefore to strip down programs known to be faulty to a mere minimum and use [MC]SQUARE to scan for such hard to detect issues. Given that [MC]SQUARE did not support the ATmega644 used in the lab, and that creating one using the manual approach would require at least six man-months with uncertain outcome, this had not been attempted before. Using SGDL, however, we expected the effort to be less than one month, which seemed reasonable enough.

Compared to the ATmega16, the ATmega644 is in most respects the "larger" device. Its internal SRAM has a size of 4 kBytes, as opposed to 1 kByte on the ATmega16. The flash memory for storing programs consists of 64 kBytes, and its EEPROM of 2 kBytes. The CPU core, however, is the same, at least from the programmer's point of view. That is, the number of instructions is also 131 and

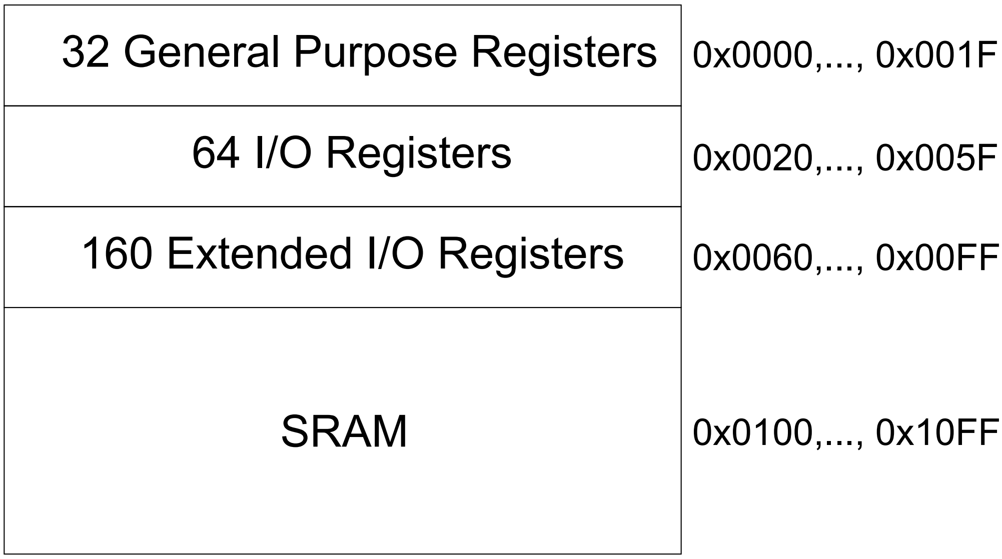

Figure 9.2: Memory layout of the Atmel ATmega644, as described and also illustrated in the device datasheet [7]

there are 32 general purpose registers.

The memory layout is roughly the same, with GPRs and I/O registers mapped into the main address space. Also, the ordering is the same: lowest addresses for GPRs, followed by I/O registers, and eventually, SRAM cells. A noteworthy difference, however, is that the ATmega644 implements a very large number of internal peripherals, which results in the need for a larger I/O register range. As shown in Fig. 9.2, the I/O register mapping range up to address *0x60* also exists on the ATmega644, but it is followed by a second one that stretches over an additional 160 bytes. Thus, the SRAM starts at address *0x100*.

Both the ATmega16 and the ATmega644 possess 32 I/O pins, distributed over I/O ports A to D. Pins are handled in the same way as on the ATmega16, that is, by means of three dedicated registers DDRx, PORTx, and PINx, where x $\in \{A, \ldots, D\}$. The most obvious difference is that on the ATmega644, each of these pins is associated with at least one interrupt. Consequently, also the number of interrupts is higher, though some of these are grouped together, reducing the number of existing individual interrupt sources. In total, the interrupt vector distinguishes 28 interrupt sources.

9.2.2 Implementation

Analogously to the ATmega16, our implementation of the ATmega644 is based on our library of AVR instruction set guards and our externalized, parameterized instruction set. Hence, the implementation of the ATmega644 required only the description of the memories, the interrupt system, and a review of instruction semantics to check for deviations that were not visible when creating the library (which was created based on experiences with the ATmega16). As a consequence, the effort for implementing the simulator for this microcontroller was considerably lower than when creating a new one from scratch using SGDL, and very low when compared to the effort for implementing it manually. Furthermore, we used the kick-starter code templates in the SGDL preprocessor to create the interrupt system, which also saved time.

The overall effort for implementing the ATmega644 amounted to approximately 21 hours of work, meaning it was possible for us to achieve it within half a week. However, we were familiar with both the SGDL system and the format of the data sheets published by Atmel. This biased the results in our favor. Therefore, we assume that for an average developer not familiar with neither the effort would be higher. An estimate would be one to two weeks, though for certainty about this, it would be necessary to conduct experiments with a sufficiently high number of participants.

The file structure for the ATmega644 simulator is structurally equivalent to the one of the ATmega16, as shown in Fig. 9.1 and described in Sect. 9.1. This includes the references to external library files. Therefore, no further detail on the structure is given except for the required lines of code. These are shown in Tab. 9.2. As can be seen from the table, more than half of the code base could be reused from the shared library (approx. 2,500 LOC).

9.2.3 Validation

Most of the validation required for the ATmega644 simulator was already included in the efforts for the ATmega16. The instruction set is not part of the corresponding SGDL descriptions but is included as a parameterized library, and was subject to extensive testing in the tests for the ATmega16. Hence, we identified the following remaining possible sources for errors: first, the guards needed to enable instructions, second, the semantics of instructions, third, the interrupt system, and fourth, additional subroutines. We now describe each of these sources and our countermeasures in more detail.

Guards. The guards for instructions decide whether an instruction is present in a specific AVR device. As described before, we extracted these guards from the PDF datasheets for individual devices by means of a tool, and stored them in a file

File	Lines of code	Device-specific
atmega644.sgdl	429	yes
atm644_interrupts.fragment.sgdl	918	yes
atm644_memories.fragment.sgdl	313	yes
atm644_on_chip_peripherals.fragment.sgdl	90	yes
atm644_subroutines.fragment.sgdl	144	yes
atmega644_instructionset.fragment.sgdl	122	no
avr_instructions.fragment.sgdl	2,057	no
avr_memories.fragment.sgdl	109	no
avr_subroutines.fragment.sgdl	279	no
Total	4,461	

Table 9.2: Code size overview for the ATmega644 SGDL description

containing the guards in the form of boolean constants. If for any reason such a file should be incomplete or erroneous, an instruction might be unintentionally active or inactive in a device. In case it is deactivated even though it should be present, the generated simulator would not know the instruction pattern, and throw an error in case it encounters it in a program. Vice versa, an instruction that should be unknown to a device must result in an error when encountered, or at least not have the effect usually associated with it. Verifying this property for the ATmega644 consisted of manually comparing the list of supported instructions in the data sheet to the list of guards, and was manageable within less than an hour.

Semantics of instructions. While the tests for the ATmega16 involved extensive testing of instructions, it was possible still that some instructions have slightly different semantics. Such deviations were to be expected for instructions dealing with addresses that are not present on the ATmega16, but are on the ATmega644 with its larger memories. We therefore performed a review of such instructions that accessed memories, e.g. those using address registers, and checked whether the SGDL code for these used possibly limiting code such as too small variables for pointers.

Interrupts. Considering that the interrupt system in SGDL was tailored to meet the requirements in describing the Atmega16, describing another device from the same family turned out to be possible without any modifications to the language. That is, the kind of preconditions for enabling an interrupt source, and for enabling the interrupt itself, were the same for both ATmega16 and ATmega644. Only the actual event sources differed. Hence, we considered it appropriate to only check the correctness by two programs: one in which no interrupt was enabled, and one in which all interrupts were enabled. In the former case, the simulator embedded into the [MC]SQUARE GUI was expected to show no branching of the computation path, whereas in the latter case, it had to show a branch for each of the enabled interrupts.

Subroutines. All subroutines used in the ATmega644 implementation are either getter methods that map a hardware state to a value, or are reused from the library. The getters are integral parts of other functionality such as the interrupt system, which is why they are covered by the respective tests. Hence, no additional tests for individual methods were added so far.

9.3 Intel MCS-51

9.3.1 Background

The MCS-51 is a family of microcontrollers from Intel [65, 78]. Its first and best-known member is the 8051, which is in production since 1981. Despite its age, variants of the device are still widespread today, and MCS-51-based designs are available from a variety of manufacturers. More recent implementations also integrate peripherals not present in the original version, such as USB and CAN. Furthermore, there are also open source implementations of the 8051 CPU core freely available on the Internet, e.g. [52].

Similarly to the AVR family, the 8051 is an 8 bit microcontroller. In contrast to the more modern AVR, however, the MCS-51 family is based on a complex instruction set computer (CISC) design. The number of instructions according to the data sheet reads 111, but, considering duplicate instructions due to different addressing modes, there are actually 256 different instruction encodings. Due to the CISC approach, the instruction set contains instructions that can modify data directly in memory, which contrasts with the load-modify-store approach found in RISC devices. Hence, the device requires far less registers.

The sizes of the addressable memories are comparatively low. In principle, MCS-51 devices can address 64 KB of program memory and 64 KB of data memory (i.e., RAM), but only when using external memories, which comes at the cost of up to two I/O ports. Internal memory sizes are typically much lower. The 8051, on which we focussed in our case study, has only 128 bytes of RAM, including the general purpose registers. These are organized in four register banks of eight registers each (i.e., $R0, \dots R7$), which can be switched by software. Special function (SFR) and I/O registers are mapped to the SRAM addresses 128 to 255. This memory organization complicates access to memory locations at the same address as the SFRs in devices such as the 8052.

A specialty of the MCS-51 is the existence of addressable single bits, and of instructions for directly manipulating single bits. There are two bit-addressable ranges in memory on the 8051, one in the lower 128 bytes and one in the upper 128 bytes (SFR region). Similarly to the duplication of addresses in devices with more than 128 bytes of memory (like the aforementioned 8052), it is therefore necessary to distinguish *bit addresses* and *byte addresses*. Which addressing is used depends on the instruction. An advantage of the existence of addressable bits is that there is very little need for specialized flags, as the developer can realize custom flags in software.

Due to the low amount of available memory, the stack pointer (SP) on the 8051 is only eight bits wide. Along with the rather tight memory bounds of 128 bytes, from which at least the lowest register bank has to be subtracted in order to provide one set of working registers, there is thus very little space for activation blocks needed for calling functions. This restriction has an impact on the design of software intended to run on the 8051, as cascading function calls could easily result in the stack hitting the register bank.

Concerning the interrupt system, there are five distinct interrupt sources on the 8051. INT0 and INT1 are external interrupts, which can be either level- or edge-triggered. The Timer 0 and Timer 1 interrupts occur whenever the respective timer overflows, and the Serial Port Interrupt signals a transmission or reception of a byte via the UART. Interrupt priorities can be configured by software, and interrupts of higher priority can interrupt the handlers of lower priority interrupts. Each interrupt can be assigned to one of two layers. Within a layer, there is a fixed ordering of interrupts, such that there is a well-defined execution order even in case of simultaneous occurrence of two interrupts of the same priority. Consequently, to technically allow for cascading interrupts, the hardware does not automatically delete the global interrupt enable flag when entering an interrupt handler. Instead, the possibility of further interrupts depends on the configuration of the priorities.

The 8051 communicates with its environment via four I/O ports, which are enumerated. From the programmer's stance, however, there are at most three ports. Port 0 and Port 2 are used for outputting 16 bit addresses, leaving only Ports 1 and 3 for I/O operations. Each of the pins of the ports can be configured individually as input or output. A peculiarity of the MCS-51 family is the way this configuration has to be performed. In contrast to, for instance, the AVRs, where port configuration is handled by means of dedicated *data direction registers*, there is no such register on the MCS-51. Instead, port pins are configured as input by writing a logical 1 into the corresponding bit in the associated port register. This is the same procedure that is to be performed when writing output to the pin. Furthermore, ports are equipped with an additional buffer (a D-flip flop), which can be read instead of the state of the actual pins. It depends on the instruction whether the pin or the buffer is accessed.

9.3.2 Implementation

The MCS-51 SGDL description was not based on any previous work, hence we were able to accurately measure the effort for implementing a simulator using our language and tool chain. As [MC]SQUARE already contains a handcrafted simulator for the 8051, we also had a reference for comparing our effort to that of a fully manual approach.

Effort

Compared to the usual six months span required for manual implementation, the effort when using SGDL was considerably lower. A first version of the simulator was operational after only 23 hours. At this stage, programs could be loaded, instructions could be disassembled and executed, and their effects could be inspected on the GUI. Implementing a first version of the interrupt system, providing names for registers such that they were available for use in atomic propositions in CTL formulas, and adapting the synthesis system wherever necessary, increased the effort to 40 hours. This did not include the rather complicated modeling of the I/O ports, which is not as cleanly structured as on the AVRs, where port configuration is controlled by separate registers. Implementing these, and improving the interrupt system such that it results in a tighter over-approximation, increased the total effort to approximately 60 hours.

Therefore, we come to the conclusion that the basic implementation of a simulator for an average microcontroller can be handled within two weeks. The 8051 may be simple with regard to on-chip peripherals and memories, but both the interrupt system and the I/O handling result in a more complicated modeling.

Technical Aspects

Similarly to our implementation of the ATmega16 and ATmega644, we have split the description of the MCS-51 into several files, in anticipation of the implementation of further members of the family. As we have not added any library of device capabilities yet, though, the file structure is simpler:

- *mcs51.sgdl*: the main file, in wich the other files are referenced by `include` statements.
- *mcs51_instructions.fragment.sgdl*: a description of the instruction set.
- *mcs51_interrupts.fragment.sgdl*: contains the complete interrupt model, including required triggers and subroutines.
- *mcs51_memories.fragment.sgdl*: declarations of memories and aliases.

File	Lines of code	Device-specific
mcs51.sgdl	85	yes
mcs51_interrupts.fragment.sgdl	269	yes
mcs51_memories.fragment.sgdl	160	yes
mcs51_subroutines.fragment.sgdl	387	yes
mcs51_instructions.fragment.sgdl	1,503	yes
Total	2,404	

Table 9.3: Code size overview for the MCS-51 SGDL description

- *mcs51_subroutines.fragment.sgdl*: general-purpose subroutines.

9.3.3 Validation

For validating the correctness of our implementation, we pursued a three-stage approach. First of all, we used our automatically generated test stubs (cf. Sect. 7.9). Selecting such instruction encodings as seemed reasonable provided us with a first result concerning the correct operation of the disassembler. We also implemented test code for some of the more frequently used subroutines, for which the SGDL compiler had also generated test stubs. The next step was to run the existing handcrafted simulator by Reinbacher and our new synthetic one in parallel for some rather small programs, and to have the simulators log the reached program counter values to a file. We were then able to compare these traces and to check for equality. This test checked mainly for correctness of the instructions relevant for program flow, but did not provide adequate coverage of the way data was manipulated.

Therefore, we performed a larger test, which we published in a paper [64]. In this contribution, six different MCS-51 simulators were checked for trace equality, but more thoroughly than in our second test. Instead of checking the sequence of program counter values, the *entire* memory content was checked. Additionally, whenever a deviation between simulators was discovered, we additionally checked the instruction specification to figure out which of the simulators operated correctly, if any. This procedure uncovered several errors in our synthetic simulator, which occurred in corner cases that we had not implemented correctly. We subsequently fixed all of these errors, such that the synthetic simulator passed all tests used in the paper.

10 Case Studies

Within this chapter, we present several case studies, covering all of the platforms implemented with SGDL. Each case study consists of one or more programs written for a specific platform. There are multiple motivations for this:

- First of all, we need to verify that the generated simulators are actually capable of simulating the behavior of the respective hardware.
- Second, we need to establish a statement about the correctness of the simulation. That is, when model checking a program, the result must be exactly the same for the generated simulator as for the handcrafted one, where applicable.
- Third, state space sizes should be in the same order of magnitude for the generated and the handcrafted simulators.
- Finally, as we have emphasized the importance of fast simulation, which influenced the design of the internal state of our simulators, it is necessary for generated simulators to be able to compete with the handcrafted ones.

The verification goals from the first and second item in the above list can be combined by conducting model checking in [MC]SQUARE on a list of programs, and requiring that for both simulators and each formula, the model checking algorithm eventually yields the same truth value. Thus, this is a strict requirement, and erroneous behavior is simple to recognize, as formulas can only evaluate to either true or false. On the other hand, goals three and four are more difficult to attain. For state spaces to be of similar size, both simulators need to use the same set of abstractions. Currently, the handcrafted simulators provide more abstractions than the generated ones, which is why some of these need to be deactivated in order to guarantee a fair comparison. Even then, a more elaborate version of a technique may result in deviating sizes, which is why we require the number of states only to be in the same order of magnitude. Similarly, the fourth criterion, which focusses on speed, cannot be subject to a strict comparison. The reason for this is that the generated simulators need to operate on generalized data types, requiring conversions even for numeric operations. Opposed to this, the developers who created the handcrafted simulators could often use very fast operations that operate directly on suitable primitive data types. Due to this fact, generated simulators are to be expected to be slower by at least a constant factor.

Note on revision numbers. Throughout this chapter, we refer to specific versions of [MC]SQUARE by means of Subversion (SVN) revision numbers. The source code of [MC]SQUARE and the SGDL compiler, which is part of it, are kept in an SVN repository. During the writing of this thesis, however, [MC]SQUARE evolved into the ARCADE project, and its architecture was thoroughly changed. While [MC]SQUARE is a plain Java project, ARCADE is an Eclipse plugin, and as such, is based on OSGi [20, 43]. Furthermore, the revision control system was changed from SVN to Git, and ARCADE now resides in a Git repository on GitHub.com. Adapting SGDL and the compiler to the new architecture has been infeasible so far due to the estimated effort, which is why we decided to stick to [MC]SQUARE for the time being. Therefore, we cloned the SVN repository of [MC]SQUARE prior to the modifications that led to ARCADE. This decision has the following consequences:

- SVN revisions up to revision 9278 (inclusive) refer to the original repository.
- After SVN revision 9278, evolution of the projects continued in parallel. That is, revision numbers may occur multiple times.

Within the scope of this thesis, revision numbers larger than 9278 always refer to the cloned SVN repository.

10.1 Atmel ATmega16 Case Studies

The Atmel ATmega16 was of particular interest for us because of the manually implemented simulator for this platform that exists in [MC]SQUARE. This simulator is the oldest simulator in [MC]SQUARE, and it has constantly been maintained and improved. Thus, it is a stable platform, and provides nearly all the abstractions developed within the project. As a consequence, creating a competitive simulator using SGDL would imply that synthetic simulators can actually be a suitable replacement for such fine-tuned handcrafted ones.

As pointed out before, we used the ATmega16 as a test bed and reference platform during the development of SGDL and the SGDL compiler. For this reason, an exact comparison with regard to implementation effort is not possible.

Schlich [68] used a series of programs to illustrate the use of [MC]SQUARE on programs for this platform. In his rather extensive overview, he examined the memory consumption, number of states in the state space, and time required for model checking these programs. Hence, we recreated parts of this series of runs using the same programs, which allows, to a certain degree, to compare the quality of our generated simulators to that of the handcrafted one at that time. The validity of any such comparison is limited, though, because [MC]SQUARE consists of several components that also evolved since then. For instance, the static analyzer is one such component, and advances therein directly impact the size of the state space.

Some of these improvements were beneficial, others detrimental because the older implementation did not always guarantee an over-approximation due to implementation faults. Therefore, our generated simulator benefits from some new features that were not present at the time Schlich conducted his case studies. Consequently, for the sake of a fair comparison, we also provide the values obtained using the current handcrafted simulator.

10.1.1 Impact of Abstractions

In this case study, we investigated the impact of our abstractions on the size of the state space, number of states, and time required for model checking. For a description of the function of the programs under test, we refer to the description by Schlich. In order to facilitate the comparison with the values obtained by Schlich, we have tried to abide by the same table scheme as far as seemed reasonable. The comparison is feasible because the generated ATmega16 simulator supports the same set of abstractions as the handcrafted simulator at the time of Schlich's case study.

Setup

Our standard case is a configuration where Lazy Stack Evaluation and the Consistency Guards are active. Additionally, the interrupt system uses delayed nondeterminism. This is similar to Schlichs setting, where also lazy stack evaluation was active, and DND for interrupts was also active.

This case study was conducted on a SUN Fire X4600 M2 server equipped with eight AMD Opteron 8220 dual core processors at 2.8 GHz, 256 GB RAM, and 2 TB of hard disk space. Windows Server 2008 was used as the operating system, and Oracle Java Development Kit 1.7 as the Java Runtime Environment (JRE). Therefore, the time required for state space building is not comparable to the values by Schlich, who used a notebook instead.

SVN revision 9283 of [MC]SQUARE was used for this case study. The test files, taken from the separate [MC]SQUARE test file repository, were used in SVN revision 55.

The CTL formula $AG\,TT$ was used for all runs. In the syntax of [MC]SQUARE, this formula denotes $AG\,true$. It is an invariant and causes the model checker to build the entire reachable state space.

Results

Table 10.1 shows the results obtained using the generated ATmega16 simulator. The results for the current version of the handcrafted ATmega16 simulator are displayed in Table 10.2, and a summary of the values obtained by Schlich for the case *all abstractions active* is shown in Table 10.3.

As is visible from the tables, our generated simulator competes very well with regard to the number of states created and the number of states stored. In most cases, the values are slightly larger than for the current handcrafted simulator, and generally at least on a par with the older version. Deviations are likely to be due to a different modeling of internal states, especially of peripheral devices.

Considering simulation speed, the handcrafted simulator is obviously faster. Most obviously this is the case for Window Lift, which is the only program in this set resulting in a larger state space. The values suggest that the handcrafted simulator is approximately five times faster. As pointed out before, a certain speed tradeoff was to be expected due to the different handling of primitive data types.

Reductions achieved by the abstractions are also in the same order of magnitude for both simulators. This result means that using SGDL neither impairs the integration of abstractions into a state space generator, nor the quality of the abstractions.

Note on Memory Consumption

During the case study, we observed that the memory consumption could not be measured reliably. As [MC]SQUARE is implemented in Java, there is no direct way for determining the sizes of data structures used. While languages such as C provide a `sizeof` operator for this purpose, Java does not have any equivalent operation. Therefore, the typical approach towards computing the size of any data structure in a Java program works as follows:

- Create the data structure
- Obtain the current memory consumption m by means of a special call to the Java Virtual Machine (JVM)
- Eliminate all references to the data structure
- Trigger the garbage collection built into the JVM (repeatedly, if necessary)
- Obtain the new memory consumption m' by repeating the call to the JVM
- Memory used by the data structure was $m - m'$

The result is always an approximation and not exact. This approach is also implemented in [MC]SQUARE. In previous case studies, this procedure returned values that were reasonable enough, though sometimes a few repetitions were required. The typical initial size of the hash tables, for instance, turned out to be around 22 MB. This was also a recurring value in this case study for some of the smaller programs and both simulators.

Program	Options used	States stored	States created	Time [s]	Reduction of states stored
Light Switch	Standard	4,021	8,119	1.42	-
	DND	353	381	0.90	91.22%
	DVR	3,233	6,981	1.27	19.6%
	OTF-PR	472	56,788	4.83	88.26%
	DVR & PR	340	51,759	4.42	91.54%
	All	20	406	0.51	99.5%
Plant	Standard	131,573	138,262	14.89	-
	DND	131,573	138,262	14.30	0%
	DVR	131,573	138,262	11.67	0%
	OTF-PR	4,049	205,010	14.16	96,92%
	DVR & PR	4,049	205,010	13.59	96,92%
	All	4,049	205,010	19.54	96,92%
Reentrance	Standard	110,960	117,583	11.71	-
	DND	110,960	117,583	11.89	0%
	DVR	109,157	115,780	8.63	1.62%
	OTF-PR	6,623	352,649	23.45	94.03%
	DVR & PR	6,623	347,240	21.87	94.03%
	All	6,623	347,240	25.58	94.03%
Traffic Light	Standard	10,256	11,285	1.99	-
	DND	10,256	11,285	1.28	0%
	DVR	10,256	11,285	2.20	0%
	OTF-PR	516	30,149	2.59	94.97%
	DVR & PR	516	30,149	2.59	94.97%
	All	516	30,149	2.45	94.97%
Window Lift	Standard	2,559,671	2,934,246	256.84	-
	DND	340,406	532,146	40.88	86.7%
	DVR	333,548	373,205	31.74	86.97%
	OTF-PR	189,604	6,128,530	420.16	92.59%
	DVR & PR	22,429	741,167	68.77	99.12%
	All	7,639	314,807	27.92	99.7%

Table 10.1: Results of the run of the synthetic ATmega16 simulator on various programs

Program	Options used	States stored	States created	Time [s]	Reduction of states stored
Light Switch	Standard	4,268	6,296	0.38	-
	DND	352	380	0.50	91.75%
	DVR	3,318	5,119	1.78	22.26%
	OTF-PR	494	25,223	0.53	88,43%
	DVR & PR	357	21,604	2.25	91.64%
	All	15	331	1.07	99.65%
Plant	Standard	130,524	135,949	3.21	-
	DND	130,524	135,949	3.25	0%
	DVR	130,524	135,949	7.03	0%
	OTF-PR	3,205	167,114	2.54	97.54%
	DVR & PR	3,205	167,114	6.51	97.54%
	All	3,205	167,114	5.39	97.54%
Reentrance	Standard	107,649	110,961	2.45	-
	DND	107,649	110,961	2.38	0%
	DVR	147,381	154,004	3.47	-36.91%
	OTF-PR	3,312	124,207	1.65	96.92%
	DVR & PR	6,623	248,361	5.55	93.85%
	All	6,623	248,361	3.96	93.85%
Traffic Light	Standard	9,998	10,514	0.56	-
	DND	9,998	10,514	0.45	0%
	DVR	9,998	10,514	2.41	0%
	OTF-PR	259	15,131	0.26	97.41%
	DVR & PR	259	15,131	2.90	97.41%
	All	259	15,131	1.79	97.41%
Window Lift	Standard	2,342,564	2,589,665	47.54	-
	DND	316,334	442,055	8.90	86.5%
	DVR	300,078	328,098	11.77	87.19%
	OTF-PR	123,585	4,123,385	46.29	94.72%
	DVR & PR	15,382	531,494	11.68	99.34%
	All	5,182	223,709	8.80	99.78%

Table 10.2: Results of the run of the handcrafted ATmega16 simulator on various programs

Program	States stored	States created	Time [s]	Reduction of states stored
Light Switch	45	262	< 0.01	99.28%
Plant	9,844	218,022	2.82	94.41%
Reentrance	6,631	122,999	1.54	93.84%
Traffic Light	522	12,812	0.20	94.78%
Window Lift	5,232	217,577	2.62	99.78%

Table 10.3: Summary of the values obtained by Schlich [68] for the handcrafted simulator, SVN revision 2233, for the case *All options used*

However, during this case study, we observed large and frequent deviations. For instance, using the handcrafted simulator and the light switch program in the DND run, we obtained memory consumptions of 22 MB and 2,2 GB in subsequent runs. Similar values became visible for the synthetic simulator, e.g. in the reentrance program. The effect did not depend on whether [MC]SQUARE was restarted in between runs.

In order to eliminate outliers, we reconducted the run in cases that were obviously not credible. As the results did not stabilize, however, we assume that the deviation is due to some change in the current JVM (1.7), and that the JVM either allocates memory in advance or performs internal housekeeping involving large amounts of memory. Consequently, we decided to drop the column for the memory consumption from our tables.

Note on Possible Errors

In the table showing the results for the handcrafted simulator, i.e., Tab. 10.2, there is one peculiarity. Activating Dead Variable Reduction (DVR) for the reentrance program actually led to an increase of the number of states. This effect can be observed both for the change from the *standard* run to the *DVR* run and for the change from *OTF-PR* to *OTF-PR & DVR*.

Such an increase should not be possible: at most DVR should not be able to reduce the number of states for a given program, but never increase it. An increase is likely to be an error in the implementation of the static analyzer component of [MC]SQUARE. We examined the issue and located the instruction causing the effect, but a resolution was not possible.

As both the handcrafted and our generated simulator rely on the static analyzer, it is likely that the generated simulator is affected as well. The corresponding values for the generated simulator in Tab. 10.1 do not show any such increase, i.e. appear to be sound. However, a thorough investigation would be required to figure out the root causes for this.

10.1.2 Case Study: Window Lift

For our second case study, we used a program called Window Lift. It models an electric window lift for cars and was originally introduced by Schlich and Kowalewski in [69]. It was also used by Schlich in [68]. Three versions of the program are available: *Window Lift Error 1*, *Window Lift Error 2* and *Window Lift Fixed*. The *Error 1* program consists of 112 lines of C code, corresponding to 288 lines of assembly code, and is compiled using `avr-gcc`. We already used this version of the program in the previous case study.

Window Lift models the functionality of the power window lift by means of a state machine. Events such as *button pressed* trigger transitions and result value changes to specific output pins, that is, a Moore / Mealy automaton [19, 48, 51][1]. In the *Window Lift Error 1* program, there is a subtle error related to the behavior of the automaton in case an object is stuck in the window. The expected behavior in this situation is that the window has to be opened completely before normal operation is allowed again. However, in case two special events occur at the same time, it is possible for the program to return to normal operation before the window has been opened completely. In terms of the underlying automaton, this corresponds to taking a transition that is not supposed to be allowed. *Window Lift Error 2* represents a first attempt to removing this undesired behavior, which results in another error, and *Window Lift Fixed* actually resolves the issue.

The aim of this case study was to check the truth value of certain formulas. We used the handcrafted simulator as an oracle that yields the correct result.

Setup

This case study was conducted on the same SUN Fire server as the previous case study. [MC]SQUARE was used in SVN revision 9287 and the test files in revision 55.

Results

Table 10.4 shows the results, which were obtained using the synthetic simulator, for all three versions of the program. Analogously, Table 10.5 contains the corresponding values for the handcrafted simulator.

The formulas that were model checked, and which are referenced in the table, are the following:

- $\Phi_1 := AG(mode \geq 0 \wedge mode \leq 6)$

[1]Moore automata write output based on their state, whereas Mealy automata write depending on transitions.

- $\Phi_2 := AG(mode = 5 \rightarrow \neg E(mode = 5\, U(mode \neq 5 \wedge mode \neq 6)))$

Concerning options, the values in the tables were the following:

- Synthetic ATmega16 simulator
  - None: All abstractions deactivated. Only the consistency guards were active.
  - LSE: Lazy Stack Evaluation and the consistency guards were active.
  - Standard: LSE, DND, DVR, and the consistency guards were active.
  - All: as Standard plus On-the-Fly Path Reduction.
- Handcrafted ATmega16 simulator
  - None: no abstraction was active.
  - LSE: Lazy Stack Evaluation was active.
  - Standard: LSE, DND and DVR active.
  - All: as Standard plus On-the-Fly Path Reduction.

These settings were chosen such that certain properties become visible. First of all, we wanted to obtain the size of the state space during verification in the absence of any abstraction. In the second setting, *LSE*, only Lazy Stack Evaluation is active, which illustrates the impact of this specific technique on programs with multiple simultaneously active interrupts. Next, we were interested in the amount of time required for the fastest settings possible. As path reduction can increase the time effort, we decided to exclude it, resulting in the *standard* setting. Finally, to obtain values for the smallest state space size, we activated path reduction (i.e., *all*).

Inspecting these values demonstrates that, within the scope of this case study, model checking using the synthetic simulator yields the same truth values as when using the handcrafted simulator. Furthermore, the truth value is not affected by the set of active abstractions. The number of states for each corresponding run is very similar.

A significant difference can be seen in the amount of time required for model checking. In this regard, the synthetic simulator is about five to six times slower than the handcrafted one, as we already observed in the previous case study.

As for the previous case study, we have decided to omit the memory consumption, as the computation in Java is currently not possible reliably.

10.2 Atmel ATmega644 Case Studies

As pointed out in Sect. 9.2, the ATmega644 forms the basis for a lab course at the Embedded Software Laboratory of RWTH Aachen University, which is compulsory

Program	Formula	Options used	States stored	States created	Time [s]	Truth value
Window Lift Error 1	Φ_1	None	184,642,321	241,875,351	21,194.6	valid
		LSE	2,559,671	2,934,246	266.7	valid
		Standard	121,775	145,101	17.64	valid
		All	7,628	314,521	27.05	valid
	Φ_2	None	7,612,190	9,930,188	909.43	invalid
		LSE	212,736	248,695	28.33	invalid
		Standard	13,117	16,219	3.83	invalid
		All	1,151	52,114	4.79	invalid
Window Lift Error 2	Φ_1	None	174,705,586	228,911,900	19,104.62	valid
		LSE	2,336,043	2,655,246	246.7	valid
		Standard	113,663	133,835	16.6	valid
		All	6,784	285,367	25.59	valid
	Φ_2	None	34,894,983	46,138,425	4,025.18	invalid
		LSE	873,766	1,007,382	114.95	invalid
		Standard	40,806	50,762	8.24	invalid
		All	3,220	142,347	12.85	invalid
Window Lift Fixed	Φ_1	None	154,984,097	197,530,241	16,492.16	valid
		LSE	1,706,583	1,953,510	170.81	valid
		Standard	24,003	34,386	5.62	valid
		All	1,063	96,855	9.67	valid
	Φ_2	None	154,984,097	197,530,241	19,107.22	valid
		LSE	1,706,583	1,953,510	213.41	valid
		Standard	24,003	34,386	5.91	valid
		All	1,510	93,200	9.71	valid

Table 10.4: Model checking results for Window Lift, synthetic ATmega16 simulator

Program	Formula	Options used	States stored	States created	Time [s]	Truth value
Window Lift Error 1	Φ_1	None	174,731,303	228,938,135	3,845.33	valid
		LSE	2,342,564	2,589,665	45.33	valid
		Standard	111,032	127,301	9.78	valid
		All	9,110	197,666	10.40	valid
	Φ_2	None	6,873,844	9,063,182	219.32	invalid
		LSE	118,945	129,604	3.86	invalid
		Standard	10,524	12,609	8.46	invalid
		All	961	25,949	7.47	invalid
Window Lift Error 2	Φ_1	None	174,585,948	228,753,466	3,845.62	valid
		LSE	2,286,938	2,510,865	48.92	valid
		Standard	110,049	124,978	8.24	valid
		All	8,247	187,421	8.16	valid
	Φ_2	None	34,832,243	46,014,095	1,089.25	invalid
		LSE	839,278	907,806	21.65	invalid
		Standard	39,412	45,887	9.57	invalid
		All	3,261	84,920	7.92	invalid
Window Lift Fixed	Φ_1	None	154,875,555	197,384,199	3,423.57	valid
		LSE	1,660,560	1,815,293	31.44	valid
		Standard	22,625	29,313	7.82	valid
		All	1,485	61,016	6.82	valid
	Φ_2	None	154,875,555	197,384,199	4,716.66	valid
		LSE	1,660,560	1,815,293	40.64	valid
		Standard	22,625	29,313	7.02	valid
		All	1,485	61,016	6.55	valid

Table 10.5: Model checking results for Window Lift, handcrafted ATmega16 simulator

to undergraduate students. The subject of this lab course is the implementation of a small operating system including resource management and preemptive multitasking. Especially experimenting with the latter was interesting because [mc]square had so far never been used on programs on top of an operating system. One reason for this was that context switching necessarily involves manipulating the stack pointer, and earlier versions of [mc]square relied on this not to happen.

Having supervised multiple runs of the lab course and several of the six experiments involved, we had access to both student solutions and the official ones. Several of the sometimes subtle issues encountered during the implementation motivated us to attempt a formal verification, at least of the core components. Given the complexity of the operating system, however, we realized that this would go beyond the scope of our work. Thus, we decided to at first only provide the simulator, which would be necessary anyway. Actually verifying the system would require more sophisticated abstractions than are available to us at the moment.

Hence, the rest of this section is structured as follows:

- In the next subsection, we illustrate our experience loading and simulating the operating system from the lab course, which is referred to as *HnPOS*.
- In the last subsection, we examine using the ATmega644 simulator on a heavily reduced operating system.

The operating system in the second section is not related to the system from the lab, but is a system we created ourselves. It is stripped down to the mere minimum required: it includes a preemptive scheduler and a means for mutual exclusion in accessing shared resources. The latter is achieved by semaphores [77]. In contrast to HnPOS, it is actually possible to conduct model checking on this system.

10.2.1 HnPOS in [mc]square

For this case study, we used an early version of HnPOS, in which only the basic functionality was implemented. It was part of our own preparation for supervising one of the lab course experiments, and as such, not designed for being model checked.

[mc]square in SVN revision 9288 was used. We compiled the project using Atmels AVR Studio 4 and `avr-gcc` from WinAVR. Loading the generated ELF file was possible with the parameterized ELF loader that is part of the synthetic ATmega644 simulator. [mc]square creates multiple tab views on the GUI, one for each source file. It is possible to step through the program using the simulation panel on the GUI.

Model checking this program using the formula *AG true* discovered an error. At some point during simulation, an invalid instruction is reached, which is located at location 211 (*0xD2*). Inspecting this location using the simulator on the GUI shows that there is actually no instruction at that location, but there is an `STS` instruction at 210. Given that

- there is a variant of the `STS` instruction in the AVR instruction set that is 32 bits long (cf. the AVR Instruction Set Manual [6])
- the bit pattern at addresses 210 and 211 matches that instruction
- the ATmega644 has a program memory that is addressed wordwise, i.e., in units of 16 bit length,

it is therefore certain that address 210 contains a *part* of an instruction, and not a valid instruction itself.

It is important to point out that we cannot state for certain whether this error is actually a problem with HnPOS, or whether this indicates a problem in [MC]SQUARE. It could be related to an implementation error in the synthetic simulator. As it is a real-world program not tailored for model checking purposes, analyzing it manually is a tedious task. We used the formula $EF\,PC = 210$ to generate a trace leading to the relevant program location, but the resulting trace consists of more than 80,000 states. Analyzing this issue is therefore beyond the scope of this thesis.

On a physical ATmega644, an invalid instruction would be ignored. Hence, if the problem were real, it would be impossible to notice it. In that case, however, there would still be a serious flaw inside the program, which causes program execution to continue at undesired locations. For a different jump target, this could easily result in serious consequences such as a system failure.

10.2.2 Simple Operating System in [mc]square

Considering the complexity of HnPOS, we created a very simple operating system ourselves. The system has the following structure

- OS core: the system can be separated into a core part and a user-process part. However, as the AVR microcontrollers provide no memory protection, this is only of architectural relevance.
- Scheduling system: preemptive scheduling is provided. One of the timers of the ATmega644 is used to trigger context switches to the operating system on a regular basis. The scheduler uses configurable time quantums for processes, and is aware of possibly blocked processes.
- Resource management: a means for mutual exclusion is provided, which is based on semaphores.

Due to its simplicity, we decided to call it SIMPLEOS.

Setup

For the verification, we used the same SUN Fire server as in the previous case study. SVN revision 9290 of [MC]SQUARE was used, and SIMPLEOS, which resides in another of our repositories, in SVN revision 394.

Three versions of SIMPLEOS exist:

- `simpleos_only_idle`: SIMPLEOS with no processes except the idle process, which is active whenever no other process requires the CPU
- `simpleos_idle_and_p1`: same as `simpleos_only_idle`, but there is an additional process P1
- `simpleos_idle_and_p1_p2`: same as `simpleos_idle_and_p1`, but there is an additional process P2

Processes P1 and P2 create output on PORTA to show that they are alive. P1 outputs an increasing bit pattern on the lower half of PORTA (i.e., the lower nibble), whereas P2 does the same on the upper half. We tested this functionality on an actual ATmega644P from Atmel [8], which was seated on an Atmel STK500 board [4].

SIMPLEOS directly manipulates the stack, that is, writes to it directly instead of using push and pop operations. As the abstractions had so far never been tested on such programs, we decided to deactivate all of them. A test run in fact indicated that LSE, DND and DVR currently all result in spurious behavior for SIMPLEOS. Probably the stack manipulation violates some assumption involved in either the techniques themselves, or in their current implementation. Due to the effort involved, we did not examine this further.

Verification of Properties

Table 10.6 provides an overview of all the model checking results of this case study.

The first property we were interested in to verify was whether the variable currentProcess has always a well-defined value. As none of the three test programs creates more than three processes, the variable should always be in $\{0, 1, 2\}$. This is encoded by the formula

$$\Phi_1 := AG(currentProcess \geq 0 \wedge currentProcess \leq 2)$$

As expected, Φ_1 is satisfied for all examined versions of SIMPLEOS.

Checking whether a context switch actually occurs was the second property of interest. However, requiring that eventually a switch from process *idle* to process *P1 must* occur is not possible, as [MC]SQUARE would discover a path on which the timer interrupt for the scheduler never occurs. Even though such a path is

Program	Formula	States stored	States created	Time [s]	Truth value
simpleos_only_idle	Φ_1	13,220	13,253	3.00	valid
	Φ_2	13,220	13,253	2.40	invalid
	Φ_3	13,220	13,253	1.86	invalid
	Φ_4	13,220	13,253	1.52	invalid
	Φ_5	13,220	13,253	3.21	invalid
simpleos_idle_and_p1	Φ_1	291,185	291,938	47.67	valid
	Φ_2	1,453	1,453	0.25	valid
	Φ_3	291,185	291,938	38.10	invalid
	Φ_4	291,185	291,938	39.97	valid
	Φ_5	291,185	291,938	49.40	invalid
simpleos_idle_and_p1_p2	Φ_1	3,049,418	3,061,718	418.30	valid
	Φ_2	1,762	1,762	0.25	valid
	Φ_3	2,337	2,337	0.34	valid
	Φ_4	3,049,418	3,061,718	430.70	valid
	Φ_5	3,049,418	3,061,718	446.69	valid

Table 10.6: Model checking results for SimpleOS

infeasible in the physical system, it is possible in the model. Using user-defined environments [70] would allow to exclude the infeasible path from the model, but as of the time of this writing, support for this technique has not been added yet to the synthetic simulators. Therefore, we decided to check for a weaker property, which is *is a context switch possible*. This property can be represented by the following formulas:

$$\Phi_2 := EF\, currentProcess = 1$$

$$\Phi_3 := EF\, currentProcess = 2$$

[mc]square reports Φ_2 to be violated for `simpleos_only_idle`, and to be satisfied for the other two versions, where a process P1 exists. Analogously, Φ_3 is violated for `simpleos_only_idle` and `simpleos_idle_and_p1`, and satisfied for `simpleos_idle_and_p1_p2`. This corresponds to the expected behavior in all cases.

Finally, we wanted to obtain a statement whether a context switch always remains possible. An error in the scheduler or the implementation of the processes, which could for instance deactivate the global interrupt enable flag, could cause the system to get stuck. The corresponding formulas are

$$\Phi_4 := AG(EF\, currentProcess = 0 \wedge EF\, currentProcess = 1)$$

$$\Phi_5 := AG(EF\, currentProcess = 0 \wedge EF\, currentProcess = 1 \wedge EF\, currentProcess = 2)$$

As is visible from Tab. 10.6, the properties are always satisfied whenever the processes requested in the formulas actually exist. Otherwise, the formulas are found to be false.

As a result of this case study, we conclude that the generated ATmega644 simulator can be used to model check programs for this platform. However, in its current state, it can only be used to verify small programs. Especially the growth of the state space when adding another process is an obstacle, as can be seen for the variants of SimpleOS. Therefore, for future work, we consider it important to adapt the abstractions such that they can be used in the presence of an operating system.

10.3 Intel MCS-51 Case Studies

The goal of this case study was to prove that the simulators generated by the Sgdl tool chain can also simulate the behavior of platforms other than Atmel's AVR family. To this end, we show that state space building and model checking is also possible for the Intel MCS-51. Given that we have already demonstrated that the approach is feasible for the AVRs, though, we have decided for a comparatively simple case study.

10.3.1 Setup

We used SDCC (Small Device C Compiler) for compiling the programs checked in this case study. SDCC is a freely available C compiler for a variety of platforms. The case study was conducted using SVN revision 9290 of [mc]square. The programs to be checked reside in the same repository, hence they were also present in revision 9290.

This case study was conducted on a notebook equipped with an Intel Core 2 Duo T7300 at 2.0 GHz, 4 GB RAM, and a 120 GB hard disk. Windows 7 Professional in a 32 bit edition served as the operating system, and an Oracle JDK 1.7 as the Java Runtime Environment.

The program we used for this case study is called `All Interrupts`. As its name indicates, it activates all interrupts of the device, but not necessarily their sources. On the 8051, this results in the following scenario:

- External Interrupt 0: active, may occur
- External Interrupt 1: active, may occur
- Timer 0 Interrupt: active, but may not occur since the timer is not running
- Timer 1 Interrupt: active, but may not occur for the same reason as Timer 0

- Serial Port Interrupt: active, may occur

The above list of interrupts and their event sources are taken from Roth [65], which we also used as a reference during the implementation of the simulator. We also obtained the position of the interrupts in the interrupt table from this source.

The expected behavior for this program is therefore that all interrupts except the two timer interrupts may occur. We also created an additional version of this program called `All Interrupts Active Timers Active`, in which we additionally activate the timers.

10.3.2 Verification

The MCS-51 family has an interrupt table that is located at a fixed position. This fact facilitates checking whether an interrupt occurs, as we only need to check whether the program counter eventually evaluates to one of these positions. Such properties can be encoded by *EF* formulas: *is there a path on which eventually the program counter has a certain value?* The corresponding formulas were the following:

- $\Phi_1 := EF\, PC = 0x03$: checks occurrence of External Interrupt 0
- $\Phi_2 := EF\, PC = 0x0b$: checks occurrence of Timer 0 Overflow
- $\Phi_3 := EF\, PC = 0x13$: checks occurrence of External Interrupt 1
- $\Phi_4 := EF\, PC = 0x1b$: checks occurrence of Timer 1 Overflow
- $\Phi_5 := EF\, PC = 0x23$: checks occurrence of Serial Port Interrupt

As expected, [MC]SQUARE reports that for the program `All Interrupts`, the formulas describing timer overflow interrupts (i.e., Φ_2 and Φ_4) are not satisfied. All other formulas are found to be satisfied. Verification took less than one second for each formula and resulted in less than 1,000 states stored.

Next, we checked the program `All Interrupts Active Timers Active` using the same set of formulas. Φ_1, Φ_3 and Φ_5 are again satisfied, and additionally, Φ_2 and Φ_4 as well.

A conclusion from this case study is that the generated MCS-51 simulator can be used to model check properties for this platform. Hence, the SGDL approach works at least for the AVR family of microcontrollers and for the MCS-51. As a possible direction for future work, a more extensive case study could be conducted to obtain a statement about qualities such as state space size and simulation speed, especially in comparison to an existing handcrafted simulator.

11 Conclusion

In this chapter, we first summarize the results obtained by our research, which are presented in this work. In the second section, we provide an overview of possible directions for future research.

11.1 Conclusion

The topic of this thesis is a new method for coping with the problem that a tool intended for model checking embedded software has to support a wide variety of platforms. The approach is based on the creation of simulators and static analyzers for specific platforms from hardware descriptions. These two components form what we call a state-space generator, i.e., a software module suitable for creating the state space for a model checker.

Chapter 4 outlines the structure of our synthesis system and points out its key features. The system consists of a hardware description language, SGDL, a compiler for processing it, and a runtime library. The SGDL compiler analyzes and translates a hardware description in SGDL to Java code, and the runtime library contains those components that are shared between such generated simulators. If provided with some additional information about a compiled simulator, the SGDL compiler can also automatically integrate it into [MC]SQUARE.

In Chapter 5, we describe the language we chose for our research, SGDL. The language is based on previous work from the AVRora project, parts of which were formerly incorporated into [MC]SQUARE for some time. In AVRora, the language had been called ISILDUR, or shorthand ISDL, and was used to describe the AVR instruction set. Several extensions and modifications of our own distinguish SGDL and ISDL, which is why the two language still strongly resemble each other, while at the same time they are no longer compatible.

Most of the extensions relate to the description of resources, the interrupt system, and language elements required for model checking. The latter include, for instance, the description of atomics to be used in CTL formulas, nondeterminism, and abstraction techniques. Opposed to this, the syntax used in code blocks, e.g. `execute` sections of instructions, remained virtually unchanged.

Chapter 6 focusses on the SGDL compiler, that is, its architecture and mode of operation. The compiler can be decomposed into an analysing part and a code generation part. The analysing part consists of a preprocessor, a parser, several intermediate representations, validity checkers, and a static analyzer, SGDL-STA. By means of the latter, the developer is not required to explicitly provide certain pieces of information about the target microcontroller because the analyzer may already be able to infer it from the given information. Finally, the code generation backend of the compiler is tasked with the creation of Java code. Technically, this is realized by means of code templates, and therefore, by exchanging these, it would be possible for the SGDL compiler to create, for instance, C code instead of Java.

An overview of generated simulators is provided in Chapter 7. This includes their mode of operation, and most importantly, illustrates a means for validating them. Assisting developers in validating their simulators is strictly necessary in order to increase confidence in the correctness of any simulation, which is in turn a precondition for obtaining useful results during model checking. The SGDL compiler assists the developer in this regard by creating a number of test cases for the JUnit testing framework from the given hardware description. These tests are deliberately incomplete, thus forcing the developer to manually describe and check the expected behavior. Thus, deviations between data sheet and actual implementation, which are not accessible to the tool chain, can also be checked.

Abstractions are the topic of Chapter 8. We added several abstractions that were previously developed by other researchers, but also some of our own. The principal result from this work is that even for generated simulators, it is possible to add abstractions, and that some of these can also be created automatically. For some abstractions, like Path Reduction, a fully automatic creation is feasible, whereas for others, like Dead Variable Reduction and Delayed Nondeterminism, some additional information is required. If that information is present, these can also be generated along with the simulator, and no manual modification of the generated code is necessary.

Our research indicates that the aforementioned additional information, which augments the instruction set and resource description, can be at least partly omitted. A static analysis of the hardware description could possibly deduce it, which is a possible direction for future research. We comment on this in the next section.

Chapters 9 and 10 concentrated on case studies. In these case studies, we first implemented three platforms using SGDL: the Atmel ATmega16, the Atmel ATmega644, and the Intel 8051. Then, we examined the behavior of the respective generated simulators on a number of test programs. For the ATmega16 and 8051, we were able to compare the behavior shown by the generated simulator to that of an already existing handcrafted one. This allowed us to evaluate the quality of the simulation with regard to state space size and simulation speed.

11.2 Future Work

One possible direction for future research would be the already mentioned automatic derivation of abstractions such as delayed nondeterminim. The current approach to adding this specific abstraction is based on adding the desired behavior in an additional `execute` section to each instruction. During simulation, the code created from the new section is executed, instead of the code in the regular one. This approach has several disadvantages. First of all, it is redundant code, and reduces maintainability. This might be of importance when fixing errors in a description, or when generalizing it such that it can be reused for further devices. Furthermore, it complicates the combination of several abstractions. In the worst case, this might lead to implementing a separate `execute` section for each possible combination of abstractions. That is, n abstractions would require 2^n `execute` sections, which is why this approach is infeasible.

An automatic derivation of abstractions could be based on a thorough analysis of the SGDL code, for instance by the SGDL static analyzer, SGDL-STA. We illustrated a concept for this in a paper [27]. The actual research in this direction, and to what extent it is feasible and applicable to other abstractions, would be subject to future research.

Another direction for future research could investigate the possibility of replacing SGDL by an established hardware description language. We designed SGDL with the intention of proving that retargeting a hardware-dependent assembly code model checker is possible, while preserving the ability to integrate abstractions. In our opinion, these goals have been met. However, SGDL is only one of several special-purpose languages for describing hardware (cf. the overview in Chapt. 3). As such, the only tools for processing it are the ones we created ourselves, and there is currently no means to connect the toolchain to existing workflows, which are based on widespread languages like SystemC, VHDL, or LISA.

Consequently, future work with regard to the language could attempt to use one of the established languages instead. This would also grant access to already existing processor descriptions, e.g. IP cores. Given that SGDL already exists and provides the necessary language elements, such work could first of all investigate the possibility of a cross-compiler, which translates parts of a hardware description to SGDL. In case that succeeds, a direct translation would also be possible, which could allow for an integration of the approach into existing toolchains.

Bibliography

[1] N. H. M. Aan de Brugh, V. Nguyen, and T. C. Ruys. MoonWalker: verification of .NET programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 of *Lecture Notes in Computer Science*. Springer, 2009. ISBN 978-3-642-00767-5. doi: 10.1007/978-3-642-00768-2_15. URL `http://dx.doi.org/10.1007/978-3-642-00768-2_15`.

[2] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compiler: Prinzipien, Techniken und Werkzeuge*. Pearson Studium, 2nd edition, 2008. ISBN 978-3-8273-7097-6. Deutsche Übersetzung.

[3] M. Aigner. *Diskrete Mathematik*. Vieweg Studium, 4th edition, 2001.

[4] Atmel Corporation. *STK500 User Guide*, March 2003. URL `http://www.atmel.com/Images/doc1925.pdf`.

[5] Atmel Corporation. *Datasheet: ATmega16*, July 2010. URL `http://www.atmel.com/dyn/resources/prod_documents/doc2466.pdf`.

[6] Atmel Corporation. *8-bit AVR Instruction Set*, July 2010. URL `http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf`.

[7] Atmel Corporation. *Datasheet: ATmega644*, July 2010. URL `http://www.atmel.com/dyn/resources/prod_documents/doc2593.pdf`.

[8] Atmel Corporation. *Datasheet: ATmega644P*, February 2013. URL `http://www.atmel.com/Images/Atmel-8011-8-bit-AVR-Microcontroller-ATmega164P-324P-644P_datasheet.pdf`.

[9] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN 978-0-262-02649-9.

[10] S. Biallas, J. Brauer, D. Gückel, and S. Kowalewski. On-the-fly path reduction. *Electronic Notes in Theoretical Computer Science*, 274:3 – 16, 2011. ISSN 1571-0661. doi: 10.1016/j.entcs.2011.07.003. 4th International Workshop on Harnessing Theories for Tool Support in Software (TTSS 2010).

[11] I. Bogosavljevic. Synthesizing an instruction set simulator for model checking embedded systems software. Master's thesis, RWTH Aachen University, 2009.

[12] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. ISSN 0018-9340. doi: 10.1109/TC.1986.1676819.

[13] F. Caron. State partitioning for model checking binary code, 2011. Bachelor Thesis, RWTH Aachen University.

[14] R. Cavada, A. Cimatti, C.A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev. *NuSMV 2.5 User Manual*, 2010. URL `http://nusmv.fbk.eu/NuSMV/userman/v25/nusmv.pdf`. Accessed on June 24, 2013.

[15] K. Chen, S. Malik, and D.I. August. Retargetable static timing analysis for embedded software. In *The 14th International Symposium on System Synthesis, 2001. Proceedings.*, pages 39 – 44, 2001.

[16] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999. ISBN 0-262-03270-8.

[17] E.W. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[18] CoWare. CoWare Processor Designer. URL `http://www.coware.com/products/processordesigner.php`.

[19] P. Forbrig. *Objektorientierte Software-Modellierung mit UML*. Carl Hanser Verlag, 3rd edition, 2007. ISBN 978-3-446-40572-1.

[20] The Eclipse Foundation. Eclipse. URL `http://www.eclipse.org/`. Accessed on May 12, 2013.

[21] M. Fowler. *UML konzentriert*. Pearson Education, 2004. ISBN 3-8273-2126-3.

[22] E. Gamma and K. Beck. JUnit. URL `http://junit.org`. Accessed on January 27, 2014.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, 1st edition edition, 1995. ISBN 978-0-201-63361-0.

[24] *GCC, the GNU Compiler Collection*. GCC Steering Committee / Free Software Foundation (FSF). URL `http://gcc.gnu.org/`.

[25] P. Grun, A. Halambi, A. Khare, V. Ghanesh, N. Dutt, and A. Nicolau. EXPRESSION: An ADL for system level design exploration. Technical report, Department of Information and Computer Science, Unversity of California, Irvine, 9 1998.

[26] D. Gückel. Retargeting a hardware-dependent model checker by using architecture description languages. In *Doctoral Symposium on Systems Software Verification (DS SSV 2009), 4th International Workshop on Systems Software Verification, Aachen, Germany*, 2009.

[27] D. Gückel and S. Kowalewski. Automatic derivation of abstract semantics from instruction set descriptions. In *6th International Workshop on Systems Software Verification (SSV 2011)*, volume 24 of *OpenAccess Series in Informatics (OASIcs)*, pages 71–83, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-36-1. doi: http://dx.doi.org/10.4230/OASIcs.SSV.2011.71.

[28] D. Gückel, J. Brauer, and S. Kowalewski. A system for synthesizing abstraction-enabled simulators for binary code verification. In *Industrial Embedded Systems (SIES 2010), Trento, Italy*, 2010.

[29] D. Gückel, B. Schlich, J. Brauer, and S. Kowalewski. Synthesizing simulators for model checking microcontroller binary code. In *13th IEEE International Symposium on Design & Diagnostics of Electronic Circuits and Systems (DDECS 2010), Vienna, Austria*, 2010.

[30] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *Proceedings of the 34th Design Automation Conference*, 1997.

[31] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Design, Automation and Test in Europe (DATE '99)*, pages 485–490. ACM, 1999.

[32] A. Halambi, P. Grun, and H. Tomiyama. Automatic software toolkit generation for embedded systems-on-chip. In *6th International Conference on VLSI and CAD. ISBN 0-7803-5727-2.*, 1999.

[33] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000.

[34] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson, 2nd international edition, 2003. ISBN 0-321-21029-8.

[35] J. Hromkovic. *Algorithmische Konzepte der Informatik: Berechenbarkeit, Komplexitätstheorie, Algorithmik, Kryptographie; eine Einführung*. Teubner, 2001. ISBN 3-519-00332-5.

[36] V. Kamin. Extending the symbolic representation of states in [mc]square. Master's thesis, RWTH Aachen University, 2008.

[37] K. Keutzer, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19:1523–1543, 2000.

[38] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6 No. 2:323–350, 1977.

[39] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1–2): 134–152, October 1997. ISSN 1433-2779. URL `http://dx.doi.org/10.1007/s100090050010`. 10.1007/s100090050010.

[40] P. Liggesmeyer. *Software-Qualität. Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, 2nd edition, 2009. ISBN 978-3-8274-2056-5.

[41] J. Löll. Application of static analysis in the field of model checking software for embedded systems. Master's thesis, RWTH Aachen University, 2007.

[42] P. Marwedel. *Embedded System Design*. Springer, 2006. ISBN 0-387-29237-3.

[43] J. McAffer, P. VanderLei, and S. Archer. *OSGi and Equinox*. Pearson Education, 2010. ISBN 978-0-321-58571-4.

[44] E. Mercer and M. Jones. Model checking machine code with the GNU debugger. In *Model Checking Software (SPIN 2005)*, volume 3639 of *LNCS*, pages 251–265. Springer, 2005. ISBN 978-3-540-28195-5. doi: 10.1007/11537328.

[45] Sun Microsystems. Using Java Reflection, January 1998. URL `http://java.sun.com/developer/technicalArticles/ALT/Reflection/`. Accessed on 08.08.2012.

[46] M. Might. Abstract interpreters for free. In R. Cousot and M. Martel, editors, *Static Analysis*, volume 6337 of *Lecture Notes in Computer Science*, pages 407–421. Springer, 2011.

[47] M. Moers. Model Checking von Sensornetzwerk-Knoten mit Hilfe von [mc]square. Diploma thesis, RWTH Aachen University, Aachen, Germany, 2008.

[48] D. P. F. Möller. *Rechnerstrukturen.* Springer, 2003. ISBN 3-540-67638-4.

[49] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 1999. ISBN 3-540-65410-0.

[50] T. Noll and B. Schlich. Delayed nondeterminism in model checking embedded systems assembly code. In *Hardware and Software: Verification and Testing (HVC 2007)*, volume 4899 of *LNCS*, pages 185–201. Springer, 2008. ISBN 978-3-540-77964-3. doi: 10.1007/978-3-540-77966-7_16.

[51] W. Oberschelp and G. Vossen. *Rechneraufbau und Rechnerstrukturen.* Oldenbourg, 2000. ISBN 3-486-25340-9.

[52] OpenCores open source hardware IP-cores. URL `http://www.opencores.org/`. Accessed on September 14, 2011.

[53] T. Parr. ANTLR, ANother Tool for Language Recognition, . URL `http://www.antlr.org/`. Accessed on September 30, 2012.

[54] T. Parr. StringTemplate, . URL `http://www.stringtemplate.org/`. Accessed on September 30, 2012.

[55] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA—machine description language for cycle-accurate models of programmable dsp architectures. In *Design Automation Conference (DAC '99)*, pages 933–938. ACM, 1999.

[56] R. Pelanek. On-the-fly state space reductions. Technical report, Masaryk University Brno, Czech Republic, 2005.

[57] JavaCC Project. JavaCC. Accessed on October 30, 2012. URL `http://javacc.java.net/`.

[58] W. Qin, S. Rajagopalan, and S. Malik. A formal concurrency model based architecture description language for synthesis of software development tools. In *Languages, Compilers, and Tools for Embedded Systems (LCTES '04)*, pages 47–56. ACM, 2004.

[59] N. Ramsey and J. W. Davidson. Machine descriptions to build tools for embedded systems. In *Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, volume 1474 of *LNCS*, pages 172—188. Springer, 1998.

[60] J. Regehr and A. Reid. HOIST: A system for automatically deriving static analyzers for embedded systems. *ACM SIGOPS Operating Systems Review*, 38(5):133–143, 2004.

[61] T. Reinbacher. MCS-51 simulator integration into the [mc]square model checker. Technical report, Department of Embedded Systems, University of Applied Sciences Technikum Wien, 2007.

[62] T. Reinbacher. Model checking and static analysis of intel MCS-51 assembly code. Master's thesis, University of Applied Sciences Technikum Wien, 2009.

[63] T. Reinbacher, M. Kramer, M. Horauer, and B. Schlich. Challenges in embedded model checking — a simulator for the [mc]square model checker. In *Industrial Embedded Systems (SIES 2008)*, pages 245–248. IEEE Computer Society Press, 2008. ISBN 978-1-4244-1994-4. doi: 10.1109/SIES.2008.4577709.

[64] T. Reinbacher, D. Gückel, M. Horauer, and S. Kowalewski. Testing microcontroller software simulators. In *Proceedings of the Workshop on SLE for CPS at INFORMATIK 2011 (WS4C11)*, 2011. To appear.

[65] A. Roth. *Das Mikrocontroller-Kochbuch*. IWT Verlag GmbH, 6th edition, unchanged reprint, 1997 edition, 1989. ISBN 3-88322-225-9.

[66] B. Rumpe. *Modellierung mit UML*. Springer, 2004. ISBN 3-540-20904-2.

[67] F. Scheuer. Extending the model checker [mc]square to handle the Infineon XC167 microcontroller. Diploma thesis, RWTH Aachen University, 2007.

[68] B. Schlich. *Model Checking of Software for Microcontrollers*. Dissertation, RWTH Aachen University, Aachen, Germany, June 2008. URL `http://aib.informatik.rwth-aachen.de/2008/2008-14.pdf`.

[69] B. Schlich and S. Kowalewski. An extendable architecture for model checking hardware-specific automotive microcontroller code. In E. Schnieder and G. Tarnai, editors, *Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007)*, pages 202–212, Braunschweig, Germany, 2007. GZVB. ISBN 978-3-937655-09-3.

[70] B. Schlich, D. Gückel, and S. Kowalewski. Modeling the environment of microcontrollers to tackle the state-explosion problem in model checking. In G. Tarnai and E. Schnieder, editors, *Formal Methods for Automation and Safety*

in Railway and Automotive Systems (FORMS/FORMAT 2008), pages 27–34. L'Harmattan, 2008. ISBN 978-963-236-138-3.

[71] B. Schlich, J. Brauer, and S. Kowalewski. Application of static analyses for state space reduction to microcontroller binary code. *Science of Computer Programming: Special Issue on FMICS 2007 & 2008*, 2010. To appear.

[72] M. Schlickling and M. Pister. A framework for static analysis of VHDL code. In *Proceedings of 7th International Workshop on Worst-case Execution Time (WCET) Analysis*, 2007.

[73] J. P. Self and E. G. Mercer. On-the-fly dynamic dead variable analysis. In *Model Checking Software (SPIN 2007)*, volume 4595 of *LNCS*, pages 113–130. Springer, 2007. ISBN 978-3-540-73369-0.

[74] I. Sommerville. *Software Engineering*. Pearson Studium, 8th edition, 2007. ISBN 978-3-8273-7257-4.

[75] S. Steele. Programmer's Notepad. URL `http://www.pnotepad.org`. Accessed on September 23, 2012.

[76] A. Stollenwerk, C. Jongdee, and S. Kowalewski. An undergraduate embedded software laboratory for the masses. In *Proceedings of the 2009 Workshop on Embedded Systems Education*, WESS '09, pages 34–41, New York, NY, USA, 2009. ACM. ISBN 978-1-4503-0021-6. doi: 10.1145/1719010.1719017. URL `http://doi.acm.org/10.1145/1719010.1719017`.

[77] A. S. Tanenbaum. *Moderne Betriebssysteme*. Pearson Studium, 2nd translated edition, 2002. ISBN 3-8273-7019-1.

[78] A. S. Tanenbaum. *Structured Computer Organization*. Pearson Prentice Hall, 5th edition, 2006. ISBN 0-13-148521-0.

[79] W. Thomas, K. Bollue, D. Gückel, G. Quiros, M. Slaats, and M. Ummels. DFG Research Training Group "Algorithmic Synthesis of Reactive and Discrete-Continuous Systems (AlgoSyn)". *it - Information Technology*, 4, 2009.

[80] TIS Committee. Tool interface standard executable and linking format specification version 1.2. `http://x86.ddj.com/ftp/manuals/tools/elf.pdf`, May 1995.

[81] B. Titzer. AVRora. URL `http://compilers.cs.ucla.edu/avrora/`.

[82] B. Titzer, J. Lee, and J. Palsberg. A declarative approach to generating machine code tools. Technical report, UCLA Computer Science Department, University of California, Los Angeles, USA, 2006.

[83] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.

[84] J. Wernerus. Model checking of instruction list programs for programmable logic controllers using [mc]square. Diploma thesis, RWTH Aachen University, 2008.

[85] Z. Yang, B. Al-Rawi, K. Sakallah, X. Huang, S. Smolka, and R. Grosu. Dynamic path reduction for software model checking. In Michael Leuschel and Heike Wehrheim, editors, *Integrated Formal Methods*, volume 5423 of *Lecture Notes in Computer Science*, pages 322–336. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-00254-0. URL `http://dx.doi.org/10.1007/978-3-642-00255-7_22`.

[86] K. Yorav and O. Grumberg. Static analysis for state-space reductions preserving temporal logics. *Formal Methods in System Design*, 25(1):67–96, 2004.

Own Publications

1. Dominique Gückel. Erweiterung des Model-Checkers [MC]SQUARE um benutzerdefinierte Umgebungen. Diplomarbeit, RWTH Aachen, 2007.

2. Bastian Schlich, Dominique Gückel, and Stefan Kowalewski. Modeling the Environment of Microcontrollers to Tackle the State-Explosion Problem in Model Checking. In *Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2008)*, pages 27–34. L'Harmattan, 2008. ISBN 978-963-236-138-3.

3. Dominique Gückel. Retargeting a Hardware-Dependent Model Checker by Using Architecture Description Languages. In *Doctoral Symposium on Systems Software Verification (DS SSV 2009), 4th International Workshop on Systems Software Verification, Aachen, Germany*, 2009.

4. Wolfgang Thomas, Kai Bollue, Dominique Gückel, Gustavo Quiros, Michaela Slaats, and Michael Ummels. DFG Research Training Group "Algorithmic Synthesis of Reactive and Discrete-Continuous Systems (AlgoSyn)". In *it - Information Technology*, Oldenbourg, 2009.

5. Dominique Gückel, Bastian Schlich, Jörg Brauer, and Stefan Kowalewski. Synthesizing Simulators for Model Checking Microcontroller Binary Code. In *13th IEEE International Symposium on Design & Diagnostics of Electronic Circuits and Systems (DDECS 2010)*, pp. 313–316, Vienna, Austria, 2010.

6. Dominique Gückel. Synthesis of Hardware Simulators for Use In Model Checking. In *Dagstuhl 2010. Proceedings of the Joint Workshop of the German Research Training Groups in Computer Science*, p. 76, 2010.

7. Kai Bollue, Dominique Gückel, Ulrich Loup, Jacob Spönemann, and Melanie Winkler (editors). *Dagstuhl 2010. Proceedings of the Joint Workshop of the German Research Training Groups in Computer Science*. Verlagshaus Mainz, 2010.

8. Dominique Gückel, Jörg Brauer, and Stefan Kowalewski. A System for Synthesizing Abstraction-Enabled Simulators for Binary Code Verification. In *Proceedings of the 5th IEEE Symposium on Industrial Embedded Systems (SIES 2010)*, pp. 118–127, Trento, Italy, 2010.

9. Sebastian Biallas, Jörg Brauer, Dominique Gückel and Stefan Kowalewski. On-The-Fly Path Reduction. In *4th International Workshop on Harnessing Theories for Tool Support in Software (TTSS'10)*, Electronic Notes in Theoretical Computer Science. Vol. 274, pp. 3–16, 2011.
DOI 10.1016/j.entcs.2011.07.003, ISSN 1571-0661.

10. Thomas Reinbacher and Dominique Gückel, Martin Horauer, and Stefan Kowalewski. Testing Microcontroller Software Simulators. In *Proceedings of the Workshop on SLE for CPS at INFORMATIK 2011 (WS4C11)*, 2011. To appear.

11. Dominique Gückel and Stefan Kowalewski. Automatic Derivation of Abstract Semantics From Instruction Set Descriptions. In *6th International Workshop on Systems Software Verification (SSV 2011)*, OpenAccess Series in Informatics (OASIcs) Vol. 24, pp. 71–83. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2012. DOI http://dx.doi.org/10.4230/OASIcs.SSV.2011.71, ISBN 978-3-939897-36-1.

List of Abbreviations

ADL	Architecture Description Language
AI	Analysis Information
AST	Abstract Syntax Tree
CFG	Control Flow Graph
CISC	Complex Instruction Set Computer
DND	Delayed Nondeterminism
GCC	GNU Compiler Collection
GPR	General Purpose Register
GUI	Graphical User Interface
HLL	High Level Language
JDK	Java Development Kit
JRE	Java Runtime Environment
JVM	Java Virtual Machine
LHS	Left Hand Side
LOC	Lines of Code
ND	Nondeterminism
NDM	Nondeterminism Mask
OTF-PR	On-the-Fly Path Reduction
PC	Program Counter
PWM	Pulse Width Modulation
RDA	Reaching Definitions Analysis
RHS	Right Hand Side
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level / Register Transfer Language
RVA	Read Variable Analysis
SGDL	State space Generator Description Language
SGDL-STA	SGDL Static Analyzer
SP	Stack Pointer
SSBS	SGDL Static Behavior Section
SVN	Subversion
UDE	User Defined Environment
USART	Universal Synchronous / Asynchronous Receiver and Transmitter
VLIW	Very Long Instruction Word
WVA	Written Variable Analysis

Index

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

http://aib.informatik.rwth-aachen.de/

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

- 2011-01 * Fachgruppe Informatik: Jahresbericht 2011
- 2011-02 Marc Brockschmidt, Carsten Otto, Jürgen Giesl: Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting
- 2011-03 Lars Noschinski, Fabian Emmes, Jürgen Giesl: A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems
- 2011-04 Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, Thomas Noll: A Local Greibach Normal Form for Hyperedge Replacement Grammars
- 2011-06 Johannes Lotz, Klaus Leppkes, and Uwe Naumann: dco/c++ - Derivative Code by Overloading in C++
- 2011-07 Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe: An Operational Semantics for Activity Diagrams using SMV
- 2011-08 Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, Carsten Fuhs: A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog
- 2011-09 Markus Beckers, Johannes Lotz, Viktor Mosenkis, Uwe Naumann (Editors): Fifth SIAM Workshop on Combinatorial Scientific Computing
- 2011-10 Markus Beckers, Viktor Mosenkis, Michael Maier, Uwe Naumann: Adjoint Subgradient Calculation for McCormick Relaxations
- 2011-11 Nils Jansen, Erika Ábrahám, Jens Katelaan, Ralf Wimmer, Joost-Pieter Katoen, Bernd Becker: Hierarchical Counterexamples for Discrete-Time Markov Chains
- 2011-12 Ingo Felscher, Wolfgang Thomas: On Compositional Failure Detection in Structured Transition Systems
- 2011-13 Michael Förster, Uwe Naumann, Jean Utke: Toward Adjoint OpenMP
- 2011-14 Daniel Neider, Roman Rabinovich, Martin Zimmermann: Solving Muller Games via Safety Games
- 2011-16 Niloofar Safiran, Uwe Naumann: Toward Adjoint OpenFOAM
- 2011-17 Carsten Fuhs: SAT Encodings: From Constraint-Based Termination Analysis to Circuit Synthesis
- 2011-18 Kamal Barakat: Introducing Timers to pi-Calculus
- 2011-19 Marc Brockschmidt, Thomas Ströder, Carsten Otto, Jürgen Giesl: Automated Detection of Non-Termination and NullPointerExceptions for Java Bytecode
- 2011-24 Callum Corbett, Uwe Naumann, Alexander Mitsos: Demonstration of a Branch-and-Bound Algorithm for Global Optimization using McCormick Relaxations

2011-25 Callum Corbett, Michael Maier, Markus Beckers, Uwe Naumann, Amin Ghobeity, Alexander Mitsos: Compiler-Generated Subgradient Code for McCormick Relaxations

2011-26 Hongfei Fu: The Complexity of Deciding a Behavioural Pseudometric on Probabilistic Automata

2012-01 Fachgruppe Informatik: Annual Report 2012

2012-02 Thomas Heer: Controlling Development Processes

2012-03 Arne Haber, Jan Oliver Ringert, Bernhard Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems

2012-04 Marcus Gelderie: Strategy Machines and their Complexity

2012-05 Thomas Ströder, Fabian Emmes, Jürgen Giesl, Peter Schneider-Kamp, and Carsten Fuhs: Automated Complexity Analysis for Prolog by Term Rewriting

2012-06 Marc Brockschmidt, Richard Musiol, Carsten Otto, Jürgen Giesl: Automated Termination Proofs for Java Programs with Cyclic Data

2012-07 André Egners, Björn Marschollek, and Ulrike Meyer: Hackers in Your Pocket: A Survey of Smartphone Security Across Platforms

2012-08 Hongfei Fu: Computing Game Metrics on Markov Decision Processes

2012-09 Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhäußer: Quantitative Timed Analysis of Interactive Markov Chains

2012-10 Uwe Naumann and Johannes Lotz: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations

2012-12 Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs: Symbolic Evaluation Graphs and Term Rewriting — A General Methodology for Analyzing Logic Programs

2012-15 Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Solvers for Systems of Nonlinear Equations

2012-16 Georg Neugebauer and Ulrike Meyer: SMC-MuSe: A Framework for Secure Multi-Party Computation on MultiSets

2012-17 Viet Yen Nguyen: Trustworthy Spacecraft Design Using Formal Methods

2013-01 * Fachgruppe Informatik: Annual Report 2013

2013-02 Michael Reke: Modellbasierte Entwicklung automobiler Steuerungssysteme in Klein- und mittelständischen Unternehmen

2013-03 Markus Towara and Uwe Naumann: A Discrete Adjoint Model for OpenFOAM

2013-04 Max Sagebaum, Nicolas R. Gauger, Uwe Naumann, Johannes Lotz, and Klaus Leppkes: Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries

2013-05 Andreas Rausch and Marc Sihling: Software & Systems Engineering Essentials 2013

2013-06 Marc Brockschmidt, Byron Cook, and Carsten Fuhs: Better termination proving through cooperation

2013-07 André Stollenwerk: Ein modellbasiertes Sicherheitskonzept für die extrakorporale Lungenunterstützung

2013-08 Sebastian Junges, Ulrich Loup, Florian Corzilius and Erika Ábrahám: On Gröbner Bases in the Context of Satisfiability-Modulo-Theories Solving over the Real Numbers

2013-10 Joost-Pieter Katoen, Thomas Noll, Thomas Santen, Dirk Seifert, and Hao Wu: Performance Analysis of Computing Servers using Stochastic Petri Nets and Markov Automata

2013-12 Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl: Alternating Runtime and Size Complexity Analysis of Integer Programs

2013-13 Michael Eggert, Roger Häußling, Martin Henze, Lars Hermerschmidt, René Hummen, Daniel Kerpen, Antonio Navarro Pérez, Bernhard Rumpe, Dirk Thißen, and Klaus Wehrle: SensorCloud: Towards the Interdisciplinary Development of a Trustworthy Platform for Globally Interconnected Sensors and Actuators

2013-14 Jörg Brauer: Automatic Abstraction for Bit-Vectors using Decision Procedures

2013-19 Florian Schmidt, David Orlea, and Klaus Wehrle: Support for error tolerance in the Real-Time Transport Protocol

2013-20 Jacob Palczynski: Time-Continuous Behaviour Comparison Based on Abstract Models

2014-01 * Fachgruppe Informatik: Annual Report 2014

2014-02 Daniel Merschen: Integration und Analyse von Artefakten in der modellbasierten Entwicklung eingebetteter Software

2014-03 Uwe Naumann, Klaus Leppkes, and Johannes Lotz: dco/c++ User Guide

2014-04 Namit Chaturvedi: Languages of Infinite Traces and Deterministic Asynchronous Automata

2014-05 Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp: Automated Termination Analysis for Programs with Pointer Arithmetic

2014-06 Esther Horbert, Germán Martín García, Simone Frintrop, and Bastian Leibe: Sequence Level Salient Object Proposals for Generic Object Detection in Video

2014-07 Niloofar Safiran, Johannes Lotz, and Uwe Naumann: Algorithmic Differentiation of Numerical Methods: Second-Order Tangent and Adjoint Solvers for Systems of Parametrized Nonlinear Equations

2014-08 Christina Jansen, Florian Göbe, and Thomas Noll: Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs

2014-09 Thomas Ströder and Terrance Swift (Editors): Proceedings of the International Joint Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments 2014

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.